# The Information Gathering Strategies of API Learners

Ekwa Duala-Ekoko and Martin P. Robillard
School of Computer Science, McGill University
Montréal, Québec, Canada
{ekwa, martin}@cs.mcgill.ca

## ABSTRACT

API users experience significant difficulties when learning how to use APIs, but little is known about the strategies used to overcome these difficulties, the motivation for each strategy, or the trade-offs between the strategies. To better understand the information seeking strategies of API users, we conducted a study in which 20 participants were asked to complete programming tasks using unfamiliar APIs, with the documentation of the APIs and the Web as learning resources. We observed that participants used one of three different strategies when seeking for information on how to use APIs: some were more inclined to using the Web, others preferred the documentation of the APIs, and others combined both the Web and the documentation. We present the characteristics, motivation, and trade-offs between these strategies, and suggests new ideas for documentation and tools to facilitate the information-seeking process of API learners.

## Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

## Keywords

Application Programming Interface (API), Software Libraries, Information Foraging, Empirical Software Engineering

## 1. INTRODUCTION

Modern-day software development is inseparable from the use of Application Programming Interfaces (APIs). Software developers make use of APIs as interfaces to code libraries or frameworks to help speed up the process of software development and to improve the quality of the software. The benefits of using APIs, however, do not come cheap: previous work on API usability showed that learning how to use APIs presents several barriers [8, 12, 15, 17], and that "understanding how the APIs are structured, selecting the appropriate classes and methods, figuring out how to use the selected classes, and coordinating the use of different objects together all pose significant difficulties" [16].

Whereas the difficulties of learning how to use APIs are known, we have yet to fully understand the strategies used by developers to gather the information needed to overcome these learning barriers, the motivation for each strategy, or the trade-offs between the strategies. This need to understand the information gathering strategies of API users has lead to calls for "a more formal, empirically based model of programmers' behaviors that would inform API usage, documentation, and tool design" [17].

In an attempt to answer the call, we conducted an exploratory study in which 20 participants were asked to complete 2 programming tasks using unfamiliar APIs. Two main information sources were used in our study: the documentation of the APIs and the Web. Half of our participants were given access to just the documentation of the APIs, and the other half had access to both the documentation of the APIs and the Web. We investigated the following questions related to the information gathering strategies of API users and the information sources:

- What are the information gathering strategies of API users? Which strategies are the most effective? Which are ineffective?
- What features of the information sources makes learning how to use APIs difficult?
- Were participants who used the Web more effective than those restricted to using just the API documentation?

The intuition amongst API users is that the Web makes learning how to use an API easier and faster since it contains several code examples. Although previous studies have made use of the API documentation [5, 17] or the Web [2, 16] as learning resources, no study, to our knowledge, has investigated this intuition. We investigated this intuition by hypothesizing that the participants with access to the Web would demonstrate an advantage, either in terms of the number of tasks successfully completed or the average time taken to complete a task, over participants restricted to using just the API documentation.

We make several contributions based on the analysis of over 20 hours of screen captured videos with the think-aloud verbalizations of the participants in our study. First, we observed that participants used one of three different information seeking strategies: some were more inclined to using the Web because they believed it makes learning how to use APIs easier and faster, others preferred the documentation because they distrust the code examples found on the Web, and others combined both the Web and the documentation because they believed some knowledge of the classes/methods of an API was essential to understanding code examples on the Web. We further observed that participants with a workflow from the documentation to the Web experienced less difficulty finding and understanding relevant code examples than participants who turned first to the Web when in need of information. Second, contrary to our expectations, we observed no noticeable difference in either the number of tasks successfully completed or the amount of time used to complete each task between the two groups of participants. Finally, we observed that hidden dependencies between related API elements accounted for most of the learning difficulties experienced by our participants, and that the placement of cues is crucial to uncovering these dependencies.

Our paper complements previous studies on API usability by investigating the rationale for the choice of API learning strategies in a detail-rich context, and by studying the eventual outcome of each strategy. Next, we compare our study to a sample of the related work. We present our study design in Section 3, the results of our study in Section 4, and the implications of our results in Section 5. We conclude the paper in Section 6.

## 2. RELATED WORK

Our work builds on previous studies related to API usability, the information needs of programmers, and the working styles of programmers. The literature in this field is abundant and we discuss a sample of the most relevant work.

**API Usability Studies:** In a study comparing the usability of the Factory pattern in contrast to constructors for object creation, Ellis et al. observed that participants experienced more difficulty constructing objects with a factory than with a constructor [5]. Stylos et al. conducted a study in which the usability of parameterless constructors was compared to constructors with parameters, and reported that programmers strongly preferred and were more effective with APIs that provide parameterless constructors [15]. In another study examining the placement of methods (that is, the class to which a method belongs), Stylos et al. reported that participants were significantly faster at identifying relevant dependencies and combining objects when the methods of a starting class referenced its dependencies [17]. Clarke uses the "Cognitive Dimensions" [4], a framework for describing API usability problems, to identify specific usability issues with Microsoft APIs, and to help inform the design of more usable APIs. Other studies have looked at the role of web resources in learning how to use APIs [2, 16]. Prior studies have either focused on the usability of different design choices (e.g., Factory pattern versus constructors) or the usability issues of specific API. These studies have been instrumental at identifying both the difficulties of learning how to use APIs and the needs of API users, but not how the difficulties are overcome. Our study, on the other hand, is concerned with the strategies used by programmers to overcome these API learning barriers, the motivation for each strategy, and the trade-offs between these strategies.

**Information Needs of Programmers:** Several contributions have been made in the area of the information needs of programmers. Ko et al. conducted a study in which novice programmers were asked to complete several tasks using Visual Basic .NET [8], and identified learning barriers and information needs that must be satisfied for the programmers to compete the tasks. In a different study, Ko et al. identified 21 different information needs of programmers in collocated teams [7]. They observed that most of the needs in collocated teams were satisfied by consulting coworkers, and that questions about APIs were answered by consulting the documentation or coworkers. Sillito et al. identified 44 types of questions asked by programmers when maintaining software code, and presented observations of how programmers use tools to support the process of answering the questions [13]. In contrast, our work is concerned with the questions asked by programmers learning to use APIs and the strategies used to answer these questions.

**Empirical Studies of Programming Strategies:** Work in the area of program comprehension has identified different strategies used by programmers to understand programs [14]. Some have argued that programmers use a top-down strategy to understand programs: that is, they work from higher level abstractions to the code [3]; Others hold that programmers use a bottom-up strategy by working from the code to the higher level abstractions [10]. Whether top-down or bottom-up, programmers either work systematically (study a system in detail to gain a global understanding of its structure) or a opportunistically (focus only on areas related to their task). Clarke calls these different work styles personas, and observed that work styles are independent of a developer's level of experience or educational background [4]. We observed comparable work styles in the context of learning how to use APIs, although the goals were different from those of program comprehension. We borrow terminologies from the program comprehension literature to help explain some of our observations.

**Information Foraging:** Pirolli and Card proposed the information foraging theory to help explain how information-seekers search for information [11]. They observed that information-seekers use the same strategies used by predators in the wild when making decisions about where to look for information, what search strategies to use, and which information to consume. Central to the theory are *information scents* (cues which guide information-seekers to relevant information), *information patches* (the information sources), and the *information diet* (the decision of how to select the most profitable patch). Pirolli and Card observed that information-seekers would adapt their search strategies and environment, if need be, to maximize the gains of relevant information per unit cost. In this paper, we make observations consistent with the information foraging theory of how API users use cues in the documentation and the Web to locate relevant classes and methods, and how they adapt their strategies to optimize the gains of relevant information.

## 3. STUDY DESIGN

We conducted a laboratory study in which 20 participants were asked to complete 2 programming tasks using unfamiliar APIs and different information sources.

### 3.1 Participants

Our participants came from the population of non-professional programmers because our work specifically targets non-expert API learners. We recruited participants from the student population of the department of Computer Science at McGill university using on-campus posters and mailing lists, and promised a monetary compensation of $20. Respondents were prescreened using a questionnaire that asked potential candidates about their programming experience and knowledge of Java.

We selected 20 participants from the respondents for our study. The selected participants reported a minimum of 1 year programming experience with Java, 1 year experience working with the Java API documentation (i.e., Javadoc), and some experience programming with Eclipse. Our participants reported between 1 and 6 years of experience programming with Java, with a median of 3.5 years, and an average of 1.5 years of paid programming experience. Five of the 20 participants were female, and our participant pool included 4 PhD students, 11 Masters students, and 5 senior undergraduate students. Although all of our participants were students, they are representative of the population of interest and their expertise level is comparable to that of recent graduates in software development positions.

## 3.2 Tasks

We asked the participants to complete two programming tasks using two real-world Java APIs: JFreeChart and The Java for XML Processing (JAXP). JFreeChart is a popular API for generating charts.[1] We used version 1.0.13 of the JFreeChart API, which has 37 packages and 426 non-exception classes. JAXP is an API for validating and parsing XML documents, developed by Sun Micosystems.[2] We used version 1.4 of the JAXP API, which has 23 packages and 207 non-exception classes.

We selected tasks that involved combining multiple objects since previous work on API usability observed that developers experienced the most difficulty performing such tasks [17]. We reasoned that tasks requiring the combination of multiple objects are more likely to reveal the different strategies and challenges experienced by API users. Participants were given a maximum of 35 minutes to complete each programming task.

**Chart-Task (T1):** We asked the participants to use the JFreeChart API to construct a pie chart with three slices (45% Undergrads, 35% Master's, and 20% PhDs), and to save the chart to a file in a graphic format. The pie chart was titled "Student Distribution at McGill".

**Listing 1: Recommended Solution for Chart-Task**

```
PieDataset dataset = new DefaultPieDataset();
dataset.setValue("PhDs", 20);
dataset.setValue("Undergrads", 45);
dataset.setValue("Masters", 35);
JFreeChart chart = ChartFactory.createPieChart(title,
    dataset, false, false, false);
ChartUtilities.saveChartAsJPEG(new File(fileName),
    chart, 400, 400);
```

**Listing 2: An Improvised Solution for Chart-Task**

```
PieDataset dataset = new DefaultPieDataset();
dataset.setValue("PhDs", 20);
dataset.setValue("Undergrads", 45);
dataset.setValue("Masters", 35);
Plot plot = new PiePlot(dataset);
//improvised code: not part of API
BufferedImage bi = new BufferedImage(400,400,
    BufferedImage.TYPE_INT_RGB);
Graphics2D g2d = bi.createGraphics();
plot.drawOutline(g2d, new Rectangle(400,400));
FileOutputStream fos = new FileOutputStream(new File(
    ""));
EncoderUtil.writeBufferedImage(bi, "jpg", fos);
```

The JFreeChart API provides two possible solutions to the Chart-Task: Listing 1 shows one of the *recommended solution*[3]. We also observed *improvised solutions* (Listing 2) — solutions in which parts of the task were implemented using the given API and other parts were implemented using types from the Java language. The relationship between the information gathering strategies and the resulting solution will be discussed in the results section of the paper.

**XML-Task (T2):** We asked the participants to use the JAXP API to verify whether the structure of an XML file conforms to a given XML schema. The participants were provided with both an XML file and an XML schema file, and were asked to implement the task in a method called *isValid*, which returns true if the XML file conforms to the given XML schema, and false otherwise.

The JAXP API provides two possible solutions to the XML-Task: the recommended solution (Listing 3) and the alternative solution (Listing 4), or its variants. The alternative solution is an outdated way of validating an XML file — the validation is performed as the XML document is being parsed. The recommended solution was introduced in Java 5 and it decouples the validation process from the parsing process.

**Listing 3: Recommended Solution for XML-Task**

```
boolean isValid(String xmlFile, String schemaType,
    String schemaFile){
try{
 SchemaFactory factory = SchemaFactory.newInstance(
     schemaType);
 Schema schema = factory.newSchema(new File(
     schemaFile));
 Validator validator = schema.newValidator();
 Source source = new StreamSource(xmlFile);
 validator.validate(source);
 return true;
 } catch (SAXException ex){ex.printStackTrace();}
   catch (IOException ex){ex.printStackTrace();}

 return false;
}
```

**Listing 4: Alternative Solution for XML-Task**

```
boolean isValid(String xmlFile, String schemaType,
    String schemaFile){
try{
 DocumentBuilderFactory dbf = DocumentBuilderFactory.
     newInstance();
 dbf.setValidating(true);
 dbf.setAttribute("type", schemaType);
 dbf.setAttribute("schemaSource",schemaFile);
 DocumentBuilder parser = dbf.newDocumentBuilder();
 parser.parse(xmlFile);
 return true;
 } catch (SAXException ex){ex.printStackTrace();}
   catch (IOException ex){ex.printStackTrace();}

 return false;
}
```

## 3.3 Study Tools and Instrumentation

Participants completed the study using the Eclipse IDE (version 3.4) and were permitted to use any of the features of the IDE. Two main information sources were used in the study: the documentation of the APIs and the Web, which provides access to example usages of the APIs. These information sources have been reported to be the primary learning resources for API users [16, 18]. We provided the participants with the Firefox browser to access these information sources, and disabled the browser's history feature to prevent any learning effect between participants.

[1]jfree.org/jfreechart/

[2]jaxp.dev.java.net

[3]API providers typically provide sample code on how to use an API, or structure the API documentation to favor a solution they consider to be of superior quality over the others. We call these favored solutions the *recommended solutions* of the API.

We used three data collection techniques in our study: the think-aloud protocol, screen captured videos, and interviews. In the think-aloud protocol [1], participants are asked to verbalize their thought process while solving a given task. Having participants think-aloud was particularly useful in our study as it permitted us to obtain an insight into the participants' understanding of the structure of the APIs and their information sources, their rationale for using a given API learning strategy, and the tactics used to select between alternative usages of API elements or example code. To get the participant comfortable with thinking aloud while working, training was provided using a video tutorial. We used the Camtasia screen capturing software (version 4) to record the contents of the screen and the think-aloud verbalizations of the participants. We also conducted semi-structured post-study interviews in which the participants were asked to comment about the challenges experienced during the programming study. The interviews lasted 5 minutes.

## 3.4 Study Procedure

Our study involved 20 participants, divided into two groups: the *documentation-group* and the *web-group*. The 10 participants of the documentation-group (D1, ..., D10) were restricted to use only the documentation of the APIs; the 10 participants of the web-group (W1,..., W10) were permitted to use both the documentation of the APIs and the Web. To ensure that groups were comparable, we ensured that each group had 5 participants who reported between 1 and 3 years of Java programming experience, and 5 participants who reported above 3 years of Java programming experience.

The programming studies were conducted individually in our research lab, and were supervised by the first author. The participants began each study by watching a 4 minutes video tutorial about the think-aloud protocol. Participants were then given time to practice thinking aloud while working on a web search task. Soon after, the participant was given the instructions for the Chart-Task and was given a maximum of 5 minutes to go over the task requirements and to ask questions relating to the requirements. The participant was then told which information sources may be used for the study depending on whether the participant was from the documentation-group or the web-group. To avoid influencing the strategy of the participants, we did not identify the classes or packages of the APIs required to complete the tasks, as was the case with previous studies (e.g., [17]). Also, the participants were advised to proceed as they would typically do when learning a new API.

Once the participant was satisfied with the task requirement, we loaded an Eclipse project which contained a class with an empty main method and the libraries of the relevant API. We then showed the participant how to use the Firefox browser to access the Javadoc pages of the APIs from the bookmark menu. At this point, the study computer was disconnected from the Internet if the participant was from the documentation-group, Camtasia was started, and the participant was asked to begin.

The screen contents and verbalization data captured by Camtasia during the study were saved once the participant completed the Chart-Task, or once the 35 minutes allocated for the task elapsed. Soon after, the Eclipse environment was once more setup and the participant was asked to begin the XML-Task. The tasks were completed in the same order by all the 20 participants. The study produced a total of 40 different programming sessions and about 20 hours of screen-captured videos and verbalizations of participants working with unfamiliar APIs and different information sources.

**Table 1: An overview of the categories of questions asked by the participants in the programming study.**

| |
|---|
| **Finding a starting point**<br>Which type(s) of the API represents this concept?<br>E.g.: *"I am going to start by searching the API to find something related to pie chart"* [D6, T1][4] |
| **Identifying task-relevant dependencies**<br>Which dependencies of this type are relevant to my task?<br>E.g.: *"are there classes related to BufferedImage that can be used for writing it to a file?"* [D10, T1] |
| **Finding object construction information**<br>How do I construct objects of a given type?<br>E.g.: *"how do I create an instance of PieDataset?"* [W5, T1] |

## 4. RESULTS

Our analysis focused on discovering the strategies used by participants to gather the information necessary to make use of APIs, identifying the trade-offs between the search strategies, and identifying the features of the APIs or the information sources that makes the process difficult. Our method for analyzing the data involved three phases. In the first phase, we went through the screen-captured videos and verbalizations to produce a list of specific questions (such as *"How do I create a Schema object?"*) asked by our participants, and to identify segments of the videos, which we called *episodes*, corresponding to the information-seeking strategy used to answer the questions. Episodes unrelated to the information needs of API users, such as when participants executed their code, were excluded from our analysis. We observed that most participants asked similar questions at similar points in the process of using an API. We refer to these points as the *context* in which the questions were asked, and used these contexts as the foundation for our categorization of the questions and episodes of the participants.

In the second phase, we grouped these questions and episodes based on the *context*, producing three categories (see Table 1). The questions in the first category are about finding a starting point — that is, finding one or more types of the API which represents a concept or requirement to be implemented. The need for a suitable starting point is not unique to API users; Sillito et al. observed that programmers working on maintenance tasks naturally began by looking for a "focus point" [13]. Once an API type relevant to the concept to be implemented is identified, participants must go through its dependencies to identify those relevant to the task. The questions in the second category are about identifying these task-relevant dependencies. The questions in the third category are about finding the information required to construct objects of a given type.

In the final phase of our analysis, we went through the questions, episodes, and verbalizations of the participants for the different categories, noting the differences in strategy between the participants when looking for the information needed to answer these questions, the motivation for each strategy, the trade-offs between the strategies, and the difficulties[5] encountered during the process. We summarize

[4] D6 represents the ID of the participant and T1 represents the task in which the comment/question was observed.

[5] We identified difficulties using verbalizations that indicate an obsta-

**Table 2: The information seeking strategies, characteristics, and rationale of our participants.**

| Strategy | Characteristics | Rationale |
|---|---|---|
| web-inclined | Goes to the Web without knowledge of the classes/methods in an API; works from code examples to documentation. | Code examples on the Web makes learning how to use APIs easier and faster [W2, W4]. |
| doc-inclined | Relies on documentation to select relevant classes/methods, and to determine how to use and coordinate the selected classes; goes to the Web only when faced with barriers. | Distrust code examples found on the Web; the cost of using documentation would eventually pay off, if not immediately [W1, W6]. |
| hybrid | Relies on documentation to select relevant classes/methods, but turns to the Web to determine how to use and coordinate the selected classes. | Having some knowledge of the classes/methods in an API is essential to understanding code examples found on the Web [W8, W10]. |

the characteristics and rationale of the major information seeking strategies in Table 2, and present our observations in the following sections. We derive our observations from three sources of evidence: the think-aloud verbalizations of the participants, their responses to the post-study interview questions, and the information-seeking patterns observed in the screen captured videos.

## 4.1 Finding a Starting Point

The participants were unfamiliar with the APIs used in the study and therefore had little knowledge about the ideal places to begin their search. Naturally, our participants began each task by finding a starting point: *"I am going to start by searching the API documentation to find something related to pie chart"* [D6, T1], and participant W9 began T1 with *"my first instinct is to see if I can find an example that's similar to this"*, then proceeded to the Web. Suitable starting points are required not just at the beginning of tasks, but also whenever participants start working on a new sub-area of a task. For instance, after creating a pie chart in T1, participants had to look for a starting point to search for information on how to save the chart. Participants approached the task of finding a suitable starting point in different ways (see Figure 1): some participants turned first to the *Web*, others turned to the API documentation but started by looking through the *packages* in the API, and others used the API documentation but started by looking through the *classes* in the API. The participants relied heavily on keywords from the task description to identify relevant packages, classes, or example code: *"the task says I should create a pie chart so I'm expecting some sort of a PieChart class to be available"* [W8, T1].

**The Web as a starting point:** Participants who turned first to the Web when in need of information were convinced it makes learning how to use APIs easier and faster: *"when you don't really know about an API its easier to just go to the Internet and look for examples"* [W2]; *"Google usually works best so I'm just going to google this"* [W4, T2]. Participants used keywords from the task description to formulate their search queries. For instance, W8 used the query *"jfreechart piechart tutorial"* for the Chart-Task, with *"jfreechart"* and *"piechart"* taken from the task description. The participants then browsed through the search results from top to bottom and would visit a result page based on the seeming relevance of its summary. Participants were mostly interested in code examples and relied on information scents such as *"demo"*, *"tutorial"*, or *"code"* to identify potentially relevant result pages. Once a page with a code example was identified, participants would skim through the code example to determine its suitability. We observed that participants used multiple attributes besides relevance to determine which code example to use. For instance, some participants used the "age" of the examples code (*"that's an old thread from 2001; I probably want something a bit more recent"* [W8, T1]); and others used facts from multiple code examples *"seems like the last two pages both use ImageIO to save a BufferedImage so I'm going to use that as well"* [W6, T1]. Code examples alone were not enough to determine relevance; participants would regularly visit the documentation to understand the behavior of the classes and methods in code examples: *"I will go back and look at the API documentation a bit more; Its [the code examples is] a bit confusing"* [W4, T2]. Participants copied relevant code snippets from code examples into Eclipse and customized them to the context of their task.

cle or struggle when looking for information on how to use an API element such as *"I can't figure out how to create a Schema object"*.

The Web was used as a starting point in only 4 of the 20 sessions in which the participants had access to the Internet. This came as a surprise; we expected participants to favor the Web as a starting point over API documentation since it is generally believed to contain several code examples. The use of the Web as a starting point did not prove to be the best strategy: 2 of the 4 participants who started with the Web soon abandoned it for the API documentation, and 2 other participants [W2, W7] were unable to complete their tasks even after finding several different example solutions. When asked why he abandoned the Web for the API documentation, W8 commented that *"having some knowledge of the classes in the API may actually be able to help me understand the information provided by the tutorials"*. The response of participant W8 explains why some web participants were unsuccessful at completing their tasks even after seeing example solutions. For instance, participant W2 successfully created the chart for T1 but was unable to save it even after seeing an example solution for saving a *BufferedImage*. Had she looked at the documentation, she would have noticed that a *BufferedImage* could be obtained from a *JFreeChart* object using its *createBufferedImage(int,int)* method. The decision to focus on the Web without assistance from the documentation hindered participant W2 from making the link between the *JFreeChart* object and the example code. Brandt et al. made a similar observation in a study investigating the role of the web in programming: "it is cheaper to search [the Web] for information, but its diverse nature may make it more difficult to understand and evaluate what is found" [2].

A second setback we observed amongst participants who used the Web as a starting point relates to query formulation. Because participants who turned first to the Web relied on keywords from the task description, not the names of the classes/methods of an API, they were inclined to formulating search queries less *specific* to the task than participants with a workflow from the documentation to the Web. Some participants of the Web group [W8, W9] went through several iterations of searching, examining the search results, and formulating new queries before finding relevant code examples. For instance, W8 started the XML-Task with the query *"java xml processing tutorials"* but found no relevant code example. He turned next to the documentation where he identified the *Schema* class as relevant, and commented *"let's go back to the Web and see if I can refine my search"*. He then used a more *specific* query *"java xml validation against schema"* from which he found a relevant code example. On average, participants who began their search for information on the Web reformulated their queries 10 times, whereas participants who started with the documentation before using the Web did not reformulate a single query.

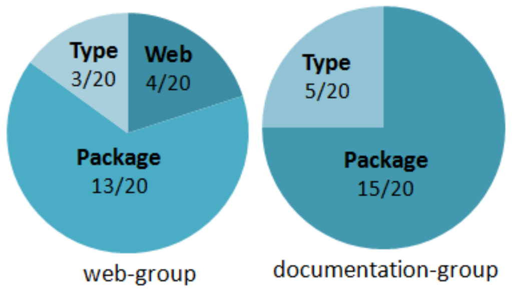

**Figure 1: A distribution of the strategies used by participants to find a starting point.**

**Package as a starting point:** A comparison of the preference for each of the three strategies for finding a starting point against the equal-likelihood multinomial distribution showed statistical evidence that the package was preferred: the package was used in 13 of the 20 sessions with access to the Internet ($p = 0.0078$), and in 15 of the 20 sessions without access to the Internet ($p = 0.0059$). Some participants who started with the API documentation were convinced that compared to the Web, the documentation would provide information relevant not just to the current need, but to other information needs that may emerge in the course of a task: *"the Internet may or may not help; ... it doesn't give you information that may be related to other areas of your task. ... since the documentation is related to your task, there is information that may not be useful right now but may be useful when you move to another task"* [W1]. Other participants expressed distrust of the code examples found on the Web: *"it's hard to find good solutions out there; I trust myself more than I trust a random Google result"* [W6]. Participants began by comparing keywords from the task description to the names and textual summary of the packages to identify potentially relevant packages. The participants would then visit the documentation of a potentially relevant package, browse through its classes, and then select a class whose name and description seemed relevant to the task. Participants would backtrack to the class or package level to select a different class if the current class was deemed irrelevant to the task. Once in the class documentation, participants used the class description and its list of methods to help determine if the class was relevant to the task: *"I found class ChartFactory which has several methods to create several charts; let's see if there is a create pie chart method"* [D10, T1]. Some participants, for instance D3, used the code completion feature of the IDE to examine the methods and documentation of potentially relevant classes. This approach seemed inferior to the others, especially at the beginning stages of finding a suitable starting point, since it requires constructing an object of the class to be examined, a process which may be difficult and time consuming (see Section 4.3), and the resulting code may not form part of the final solution to the task. For instance, D3 commented *"I am still not sure whether this is the right class or not"* after spending 4 minutes experimenting with a class of the JFreeChart API not relevant to the Chart-Task.

Participants browsed through the list of API elements (that is, packages or classes) from top to bottom, but employed different strategies when deciding which elements to visit. Some participants (14/20) first scanned through the list of API elements to identify a set of potentially relevant elements before making a selection: *"these are the packages: parsers, stream, ... and there is a validation package. ... this should be what we need"* [D1, T2]; others participants (6/20) visited every seemingly relevant API element, in the order they appeared, until a relevant package or class was found. This difference in strategy may seem insignificant but influenced the solution provided for a task. For instance, the JAXP API provides two possible solutions for the XML-Task: an older approach that combines parsing with validation (Listing 4), and a newer approach that decouples validation from parsing, introduced in Java 5 (Listing 3). The older approach is supported by the *parsers* package, the newer approach by the *validation* package, and the *parsers* package comes before the *validation* package. Participants who scanned through all the packages before choosing a package selected the *validation* package and benefited from the performance gain (e.g., D1), while the participants with the other strategy selected the *parsers* package (e.g., W6).

Participants from the web-group who turned first to the API documentation when in search for a starting point used the documentation in one of two ways. Some used the *hybrid* strategy: they used the documentation to identify relevant types, but turned to the Web to find code examples on how to use and combine them. For example, W10 started the XML-Task by visiting the JAXP API documentation from which he learned the *Schema* class is relevant. He then used the class name as one of the keywords in the search query to locate code examples: *"we need to find some examples which make use of the Schema class"*, and used the search query *"example code schema class java"* to look for code examples. Other participants used the *doc-inclined* strategy: they used the documentation not only to identify relevant types, but also to look for information on how to combine them; they turned to the Web only when faced with a barrier: *"I get the impression that I am no going to get forward with that [API documentation] so am going to use Google"* [W5, T1]. Participants who began with the documentation, when in need for information, gained valuable knowledge of the classes and methods of the API relevant to a task. This knowledge proved helpful when searching for, and evaluating the relevance of code examples than when the search for information began with the Web.

**Type as a starting point:** The final strategy we observed for finding a starting point was to browse through the list of types provided by the API. Participants who began by browsing through the types expected the API to provide types which correspond to the concepts to be implemented: *"the task says I should create a pie chart so I'm expecting some sort of a PieChart class to be available"* [W8, T1]; *"let's look is there is something straightforward, say PieChart, which may save time"* [D10, T1]. This strategy was used in 3 of the 20 sessions with access to the Web, and in 5 of the 20 sessions without access to the Web. The participants in this group used techniques similar to those in the "Package as a starting point" group to identify classes and methods relevant to their tasks. Although both the "Package as a starting point" strategy and the "Type as a starting point" involved the use of API documentation, we observed trade-offs between them which merit distinction.

We observed that participants who used the "Type as a starting point" strategy experienced more difficulty locating relevant dependencies than participants who used the "Package as a starting point" strategy. For instance, the participants D6 and W1 both started the Chart-Task by browsing through the list of classes in the JFreeChart API to find a starting class. Both selected the *PiePlot* class as a starting point, successfully created a pie chart, but were unable to locate the *ChartUtilities.saveChartAsJPEG(...)* method provided by the API for saving the chart to a file. Participant D6 eventually came up with an *improvised solution* (i.e., a solution of an inferior quality when compared to the solution provided by the JFreeChart API) for saving the chart to a file (Listing 2, lines 7 – 11); participant W1 was unable to complete the task. The difficulty experienced by participants D6 and W1 to locate the *ChartUtilities* dependency could be explained by their search strategy. We observed that participants used the same strategy whether finding a starting point or looking for a relevant dependency. This implies that participants who use the "Type as a starting point" strategy would often ignore the package from which a type comes from, a logical place to begin the search for relevant dependencies. On the other hand, participants who use the "Package as a starting point" strategy would often begin in the package from which a type comes from when looking for relevant dependencies. Had the participants D6 and W1 started with the packages, they could have noticed the *ChartUtilities* class amongst the other chart-related classes in the *chart* package.

## 4.2 Identifying Task-Relevant Dependencies

API classes are seldom used in isolation; often, each class has several other dependencies. However, not every dependency is relevant to a given programming task. Developers must therefore go through the dependencies of a class to identify those dependencies relevant to the task to be implemented. We call such dependencies *task-relevant dependencies.* For instance, to save the pie chart using the JFreeChart API, the participants in our study had to identify the dependency between *JFreechart* (the class provided by the API for holding the states of charts) and *ChartUtilities* (the API class to be used for saving a chart to a file). Developers unable to identify this dependency either came up with an improvised solution or failed to complete the task.

Participants were often unaware of all the relevant dependencies for a given API type. For instance, the participants in our study expected the *JFreeChart* class to provide a method for saving the chart to a file: *"now I go back to the JFreeChart class to check if there is a method to save"* [D10, T1]; *"I am going to look at the methods of JFreeChart; hopefully there is a render, save, or something to that effect"* [W6, T1]. Some participants used the API documentation to browse through the methods of the *JFreeChart* class; others used the code completion feature of the IDE to look for a "save" method on the JFreeChart object. The search for a task-relevant dependency began once our participants realized that the JFreeChart class does not provide a method for saving to file.

The participants exhibited several strategies for finding task-relevant dependencies. Some participants searched for task-relevant dependencies by looking through the list of classes in the API documentation for a class whose name suggests "saving" or "rendering" objects to a file: *"let's go back to all classes [in the API], perhaps there is a ToFileSaver [class] or something"* [W8, T1]. A second strategy was to browse through the list of packages; some participants did not visit every package, but only those which could be related to saving a chart to a file. For instance, 7 of the 20 participant visited the "util", "io", and "renderer" packages looking for utility classes for saving a chart, but found no relevant class because *ChartUtilities* was in the "chart" package. Participants in the web-group eventually went to the Web when unable to locate a class for saving the chart to a file from the API documentation: *"I get the impression that I'm not going to get forward with the Javadoc so I'm going to use Google"* [W5, T1]. Only one participant [W4] used the "Use" page of the API documentation of the *JFreeChart* class to look for a class for saving the chart. This proved to be the most successful strategy for locating task-relevant dependencies since the "Use" page contains all the dependencies of a given class.

Participants from the web-group (WG) spent an average of 5 ($\pm 4$) minutes looking for the dependency *ChartUtilities.-saveChartAsJPEG(...)*, while participants from the documentation-group (DG) spent an average of 7 ($\pm 6$) minutes. Thirteen of the 20 participants in our study experienced some difficulty finding this dependency required for saving the chart, and 3 of the participants were unable to complete the Chart-Task because they could not find the *ChartUtilities* class. This difficulty was due in part to the design of the API (the *ChartUtilities* class is not referenced as either a parameter or a return type in any of the methods of the *JFreeChart* class [17]), the absence of cues in the documentation of the *JFreeChart* class which points to *ChartUtilities*, and the failure by participants to leverage the support provided by the "Use" page.

Six (3 DG; 3 WG) of the 13 participants (6 DG; 7 WG) who successfully completed the Chart-Task came up with improvised solutions of inferior quality compared to the option provided by the API. For instance, one of the improvised solutions involved creating a *BufferedImage*, extracting a *Graphics2D* object from the *BufferedImage*, drawing the chart to the *Graphics2D* object, and saving the *BufferedImage* to a file in JPEG format(Listing 2, lines 7 – 11). This improvised solution is less elegant and more complex than the call to *ChartUtilities.saveChartAsJPEG(...)* provided by the API. We observed that the use of the Web is no guarantee that an API would be used as intended. For instance, 3 of the 7 web-group participants who successfully completed the Chart-Task ended up with an improvised solution by using code snippets found on the Web.

## 4.3 Finding Object Construction Information

One category of questions we observed in our study was about finding the information required to construct objects of classes relevant to a task: *"how do I instantiate a PiePlot?"* [W1, T1]; *"how do I create a Schema?"* [D7, T2]; *"this [Validator] is what I want; how do I make this thing?"* [D4, T2]. This is hardly surprising since the use of classes often requires object construction. Half of our 20 participants began their search for object construction information by attempting to use the default constructor regardless of whether the type was an interface or abstract. For instance, participant D5 commented *"let's see how they[SchemaFactory] are actually created"* after an attempt to instantiate *SchemaFactory*, an abstract class, from the default constructor failed; and both participants D6 and D8 commented *"how can I get an instance of Validator?"* after their attempt to instantiate *Validator*, an abstract class, from the default constructor failed. Our participants seemed to expect classes to provide a default constructor and were surprised when they did not: participant W5 commented *"what is wrong here?"* after an attempt to instantiate *PieDataset*, an interface, from the default constructor failed. This observation corroborates previous findings that programmers not only expect classes to provide default constructors, but also prefer the use of the default constructor over other object construction patterns [5, 15].

Some participants (9/20) continued their search for the object construction information of a class by visiting the constructor section of the API documentation. Although this strategy was helpful for classes with public constructors, it did not prove helpful for classes with other construction patterns. We observed that, if available, the cue on how to create an instance of a class from non-constructor patterns is seldom in the constructor section. For instance, the cue on how to create an instance of *Schema* from *SchemaFactory* is in the description section, not the constructor section of the JAXP API documentation for *Schema*. Thus, participants who went directly to the constructor section in search for information on how to create a *Schema* object missed the cue (e.g., W3), while participants who systematically went through the page from the description section to the constructor section found it (e.g., D7).

Most participants (18/20) turned next to the subclasses when neither the constructor section nor the description section proved helpful: *"this [Schema] is an abstract class, so we need to find a derived class"* [W3, T2]; *"I'm looking for a concrete class [derived class] which I can instantiate to generate a SchemaFactory object"* [D7, T2]. Our participants then browsed through the subclasses looking for a class suitable to the task. Participants often favored relevant subclasses which could be created from public constructors over subclasses with non-constructor construction patterns. However, not every abstract class has derived classes. This is often the case for objects such as *Schema* and *Validator* to be constructed from factories. The lack of derived classes posed significant difficulties for some participants: *"I need to create a Schema object and also ... a Validator object. The problem is these two classes are abstract and I can't find their derived classes."* [W3, T2]; and participants D6 commented *"so where are the derived classes?"* after observing that *Schema* has no derived classes. In the absence of derived classes, some participants resorted to examining the classes of the package in which *Schema* is located; others browsed through the classes of the API. Participants from the web-group supplemented their search for object construction information with web examples when the API documentation ceased to be helpful: participants W3 used Google with the query *"java.xml.validation.Schema"* and was able to find example code on how to create objects of type *Schema* and *Validator*.

Participants from the web-group spent an average of 4 ($\pm$3) minutes looking for information on how to create objects, while participants from the documentation-group spent an average of 6 ($\pm$4). Participants from both groups experienced some difficulty with object construction: 4 from the web-group and 9 from the documentation-group. The participants from the web-group eventually found help from code examples on the Web; 4 of the 9 participants from the documentation-group who experienced difficulty with object construction were unable to complete a task because of this difficulty. Although the "Use" page of each class provides information on how to create it, non of our participants used this strategy.

In post-study interviews, participants attributed the difficultly of using abstract class, interfaces, or classes created through factories to the absence of a "link" between related API elements: for instance, participant D4 commented *"there is no cross-reference in the API documentation that says get a Validator instance from a Schema"* when asked about the difficulty experienced when creating a Validator object [D4, T2]. Participants gave similar explanations for the challenges experienced when looking for task-relevant dependencies. This response only partly explains the problem since the "links" between related API elements are in the "Use" page of the API documentation, a page seldom visited by our participants. Participants expected the "links" between related API elements to be on the element they are exploring or on its API documentation page, and experienced difficulties when their expectation of where the links should be are not meet.

## 4.4 Web-Group Versus Documentation-Group

Some participants echoed the intuition that the Web makes learning how to use APIs easier and faster: *"it is easier to just go to the Internet and look for examples* [W2]; *"it's taking too long so I'm just going to Google it"* [W4]. If this intuition is correct, then the participants of the web-group (WG) should demonstrate some strong advantage, either in terms of the number of tasks successfully completed or the average time taken to complete a task, over participants of the documentation-group (DG). We formulated the following null hypothesis to investigate this intuition:

$H_0$: *The use of the Web has no effect on the number of tasks successfully completed or the time taken to complete a task.*

**Table 3: The number of tasks successfully completed between the two groups.**

| | DG | | WG | |
|---|---|---|---|---|
| | Successful | Unsuccessful | Successful | Unsuccessful |
| **T1** | 6 | 4 | 7 | 3 |
| **T2** | 6 | 4 | 5 | 5 |

We compared the task completion statistics between the two groups to look for evidence which would allow us to reject the null hypothesis. The results are summarized in Tables 3 and 4. Six participants from the DG and 7 participants from the WG successfully completed T1, and 6 participants from the DG and 5 participants from the WG successfully completed T2. We obtained a chi-squared statistic of 0 when we compared the number of tasks successfully completed between the two groups.

Looking at the task completion times, the participants of the DG spent an average of 29 ($\pm$7) minutes on T1 while participants from the WG spent an average of 25 ($\pm$9) minutes. We observed similar results for T2: participants of the DG spent an average of 29 (±8) minutes while participants from the WG spent an average of 26 (±8) minutes. We used the Rank test to compare the task completion time between the two groups and obtained a p-value of 0.45 for T1 and a p-value of 0.26 for T2. Both p-values are significantly greater than the conventional $\alpha = 0.05$. Based on the chi-squared statistics of the number of tasks completed and the p-values of the task completion times, there is no evidence that our null hypothesis is false. Our data suggests that, at least in the context of our study, the use of the Web did not prove to be a significant advantage over the use of the API documentation in learning how to use APIs.

Participants often underestimate the time required to find code examples on the Web, extract the relevant code snippets, and to customize the snippets into the context of a task. Some participants spent a significant amount of time extracting and customizing relevant snippets. For example, participant W3 found a code example for T2 at the 16 minutes mark, but was unable to complete the task in the remaining 19 minutes because of difficulties in extracting and customizing relevant code snippets. When asked about this in the post-study interview, participant W3 commented that *"the example had a different context from our task, so I had to translate their ideas to ours and that takes some time"*. The absence of a significant difference between the two groups suggests that the time required by non-expert API users to find, extract, and customize code snippets from code examples may be comparable to the time needed to learn how to use APIs from the API documentation.

**Table 4: The average task completion time (in minutes) for our 20 participants for both tasks.**

| | DG | | WG | |
|---|---|---|---|---|
| | MEAN | STDEV | MEAN | STDEV |
| **T1** | 29 | ±7 | 25 | ±9 |
| **T2** | 29 | ±8 | 26 | ±8 |

## 5. IMPLICATIONS

### 5.1 API Design and Documentation

We identified several strategies used by participants to explore APIs and their learning resources. Notwithstanding the differences in strategies, our participants shared a common problem: they experienced the most difficulty when cues of how to satisfy information needs were not where participants expected them to be. This difficulty was not caused by the absence of useful information in the documentation, but by the location of the cues within the documentation. For instance, cues on how to create objects of a given type were in other places besides the constructor section of the documentation. A simple restructuring of the documentation to place cues where participants expect them to be could greatly facilitate the API learning process. Our study also revealed that some participants browse the documentation systematically from top-to-bottom when searching for information, while others go directly to the section of interest. Documentation providers should therefore factor in not only the expectations, but also the different search strategies of programmers to provide more useful documentation.

One additional observation made by this study was about the structure of the APIs. Most of our participants expected the class for saving a chart to a file, *ChartUtilities*, to be in either the "util", "io", or "renderer" package of the JFreeChart API, and not in the "chart" package. This was evident by both their visits to these packages and their think-aloud verbalizations. This mismatch between the expected structure of the API and the actual structure of the API lead to several exploration difficulties. Our findings suggest that the placement of classes in packages is not just a matter of internal design, but it also has learnability implications, and that eliciting the expectation of API users during the design of an API could improve its learnability.

### 5.2 Tool Design

Research efforts on tools for learning how to use APIs have predominantly focused on recommending code examples [6, 19, 20]. These code recommenders are based on the premise that programmers already know the classes and methods of an API relevant to their tasks. Our study however paints a different picture: finding the classes and methods of an API relevant to a task remains a significant problem, and that participants are most effective at finding suitable code examples once relevant API elements are known. Jadeite [18], a tool which uses code examples on the Web to highlight the most commonly used classes of an API is the only tool, to our knowledge, aimed at helping programmers find a starting point. Both code recommenders and API learnability could benefit from newer research tools such as Jadeite which assist programmers in finding suitable starting points.

Code recommenders use relevance and popularity to rank recommendations. Our study revealed that these attributes may not be sufficient in all situations because as APIs evolve, popular solutions may no longer reflect the recommended usage of some parts of an API. Other attributes such as the "age" of a code example, used by some participants in our study, combined with relevance and popularity may help point programmers towards the newer and improved parts of an API.

Some participants expressed distrust of code examples found on the Web, and preferred working with the documentation of APIs. We observed that much of the difficulties experienced when using API documentation occurred when the dependencies between related API elements are hidden. For instance, although the *Validator* class and *Schema* are related (a *Validator* object is created from a *Schema* object), this relationship cannot be inferred from the *Validator* class. Participant D4 referred to this as the absence of a *"cross-reference in the API documentation that says get a Validator instance from a Schema"* when commenting about the difficulty experienced when creating a Validator object. Research tools which make the relationship between API elements explicit could improve the API learning process, and facilitate API usability.

### 5.3 Threats to Validity

The contributions of our study are based on a systematic observation of non-professional programmers working with real-world APIs in a laboratory environment. Given this setting, there are factors which limit the generalizability of our observations.

The use of non-professional programmers was intentional; we wanted a homogeneous group of participants to facilitate comparisons with prior work on the learning barriers of API users. The barriers we observed in our study matched those observed in previous studies with both non-professional and professional programmers [5, 8, 15, 17]; we therefore expect the difficulties experienced by the participants in our study to generalize beyond this population. The API learning strategies and the trade-offs between strategies observed in our study could have been a result of the limited experience of our participants. Professional API users may exhibit different learning strategies because of their experience or work context. Other studies on the information needs of programmers working on maintenance tasks have observed comparable exploration strategies [9, 13]. Whether our observations would generalize to the population of professional API users would have to be determined by another study.

We did not look at the relationship between the experience level of our participants and their information-seeking strategies in this paper. Because our study was exploratory, our main aim was to catalogue the strategies of API users and to study their rationale and trade-offs in detail. The connection between experience and strategy is for future work and would likely require a larger sample to produce reliable results.

The size of our tasks, the number of tasks, and the number of participants all limits the generalizability of our observations. Although our tasks represented real usages of real-world APIs, they were limited in size to permit our participants to complete a task within the 35 minutes time frame. With only 2 tasks and 20 participants, the variation of the API learning strategies observed in our study could be limited. However, given the observation that "programmers often approach larger programming tasks by focusing on smaller subtasks" [17], we believe that the strategies and trade-offs we have observed, possibly limited, would generalize to other tasks.

Our study involved only Java APIs and the Java API documentation. Some of our observations may be different if APIs and API documentation in other languages are used. For instance, the flat, alphabetical structure of the Java API documentation could have encourage participants to engage in more browsing than searching. The use of other documentations such as that for the .NET framework with a prominent search interface may encourage more searching than browsing. Furthermore, because our study focused on programmers learning how to use unfamiliar Java APIs, our observations may not be applicable to programmers working with familiar Java APIs. Further studies on API usability are required to verify the generalizability of our observations to these contexts.

## 6. CONCLUSIONS

The goal of our study was to identify the information-seeking strategies of API users, the motivation for each strategy, and the trade-offs between the strategies. We found that participants relied on cues in an API or its documentation to identify relevant elements using one of three strategies: some turned first to the Web when in need for information because they believed it makes learning how to use APIs easier and faster, others preferred the documentation because they distrust the code examples found on the Web, and others combined both the Web and the documentation because they believed some knowledge of the classes/methods of an API was essential to understanding code examples on the Web. We identified trade-offs between these "starting points" , and showed how the choice of a "starting point" could influence both the challenges experienced by programmers and the solution they end up with. Furthermore, we observed that participants experienced difficulty when there is a mismatch between the location of cues to relevant information and the participants' expectation of where cues should be located.

## Acknowledgments

The authors would like to thank all the participants who made this study possible. We are also grateful to our colleagues of the SWEVO research group for their valuable comments on this paper. This work was supported by NSERC.

## 7. REFERENCES

[1] T. Boren and J. Ramey. Thinking aloud: reconciling theory and practice. *IEEE Transactions on Professional Communication*, 43(3):261–278, 2000.

[2] 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 *Proceedings of the 27th international conference on Human factors in computing systems*, pages 1589–1598, 2009.

[3] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[4] S. Clarke. Measuring API usability. *Dr. Dobbs Journal*, pages S6 –S9, 2004.

[5] B. Ellis, J. Stylos, and B. Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, pages 302–312, 2007.

[6] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, pages 117–125, 2005.

[7] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, pages 344–353, 2007.

[8] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 199–206, 2004.

[9] 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. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[10] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[11] P. Pirolli and S. K. Card. Information foraging. *Psychological Review*, 106:643–675, 1999.

[12] M. P. Robillard. What makes APIs hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[13] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[14] M.-A. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 181–191, 2005.

[15] J. Stylos and S. Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th international conference on Software Engineering*, pages 529–539, 2007.

[16] J. Stylos and B. A. Myers. Mica: A web-search tool for finding API components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing*, pages 195–202, 2006.

[17] J. Stylos and B. A. Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 105–112, 2008.

[18] J. Stylos, B. A. Myers, and Z. Yang. Jadeite: Improving API documentation using usage information. In *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, pages 4429–4434, 2009.

[19] S. Thummalapenta and T. Xie. Parseweb: a Programmer Assistant for Reusing Open Source Code on the Web. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, pages 204–213, 2007.

[20] T. Xie and J. Pei. MAPO: Mining API Usages from Open Source Repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 54–57, 2006.