Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study

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

***Abstract*—The increasing size of APIs and the increase in the number of APIs available imply developers must frequently learn how to use unfamiliar APIs. To identify the types of questions developers want answered when working with unfamiliar APIs and to understand the difficulty they may encounter answering those questions, we conducted a study involving twenty programmers working on different programming tasks, using unfamiliar APIs. Based on the screen captured videos and the verbalization of the participants, we identified twenty different types of questions programmers ask when working with unfamiliar APIs, and provide new insights to the cause of the difficulties programmers encounter when answering questions about the use of APIs. The questions we have identified and the difficulties we observed can be used for evaluating tools aimed at improving API learning, and in identifying areas of the API learning process where tool support is missing, or could be improved.**

I. 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. Before leveraging the benefits of an API, a developer must discover and understand the behavior and relationships between the elements of an API relevant to their task. Given the increase in the size of APIs and the increase in the number of APIs developers have to work with, even experienced developers must frequently learn newer parts of familiar APIs, or newer APIs when working on new tasks. Recently, researchers started investigating how design choices common to several APIs affect the API learning process. For instance, Ellis et al. observed that the Factory pattern hinders API learning [1], and a study by Stylos et al. observed that method placement — for instance, placing a "send" method on a convenience class such as `EmailTransport.send(EmailMessage)`, instead of having it on the main-type such as `EmailMessage.send()` — hinders API learning because convenience methods are difficult to discover when learning to use an API [2].

In this paper, we expand on the body of work on API learning by investigating the different types of questions developers ask when working with unfamiliar APIs, investigating why some questions are difficult to answer, and researching the cause of the difficulty. Our study was inspired by the work of Sillito et al., who looked at the different types of questions developers ask when working on maintenance tasks [3]. To investigate those questions about the use of APIs that are difficult to answer, we conducted a study in which twenty participants worked on two programming tasks using different real-world APIs. The study generated over twenty hours of screen captured videos and the verbalization of the participants spanning 40 different programming sessions. Our analysis of the data involved generating generic versions of the questions asked by the participants about the use of the APIs, abstracting each question from the specifics of a given API, and identifying those questions that proved difficult for the participants to answer. Based on the results of our analysis, we isolated twenty different types of questions the programmers asked when learning to use APIs, and identified five of the twenty questions as the most difficult for the programmers to answer in the context of our study. Drawing from varied sources of evidence, such as the verbalizations and the navigation paths of the participants, we explain why they found certain questions hard to answer, and provide new insights to the cause of the difficulties.

The different types of questions we have identified and the difficulties we observed can be used for evaluating tools aimed at improving API learning, and in identifying areas of the API learning process where tool support is missing, or could be improved. As an example, we identified some areas where support is limited from existing tools including the need for tools that would assist a developer in easily identifying types that would serve as a good starting point for searching for code examples, or for exploring the API for a given programming task.

II. Related Work

API Usability Studies: Previous studies on API usability sought to identify factors that hinder the usability of APIs and to understand the trade-offs between design options. Ellis et al. conducted a study to compare the usability of

the Factory pattern in contrast to constructors for object creation [1]: they observed that the participants experienced difficulty and required significantly more time to construct objects with a Factory than with a constructor. Stylos et al. conducted a user study in which the usability of parameterless constructors was compared to constructors with parameters: they reported that programmers strongly preferred, and were more effective with, APIs that provide parameterless constructors [4]. 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 [2]. Clarke uses the "Cognitive Dimensions" [5], [6], 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: in a lab study involving twenty participants and five tasks, Brandt et al. observed that programmers used the Web primarily for just-in-time learning of new skills, and to clarify or remind themselves of previously acquired knowledge [7]. Prior studies have either focused on the usability of different API design choices (e.g., Factory pattern versus constructors), or the usability issues of a specific API, or a learning resource. Our study complements previous efforts by looking at the types of questions developers ask when working with unfamiliar APIs, investigating the cause of the difficulties they encounter answering these questions, and providing suggestions on how tool support for learning how to use APIs could be improved.

Information Needs of Programmers: Several contributions have been made in the area of the information needs of programmers in general. Ko et al. conducted a study in which forty 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. observed and transcribed the activities of seventeen developers working on different tasks during a ninety minutes session [9]. Ko et al. analyzed the transcripts for the type of information that developers sought, the sources they used, and the situations that prevented them from acquiring information. They identified twenty one different information needs of programmers, grouped into seven categories: writing code, submitting a change, triaging bugs, reproducing a failure, understanding execution behavior, reasoning about design, and maintaining awareness. They also observed that the most difficult needs to satisfy were questions about the rationale for design decisions, and that questions about APIs that could not be answered using the documentation or tools, were answered by consulting the coworkers. Sillito et al. identified forty four different types of questions asked by programmers when maintaining software code, and investigated the degree to which existing tools support the questions programmers ask when modifying the source code [3]. The contributions of our study are similar to that of Sillito et al. that looked at the questions programmers asked when maintaining the source code, but ours is in the context of API learning. In addition, we utilized a more objective criterion for determining hard-to-answer questions, provide a catalog of qualitative evidence explaining why developers find certain questions hard to answer, and used more varied sources of evidence (such as navigation paths, verbalizations, and the time spent on various micro tasks) in our analysis than either Sillito et al., or Ko et al.

III. Programming Study

A. Participants

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 about their programming experience and their knowledge of Java and Eclipse.

We selected 20 participants from the respondents for our study. The participants reported a minimum of 1 year programming experience with Java, 1 year experience 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; our participant pool included 4 Ph.D. students, 11 M.Sc. 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, which is our target population since our work aims to support novice programmers.

B. Tasks

We asked each participant to complete two programming tasks: the first task using the JFreeChart[1] API, and the second task using the Java API for XML Processing (JAXP)[2]. JFreeChart is a popular open-source API for generating charts. 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. 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 because previous work on API usability observed that

[1] www.jfree.org/jfreechart/

[2] http://jaxp.java.net

developers experienced the most difficulty performing such tasks [2]. We reasoned that tasks requiring the combination of multiple objects are more likely to reveal a variety of questions developer want answered and typical challenges they encounter when learning to use APIs. The participants were given a maximum of 35 minutes per programming task. All the 20 participants were unfamiliar with the APIs used in the study.

Chart-Task.
We asked the participants to use the JFreeChart API to construct a pie chart with three slices (45% Undergrads, 35% Master's, and 20% Ph.D.s), and to save the chart to a file in a graphic format. To complete this task, a participant needed to construct and configure at least five API types (`JFreeChart`, `PiePlot`, `PieDataset`, `ChartFactory`, and `ChartUtilities`), and had to discover key relationships between the types: for instance, the relationship between `JFreeChart`, the type for representing charts, and `ChartUtilities`, the type needed to save the chart.

XML-Task.
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 a solution that returns true if the XML file conforms to the given XML schema, and false otherwise. This task required the combination of at least four API types (`Schema`, `Validator`, `SchemaFacotry`, and `Source`) and was selected because of the unique challenges it presents to object construction — all the required types are abstract with no subtypes; the types must be created from factory or public methods.

C. Study Setting

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 [10], [11]. 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.

The programming studies were conducted individually in our research lab. The participants began each study by watching a four-minute 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. 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 [2]. 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 screen and voice recording software — Camtasia, version 4 — was started, and the participant was asked to begin the task. The participant was asked to move to the next task upon completion of the Chart-Task, or once the 35 minutes allocated for the task elapsed. The tasks were completed in the same order by all the 20 participants.

D. Data Collection

We used three data collection techniques in our study: the think-aloud protocol, screen captured videos, and interviews. In the think-aloud protocol [12], 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, to identify the types of questions participants ask when learning to use APIs, and to understand why a participant may have difficulty answering a given question. 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.

The screen contents, the verbalizations of the participants, and the interview sessions were captured using Camtasia. 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 real-world APIs.

IV. Data Analysis and Results

Our analysis focused on the questions the participants want answered about the use of an API. Our goal was to identify those questions that are difficult to answer, to understand why these questions proved difficult to answer, and to recommend programming tools that could facilitate the API learning process. Our method for analyzing the data involved three phases: identifying the different types of questions asked by the participants, categorizing the questions, and coding the exploration patterns used by the participants when searching for answers to these questions. We refer to a participant by their ID (for instance, P5 for the fifth participant) and to the tasks as T1 (for the Chart Task) and T2 (for the XML Task).

Table I
DIFFERENT TYPES OF QUESTIONS OBSERVED DURING THE PROGRAMMING STUDY

Generic questions, with specific examples in italics	# of occurrences	# of participants
Q.1 Which packages or namespaces of an API provide types relevant to my task? *"I'm trying to find out which package has classes for creating a pie chart"* — P5,T1	31	16
Q.2 Is there an API type that provides a given functionality? *"the task says I should create a pie chart; I'm expecting some sort of a PieChart class to be available"* — P18,T1	11	7
Q.3 Does an API type provide a method for performing a given operation? *"Is there a method on* `BufferedImage` *that helps to save?"* — P10,T1	58	19
Q.4 What is the functionality of a given API type? *"Let's look at what the* `Validator` *class does"* — P18,T2	32	17
Q.5 Can a method intended to perform operation A be used to perform operation B? *"I'm hoping that the draw method can be used to save to a file, but I'm not too optimistic about it"* — P6,T1	3	2
Q.6 Which keywords best describe a functionality provided by the API? *"I'm going to use the Firefox search to look if there's any thing involving[containing the word] "schema"s"* — P9,T2	38	13
Q.7 How is the type X related to the type Y? *"How is* `Validator` *related to* `Schema`*?"* — P18, T2	8	7
Q.8 How do I get an object of type X from the type Y? *"I need to figure out how to get a* `BufferedImage` *from a* `PiePlot`*"* — P6,T1	2	1
Q.9 Which elements of the API are of the type X? *"Which classes of the API are* `Comparable`*?"* — P11, T1	4	4
Q.10 Is the object X of the type Y? *"let's see if* `RenderedImage` *takes* `BufferedImage`*"* — P1,T1	2	2
Q.11 Does the API provide a helper-type for manipulating objects of a given type? *"let's see if there are classes related to* `BufferedImage` *which can give me the possibility to write the image to a file"* — P10,T1	19	13
Q.12 How do I create an object of a given type without a public constructor? *"the constructor is protected; so how do I create a* `Graphics2D` *object?"* — P11,T1	57	19
Q.13 Which other API elements are necessary to use a given API type? *"I think I need something else that would save the chart to an image"* — P1,T1	6	5
Q.14 Which subtype of an interface or class is the most appropriate for my task? *"I don't know exactly which subtype of* `Source` *to use for reading an XML file"* — P6,T2	29	17
Q.15 Which types of a given domain (package, namespace) are relevant to my task? *"Which classes of the "parsers" package could be used for validation?"* — P18, T2	27	17
Q.16 Which method from a list of overloaded methods is relevant to my task? *"I'm trying to find the appropriate create-piechart method because it seems to be overloaded"* — P16,T1	4	4
Q.17 What role do the arguments of a given method play in its usage? *"we have a* `newInstance(String)` *method that takes a* `String` *argument and I have no idea what this* `String` *is suppose to be"* — P9,T2	23	17
Q.18 What is the valid range of values for a primitive argument, such as an integer, of a given method? *"I don't know if this [double] value should be between 0 and 1"* — P10,T1	3	3
Q.19 Is NULL a valid value for a non-primitive argument of a given method? *"let's use NULL for* `Comparable` *and see if the method throws an exception"* — P1,T1	4	4
Q.20 How do I determine the outcome of a method call? *"[the method]* `Validator.validate(Source)` *returns void; how do I know the results of the validation?"* — P12,T2	15	13

A. Identification of Questions

In this phase, we went through the screen-captured videos and the verbalizations to produce a list of *specific* questions asked by each participants, and to identify segments of the videos, which we called *episodes*, corresponding to each question. Each episode also captured the specific approach used by a participant to answer a given question. Some questions were explicit: for instance, participant P11 asked *"How do I create a Graphics2D object?"* while working on the Chart-tasks. Other questions were easily inferred from the actions and verbalizations of the participants: for instance, P1 came across the method `ImageIO.write(RenderedImage, ...)`, and said "let's see if `RenderedImage` takes `BufferedImage`", then went ahead and used a `BufferedImage` object where `RenderedImage` was expected. The actions and verbalization of P1 in this example is phrased into the question: *"Is* `BufferedImage` *of the type* `RenderedImage`*?"*. After identifying the list of specific questions for each participant, we then developed *generic* versions of the questions that slightly abstract from the specifics of a given API. For instance, the question *"Is* `BufferedImage` *of the type* `RenderedImage`*?"* can be stated more generally as *"Is the object X of the type Y?"*. Based on these generic questions, we identified twenty different types of questions asked by the participants across both tasks (see Table I). We also provide a specific instance for each generic question as an example, in italics. The generic questions highlight, to a certain extent, the type and scope of the information developers need when learning how to use APIs. The number of times each type of question was observed (# of occurrences) and the number of participants that asked each type of question (# of participants) are listed in Table I.

B. Abstraction of Developer Behavior

We needed a high-level abstraction of the actions of the participants to facilitate the analysis of their behavior and

the challenges they experienced when learning to use APIs. Since our analysis is centered around the questions asked about the use of the APIs, we transcribed the segments of the videos corresponding the time frame during which a participant asked and searched for answers to a given question. Specifically, for each participant and for each episode corresponding to a specific question, we transcribed the video into a series of actions that summarizes the steps taken by the participant to answer the question. We considered the following actions:

- **Browse:** the participant looked through a list of *API elements* (packages, types, or methods) either within Eclipse, or in the documentation, before making a selection. The Browse action has a *target* to denote the items (either packages, classes, methods, or search result) the participant was browsing through.
- **Select:** the participant selected an item from a list of API elements, or the results of a search query after browsing. The Select action has a target — the name of the item selected, and a flag (Yes/No) to indicate if the selected element was relevant to the question being answered. The target of the Select action could also be *None*, if no item was selected.
- **Read:** the participant focused on a portion of text or code. The Read action has a target — the name of the element being read, and a section (e.g., either the introduction section, constructor section, or the method description of an API element) to indicate the location the participant focused on.
- **Navigate:** the participant followed a dependency or a link to another element. The Navigate action has a *target* — the name of the item navigated to, a flag (Yes/No) to indicate if the *target* led to information relevant to answering the question.
- **Search:** the participant performed a search of the documentation or the Web. The Search actions has a *target* (Documentation/Web), and a flag (Yes/No) to indicate if the search query contained the name of an API element.
- **Switch:** the participant moved from the documentation to the Web, or IDE, and vice versa.
- **Use:** the participant attempted to use an API element or code example found on the Web. This action has a *target* — the element or code the participant attempted to use, and a flag (Yes/No) to indicate if the participant was successful.
- **Backtrack:** the participant stepped back to a previous location of certainty, then decides to explore a different path. This action has a *target* — the location the participant backtracked to.

As an example, Table II shows a partial transcript[3] of the participant P15 looking for information on how to save a `BufferedImage`; the transcribed actions is shown under the "Action Sequence" column. P15 started by navigating to the documentation page of `BufferedImage`, browsed through its subtypes, and then selected the subtype `WritableImage`, not relevant to saving an image. P15 read the introduction section of `WritableImage`, then backtracked to `BufferedImage`. P15 then browsed through the methods of `BufferedImage`, focusing on the `createGraphics` method, before switching to the Web. P15 then searched the Web with a query containing an API element, selected the third results, read through the code example and discovered the `ImageIO.write(RenderedImage, ...)` method. P15 then used `ImageIO.write(RenderedImage, ...)` successful to save the image to a file.

Table II
TRANSCRIPT EXCERPT FOR PARTICIPANT P15 — CHART TASK

Time	Question	Action Sequence
0:18:10	How do I save a BufferedImage?	**Nav**[BufferedImage]: **Browse**[subtypes]: **Select**[WritableImage,No]: **Read**[WritableImage,Intro]: **Backtrack**[BufferedImage]: **Browse**[methods]: **Read**[createGraphics,description]: **Switch**[web]:**Search**[web,Yes]: **Select**[3rd,Yes]: **Read**[ImageIO.write,code]: **Use**[ImageIO.write,Yes]

What is a Difficulty?

As part of our analysis, we intended to identify questions that proved difficult for the participants to answer and to understand the cause of the difficulty. To accomplish this, we needed an objective measure as to what constitutes a difficulty in the context of API learning. We decided not to use the amount of time taken to answer a question as the main measure of difficulty since significant performance variations have been observed amongst developers with similar level of experience [13]. At a higher-level, we observed that some of the actions, or sequence of actions, of a participant that reflected a lack of progress in the search for answers to a given question would serve as a good measure for capturing difficulty. We used the following action sequences as a definition of the difficulty participants encountered when answering questions about the use of the APIs:

- **Use[*target*, No]**: This action sequence captures instances in which a participant attempted to use an API element but was unsuccessful because the API does not support the given usage. For instance, the participants P6 and P8 commented *"How can I get an*

[3]The entire transcripts for both tasks, and for all the participants are available as an online appendix: http://www.cs.mcgill.ca/~eduala/apistudy/

Table III
A SUMMARY OF THE DIFFICULTIES PARTICIPANTS EXPERIENCED ANSWERING DIFFERENT TYPES OF QUESTIONS ABOUT THE USE OF APIS.

Question ID	#times	#instances	#participants	#difficulty
Q.1	31	1	16	1
Q.2	11	5	7	3
Q.3	58	14	19	13
Q.4	32	3	17	3
Q.5	3	1	2	1
Q.6	**38**	**17**	**13**	**7**
Q.7	**8**	**5**	**7**	**5**
Q.8	2	1	1	1
Q.9	4	2	4	2
Q.10	2	0	2	0
Q.11	**19**	**17**	**13**	**12**
Q.12	**57**	**35**	**19**	**17**
Q.13	6	1	5	1
Q.14	29	8	17	8
Q.15	27	5	17	4
Q.16	4	0	4	0
Q.17	23	1	17	1
Q.18	3	2	3	2
Q.19	4	1	4	1
Q.20	**15**	**11**	**13**	**11**

instance of Validator?" after their attempt to instantiate *Validator*, an abstract class, from the default constructor failed. This object instantiation difficulty is captured by the action sequence *Use[Validator.Constructor, No]*. We observed that participants often had expectations about the design of an API and expressed frustration when the structure of an API did not match their expectations.

- **Browse[*list*]**, **Select[*target*, No]**, ..., then **Backtrack[*list*]**, or **Navigate[*target*, No]**, ..., then **Backtrack[...]**: The **Select[*target*, No]** and **Navigate[*target*, No]** action sequences capture instances in which a participant went down an irrelevant information search path. Once a participant realized the information on a given path was not relevant to answering their question, they backtracked to previous location of certainty, and then chose a different path to explore: captured by the **Backtrack** action. We consider action sequence **Select[*target*, No]**, or **Navigate[*target*, No]**, followed by a **Backtrack** as an indication of difficulty in searching for answers to a given question. The participants relied on cues in the documentation or code examples when looking for answer to a given question. At times, the clues were not available or perceivable. In the absence of strong cues, participants were left to guess which search paths to follow, and some participants inadvertently went down irrelevant search paths.

We summarize the difficulties the participants experienced answering the different types of questions in Table III. For each question, we provide the number of times the question was observed (#times), the number of instances with a difficulty (#instances), the number of participants who posed the question (#participants), and the number of participants who experienced a difficulty answering the question (#difficulty). As a baseline, we considered a question difficult to answer if all of the following conditions apply:

- At least half of the participants who posed a question experienced some difficulty answering the question.
- At least five participants experienced some difficulty answering the question.
- A difficulty was observed in about half the total number of the instances in which a question was asked.

For instance, we considered the participants to have experienced difficulty answering question Q.20 since eleven of the thirteen participants who posed the question experienced difficulty answering it, and since a difficulty was observed in eleven of the fifteen instances in which the question was asked. We identified five questions that proved difficult for the participants to answer (boldfaced in Table III):

Q.6 Which keywords best describe a functionality provided by the API?
Q.7 How is the type X related to the type Y?
Q.11 Does the API provide a helper-type for manipulating objects of a given type?
Q.12 How do I create an object of a given type without a public constructor?
Q.20 How do I determine the outcome of a method call?

C. Observations

We present our findings as observations of the challenges a developer may encounter when learning to use an API, along with the supporting evidence for each observation. These observations are supported by the results of our analysis of the data from the study, and by the verbalizations of the participants.

Observation 1 (Discovering Relevant Dependencies). *Discovering relevant API types not accessible from the type a developer is working with is a major challenge to API learners.*

Three questions (Q.7, Q.11, and Q.12) of the five we identified as being difficult to answer involved a participant either looking for types related to, and relevant to the use of a type they were working with (*"let's see if there are classes related to* `BufferedImage` *which can give me the possibility to write the image to a file"* — P10, T1), or a participant seeking to discover the relationship between API types (*"How is* `Validator` *related to* `Schema`*?"* — P18, T2). Although different, these three questions illustrate a common problem: the participants experienced significant difficulty when relevant API types were not accessible from the type they were working with (i.e., these helper-types were not referenced or reachable from any of the public members of the type the

Table IV
A COMPARISON OF THE SEARCH QUERIES WITH, AND WITHOUT, AN API ELEMENT.

Search Queries With an API Element	
Total Queries:	25
Reformulated Queries:	4
Successful Queries:	21

Search Queries Without an API Element	
Total Queries:	13
Reformulated Queries:	12
Successful Queries:	1

developer was working with). For instance, in the Chart Task, the participants could save the `JFreeChart` object using `ChartUtilities.saveChart(JFreeChart,...)`, but most experienced difficulty locating `ChartUtilities` because it is not accessible from `JFreeChart`. Twelve of the 20 participants in our study experienced some difficulty finding `ChartUtilities` (Q.11, Table III), and three of the participants were unable to complete the Chart-Task because they could not locate this relevant dependency. This observation corroborates the findings of Stylos et al. [2] that method placement — the class on which a method is placed — affects API usability. However, *Observation 1* extends beyond method placement: the participants also had difficulty discovering the relationships between types (Q.7, Table III), or creating objects for types without a public constructor (Q.12, Table III) because the relevant helper-types were not accessible from the type they were working with. Participant P4 attributed this difficulty to the lack of a *"cross-reference in the API that says get a* `Validator` *instance from a* `Schema`*"*.

Observation 2 (Query Formulation). *In the context of our study, having an API element as one of the keywords in a search query was an effective strategy for locating relevant code examples.*

We analyzed the queries the participants formulated when searching for code examples on the Web and observed that queries that contain an API element were more *successful* (that is, led to a relevant code example) than those without an API element (see Table IV). There were a total of 25 queries containing an API element, and of those, only four were reformulated, and 21 of the 25 queries containing an API element led to a relevant code example. On the other hand, there were a total of 13 queries without an API element: 12 of the 13 queries were reformulated, and only one of the 13 queries led to a relevant code example. As an example, participant P18 started the XML-Task with the search query *"java xml processing tutorials"* but found no relevant code example. He then turned to the documentation where he identified the `Schema` class as relevant to the validation task. Participant P18 then reformulated the search query to *"java xml validation against schema"* from which he found a relevant code example. Our observation about query formulation corroborates the finding from the analysis of the Koders' search engine logs where queries with code elements lead to the most downloads [14].

A complementary observation about query formulation is the difficulty of guessing keywords that correspond to word usage in APIs, or their documentation. This difficulty was captured by the question Q.6 (*Which keywords best describe a functionality provided by the API?*): up to seven of the thirteen participants who asked this question experienced some difficulty guessing a correct keyword.

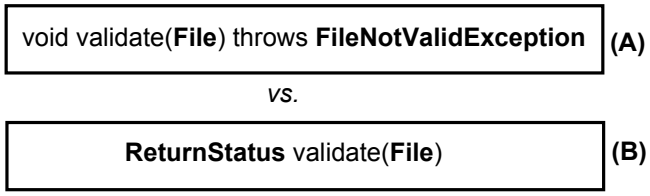

Figure 1. Two possibilities for communicating method-level execution failures.

Observation 3 (Exceptions). *The use of an exception to communicate the outcome of a method execution hinders API comprehension.*

There is a long standing debate in the software development community regarding whether an *exception* (as in Figure 1 (A)), or a *return-status-object*[4] (as in Figure 1 (B)) should be used to communicate method-level execution failures, and when each design choice may be appropriate [15]–[17]. When an exception is used to communicate the failure of the `validate(File)` method, the implication is that the `File` is considered valid if the method does not throw an exception. In other words, the exception is used to communicate the return status (*failure* or *success*) of the `validate(File)` method: the validation is said to have failed if the method `validate(File)` throws an exception, and successful otherwise. However, the extent to which this implication is apparent to a developer learning to use an API remains uncertain. In the XML tasks, the participants had to use the method `Validator.validate(Source)`, that used an exception to communicate outcome, to validate an XML file. We observed that the use of an exception to communicate outcome was problematic to the participants: 11 of the 20 participants experienced significant difficulty realizing the implication that if the method `validate(Source)` does not throw an exception, then the `Source` file is considered valid (Q.20, Table III). The 20 participants spent an average of 4.2 minutes each before becoming aware of the implication that the XML file is considered valid if

[4]We used the word *return-status-object* to represent either an error code (a primitive such as a boolean or an integer), or an object that contains the return status of a method execution.

the validation method does not throw an exception; the average time spent to make this discovery increases to 6.7 minutes if we consider just the 11 participants that faced a difficulty. Participants P5, P14, and P20 were unable to make the discovery within the alloted time for the task, even after spending 14 minutes, 9 minutes, and 21 minutes, respectively, on this part of the task.

We looked at the verbalizations of the participants and the post-study interviews in an attempt to understand why they could have missed the implication that the XML file is considered valid if the validation method does not throw an exception. We identified two possible reasons for this difficulty.

The Expectation of the Participants. The participants expected the API to provide a validation method with a return-status-object (such as in Figure 1 (B)), but `validate(Source)` had no return-status-object: "`validate(Source)` *does not give us something like true or false; I better look for a method that gives us a boolean*" — P1. Not finding the expected method (that is, a validate method with a return-status-object), participants would initially assume that `validate(Source)` is not the right method to use, instead of making the connection that an exception is used to communicate the success or failure of the validation. They would then spend time looking for other methods in the API with a return-status-object that could be used to validate the XML file. Not finding an alternative, the participants would then return to the `validate(Source)` method, re-read its documentation, and realize that the XML file is considered valid if the validation method does not throw an exception.

Disagreement between API Designers and Participants as to what Constitute an "*error condition*". The second reason expressed by the participants as to why they experienced difficulty associating exceptions to the success, or failure, of the validation relates a disagreement as to what constitute an error condition. According to expert API designers: "*if a member [method] cannot successfully do what it was designed to do — what the member name implies — it should be considered an execution failure, and an exception should be thrown*" [15, p. 218]. In other words, an execution failure is said to have occurred if the `validate(Source)` method cannot validate the XML file, and an exception should therefore thrown. Our participants, on the other hand, seem to associate the throwing of an exception to something catastrophic:

"*if you're just trying to validate something why would it throw an exception; It doesn't make sense; I expected it [*`validate(Source)`*] to return an object*" — P14.

"*in my experience ... I find consensus that you throw exceptions only when you find an error. In a Validator you expect some thing to be valid or invalid. And if its invalid that should be a common occurrence just as much as it is valid. So throwing an exception for common occurrence is not a good idea*" — P11.

This disagreement between API designers and our participants reflects the debate as to when exceptions should be used. Some argue that exceptions are for "exceptional conditions"; others argue that "*exceptions should be used to report all errors*" [15, p. 212]. The software development community has yet to agree on what constitutes an exceptional condition. Our study indicates that this disagreement has the potential for influencing API comprehension.

Observation 4 (Web versus Documentation). *The use of the Web had no effect on the number of tasks successfully completed or the time taken to complete a task.*

In designing the study, we had ten of the participants use the Web and the API documentation as learning resources (the Web Group — WG), and the other ten using just the documentation (the Documentation Group — DG). We reasoned that partitioning the participants into two groups would help us identify the challenges programmers faced when looking for answer to the questions using both learning resources. We expected the participants in the Web Group to be significantly more successful since the Web provides several code examples for both tasks. However, we observed no significant advantage, either in terms of the number of tasks successfully completed or the average time taken to complete a task, between the participants of the Web Group over the participants of the Documentation Group. Six participants from the Documentation Group and seven participants from the Web Group successfully completed task T1, and six participants from the Documentation Group and five participants from the Web Group successfully completed task 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 Documentation Group spent an average of 29 ($\pm$7) minutes on task T1 while participants from the Web Group spent an average of 25 ($\pm$9) minutes. We observed similar results for task T2: participants of the Documentation Group spent an average of 29 ($\pm$8) minutes while participants from the Web Group spent an average of 26 ($\pm$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 task T1 and a p-value of 0.26 for task T2. But why were the participants who used the Web not significantly better than those who used the API documentation?

We observed that some 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 P13 found a code example for task

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 P13 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"*. Other participants started with the Web but soon realized the difficulty of finding relevant code examples with no knowledge of the types in the API. For instance, P18 started with the Web but soon abandoned the Web for the API documentation after two unsuccessful searches, commenting *"having some knowledge of the classes in the API may actually be able to help me understand the information provided by the tutorials"*. In general, we observed that both learning resources provide complementary support to programmers learning to use APIs. Also, the absence of a significant difference between the two groups suggests that the time required by novice 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 for basic tasks such as the ones in our study.

V. Implications

A. API Design and Documentation

Proponents of the debate on how to communicate method-level failures typically endorse either the use of an exception, or the use of a return-status-object, but seldom both: "*exceptions should be used to report all errors for all code constructs*" — [15, p. 212]. Our results explain and document why the use of an exception to communicate the outcome of an operation may be problematic from an API-comprehension perspective. In such situations, it seem reasonable for API designers to consider providing both a return-status-object (to provide status information in the case of a successful operation) and an exception (to communicate method execution failure). Steven Clarke of the user experience group at Microsoft Research, and a pioneer of the work on API usability, echoed this view in a book on Framework Design Guidelines: "*although return codes should not be used to indicate failures, you can still consider returning status information in the case of a successful operation*" [15, p. 213].

In general, our study underscores the need to investigate the impact of API design choices on API learning and usability before adopting a given design choice. APIs are provided to improve programmers' productivity, but poorly designed, or poorly documented, APIs may produce a counter effect. In our work with APIs, we have observed situations where programmers had to re-invent the wheel because APIs designed for their task were difficult to understand [18]. API usability studies provide a venue for identifying and fixing usability and comprehension problems before an API is made public.

B. Tool Design

The primary motivation for our study was to understand the nature of API learning and how best to support programmers learning to use a new API. We have identified 20 different types of questions the programmers asked about the use of APIs and also five questions that proved the most difficult for the programmers to answer. We believe these questions could help evaluate existing tool support, and identify areas where support is lacking. As an example, we present three areas of difficulty where support is currently limited.

Discovering Relevant API Elements not Accessible from the Type a Programmer is Working With. Jadeite [11] uses a concept known as a "placeholder" to allow a developer to annotate the documentation of an API type with other API types or methods not accessible, but relevant to its use. Given a particular function, Altair [19] and FRIAR [20] use heuristics and structural relationships to find other related functions. Jadeite is the only tool, to our knowledge, aimed at helping programmers discover types or methods not accessible from a main-type. We consider Jadeite a precursor to an ideal tool for making relevant API elements not accessible from a type discoverable. Our ideal tool would automatically generate and recommend placeholders and would be integrated with the IDE, preferable with the content assist feature of the IDE.

Discovering the Types of an API Relevant to Implementing a Task. The names of API types and methods provide a common vocabulary between API users and API designers; consequently, the use of types and methods for query formulation proved to be an effective strategy (in the context of our study) for locating code examples relevant to implementing a task. Surprisingly, support for helping programmers discover the types of an API relevant to a task is limited. In fact, most code recommendation tools are based on the premise that programmers already know the types of an API relevant to their tasks [21]–[24], but this is not often the case. Sourcerer helps programmers locate relevant API elements by suggesting words from open source systems that share concepts that are related to the terms in a search query [25]. Jadeite leverages usage statistics from code examples on the Web to display commonly used types of an API more prominently. Jadeite and Sourcerer have one drawback: they are unusable in the absence of a corpus of code examples; consequently, APIs without a corpus of examples, or less commonly used parts of an API, may not be supported. There is a need to further explore complementary approaches (such as the relationships between API elements used by Prospector [26]) for recommending API types relevant to a task.

Unmasking the Relationships between API Types. Some of the difficulties we observed occurred when the dependen-

cies between related API elements were not obvious, or not properly documented. 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 P4 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 relating these types. One potential solution would be to explicitly document such hidden dependencies. Alternatively, tools could be developed to automatically identify and reveal such hidden relationships between API elements to developers through the content assist feature of IDEs.

C. Threats to Validity

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

The types of questions we observed in the study, the process of answering the questions, and the challenges the participants experienced are related to a certain extent to the tasks and the experience of the participants. Some of the questions and the difficulties we observed in the study have been observed in previous API usability studies in different settings [1], [2], [4], [8]; However, given that our study was exploratory in nature and intended to probe *why* developers experience difficulties, and also given the lab setting and predefined tasks, the catalog of questions cannot be considered complete, but a starting point.

The difficulty the participants experienced in associating the throwing of an exception to the success, or failure, of the validation could have been a result of their limited programming experience. This threat was mitigated by our use of the think-aloud protocol which showed that our participants had no apparent confusion with the validation domain (they could implement a solution to validate the XML file). Rather, the difficulty they encountered was well isolated to the use of exceptions to communicate method-level failures: the implication that an operation is considered successful if the method does not throw an exception was not apparent to our participants. Furthermore, Steven Clarke is quoted as reporting similar observations amongst professional programmers in a book on Framework Design Guidelines: "*In one API usability study we performed, developers had to call an* `Insert` *method to insert ... records into a database. If the method did not throw an exception, the implication was that the records had been inserted successfully. However, this was't clear to the participants in the study. They expected the method to return the number of records that were successfully inserted*" [15, p. 212]. The results of our study closely corroborate Clarke's observation amongst professional programmers; the extent and reasons for the difficulty for the population of professional programmers would have to be determined by another study.

The size of our tasks, the number of tasks, and the number of participants also 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 two tasks and 20 participants, the questions and the challenges observed in our study could be limited. However, our use of 20 participants is equal or above the current standard of evidence in user studies of software engineering tools [7]. Furthermore, given the observation that "programmers often approach larger programming tasks by focusing on smaller subtasks" [2], we believe that the different types of questions and the challenges we observed, possibly limited, would generalize to other API learning tasks.

Lastly, our study involved only Java APIs and the Java API documentation. Some of our observations may be different for APIs and documentation in other languages. Also, since 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.

VI. Conclusion

To understand the difficulties programmers encounter when working with unfamiliar APIs, the cause of the difficulties, and to investigate how best to support API learning, we conducted a study in which 20 programmers worked on programming tasks using two real-world APIs. The study generated over 20 hours of screen captured videos and the verbalization of the participants spanning 40 different programming sessions. Our analysis of the data involved generating generic versions of the questions asked by the participants about the use of the APIs, identifying those questions the proved difficult to answer, and investigating the cause of the difficulty using the verbalization and the actions of the participants. Based on the results of our analysis, we identified 20 different types of questions programmers ask when learning to use APIs. We also identified five of the 20 questions as being the most difficult for the programmers to answer, and provide observations to explain the potential causes of the difficulties. We believe the questions we have identified and the difficulties we observed can be used for evaluating tools aimed at improving API learning, and to identify areas of the API learning process where tool support is lacking, or could be improved. As an example, we identified some areas where tool support is currently limited including the need for tools that would assist a programmer easily identify the types of an API that would serve as a good starting point for searching for code examples, or a starting point for exploring the API for a given programming task.

REFERENCES

[1] 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*, 2007, pp. 302–312.

[2] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *Proc. of the 16th International Symposium on Foundations of Software Eng.*, 2008, pp. 105–112.

[3] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[4] J. Stylos and S. Clarke, "Usability implications of requiring parameters in objects' constructors," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 529–539.

[5] S. Clarke, "Measuring API usability," *Dr. Dobbs Journal*, pp. S6 –S9, 2004.

[6] ——, "Evaluating a new programming language," in *Proceedings of the 13th Workshop of the Psychology of Programming Interest Group*, 2001, pp. 275–289.

[7] 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*, 2009, pp. 1589–1598.

[8] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *Proc. of Visual Languages and Human Centric Computing*, 2004, pp. 199–206.

[9] 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*, 2007, pp. 344–353.

[10] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding API components and examples," in *Proc. of the Visual Languages and Human-Centric Computing*, 2006, pp. 195–202.

[11] J. Stylos, B. A. Myers, and Z. Yang, "Jadeite: improving API documentation using usage information," in *Extended abstracts on Human factors in computing systems*, 2009, pp. 4429–4434.

[12] T. Boren and J. Ramey, "Thinking aloud: reconciling theory and practice," *IEEE Transactions on Professional Communication*, vol. 43, no. 3, pp. 261–278, 2000.

[13] B. Curtis, "Substantiating programmer variability," in *IEEE*, ser. 7, vol. 69, 1981, pp. 846–846.

[14] S. Bajracharya and C. Lopes, "Analyzing and mining a code search engine usage log," *Empirical Software Engineering*, pp. 1–43, 2010.

[15] K. Cwalina and B. Abrams, *Framework design guidelines: conventions, idioms, and patterns for reusable .Net libraries*, 2nd ed. Addison-Wesley Professional, 2009.

[16] D. Katz, "Error codes or exceptions? Why is reliable software so hard?" April 2006. [Online]. Available: http://damienkatz.net/2006/04/error_code_vs_e.html

[17] J. Spolsky, "Exceptions," October 2003. [Online]. Available: http://www.joelonsoftware.com/items/2003/10/13.html

[18] E. Duala-Ekoko and M. P. Robillard, "The information gathering strategies of API learners," TR-2010.6, School of Computer Science, McGill University, Tech. Rep., 2010.

[19] F. Long, X. Wang, and Y. Cai, "Api hyperlinking via structural overlap," 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*, ser. ESEC/FSE '09, 2009, pp. 203–212.

[20] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird, "Recommending random walks," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07, 2007, pp. 15–24.

[21] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Software*, vol. 25, pp. 45–52, 2008.

[22] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th International conference on Software Engineering*, 2005, pp. 117–125.

[23] 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*, 2007, pp. 204–213.

[24] T. Xie and J. Pei, "MAPO: mining API usages from open source repositories," in *Proceedings of the workshop on Mining software repositories*, 2006, pp. 54–57.

[25] S. Bajracharya, J. Ossher, and C. Lopes, "Searching API usage examples in code repositories with sourcerer API search," in *Proceedings of 2010 ICSE Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation*, 2010, pp. 5–8.

[26] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *Proceedings of the International conference on Programming language design and implementation*, 2005, pp. 48–61.