An Empirical Study of API Usability

Marco Piccioni, Carlo A. Furia, and Bertrand Meyer
ETH Zurich, Switzerland
Email: {marco.piccioni, carlo.furia, bertrand.meyer}@inf.ethz.ch

***Abstract*—Modern software development extensively involves reusing library components accessed through their Application Programming Interfaces (APIs). Usability is therefore a fundamental goal of API design, but rigorous empirical studies of API usability are still relatively uncommon. In this paper, we present the design of an API usability study which combines interview questions based on the cognitive dimensions framework, with systematic observations of programmer behavior while solving programming tasks based on "tokens". We also discuss the implementation of the study to assess the usability of a persistence library API (offering functionalities such as storing objects into relational databases). The study involved 25 programmers (including students, researchers, and professionals), and provided additional evidence to some critical features evidenced by related studies, such as the difficulty of finding good names for API features and of discovering relations between API types. It also discovered new issues relevant to API design, such as the impact of flexibility, and confirmed the crucial importance of accurate documentation for usability.**

I. Introduction

It is not by chance that software has become so complex, but by building over the solid underpinnings of abstraction and modularity. Modern software is a complex composition of library components, whose elementary functionalities are combined to achieve domain-specific applicative goals. A component's *interface* consists of features that clients can call; the combination of interface features and a description of their usage protocol and semantics is what is normally called API: Application Programming Interface. Using library components solely based on their APIs makes it possible to abstract away implementation details and practice modularity; therefore, learning and using APIs are everyday tasks for software developers. From a research perspective, it is interesting to understand what makes APIs easy or hard to use; this is the overall goal of this paper.

APIs are a fundamental interface for the interactions between programmers and computers [1]. Thus, API usability impacts software development quality [2], [3]: usable APIs are more intuitive, require less documentation browsing, and encourage reuse, thus increasing developers' productivity. Conversely, APIs that are hard to use reduce programmer productivity and quality of the final product, as shown, for example, by measuring requests for technical support [3].

Previous empirical studies of API usability—reviewed in Section VIII—point to some critical issues such as the ease of discovering relationships between types offered by an API and how to instantiate new objects. With our work, we find additional evidence to confirm or dispute these issues, as well as to find out new criticalities not detected in previous work.

To this end, the first contribution of our paper is the design of an empirical study of API usability. The study's research questions—described in Section II—are based on the findings of previous studies with the goal of corroborating and extending them. Section III describes the specific study design, which combines two methodologies useful for assessing usability. The first is the notion of *cognitive dimensions*: elements that characterize the expectations of users and what an API actually provides, such as the abstraction level and the consistency between different API functionalities [4]. In our study, we collected and interpreted the feedback of participants according to the cognitive dimensions. The second methodology used in our study design is based on *usability tokens*, an original classification of the participant reactions (such as "surprise" or "incorrect choice") as they try to understand and use API functionalities to perform the assigned tasks. We tallied usability tokens when replaying the videos of the participants' performances. As we discuss in Section VI, combining cognitive dimensions and usability tokens leads to a rich characterization of API usability issues.

The second paper contribution is an actual execution of the study to evaluate the usability of a persistence library written in Eiffel. The study—which we detail in Section IV—involved 25 participants, including both students and professional developers. The participants solved five tasks requiring to access, modify, and query a relational database through the services offered by the persistence library API; we recorded their performance to be able to analyze it postmortem. After each session, we asked the participants about their experience and expectations using a structured interview whose questions were characterized by the cognitive dimensions.

Section V analyzes the study results based on the analysis of performance and interviews; and Section VI discusses the findings in a more general context, which include:

- Finding descriptive, non-ambiguous names for API features is problematic given that programmers may be used to different terminologies.
- Discovering relations between API types (classes) requires significant effort; simple designs are beneficial, especially to less experienced programmers.
- Accurate and complete documentation is a crucial issue for API usability; all the major usability flaws discovered in our study trace back to unsatisfactory documentation.[1]
- Flexibility is a double-edged sword in API design: experienced programmers can take advantage of it, but it may confuse those with less practice.

Finally, Section VII mentions possible threats to the validity of the study; and Section IX concludes.

[1] As documentation, we mainly consider: public method signatures, comments, and contracts of the class features.

II. Research Questions

We organize the study around four main research questions, which characterize fundamental aspects of usability: understandability, abstraction level, reusability, and learnability. The questions cover the critical aspects emerged in previous API usability studies (see Section VIII) but are sufficiently general to make room for new findings and specific aspects emerged in our study (see Sections V and VI).

- **RQ1** What is the effort required to understand the semantics of API features based on their names and documentation?
- **RQ2** Does the API's abstraction level cater to usability?
- **RQ3** Does the API's design facilitate reuse and conciseness in client code?
- **RQ4** Can API usage be learned easily and incrementally?

RQ1 addresses the fundamental problems tackled by programmers using the services offered by an API: what they have to do to understand what each feature of the API represents and how it should be invoked. This question also encompasses specific issues such as whether the API feature names are descriptive, whether the relationships between API elements are clear and unambiguous, and how to access the features through object creation, method call, or other means.

RQ2 evaluates whether the level of abstraction is good for usability. On the one hand, it should guarantee that programmers can proficiently use the API without knowing (or assuming) implementation details. On the other hand, the abstractions should match the conventions and practices of programmers, without being elegantly abstract at the expense of understandability and other practical concerns. Summarizing with a slogan, this research question asks whether the API "makes simple things simple, and complex things possible" [5].

RQ3 determines to what extent the client code that can be written using the API is concise, terse, maintainable, and extensible. Concretely, this question addresses problems such as: if we have an application using some API features and slightly vary or generalize its requirements, how hard is it to adapt the application to meet the extended requirements?

RQ4 targets the API learning process to see if it can be incremental (that is, gradual and not requiring disproportionate efforts) and whether performing a certain programming task using the API has a positive impact on performing other, related but different, tasks. This question is related to RQ1 and RQ3 but emphasizes the learning process rather than its practical outcomes.

III. Study Design

Usability is a multifaceted feature which is hard to assess reliably because it involves somewhat subjective aspects such as the specific habits and styles of individual programmers. To address such problems, the cognitive dimensions framework [4] suggests to evaluate usability operationally based on comparisons. First, we outline a number of aspects that are sufficient to characterize the multiple facets of usability in our specific domain. Then, we base the assessment on the *comparison*, for each aspect, between the expectations of users and what the system actually provides. For example, regarding incremental learnability in APIs, users may expect to be able to execute incomplete code to obtain feedback on API behavior, whereas the API might require specific complete call sequences before the code is executable without errors; this would signal an imperfection in the API's usability.

Our empirical study follows the guidelines of the cognitive dimensions framework, and more specifically Clarke's dimensions of API usability [6]. It is based on the execution of programming tasks that require the participants to write client code combining API features to achieve certain functionalities; the comparison between expectations and reality is based on the participants' performance on the programming tasks. With the goal of having a multidimensional assessment, which helps reduce the impact of subjective perceptions, we collect data about the discrepancies between user expectations and actual system features from two different sources.

After completing the tasks, we conducted a structured interview with the participants involving a fixed list of questions about their experience and expectations. This explicit feedback provided by programmers through the interview highlights their perceived usability, the difficulties they experienced, and the features that explicitly appreciated. Section III-A describes the interview questions in some detail and discusses how they articulate the research questions of Section II.

Independent of the interview data, we also collected feedback given implicitly by the participants during their performance. To this end, we asked them to follow the thinking-aloud protocol and recorded the video and audio of their performances. The thinking-aloud protocol is based on Ericsson and Simon's seminal work [7] and consists in having the participants declare in speech their mental process as it develops, including the doubts and questions they have, the solution strategies they consider, and the reasons that justify their decisions. This protocol is widely used in usability testing [8], because it makes the external observer aware of the cognitive processes leading to a certain performance. In our study, we perused postmortem the recordings of the performances searching for episodes revealing the expectations of programmers and how they compared to the API features. As we describe in Section III-B, we classify the episodes in *usability tokens*, which express mismatch between user expectations and actual performance. The tokens make the implicit data collection more uniform and aligned to the issues targeted by the research questions.

A. Structured Interview

Following Clarke's guidelines [6], and reflecting the research questions of Section II, the interview consists of twelve questions which characterize various aspects of usability as assessed by users. The questions are as follows, roughly grouped by the research question they target (although some of them target multiple research questions).

Questions regarding RQ1 (understandability):

1) Do you find that the API types map to the domain concepts in the way you expected?
2) Do you feel you had to keep track of information not represented by the API to solve the tasks?

3) Does the code required to solve the tasks match your expectations?

Questions regarding RQ2 (abstraction):

4) Do you find the API abstraction level appropriate to the tasks?
5) Did you need to adapt the API (inheriting from API classes, overriding default behaviors, providing non-API types) to meet your needs?
6) Do you feel you had to understand the underlying implementation to be able to use the API?

Questions regarding RQ3 (reusability):[2]

7) Does the amount of code required for each task seem about right, too much, or too little for you?
8) How easy was it to evaluate your own progress (intermediate results) while solving the tasks?
9) Do you feel you had to choose one way (out of many) to solve a task in the scenario?
10) Do you feel you would have to change much in your code to access another kind of persistence store, or write another query?

Questions regarding RQ4 (learnability):

11) Once you performed the first two tasks, was it easier to perform the remaining tasks?
12) Do you feel you had to learn many classes and dependencies to solve the tasks?

We use the cognitive dimensions framework to formulate questions that may discover the existence of problematic issues with the API, rather than aiming at "proving" its usability. This angle—reminiscent of the way testing cannot not prove programs correct but discover the existence of errors—is aligned to the best practices of using the cognitive dimensions framework [9], [10].

B. Usability Tokens

We classify the salient events occurring during participant performances into five tokens. While the tokens describe different dimensions, the same event may be associated to multiple tokens when it reveals aspects of different nature. For each token, we make simple examples based on a hypothetical data-structure library to make the description independent of the specific application domain used in our study (namely, persistence).

Token "surprise": the developer discovers aspects of the API that clearly go against her expectations and original intuitions. For example, the generic container classes may not be usable with primitive types (e.g., integers) but only with object types; this requires a special treatment for certain types, which the user may find surprising.

Token "choice": the developer is faced with a choice and she must understand the alternatives to proceed in the right way. For example, the element at a given position in a list may be accessible either with a method call *s.at(i)* or using the array notation $s[i]$; the user may have to understand if the two options are equivalent and choose the most appropriate one (in terms of correctness and code readability).

Token "missed": the developer's activity shows that she has missed some important abstraction or feature of the API, which would have been useful to effectively solve the current task. For example, the API may offer a feature to remove the first occurrence of a given element in the list, but the developer may miss it and implement the same functionality by first searching and then deleting.

Token "incorrect": the developer uses the API incorrectly, in a way that introduces errors or implements an incorrect functionality. For example, popping elements from a stack without checking for emptiness may lead to errors when the stack is empty.

Token "unexpected": the developer uses the API in a way undocumented or otherwise unforeseen by the API intended design. Unlike the "surprise" token, this token adopts the point of view of API designers. For example, the user may inherit from a stack class and override the push method to allow insertion in the middle of the stack; this is not necessarily wrong, but violates the original design abstraction.

The usability tokens are largely orthogonal to the cognitive dimensions (on which the interview questions are based), in that the same token may characterize events involving different cognitive dimensions. For example, an element of surprise may reveal inconsistent abstractions, poor understandability, or both. This helps make the data from the interviews largely independent of the usability token classification.

IV. Study Setup

We implemented the study design discussed in Section III to assess the API usability of a recently developed persistence library for Eiffel. In the following subsections, we give a few details about: the library and its API; the programming tasks involving the API which we submitted to the participants; the demographics of the participants; and the protocol used to carry out the experiments.

A. Persistence Library API

We evaluated the API of ABEL (A Better EiffelStore Library), a library offering an object-oriented interface to persistence-related functionalities like storing and retrieving data using files (serialization) and relational databases. The first author developed ABEL as part of his PhD work [11]. ABEL is written in Eiffel and is meant as an improvement over the persistence services offered by Eiffel's standard libraries. As it introduces features normally available in other mature persistence frameworks such as Spring [12] and Hibernate [13], it should be usable by "standard" programmers, and hence it is interesting to assess its usability.

ABEL consists of 81 classes grouped into 11 clusters (roughly equivalent to Java packages) for a total of roughly 10500 lines of code, comments, and assertions. In Eiffel, it is customary to use assertions in the form of contracts (pre- and postconditions, and class invariants) to specify the essential requirements and behavior of methods. Therefore, when browsing the API of ABEL, developers can see the signature,

[2]Question 10 is specific to the domain of the API we evaluated, but it can be easily generalized to different application domains.

comments, and contracts of its features, which constitute the documentation of the API's functionalities.

B. Tasks

The study participants solved five tasks involving accessing a relational database using ABEL's features. Solving the tasks requires to store objects of a simple class *PERSON*, also given to the participants, and includes operations found to be critical in previous work [14], [15]. An outline of the five tasks follows.

Task 1: initialize a *REPOSITORY* instance through a factory method; use the *REPOSITORY* instance through a *CRUD_EXECUTOR* to insert instances of *PERSON* into the repository.

Task 2: instantiate a *QUERY* class; use it through the *CRUD_EXECUTOR* to retrieve *PERSON* objects from the repository; and inspect the results.

Task 3: change the state of the *PERSON* objects; update the repository with the new objects using the *CRUD_EXECUTOR*.

Task 4: delete one of the *PERSON* objects and remove it from the repository using the *CRUD_EXECUTOR*.

Task 5: execute a complex query that requires selection criteria. Developers can choose between using "predefined" criteria (implemented using strings) and using function objects (called "agents" in Eiffel, and similar to C#'s delegates).

C. Participants

We recruited 25 participants including 10 computer science students at ETH (6 bachelor's and 4 master's), 7 researchers pursuing a PhD (2 in our group, 2 in other groups of the computer science department of ETH, 2 in the computer science department of other universities, and 1 from the robotics group in the mechanical engineering department of ETH), 2 post-doctoral researchers also at ETH (1 from the computer science department, and 1 from the mechanical engineering department), and 6 professional programmers working for various software companies mostly in the Zurich area. All the participants to the study were unpaid volunteers; the only requirement on our side was that they had at least one year of experience with object-oriented programming.

The following table shows some statistics about their background in terms of: years of programming experience with object-oriented programming languages; years of experience with the Eiffel language; and years of experience as professional programmers.

	min	*median*	*max*	*stddev*
Object-Oriented	1	5	22	4
Eiffel	0	1	18	5
Professional	0	2	21	5

The statistics show that the participant pool covers a wide range of experiences; all participants, however, have a programming background sufficient to make the experiment meaningful. Furthermore, we ascertained that all the participants had at least some familiarity with the basic operations of a relational database, but none of them had used the ABEL library before.

D. Protocol

Each participant performed in an individual session taking place in an isolated office. The first author, henceforth the "proctor", administered all the sessions according to the following protocol.

The proctor starts with a brief overview of the whole process. He then administers a fifteen-minute tutorial showing the basics of the Eiffel language and of the EiffelStudio IDE running on the laptop used for all the experiments. In particular, the tutorial highlights the IDE functionalities useful to browse library documentation (consisting of classes, method signatures, header comments, and contracts) and to inspect the inheritance relations among classes (such as listing all ancestors, descendants, clients, or suppliers of a given class). The only documentation about ABEL available to the participant during the study is accessed using these IDE features.

After the tutorial, the proctor describes the thinking-aloud protocol (Section III) and asks the participant to stick to it during his or her performance. Then, the proctor opens a project consisting of the *PERSON* class and a "main" client class including a terse description of the five tasks as comments (see Section IV-B). The project is set up with the "full void-safety" flag, which entails that the compiler statically checks for possible dereferencing of void references (null references in Java); this helps avoid basic programming mistakes and lets the participants focus on correctly using the API functionalities. Finally, the proctor starts the audio/video recording of the session and invites the participant to begin.

During the experiment, the proctor sits in the same room avoiding interactions with the participant. In a few cases, some of the participants asked for the proctor's instant help, mainly with using functionalities of the IDE or with the syntax of the Eiffel language. The proctor only answered requests that were independent of the specific tasks or the API functionalities, giving the minimal information necessary to proceed. Section VII discusses this potential threat, to demonstrate that its impact on the soundness of the experiments is negligible.

The proctor lets the experiment continue until the participant completes all the five tasks. There is no time limit because the focus of the experiments is assessing usability, not measuring programming efficiency (something would have made not much sense anyway, given the heterogeneous experience of the participants). We still report the time taken by the participants to show that it always was within a reasonable range: the fastest participant finished in 32 minutes, the slowest in 118 minutes, the median time was 70 minutes, and the standard deviation 22 minutes.

After completion of the tasks, the proctor interviews the participant asking the questions of Section III-A and recording his or her answers. This concludes the experimental session.

V. Results

We present the results of the study in subsections corresponding to the four research questions of Section II. For each question, we discuss the data both from the interviews and from the usability tokens, which is summarized in Tables I and II.

Table I classifies answers to each interview question (Section III-A) into "yes", "no", and "sometimes", giving both the absolute number of replies in that category and the corresponding percentage. By recording and checking the behavior of participants during interviews, we ascertained that the classification in three discrete categories is sufficiently reliable: we found no case were the results of mapping the participant's answer to one of the categories was ambiguous or questionable. The few cases were the given answers were considerably more articulate are mentioned in the analysis.

Table II lists the usability tokens occurring most frequently during the experiments; for each token, the table gives an identifier, its type (Surprise, Choice, Missed, Incorrect, or Unexpected) a brief description, the research questions the token addresses, and the number and percentage of participant sessions where the token surfaced.

We focus the discussion on the most significant points; the complete dataset is presented elsewhere [11, Chap. 6].

Q	YES		NO		SOMETIMES	
1	4	16%	0	0%	21	84%
2	4	16%	21	84%	0	0%
12	4	16%	1	4%	20	80%
4	20	80%	5	20%	0	0%
5	2	8%	23	92%	0	0%
6	12	48%	13	52%	0	0%
7	22	88%	3	12%	0	0%
8	17	68%	8	32%	0	0%
9	0	0%	3	12%	22	88%
10	4	16%	21	84%	0	0%
11	23	92%	2	8%	0	0%
3	7	28%	18	72%	0	0%

Table I. SUMMARY OF ANSWERS GIVEN DURING THE INTERVIEWS.

In each of the following subsections, we summarize the results in the first paragraph, and then we present the quantitative data in detail.

A. RQ1: Understandability

The effort required to understand the semantics of API features based on their names and documentation was considered acceptable by most participants, but a few class and method names were found confusing and potentially misleading.

Over 80% of the participants did not have to process information not explicitly part of the API (question 2). This suggests that the API documentation is sufficiently self-contained, and the API classes are acceptable for standard tasks.

The majority (84%) of the participants declared that the API types map to domain concepts only "sometimes" (question 1). More specifically, several usability tokens highlight mismatches between names and underlying concepts. Token T3 corresponds to 44% of the participants not being familiar with the acronym CRUD (Create, Read, Update, and Delete) to represent the basic functions for databases access. As a consequence, the API designer decided for the subsequent release an alternative, more general name such as *EXECUTOR* instead of *CRUD_EXECUTOR*. Token T4 corresponds to 40% of the participants expecting the *REPOSITORY* class to be named *DATABASE* instead, since all the tasks involved relational databases. This is a valid point, even though the name *REPOSITORY* was preferred by the designer because the API also supports serialization operations which do not fit the database abstraction.

Whether the code written matches the expectations (question 3), and hence developers have a positive feedback that reinforces learning, largely depended on the individual backgrounds. Participants not familiar with Eiffel conventions—or familiar but unappreciative—suggested a simplified design (for example, not using command/query separation as discussed in Section V-B). Participants used to work with relational databases at a lower level of abstraction expected to have to implement the object-to-relational mapping themselves, and hence were hesitant to use some of the features of the API that transparently took care of the mapping. These are significant examples of the trade-offs between abstraction and efficiency which populate the rich design space of persistence APIs.

Finally, token T9 highlights a choice developers faced when understanding how updates work (Task 3). The confusion was due to the fact that querying is not necessary before updating to solve Task 3 correctly, because the modified *PERSON* objects are still in local memory from Task 1. In more general situations, however, querying is necessary, and the API documentation reflects this more general scenario which guarantees correctness.

B. RQ2: Abstraction

The abstraction level of the API was largely considered appropriate and the functionalities offered were found suitable to solve the tasks. Nearly half of the participants, however, occasionally found it useful to peek at some implementation details in order to more readily understand relations between classes. This reveals some weak spots in the API abstraction which are to be improved.

Over 90% of the participants did not have to modify the library classes to solve the tasks (question 5). Three of the participants slightly modified some classes by inheriting even if that was not necessary. This usage was benign as it did not break the abstraction of the API; it was done by some of the more experienced Eiffel programmers, probably out of habit since multiple inheritance is used extensively in Eiffel. These were the only cases of "unexpected" usability tokens, which do not feature in Table II, which only reports frequently occurring tokens. This suggests that the API can be largely used as intended by the designer.

The majority (80%) of the participants also found the API abstraction level appropriate to the tasks (question 4). Seven participants, however, were surprised that method *execute_query* does not return the query result (token T11). The reason is that it is customary to practice command/query separation [16] in Eiffel libraries: each method should either be a function (returning a component of the object state without modifying it) or a command (a procedure modifying the object state without directly returning a result). Method *execute_query* is a command, which should be followed by a function call to retrieve its result. Non-Eiffel programmers may not be familiar with this design principle, which reveals a trade-off between design clarity and practical usability.

The fact that 48% of the participants had to understand some implementation details to use the API (question 6) reveals a few significant deficiencies in some abstractions of the API. Specifically, tokens T5 and T12 point to two critical aspects for several participants. Token T5 refers to the fact that

ID	TOKEN	DESCRIPTION	RQs	#	%
T1	M	criterion factory	1	14	56%
T2	C	predefined criterion	2, 3	13	52%
T3	S	what does CRUD mean?	1, 4	11	44%
T4	S	expecting database, not repository	1, 2	10	40%
T5	S	*REPOSITORY* cannot execute CRUD operations	1, 2	10	40%
T6	S	a connection class cannot be used	1, 2	9	36%
T7	C	agent criterion	2, 3	9	36%
T8	C	which strings are valid operators?	1, 4	9	36%
T9	C	read required before update?	1	8	32%
T10	M	default query retrieves all objects	1, 4	7	28%
T11	S	expecting method *execute_query* to return result	1, 2	7	28%
T12	S	*QUERY* also contains query result	2, 4	6	24%
T13	I	reusing the same *QUERY* object without resetting	2, 4	6	24%

Table II. USABILITY TOKENS OCCURRING MOST FREQUENTLY.

ten participants expected class *REPOSITORY* to also directly offer features to access the database, whereas such operations were offered by the *CRUD_EXECUTOR* class. The problem was worsened because *REPOSITORY* does not mention class *CRUD_EXECUTOR* explicitly in the method signatures, and hence some participants decided to inspect its implementation to find references to the other class. Token T12 is connected to token T11, and prompted a few participants to peek into the implementation of *QUERY* to understand how to access result objects. Such difficulties in understanding relations between types and in discovering new classes corroborate previous empirical observations along the same lines [14].

C. RQ3: Reusability

The participants agreed that they managed to write concise client code in an incremental fashion, and that their solutions were reusable to solve variants of the problems. They also agreed that the API offers different ways of implementing certain functionalities.

Over 90% of the participants thought that it would have been easy to modify their code to access different database or to perform different queries (question 10). This is an important goal which the API seems to achieve satisfactorily. Nearly 90% of the participants also found "about right" the amount of code they had to write (question 7). Even the few in disagreement mainly found the keywords and names a bit verbose, but did not express a strong criticism about succinctness.

Nearly 70% of the participants positively answered question 8, which asks whether it was easy to keep track of progress while solving the tasks. The recurring problem that the other 30% experienced was that they had no simple programmatic way to clean up the database after failed attempts and restore it to the original state to try again. This was more a deficiency of the experimental setup than a flaw of the API, even if quick trials are something that could often be useful to support programmers learning an API.

The answers to question 9 show that the API provides alternatives to solve certain tasks. This carries both a positive and a negative connotation. On the positive side, it shows that the API has a certain flexibility, and that certain tasks can be solved very concisely (using the defaults) or less concisely but with more control on the individual steps. This was particularly true for the query mechanism, which offers some simple defaults but also more flexible features based on *CRITERION* classes. On the negative side, choice may also be confusing or slow down programmers, as evidenced in a couple of the usability tokens. Token T2 reveals that 52% of the participants pondered whether a predefined criterion or a default query was better suited for Task 2. Token T7 concerns a similar choice of how to use criteria with agents (Eiffel's function objects), useful in Task 5 but unnecessary in the other tasks. Neither token highlights flaws in the API, but both suggest possible elements of design simplification.

D. RQ4: Learnability

The learning curve for the API was initially steep, as it required to become familiar with a few non-trivial abstractions. After the initial learning phase, however, solving more advanced tasks became relatively simpler, as the learning curve flattened.

Over 90% of the participants agreed that they became more efficient after completing the first two "exploratory" tasks (question 11). The two participants who disagreed were slowed down by the choice offered by the *CRITERION* classes, already discussed in relation to RQ3 in Section V-C.

Three usability tokens point instead to usability issues which negatively impact learnability and other aspects. The three tokens originate in incomplete or inaccurate documentation (in terms of comment and contracts, see Section IV-A). Token T10 shows that seven participants missed the fact that a *QUERY* object returns all objects of its generic parameter type "by default", that is when created with the argumentless constructor. Indeed, this is not clear from the constructor's comments and could only be surmised indirectly or by looking at the constructor's implementation. Token T8 points to a deficiency in the documentation of a constructor of the *CRITERION* class: while its precondition imposes a constraint on its string argument, the semantic of the constraint is obscure as it involves how strings represent operators (e.g., "**and**" rather than "&&" for logical conjunction). Finally, token T13 reveals that six participants incorrectly used the same *QUERY* object multiple times without calling a *reset* method after each usage. This is indeed something not adequately documented, and a good example of the kinds of issues usability testing may find.

Over 70% of the participants did not have to become familiar with many classes to use the API (question 12). This suggests that the API documentation is sufficiently self-contained, and the API classes are acceptable for standard tasks.

Three of the seven participants who claimed to have learned "many classes and dependencies" missed a concrete factory

class [17] named *REPOSITORY_FACTORY*, which was quite useful to set-up repositories with only few operations. This is connected to a more general problem of using factories instead of ordinary constructors to create objects and initialize them. *REPOSITORY_FACTORY* is also the source of token T6, where 36% of the participants were confused by not having to use class *CONNECTION* to establish a database connection; in fact, the factory took care of establishing the connection. The most frequently occurring usability token T1 corresponds to participants who missed the other concrete factory *CRITERION_FACTORY* to create criteria (a form of query useful in Tasks 2 and 5). Since it is still possible to instantiate criteria without using the corresponding factory, all participants could successfully complete Tasks 2 and 5 even if they had problems with using factories. For API designers, however, this casts some doubts on the practicality of concrete factories, corroborating previous empirical findings on this design pattern [15].

VI. Discussion

We now summarize the overall findings of our study in more general terms than Section V. We organize the discussion in three parts: Section VI-A targets issues with API usability that were discovered in previous work and replicated in our study; Section VI-B targets issues that emerged in previous work but were not critical in our study; Section VI-C discusses new findings and lessons that specifically emerged in our study.

In the following discussion, we also report on the usability issues that are significantly affected by the previous experience of developers. We partitioned the interview answers into two groups: 11 "experienced" participants, whose years of experience with object-oriented programming is greater than the median; and the other 14 "novice" participants with below-median experience. In the reminder, we point out the few questions whose answers look qualitatively different in the experienced and novice groups.

A. Confirmation of Previous Issues

Associating API feature names to functionalities is a problematic issue because it is hard to select names that conform to the heterogeneous jargon of programmers and are descriptive but not verbose. Even the basic object-oriented terminology may vary from language to language: for example, Eiffel calls "routine" [16] what normally is a "method" in Java [18] and a "member function" in C++ [19]. Another difference is in the naming convention for features, such as using the prefix "*is*" for functions returning Booleans—such as in *is_empty*. Specific to the persistence domain targeted in our study, we discussed the problem with the class name *REPOSITORY* vs. the more specific one *DATABASE*, and the unfamiliarity of several programmers with the acronym CRUD which is, however, popular in the database community [20]. Our study reinforced the lessons learned by others [21], [22], [23]: name API features consistently and use common names that are still descriptive and not vague.

Another issue which was found problematic by others [14], as well as in our study, is discovering relationships between types. This issue typically emerged in connection to known usability difficulties [15] with the concrete factory pattern to create objects without calling constructors. Specifically, we often detected a clash between the expectations of developers and the usage intended by the API developer. Given the central role of classes and types in object-oriented programming, developers are used to ground their understanding of the API on the relationships between types; it is thus confusing when these are not crystal-clear from reading the documentation. Instantiating classes using constructors is also almost second-nature to object-oriented programmers; when factories should be used instead, the API design must be tailored to emphasize this exceptional usage.[3] None of the experienced developers, however, reported difficulties in discovering relationships between types. Since the same group also declared that they hardly needed to look at the implementation to find their way through the API, this suggests that experienced programmers have enough flexibility to interpret non-plain choices of API design, whereas the novices need significantly more support.

Determining the outcome of method calls is another issue mentioned in other studies of API usability [24]. Developers tend to rely on a method's return type to access its result; when the method is a procedure not returning anything, they incur a cognitive overhead. As we discussed in Section V-B, the Eiffel design style involves the command/query separation, which caused issues with determining the outcome of method calls in our study. This shows a conflict between design clarity and practical usability.

B. Potential Issues Not Critical

A positive note of our study is the infrequent occurrence of usability tokens characterizing *incorrect* or *unexpected* usage of the API (see Section III-B and Table II). This suggests that, even if programmers may have been slowed down by other deficiencies, the API design normally avoids at least the most obviously incorrect usages and makes it hardly necessary to override the designer's intentions. This is one way to resolve the friction between correctness and practical usability.

A more specific issue emerged regarding argumentless constructors. Argumentless constructors are used with (but not limited to) a create-set-call style, whereas constructors with arguments are used with a create-call style. Previous work suggests [25] that argumentless constructors are preferred, that is considered easier to use, to constructors with arguments. In our study, however, the participants had no particular difficulties with using constructors with arguments, which are in fact extensively present in the ABEL API. On the contrary, we had one case of argumentless constructor (of class *QUERY*, discussed at the end of Section V-D) which was found confusing because its documentation did not spell out what the default behavior was. More generally, choosing argumentless vs. with-argument constructors also exercises a friction between correctness and practical usability. Eiffel classes use invariants to characterize valid object states; constructors must return objects satisfying their invariants, which then every method must preserve. To take advantage of this guard against incorrect behavior (particularly useful for consistent object storage [11]), constructors may require arguments to correctly initialize objects without relying on successive method calls

[3] Note that this issue applies only to the *concrete* factory pattern; abstract factories export abstract types, which cannot be directly instantiated using constructors [17].

which programmers may forget. Hence, such a stricter design style comes at a cost but also brings tangible benefits.

C. Other Lessons

A recurring lesson emerging from our study—and an unsurprising one at that—is the critical importance of having accurate, unambiguous, and self-contained documentation [26]. While previous work suggests that API documentation should ideally include code snippets and tutorials [27], the participants of our study faltered whenever the API documentation given to them was imprecise or incomplete, i.e. omitting details relevant to the task at hand (see Section V and [11] for examples). The lesson should be familiar, but is worth emphasizing: bad documentation is a nonstarter.

Somewhat related to the problem of documentation is another dimension of the API design space which surfaced in relation to the usability token T8 (discussed at the end of Section V-D). Developers had trouble understanding how to instantiate an argument of string type with a valid representation of logic operators. While this is also an instance of incomplete documentation, the operators passed as argument could have been represented as types instead of constant strings. This would have required some additional effort to discover the new types, but would also have removed the ambiguity and made it possible to check the actual argument at compile time (as a type constraint) rather than relying on the weaker checks done at runtime.

A final lesson emerged from our study is the role of *choice*, also discussed in Section V-C. When the API provides different ways to solve a task, programmers have more flexibility but also more difficulties to fully understand the API design. When answering questions during the interview, the more experienced developers tended to emphasize the positive connotation of choice, whereas the less experienced ones often considered choice as a negative feature. It is clear, however, that, even for experienced developers, choice is positive only when the options are really complementary, rather than being just unnecessary complications of the design.

VII. Threats to Validity

We discuss the main threats to validity and what measures we deployed to minimize their impact.

A. Construct Validity

A multifaceted feature such as usability can elude attempts to get objective measures. In our study, we addressed such potential threat to construct validity with a careful design which follows well-established protocols (the cognitive dimension framework and the thinking-aloud protocol for data collection). As we mention in Sections IV and VIII, these protocols have already been successfully used for usability studies, which vouches for their soundness. Additionally, we tried to complement the inherent limitations of structured interviews by also analyzing the recordings of the experiments according to a few "usability tokens" (Section III-B). Combining the explicit answers given during interviews with the implicit tokens emerged during the study gives a richer set of data which help reduce the impact of inaccurate measurements.

B. Internal Validity

A possible bias exists in the selection of study participants, most of whom were known to the first author (such as former students, former colleagues, or their acquaintances); none of them, however, knew the author's work on ABEL or participated in any way to the design and preparation of the study. The participants' programming background was sufficiently heterogeneous (Section IV-C) to guarantee that the study is representative of programmers with quite different experience. We do not believe that the lack of complete strangers has influenced the outcome of the study in any significant way; it may have even been conducive to reducing performance stress, and hence removed a potential factor of disturbance.

The performance of the study participants might have been different, and possibly better representative of usability in standard conditions, if they had been allowed to access external documentation (for example, about Eiffel or the IDE) and to search for code snippets on the Internet. However, we introduced this restriction to focus the evaluation on the features of the API and its official documentation; evaluating the effects of browsing the web on programmer performance is a problem in its own right [24], whose consequences would have been difficult to control for.

Collecting answers to questions through an interview carries a risk of "interviewer effect", where the interviewer biases the answers by giving involuntary subconscious clues [28]. A similar threat is involved in the proctor sitting through the programming sessions, where he occasionally answered simple questions by the participants (Section IV-D). We minimized the impact of these two threats as follows. First, the proctor only answered generic questions about the Eiffel language or the IDE used during the study, giving succinct verbal answers and avoiding as much as possible any reference to the specific functionalities of the API evaluated in the study, or to the tasks to be solved. Second, the interviews was structured and consisted only of twelve predefined questions, which the interviewer read from a printout. Third, all sessions were recorded; after the experiments we replayed the recordings checking that the intended protocol was followed. During the replaying, we noted down the interactions between the proctor and the participants. As required by the protocol, all interactions were only about Eiffel syntax details or EiffelStudio IDE functionalities. To better identify potential sources of such requests, we measured correlations between the participants' background data (Section IV-C) and the number of requests for clarification, using Kendall's τ. We found a significant negative correlation with Eiffel experience ($p \simeq 0.003$, $\tau \simeq -0.59$), and a significant positive correlation with time to complete the tasks ($p \simeq 0.0002$, $\tau \simeq 0.44$). This gives independent support to the claim that this threat did not have a significant impact: the more Eiffel experience a participant has—and the faster the participant is—the fewer clarifications he or she requests. The negative correlation with Eiffel experience also somewhat explains the effect of having 5 out of 25 participants who were not familiar with the programming language of the API: these developers compensated by asking more language-related questions, which limited the effect of this potential threat and ultimately contributed some interesting insights originating in their different background.

C. External Validity

The main threats to the generalizability of the findings of our study come from the fact that it targeted a single API and a single programming language. In fact, some of the issues emerging during the study are somewhat specific to the application domain or to the Eiffel design style. However, the discussion in Section VI also shows that the study confirmed several issues that emerged in other API usability studies targeting different languages and different domains. This suggests that the gist of our findings are also applicable to different contexts.

VIII. Related Work

Despite the critical impact that API usability seems to have, there are only a few rigorous studies of API usability in the literature. This section briefly reviews them and highlights the connections with the rest of the paper.

An early attempt at investigating API usability analyzed the role of *examples* to help design understandable APIs [29]. According to the study, APIs designed around examples are simple to use for programming tasks that follow closely the original examples. Conversely, when developers need to use the APIs in scenarios significantly different from the original examples, they may prove hard to use or inadequately designed. A more recent study [30] presents an automatic technique for mining and synthesizing API usage examples. In our empirical study, we did not specifically consider the role of examples but focused on the evaluation of the usability of a self-documented API solely based on its features' signatures, comments, and contracts. Some of the difficult issues developers face when programming using APIs are understanding the rationale of design decisions and answering questions about APIs that are not covered by documentation or tool support [31]. These observations suggested to restrict our study participants to access only the official API documentation plus take advantage of IDE support, so as to focus the evaluation on the actual API design rather than on external factors such as generic documentation, other programmers' suggestions, or specific hands-on guidance (as in hierarchy-focused techniques [32]).

Software development practices for API design are typically specific to one programming language, such as C# [21], Java [22], or C++ [23]. [33] suggests to generalize the techniques used to evaluate specific APIs in order to investigate the impact on usability of different design choices. To this end, [33] uses APIs created ad hoc to compare the usability of specific features and to evaluate the client code developers would write in an abstract setting. Related work [34] attempted to reconstruct the relations between dimensions in the space of API design and their impact on usability, and in particular outlined the involved trade-offs. Our study also evidenced some of these trade-offs, but based on an "in vivo" empirical study based on an actual API.

A recent study about the performance of developers facing unfamiliar APIs [24] highlighted several important issues that were also confirmed in our study. Developers often have difficulties associating API feature names to functionalities, discovering relationships between API types (i.e., classes), and determining the outcome of method calls for methods that do not explicitly return values. Specifically, developers tend to expect methods to return values that notify about the success or failure of a method call. We observed a similar problem in the case of a method that executes a query, where almost one fourth of the participants to our study expected the method to also explicitly return the result of the query itself.

Another issue raised in [24] as well as in ours and in other studies regards the fact that programmers rely on feature names to make educated guessues about the features' semantics, especially when the documentation is difficult to access or incomplete. The task of selecting the appropriate abstractions creates then potential selection barriers [2]. For example, an empirical study of the names used in Java API class names and JavaDoc documentation [35] shows that the most frequently used words are "Exception", "UI", "Helper", "Type", "Event", and "Factory". Programmers who become familiar with such a terminology may then experience a worse usability when working with frameworks sticking to different jargon, as we discuss in Section VI-A.

Yet another issue raised in [24] and as well [14] and confirmed in our study regards the recurring difficulties of discovering relationships between API types. [14] points out that types are difficult to discover when they are not mentioned as attributes, local variables, arguments, or even in comments. In our experiments, the issue of discovering types emerged mostly in connection with another usability issue reported in related work [15]: effectively using the concrete factory design pattern [17]. Combining evidence from different sources, it is remarkable that the word "Factory" is frequently used to label API features [35] and the concrete factory pattern is one of the most used; and yet a source of common usability problems.

The thinking-aloud protocol [7], used in our experiments for data collection, is widely used in usability testing. We applied it in a less strict variant [8] suitable for usability tasks where not only the human subjects but also the "product" being used (in our case, an API) should be inspected and evaluated.

Methods to *improve* API usability are a natural complement to techniques to assess the usability. One approach [36] consists of extracting additional information from the API documentation—such as usage rules and special cases—and displaying such information within the IDE whenever the information is relevant to the current activity. Another approach [37] suggests that documentation writers should be involved early on during API design stages; applying text analysis and other documentation writing techniques can help to write APIs with improved usability.

IX. Conclusions

This paper presented the design of an empirical study to assess the usability of an API by finding issues that may hamper it. The design is based on the idea of comparing the expectations of programmers to their actual performance on programming tasks requiring to write client applications using API features. To provide a richer characterization of their behavior, the study collected programmers' feedback both directly—through structured interviews with questions based on the cognitive dimensions framework—and indirectly—by observing recordings of the performances and classifying

episodes into "tokens" revealing of usability issues. We executed the study with 25 programmers (students, researchers, and professionals) working on a persistence library written in Eiffel. The study confirmed usability issues that emerged in related work, such as the difficulty of assigning names to API features and of discovering relations between types (i.e., classes) of the API. It also found how several usability flaws are ultimately due to incomplete or unclear documentation; and revealed that flexible features are appreciated by experienced programmers but may disorient novices.

ACKNOWLEDGMENTS

We would like to thank Martin Robillard for his useful comments on preliminary versions of this content; Roman Schmocker for his implementation work on the ABEL library; and all the participants to the usability study. Work partially supported by the ERC grant CME/291389.

REFERENCES

[1] K. Arnold, "Programmers are people, too," *Queue*, vol. 3, no. 5, pp. 54–59, Jun. 2005.

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

[3] M. Henning, "API design matters," *Commun. ACM*, vol. 52, no. 5, pp. 46–56, May 2009.

[4] A. F. Blackwell, C. Britton, A. L. Cox, T. R. G. Green, C. A. Gurr, G. F. Kadoda, M. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young, "Cognitive dimensions of notations: Design tools for cognitive technology," in *Cognitive Technology*, 2001, pp. 325–341.

[5] A. C. Kay, "Quoted in: The wiki way: Quick collaboration on the web by B. Leuf and W. Cunningham," http://en.wikiquote.org/wiki/Alan_Kay, 2001.

[6] S. Clarke, "Measuring API usability," http://www.drdobbs.com/windows/measuring-api-usability/184405654, 2004.

[7] K. A. Ericsson and H. A. Simon, *Protocol Analysis: Verbal Reports as Data*. MIT Press, 1984.

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

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

[10] J. Dagit, J. Lawrance, C. Neumann, M. M. Burnett, R. A. Metoyer, and S. Adams, "Using cognitive dimensions: Advice from the trenches," *J. Vis. Lang. Comput.*, vol. 17, no. 4, pp. 302–327, 2006.

[11] M. Piccioni, "A seamless framework for object-oriented persistence in presence of class schema evolution," Ph.D. dissertation, ETH Zurich, 2012.

[12] "Spring framework data access," http://www.springsource.org/features/data-access, last visited: 15.03.2013.

[13] "Hibernate relational persistence for Java and .NET," http://www.hibernate.org, last visited: 15.03.2013.

[14] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *FSE*, 2008, pp. 105–112.

[15] B. Ellis, J. Stylos, and B. A. Myers, "The factory pattern in API design: A usability evaluation," in *ICSE*, 2007, pp. 302–312.

[16] B. Meyer, *Object Oriented Software Construction*, 2nd ed. Prentice Hall PTR, 1997.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[18] J. Gosling, B. Joy, and G. L. S. J. and Gilad Bracha and Alex Buckley, *The Java Language Specification, Java SE 7 Edition (Java Series)*. Addison-Wesley Professional, 2013.

[19] B. Stroustroup, *Programming: Principles and Practice Using C++*. Addison-Wesley Professional, 2008.

[20] J. Martin, *Managing the Data-base Environment*. Pearson Education Canada, 1983.

[21] K. Cwalina and B. Abrams, *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley Professional, 2008.

[22] J. Tulach, *Practical API Design: Confessions of a Java Framework Architect*. Apress, 2012.

[23] M. Reddy, *API Design for C++*. Morgan Kaufmann, 2011.

[24] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," in *ICSE*, 2012, pp. 266–276.

[25] J. Stylos and S. Clarke, "Usability implications of requiring parameters in objects' constructors," in *ICSE*, 2007, pp. 529–539.

[26] D. L. Parnas, "Precise documentation: The key to better software," in *The Future of Software Engineering*. Springer, 2011, pp. 125–148.

[27] M. P. Robillard, "What makes APIs hard to learn? Answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 26–34, 2009.

[28] F. J. Fowler and T. W. Mangione, *Standardized Survey Interviewing: Minimizing Interviewer-Related Error*. Sage Publications Inc., 1989.

[29] S. McLellan, A. Roesler, J. Tempest, and C. Spinuzzi, "Building more usable APIs," *Software, IEEE*, vol. 15, no. 3, pp. 78–86, May/Jun.

[30] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *ICSE*, 2012, pp. 782–792.

[31] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *ICSE*, 2007, pp. 344–353.

[32] F. Shull, F. Lanubile, and V. R. Basili, "Investigating reading techniques for object-oriented framework learning," *IEEE Trans. Software Eng.*, vol. 26, no. 11, pp. 1101–1118, 2000.

[33] J. Stylos, "Informing API design through usability studies of API design choices: A research abstract," in *VL/HCC*, 2006, pp. 246–247.

[34] J. Stylos and B. A. Myers, "Mapping the space of API design decisions," in *VL/HCC*, 2007, pp. 50–60.

[35] C. Anslow, J. Noble, S. Marshall, and E. Tempero, "Visualizing the word structure of Java class names," in *Companion to OOPSLA '08*. ACM, 2008, pp. 777–778.

[36] U. Dekel and J. D. Herbsleb, "Improving API documentation usability with knowledge pushing," in *ICSE*. IEEE Computer Society, 2009, pp. 320–330.

[37] R. B. Watson, "Improving software API usability through text analysis: A case study," in *IEEE IPCC*, 2009.