Unit Tests as API Usage Examples

Seyed Mehdi Nasehi, Frank Maurer
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{smnasehi, frank.maurer}@ucalgary.ca

Abstract**—***This study aims to find out if API unit tests can provide good usage examples, and if so, what prevents developers from finding and using those examples. The results of an experiment we performed with two groups of developers showed that unit tests can be very helpful, especially when the task is complicated and involves multiple classes and methods. Well-written tests proved to be a good source of examples, but finding the relevant examples using the standard tools might be very difficult. We propose to supplement the standard API documentation with relevant examples taken from the unit tests. To further improve the learnability of the API, presentation of the documentation and examples has to be tailored in a way that separates or hides advanced usage scenarios from the commonly used ones.*

Keywords: API, Usability, Unit Test, Documentation, Code Example

I. INTRODUCTION

An Application Programming Interface (API) is the interface to existing software functionality that can be accessed to perform various tasks. APIs facilitate code reuse, make programming tasks easier by providing high-level abstractions, and help unify the programming experience [19]. They are widely used in software development and using them becomes a larger part of programming. Learning new APIs becomes essential in both the software development and maintenance phases. To build software, developers almost always have to use one or more APIs. Knowledge of APIs is crucial to software maintainers that need to understand the code in order to fix bugs, modify, or extend functionality. Apart from learning the basics of an API, software developers are interested to know what the best practices of using them are.

The number of new APIs that programmers need to learn is ever increasing. There is already thousands of APIs around and among them, there are very large APIs, containing thousands of classes and tens of thousands of methods. The Java JDK and .NET framework are two well-known examples of such huge general-purpose APIs. Domain-specific APIs, on the other hand, are used on a daily basis by software maintainers. The most challenging aspect of APIs for software maintenance is to learn how to use them correctly and effectively. Software maintainers usually have to deal with a variety of systems and components. They often have to use many different APIs during their daily work and might not have used any of them recently. Considering the sheer diversity of APIs, it is almost impossible for anyone to be familiar with all of them. Even if a developer has been using a large API for years, they might not be fully familiar with all parts of it.

Developers rely on different information sources to learn about APIs. The standard documentation of the API, sample code, online tutorials, and books are some examples of the supporting resources. Ideally, a developer would be able to learn how to use the API quickly using the aforementioned supporting materials. Unfortunately, many APIs lack these artifacts or come with imperfect or low quality content. For example, the standard mechanism of producing documentation in Java is to extract the Javadoc comments from the code and build hyper-linked documentation from them. The user browses the Javadoc documentation by selecting a class. They are then given a list of the methods that are normally accompanied by short descriptions. However, to use the Javadoc and similar forms of documentation, the user has to know what they are looking for. Otherwise, browsing the Javadoc documentation of a large API would be a tedious task that usually ends without any success.

Ko et al. found six learning barriers which users of programming interfaces would encounter [13]. These barriers are the inherent difficulty of the problem (design), finding available interfaces (selection), combining available interfaces (coordination), using available interfaces (use), evaluating the external behavior of the program that does not match the expectations (understanding), and acquiring info about a program's internal behavior (information). According to Ko et al. using examples helped programmers overcome selection and coordination barriers.

For general-purpose APIs, developers could search the Internet for good examples using one of the code search engines, like Google Code Search[1]. However, it is not an option for internal APIs developed in house. Developers and maintainers need a good source of examples for their internal APIs to facilitate their maintenance tasks. Maintaining a collection of good API usage examples can come handy to software maintainers. They will be able to utilize the examples, alongside with other resources, to educate themselves or refresh their memory on an API. However, creating and maintaining a collection of examples can become undesirable; it would simply create more development and maintenance effort. Reusing existing examples therefore could result in cost

[1] http://www.google.com/codesearch

This research is funded in part by NSERC Discovery Grants Program.

savings due to eliminating the need to develop the examples from scratch.

This gives rise to the question: "Where can we find reliable examples?" The answer lies in the API unit tests. Unit tests are executable tests that verify the smallest separate module in the code [21]. They are developed using a unit-testing framework, like xUnit. Unit tests invoke the class or method under test and make assumptions about the logical behavior of that element [18]. They are often written by the very developers who develop the code and are maintained together with the rest of the code. When changes are going to be applied to the code, its unit tests must also pass. If the tests fail, either the code or the tests will have to be modified to make it pass. In other words, the consistency between code and tests has to be preserved. This would ensure that changes made to the code do not break anything. Zaidman et al. showed that the code and its unit test are not always kept in sync [27]. This is not the case for agile teams, since all the tests must pass during each build [15].

Having a comprehensive test suite helps developers to perform change tasks during the maintenance phase faster with fewer defects [8]. Therefore, it is crucial to keep an up-to-date collection of unit tests during the software development and maintenance phases. In addition to their validation and verification role, unit tests can come handy in providing developers with insight into the way the unit under test is used. They can be considered as one of the sources of documentation of the system under test. Tests act as a suite of examples that help developers figure out how to work with the code [15]. If tests are kept up-to-date with the code, they can serve as up-to-date usage examples of the unit under test. However, it is not clear if unit tests are a good source of API usage examples.

Would the developers new to an API benefit from looking into the API unit tests? Do unit tests provide helpful usage examples that facilitate the learning process? To answer the above questions, we conducted a study that investigates the effects of using API unit tests and documentation as the main source of developer information in a setting involving four development tasks. Qualitative analysis of the experiment results revealed that unit tests are in fact very helpful, especially when the task is complicated and involves multiple classes and methods. We also found that well-written unit tests are good sources of examples, yet finding the right test scenario might be a bit tricky. On the other hand, the standard documentation had limited usefulness, unless the users already have access to good examples. A number of good examples, which for instance can come out of the unit tests, could still improve the usefulness of the standard documentation.

This study makes the following contributions:

- It adds to the empirical body of knowledge about treating unit tests as the system documentation.
- We provide suggestions to enhance the API standard documentation by examples extracted from unit tests.
- We provide suggestions to test developers for developing unit tests that can serve as better API usage scenarios.

The rest of the paper is organized as follows: The next section provides the list of the study questions and describes the design of our study. It also explains how we planned to find the answers to those questions. Section 3 describes the analysis of the data we gathered during the study. Section 4 describes the results we obtained from the analysis of the experiment data, and provides study observations, hypotheses, and (design) implications. Section 5 explains threats to the validity of the study. Section 6 summarizes the related work and relates our findings to that of related studies. Section 7 concludes the paper.

II. Methodology

Our goal was to determine the usefulness of unit tests as a source of usage examples of an API. We have been seeking to answer the following three main questions:

Q1. How useful is the standard documentation of an API for learning it?

Q2. Do unit tests provide useful examples, especially for developers new to a given API?

Q3. Do unit tests improve the learnability of the API?

To address these questions, we conducted a study. We asked participants to do four programming tasks using an unfamiliar API. For the study to be valid it must have met the following requirements:

R1. The study needed to resemble reality. Our goal was to investigate the effects of using unit tests as examples in learning a new API. So, we had to choose an API that was not familiar to most of the participants, yet the functionality it provided had to be familiar. In other words, participants had to be familiar with the domain of the development tasks, but should have never done it with the API used in the context of this study.

R2. The programming tasks had to be real scenarios of using the API. We had to choose some scenarios that were representatives of the API usage. Moreover, the development scenarios had to be suitable to be done in a limited amount of time.

R3. The participants needed to be representatives of the programmers who are exposed to the selected API.

R4. The API needed to have a good coverage of unit tests.

A. *The API*

To meet these requirements, we selected the Apache POI API[2], version 3.5. The API is an open source project developed in Java to provide the manipulation of the Microsoft Office documents for Java programmers. The programming tasks of the study were manipulations of Microsoft Excel spread sheets. The subset of the API that provides the spreadsheet support is called HSSF. There is another project to deal with the Excel 2007 OOXML file format which is called XSSF. We did not use the second subset in our study. The tasks were designed to be done using the HSSF API. The overview page of the API

[2] http://poi.apache.org/

describes it as a mechanism to manipulate spread sheet files. It states that the API contains low level structure for the special needs, an event-model API for efficient read-only access, and a high level API for creating, reading, and modifying spread sheet files. The latter part of the API is called the usermodel API[3].

The selection of this API fulfilled three study requirements: Most of the participants had not worked with it prior to the study (R1). At the same time, all of them were familiar with the notion of spread sheets and had been using the spread sheets beforehand (R2). Using EclEmma[4], we found that the total coverage of the API is 89.3%, while the coverage of `hssf.usermodel` package, that contains most of the high level classes, is 93.6% (R4).

B. *The Design of the Study*

1) *Tasks*

The study comprised four programming tasks. The first three tasks were programming tasks that subjects had to start from scratch. The fourth one was a completion task that needed to be finished by the subjects. Here is a brief description of each task:

Task 1: Subjects were asked to create a spread sheet, called a workbook in the API documentation, with one sheet. The first four columns of the sheet needed to be filled with numbers 1 to 4. They were also asked to define a named range that contained all the filled cells and then save the spread sheet. This was a warm-up task to help them to familiarize themselves with the API.

Task 2: This task was about the cell comments. The subjects were asked to read a spread sheet file and change the author and the text of the cell comments based on the values of these fields. Finally, they had to save the modified spread sheet to another file.

Task 3: This task was the most challenging one. The subjects were asked to create a spread sheet and define conditional formatting rules for the first two columns of the first sheet. A conditional formatting rule changes the appearance of a cell based on the content of the cell. The first rule was to change the background color of the cells whose values are greater than 100. The second rule, applied to the second column, needed to change the background color of the cells with duplicate values. The formula for finding duplicate values had been supplied in the task description sheet.

Task 4: This task was a completion task. The subjects were given a sample spread sheet file, and then had to complete a semi-complete program to produce a similar output file. They needed to create cells, put values in them, change the appearance of the cells using cell styles, and set formulas for some cells.

We conducted a pilot study with two graduate students before we started the actual study to assess the difficulty of the tasks and the clarity of the task descriptions. Based on the outcome of the pilot study, some minor changes in the task descriptions were made and tasks 3 and 4 were further simplified.

2) *Subject Selection*

Subjects were recruited through the mailing lists of the Department of Computer Science at the University of Calgary and the Calgary Agile Methods User Group (CAMUG). Subjects were required to be medium level Java programmers, i.e. they had to have at least one year of Java programming experience. The four tasks had been designed so that they did not need to have knowledge about sophisticated Java programming concepts, like multi-threading. 16 participants took part in the experiment; 13 graduate students, 2 senior undergraduate students and one participant from the industry. Each participant was paid $30 for the two hours of experiment.

3) *Study Setting*

The study consisted of two main phases. The first phase was the initial learning phase in which the subjects were given a short tutorial document, adapted from the HOWTO and the Quick Guide pages of the POI website. We selected a subset of the original pages, due to the limited time of the experiment. The tutorial introduced basic abstract classes of the API, namely `Workbook`, `Sheet`, `Row`, and `Cell` and their concrete class relatives. The following sections were selected from the online HOWTO page: creating a new spread sheet, reading from an existing file, named ranges, and comment creation. The tutorial covered how to create new workbooks and sheets, how to add rows and cells, how to set cell values, how to change their appearance, and how to save the workbook to a file. It also contained code examples for all the mentioned topics. The API has some classes that were developed to provide API usage examples, and the examples in the tutorial document were actually retrieved from them. Subjects had a maximum of 20 minutes to read the tutorial document and to take notes. They were allowed to use it during the next phase.

The second phase was a programming phase. A fixed amount of time was allotted to each task. Participants were given 15 minutes for the first task, 20 minutes for the second task, 20 minutes for the third task, and 45 minutes for the last task. The Eclipse IDE[5] was used for the development tasks. At the start of the second phase, participants were given a cheat sheet that described the overall objectives of the phase. It also provided hints on the name of the package that contained the high level concrete classes for manipulating spread sheets. The name of the package was already mentioned in the tutorial document, by repeating it in the cheat sheet we meant to help the participants staying focused and not get lost in the irrelevant parts of the API[6]. The participants did not have access to the Internet during the study, so they could not find related material by searching the net.

The subjects were assigned randomly into two groups. The first group, which we refer to them as the UT group, was given access to the unit tests of the API during the programming phase. The unit tests were placed in a separate Eclipse project

[3] http://poi.apache.org/spreadsheet/index.html

[4] http://www.eclemma.org

[5] http://www.eclipse.org

[6] All the documents that were given to subjects can be downloaded as a compressed file from: http://ase.cpsc.ucalgary.ca/uploads/Main/SMNasehi-ICSM2010-Study-Documents.zip

in the same workspace. The unit tests follow the same packaging structure and naming convention as the API itself. Hence, the unit tests of the class `X` is named `TestX` and is located in the same package. This information was included in the cheat sheet that was given to the participants at the start of the programming phase. They did not have access to the Javadoc documentation of the API.

The second group, which we refer to them as the JD group, was given access to the standard Javadoc documentation of the API. The Javadoc documentation was provided in a separate browser window during the study and participants were able to consult it during the programming phase. The JD group was also provided with a cheat sheet that contained information about the package containing the high level classes.

C. Data Collection

We collected the programs that were developed by the subjects during the programming phase. The subjects were asked to think aloud while they were doing the programming tasks. We recorded their voice and captured a video of the screen using the Camstudio software[7]. The screen capturing was done at a frame rate of 8 frame/seconds with a resolution of 1440 by 900 pixels. The subjects were asked to fill a short survey after finishing the programming phase. The survey consisted of statements about the usefulness of the API and the information resources provided during the experiment. We used a combination of semantic differential statements and Likert scales with a 7 point scale to get the subjects' level of agreement with the statements [25].

III. Data Analysis

We measured whether or not subjects were successful doing a programming task, and how much time they spent completing each task. Almost all the subjects used the entire allotted time for each task; therefore completion time was not a good differentiating factor for judging developer success. We decided to base our analysis on the success/failure of the tasks. Accordingly, a system of three-valued success levels for the tasks, similar to [25], was devised. The task could be a complete success, a partial success, or a complete failure. We further subdivided each task to a set of sub-tasks and adapted a system of four-valued success levels for each sub-task adapted from [20]. A sub-task could be labeled as success, partial success or buggy, unworkable, or not attempted. The partial success means that the subject implemented the sub-task with the right classes and methods, but it contained some problems that needed to be fixed. The unworkable sub-task means that the sub-task was implemented with irrelevant classes and methods and could not be easily fixed. The sub-task success levels were assigned by examining the final code developed for each task. Table 1 provides the breakdown of each task to its sub-tasks, and how the success levels of its sub-tasks map into the success level of the task itself.

We consider the task to be successful if the subject managed to finish all but one sub-task. The remaining sub-task should also be partially successful. It is interpreted as the subject was on the right track to finish the task, and they could have finished it successfully had they had been given more time.

The partial success level of each sub-task was measured differently for each task. Task 1 was a warm-up task and all the needed information was provided in the tutorial document. Since it was the first time that subjects were exposed to the API, we were expecting that they would have some difficulty using it. Therefore, we were lenient in measuring this task as a partial success. The first and the last sub-tasks of tasks 2 and 3 were quite straightforward and were addressed in the tutorial document. If a subject completed one of these sub-tasks successfully and did one of the remaining sub-tasks with partial success, the task is considered as a partial success. Comment creation was already addressed in the tutorial document, but Task 2 asked for comment modification. Subjects had to seek the required information from additional sources. Task 3 was about creating conditional formatting rules that was not covered in the tutorial document. Again, subjects had to consult additional sources to find relevant information. Task 4 merely consisted of sub-tasks that were addressed in the tutorial document. The task is considered as a partial success, if 4 out of 6 sub-tasks were completed successfully.

The next step of the data analysis was to find out why some participants were unsuccessful in doing tasks. To do this, we transcribed the recorded subject voices and screen captures and coded the actions that the subjects did during the tasks. The actions that we are interested in are:

- Browsing the information source: Browsing the unit tests in Eclipse's package explorer or the list of classes/packages in the Javadoc documentation.
- Analyzing the information source: Reading the body of the methods in a unit test or the class/method descriptions in the Javadoc documentation.
- Searching: Searching the unit tests using Eclipse's search facility or searching the Javadoc documentation using the browser search.
- Auto-complete: Using the Eclipse's auto-complete feature to find a relevant method.
- Coding, running the program, debugging, and thinking.

The first four actions can result in success or failure. If a subject found an element relevant to the task, the action is considered as a success. Otherwise, it is labeled as a failure. After coding the videos, we analyzed the codes to find interesting patterns in subject's actions. This helped us to identify different causes of failure for the tasks that were not finished successfully. We also counted the number of errors that each subject made during a task. Errors were categorized and their frequencies were recorded.

[7] http://camstudio.org/

TABLE I. BREAKDOWN OF PROGRAMMING TASKS AND HOW THEIR SUCCESS LEVEL IS MEASURED

Task	Sub-tasks	Success	Partial Success
1	1. Creating the workbook, sheet, rows, and cells; Putting values into the cells. 2. Creating the named range. 3. Saving the workbook into a file.	Two successful sub-tasks and a partially successful one	One successful sub-task and one partially successful one
2	1. Reading the workbook from the file. 2. Changing the comment author. 3. Changing the comment text. 4. Saving the workbook into a file.	Three successful sub-tasks and a partially successful one	Success in either sub-task 1 or 4 and partial success in either sub-task 2 or 3
3	1. Creating the workbook and sheet. 2. Creating conditional formatting rules for the first column. 3. Creating conditional formatting rules for the second column. 4. Saving the workbook into a file.	Three successful sub-tasks and a partially successful one	Success in either sub-task 1 or 4 and partial success in either sub-task 2 or 3
4	1. Creating the first row and merging the cells. 2. Creating the second row and setting the header values. 3. Creating the data rows and setting their styles. 4. Putting the sample data inside these rows. 5. Creating columns for weekly sums, setting the formulas, and setting the styles. 6. Creating the last row, setting the formulas, setting the styles.	Five successful sub-tasks and a partially successful one	Four successful sub-tasks, or three successful and two partially successful sub-tasks

IV. RESULTS

After analyzing our data, we observed different behavior in successful and unsuccessful subjects towards the information sources they had been given. We derived some hypotheses that describe those observations and the (design) implications regarding the hypotheses. In the following, we will provide evidence from the study that backs up our observations and hypotheses.

Observation (Sophisticated Scenarios): *All the JD group members failed in doing Task 3, while more than half of the UT group members were partially successful.*

Fig. 1 shows the task success results for the two groups of subjects. There were 9 subjects in the UT group and 7 subjects in the JD group. The most interesting thing in comparing two charts is that everybody in the JD group failed Task 3. While, 5 subjects in the UT group managed to partially complete the task. The task was the most challenging one which needed multiple classes and methods and was not addressed in the initial tutorial document.

To define conditional formatting rules, several classes like `HSSFSheetConditionalFormatting`, `CellRangeAddress`, and `HSSFConditionalFormattingRule` must be used. Thus, the basic scenario of creating conditional formatting rules is a complex scenario. Subjects probably needed examples to understand what to do. Nobody in the JD group was able to accomplish the task even partially. After analyzing the unsuccessful subjects' actions during the task, we noticed that only two subjects in the JD group managed to find some of the mentioned classes. The standard Javadoc documentation of the API does not contain many examples, but it provides a short example on how to define conditional formatting rules. The first subject used the browser search and found the `HSSFConditionalFormattingRule` class. Then, he scrolled down the class list in the same package and found some other classes. It took him a long time to reach the example and even though he came across the example he did not manage to produce anything meaningful. The second subject found class `HSSFConditionalFormatting` by browsing classes in the Javadoc documentation and scrolling classes whose names started with prefix HSSF. He copied the example but was not able to do the task, because the example was too brief.

The partially successful members of the UT group managed to find the unit test class `TestHSSFConditionalFormatting` that provided them with multiple related examples. Four subjects managed to find the test by using Eclipse's package explorer to browse the classes in the `hssf.usermodel` package[8] that contains most of the high level concrete classes. One subject used the Eclipse's search facility and found the same unit test searching for the 'format' keyword. Three subjects modified code, copy & pasted from the unit test, in order to do the task. In other words, they took care of the task by transforming it from a development task to a change task.

Hypothesis 1: *Examples are crucial in tasks involving complex API usage scenarios. A task of such nature could not be done easily if the only information source available to the developer is the standard Javadoc documentation.*

More evidence for this hypothesis comes from the results of the second task. Although subjects in the JD group were not provided with any examples for this task, only two of them failed. The rest of the JD group members managed to do the task using the standard documentation and/or Eclipse's auto-complete feature. The difference between this task and Task 3 lies in the fact that Task 2 has a more straightforward scenario that could be done without the help of any examples. It did not involve any hidden rules that developers needed to find out about (coordination barrier). The developers only needed to have an accurate image of the relationships between classes in the API, i.e. a correct mental model of classes and their relationships. They could have gained this insight from the tutorial documentation and doing Task 1. Therefore, examples can be helpful in rather simple scenarios, but are not crucial.

[8] All the package names start with `org.apache.poi` which we do not show in this paper for the sake of simplicity.

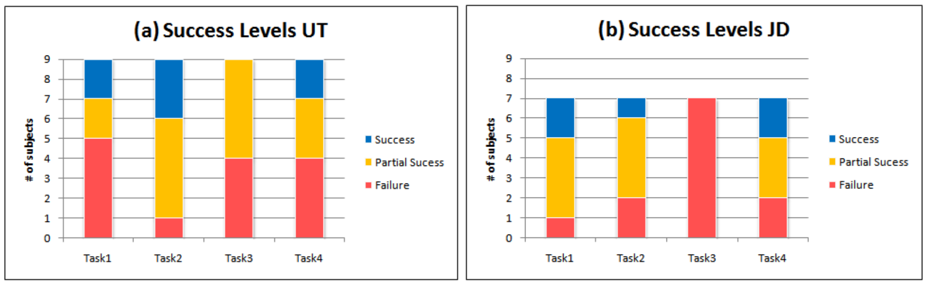

Figure 1. *The success levels of tasks (a) the UT group (b) the JD group. The UT group perfromed better in completing tasks 2 and 3.*

Fig. 1 also shows the UT group members failed tasks 1 and 4 more than the JD group members did, but it does not contradict our hypothesis. Based on our observation, almost all the subjects in both groups only relied on the tutorial document as the source of information during tasks 1 and 4. Therefore, the success rate of the group members is not related to having access to unit tests or Javadoc documentation.

Observation (Scenario Coverage Issues): *Nobody managed to finish Task 3 successfully, even though some of the UT group members spotted related examples in the unit tests.*

There are two ways to define a formula or conditional formatting. The formula can be defined as a string, or it can be defined using what is called a `ComparisonOperator`. The first example of a string formula in the unit test is the literal "7" which is not really helpful. Subjects had been trying to define string formula, but this example did not give them any clue about what a string formula is. The `TestHSSFConditionalForamtting` unit test only provides examples of the `BETWEEN` comparison operator. To do the task, subjects needed to use the `GT` (greater than) comparison operator.

Things become more complex here because the method that creates a conditional formatting formula takes three arguments: a comparison operator followed by two string arguments. However, the third argument is only used for ternary comparison operators like `BETWEEN`. All the subjects who found the examples in the unit tests struggled with this method. They were trying to figure out what to supply as the third argument. The unit test does not have any example that uses a binary comparison operator like `GT`. Eventually, the subjects ran out of time trying to figure out how to resolve this problem. They could probably have found the solution, if they had been given more time. However, if the unit tests had covered this scenario, it would have saved the developers' time. Putting it another way, the unit test does not cover all the common usage scenarios of creating conditional formatting rules, so it is not a very good example provider.

Hypothesis 2: *Good examples should cover all the common usage scenarios of the API.*

APIs must be designed starting from a set of usage scenarios and code samples implementing these scenarios [4]. Due to inherent complexities in the design of an API, it might not make the developers' life easier. Take the conditional formatting scenario of the POI API as an example. However, if supplemented with helpful examples it can better guide the users how to use the API.

Implication: *Unit test developers should cover all the basic scenarios.*

The selected API comes with packages that contain classes that are meant to provide API usage examples. The subjects did not have access to these examples during the experiment, because we wanted to examine the usefulness of unit tests as example providers. We examined those classes and found out that they usually provide examples similar to the ones that can be found in the unit tests. Therefore, if unit tests were treated as a source of API usage examples, developing separate example classes would be redundant. If unit test developers consider the unit tests as a source of examples, then they should also provide tests for common scenarios that API users would want to implement. This will require a change in the way API unit tests are developed and ultimately makes them better documentation artifacts.

Observation: *The subjects had difficulty finding relevant examples. Browsing or searching the unit test code repository did not always result in helpful examples.*

Subjects were provided with hints where to look in order to find the relevant unit tests. Even so, finding the right unit tests was not always straightforward. For Task 2, three subjects speculated that the name of the unit test would be `TestComment`, while the name of the actual unit test was `TestHSSFComment`. Another subject used Eclipse's search facility to find unit tests with the 'Comment' keyword. In that case, he did not pay attention to the package name of the results. So he wasted time looking at irrelevant unit tests. Examining the unit tests also revealed that some of them are very long and contain multiple test methods. Going through them made finding examples a tedious and time consuming task for subjects. One subject just examined the first method that looked promising, which was not the best example in this case. Other subjects just gave up upon encountering such a lengthy unit test class.

Hypothesis 3: *The standard mechanisms provided by the IDE are not enough for finding useful API usage examples.*

In our experiment, the subjects were provided with the information about the relevant package names, and naming conventions of the unit test classes. Not having such information, finding a relevant unit test by either browsing the list of unit tests or searching them would have been a very time-consuming and tedious task.

Table 2 provides a partial summary of our questionnaire results. We used a combination of semantic differential statements and Likert scales on a 7-point scale [25]. This table provides the frequency of responses to these three questions. The frequency of the responses to the third question indicates that the UT group members believed that finding good examples was neither too difficult nor too easy. This backs up our observation that a few subjects had difficulty finding relevant unit tests. The UT group response to this question was better than what we observed, because some of them found irrelevant examples that they thought are the right ones. Interestingly, the JD group members replied that finding relevant info in the Javadoc documentation was not very easy. Both groups agreed that the supporting material was somewhat helpful.

Implication: *Tool support should be provided to automate the extraction of examples from the unit tests.*

When developers have a scenario in their mind to implement, they need examples that are similar to their scenarios. The IDE facilities to find classes and methods are not adequate for scenario-based search.

Observation (Alleviating API Usability Issues): *Subjects who used examples extracted from the unit tests faced fewer problems arising from the design of the API.*

One of the very commonly sighted problems, especially in Task 1, was importing wrong required API classes. The POI API version 3.5 has classes with the same name in different packages. There is a group of interfaces whose names do not have any prefixes. All concrete classes in `hssf.usermodel` and `xssf.usermodel`, like `HSSFWorkbook` and `XSSFWorkbook`, implement those interfaces, e.g. the `Workbook` interface. The name conflict has been generated as a result of choosing names for these interfaces that duplicate the names of already existing classes. Therefore, the same name now may stand for a high-level interface and some low-level concrete class. For example `ss.usermodel.Workbook` is a high-level interface, while `hssf.model.Workbook` is a concrete class that is not related to the former. It is legal to have classes with similar names in different packages, but they can be the source of confusion.

During Task 1, subjects heavily relied on the tutorial document. The examples in that document use the aforementioned interfaces, but the examples did not contain the required import statements. As a result, 11 out of 16 subjects had one or more wrong imports. They used the quick-fix suggestions provided by Eclipse, and just selected the first choice that happened to be the wrong class. They spent a lot of time figuring out what was wrong. Subjects that copy-and-pasted code from the unit tests for tasks 2 and 3 did not make this type of error, because Eclipse generates the required import statements automatically.

Hypothesis 4: *Good examples can alleviate the problems arising from less usable API designs.*

We had one subject in the JD group that imported the wrong low level class `Sheet` instead of the high level interface. He did not have access to any examples, so he used Eclipse's auto-complete feature to browse the list of the class's methods. He went further on using more and more low-level classes, and finally failed to do the task. The problem arises from the evolution of the API. The changes that have been made to the API have deteriorated its usability. A less usable API can cause more errors. Addressing issues like this by changing the design of the API may not be possible for reasons like backward compatibility. However, providing good examples can alleviate the problem and help the developers avoid falling in such pitfalls.

Implication: *The high-level parts of the API that are to be used by the API users should be properly separated from the rest of the API.*

This would require the API to be properly modularized in the first place. However, Java packaging is not enough for this purpose, because all the public classes are made visible to everyone. Each API must export its public interfaces, and possibly some low-level interfaces for advanced users, but it should hide its internal parts. The standard documentation could then follow the modularization of the API. The documentation for the internal and low-level parts of the API could be separated and made hidden by default.

Observation (Wrong Mental Models): *Unsuccessful subjects had a wrong understanding of the design of the API, or had wrong assumptions about the responsibility of certain classes and methods.*

More than half of the subjects (4 developers in the UT group and 5 developers in the JD group) failed in doing Task 3, because they formed a wrong model of the API in their minds. In the task description, there are references to keywords 'background color' and 'formula'. The unsuccessful subjects remembered from the tutorial document that cell styles can be used to change the appearance of cells. They tried to combine cell styles and cell formulas to accomplish the task. The JD members just went down this wrong path by looking at the Javadoc documentation of the `CellStyle` interface or browsing its methods using Eclipse's auto-complete. They did not manage to find anything to help them correct their mental model. Likewise, two subjects in the UT group had the same wrong mental model. After looking at some unit tests that contained examples of using cell styles, they finally realized that cell styles are not conditional. Eventually, they tried to find something 'conditional' and one of them managed to find the right unit test and became partially successful.

Hypothesis 5: *Finding relevant examples can help developers correct their wrong perception of the API. Therefore, prevents them from committing numerous errors due to their flawed mental model.*

If developers have completely flawed mental models, nothing can prevent them from committing numerous errors. Having a good source of examples might help them to backtrack and correct their mental model faster. Subjects in the JD group had difficulty understanding the relationship between high-level classes and their responsibilities. For example, they speculated that a method in the `Sheet` interface should provide a `Cell` object, and wasted a lot of time looking at the documentation. This method is actually in the `Row` interface. In another case a subject thought he had to set the comment back after he had changed its author. Access to the basic examples of the API could have helped them quickly recover from these misunderstandings.

TABLE II. QUESTIONNAIRE RESULTS ON THE USEFULNESS OF THE PROVIDED INFORMATION SOURCES. THE JD GROUP THOUGHT THAT USING DOCUMENTATION AND FINDING RELEVANT INFO HAD NOT BEEN EASY.

Question	Meaning of the scale	UT group responses							JD group responses							
		7	6	5	4	3	2	1	7	6	5	4	3	2	1	Scale Level
1. The initial tutorial materials helped me complete the tasks	1: Strongly disagree 7: Strongly agree	0	0	2	1	5	1	0	1	2	3	1	0	0	0	Frequency
2. The unit tests/Javadoc documentation helped me complete the tasks	1: Strongly disagree 7: Strongly agree	0	1	5	1	1	0	1	0	1	2	2	2	0	0	
3. Finding relevant classes and methods using the unit tests/Javadoc documentation to complete a task was	1: Difficult 7: Easy	0	1	4	2	1	1	0	0	0	1	1	4	1	0	

Implication: *Adding examples to the API standard documentation would help developers shape correct mental models of the API.*

Two subjects complained about the unit tests. They thought that the unit tests lacked comments and therefore the examples were difficult to comprehend. Being able to see examples and documentation together would be a solution to this problem. Examples can help developers partly overcome selection and coordination barriers. However, developers also need documentation at the same time to overcome use and coordination barriers.

V. EXPERIMENT VALIDITY

There are some factors that can affect the validity of our study. We assumed that the subjects were not familiar with the API. To ensure this, we asked them to fill out a form at the end of the experiment. Only two subjects had previous exposure to subsets of the API, but none of the two were familiar with the subset of the API being used in our study. We believe our subject makeup is representative of the developer community. Even though most of the subjects were students, some of them had previous industrial experience. At least they had one year of Java programming experience, which was enough for the tasks of this study. We required participants to be familiar with unit testing. The UT group members had practiced unit testing for at least one month, while the average was 22.6 months. Therefore, subjects' unfamiliarity with unit testing did not affect our experiment.

We defined tasks that embodied typical usage scenarios of the API. The tasks were defined with different difficulty levels. The tasks were defined based on the information extracted from the API web site and user documentation. By basing our experiment on these resources, we ensured that our development tasks closely parallel that of common scenarios of using the API. For each task, the subjects were given a fixed amount of time that makes it a bit unrealistic. Multiple studies show that program creation time is different on the order of 10 to 1 for different developers [16]. Therefore, we tried to compensate for this limitation by the way we evaluated the task results, which we will discuss later. We did not provide the subjects with access to the Internet. This helped us to make sure that the main source of information during the tasks, at least for tasks 2 and 3, is the controlled resources we provided to the developers; that is the unit tests for the UT group and the Javadoc documentation for the JD group. The POI API is a general-purpose API and developers can find information about it on the Internet. We wanted to investigate the effects of not having access to such information. The Internet comes handy in every day development tasks. However, the Internet access would not have an effect in the case of internal APIs where supplementary resources are not available publicly to the developers.

We defined a three-valued success indicator for our tasks. The success values of the tasks were a main source for our hypotheses and implications. To decrease the effect of the time limits of the tasks, we defined a task to be partially successful, if the subject had found the critical classes and methods that must be used to finish the task. We watched and transcribed the captured videos of the experiments, and carefully examined the final programs. For each task the list of the critical classes and methods are known, and we assign the success level of the task based on this list. In other words, a partially successful subject was a developer who is on the right track and could have finished the task if they had had more time. On the other hand, a failed subject could not have finished the task, even if we would have given them a reasonable longer period of time.

The other concern would be the selection of the API and its effect on the results. We chose a general-purpose API that has a decent amount of documentation and comes with a collection of unit tests that test almost every part of the API. However, not all APIs come with such a complete suite of unit tests. Hence, the question "Can our results be generalized to that of the APIs without good unit test coverage, or without any unit tests?" Our findings showed that if an API has good unit tests, they can be used as the source of examples. The learnability of the API can be increased if the API developers supplement the API with a good coverage of unit tests. The existence of APIs without such coverage does not contradict our results.

VI. RELATED WORK

Usability of APIs is a research topic that recently has received more attention in both industry and academia. A few researchers have provided API design guidelines based on API designers' experience [2, 4, 24]. While other studies aimed at finding ways to improve the usability of APIs. Better design options to improve API usability have been proposed [7, 22, 26]. Other studies have provided techniques to assess the usability of a specific API and how to improve it [3]. However, these studies usually provide suggestions and practices that have to be employed during the API design phase. If we have an API that has been around for quite a while, it is difficult to change its design without breaking systems that are built upon it.

If the design of an API cannot be touched, there are other ways to help API users overcome the difficulty of using the API. Some studies tried to make APIs more useful by changing how the documentation is presented to the users. One approach is to find the most commonly used classes and highlight them. At the same time, limited usage examples can be extracted from code search engine results [23]. Another study tried to help API users by providing a tool that helps them find relevant elements of the API. Suggestions are provided based on the association relationships between the elements [6]. In another study, researchers built a tool to extract JDK API usage examples from the results of a code search engine and added them to the standard Javadoc documentation [12]. All these techniques use the publicly available resources for the general-purpose and widely used APIs. There are numerous examples available for such APIs. Users of internal and closed source APIs do not have access to such example resources, but our findings show that they can still be benefited from examples provided by the API unit tests.

Robillard [19] investigated problems that developers in Microsoft had in using APIs. He found that the API documentation must include good examples, be complete, and support many complex usage scenarios. Our study provides more evidence for the findings of Robillard’s study.

Gaelli et al. proposed an approach to organizing unit tests [9]. They defined a meta-model for unit tests to enable test developers to make explicit links between tests and units under test. This makes it easier to find a test, if the developer knows the name of the method under test. Hence, finding method examples is now much easier, since unlike normal xUnit tests, there are direct links between methods and their unit tests. Our study was different, because we wanted to know whether unit tests, developed using traditional xUnit frameworks, are helpful as example providers. Besides finding relevant unit tests, the tests themselves might be difficult to use. Research in the past has addressed the problem of long unit tests. Meszaros proposes several best practices and refactoring ideas to develop better unit tests [17]. Another solution is to use xUnit extensions like JExample to produce more concise test methods [14]. We observed that the POI API has some lengthy unit tests that are difficult to read and understand. Using the aforementioned techniques, it is possible to make tests more concise and readable, i.e. make them better examples.

Examples can be used for different purposes: to add new functionality, to show how to use a new environment or library, and to fix syntax errors. Barzilay et al. defined example embedding (EE) as the use of already written code in the development of new code. They claimed that using EE will liberate developers from the belief that they need to reuse code as it is. They believe that programmers should write examples for using the code that they develop [1]. Here, we have shown that unit tests can be used as examples. Therefore, there is no need for developing examples separately, if developers develop a comprehensive set of examples in unit tests.

Several studies have investigated ways for finding code examples. Holmes et al. [10] argue that the process of reusing code from examples would follow the general three phases of source code reuse: location, selection, and integration. Without proper tool support for interacting with source code examples, reusing examples would be a tedious task. This would decrease a developer's willingness to find the best example that matches a desired scenario. We found developers tend to give up on finding examples upon encountering long unit tests. Therefore, automating the extraction of examples from unit tests seems to be the natural continuation of our approach.

Strathcona [11] is a tool that mines a local code repository to find code examples based on the current development task. It would be useful if such a code repository exists, and furthermore the developer has a basic idea of what they want to do. If the developer does not have any clue about the API they want to use, this tool cannot find any related code snippets for them. A tool like Strathcona can be utilized for extracting code snippets similar to the current development context from the API unit tests. Unit tests can be a valuable asset if made part of the repository of tools like Strathcona or public code search engines.

Dagenais and Ossher introduced a tool that locates Eclipse extension examples. It relies on the structured documentation provided by developers which defines the framework creation steps. It uses this structured guide to mine examples for each step of the extension creation [5]. This approach is useful if the steps of a usage scenario are predetermined by developers who are using the API. If such a guide does not exist, the tool is not able to find useful examples.

VII. CONCLUSIONS

To develop and maintain software systems, developers need to learn how to use different APIs. Learning a new API can be an intimidating task, especially if the API is very large and complicated, or enough supporting material is not provided with the API. The problem can be partially resolved by applying the API design guidelines derived from the API usability research [4, 7, 22, 26]. However, it is not always possible to reduce the complexity of an API, because some large APIs are designed to provide diverse functionality. Hence, the API designers have to compromise on the simplicity and usability of the API in favor of its power. Supplemental information resources, like the standard documentation and code examples can be used to shorten the learning curve of a complex API.

In this paper, we reported on a study that we had conducted to evaluate the suitability of unit tests as a source of API examples. The experiment we performed with two groups of developers revealed that usefulness of the standard API documentation is limited, especially when it comes to complicated usage scenarios. Developers need examples to comprehend how those scenarios can be implemented with the API. We showed that well-written unit tests provide good usage examples. To be a good unit test of the public interface of an API, it must be developed with the common usage scenarios in mind. Hence, test developers must cover the common usage scenarios thoroughly in the unit tests. This can change the way API unit tests are developed and maintained. We also found that finding useful examples in the unit tests can be tricky, but it can still be more helpful than the standard documentation. The solution we hereby propose is to extract relevant examples from the unit tests and place them in line with the related entries in the standard documentation.

Our experiment was based on a general-purpose API that comes with the standard documentation. However, the internal APIs may not necessarily come with the standard documentation, or it might be either incomplete or outdated. Maintainers of the systems developed with such APIs definitely need usage examples, in order to comprehend the code for carrying out change tasks without creating too many defects. If organizations realize the usefulness of the unit tests as example repositories, they would be inclined to invest further in development processes that require developers to develop unit tests during the development phase. This would affect the maintenance costs and the learnability problems of the APIs.

In this study, we used an API, the Apache POI, whose unit tests were not written as API usage examples. Yet, they proved to be helpful in learning how to use the API. However, finding these examples is not always easy, if developers have to use the standard facilities provided by IDEs. Next, we will focus on finding different usage scenarios from unit tests and presenting them to developers in a helpful manner.

ACKNOWLEDGMENT

The authors of this paper feel indebted to Mehrdad Nurolahzade and Jonathan Sillito for the insightful and fruitful discussions and feedback throughout this research. We also wish to extend our thanks to our experiment participants and the unknown reviewers of this paper.

REFERENCES

[1] O. Barzilay, O. Hazzan, and A. Yehudai, “Characterizing Example Embedding as a software activity,” In Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE), May 16, 2009, pp.5-8.

[2] J. Bloch, “How to design a good API and why it matters,” Companion to the 21st ACM SIGPLAN Symp. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 06), ACM Press, 2006, pp. 506–507.

[3] S. Clarke, “Measuring API usability,” Dr. Dobb’s Journal, Special Windows/.NET Supplement, May 2004, pp. S6--S9.

[4] K. Cwalina and B. Abrams, Framework design guidelines: Conventions, idioms, and patterns for reusable .NET libraries, 2nd ed., Addison-Wesley, 2009.

[5] B. Dagenais and H. Ossher, “Automatically locating framework extension examples,” In Proceedings of the 16th ACM SIGSOFT international Symposium on Foundations of Software Engineering, November 09-14, 2008, Atlanta, GA, pp. 203-213.

[6] D. S. Eisenberg, J. Stylos, and B. A. Myers, “Apatite: A new interface for exploring APIs,” In Proceedings CHI 2010: Human Factors in Computing Systems, Atlanta, GA, April 10-15, 2010, pp. 1331-1334.

[7] B. Ellis, J. Stylos, and B. A. Myers, “The factory pattern in API design: A usability evaluation,” In Proceedings of the 29th International Conference on Software Engineering, May 20-26, 2007, Minneapolis, MN, pp. 302-312.

[8] F. Fioravanti, “The impact of extreme programming on maintenance,” In Advances in Software Maintenance Management: Technologies and Solutions, Hershey, PA: Idea Group Publishing, 2003, pp. 75-92.

[9] M. Gaelli, R. Wampfler, and O. Nierstrasz, “Composing tests from examples,” Journal of Object Technology, 6(9), Special Issue. TOOLS EUROPE 2007, October 2007, pp. 71-86.

[10] R. Holmes, R. Cottrell, R. J. Walker, and J. Denzinger, “The end-to-end use of source code example: An exploratory study,” In Proceedings of the 25th IEEE International Conference on Software Maintenance, September 20-26 2009, Edmonton, Canada, pp. 555-558.

[11] R. Holmes, R. J. Walker, and G. C. Murphy, “Approximate structural context matching: An approach for recommending relevant examples,” IEEE Transactions on Software Engineering, 32(12), pp. 952-970, December 2006.

[12] J. Kim, S. Lee, S. Hwang, and S. Kim, “Adding examples into Java documents,” In Proceedings of the 2009 IEEE/ACM international Conference on Automated Software Engineering, November 16-20, 2009, Auckland, New Zeland, pp. 540-544.

[13] A. J. Ko , B. A. Myers, and H. H. Aung, “Six learning barriers in end-user programming systems,” In Proceedings of the 2004 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), September 2004, pp. 199-206.

[14] A. Kuhn, et al., “JExample: Exploiting dependencies between tests to improve defect localization,” In P. Abrahamsson (Ed.), Extreme Programming and Agile Processes in Software Engineering, 9th International Conference, XP 2008, Lecture Notes in Computer Science, Springer, 2008, pp. 73-82.

[15] R. C. Martin and M. Martin. Agile principles, patterns, and practices in C#, 1st ed., Prentice Hall, July 2006.

[16] S. McConnell, Code complete: A practical handbook of software construction, 2nd ed., Microsoft Press, June 2004.

[17] G. Meszaros, xUnit test patterns: Refactoring test code, Addison-Wesley, May 2007.

[18] R. Osherove, The art of unit testing: with examples in .NET, Manning Publications, June 2009.

[19] M. P. Robillard, “What makes APIs hard to learn? Answers from developers,” IEEE Software, 26(6), pp. 27-34, November 2009.

[20] M. P. Robillard, W. Coelho, and G. C. Murphy, “How effective developers investigate source code: An exploratory study,” IEEE Transactions on Software Engineering, 30(12), pp. 889-903, December 2004.

[21] P. Runeson, “A survey of unit testing practices,” IEEE Software, 23(4), pp. 22-29, July 2006.

[22] J. Stylos and S. Clarke, “Usability implications of requiring parameters in objects' constructors,” In Proceedings of the 29th International Conference on Software Engineering, May 20-26, 2007, Minneapolis, MN, pp.529-539.

[23] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, “Improving API documentation using API usage information,” In Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), September 20-24, 2009, Corvallis, OR, pp. 119-126.

[24] J. Tulach, Practical API design: Confessions of a Java framework architect, 1st ed., Apress, July 2008.

[25] T. Tullis and B. Albert, Measuring the user experience: collecting, analyzing, and presenting usability metrics, Morgan Kaufmann, 2008.

[26] R. B. Watson, “Improving software API usability through text analysis: A case study,” In Proceedings 2009 IEEE International Professional Communication Conference, July 19-22, 2009, Waikiki, HI, pp. 1-7.

[27] A. Zaidman, B. Van Rompaey,S. Demeyer, and A. van Deursen, “Mining software repositories to study co-evolution of production & test code,” In Proceedings International Conference on Software Testing, Verification, and Validation, April 9-11, 2008, Lillehammer, pp. 220-229.