Comparing API Design Choices with Usability Studies: A Case Study and Future Directions

Jeffrey Stylos[1], Steven Clarke[2] and Brad Myers[1]

[1] Carnegie Mellon University,
5000 Forbes Ave, Pittsburgh, PA, USA
{ jsstylos, bam }@cs.cmu.edu
[2] Microsoft,
1 Microsoft Way, Redmond, WA, USA
stevencl@microsoft.com

Abstract. There are more APIs than ever, and designing APIs that are usable by their target audience is difficult. Work at Microsoft has demonstrated that running controlled usability studies with participants from different personas and analyzing the results of these studies using the cognitive dimensions framework is effective at identifying and preventing usability problems in APIs. This paper presents a generalization of that technique in the form of usability studies of common API design choices. By studying a single design choice from multiple perspectives, we can generalize our results to any API that might be affected by the design choice. We show the feasibility of our approach with an initial study on whether or not to require constructor parameters, and present our current and planned work to study more API design choices.

1 Introduction

An important and challenging programming activity is using application programming interfaces (APIs), frameworks, toolkits and libraries. Programmers implementing new functionality need to figure out which APIs to use and how to combine them [10]. Programmers reading or modifying code have to understand how existing code that calls APIs works, what assumptions the code makes, and how to change or add to the code without breaking these assumptions. These tasks are increasingly challenging with the growing number and size of APIs; there are many options of which frameworks to use and current frameworks have tens of thousands of classes and methods.

One way to help this problem is to design APIs that are more usable by their target audience. Well designed APIs can be more quickly and effectively used and can lead to fewer programming bugs. However, the usability of an API can be difficult for an API designer to predict ahead of time, and difficult to fix after releasing an API, because of compatibility concerns.

Running a usability study of a specific API before it is released, using programmers representative of the target audience and the most common programming tasks that the API is intended to support, is an effective technique for detecting non-obvious usability issues with APIs and improving the usability of the released APIs [4]. However, these studies are expensive to run on thousands of APIs, and organizations producing only a few APIs might not have the resources available to run a controlled study.

This paper presents a technique that generalizes from running usability studies on a specific API to usability studies that focus on a particular design choice that is common across many different APIs, for example how to structure an object's constructor parameters. The next section describes this technique in more detail, using as an example an initial case study we performed. We then present our plans for specific studies of other API design choices. Finally, we relate our research to existing work and offer conclusions.

2 API Design Choice Studies

2.1 Compared to Specific API Studies

To study the usability of a specific API, one can identify the common tasks that an API is designed to provide and watch programmers representative of the API's target population solve these tasks. By performing these studies in a usability lab, recording them and asking programmers to think aloud as they work, one can review and identify the specific problems encountered by different users and the assumptions and strategies that led to these problems. For example, to study the usability of a file API, participants might be asked to write code that reads a file in one task and writes a file in another task. Screen captures can record the common problems – for example, programmers of a particular persona might have a difficult time selecting or combining the objects needed to write a file. Audio recordings of programmers' spoken thoughts can then identify the reasons for these problems – they might assume that the classes have a different name, or they might expect to only have to use one object instead of combining the multiple objects that the API provides.

Usability studies of an API design choice can use similar techniques, however instead of having programmers solve tasks from the same API, they solve small tasks using different APIs that make different choices about a common design decision.

2.2 Constructor-Parameter Options

To better understand how to design object constructors, we created a study that compared two different approaches.

The first approach was to provide only object constructors that required as parameters all of the "essential" properties of that object. For example, a file object would provide a constructor that required a string containing the path of the file: "`File (String path)`", or an email message might require a sender, recipient, subject and body: "`MailMessage (MailAddress sender, MailAddress recipient, String subject, String body)`". By providing only constructors that required the necessary objects, these APIs could statically prevent certain errors. For example, a programmer could write compilable code that tried to read a file without specifying which file to read, or tried to send a message without specifying whom to send it to.

The second approach was to provide, either instead of these constructors or in addition to them, "default" constructors that required no parameters. Using these constructors, a File object could be created with a call of "`File()`" or an email message with "`MailMessage()`". To specify the essential information, such as the file-path or mail-recipient, programmers could set individual properties of an object, such as "`file.path = "foo.txt"`" or "`mailMessage.recipient = "cj@msn.com"`". We called this approach, "create-set-call," since it involved creating an empty object and setting essential properties before calling other methods (such as read or send). A seeming disadvantage of this approach is that it does not make the API's dependencies explicit at compile-time – a programmer might not know that a mail message requires a sender, or might forget to specify one.

2.3 Case Study Design

Our study design used several small programming tasks involving objects that used both of these construction patterns. We used real APIs in some tasks and artificially constructed APIs in others for the sake of factoring out experience with real domains and being able to control the number of required parameters. We also created alternate versions of real APIs that used the opposite constructor choice to directly compare the two constructor options on the same objects.

We were interested in not just the activity of initial code creation but also the activities of reading code and modifying and debugging existing code. For example, we wanted to find out if setting individual properties, where the property name and value are shown together, was more readable than constructor calls, where only the values are shown. Different tasks in our study looked at code creation and modification.

To factor out the influence of the developing environment, we also designed tasks outside of an IDE. For example, we had programmers write the code they would expect to write to solve a task using only the Notepad text editor. These tasks helped capture programmers' initial assumptions and expectations.

2.4 Case Study Results

We found the tasks and techniques we used in this study were effective at showing common effects of the constructor design choices on users of different personas. In particular, we found that users of some personas *never* expected constructors to require parameters, and were less successful using these constructors, even after being exposed to several different APIs that used this pattern. We also found that even when programmers did not have strong expectations against constructor parameters, they preferred and were more successful with APIs that did not require them, preferring to use the "create-set-call" pattern of object construction instead. One reason for this was that required constructor parameters forced programmers to instantiate each of the parameter objects before they felt they could explore the constructed object using design-time tools like code completion. This caused a problem of *premature commitment* [8] in the APIs that used required constructor parameters.

We were surprised to find that the “create-set-call” pattern, which seemed to allow more errors at compile-time, was actually more effective at helping guide programmers toward effective use of the APIs, and were encouraged by the effectiveness of running usability studies on API design decisions.

3 Planned Work

Based on our experience with studying constructor-parameter API design choices, we propose to apply similar methodology to study other API design choices. This section presents several specific design patterns for future study.

3.1 The Factory and Builder Construction Patterns

Another construction-related design choice is whether or when to use factories to generate objects, instead of constructors [7]. These can take several forms in object oriented languages, including factory classes and static construction methods on the target class. Our experience with studying construction parameters suggests that use of factories is likely to have discoverability problems for some personas in many if not all situations. While we also have anecdotal evidence that supports this for specific cases, we still do not understand the entire picture.

For example, we found that even when programmers’ mental models of an object included required properties, they did not extend this to their expectations for the constructor. However, if some programmers’ mental models include the idea of a factory, it is possible that, unlike the constructor parameters case, these expectations would change how they would try to construct objects.

Our goals will be to discover in what cases programmers of different personas expect factories (and when this is due to specific programming experience as opposed to assumptions about how objects should interact), and what the impact is when their expectations are not met (by either the presence or absence of a factory).

3.2 Compile-time versus Runtime Error Messages

A more general design choice, for APIs as well as developing environment and language design, is the decision of when to provide feedback to the programmer about incorrect API usage. While the amount of error-checking an API can do at compile-time is generally limited to what can be encoded in a static type-checker, it is often possible to postpone errors and display them as runtime exceptions. For example, a method that requires a certain object could either encode that requirement using a required parameter to the object (by placing the method on the required object itself) or it could use a property and throw a runtime exception when it is invoked before the property has been set.

Convention wisdom suggests that catching more errors earlier is usually better. However we were surprised to find in our study of constructor parameters that programmers were usually more successful using objects whose incorrect use was allowed but always caused a runtime exception than objects that statically enforced correct usage. One reason for this seemed to be that when programmers encountered compile-time problems – because they omitted necessary objects in a constructor or method call – they tended to assume that the problem was syntactic and try to fix it by the insertion or deletion of parentheses or the "`new`" keyword. They would tinker with what was usually already correct syntax, often introducing new problems before discovering the true source of the problem. Indeed, their assumption that the problems were syntactic was so strong that many programmers would not fully read the error message, even when explicitly prompted to by the experimenter. On the other hand, these same programmers would tend to read and understand the runtime exceptions relating to the same problems, and were faster and more successful at fixing these problems using these exceptions.

One reason for programmers' assumptions might be the specific location of the messages in the developing environment – programmers might be trained to think of messages on the bottom of the screen with a corresponding wavy-red-underline as being related to syntax. Another possible factor is the timing of the messages, at compile-time while there in the middle of typing an expression versus when they are stepping through the logic of their code. Still another possibility is that the specific wording makes a large difference, although in our initial observations many programmers tended to read only a few words if any of most error messages.

While one argument against delaying messages until compile-time is that they might not appear during testing, causing an unexpected failure later, large classes of errors can be guaranteed to occur at runtime. All of the exceptions we observed had this property.

A study that looked specifically at how programmers read and respond to different types of error and exception messages could test these hypotheses, identify the objects and situations for which programmers respond differently, and use these to make design recommendations.

3.3 Object Decomposition

A fundamental API design decision in an object oriented language is the question of how to decompose functionality into different objects. For example, there might be a "mail-client" and "mail-server" object, or this functionality might be combined in a single "mail" object. Whether or not to use the factory design pattern is also a specific case of an object decomposition decision.

Our initial experience suggests that programmers can have strong, unconscious assumptions about which objects exist, and when their expectations are not matched by the objects provided by an API, then programmers are slow to recognize or question these assumptions. Instead then tend to first question their assumptions about what the object is called, where it would be located, or whether the API is even capable of the desired functionality. For example, users would spend time analyzing all of the methods of a `MailMessage` class multiple times looking for a `Send()` method before considering that there might be a `MailServer` class.

Because mismatch between the programmers' expectations about what objects exist and what the API provides is so strong, we would like to better understand what influences programmers' expectations and how APIs and tools can mitigate the problems that occur when the expectations are not met. For example, the code completion in the IDE did *not* help in the mail example because it showed only the methods provided by the existing `MailMessage` class and provided no easy way to discover the existence of the `MailServer` class.

3.4 Asynchronous Functionality

A specific question in the creation of some APIs is how to expose asynchronous functionality; for example, a file read operation that will run simultaneously with other computation. An API might provide no direct support for asynchronous file reading, requiring the programmer to explicitly create a separate thread and manage its synchronization. Or, an API might provide a file read method that accepted a callback that gets called when the read completes or aborts. Similarly, there might be event system that allowed multiple objects to register to be automatically notified about different file events.

These and other patterns can expose the same type of functionality in different ways. Some of the advantages and disadvantages of the different approaches are initially apparent. However, a comparative lab study could help us better understand the tradeoffs and the types of assumptions programmers make for different types of objects.

3.5 Usability Study Challenges

One challenge in following our research at Microsoft with the design and execution of the above studies in our research at Carnegie Mellon is the difference in available programmers. In particular, it is much harder to find experienced professional programmers and easier to find undergrad and graduate students of varying levels of expertise. An important question is whether these students represent each of the programming personas that we would like to study, and the extent to which results for students of a particular persona can generalize to other programmers of that persona.

Because the strategies and behaviors of programmers we have observed correlate much more strongly with persona than experience, we are optimistic that the relative inexperience of the student population will not be a large factor. We are unsure as to how well represented some personas – in particular the systematic persona – will be represented in this population however.

4 Related Work

4.1 Usability Studies of Individual APIs

Our work follows most directly the usability studies of specific APIs at Microsoft [4]. We were able to borrow and adapt techniques from these lab studies and use of similar evaluation techniques.

The creators of the Alice programming environment found that studying how programmers used their graphics APIs allowed them to greatly improve their usability [5]. The encountered usability problems highlighted pertain to naming – fixing mismatches between the vocabulary of the system and the user by using words like "up, move and speed" instead of "z, translate and rate."

In addition to studying individual APIs, there has been research looking at the usability of APIs developed and consumed internally within a company [11]. When developing APIs for external use, there is a broader audience and there are different tradeoffs.

4.2 Eliciting Programmers' Assumptions and Strategies

Some of our techniques are influenced by other research into the psychology of programming. For example, John Pane's formative studies when developing a new programming language and environment [13][15] used techniques like "paper programming" to capture how programmers (and non-programmers) formulated algorithms and expressed Boolean constraints [14].

We adapt a similar technique to see what assumptions programmers make about imagined APIs and how they naturally express solutions using them by having them write code in an informal (non-programming tool) environment, though in our case our programmers are mostly constrained to an existing language.

Our overall strategies of understanding programmers' behavior and using empirical studies to test specific hypothesis is influenced by the human-centered techniques in the Natural Programming Project [12].

4.3 Cognitive Dimensions Framework

During and after our study, we used the Cognitive Dimensions Framework [8] to interpret the results. This framework was designed to evaluate programming language environment usability, but has previously been adapted to evaluate API usability as well [3].

The framework allows for the classification of different problems users encounter using a particular API, to be able to better communicate the problems to others and to better understand the problems through the process of classification. For example, a user spending a long time browsing through documentation might at first seem to indicate a problem with the documentation; however looking at the user's goals and behavior might reveal that the problem arouse because they could not find the method name they were looking for – a *role-expressiveness* problem – and so a more appropriate fix would be to change the class or method name.

Others have proposed a reduced set of dimensions for API usability [2]. While we feel there is potential value to be gained from simplifying and clarifying the dimensions for this particular domain, we felt that these particular reduced set of dimensions were too simplistic for our purposes.

4.4 Framework Design Guidelines

There is a small but growing field of literature that advises those faced with the challenge of designing an API, framework, library or toolkit [6][9][1]. While many of these guidelines are insightful, sometimes the common assumptions which design are wrong, or there are conflicting opinions.

Performing studies to see empirically how representative programmers actually work lets us validate or challenge these guidelines, as well as refine and generalize them and build a greater understanding of why they are useful based on models of programmers' behavior.

5 Conclusions

We provide a generalization of the approach of studying the usability of specific APIs to more abstract studies of design choices that are shared across APIs. Our experience studying the usability effects of construction API design choices suggests that this technique is a valuable tool for informing the design of many new APIs and gaining a broader understanding of how programmers consume APIs. We present several specific API designs for future study that this technique could help illuminate.

6 Acknowledgements

This work was started at Microsoft in the Visual Studio User Experience Group and was aided by the guidance of Monty Hammontree.

It has continued at Carnegie Mellon University and is partially supported by NSF grant IIS-0329090 and the EUSES Consortium via NSF grant ITR- 0325273. Opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the NSF.

Andrew Ko, Christopher Scaffidi and Michael Coblenz contributed helpful ideas and feedback.

References

1. Bloch, J.: Effective Java Programming Language Guide. Mountain View, CA, Sun Microsystems (2001)
2. Bore, C. and S. Bore: Profiling software API usability for consume electronics. Consumer Electronics, 2005. ICCE (2005)
3. Clarke, S.: API Usability and the Cognitive Dimensions Framework, http://blogs.msdn.com/stevencl/archive/2003/10/08/57040.aspx (2003)
4. Clarke, S.: Measuring API usability. Dr. Dobbs Journal (2004) S6-S9
5. Conway, M. J.: Alice, Easy-to-Learn 3D Scripting for Novices, University of Virginia (1997)
6. Cwalina, K. and B. Abrams: Framework Design Guidelines, Addison-Wesley Professional (2005)
7. Gamma, E., R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc. (1995)
8. Green, T. R. G. and M. Petre: Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework., Journal of Visual Languages and Computing 7(2) (1996) 131-174
9. Jacques, M.: API Usability: Guidelines to improve your code ease of use. The Code Project. http://www.codeproject.com/gen/design/APIUsabilityArticle.asp (2004)
10. Ko, A. J., B. A. Myers, H. H. Aung: Six Learning Barriers in End-User Programming Systems. Visual Languages and Human Centric Computing, 2004 IEEE Symposium on. (2004)
11. McLellan, S. G., A. W. Roesler, et al.: "Building More Usable APIs." Software, IEEE 15(3) (1998) 78-86
12. Myers, B. A., J. F. Pane, A. J. Ko.: "Natural programming languages and environments." Commun. ACM 47(9) (2004) 47-52
13. Pane, J. F.: Designing a programming system for children with a focus on usability. CHI '98: CHI 98 conference summary on Human factors in computing systems, Los Angeles, California, United States, ACM Press (1998)
14. Pane, J. F. and B. A. Myers: Improving user performance on Boolean queries. CHI '00: CHI '00 extended abstracts on Human factors in computing systems, The Hague, The Netherlands, ACM Press (2000)
15. Pane, J. F. and B. A. Myers: The impact of human-centered features on the usability of a programming system for children. CHI '02: CHI '02 extended abstracts on Human factors in computing systems, Minneapolis, Minnesota, USA, ACM Press (2002)