Usability Implications of Requiring Parameters in Objects' Constructors

Jeffrey Stylos
Carnegie Mellon University
jsstylos@cs.cmu.edu

Steven Clarke
Microsoft
stevencl@microsoft.com

Abstract

The usability of APIs is increasingly important to programmer productivity. Based on experience with usability studies of specific APIs, techniques were explored for studying the usability of design choices common to many APIs. A comparative study was performed to assess how professional programmers use APIs with required parameters in objects' constructors as opposed to parameterless "default" constructors. It was hypothesized that required parameters would create more usable and self-documenting APIs by guiding programmers toward the correct use of objects and preventing errors. However, in the study, it was found that, contrary to expectations, programmers strongly preferred and were more effective with APIs that did not *require constructor parameters. Participants' behavior was analyzed using the cognitive dimensions framework, and revealing that required constructor parameters interfere with common learning strategies, causing undesirable premature commitment.*

1. Introduction

Microsoft has created and supported many different application programming interfaces (APIs) that are in wide use today. The .NET framework APIs alone include more 140,000 methods and property fields and are shared by a collection of programming languages including C#, VB.NET and C++. To increase the usability of these and future APIs, and to improve the productivity of the programmers who use them, the Visual Studio User Experience group conducts user studies of APIs before they are finalized. These studies involve representative programmers of an API's target audience performing typical programming tasks using pre-release versions of an API, and the studies have proven effective at increasing these APIs' usability [4].

However, because Microsoft creates so many APIs, it is not always possible to run user studies on each API designed. Smaller organizations may not have the resources to run user studies of their APIs at all. This paper describes a case study of a new type of API usability study that instead of testing the usability of a *specific API*, tests the usability of a *design choice* that is common to many APIs. By better understanding API design choices, we can better inform the design of all APIs. This study was performed as part of an internship at Microsoft in the fall of 2005 and was designed to inform the .NET framework developers, who develop new APIs.

```
var foo = new FooClass();
foo.Bar = barValue;
foo.Use();
```

Default constructor ("create-set-call")

vs

```
var foo = new FooClass(barValue);
foo.Use();
```

Required constructor

Figure 1. The "create-set-call" construction pattern and required constructor parameters. We compared the usability of these two construction options.

In object-oriented languages like the .NET languages and Java, one of the most common API design choices involves what object constructors to provide. These constructors are also some of the most commonly encountered parts of an API by programmers, who have to figure out whether or how to construct each object before they using that object. There are two common design choices: provide only constructors that require certain objects (a "required constructor"). This option has the benefit of enforcing certain invariants at the expense of flexibility. An alternative design, "create-set-call," allows objects to be created and then initialized. Examples of these two are shown in Figure 1.

To get a fuller understanding of when to use each in API design, we compared the two approaches in a user study of thirty professional programmers from three distinct programming *persona* [8]. Personas are an archetypical representation of users, and we used three personas that had been developed to help target APIs to different sets of programmers [4]. The programmers

performed several tasks; some these tasks involved APIs with required constructors and other tasks used create-set-call APIs. Some of the tasks involved code creation, and others involved debugging or reading of code.

We found that APIs that used the create-set-call pattern (not requiring any constructor parameters) were preferred and less problematic for all three programming personas. The reasons for this differed for each persona. *Opportunistic* programmers are more concerned with productivity than control or understanding. For these programmers objects that required constructor parameters were unfamiliar and unexpected, and even after repeated exposure these programmers had difficulty with these objects. *Pragmatic* programmers balance productivity with control and understanding. These programmers also did not expect objects with required constructors, and while pragmatic programmers were more effective than opportunistic programmers at using these objects, the objects still provided a minor stumbling block and these programmers preferred the flexibility offered by objects that used the create-set-call pattern. *Systematic* programmers program defensively and these are the programmers for whom low-level APIs are targeted. These programmers were effective at using all of the objects; however, they preferred create-set-call because of the finer granularity of control it offered by allowing objects to be initialized one piece at a time.

The rest of the paper describes the study and related research in more detail. Section 2 summarizes related work done by ourselves and others. Section 3 describes the specifics of the study design. Sections 4 presents the results, and Section 5 offers related discussion. Section 6 presents a model of programmer behavior derived from these results. Section 7 discusses limitations of this research. Section 8 discusses directions for future research and Section 9 offers some final conclusions.

2. Related Work

2.1. API Usability Studies

Our study was motivated by the usability studies of specific APIs done at Microsoft [4] and elsewhere [9][2]. These studies involve studying a particular API – a mail API, for example – then having programmers representative of the intended audience for the API perform tasks intended to be common for the API – sending an email message and connecting to a POP server, for example.

The primary difference between these studies and ours is that while the prior studies focus on finding the usability issues of a specific API, ours looked at the usability issues of a more abstract pattern that occurs in many different APIs. Our construction of study tasks also differs from these previous studies. Studies of specific APIs can use common tasks that the API is intended to perform, while our study used tasks from a variety of real and artificial APIs so that our results would generalize to many types of APIs.

2.2. Cognitive Dimensions

The Cognitive Dimensions framework was designed to describe usability problems in programming environments [7] and has been adapted to describe API usability problems [3]. The dimensions help differentiate symptoms of usability issues from the root problems. We used the API adaptation of this framework to help analyze and understand the results of our study.

2.3. Framework Design Guidelines

Based on years of .NET library development, experienced API designers at Microsoft [5] and elsewhere [1] have begun to create a set of guidelines on how to create usable APIs. Our research approach aims to validate, explain and add to these guidelines by providing direct comparisons between specific API design choices in a controlled lab environment.

The framework design guidelines recommend the use of the create-set-call pattern for opportunistic programmers. Our study was designed to better inform this recommendation by directly comparing create-set-call to required constructors in a variety of tasks and by studying three different personas.

2.4. Design Patterns

Design patterns capture common implementation techniques used in building large software systems [6]. API design choices differ from design patterns because by relate only to the externally visible choices, while design patterns often relate to internal architecture decisions. The two overlap when design patterns *are* externally visible, such as the factory-builder pattern of object construction [6].

Our research is focused on the usability of APIs by programmers who use APIs while design pattern research has traditionally focused on how patterns can create maintainable architectures for implementers of APIs or large systems.

3. Study Method

The study involved thirty participants performing a collection programming tasks, some of which had multiple conditions. Participants were asked to think aloud.

In order to make an informed decision about when to use or avoid the create-set-call pattern, our study elicited the *expectations*, *preferences* and *effectiveness* of different users performing different types of tasks. Understanding programmers' expectations can help us create APIs that are more discoverable and less dependent on documentation.

By having programmers use APIs with different object construction patterns and by using the think-aloud technique, we elicited programmers' preferences without revealing what we were studying. Programmers' preferences let us know what types of APIs they enjoyed working with and which ones they found annoying.

By administering several different tasks and having two different conditions for some of these tasks, we compared how effective programmers are between and within participants, including wrong assumptions programmers made, problems they overcame as a result of these assumptions and the time programmers took to finish the tasks.

We were interested in discovering this information for each of the three personas described in Section 3.2 because these are the personas used by the API designers to target their APIs.

Our tasks were designed to assess code readability, debug-ability and initial writability.

3.1. Participant Personas

Personas are archetypical representations of users [8]. Well-designed personas capture the typical motivations and behaviors of different target populations. The three programmer personas we used were developed at Microsoft through observations of many Visual Studio users [4]. These personas capture different *work styles*, not experience or proficiency. We used these personas to tailor our experiments and participant recruitment criteria. By studying a number of programmers of a specific persona, we are able to get results that generalize well to other programmers of that same persona.

Systematic programmers work from the top down, attempting to understand the system as a whole before focusing on an individual component. They program defensively, making few assumptions about code or APIs and mistrusting even the guarantees an API makes, preferring to do additional testing in their own environment. They want not just to get their code working, but to understand *why* it works, what assumptions it makes and when it might fail. They are rare, and prefer languages that give them the most detailed control such as C++, C and assembly.

Pragmatic programmers are less defensive and learn as they go, starting working from the bottom up with a specific task. However when this approach fails they revert to the top-down approach used by systematic programmers. Pragmatic programmers are willing to trade off control for simplicity but prefer to be aware of and in control of this trade off. For example, pragmatic programmers often use tools such as graphical layout editors but prefer to be able to edit the automatically generated code in case they need additional control later. Pragmatic programmers use languages that offer elements of both control and simplicity such as Java and C#.

Opportunistic programmers work from the bottom up on their current task and do not want to worry about the low-level details. They want to get their code working and quickly as possible without having to understand any more of the underlying APIs than they have to. They are the most common persona and prefer simple and easy to use languages that offer high levels of productivity at the expense of control, such as Visual Basic.

To recruit participants of the above personas, we used the following prescreening guidelines.

- To recruit *systematic* programmers, we prescreened for professional programmers with at least five years experience who used C or C++ as their primary programming language. We preferred programmers who typically worked on large projects with an emphasis on reliability.
- To recruit *pragmatic* programmers, we prescreened for professional programmers with at least two years professional experience who used C# as their primary language. We preferred programmers whose typical application was a desktop application using WinForms.
- To recruit *opportunistic* programmers, we prescreened for professional programmers with at least two years experience who used Visual Basic as a primary programming language. We preferred programmers without a formal computer degree and whose typical project was a web-application using HTML and Visual Basic.

We recruited participants who had registered with the Microsoft Usability Research website: http://microsoft.com/usability. In compensation for their time participants were given two vouchers each

redeemable for a software or hardware item at the Microsoft Store, such as Visual Studio, Windows XP or a Microsoft keyboard.

The actual participants we studied matched or exceeded our desired levels of experience. All of the participants were current or retired professional programmers, and none were currently students. All of the programmers had recent experience with the language in which they performed the programming task. The opportunistic programmers we studied most commonly had experience with programming the back-ends of web-based applications. The pragmatic programmers we studied most commonly had experience programming desktop applications. The systematic programmers we studied each had professional experience programming low-level devices such as embedded device drivers for laser-etching systems and work on the Linux Kernel.

During the study, participants demonstrated proficiency in the languages in which they used as part of the study, as well as the Visual Studio programming environment.

3.2. Study Environment

The studies were performed in a usability lab that separated the participant and experimenter by a one-way mirror. Participants worked on a PC running screen-capturing software, and could not see the experimenter. The experimenter could see the participant directly, as well as being able to see a copy of the participant's screen.

Participants performed their tasks using Visual Studio 2005, with the exception of one task that required the use of Notepad. Participants had access to the internet.

Participants recruited using the systematic recruitment criteria were given their tasks in C++; pragmatic programmers were given identical tasks in C#, opportunistic programmers were given identical VB.NET tasks. This design was intended to give each persona a familiar and representative work environment. We were able to let each persona use a different language while having them use the same APIs by using cross-language .NET assemblies.

The study involved six different programming tasks that participants performed in-order over 2 hours and 15 minutes. Some of these tasks had two conditions to allow us to compare two possible versions of an API. The tasks were chosen to be of several different domains to gain a more general understanding of the usability tradeoffs.

Each session lasted up to 2 hours and 15 minutes. Participants were able to speak to the experimenter over an intercom system, and were allowed to ask questions, however most questions were not answered, to avoid influencing participants' behavior. Participants were asked to vocalized their thought process throughout the tasks and were reminded by the experimenter if they fell silent.

When participants reached a point in a task when unable to make any further progress, the experimenter first sought to get the participant to vocalize what they thought the current problem was, what they had tried to solve it, and what they thought other possible solutions might be. When participants had attempted all of the possible solutions they could think of, the experimenter would offer advice to help the participants make further progress.

3.3. Task 1: Notepad Programming

Task 1 instructed participants to "Write the code they would expect would read in a file and send its contents in the body of an email message." They were asked to use only the text editor Notepad to write their code.

In addition to being a warm-up task (because their code is not error-checked or compiled there can be no "wrong code"), this task was designed to elicit participants' expectations and mental models without the influences of code-completion, example code or extensive task wording. Specifically the task makes it likely that participants will initialize multiple objects so that we can see what type of constructors they expect to be able to use.

Because there was no provided code, there was only a single condition for this task.

3.4. Task 1-B: File API design

Task 1-B involved using Notepad to design an API for file reading and writing operations. This task was given *only* to systematic programmers and was used in place of Task 1 for these participants. The motivation for changing Task 1 for systematic programmers was to elicit even more assumptions from the programmers who had more experience *designing* APIs about all of the objects constructors that should be offered in a class. Participants were asked to write the declarations for the API without implementing it.

The file domain was chosen because it offered at least one likely candidate for a required property – the file's name or "path" – and all of the existing .NET APIs for files include this as a required constructor parameter.

As with Task 1, because of the free-form nature of the task, there was only one condition.

3.5. Task 2: Files and Emails

In Task 2 participants were asked to write code that performed the same function as the code in Task 1, however this time using the Visual Studio IDE and real APIs. Participants were given a template project in which to write their code and the project was linked to one of two libraries, depending on the experimental condition. The libraries each provided APIs for File and Mail operations, the difference being that one provided only default constructors (taking no arguments) for each object and the other provided only required constructors (requiring all parameters to be provided on construction).

This task was designed to compare between participants the ease of use of the create-set-call APIs to the required-constructor APIs. It also provided an opportunity for participants to comment on differences in the provided API and their imagined API from the previous task.

This was a code-creation task that used real APIs (which we hid with wrapper APIs when it was necessary to change which constructors were provided) and had two different conditions.

3.6. Task 3: Domain-Independent Classes

Task 3 had participants create and use two objects. Using the object involved calling a specific method "`use()`", on each object. The objects were given plausible but not understandable names and properties (the objects were "`CptrObject`" and "`CptrModel`"). By using a made-up domain whose requirements participants have no experience with or intuition about, this task was designed to also help answer the question of how well the different patterns convey object requirements to programmers who are unfamiliar with them.

Of the two objects, each had several required properties. One required these properties in its only constructor and the other provided a default constructor and a constructor that took different combinations of the required properties. When the create-set-call object was used without initializing all of the required properties, the object threw a runtime exception, while code that constructed the required-constructor object would not compile unless the proper arguments were provided.

This task was a code-creation task with a single condition. Each participant created both objects.

3.7. Task 4: Message Queue Debugging

In Task 4 participants were given a short (100 line) program that sent and received messages using the .NET System.Messaging API. A bug in the construction of the `MessageQueue` class prevented messages from being received: the instances were created with the Boolean "`DenySharedReceive`" argument set to true, which caused the `MessageQueue` to throw an "access denied" runtime exception.

There were two conditions for this task. In the first condition the `MessageQueue`s were constructed using a default constructor and the `DenySharedReceive` property (along with other properties) was set on a separate line (`messageQueue.DenySharedReceive =true;`). In the second condition the `MessageQueue`s were constructed using a four-parameter constructor where the second argument represented the `DenySharedReceive` parameter.

In order to solve the task, participants had to change the `DenySharedReceive` property or constructor argument from true to false for both `MessageQueue` instances.

This task was designed to compare the readability and debugability of code that uses constructors vs. code that uses create-set-call. By requiring a small fix in two separate code locations, the task was intended to be complex enough to require understanding of the code while still being solvable in a reasonable amount of time.

3.8. Task 5: Optional Constructors

Task 5 involved a small application that initialized the inventory of an online store and it required the creation of several objects of different complexity. (For example, a book required an author, title and ISBN, while a magazine only required a title and ISBN). There were 5 different objects, which required up to 5 properties, and each provided a range of constructors that included a default constructor and a constructor that took all essential parameters.

By providing participants the choice of which constructors to use, after having seen APIs that used required-constructors and create-set-call in earlier tasks, this task was designed to test the usability of optional "convenience" constructors. By providing objects of a range of complexities, the task sought to test whether there were trade-offs in construction approaches depending on the number of arguments.

There was only one set of APIs and each participant constructed each object, however there were two conditions: one where the task instructions presented the objects to create in increasing order of complexity and one where the objects were presented in decreasing order of complexity.

3.9. Task 6: Reading Code on Paper

In Task 6 participants were given a paper printout of a short program (a dozen lines) and asked what the program would do. The program called imaginary APIs that took either Boolean constructor arguments of ambiguous meaning or used create-set-call to set Boolean properties.

This task was designed to test the readability of printed code (in the absence of IDE features like code-completion) for each construction pattern. While constructor calls clearly convey less information, since they do not include the parameter names, we hypothesized that by being easy to overlook, participants might skim over unnamed parameters and fail to realize their lack of comprehension. There were two conditions: one that used constructors and another that used create-set-call.

3.10. Interviews

In addition to the programming tasks, we prepared questions for a semi-structured face-to-face interview to follow the tasks. We began the interviews by describing to the participant the focus of the study. (The participants had previously only been told that the study involved performing small programming tasks.) Hearing the focus of the study, participants would usually offer their opinion on why APIs should or should not require constructors. After listening to their opinion, we offered our own study observations so far to engage a dialog of the advantages of each option. We then asked participants which APIs they used in their professional programming work, and what API design practices they used if API creation was a part of their job.

4. Study Results

The following observations were taken from notes the experimenter made while running the study and while reviewing screen-captured video taken during the study. Although we could have made more quantitative assessments of participants' behavior, the primary goal of the study was to communicate our perceptions to .NET framework developers by example and trend rather than statistics. Nevertheless, the process by which we derived our observations was systematic. For example, if we believed we had observed a particular trend, we did investigate the videos thoroughly to verify its existence.

4.1. Common Participant Behavior

We consistently found that opportunistic and pragmatic programmers assumed that a default constructor exists for any class. This was often evident by participants writing code to call a default constructor and not noticing until the next line of code or two that the constructor call would not compile. Their expectations were also evident by a common misunderstanding of why the constructor call would not compile, especially by opportunistic programmers. These programmers were much more likely to initially assume the compiler error resulted from incorrect syntax – a missing parenthesis or keyword – than a more semantic error. This often caused participants to doubt their own syntactic understanding of a language; however when using create-set-call APIs, these same participants rarely made syntactic errors, indicating that these programmers were in fact relatively familiar with the language's syntax. We found that these assumptions did not change over the course of the study, even after exposure to several APIs that used required constructors.

When opportunistic and pragmatic participants discovered that they needed to use a required constructor, they tended not to interpret this as a functional requirement imposed by the API but rather a syntactic barrier to compilation. A common reaction to a required constructor was to try to pass null for the parameter (in the APIs in this study, this would always cause a runtime exception). Another strategy we observed was to create empty new objects for each required parameter, without trying to initialize or validate these objects.

Though we found required constructors to be less usable when creating code, we did not find the same to be true when participants debugged code. Even when code used ambiguous constructor parameters such as "`true, true`", programmers did not a have significantly harder time debugging this code compared with seemingly more self descriptive code like "`obj.sharing = true; obj.caching = true;`". This was because all of our participants used IDE features like code-completion to easily access constructor parameter information when it was not directly visible in the code.

While required constructors hurt usability, we found no negative impact from *optional* constructors: constructors provided in addition to a default constructor. Optional constructors were sometimes helpful, most often to pragmatic programmers, by suggesting what combinations of properties might be used together, and by provided a shorter mechanism for initializing multiple properties.

4.2. Task 1 Results: Notepad Programming

All the participants used create-set-call when creating objects in their Notepad programming task.

The opportunistic programmers were more resistant to the idea of writing code outside of an IDE than pragmatic programmers.

4.3. Task 1-B Results: File API Design

All of the systematic participants designed APIs for a File class that included a default constructor, allowing the possibility that a File could exist in a state where the filename had not yet been set. This was surprising as all of the participants had experience with real file APIs from Microsoft libraries, none of which provide a default constructor.

In addition to a default constructor, most participants also provided at least one additional constructor that accepted a filename as a constructor parameter.

4.4. Task 2 Results: Files and Emails

Participants in the create-set-call condition completed this task with less difficultly than participants in the required-constructor condition.

4.5. Task 3 Results: Domain-Independent Classes

Multiple participants, of pragmatic and opportunistic personas, attempted to pass in the value null to the required constructor in this task. Passing null caused a runtime exception to be thrown. No participant ever tried setting a property to null to satisfy a create-set-call condition.

For the create-set-call object, participants tended to quickly discover which three of the nine possible properties were necessary to complete the object, even though these requirements were completely arbitrary. In contrast, many participants vocalized wrong assumptions about why they thought compiler errors appeared when these participants had failed to use the required constructor. These participants often assumed that the error was one of programming syntax, and were often slow in discovering the actual problem.

Unlike the Notepad programming task, opportunistic participants were less hesitant to start this task, voicing less reluctance, while pragmatic programmers were less comfortable with starting a task when they did not understand the domain or overall goal.

4.6. Task 4 Results: Message Queue Debugging

A few participants in the create-set-call condition of the debugging task did have some difficulty stemming from the Boolean constructor arguments of this condition. However, this was neither common nor severe, and we found that participants used IDE features to overcome any difference in readability.

4.7. Task 5 Results: Optional Constructors

Most participants used create-set-call when using objects that provided either constructor mechanism. Despite the fact that the objects were of different complexities, participants tended to use either create-set-call or convenience constructors for all of the objects, instead of mixing and matching constructor approaches. Starting with complex or simple objects did not seem to influence whether or not participants used convenience constructors.

4.8. Task 6 Results: Reading Code on Paper

Printed code that called constructors conveyed less information than create-set-call code, since the constructor parameter names were not visible and there were no IDE tools to help display them. However, this did not affect participants' awareness of their lack of knowledge as we had hypothesized. In addition, none of the participants reported reading paper code printouts as part of their professional programming job.

4.9. Interview Results

In the post-task interviews, nearly all of the participants expressed a preference for the create-set-call pattern. Following are some of the justifications they gave for their preference.

- *Initialization flexibility*: By allowing objects to be created before all the property values are known, create-set-call allows objects to be created in one place and initialized someone else, possibly in another class or package. This was a common justification given by pragmatic programmers.
- *Less restrictive*: In general, APIs should let their consumers decide how to do things, and not force one way over another.
- *Consistency*: Most APIs have default constructors, and so people will expect them. This reason was given by two programmers who created APIs that were used by other members of their teams.

- *More control*: Several systematic programmers cited the fact that create-set-call let them attempt to set each property individually and deal with any errors that might come up using return-codes, while constructors only allowed for exceptions.

5. Discussion

We found the create-set-call pattern to be more usable than, and preferred to, required constructors. The reasons for this differed based on persona, but this held for each persona.

Opportunistic programmers benefited the most from the create-set-call pattern. Even experienced opportunistic programmers experienced difficulties using APIs that did not offer a default constructor, and this effect continued even after participants had used multiple APIs with each pattern. Opportunistic programmers expressed a preference for the create-set-call pattern, and this issue is important to these programmers' effectiveness.

Pragmatic programmers were more effective using required constructors than opportunistic programmers, however they too were more effective with create-set-call and had preferred create-set-call. While not as critical of an issue for these programmers, required constructors provide a minor stumbling block to opportunistic programmers' effectiveness and a minor annoyance that can cause them to prefer one API over another.

Systematic programmers were equally effective at using each type of API, however as with the other personas they too preferred APIs that used create-set-call. There reasons were different, citing the greater flexibility that create-set-call provides in initializing objects in any order and by being able to return error-codes. Contrary to our expectations, they did not feel that required constructors offered any assurances about the validity of an object. For this persona the choice of constructors was relatively unimportant to their effectiveness or preference, however the create-set-call pattern was consistently favored.

Based on these observations, we recommend against the use of required constructor parameters in new APIs, favoring instead the create-set-call pattern, especially for APIs targeting the opportunistic persona.

6. Model of Participants' Strategies

To better understand the underlying causes of opportunistic and pragmatic programmers' greater effectiveness with create-set-call we analyzed videos of participants' work and created a model of the participants' strategies for creating and using new objects. This model is represented graphically in Figure 2 and described in more detail below.

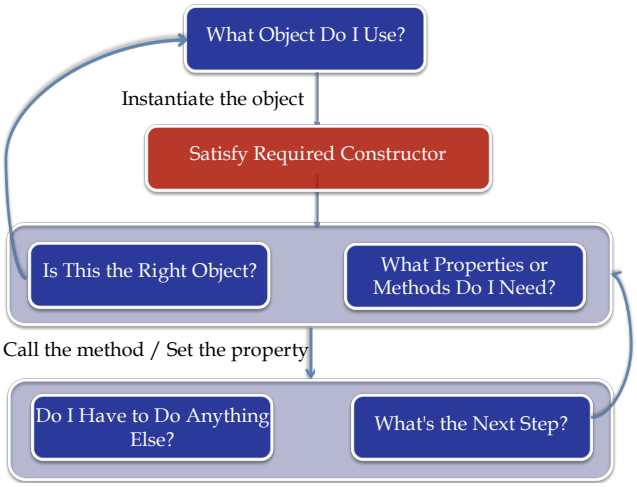

Figure 2. When constructors were required, the IDE indicated a compiler error, leading users to interrupt their exploration to satisfy the required constructor.

When participants encountered a specific problem, such as how to read in a text file or send an email message, they first looked for a class they could instantiate. As an implicit part of this step, participants assumed how many objects the API would provide and what their general function would be. When APIs provided this functionality using a different number and composition of objects, participants had great difficulty (see Section 6.2). Code-completion was the most common tool participants used in this step. Other tools include the IDE's object-browser and searching of the documentation.

When participants had a candidate class, they then attempted to instantiate it and explore the resulting object, again using code-completion as the primary means of exploration. As part of the exploration process they attempted to answer two questions: (1) is this the correct object?, and (2) what methods or properties perform the needed functions? If after exploration they felt that they probably had the wrong object, they would return to search for more objects.

If they felt the it was the correct object, then after calling a method or setting a property they would try to determine whether they were done (with this object) and if not what the next step was, figuring out how to solve the new step in the same exploration manner as the previous step.

When objects used the create-set-call pattern, the exploration of an instance's properties and methods directly followed finding a candidate object. However, in the case of objects with required constructors,

participants were forced to satisfy the required constructor (the second box from the top in Figure 2) – usually by figuring out what the compiler error was, then recursively trying to instantiate objects for each of the required parameters – before they could finish deciding if the object was even the one they wanted.

By requiring more effort at such an early stage of object exploration, required constructors created, in terms of the cognitive dimensions, a larger *work-step unit*, and greater *premature commitment*. Required constructors decreased *diffuseness* of the code, however we did not see an increase in readability as a result. In addition, participants were often annoyed by the unexpected interruption of their exploration, and simply wanted the current construction problem to go away so they could continue their task of finding the right object.

7. Study Limitations

From a research perspective, we do not intend to make causal claims about the patterns described in this paper. These patterns are safely interpreted as hypotheses, backed by systematic observation.

Given this limitation, there were several factors that could influenced the observations we made. We observed ordering effects resulting from having programmers perform tasks in the same order. We intentionally maintained this task ordering so that participants would be guaranteed to have seen both create-set-call and required constructor APIs by the time they reached tasks involving debugging or optional constructors. Because we found participants' expectations of create-set-call did not change even after having recently used two or three APIs with required constructors, the ordering effects do not weaken the results.

Because the individual tasks we tested were of relatively small length and the participants had not used the APIs before, our results might not generalize to how programmers use APIs that they are familiar with. Studying behavior in long-term use of APIs would require either much longer studies, over multiple sessions, or a less controlled study of how programmers use APIs in their real projects.

The consistency of the results we saw across three different programming languages suggest that the results generalize to other object oriented languages as well. However, differing syntax, for example named constructor parameters in languages like Objective C, may offer additional variables that need to be taken into account.

We feel the programmers were representative of professional programmers who use the .NET framework. Because even experienced professional programmers encountered difficulty using required constructors, we feel that less professional or less experienced developers would experience at least as much difficulty.

The tasks participants performed were smaller than typical programming tasks, however because object construction typically occurs at the beginning of a task, we feel that the constructor and parameter setting in the study tasks is representative of larger tasks.

8 Future Work

The tasks in this study several common usability issues that while not directly related to create-set-call, suggest several fruitful avenues of API usability research.

8.1. Runtime Exceptions vs. Compiler Errors

We were surprised by the effectiveness of runtime exceptions in conveying API requirements when the opportunistic and pragmatic participants had more difficultly understanding these requirements from similar compiler errors. Part of this seemed to stem from a common assumption that compiler errors related to syntactic rather than semantic problems.

The comparison between the two forms of feedback is interesting in part because compiler research is often focused on attempting to catch errors that might get left until runtime and detecting them at compile-time. However, because of programmers' work strategies, some of these errors might be better dealt with at runtime.

While we have consistent evidence of this effect in the case of constructor and property requirements, a study that compared a broader range of compiler errors and runtime-exceptions, and how these interact with programmers work styles, would be able to offer more complete guidance to API designers and compiler writers.

This guidance might suggest a third approach that detected errors at compile time but not making this error prominent to developers when in "exploration mode," only revealing the error when the developers are ready to "spot check" the code for bugs.

8.2. Object Decomposition

Another important API usability theme that occurred in our study was how API functionality was broken-up between different objects, for example whether there is a single Mail class or both MailMessage and MailServer classes.

This was of particular importance because participants were slow to realize or change their assumptions about what classes should exist and any violations of their assumptions created a significant barrier to their effective use of an API.

A better understanding of how programmers make their assumptions, how APIs can be designed to simultaneously service multiple sets of assumptions, and how development tools can give programmers a greater awareness of their assumptions could reduce this barrier to productivity.

8.3. Debugging Strategies by Persona

We found different personas to have markedly different effectiveness with the debugging problem in Task 4 that involved multiple interacting components.

Systematic programmers tended to debug programs from the top down, trying to understand the system as a whole and the overall architecture before focusing on a specific piece. Opportunistic programmers tended to debug from the bottom up, starting from the line of code that first exhibited the error. Pragmatic programmers debugged using a bottom-up strategy, but would switch to top-down when the bottom-up strategy was ineffective.

Systematic and pragmatic programmers were more effective when debugging multiple components, while opportunistic programmers would often focus their on a single component at a time, making multi-component problems much more difficult.

A study that compared a wider variety of multi-component debugging tasks would offer a more precise view of different personas' strategies and the types of bugs likely to be most problematic for each. This knowledge would be applicable to API designers creating APIs that avoid difficult errors by their audience.

9. Conclusions

Based on a study of 30 programmers of three different personas, we have found that APIs that required constructor parameters did not prevent errors as expected and that APIs that instead used the create-set-call pattern of object construction were more usable.

This study offers evidence that API usability can be a significant barrier for programmers, but that despite the challenges facing API consumers and creators, it is possible to create APIs that are highly usable by a broad range of programmers. Some of this recent success in creating usable APIs is due to running studies of specific APIs: users of a target population perform realistic tasks with an early version of an API [4]. However, this approach is difficult for smaller organizations to apply, requiring resources for a user study, and expensive for larger organizations, which might produce thousands of different APIs a year.

Our approach is to study API design choices relevant to many different APIs. By using several tasks that include different instances of a specific API design choice, we can develop general usability guidelines that are not specific to any particular API or domain. By creating a set of API design recommendations, we hope to be able to significantly improve the usability of newly designed APIs.

10. Acknowledgements

This study was completed with the help of the Visual Studio User Experience group at Microsoft, and under the guidance of Monty Hammontree. We wish to thank Andrew Ko, Brad Myers, Jonathan Aldrich, Justin Weisz, Christopher Scaffidi and Brian Ellis for their helpful comments on drafts of this paper.

11. References

[1] Bloch, J., *Effective Java Programming Language Guide*, Sun Microsystems, Mountain View, CA, 2001.

[2] Bore, C, and S. Bore, "Profiling software API usability for consumer electronics", *Consumer Electronics*, 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, pp S6-S9.

[5] Cwalina, K. and B. Abrams, *Framework Design Guidelines*, Addison-Wesley Professional, 2005.

[6] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, 1995.

[7] Green, T.R.G., and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework", *Journal of Visual Languages and Computing*, 1996, pp 131-174.

[8] Grudin, J. and J. Pruitt, *Personas, Participatory Design and Product Development: An Infrastructure for Engagement*, 2002.

[9] McLellan, S.G., A.W. Roesler, et al, "Building More Usable APIs", *Software*, IEEE, 1998, 15(3) pp 78-86.