The Factory Pattern in API Design: A Usability Evaluation

Brian Ellis, Jeffrey Stylos, and Brad Myers
Carnegie Mellon University
firebird@cs.cmu.edu, jsstylos@cs.cmu.edu, bam@cs.cmu.edu

Abstract

The usability of software APIs is an important and infrequently researched topic. A user study comparing the usability of the factory pattern and constructors in API designs found highly significant results indicating that factories are detrimental to API usability in several varied situations. The results showed that users require significantly more time (p = 0.005) to construct an object with a factory than with a constructor while performing both context-sensitive and context-free tasks. These results suggest that the use of factories can and should be avoided in many cases where other techniques, such as constructors or class clusters, can be used instead.

1. Introduction

Whether creating a piece of desktop software, writing applications for handheld devices, or scripting the Web, the use of application programming interfaces (APIs) in modern software development is ubiquitous. These APIs, also called software development kits (SDKs) or libraries, are often large, complex, and broad in scope, containing many hundreds or thousands of classes and interfaces. A typical developer may use only a small portion of the total functionality of an API, but learning even that subset can be a daunting task for new programmers [1].

API designers must consider many different factors when creating an API, such as class granularity, level of abstraction, consistency with other APIs, etc. Research has also shown that designing APIs carefully for their intended audience improves usability [2]. To date, usability studies of APIs have mostly considered the usability of the API as a whole, providing minimal guidance for future API designers. Little research has examined the usability of specific design patterns and programming paradigms as applied to API design.

In a previous paper, Stylos et al. [3] discussed the usability of object constructors with required parameters as compared to default constructors. Here, we consider the usability implications of one of the best-known object-oriented design patterns: the factory pattern [4]. Our new study shows that creating objects from factories used in APIs is significantly more time-consuming than from constructors, regardless of context or the level of experience of the programmer using the API. The reasons for this, as well as a discussion of specific stumbling blocks and possible alternative patterns, are discussed below.

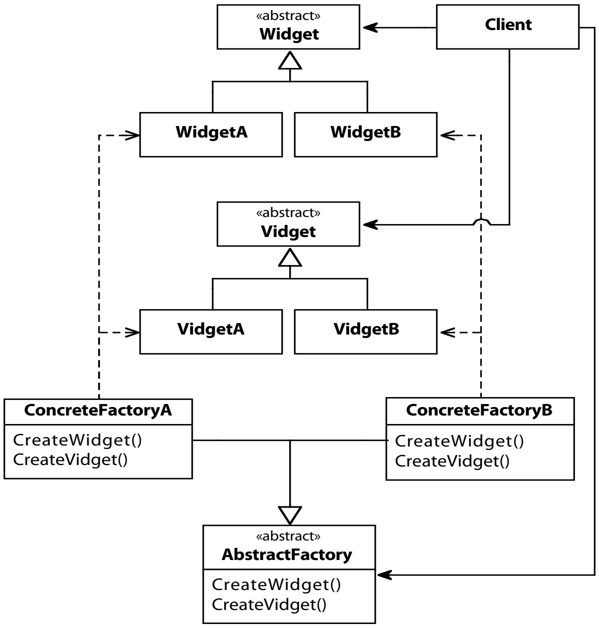

Figure 1. The abstract factory pattern in UML

2. The Factory Pattern

The "factory pattern" refers to two distinct design patterns, both first described by the "Gang of Four" (Gamma, Helm, Johnson, and Vlissides) [4]. The "abstract factory" pattern provides an interface with which a client can obtain instances of classes conforming to a particular interface or protocol without having to know precisely what class they are obtaining. This has advantages for encapsulation and code reuse, since

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 © 2007 IEEE

implementations can be modified without necessitating any changes to client code. Factories can also be used to closely manage the allocation and initialization process, since a factory need not necessarily allocate a new object each time it is asked for one. The abstract factory pattern is usually implemented as shown in Figure 1. To obtain a Widget instance, a programmer would first obtain a reference to one of the hidden factory subclasses, usually through a factory method in the abstract factory superclass, then use that reference to create an object of the product type. In the example shown, the code to do this might look like this:

```
AbstractFactory f =
  AbstractFactory.getDefault();
Widget w = f.createWidget();
```

The "factory method" pattern is related but simpler: like the abstract factory pattern, the factory method pattern allows a client to obtain objects of an unknown class that implement a particular interface. Rather than relying on a separate factory class to create instances of the product classes, the product class itself has a factory method that returns an object that conforms to the interface defined by that class. Typically, a class implementing a factory method pattern would be an abstract class with several concrete subclasses, and would present a static method that could be called like this:

```
Widget w = Widget.create();
```

This offers some of the same benefits as the abstract factory (e.g., the ability to return objects of a subclass type or objects that already exist) while still maintaining an ease of implementation and locality of reference that make it an attractive solution to many problems. Strictly speaking, for instance, the standard implementation of the singleton pattern [4] is a factory method pattern. Factory method patterns are also often used in an abstract factory implementation as an entry point to the factory class hierarchy. For example, an abstract factory superclass might define a *getDefault* method that would return an appropriate concrete factory subclass.

2.1. Why Use a Factory Pattern?

Gamma, et al. describe in some detail both the benefits and liabilities of the abstract factory and factory method pattern as they see them [4]. In terms of benefits, the factory pattern enforces the dependency inversion principle: the dependencies of the client are solely to abstract classes and interfaces, and never to the concrete subclasses they are passed. Second, it decouples the concrete factory and product instances from everything but their point of instantiation. This means, in the case of the abstract factory, that factories can be swapped in and out simply by changing which factory is instantiated, and without touching any other code. Third, the factory pattern facilitates the creation of consistent products (since they are presumably all instantiated using the same factory).

Gamma, et al. [4] only mention one liability: the difficulty of adding new types of products, due to the need for a separate factory class (in the case of an abstract factory) or a special case of the factory method. Later publications, however, discuss another problem, which stems, ironically, from one of the benefits described above. The concrete factory and product instances are decoupled *from everything but their point of instantiation*. This is not merely an implementation detail; it is a necessary consequence of the design of most modern object-oriented programming languages, which implicitly use a very strict constructor pattern for object instantiation. Because the exact concrete class of an object must be explicitly named in order for it to be constructed, it is impossible not to have a concrete dependency on that name.

To avoid requiring the client to directly instantiate a concrete factory subclass, the abstract factory must itself employ the factory method pattern to return a polymorphically typed instance of one of its concrete subclasses. (Theoretically, another class in the API could contain the factory method instead, but in practice this is rarely the case.) This increases the complexity of the code demonstrably, as we shall see later on, and requires that the abstract factory superclass contain concrete references to all of its subclasses.

Lastly, a true abstract factory implementation will by necessity require developers to explicitly downcast its product instances if they are to use any subclass-specific functionality. If the abstract factory superclass provides a creation method, subclasses must override that method, including its return type. This means that even if the subclass factory only ever returns objects of a certain concrete class, the returned type will be of the abstract product superclass. This does not pose a problem if the product subclasses are to be hidden from the user, but in many real-life abstract factories (such as Java's SocketFactory discussed below) this is not the case, and explicit downcasting is required.

2.2. Applications of the Factory Pattern

Many popular object-oriented APIs make use of factory patterns. It is difficult to estimate the number of factory method patterns in use, since any class may in fact be implemented as a factory, and algorithmic means of detecting them are non-trivial [5]. By convention, however, factory classes often end with the word "Factory" — using this simple metric, the Microsoft .NET API contains 13 classes (out of 2,686) that definitely play roles in an abstract factory pattern; these often come in pairs (ISecureFactory and SecureFactory, for example) where one is an abstract factory interface and the other a single concrete factory implementing that interface [6]. More prolifically, the Java 1.5 SE API boasts some 61 factory classes and interfaces (out of 3,279 total) [7]. These numbers definitely exclude many factories, however, especially in Java: the DocumentBuilderFactory class, for example, generates DocumentBuilders, which are themselves factories used to generate Documents. .NET takes a more monolithic approach to factories when they are used; a single factory class can return a wide range of different objects, whereas in Java there is typically a strong mapping between product class and factory class name.

In both .NET and Java, the abstract factory pattern is used especially in the context of allocating shared resources and objects managed by the operating system: Java has factory classes for several kinds of sockets, preferences objects, threads, and user interface controls. .NET mirrors this focus, with database connector factories, configuration and settings factories, and a factory class devoted to security measures [6]. The wide adoption of the factory pattern in large, well-known APIs such as these shows the importance of studying the use of factory patterns in API designs.

2.3. Alternatives to the Factory Pattern

In simple cases, a constructor can often be directly used in place of a factory. This obviates the need for a hierarchy of factory or product classes. It also requires programmers to refer directly to the concrete subclass being constructed, however, and therefore cannot be used when the designer wishes to hide the existence of subclasses or eliminate concrete dependencies.

However, there are other patterns that have many of the same benefits as the factory pattern and overcome the usability problems. One such pattern is called the class cluster [8]. Class clusters are designed for dynamically typed languages such as Smalltalk and Objective-C, but can be adapted to languages like Java by applying the handle-body idiom [9]. Like a factory, the parent class depends directly upon its children, but no special factory class is necessary. Instead, the "factory" *is* the "product." From the perspective of the API designer, writing a class cluster in Java is somewhat more complex than writing a factory would be for the same task. The interface presented to the programmer, however, is much simpler.

A class cluster could be used to implement the example shown in Figure 1: a Widget class that dynamically determines its behavior given some condition, perhaps passed in as a constructor parameter. From outside the class, the Widget object would appear the same regardless of the condition. Internally, however, the class might use that condition to determine what private subclass to create, exactly as a factory would do. The Widget class could be implemented as follows:

```
public class Widget {
  private Widget body;
  public Widget(boolean b) {
    if (b) {
      body = new WidgetA();
    } else {
      body = new WidgetB();
    }
  }
  public void performAction() {
    body.performAction();
  }
}
class WidgetA extends Widget {
  public WidgetA() { ... }
  public void performAction() { ... }
}
class WidgetB extends Widget {
  public WidgetB() { ... }
  public void performAction() { ... }
}
```

Many variations and improvements upon this basic idea could easily be realized: using reflection to obviate the need for explicit method forwarding, for example. In any event, the interface presented by the Widget class is exactly the same as it would be if no subclasses existed. Users can type:

```
Widget w = new Widget(true);
```

and get back a Widget conforming to the implementation for WidgetA. The Widget constructor could perform whatever environment-specific checks the factory would otherwise perform.

The class cluster provides several of the same advantages over constructors that factories do, most importantly the ability to hide private subclass implementations behind an abstract superclass. A class cluster can also be used, like a factory, to avoid allocating a new subclass object each time one is requested (in, for example, a socket pool). Although the superclass instance is created using the "new" operator and therefore allocates memory, the same is true of a factory class instance unless it is generated some other way, such as by a factory method. Such an approach could be used with a class cluster as well without resorting to a factory class. Note also that when using class clusters, there are no longer two parallel class hierarchies, one of products and the other of factories, which is another advantage over factories.

3. Related Work

The usability analysis of API designs is a relatively new area. However, there have already been several relevant explorations into the subject. Microsoft has employed the "cognitive dimensions" framework [10] to compare the usability of different API designs for three "personas" representing different archetypes of developers likely to use the API [11]. The results of these comparisons are used to inform the design process and improve the API.

Research has also been conducted into the role of design patterns in general, and the abstract factory pattern in particular, in computer science curricula [9]. This work shows that although educators consider the factory pattern a superior method, they feel that it is too difficult to explain to beginning students, and therefore they avoid it in favor of others such as the handle-body idiom.

Our study focuses on the usability of APIs that employ the factory pattern, while past research on factories has mostly focused on the architectural advantages of the factory pattern for system implementers. An earlier study [3] demonstrated that user testing is an effective means of determining usability properties of APIs such as discoverability and adherence to user expectation. Here, we apply this approach to the use of the factory pattern, and a companion paper [12] compares the usability of parameterized constructors with that of default constructors.

4. Study design

In designing our study, we tried to minimize the dimensions of variability to isolate inherent differences in usability between API implementations that use constructors compared to those that use the factory pattern. We also presented the factory pattern in as many contexts as possible to account for any bias toward or against factories in one particular context (e.g., in networking) by participants who may have seen factories in that context before.

A second goal was to maximize the external validity of our results by presenting factories and constructors in at least one "real-world" use, in order to better capture the complex interactions between the means of construction of an object and the role that the object plays in the user's conceptual model of the API.

4.1. Methodology

We crafted a series of five Java programming tasks to explore the use of the factory pattern in APIs. In order to gain a broad understanding, each task was constructed to differ from all the others along as many dimensions as possible. The order of the tasks was randomized as much as possible to minimize confounding due to learning effects. All tasks except the first were presented in the form of an Eclipse [13] project. Participants were also given access to the Sun Java 1.5 SE API documentation [7].

Participants were selected from a pool of applicants generated using an advertising service for on-campus experiments, postings to an on-campus bulletin board, paper flyers posted around campus, and word-of-mouth advertising. We used a pre-screening survey to eliminate candidates from this pool that did not have at least one year of Java experience. This resulted in a diverse group of participants that included professional developers and software engineers, electrical and computer engineers, and non-technical hobbyist programmers, as well as computer science students. Twelve participants were selected, with programming experience ranging from one to twenty-two years. Eight had professional programming experience, eight were students in a computer- or electronics-related major, and two were non-technical students. Six had at least some experience with the factory pattern, and four had considerable experience with the factory pattern. All participants were males between 18 and 35 years old.

Participants were randomly put into the factory or constructor conditions for those tasks which had two versions. Each participant was given written instructions for completing each task, and was asked to verbalize his goals, assumptions, suppositions, and strategies for completing the tasks using a think-aloud protocol. Participants were told to complete each task in the order it was presented, and not to move on to subsequent tasks until the task was completed. Whenever possible, tasks were designed so the subjects knew when they had been successfully completed.

4.2. Measurement

A major goal of this study was to provide quantitative measurements of the differences in usability between factories and constructors. In the context of APIs, where the goal is often to write correct code as quickly and efficiently as possible, usability is highly correlated with time to task completion, which also includes such activities as researching the documentation. Due to large individual differences, completion time is easiest to compare when measured within subjects; hence, presenting both a factory and a constructor was used instead of separate conditions in two of the tasks.

We also administered a survey to each participant after they completed the programming tasks to find out about their programming background and familiarity with design patterns.

4.3. Notepad Email Task

The Notepad email task was always the first task administered. It differed from the other four tasks in that, rather than using Eclipse, participants were presented with a blank plain-text document in the Notepad text editor and asked to write Java code using whatever real or imaginary APIs they wanted. This task was designed to elicit the programmer's expectation regarding object creation.

Participants were asked to construct an email object with a list of information including the sender and recipient address, email body, and (most importantly) whether the email was plain or rich text. The last parameter makes the task a candidate for the use of a factory pattern by suggesting two subtypes of email whose implementation might be hidden by a factory.

4.4. Eclipse Email Task

A second email-related task, administered as an Eclipse project, used the same task description as the Notepad email task: write a method that takes parameters for an email and returns an Email object. This time, however, participants were asked to use a simple email API pre-built by the experimenters. The presented API created its emails using a factory rather than constructors. Although the lack of a constructor condition for this task precluded a direct comparison, the task, coupled with the think-aloud process, was intended to elucidate users' reactions to finding a factory when a constructor was expected.

4.5. Thingies Task

The "Thingies" task was designed to be an entirely context-neutral task, with the intent of measuring user expectation, preference, and responses to both factories and constructors in the absence of any prior domain knowledge. Participants were asked to create a "Squark" and a "Flarn", two subclasses of the abstract "Thingy" class, and then call a simple *run* method on each of them. The Squark was implemented as the product of a (concrete) SquarkFactory, whereas the Flarn was implemented as a simple concrete class with a public default constructor.

4.6. PIUtils Task

We were interested in the usability of the factory pattern while debugging, and not just when constructing new objects. The "PIUtils" task consisted of a pre-written method that was intended to display two dialog boxes on screen, one which laid out its controls according to the Windows user experience guidelines, the other according to the Macintosh human interface guidelines. A bug was introduced into the code that caused both dialogs to lay out their controls as on Windows, and participants were asked to find and fix the bug. The bug was in fact due to a misinterpretation of the role of a method in the PIDialogLayout class, which was provided (with documentation, but without source code) as part of the task.

The PIUtils task had two conditions, of which only one was given to each participant. In the factory condition, the PIDialogLayout class was created by passing parameters into the *createLayout* method of a layout factory class. Here, the bug could be fixed by passing different parameters to the factory. In the constructor condition, the PIDialogLayout class was actually implemented with a class cluster, and instances were created directly using a default constructor. In this condition, the bug could be fixed by calling the *addOperatingSystem* method with a different value.

4.7. Sockets Task

The Sockets task was designed to represent as realistic an experience as possible with a real-life factory pattern from the Java API. Participants were instructed to construct an SSLSocket and a MulticastSocket (defined in the Java API), configure them to connect to a particular server and port, and pass them into a method that would perform the actual connection. The SSLSocket class cannot be directly constructed, and must instead be created by first obtaining a reference to an SSLSocketFactory (which is itself a concrete subclass of the SocketFactory class — a textbook example of an abstract factory pattern) and then calling a factory method on it. The MulticastSocket, on the other hand, is a concrete subclass of Socket and has several public constructors.

5. Results

5. 1. Notepad Email

All twelve participants used a constructor call in their implementation of the method. Three created separate subclasses for each type of email (rich-text and plain-text), whereas two passed the type of email as a parameter, and seven used a setter method on an already constructed object. None used, or reported that they even considered using, a factory during the task.

5.2. Eclipse Email

Two out of twelve participants randomly were assigned to do the Eclipse Email last and did not have sufficient time to begin it (so n = 10). Of those who performed the task, seven participants attempted to use a constructor, despite the lack of one in the documentation, before concluding that there was no public constructor. Three of these participants then attempted to create a concrete subclass of the abstract Email class. All ten eventually found and successfully used the factory, even though all of them had used a constructor call in their hypothetical implementation during the Notepad Email task.

5.3. Thingies

Two participants did not have time to begin the Thingies task (n = 10). All of those who reached the task completed it successfully. The median time for constructing a Squark (using a factory) was 7:10 (minutes:seconds, *SD* = 3:53). The median time for constructing a Flarn (using a constructor) was 1:20 (*SD* = 0:50). On average, participants spent 84.3% of their time constructing objects during the Thingies task working on Squark construction, as compared to 15.7% of the time working on the Flarn construction.

The time data were tested for normality, and although deviations from normality were not significant (p = 0.274 for Squark, p = 0.129 for Flarn), the data were sufficiently skewed that we used the Wilcoxon Signed Ranks test. Highly significant differences were found between the time to complete the Squarks portion of the task (using a factory) and the Flarns portion of the task (using a constructor), with a Z-score of -2.81 (p = 0.005). Figure 2 summarizes the times for the Thingies and other timed tasks.

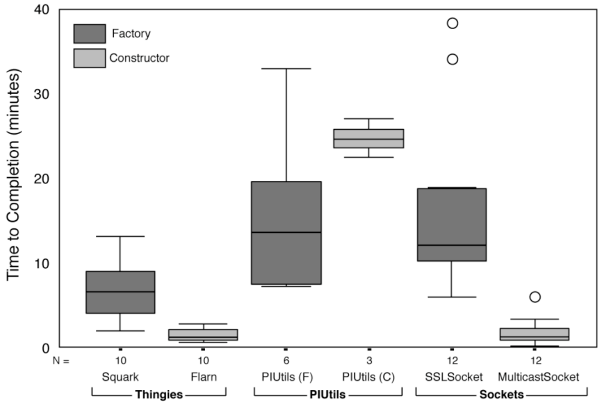

Figure 2. Time to Completion by Task

5.4. PIUtils

Due to the between-subjects nature of the PIUtils task, it was evaluated using a one-way ANOVA. Of the twelve participants, three had insufficient time to begin the task (n = 9). All those who reached the task completed it successfully. Of those, three were in the constructor condition, and six were in the factory condition. (This disparity was the result of an unfortunate coincidence; all three participants who did not have time to perform the PIUtils task had been randomly assigned to the constructor condition.) The mean time to completion in the constructor condition was 26:40 (*SD* = 2:26), and 17:00 in the factory condition (*SD* = 10:26). Since the standard deviation of the factory condition times was 4.2 times that of the constructor condition times, a possible violation of the equal variance assumption was indicated. No significant differences were found between the two conditions (F = 2.35, p = 0.169).

While the data for this task do suggest a general trend toward longer times for the constructor condition, the lack of statistical significance and the very high standard deviation of the factory condition make it difficult to say whether this is an artifact of the sample or a real trend. This is consistent with the findings of previous studies, which have shown that debugging and reading tasks are less apt to reflect significant differences in comprehension or efficiency than authoring tasks [16]. Nevertheless, the question of the factory pattern's ease of debugging relative to constructors might be an avenue for future work.

5.5. Sockets

To better understand participants' behavior on the Sockets task, an experimenter rated the completion of the task into three subtasks: the SSLSocket, the MulticastSocket, and "other activities."

The SSLSocket and MulticastSocket subtasks included such activities as reading documentation in the process of creating an object, writing the code to create the object, and correcting syntax errors in the creation code. Each subtask also encompassed activities related specifically to one or the other socket object, but not both. This included reading documentation relevant to the object; adding, changing, or removing code in support of the constructed object; writing exception handlers for a single subtask; and creating other objects in support of the constructed object.

The "other activities" subtask included all activities that were not directly related to one or the other subtask. This includes such activities as reading documentation for and constructing an instance of supplied helper classes, writing exception handlers common to both tasks (such as wrapping the entire method in an exception handler), and running the task.

The mean time to completion of the SSLSocket subtask was 20:05 (*SD* = 11:17). The median time was 16:05. The mean time to completion of the MulticastSocket subtask was 9:31, with a standard deviation of 8:04 and a median time of 7:41.

We applied the Wilcoxon Signed Ranks test to this task because the data showed significant floor effects. All twelve participants started the Sockets task. Of those, five participants were unable to complete the task before the end of the study; these participants' results were recorded using the total time spent rather than the time to completion. All five failures resulted from an inability to successfully construct an SSLSocket and all had successfully completed all the other parts of the task.

There were highly significant differences between the time to perform the SSLSocket subtask (using a factory) and the MulticastSocket subtask (using a constructor), with a *Z*-score of -2.803 ($p = 0.005$).

5.6. Threats to Validity

Although much care was taken to make the results of the study as generalizable as possible, the selection of such diverse tasks itself presents questions about the validity of the data gathered. For instance, it is unlikely that any reasonable developer would implement an API as in the Thingies task using a factory, so the results of that task can hardly be said to support the notion that factories are worse than other patterns when used for their intended purpose. Nonetheless, several of the tasks (notably PIUtils and Sockets) either used existing factory APIs or recreated APIs for which factories have been used in the past. We therefore feel that the inapplicability of tasks like Thingies does not detract from the validity of the results.

6. Discussion

The most striking result of our study is that factories are demonstrably more difficult than constructors for programmers to use, regardless of context. Both the Sockets results and the Thingies results show a highly significant difference in the time needed to construct an object using a factory vs. using a constructor. This difference is especially meaningful because it implies that it does not matter whether the factory is presented in a vacuum or as the implementation for a particular framework. All subjects found the constructor pattern more "natural" [14], in that they expected that to be the way to create objects, and that was the first technique they tried.

Some patterns such as class clusters can perform many of the same roles in an API as a factory without the same usability costs. Since a class cluster appears externally identical to a single concrete class, the comparison between factories and constructors discussed here also holds true between factories and class clusters. Since many of the benefits of factories can be achieved by alternative solutions that do not incur the same usability penalty, the results of this study suggest that such alternatives are often preferable to factories.

6.1. Finding Factories

Every participant in the study attempted to use a default constructor for SSLSocket, whether or not they had first looked at the documentation for that class. Those who had, and had seen that the constructors were protected, tried them anyway when no other means of creating the object were apparent. Those who had not yet read the documentation, and were engaging in a more exploratory method of programming, fully expected the constructor to succeed, and were puzzled when it did not. Added confusion arose due to the particular error message from the Java compiler: because the SSLSocket class is marked abstract, the error message was "Cannot instantiate the type SSLSocket." This message caused participants to believe they had failed to correctly import the SSLSocket class or had introduced a syntax error in the class name. Indeed, "cannot instantiate the type SSLSocket" was the single most frequently heard comment from our participants, as they repeated it aloud apparently struggling to make sense of it. A more helpful and relevant error message would have been something like "the constructor *SSLSocket()* is protected".

Indeed, participants experienced a strong bias toward trying to find a public subclass of SSLSocket rather than looking for ways of obtaining one indirectly. This was due in part to the Java documentation: the protected constructors for SSLSocket were all listed in that class's documentation, but the description for each read "Used only by subclasses." This phrase, which was often repeated like a mantra by perplexed participants, was universally understood to mean that subclasses must either exist or that the users must create one. One participant made this point explicitly during the debriefing, mentioning, "'Used only by subclasses' makes you want to instantiate subclasses. That's really really confusing." In fact, fully half of the participants (six out of twelve) either expressed their belief that subclassing would be necessary or actually started implementing one before deciding that it would be too much work and looking for another solution.

6.2. Using Factories

Even after discovering the factory, participants were often unable to make immediate progress because in a true abstract factory pattern, the factory itself is also an abstract class. This resulted in much frustration, as expressed by one participant while reading the documentation for SSLSocketFactory: "'Public abstract class'. It extends SocketFactory. It's an abstract class. SSLSocket is an abstract class too. Why is it an abstract class?"

After a close examination of the factory class, the nine participants who finished the task eventually noticed the static *getDefault* factory method that would give them a factory instance. Clearing this hurdle was not sufficient, however, because SSLSocketFactory's *getDefault* method had been overridden from the parent factory class, SocketFactory. Since one cannot change the stated return type of an overridden method, the return value of the *getDefault* method was typed not as an SSLSocketFactory, but as a SocketFactory. Participants were often uncertain whether the instance obtained from *getDefault* was actually an SSLSocketFactory at all, or might simply return generic sockets. Several participants therefore decided that SocketFactory must be a dead end and abandoned it to pursue other possibilities. After discovering this property of the SSLSocketFactory, one participant complained, "So it seems like I can't instantiate an SSLSocket. And it won't tell me who can."

This was also a problem with the *createSocket* methods, as only one *createSocket* method was defined in the SSLSocketFactory subclass, and the ones inherited from the superclass were barely mentioned in the subclass' documentation. This had two deleterious effects. First, participants were misled into thinking that the only method they could use was the one explicitly defined in SSLSocketFactory, which was in fact inapplicable to the situation. Second, the signature of the correct method had to be retrieved from SocketFactory's documentation because only its name was listed on the SSLSocketFactory page, right beside four identically named methods. One participant dryly illustrated this point by reciting off the screen, "'Methods inherited from SocketFactory: createSocket, createSocket, createSocket, createSocket, createSocket.' Sigh."

We were at first surprised that participants were so quick to dismiss SSLSocketFactory, considering that the documentation for SSLSocket explicitly states that SSLSockets are created using SSLSocketFactories. We quickly discovered, however, that the vast majority of users never read that text. It was placed at the bottom of a long class description, under several paragraphs discussing cipher suites and large blocks of sample code. Given the speed at which users scrolled past this class description, it would have been impossible for them to read any more than the first sentence of each paragraph, and many clearly did not even read that much. The lists of fields and methods were of much greater interest, and so most participants' first inkling that something was amiss was the misleading "Used only by subclasses" description for the constructors. Only three participants appeared to actually read the relevant sentence at all; the rest found the SSLSocketFactory class solely by its lexical proximity to SSLSocket in the class list.

Since *createSocket* returned generic Socket objects (which were, in fact, SSLSockets polymorphically typed as their parent class), but the participants needed to call methods specific to SSLSocket on these instances, they were forced to explicitly downcast from Socket to SSLSocket. This "leap of faith" severely eroded participants' confidence in the correctness of their final solution, prompting one to remark, "I don't like doing this. It probably won't work." One participant responded to this requirement with disbelief and said: "You should never have to typecast. If you write programs that require you to typecast, you've either done something wrong or you need to support covariant typing." Another had some words for the folks at Sun, which we shall pass along here: "It's counterintuitive where you have to downcast to something. It's really bad. You should write to the Java people; you should say in your paper, 'get rid of it.'"

These problems with the factory pattern are not limited to the particular implementation in the Sockets task. Indeed, we found similar problems for all designs that used factories. While better documentation would help in the Sockets and Thingies tasks, no amount of documentation would alleviate the puzzlement of users trying to obtain an instance of an abstract class with no known subclasses, nor would documentation remove the need for explicit downcasting (which, as we have seen, is an inherent drawback of abstract factories not shared by alternatives such as class clusters). Adding explicit support for factories into the language or development environment could improve the experience of a user deciphering misleading error messages or trying in vain to find an entry point, but the level of complexity alone was frequently overwhelming in its own right. One participant summarized their experience with the abstract factory pattern with impressive clarity: "I'm trying to figure out how to use these factories. It seems like there's a whole lot of abstract stuff floating around, and I'm not going to be able to actually instantiate anything that I need. In fact, I forgot how I even got here."

Constructors, conversely, posed no problems for any participant in either the Sockets or Thingies task. The most common comment about creating a Flarn was "oh, that should be easy." Participants expressed similar relief in the Sockets task upon seeing that MulticastSocket has constructors; one participant said, "oh good, I can just create one" — implying that obtaining one from a factory was something fundamentally more complex than "just creating one."

We also noticed a tendency on the part of certain participants, especially those who claimed to have relatively little programming experience, to treat factory methods as if they *were* constructors. Five participants were observed calling factory methods and ignoring the return value; all but one eventually added an assignment to the product type. That participant, however, instead called the factory method and then typecast the factory to the product type, as if the factory method had somehow acted as a constructor *post facto* and transformed the factory into the product.

6.3. Debriefing

In the debriefing survey following the last of the tasks, we showed participants two pieces of sample code. Both samples performed the same simple task: adding a border to a panel using Swing. One sample used a BorderFactory, while the other directly constructed the appropriate type of border. We proceeded to ask each participant which approach they felt was "better." We found that participants often voted in favor of the factory pattern, including those participants who had struggled most bitterly with the SSLSocket class. Of the twelve participants, six felt the factory sample was better, whereas five decided in favor of constructors (the twelfth participant's choice was not clear).

The reasons for this, as given by participants, fell into two categories. The first, given by two of those who preferred factories, was the perception that factories hide complexity behind a simple, consistent exterior. Participants felt that "opaque" objects — that is, objects which would be instantiated, passed to another class, and then discarded without being mutated or having methods called on them — should be returned by factories, whereas objects upon whose functionality their code directly depended should be constructed.

The other four participants who preferred factories had a different sort of reasoning behind their preference. Their responses all shared the sense that the developers who designed the APIs must be far more knowledgeable and experienced than they, and therefore any decision made by the API designers must be the better one; factories only appeared more difficult, they reasoned, as a result of some failure on their own part to understand. This reasoning is well summarized by the following comment: "I think that [the constructor example] is easier to understand, and therefore I like it better. However [the factory example] is probably better since it uses a factory and it appears that factories are probably useful in some way." We found no strong relationship between this sort of response and a lack of familiarity with the factory pattern, and neither was this response limited to those with little programming experience; a participant who indicated he had learned about factories extensively in his coursework said, "I like [the factory example] better. I can't quite recall all the benefits of using [the] factory pattern, but I guess from all the training and previous programming experiences I just feel safer and more in control using factories." This individual had struggled just as much with the Sockets task as the other participants.

The seeming contradiction between what some users preferred and what they were best able to use is a common result in human-computer interaction research. Users often cannot identify the solution that is best for them when presented with an explicit choice [16]. For users experienced with the factory pattern, the supposed superiority of the factory was backed up by their formal coursework and it therefore felt "safer." For less experienced users, the very complexity of the factory may have been an appealing feature, as they may have interpreted the complexity of the design as evidence of advanced underlying ideas — a programmer capable of designing and understanding such complexity must be knowledgeable and experienced, the reasoning might go, and therefore is more likely to know best what is good and bad. However, we feel that our results show significant negative impacts on programmers' real ability to use APIs.

7. Future Work

We have focused in this paper on the use of APIs. Future research should explore the similarities and differences between class clusters and factories from the API developer's point of view as well. If class clusters proved to be both easier to use and equally suited to the role currently played by the factory pattern, this could potentially spur a widespread adoption of alternatives to factories in future API designs.

Research could also further examine the relative ease of debugging objects created using factories as opposed to constructors. Additional dimensions could be considered, such as perhaps making a distinction between compile errors and runtime errors, and a larger sample could be gathered.

There are several other design patterns in common use in APIs that should be studied, most notably the singleton pattern, the observer pattern, and the command pattern. Future studies could examine the usability of these patterns, either independently or relative to some alternative design. Research could also be conducted into other common API metaphors such as event handlers, threading models, etc., and their implications for API usability.

Finally, further explorations could be made into reconciling the need for an API that matches the developer's expectations with the need for an API that conforms to the usability guidelines suggested by the cognitive dimensions framework [10][11].

8. Conclusions

Our study finds that the factory pattern erodes the usability of APIs in which it is used. There are alternatives with better usability, such as class clusters, which can be used in many situations in which a factory might normally be used. Since the factory pattern is quite popular with today's API designers, it is important to investigate tradeoffs from the designer's point of view. However, there are thousands of times more people *using* APIs than designing APIs, so designs that degrade API users' productivity should be avoided. Hopefully, there will be many more studies of the impact of API features on programmer productivity, which can guide future API designs.

9. Acknowledgements

We would like to thank Andrew Ko and Justin Weisz for their valuable help with this paper. This work was funded in part by the National Science Foundation, under NSF grant IIS-0329090, and as part of the EUSES consortium (End Users Shaping Effective Software) under NSF grant ITR CCR-0324770. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation.

10. References

[1] A. J. Ko, B. A. Myers, and H. Aung, "Six Learning Barriers in End-User Programming Systems", IEEE Symposium on Visual Languages and Human-Centric Computing, Rome, Italy, Sep 26-29, 2004, 199-206.

[2] S. Clarke, "Measuring API usability", Dr. Dobb's Journal Windows/.NET Supplement, May 2004, pp. S6-S9.

[3] J. Stylos, S. Clarke, and B. A. Myers, "Comparing API Design Choices with Usability Studies: A Case Study and Future Directions", PPIG 2006.

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

[5] G. Florijn, M. Meijers, and P. van Winsen, "Tool Support for Object-Oriented Patterns", Proceedings, ECOOP '97, Springer-Verlag, 1997, pp. 472-495.

[6] ".NET Framework Class Library", *http://msdn2.microsoft.com/en-us/library/ms229335.aspx*

[7] "Java 2 Platform Standard Edition 5.0 API Specification", *http://java.sun.com/j2se/1.5.0/docs/api/index.html*

[8] "Cocoa Fundamentals Guide: Class Clusters", *http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaObjects/chapter_3_section_9.html*

[9] O. Astrachan, G. Mitchener, G. Berry, and L. Cox, "Design patterns: an essential component of CS curricula", SIGCSE Bull. 30, 1, Mar. 1998, pp. 153-160.

[10] T.R.G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework." *Journal of Visual Languages and Computing*, 1996. **7**(2): pp. 131-174.

[11] S. Clarke, "API Usability and the Cognitive Dimensions Framework", 2003, *http://blogs.msdn.com/stevencl/archive/2003/10/08/57040.aspx,*

[12] J. Stylos, S. Clarke, "Usability Implications of Requiring Parameters in Objects' Constructors", to appear in ICSE '07.

[13] "The Eclipse Project", *http://www.eclipse.org*

[14] B.A. Myers, J.F. Pane, and A. Ko, "Natural Programming Languages and Environments." CACM, Sept, 2004. 47(9): pp. 47-52..

[15] F. Modugno, A.T. Corbett, and B.A. Myers, "Evaluating Program Representation in a Demonstrational Visual Shell," in Empirical Studies of Programmers: Sixth Workshop, 1996, Ablex Publishing Corporation, pp. 131-146.

[16] J. Nielsen, *Usability Engineering*, AP Professional, 1993.