The Factory Pattern in API Design
A Usability Evaluation

Brian Ellis, Jeffrey Stylos, and Brad Myers

Carnegie Mellon
I’m going to talk about two different factory patterns today, but I’d like to start with, and focus on, the abstract factory pattern. Most of you are probably already familiar with the abstract factory pattern, but since we’re going to be talking about it quite a lot in the next few minutes, let’s have a look at how the pattern looks to the client–side programmer. Typically when you create an object in a language like Java or C#, you call a constructor of the class you want with the “new” keyword, like you see here. Introducing an abstract factory pattern changes this quite a lot. The two pieces of code you see below here are two ways in which a piece of client code might use an abstract factory to get an instance of a Widget. The first way is to construct a particular concrete factory, then call `createWidget` on it and get back a Widget. That means that the client code has to depend on the concrete factory classes, though, so a more common way is to use the other kind of factory pattern, a factory method, to get an instance to one of the factory subclasses instead. So we’d say `AbstractFactory.getFactoryA`, and then call `createWidget` as before and get back our Widget. Now, this code is considerably more complex than the code the client would need to write to get a Widget using a default constructor, so why would you do it this way?
The Abstract Factory Pattern

- Described in *Design Patterns* by Gamma, Helm, Johnson, and Vlissides (a.k.a, the “Gang of Four”)
- Hides product subclasses
- Permits object caching
- Makes implementation changes transparent to the client
- Decouples the client code from products

Well, the answer, of course, is because the abstract factory pattern provides some architectural advantages over a constructor. The benefits of using an abstract factory were described in the Gang of Four book of design patterns back in 1994. An abstract factory allows client code to get a product of some concrete class, typed as an abstract superclass, without ever depending on that class or even knowing that the class exists. This allows you to use different classes in different circumstances, and return the same instance over and over again, which I’m referring to here as object caching. It also makes changes to the implementation completely transparent to the client, and thereby forces the client to depend only on interfaces rather than the individual implementations. This has the effect of loosening the coupling between the client code and the products, which is quite definitely a good thing.
Let’s look at the architecture of the pattern itself. As you can see, it’s actually fairly complicated: there are six inheritance relationships and a bunch of dependencies. The important thing to notice, though, is that there is a single AbstractFactory class and two concrete factory subclasses, each of which can create some number of different products. Here we have two different products, Widgets and Vidgets, and each concrete factory class creates one specific subtype of each of those, in this case denoted by A and B; so ConcreteFactoryA creates WidgetAs and VidgetAs, et cetera. Each of the product subtypes inherits from an abstract class, in our case Widget and Vidget, and the abstract factory defines methods that return objects typed as those abstract classes.
Factories in API Design

• The factory pattern is used often in public APIs
  • 61 obvious factories in Java 2 SE 5.0 (out of 3,279 classes)
  • 13 obvious factories in .NET (out of 2,686 classes)

• Use in public APIs fundamentally different from internal use
  • Different requirements
  • Different quality attributes

...the factory pattern is used quite often in public APIs such as the Java and .NET frameworks. It’s hard to determine exactly how many factories exist in these frameworks, but simply by looking at class names we can find 61 obvious factories in J2SE 5.0, and 13 in .NET. That doesn’t count classes that happen to use factory methods, just classes whose entire purpose is to be a factory. Now, public APIs like these are intended for mass use, potentially by millions of people. Using any design pattern, or indeed any development paradigm at all, in a public API is a very different thing from using it internally. There are different requirements, different quality attributes, and different metrics for evaluation; the inherent complexity and ease of use of a design pattern is much more important here.
How usable is the abstract factory pattern?

So here is the question: how usable is the abstract factory pattern? Preliminary observation at Microsoft Research suggested that programmers were having trouble using factory patterns, and it’s not hard to imagine that this degree of added complexity I mentioned might be posing a usability problem to client-side developers. We set out to determine whether this was indeed the case, and whether this would be a significant hurdle to the use of the factory pattern, especially after the initial learning period.
One immediate challenge we faced in this was determining exactly what “usability” means for APIs, and how to measure it. There has been some previous research into the elements of API usability, and it turns out there are quite a few: how efficiently can you write code in it? How easy is it to figure out how it works? How easy is it to modify and maintain the code you’ve already written against it? And so on. But if you’re a developer trying to do one of these things, a very reliable metric for how much trouble you’re having with any of them is to measure how long it takes you to get it right. So if we’re careful, we can measure each of these by measuring how long it takes to correctly complete particular tasks involving lots of code creation, code modification, et cetera. All of these numbers are especially important, too, because they’re multiplied by the number of people using the API: if it takes each person an extra minute, and a million people use your API, that’s nearly 17,000 wasted person-hours.
Participants

• Recruited using many diverse sources

• Pre-screened to eliminate unexperienced candidates

We wanted to test a representative sample of people who would actually be using these APIs, and for public APIs like Java and .NET this includes both expert programmers and those who, while still competent, are less experienced with that API. So we recruited our participants in and around Carnegie Mellon university using various methods, from an online study recruitment site to plastering the vicinity with flyers. We used a pre-screening questionnaire to eliminate candidates who had insufficient experience programming in Java, which was the language we selected for the study, and ultimately we ended up with twelve qualified participants.
Participants

- Eight technical students

Eight of them were students in technical disciplines such as computer science,
Participants

- Eight technical students
- Two non-technical students
Participants

- Eight technical students
- Two non-technical students
- Two non-students

and the remaining two were non-students.
Eight of our participants, including both non-students, had some professional software engineering experience.
Participants

- Eight participants with professional software engineering experience
- Four participants without

The other four did not.
Methodology

- Think-aloud protocol
- Time to completion, time spent constructing
- Broad range of tasks
- Randomized condition and task order to account for learning effects

Specifically, we decided to use something called a think-aloud protocol to compare the abstract factory pattern with the use of constructors. This is a very popular and well-studied method in the human–computer interaction community, typically applied to users trying to puzzle out a user interface. The user is given a set of tasks to complete and instructed how to “think aloud,” voicing their thoughts as they work but without attempting to justify or explain them. From this point of view, an API is just another interface, so by observing how the developer interacts with the API we can make good judgments about the specific areas of difficulty the developer has.

In addition to the amount of time the developer needed to complete each of the tasks we gave them, we specifically considered how much of that time was spent constructing objects. We also strove to make the study as broad as possible, so we designed tasks spanning the gamut between using real-world abstract factories encountered in the J2SE API, realistic but artificial tasks involving typical uses of the abstract factory pattern (namely platform-specific interface widgetry), and completely contrived, simplified examples of the pattern in a completely artificial task, involving both code creation and code modification.

And lastly, we randomized which conditions we presented to each participant and in what order they were asked to perform the tasks, in order to account for learning effects.
Task: Email (Notepad)

- Presented with empty Notepad document
- Write a method to create and return a blank email message, either rich-text or plain-text depending on a user preference
- Assume anything desired about the API
- Can be done with either factory or constructor
- Only one condition (no comparison)
- **Goal:** to elicit the programmer’s expectation regarding object creation.

The first task, which was an exception to the random ordering because it was always given at the very beginning, was to simply write some Java code in a Notepad document. We asked participants to write a method that would create and return a blank email message, set to be either rich-text or plain-text depending on a user preference, and to assume absolutely anything they wanted to about the API. We chose email formats because we wanted a task that would be reasonable to tackle with either a factory or a constructor-based approach. We also weren’t looking at things like completion time here, because we had only one condition, so there was no comparison to be made anyhow. What we were really after was to see what the programmer’s expectation would be like regarding how objects would be created.
What we saw was, first, that our participants wrote believable code against believable APIs. We didn’t have anyone write a magic function that they declared by fiat to do the right thing, for example. The breakdown of strategies was interesting: three people used separate subclasses for plain-text and rich-text emails, seven people set whether the email was rich or plain text after constructing a generic Email object, and two passed a boolean in as a constructor parameter. What we didn’t see was anyone using a factory, or indeed anything else other than a constructor. It seemed quite clear that the expectation of the participants was that such an API would use constructors.
Task: Email (Eclipse)

- Given an API to use for the same task as before
  - Documentation provided as Javadoc
- Told to use Eclipse IDE
- API always used abstract factory pattern
  - One condition, no comparison
- **Goal:** to elucidate users’ reactions to finding a factory when a constructor was expected

We also administered a second task related to the first one; we gave participants the same instructions as before, but this time we gave them an API to use and asked them to write the code in Eclipse. Our API used an abstract factory pattern rather than constructors, and since we didn’t have a constructor condition we didn’t measure time to completion here, either. Our goal was related to that of the notepad email task: we wanted to see how users would react to finding a factory when they were expecting a constructor.
Results: Email (Eclipse)

- 10 participants performed task
- All 10 completed the task
- 7 tried a constructor anyway
- 3 of these tried to write concrete subclasses

This task was randomly ordered, so only 10 out of 12 participants started the task, and all ten completed it. Seven people tried to use a constructor, even after seeing that no public constructor was defined. Three people actually tried to write a new concrete subclass of our abstract product class because they couldn’t see any other way to instantiate one.
Task: Thingies

- Create a “Squark” and a “Flarn” and call `run()` on each
- Both subclasses of abstract “Thingy” class
- Squark uses “SquarkFactory”, Flarn uses default constructor
- SquarkFactory is concrete with a public constructor
- Each participant creates both a Squark and a Flarn

**Goal:** to measure user expectation, preference, and responses to both factories and constructors in the absence of any prior domain knowledge

Our third task, again in no particular order, we called Thingies. As you can probably guess from the name, this was our context-free, totally artificial task. Participants were asked to create two subclasses of Thingy, which we called Squark and Flarn, and call a `run` method on each. Squark used a SquarkFactory class, which was a simple concrete class with a public constructor, and Flarn had a public default constructor. Since each participant created both a Squark and a Flarn, this is a within-subjects task, so we did measure time to completion and object construction here. And again, our goal for this task was to compare factories and constructors in the absence of any prior domain knowledge.
Results: Thingies

- 10 participants performed task
- Median time to construct a Squark: 7m 10s
- Median time to construct a Flarn: 1m 20s
- Highly statistically significant

So again, ten participants performed this task, and again all ten finished. The median time it took to construct a Squark, using the SquarkFactory, was 7 minutes and 10 seconds, as compared with 1 minute and 20 seconds to construct a Flarn. This result was highly statistically significant, and I'll go over the details of that here in a few slides.
Task: PIUtils

• Users asked to find and fix a bug in interface code

• Two conditions: code is written either using abstract factory or constructor

• Participants only perform one of two conditions

  • Between-subjects design

• Factory condition uses a factory method to obtain factory instance

• **Goal:** To determine the ease of maintenance of code written using a factory pattern relative to one using constructors

But first, here’s the next task, which we called PIUtils (for platform-independent utilities). The scenario is a classic factory example: cross-platform UI widgetry, with hidden platform-dependent product subclasses accessed using an abstract factory. But we also implemented it using constructors, and presented both conditions. Each participant only saw one or the other condition, making this a between-subjects design instead of a within-subjects design like our other comparative tasks. We also used a factory method pattern in our factory condition rather than exposing concrete factory subclasses. The goal here was to determine the relative ease of maintenance, not just construction, of code written using a factory pattern.
So 9 participants out of the 12 performed this task, but all three of the participants that didn’t had been, by sheer coincidence, placed into the constructor condition, which left us with a rather lopsided distribution, 6 in the factory condition and 3 in the constructor condition. The mean time to completion with a constructor was 26 minutes 40 seconds, as opposed to only 17 minutes even with a factory... but the individual completion times in the factory condition varied four times as much as those in the constructor condition. The range of completion times in the factory condition was about 26 minutes, whereas the range in the constructor condition was about 5. This result turned out not to be statistically significant, and there are quite a few possible reasons for this, including those I just mentioned. But it’s also been shown that debugging and code modification tasks are generally less likely to show statistical differences than creation tasks in studies like this; there were too many confounding variables and too much individual variation to make a strong conclusion.
The last task we designed was called Sockets, and we asked participants to create and pass along an instance of two subclasses of the Java Socket class, SSLSocket and MulticastSocket. This was a beautiful example of an abstract factory pattern right in the Java API, because SSLSocket uses an abstract factory whereas MulticastSocket uses a constructor. We considered this to be a great opportunity to see how users react to factories in an extremely realistic task with a real-life example straight from the Java API.
Results: Sockets

- All twelve participants started task
- Only seven finished
  - All due to failure to complete SSLSocket condition in time
- Median time for SSLSocket: 13m 05s
- Median time for MulticastSocket: 1m 26s
- Highly statistically significant

All twelve participants started this task, but only seven finished. The other five ran out of time, and in every case it was because they couldn’t finish the SSLSocket condition. The median time to construct an SSLSocket was 13 minutes and 5 seconds, compared to 1 minute 26 seconds to construct a MulticastSocket; and not surprisingly, this too was highly statistically significant.
So here is a summary of the quantitative results we obtained from these last three tasks. Here along the x-axis we have our tasks, with factory conditions shown in blue and in the constructor conditions shown in green, and on the y-axis we have time spent constructing objects, though in the PIUtils code modification task we’re measuring total time to completion. [Click.] Here’s the data from the Thingies task, and you can see the difference in construction time; this had a p-value of 0.005. [Click.] Here’s PIUtils, and looking at the factory completion times, you can see the wide range there compared to the constructors. The p-value there was 0.169. And here in the Sockets task [click], you can see that the difference between the conditions is even more pronounced than for Thingies. In fact, the median time spent constructing objects in the factory condition is roughly thirteen times that spent in the constructor condition. Now, these results show very clearly that the factory pattern was less usable than constructors in code creation tasks, but this still leaves open a very important question: why?
Results

So here is a summary of the quantitative results we obtained from these last three tasks. Here along the x-axis we have our tasks, with factory conditions shown in blue and in the constructor conditions shown in green, and on the y-axis we have time spent constructing objects, though in the PIUtils code modification task we’re measuring total time to completion. [Click.] Here’s the data from the Thingies task, and you can see the difference in construction time; this had a p-value of 0.005. [Click.] Here’s PIUtils, and looking at the factory completion times, you can see the wide range there compared to the constructors. The p-value there was 0.169. And here in the Sockets task [click], you can see that the difference between the conditions is even more pronounced than for Thingies. In fact, the median time spent constructing objects in the factory condition is roughly thirteen times that spent in the constructor condition. Now, these results show very clearly that the factory pattern was less usable than constructors in code creation tasks, but this still leaves open a very important question: why?
Results

So here is a summary of the quantitative results we obtained from these last three tasks. Here along the x-axis we have our tasks, with factory conditions shown in blue and in the constructor conditions shown in green, and on the y-axis we have time spent constructing objects, though in the PIUtils code modification task we’re measuring total time to completion. [Click.]

Here’s the data from the Thingies task, and you can see the difference in construction time; this had a p-value of 0.005. [Click.]

Here’s PIUtils, and looking at the factory completion times, you can see the wide range there compared to the constructors. The p-value there was 0.169. And here in the Sockets task [click], you can see that the difference between the conditions is even more pronounced than for Thingies. In fact, the median time spent constructing objects in the factory condition is roughly thirteen times that spent in the constructor condition. Now, these results show very clearly that the factory pattern was less usable than constructors in code creation tasks, but this still leaves open a very important question: why?
So here is a summary of the quantitative results we obtained from these last three tasks. Here along the x-axis we have our tasks, with factory conditions shown in blue and in the constructor conditions shown in green, and on the y-axis we have time spent constructing objects, though in the PIUtils code modification task we’re measuring total time to completion. [Click.] Here’s the data from the Thingies task, and you can see the difference in construction time; this had a p-value of 0.005. [Click.] Here’s PIUtils, and looking at the factory completion times, you can see the wide range there compared to the constructors. The p-value there was 0.169. And here in the Sockets task [click], you can see that the difference between the conditions is even more pronounced than for Thingies. In fact, the median time spent constructing objects in the factory condition is roughly thirteen times that spent in the constructor condition. Now, these results show very clearly that the factory pattern was less usable than constructors in code creation tasks, but this still leaves open a very important question: why?
To address that question, I’d like to go into a bit more depth about one task in particular: Sockets. As I said before, MulticastSocket is a subclass of Socket that’s created using a constructor, and SSLSocket is a subclass of Socket that uses a classic abstract factory pattern. MulticastSocket has a public default constructor, but SSLSocket’s constructors are all protected. To construct an SSLSocket, you call a createSocket() method on an instance of SSLSocketFactory, which is a subclass of the abstract SocketFactory class. SocketFactory, though abstract, has a static getDefault() method, which as we saw before uses the factory method pattern.
To address that question, I'd like to go into a bit more depth about one task in particular: Sockets. As I said before, MulticastSocket is a subclass of Socket that’s created using a constructor, [click] and SSLSocket is a subclass of Socket that uses a classic abstract factory pattern. [Click.] MulticastSocket has a public default constructor, but SSLSocket's constructors are all protected. To construct an SSLSocket, you call a createSocket() method on an instance of SSLSocketFactory, which is a subclass of the abstract SocketFactory class. SocketFactory, though abstract, has a static getDefault() method, which as we saw before uses the factory method pattern.
To address that question, I'd like to go into a bit more depth about one task in particular: Sockets. As I said before, MulticastSocket is a subclass of Socket that’s created using a constructor, [click] and SSLSocket is a subclass of Socket that uses a classic abstract factory pattern. [Click.] MulticastSocket has a public default constructor, but SSLSocket's constructors are all protected. To construct an SSLSocket, you call a createSocket() method on an instance of SSLSocketFactory, which is a subclass of the abstract SocketFactory class. SocketFactory, though abstract, has a static getDefault() method, which as we saw before uses the factory method pattern.
To address that question, I’d like to go into a bit more depth about one task in particular: Sockets. As I said before, MulticastSocket is a subclass of Socket that’s created using a constructor, [click] and SSLSocket is a subclass of Socket that uses a classic abstract factory pattern. [Click.] MulticastSocket has a public default constructor, but SSLSocket’s constructors are all protected. To construct an SSLSocket, you call a createSocket() method on an instance of SSLSocketFactory, which is a subclass of the abstract SocketFactory class. SocketFactory, though abstract, has a static getDefault() method, which as we saw before uses the factory method pattern.
The Sockets Task

- Constructing an SSLSocket:

So let’s see what code we have to write to complete this task. First we need to get an SSLSocketFactory. As we’ve seen, the way to do that is with the getDefault() method [click], which is inherited from SocketFactory and overridden. But that won’t work [click], because the return type of getDefault() is still SocketFactory.
The Sockets Task

• Constructing an SSLSocket:

```java
SSLSocketFactory factory = SSLSocketFactory.getDefault();
```
The Sockets Task

- Constructing an SSLSocket:

```java
SSLsocketFactory factory = SSLsocketFactory.getDefault();
```
The Sockets Task

- Constructing an SSLSocket:

```java
SocketFactory factory = SSLSocketFactory.getDefault();
```

So we’ll change that to just SocketFactory, and now we have to remember that since SSLSocketFactory overrides getDefault, factory is actually an SSLSocketFactory. Okay, so now we have to get our SSLSocket from the factory. [Click.] So we’ll call createSocket() on the factory. [Click.] Except that won’t work either, because createSocket() is also inherited from SocketFactory, and therefore it returns a Socket, not an SSLSocket. To fix this [click], we’ll have to downcast the return value of createSocket() to an SSLSocket, making the assumption that SSLSocketFactory has overridden createSocket() to return an SSLSocket instance instead of just a generic Socket. Now, you might be wondering why we can’t just specify the type of s as “Socket” and not bother with the cast; the answer is that SSLSocket has several methods that aren’t inherited from Socket, so if you want to call any of those [click], the compiler has to be convinced that what you’ve got is actually an SSLSocket, and the only way to do that is to downcast. In the Sockets task, the participants are asked to pass the SSLSocket object they’ve created to a method we wrote that will connect the socket for them, so the downcast is necessary in the task as well.
• Constructing an SSLSocket:

```java
SocketFactory factory = SSLSocketFactory.getDefault();
SSLSocket s = factory.createSocket();
```

So we'll change that to just SocketFactory, and now we have to remember that since SSLSocketFactory overrides getDefault, factory is actually an SSLSocketFactory. Okay, so now we have to get our SSLSocket from the factory. [Click.] So we'll call createSocket() on the factory. [Click.] Except that won't work either, because createSocket() is also inherited from SocketFactory, and therefore it returns a Socket, not an SSLSocket. To fix this [click], we'll have to downcast the return value of createSocket() to an SSLSocket, making the assumption that SSLSocketFactory has overridden createSocket() to return an SSLSocket instance instead of just a generic Socket. Now, you might be wondering why we can't just specify the type of s as “Socket” and not bother with the cast; the answer is that SSLSocket has several methods that aren't inherited from Socket, so if you want to call any of those [click], the compiler has to be convinced that what you've got is actually an SSLSocket, and the only way to do that is to downcast. In the Sockets task, the participants are asked to pass the SSLSocket object they've created to a method we wrote that will connect the socket for them, so the downcast is necessary in the task as well.
The Sockets Task

• Constructing an SSLSocket:

```
SocketFactory factory = SSLSocketFactory.getDefault();
SSLSocket s = factory.createSocket();
```

So we’ll change that to just SocketFactory, and now we have to remember that since SSLSocketFactory overrides getDefault,
factory is actually an SSLSocketFactory. Okay, so now we have to get our SSLSocket from the factory. [Click.] So we’ll call
createSocket() on the factory. [Click.] Except that won’t work either, because createSocket() is also inherited from SocketFactory,
and therefore it returns a Socket, not an SSLSocket. To fix this [click], we’ll have to downcast the return value of createSocket()
to an SSLSocket, making the assumption that SSLSocketFactory has overridden createSocket() to return an SSLSocket instance
instead of just a generic Socket. Now, you might be wondering why we can’t just specify the type of s as “Socket” and not bother
with the cast; the answer is that SSLSocket has several methods that aren’t inherited from Socket, so if you want to call any of
those [click], the compiler has to be convinced that what you’ve got is actually an SSLSocket, and the only way to do that is to
downcast. In the Sockets task, the participants are asked to pass the SSLSocket object they’ve created to a method we wrote
that will connect the socket for them, so the downcast is necessary in the task as well.
The Sockets Task

- Constructing an SSLSocket:

```java
SocketFactory factory = SSLSocketFactory.getDefault();
SSLSocket s = ((SSLSocket)factory.createSocket());
```

So we’ll change that to just SocketFactory, and now we have to remember that since SSLSocketFactory overrides getDefault, factory is actually an SSLSocketFactory. Okay, so now we have to get our SSLSocket from the factory. [Click.] So we’ll call createSocket() on the factory. [Click.] Except that won’t work either, because createSocket() is also inherited from SocketFactory, and therefore it returns a Socket, not an SSLSocket. To fix this [click], we’ll have to downcast the return value of createSocket() to an SSLSocket, making the assumption that SSLSocketFactory has overridden createSocket() to return an SSLSocket instance instead of just a generic Socket. Now, you might be wondering why we can’t just specify the type of s as “Socket” and not bother with the cast; the answer is that SSLSocket has several methods that aren’t inherited from Socket, so if you want to call any of those [click], the compiler has to be convinced that what you’ve got is actually an SSLSocket, and the only way to do that is to downcast. In the Sockets task, the participants are asked to pass the SSLSocket object they’ve created to a method we wrote that will connect the socket for them, so the downcast is necessary in the task as well.
• Constructing an SSLSocket:

```java
SocketFactory factory = SSLsocketFactory.getDefault();
SSLSocket s = ((SSLSocket)factory.createSocket());
SSLSession session = s.getSession();
```

So we’ll change that to just SocketFactory, and now we have to remember that since SSLSocketFactory overrides getDefault, factory is actually an SSLSocketFactory. Okay, so now we have to get our SSLSocket from the factory. [Click.] So we’ll call createSocket() on the factory. [Click.] Except that won’t work either, because createSocket() is also inherited from SocketFactory, and therefore it returns a Socket, not an SSLSocket. To fix this [click], we’ll have to downcast the return value of createSocket() to an SSLSocket, making the assumption that SSLSocketFactory has overridden createSocket() to return an SSLSocket instance instead of just a generic Socket. Now, you might be wondering why we can’t just specify the type of s as “Socket” and not bother with the cast; the answer is that SSLSocket has several methods that aren’t inherited from Socket, so if you want to call any of those [click], the compiler has to be convinced that what you’ve got is actually an SSLSocket, and the only way to do that is to downcast. In the Sockets task, the participants are asked to pass the SSLSocket object they’ve created to a method we wrote that will connect the socket for them, so the downcast is necessary in the task as well.
Now, that was fairly complicated to walk through just now, even with a complete and full understanding of the object model and the knowledge that this is an abstract factory pattern. We didn’t warn our participants that they were going to run into any of this, however, just as a typical developer wouldn’t know if they were just interested in making an SSL-enabled socket in Java, so they just tried to create an SSLSocket instance using a default constructor [click], like this. This was very interesting to note, because every single participant, regardless of their level of expertise or experience, wrote this line of code. Some looked at the documentation first, some didn’t, but all of them wrote this. We speculate, based on participants’ behavior during the constructor conditions, that the intent here was to first construct the object and then use autocomplete to figure out what methods they could call on it. In any event, the error message they got when they wrote this line was this: [click] cannot instantiate the type SSLSocket. Now we know that this is because SSLSocket is just an abstract superclass for the platform-specific native SSL socket subclasses, and so you can’t directly instantiate one, but the participants were utterly confused by this error message. The most common utterance we heard from our participants throughout the entire study was these five words, repeated over and over again as they tried to make sense of them.
The Sockets Task

- Our participants’ approach:

```java
SSLSocket s = new SSLSocket();
```

Now, that was fairly complicated to walk through just now, even with a complete and full understanding of the object model and the knowledge that this is an abstract factory pattern. We didn’t warn our participants that they were going to run into any of this, however, just as a typical developer wouldn’t know if they were just interested in making an SSL-enabled socket in Java, so they just tried to create an SSLSocket instance using a default constructor [click], like this. This was very interesting to note, because every single participant, regardless of their level of expertise or experience, wrote this line of code. Some looked at the documentation first, some didn’t, but all of them wrote this. We speculate, based on participants’ behavior during the constructor conditions, that the intent here was to first construct the object and then use autocomplete to figure out what methods they could call on it. In any event, the error message they got when they wrote this line was this: [click] cannot instantiate the type SSLSocket. Now we know that this is because SSLSocket is just an abstract superclass for the platform-specific native SSL socket subclasses, and so you can’t directly instantiate one, but the participants were utterly confused by this error message. The most common utterance we heard from our participants throughout the entire study was these five words, repeated over and over again as they tried to make sense of them.
Our participants’ approach:

```java
SSLSocket s = new SSLSocket();
```

Cannot instantiate the type SSLSocket.
Finding the Factory

handshaking process, as well as which messages should be sent by each party. Each connection must have one client and one server, or handshaking will not progress properly. Once the initial handshaking has started, a socket can not switch between client and server modes, even when performing renegotiations.

Since:
1.4
See Also:
Socket, SSLServerSocket, SSLSocketFactory

Constructor Summary

<table>
<thead>
<tr>
<th>protected</th>
<th>SSLSocket ()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>protected</th>
<th>SSLSocket (InetAddress address, int port)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>protected</th>
<th>SSLSocket (InetAddress address, int port, InetAddress clientAddress, int clientPort)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>protected</th>
<th>SSLSocket (String host, int port)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>protected</th>
<th>SSLSocket (String host, int port, InetAddress clientAddress, int clientPort)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

Eventually, all the participants went to look at the SSLSocket documentation to find out why they couldn’t instantiate one. And every single participant skipped the paragraphs of text describing the function of the class, and how to set up cipher suites, and went straight to the constructor summary, where they quickly discovered that all the constructors are protected and marked as “used only by subclasses.” A little under half of the participants actually tried to implement a new subclass of SSLSocket at this point, though they gave up fairly quickly once they realized how much work would be involved in that.
Finding the Factory

handshaking process, as well as which messages should be sent by each party. Each connection must have one client and one server, or handshaking will not progress properly. Once the initial handshaking has started, a socket can not switch between client and server modes, even when performing renegotiations.

Since:
1.4
See Also:
Socket, SSLServerSocket, SSLSocketFactory

---

### Constructor Summary

<table>
<thead>
<tr>
<th>Protected</th>
<th>SSLSocket()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Used only by subclasses.</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protected</th>
<th>SSLSocket(InetAddress address, int port)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protected</th>
<th>SSLSocket(InetAddress address, int port, InetAddress clientAddress, int clientPort)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protected</th>
<th>SSLSocket(String host, int port)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protected</th>
<th>SSLSocket(String host, int port, InetAddress clientAddress, int clientPort)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

---

Eventually, all the participants went to look at the SSLSocket documentation to find out why they couldn’t instantiate one. And every single participant skipped the paragraphs of text describing the function of the class, and how to set up cipher suites, and went straight to the constructor summary, where they quickly discovered that all the constructors are protected and marked as “used only by subclasses.” A little under half of the participants actually tried to implement a new subclass of SSLSocket at this point, though they gave up fairly quickly once they realized how much work would be involved in that.
Finding the Factory

You may register to receive event notification of handshake completion. This involves the use of two additional classes. `HandshakeCompletedEvent` objects are passed to `HandshakeCompletedListener` instances, which are registered by users of this API. SSLSockets are created by `SSLSocketFactory`s, or by accepting a connection from a `SSLServerSocket`.

A SSL socket must choose to operate in the client or server mode. This will determine who begins the handshaking process, as well as which messages should be sent by each party. Each connection must have one client and one server, or handshaking will not progress properly. Once the initial handshaking has started, a socket can not switch between client and server modes, even when performing renegotiations.

**Since:**
1.4

**See Also:**
[Socket](#), [SSLSocket](#), [SSLSocketFactory](#)

### Constructor Summary

<table>
<thead>
<tr>
<th>Access Modifier</th>
<th>Method Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected</td>
<td><code>SSLSocket()</code></td>
<td>Used only by subclasses.</td>
</tr>
<tr>
<td>protected</td>
<td><code>SSLSocket(InetAddress address, int port)</code></td>
<td>Used only by subclasses.</td>
</tr>
<tr>
<td>protected</td>
<td><code>SSLSocket(InetAddress address, int port, InetAddress clientAddress, int clientPort)</code></td>
<td></td>
</tr>
</tbody>
</table>

Now, the documentation is actually quite clear on how to construct an SSLSocket. It's in this paragraph above the constructor summary, and it's also in the “See Also” list. But users typically didn't look at either of those things, because they were still thinking in terms of constructors.
Finding the Factory

You may register to receive event notification of handshake completion. This involves the use of two additional classes. HandshakeCompletedEvent objects are passed to HandshakeCompletedListener instances, which are registered by users of this API. SSLSockets are created by SSLSocketFactories, or by accepting a connection from a SSLServerSocket.

A SSL socket must choose to operate in the client or server mode. This will determine who begins the handshaking process, as well as which messages should be sent by each party. Each connection must have one client and one server, or handshaking will not progress properly. Once the initial handshaking has started, a socket can not switch between client and server modes, even when performing renegotiations.

Since:
1.4
See Also:
Socket, SSLServerSocket, SSLSocketFactory

Constructor Summary

<table>
<thead>
<tr>
<th>protected</th>
<th>SSLSocket()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>protected</th>
<th>SSLSocket(InetAddress address, int port)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Used only by subclasses.</td>
</tr>
</tbody>
</table>

| protected | SSLSocket(InetAddress address, int port, InetAddress clientAddress, int clientPort) |

Now, the documentation is actually quite clear on how to construct an SSLSocket. It's in this paragraph above the constructor summary, and it's also in the “See Also” list. But users typically didn't look at either of those things, because they were still thinking in terms of constructors.
Using the Factory

Class SSLSocketFactory

```java
public abstract class SSLSocketFactory
extends SocketFactory

SSLFactory creates SSLSockets.
```

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>abstract <code>Socket</code></td>
</tr>
<tr>
<td>static <code>SocketFactory</code></td>
</tr>
<tr>
<td>abstract <code>String[]</code></td>
</tr>
<tr>
<td>abstract <code>String[]</code></td>
</tr>
</tbody>
</table>

Methods inherited from class `javax.net.SocketFactory`:
`createSocket, createSocket, createSocket, createSocket, createSocket`

Once they found the SSLSocketFactory, in many cases simply by scrubbing the class list looking for classes lexically similar to SSLSocket that looked like they might be helpful, they discovered that SSLSocketFactory is also an abstract class. At this point they had already been struggling for several minutes to figure out how to instantiate an abstract class, so to be presented with another one tended to be very disheartening. In this case, the solution was different as well: there is, thank goodness, no SSLSocketFactoryFactory. Instead, there’s the factory method, `getDefault`. We saw that participants took nearly as long to find and make use of this as they had to find the SSLSocketFactory class itself, which suggests at least anecdotally that the factory method pattern may be susceptible to the same problem as the abstract factory pattern, especially when used in conjunction with it.
Using the Factory

Class SSLSocketFactory

```java
java.lang.Object
  ↓ javax.net.SocketFactory
     ↓ javax.net.ssl.SSLSocketFactory

public abstract class SSLSocketFactory
  extends SocketFactory
```

SSLSocketFactory creates SSL sockets.

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createSocket(Socket s, String host, int port, boolean autoClose)</code></td>
<td>Returns a socket layered over an existing socket connected to the named host, at the given port.</td>
</tr>
<tr>
<td><code>getDefault()</code></td>
<td>Returns the default SSL socket factory.</td>
</tr>
<tr>
<td><code>getDefaultCipherSuites()</code></td>
<td>Returns the list of cipher suites which are enabled by default.</td>
</tr>
<tr>
<td><code>getSupportedCipherSuites()</code></td>
<td>Returns the names of the cipher suites which could be enabled for use on an SSL connection.</td>
</tr>
</tbody>
</table>

Methods inherited from class `javax.net.SocketFactory`

- `createSocket`, `createSocket`, `createSocket`, `createSocket`, `createSocket`
Using the Factory

Class SSLSocketFactory

```java
java.lang.Object
  javax.net.SocketFactory
    javax.net.ssl.SSLSocketFactory
```

```java
public abstract class SSLSocketFactory
extends SocketFactory

SSL Socket Factory s create SSL Sockets.
```

Method Summary

<table>
<thead>
<tr>
<th>abstract Socket</th>
<th>createSocket(Socket s, String host, int port, boolean autoClose)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns a socket layered over an existing socket connected to</td>
</tr>
<tr>
<td></td>
<td>the named host, at the given port.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>static SocketFactory</th>
<th>getDefault()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns the default SSL socket factory.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>abstract String[]</th>
<th>getDefaultCipherSuites()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns the list of cipher suites which are enabled by default.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>abstract String[]</th>
<th>getSupportedCipherSuites()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Returns the names of the cipher suites which could be enabled for use on an SSL connection.</td>
</tr>
</tbody>
</table>

Methods inherited from class javax.net.SocketFactory

`createSocket`, `createSocket`, `createSocket`, `createSocket`, `createSocket`
Making a MulticastSocket

- Constructing a MulticastSocket:

Now, let’s compare this to constructing a MulticastSocket. Here, the participants’ first instinct [click], to simply call the default constructor and assign to an object, pays off [click]. Here’s a comment from one of our participants [click]: “oh good, I can just create one.” This participant had just finished waging epic battle with SSLSocket, and I find this quote interesting because it seems to imply that at least in his mind, going through a factory was something very different than “just creating one.”
Making a MulticastSocket

• Constructing a MulticastSocket:

```java
MulticastSocket s = new MulticastSocket();
```

Now, let’s compare this to constructing a MulticastSocket. Here, the participants’ first instinct [click], to simply call the default constructor and assign to an object, pays off [click]. Here’s a comment from one of our participants [click]: “oh good, I can just create one.” This participant had just finished waging epic battle with SSLSocket, and I find this quote interesting because it seems to imply that at least in his mind, going through a factory was something very different than “just creating one.”
Making a MulticastSocket

• Constructing a MulticastSocket:

    MulticastSocket s
    = new MulticastSocket();

Now, let’s compare this to constructing a MulticastSocket. Here, the participants’ first instinct [click], to simply call the default constructor and assign to an object, pays off [click]. Here’s a comment from one of our participants [click]: “oh good, I can just create one.” This participant had just finished waging epic battle with SSLSocket, and I find this quote interesting because it seems to imply that at least in his mind, going through a factory was something very different than “just creating one.”
Making a MulticastSocket

• Constructing a MulticastSocket:

```java
MulticastSocket s = new MulticastSocket();
```

“Oh good, I can just create one.”
After all the tasks were over, we showed participants two nearly identical code snippets, one of which used a factory and the other which used a constructor, and asked them which approach they thought was better. The responses were split nearly perfectly down the middle. Of the eleven responses that expressed a clear preference, five people preferred the constructor, and six preferred the factory. What’s interesting is the reasons people gave for preferring one over the other. Everyone who preferred the constructor cited [click] simplicity and ease of use, but the reasons for choosing the factory pattern fell into two categories. [Click.] Two users felt that since factories can hide complex object hierarchies behind a consistent exterior, they are better for “opaque” types — that is, those which are never operated upon directly by the client programmer, but are simply created and passed to other objects. Considering that the example we showed used precisely such an object, a Java Swing Border, that certainly seems like a reasonable justification, and indeed that could be an advantage of factories. The other four participants, however, took the stance that there must be some benefit of factories that they simply don’t understand, since the designers of the API used the factory pattern, and they must be very smart indeed.
Debriefing

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Factory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Constructor
  - Simplicity and ease of use (5)

After all the tasks were over, we showed participants two nearly identical code snippets, one of which used a factory and the other which used a constructor, and asked them which approach they thought was better. The responses were split nearly perfectly down the middle. Of the eleven responses that expressed a clear preference, five people preferred the constructor, and six preferred the factory. What’s interesting is the reasons people gave for preferring one over the other. Everyone who preferred the constructor cited simplicity and ease of use, but the reasons for choosing the factory pattern fell into two categories. Two users felt that since factories can hide complex object hierarchies behind a consistent exterior, they are better for “opaque” types — that is, those which are never operated upon directly by the client programmer, but are simply created and passed to other objects. Considering that the example we showed used precisely such an object, a Java Swing Border, that certainly seems like a reasonable justification, and indeed that could be an advantage of factories. The other four participants, however, took the stance that there must be some benefit of factories that they simply don’t understand, since the designers of the API used the factory pattern, and they must be very smart indeed.
Debriefing

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Factory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Constructor
  - Simplicity and ease of use (5)

- Factory
  - Hide complexity behind consistent exterior, so better for opaque types (like Border) (2)
  - API designers know best; I must be missing something. (4)

After all the tasks were over, we showed participants two nearly identical code snippets, one of which used a factory and the other which used a constructor, and asked them which approach they thought was better. The responses were split nearly perfectly down the middle. Of the eleven responses that expressed a clear preference, five people preferred the constructor, and six preferred the factory. What’s interesting is the reasons people gave for preferring one over the other. Everyone who preferred the constructor cited simplicity and ease of use, but the reasons for choosing the factory pattern fell into two categories. Two users felt that since factories can hide complex object hierarchies behind a consistent exterior, they are better for “opaque” types — that is, those which are never operated upon directly by the client programmer, but are simply created and passed to other objects. Considering that the example we showed used precisely such an object, a Java Swing Border, that certainly seems like a reasonable justification, and indeed that could be an advantage of factories. The other four participants, however, took the stance that there must be some benefit of factories that they simply don’t understand, since the designers of the API used the factory pattern, and they must be very smart indeed.
What Can We Do?

• So the abstract factory pattern has major usability issues... but sometimes there’s no choice!
  • Hidden subclasses
  • Object caching
• How can API designers get these without factories?

So we’ve seen that the abstract factory pattern has some major usability issues, and we have some clues as to why. But you may feel that it’s the only thing you can use in many cases, such as when you need hidden subclasses or object reuse. Which leads one to wonder, how can API designers get these things without having to use a factory?
An Alternative

- The “class cluster” pattern
  - Adapted from Objective-C
- Permits hidden subclasses
  - Uses true subclassing, just like abstract factory
- Permits object caching (mostly)
- Method of construction:
  - MyClass c = new MyClass();

Well, there is an alternative to the abstract factory pattern. Actually there are many, but here’s an interesting one. It’s called a "class cluster," and it’s borrowed from the Objective-C language. Objective-C is more like Smalltalk than Java, so there’s no concept of a constructor at all, but we can adapt this to a language like Java or C# without much trouble. It permits hidden subclasses, just like the abstract factory, and the subclasses are true subclasses of the public superclass, not just composed classes or delegates. It also mostly permits object caching. The nice thing about it is that the way you construct something in a class cluster is just call its constructor, so it imposes absolutely no added burden on the end-user developer.
The class cluster pattern can be implemented in a language like Java like this. Here we have our public superclass, Widget, and two package-protected subclasses, WidgetA and WidgetB. We put a method call in the constructor for Widget that determines which subclass we should create, but we could easily decide that based on any other factor we like, or have multiple constructors that each instantiate a different subclass. Then we just assign a new instance of our subclass to our body field, and delegate our other method calls to the body as necessary. If we want to reuse objects, we just store a static reference or object pool in our Widget class. The Widget is still constructed every time using the new operator, which is why this only mostly permits object reuse, but the subclasses can be reused, and you can still use a factory method or singleton pattern to get your reference to the superclass instance if that’s a big concern. This still has many of the disadvantages of the factory pattern from the API designer’s point of view: it depends on its own subclasses, for example. But from the client-side developer’s point of view, it’s just an object. It can be constructed like any other object, you can use autocomplete to explore its methods, and not a single class is marked abstract. If you’re looking for an alternative to an abstract factory, you might consider this.
Future Work

- The usability aspects of other API design choices
  - Singleton
  - Observer
- Other API metaphors
  - event handlers
  - threading models

All of this leaves a lot of questions open, as well, and many of these could be avenues for future work. In particular, there are many other design patterns whose usability have never been formally evaluated. The singleton, observer, and command patterns in particular seem especially suitable for this sort of evaluation. Outside the realm of design patterns completely, it would be interesting to study other API metaphors from a usability perspective, such as event handlers and threading models.
Acknowledgements

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.
A Word on Documentation

- Better documentation *would* help in many cases.
- Factory pattern is inherently complex.
  - Abstract classes with no obvious subclasses.
  - Does not use “natural” construction methods.
  - Is not discoverable using autocomplete, etc.
- Neither better documentation nor even IDE/language-level support for factories will solve all these problems.
A Word on Documentation

- Better documentation *would* help in many cases.
- Factory pattern is inherently complex.

“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.”