Making APIs More Usable with Improved API Designs, Documentation and Tools

Jeffrey Stylos

May 2009
CMU-CS-09-130

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Brad A. Myers (Chair)
Jonathan Aldrich
David Garlan
Steven Clarke (Microsoft)

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

This work was funded in part by a grant from SAP, and in part by the National Science Foundation, under NSF grants IIS-0329090, IIS-0757511 and CCF-0811610, 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 thesis are those of the author and do not necessarily reflect those of the National Science Foundation.

Keywords: Application Programming Interfaces (APIs), libraries, frameworks, Software Development Kits (SDKs), usability, documentation, natural programming, user studies, integrated development environments (IDEs), Java, Eclipse, design patterns.

ABSTRACT

An important and challenging programming activity is using application programming interfaces (APIs), frameworks, toolkits and libraries. Programmers implementing new functionality need to determine which APIs to use and how to combine them. Programmers reading or modifying code need to understand how existing code that calls APIs works, what assumptions the code makes, and how to change or add to the code without breaking these assumptions. These tasks are increasingly challenging for programmers with the growing size and complexity of APIs; the Java 1.5 JDK, for example, has more than 4,000 classes and more than 30,000 methods. API designers have a challenging task in creating and maintaining these APIs, and they must make many different API design decisions that may affect the usability of APIs in unexpected ways.

This thesis presents ways to improve the usability of the APIs and programming tools programmers use by examining the decisions that API designers make and performing studies comparing design alternatives. It provides evidence that more usable APIs and tools can make programmers faster and can more closely match programmers' mental models. The studies inform recommendations and highlight usability tradeoffs that provide API designers with an empirical basis for the design of usable APIs. The studies also provide insight into how programmers work and how programming tools should be designed.

I first mapped out the space of API design decisions. I examined three API design decisions relating to object creation using constructors and factories, and method placement. Based on the results of these studies, I also designed, implemented and evaluated new API documentation and programming tools.

To Noam, for the high point of the last six years

ACKNOWLEDGMENTS

I would like to thank my colleagues, whose collaboration and help made this research possible. I would also like to thank all 32 of my classmates who began the CMU CSD PhD program in the Fall of 2003: Aditya Ganjam, Chris Wang, Colin McMillen, Daniel Golovin, David Brumley, Deepak Garg, Elisabeth Crawford, Farokh Jamalyaria, Himanshu Jain, Indrayana Rustandi, Jahanzeb Sherwani, James Hays, Jeffrey Pang, Jimeng Sun, Jing Chen, Jonathan Derryberry, Jordan Wales, Justin Weisz, Kevin Bowers, Manfred Lau, Matthew Streeter, Neel Krishnaswami, Noam Zeilberger, Pratyusa Manadhata, Runting Shi, Sandeep Pandey, Sonia Chernova, Srinath Sridhar, Virginia Vassilevska, Vyas Sekar, Yang Gu, and Yong Lu. We are, and will always be, The Thirty-Three. Finally, I would like to thank the Squirrel Hill Giant Eagle for beginning 24-hour service during my stay here in Pittsburgh.

TABLE OF CONTENTS

1.

INTRODUCTION*

Programming is the act of telling a computer what to do, in a manner that allows the program to be re-executed later, and is typically performed using one or more textual programming languages. Programming is often difficult. There are many reasons for this, and some of these difficulties are likely intrinsic to the task. However, other difficulties arise from specific details of the programming language and tools that programmers use, and it might be possible to significantly reduce these difficulties. One large source of programming difficulty on which we might have a large impact is that of using APIs.

1.1. WHAT IS AN API?

The term API stands for **Application Programming Interface** and represents, at a high level, a mechanism for code reuse. Code reuse allows programmers to build on top of the work that other programmers (or they themselves) have already done, rather than starting from scratch with every program. The two main forms of code reuse are directly copying code and placing it in one's program — a copy and paste style of reuse — or by calling functions that are packaged up and explicitly intended to be reused. These functions are referred to as libraries, frameworks, or generically in this thesis as APIs. With APIs, programmers can reuse code without needing to modify, understand or even see the implementation, and instead interacting only with the programming interface.

* Portions of this chapter previously appeared in [Stylos 2006b].

Libraries	Math library, “standard” library in C
Frameworks	.NET Framework, Eclipse Framework
Software Development Kits	.NET Development Kit, Java Development Kit
Toolkits	The GIMP Toolkit (GTK), Google Web Toolkit
APIs	Win32 APIs, Google Map APIs

Table 3.1. Different API-related terms and examples for each. In this thesis, we use “APIs” to refer to all of these together.

APIs can enable programmers to create remarkably sophisticated programs with a relatively small amount of work. This is in part because APIs can provide access to large amounts of functionality at a very high abstraction level. For example, using a web-browser widget it is possible to embed an entire web-browser with a single line of code, rather than the tens or hundreds of thousands of lines of code it would take to create this functionality from scratch. This power comes with a price, however, as APIs often limit what programmers can do, and often impose non-obvious constraints on how they need to use the APIs to get a working program. Some APIs, however, seem to be much more difficult to use than others, even when the two APIs provide similar functionality at the same abstraction level. This provides hope that by learning how to make APIs more usable, we can provide powerful yet easy to use abstractions, and make the task of programming easier.

1.1.1. What Is *Not* an API?

The terms above are inclusive enough that it merits asking what they are not used to describe.

A programming language is not an API. This is perhaps contentious and it can be difficult to separate the two (the Java APIs are integral to the Java language), but for our definitions a language includes a syntax and compiler or interpreter, while an API is always built on a language.

A tool is not an API. An API consists of binaries, definition files and documentation, but not binary applications. This is potentially tricky when the tools are required to create or edit resources used by any API.

Documentation is not, by itself, an API, though it can be essential to an API's usability.

Example code is not an API, although it may be included in documentation, and is indeed often the primary or only means of documentation.

An application's source code is not an API. If you are creating code that will be used by only one application, then it is not an API.

1.2. The Problem

The use of APIs is of growing importance to programmers. One reason for this is that the programs that programmers are creating are also increasing in complexity, both in terms of overall scope and specific details. Word processors, for example, have evolved from simple text editors to powerful document development environments with thousands of features. Individual buttons and menus now have multiple color gradients, drop shadows, and glow or pulse when the mouse hovers over them. These details would take hundreds or thousands of lines of code to program from scratch, but are often included with no additional work when using standard widgets provided by an API. This in turn makes programmers more reliant on the API; while programmers used to be able to code their own button, for example, it is increasingly difficult to do so while providing the look and polish that users have come to expect.

APIs are not just a helpful tool to enable reuse when programming; in many cases they are a *required* part of programming. For reasons of security and encapsulation, some functionality is only available through APIs, and cannot be replicated directly. For example, access to the graphics card is no longer directly allowed by most operating systems, and programmers must access it indirectly through one of several APIs instead.

This reliance on APIs is visible in the code that people write. Anecdotally, a framework developer at Microsoft described how the code he saw used to consist mostly of basic programming language constructs — `if` statements, `for` loops, variable declarations — but that now code consists mostly of calls to APIs.

A result of these trends is that the task of programming has changed, so that the fundamental activity is no longer specifying functionality from scratch, but is instead that of stitching together existing functionality from APIs. When multiple APIs exists with overlapping functionality, the usability of the different APIs can be a deciding factor in choosing which one to use. Anecdotally, even seemingly small usability flaws in an API, such as a poorly named or placed method, can quickly add up to "death by 1000 paper-cuts", creating an overwhelmingly negative experience.

This change has affected programmers we have observed, from novices to experts, and in all types of domains and environments. It is also one of the barriers that can prevent novice programmers from being successful [Ko 2004].

This reliance on APIs has changed not only how code is initially written, but also how it is later understood, debugged and maintained, and affects not only the productivity of the programmer but also the reliability of the resulting program. Finding bugs that are a result of implicit assumptions made by APIs — for example that a certain method only be called from a certain thread, or that a particular return value cannot be cached — can be very difficult, as they are not readily apparent when examining the code that calls the APIs. Because APIs often encapsulate large amounts of functionality, they also often limit the ways in which programs can be easily changed, sometimes forcing the choice between limiting a feature to the capabilities provided by an API or not using the API at all and re-implementing its functionality from scratch.

Another reason for the difficulty in using APIs is that modern APIs are large — the base Java 1.5 APIs, for example, contain more than 4000 classes and and more than 35,000 different methods and Microsoft's .NET 2.0 Framework has more than 140,000 classes, methods properties and fields. These APIs grow larger with every release, as the old functionality is retained for backwards compatibility. The MSDN documentation, which used to be available in printed format, is now only available electronically and takes up three DVDs of compressed textual documentation, more than would fit on any bookshelf.

1.3. THE APPROACH

This thesis sets out to first understand some of the core reasons why APIs are often not usable, then develops a set of proposed design improvements to APIs, API documentation and programming tools, and tests if these changes make the APIs easier to use.

In the first part of this work, I identify common patterns in API design that might make APIs harder to use. I base these hypotheses on insights gained from previous studies of APIs. I then compare the usability differences of specific API design decisions, while controlling for biases of any particular API. For example, I conducted one study that evaluated an *object constructor* design choice: whether or not to require constructor parameters (instead of making them optional). The results of these studies inform the design of any API that might use these patterns, for different groups of programmers.

Having acquired a better understanding of the problem, I develop solutions for making APIs easier to use. There are several possible approaches to improving API usability. In this thesis I discuss changing the API's design, the API's documentation and the programming tools. Other approaches, such as changing the programming language or the way programmers are educated, are discussed in the Future Work.

The work in this thesis is inspired by techniques from the Human-Computer Interaction field, and by the Natural Programming Project in particular [Myers 2004]. In this approach, I start by studying real programmers to see what their problems are. I then come up with different solutions and then test prototypes of these solutions on users, to verify that the solutions actually work.

1.4. STAKEHOLDERS

To identify the attributes that affect the quality of an API, we look at the different parties that are most affected by the API.

API designers are involved early in the lifetime of an API. Some of their goals are: to maximize the adoption of an API, to minimize the support costs, to minimize development costs (this is perhaps less important since it is a one-time cost), and to be able to release the API in a timely fashion.

API users are the programmers who use an API to help write their programs. Their goals are: to be able to quickly write error-free programs (without having to limit their scope or features), to use APIs that many other programmers use (so that other users can test the APIs, provide answers to questions and post sample code using the APIs), and to have their applications run quickly and efficiently.

Consumers of resulting products are also affected by APIs, though because they are often unaware of the specific APIs, this can be indirect. In the case of user interface widgets, however, the consumers might be aware of which API is being used; for example, some might prefer products created using Eclipse's SWT API rather than the JDK's Swing API since SWT uses OS-specific widgets that respect the OS settings like widget style and size. Consumers' goals include: having products with desired features and no bugs, and consistency, including the use of standard widgets.

1.4.1. CLASSIFYING API USERS

A commonly acknowledged property about APIs is that they need to be appropriate for their audience [Bloch 2001][Cwalina 2005]; an API that works well for one set of programmers might not work well for others. This raises the problem of how API designers can cluster and classify programmers into groups that accurately correspond to different API requirements.

One approach is to group programmers by how much experience they have, and with what languages and tools. In this approach, an API designer might make one set of APIs for novice programmers and another for experts. For example, one set of APIs for programmers comfortable with Visual Basic, and another for programmers comfortable with C++.

A related approach is to classify programmers by their job type [Scaffidi 2005]. "Professional" programmers are typically software engineers whose primary job is coding and who often have formal programming education. "End-user programmers," on the other hand, create code only as needed to support another primary occupation, such as physicist or administrative assistant.

A third approach is to use programmer "personas" [Pruitt 2003][Clarke 2004]. Personas are user archetypes often used in design to make different groups of users more concrete and understandable during the design process. Microsoft uses three different programming personas constructed after observing several hundred Visual Studio users [Clarke 2004]. These personas attempt to capture the most common programming work styles. While the personas correlate roughly with different experience levels and job types, they do not correspond directly; any programmer can potentially have any persona, which is most commonly judged by the different approaches they take to different programming tasks. These personas capture different *work styles*, not experience or proficiency. We used these personas to tailor our experiments and participant recruitment criteria. By studying a number of programmers of a specific persona, our results generalize well to other programmers of that same persona.

1.4.2. Programming Personas

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

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

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

1.5. QUALITY ATTRIBUTES

There are many different *qualities* that are desirable for APIs, though previous research had not attempted to enumerate them all. We create a set of quality attributes here, forming a hierarchy of attributes. Figure 1.2 includes a summary of these attributes and the stakeholders most affected by each.

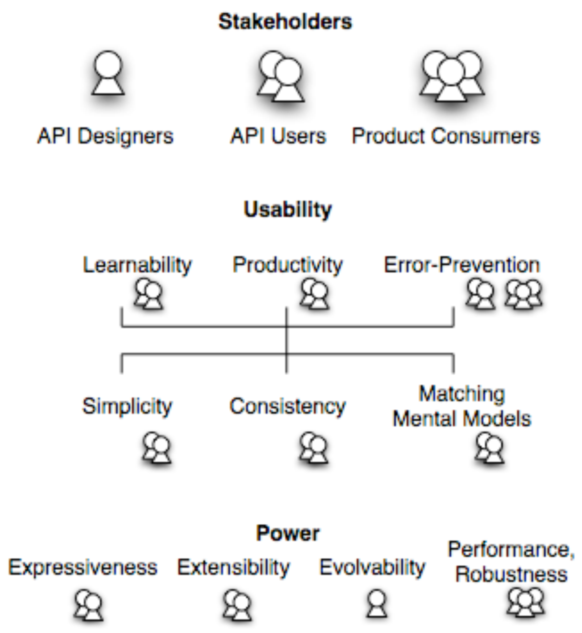


Figure 1.2. Quality attributes of APIs, and the stakeholders most affected by each quality.

At the highest level, the two basic qualities of an API are its **usability** and its **power**. Roughly, "usability" refers to the qualities of an API that affect its use when creating and debugging code, while "power" refers to limits of the code that can be created.

Usability includes such attributes as how easy an API is to learn; how productive programmers are using it; how an API prevents errors; how simple it is; how consistent; and how well it matches its users' mental models.

Power includes an API's expressiveness (the sorts of programs it can create); its extensibility (how users can extend the API to create convenient user-specific components); its evolvability for the API designers who will update the API and create new versions; its performance (in terms of speed, memory and other resource consumption); and the robustness and bug-free-ness of the API implementation.

The usability mostly affects API users, though the error prevention affects the consumers of the resulting products. The power affects mostly API users and product consumers, though the evolvability affects API designers.

We have heard anecdotal evidence that usability can also affect API adoption. If an API is taking too long for a programmer to learn, some companies will choose to use a different API, or write simpler functionality from scratch.

In some cases, these one or more of these attributes must be sacrificed in order to fully achieve another. (For example, maximal speed might require conceding some usability.) However, in other cases, changes to APIs can improve all of the metrics, or at least not harm them.

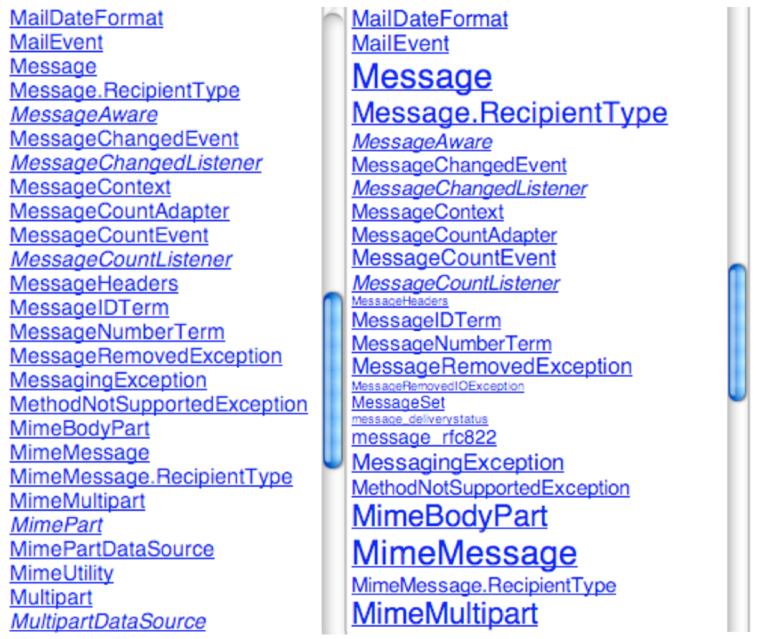
1.6. An Example

To show how the knowledge and tools of this thesis can be used to help solve real problems with today's real APIs, consider the case of the Java APIs for sending email.

When trying to send an email message, the users in our studies all began by looking for a class representing email messages in the Java Mail APIs. The class most users initially found was the `Message` class, which according to the description can represent email messages. Most users would then try to create an instance of `Message` by typing `Message m = new Message();` however this did not work for two reasons. First, because Message is an abstract class and so cannot be directly instantiated. Once users had found MimeMessage, the only public subclass of Message, and changed their code to `new MimeMessage()` they then received a message that `MimeMessage()` was not a known constructor. This is because all of the public constructors in the MimeMessage class require at least one parameter. Once users discovered that they had to use one of the required constructors, and how to instantiate the required parameter, they then proceeded to use the setter methods to fill in properties of the message such as sender, recipient and subject. After filling in all of these properties, users would look for the `send()` method, the last method they needed to complete their task. However, neither the `MimeMessage` nor the `Message` class contain a method named `send()`. Users would look through every method that was provided by these classes, thinking that maybe the method was called something else. When none of the more than 100 methods mentioned sending, users would sometimes conclude that MimeMessage was in fact the wrong class, or that the Java Mail APIs simply did not support sending messages. When users were instructed to keep trying to solve the task, they would eventually start examining the other 200 classes in the API and eventually find the `Transport` class, which does contain a `send` method. Users would attempt to create an instance of the `Transport` class so they could call its send method, passing the MimeMessage instance, but would eventually realize that was both difficult and unnecessary, because `send` was a static method.

Now let us look at how we would design this API differently based on the results of the studies described in this thesis. First, we would recommend that the most attractive class name — `Message`, in this case — be used for the concrete class rather than the abstract interface, and that this class be at the top level in the package hierarchy. The interface could then be named `MessageInterface` or `AbstractMessage`. Because the API only supports Email messages, the `Message` class could more descriptively be named `MailMessage` or `EmailMessage`. Because all of the users in our studies expected and preferred to be able to create objects using an empty, default constructor, the `EmailMessage` class would have a default constructor, in addition to its current ones. Because sending is an important action to take on `EmailMessage`s, the method should be placed on the `EmailMessage` class, not (or in addition to) the `Transport` class.

Now let us look at how a programmer could use one of the tools presented in this thesis to more easily use the actual, original API. Jadeite, a variant of the normal Javadoc documentation designed based on our observations of programmers, displays popularity of different classes as variations in their font sizes. Because `Message` and `MimeMessage` are used often in the most common tasks in the Java Mail APIs, Jadeite shows them prominently (shown in Figure 1.1).

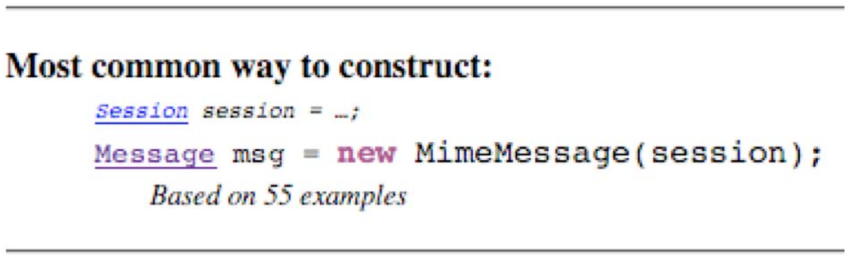


MailDateFormat
MailEvent
Message
Message.RecipientType
MessageAware
MessageChangedEvent
MessageChangedListener
MessageContext
MessageCountAdapter
MessageCountEvent
MessageCountListener
MessageHeaders
MessageIDTerm
MessageNumberTerm
MessageRemovedException
MessagingException
MethodNotSupportedException
MimeBodyPart
MimeMessage
MimeMessage.RecipientType
MimeMultipart
MimePart
MimePartDataSource
MimeUtility
Multipart
MultipartDataSource

MailDateFormat
MailEvent
Message
Message.RecipientType
MessageAware
MessageChangedEvent
MessageChangedListener
MessageContext
MessageCountAdapter
MessageCountEvent
MessageCountListener
MessageHeaders
MessageIDTerm
MessageNumberTerm
MessageRemovedException
MessageRemovedIOException
MessageSet
message_deliverystatus
message_rfc822
MessagingException
MethodNotSupportedException
MimeBodyPart
MimeMessage
MimeMessage.RecipientType
MimeMultipart

Figure 1.1. Jadeite, one tool created in this thesis, provides cues about the popularity of different classes by using font size varation. The orginal documentation is shown on the left and Jadeite is shown on the right.

Jadeite also displays examples of the most common ways to get instances of any class or interface. When looking at the `Message` interface, Jadeite shows that it is usually instantiated by using the `MimeMessage` class with the constructor that takes a `Session` object, showing both the concrete class and how to use the required constructor (Figure 1.2).



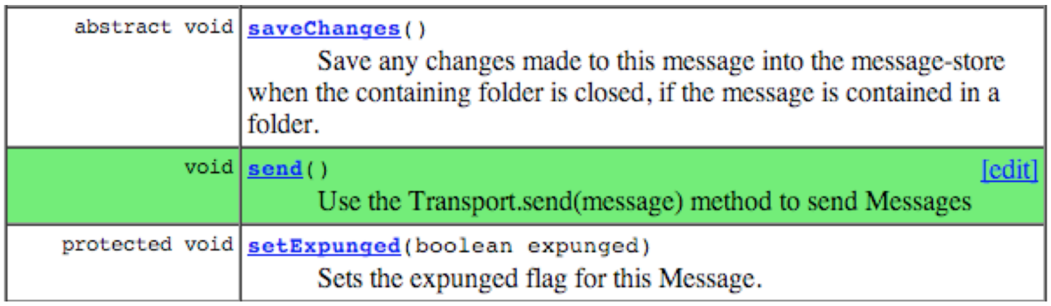
Most common way to construct:

```
Session session = …;
Message msg = new MimeMessage(session);
```

Based on 55 examples

Figure 1.2. Jadeite automatically finds and displays examples for the most common way to instantiate classes and interfaces.

Because programmers in our studies often strongly expected that a particular class or method should exist when it did not, Jadeite allows programmers to create "placeholder" classes and methods that appear in the documentation alongside the real API. When looking for a `send()` method on the `Message` interface or `MimeMessage` class, a programmer using Jadeite would see placeholder methods created by other users who had expected and not found the method on these classes, annotated with the actually code required, using the `Transport` class (Figure 1.3).



abstract void	saveChanges() Save any changes made to this message into the message-store when the containing folder is closed, if the message is contained in a folder.
void	send() [edit] Use the Transport.send(message) method to send Messages
protected void	setExpunged(boolean expunged) Sets the expunged flag for this Message.

Figure 1.3. Jadeite lets anyone create "placeholder" methods that they expected to exist but are not actually provided by the API.

This brief example shows three different ways Jadeite provides to help with the problems in this one API. Alternative solutions to these types of problems using APIs are also presented in the other tools described later in this thesis.

1.7. CONTRIBUTIONS

This dissertation offers several contributions:

- A categorization of API design decisions and their importance.
- A methodology for designing and conducting usability studies of APIs.
- Evidence that small changes to APIs can have large impacts on their usability.
- Evidence that default (parameterless) constructors should be provided in objects to support usability, rather than only required constructors.
- Evidence that constructors should be provided instead of factories to support usability.
- Evidence that common methods that act on common objects should be placed on the object they act on, rather than on helper objects.
- Three prototype tools to help users with existing APIs: The search engine Mica, the documentation tool Jadeite, and the API browsing tool Apatite.
- Evidence that users are significantly more productive when given access to these tools.

My thesis is: *Programmers are significantly more productive using APIs and tools designed based on knowledge of the usability impact of API design choices.*

1.8. Outline

This thesis is organized in four main sections. The first section (Chapters 1 through 3) looks at the related work and examines the stakeholders and issues more closely. The second section (Chapters 4 through 7) delves into a deeper understanding of what makes APIs hard to use. The primary approach used here is the usability lab study, with which we identify several specific sources of problems in modern APIs. The third section (Chapters 8 through 10) examines different ways to solve these problems, including changes to API documentation. These changes take advantage of data from other people's usage of these same APIs to provide additional cues about what parts of APIs are likely to be most helpful and how to use them. Finally, the closing section (Chapters 11 and 12) steps back and examines the limitations of our approaches, future directions, and what general lessons we can take away from this work.

2.

RELATED WORK

This work builds directly and indirectly on the research of others, in the areas of running user studies, deciding how to build APIs, and building programming tools.

2.1. STUDIES, OPINIONS AND THEORIES

The most relevant work to the first half of my research is the writing others have done on the design of APIs, the lessons they have learned and the techniques they use to design APIs and reason about their effectiveness.

2.1.1. API USABILITY STUDIES

In particular, my research is especially motivated by the usability studies of specific APIs done at Microsoft [Clarke 2004][Clarke 2005][Clarke 2007] and elsewhere [McLellan 1998][Bore 2005]. The prior work studied a particular API – a mail API, for example – by having programmers who are representative of the intended audience for the API perform tasks intended to be common for the API – for example, sending an email message and connecting to a mail server.

The primary difference between these studies and those in this thesis is that, while the prior studies focus on finding the usability issues of a specific API, mine looked at the usability issues of a pattern that occurs in many different APIs. While studies of specific APIs are able to offer the most complete and tailored advice for a particular API, my studies are able to provide a smaller piece of advice applicable to many different APIs. My design of study tasks also differs from these previous studies. Studies of specific APIs can use common tasks that the API is intended to perform, while my studies necessarily use tasks from a variety of real and artificial APIs so that my results generalize to many types of APIs.

Other work has looked at decisions made in programming language design [Jones 2003]. My work focuses on decisions made at the API rather than language level.

2.1.2. Advice from API Designers

There are many scattered sources of API recommendations in print and online. Two of the most comprehensive are the books by Joshua Bloch from Sun Microsystems (now at Google)[Bloch 2001] and Krzysztof Cwalina from Microsoft [Cwalina 2005]. Each book presents guidelines that have been developed over several years and during the creation of such widespread APIs as the Java Development Kit and the .NET base libraries, respectively. For example, Bloch discusses the many architectural advantages of the factory pattern, which we study in Chapter 5, and advises its use in many scenarios. These books are informed not just by the authors' experience but also by the experience of their companies. My research builds on this information and adds to it by running user studies to see how programmers actually use different APIs. The results show that API designers' assumptions about how programmers code and what will make a better API can be incorrect.

2.1.3. Informal Opinions on API Usability

With the growing popularity of blogs and forums, one can find many, possibly less well informed, opinions on API design. Additionally, new documentation and websites that solicit user feedback, such as Microsoft's MSDNWiki, make it easier to find more voices on API design. These less formal sources provide venues for debates of contentious API decisions, providing more viewpoints than the API guideline books tend to provide for a given decision.

2.1.4. Empirical Studies of Programmers

Much research has studied the strategies and effectiveness of how different programmers program. Much of this research is relevant to API design, although it is not often possible to directly make conclusions about how APIs should be designed from these studies. Studies of programming comprehension have compared how novices and experts understand object-oriented code in different tasks and phases of development [Beurkhardt 2002], and how programmers understand and decompose solutions in object-oriented environments [Rosson 1990]. My research attempts to answer some of the same questions, but is focused on the ones directly relevant to API design.

2.1.5. Design Fragments: Codifying API Usage

One line of research has been to explicitly capture the semantics of API usage that is currently only available in an API's textual documentation, if at all. *Design fragments* [Fairbanks 2006] capture the common idioms of API use and, like my research, are motivated by the difficulty of using modern frameworks. Like some of my programming tools, they try to make the common uses of an API easier. Design fragments are manually created, and are displayed in the context of the IDE, while I have explored ways of automatically detecting usage. However, design fragments can be used to capture longer and more complicated usage patterns than what are currently addressed in my tool research, and code usage can be automatically checked against design fragments to catch common bugs. Design fragments offer a way to improve API usability complimentary to my approaches, provided that API designers or others create these design fragments.

2.1.6. Verifying API Usage

While design fragments capture API patterns that are common, but not strictly necessary or inflexible, other research [Bierhoff 2008][Gopinathan 2008] involves the specification and verification of API usage constraints that must *always* hold for code to be correct. While *correct* API use is one important aspect of using APIs, the usability problems we have observed so far have been focused on programmers figuring out how to get APIs to do what they want, not just to do something correctly.

2.1.7. Object-Oriented Design Literature

Many books and articles aimed at software engineers offer recommendations on how object-oriented systems should be designed and implemented. Some of these recommendations are relevant to API design. Some authors recommend that object-oriented frameworks should be designed by first modeling domain knowledge and then building objects that represent this knowledge [Aksit 2000][Evans 2003]. Others say that systems should be designed to match the organization of the people who will maintain the code [Wirfs-Brock 1990]. My research focuses on the API-relevant aspects of these recommendations, and uses empirical data from users studies rather than opinions.

2.1.8. Design Patterns

Design patterns capture common implementation techniques used in building large software systems [Gamma 1995]. API design choices differ from design patterns because they relate only to choices visible in the programming interface, while design patterns often relate to internal architecture decisions. The two overlap when design patterns are externally visible, such as the factory pattern of object construction [Gamma 1995]. A previous study [Vokáč 2004] has examined the usability effects of using design patterns inside code implementations, and found that some design patterns are much more difficult for programmers to learn and use than others. For example, while the Observer and Decorator patterns seemed easy for programmers to learn, the Visitor pattern caused much confusion. This work suggests that some patterns, at least, can be the cause of considerable programmer difficulty.

Research has also been conducted into the role of design patterns in general, and the abstract factory pattern in particular, in computer science curricula [Astrachan 1998]. This work shows that although the factory pattern is considered a superior method by educators, they overwhelmingly 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, in which implementation details are kept in a private subclass.

My research is focused on the usability of *using* APIs by professional programmers while design pattern research has traditionally focused on how patterns can create maintainable architectures for implementers of APIs or large systems.

2.1.9. Information Foraging Theory

Information foraging theory describes users' exploration of many different types of data in terms of Information Scent [Pirolli 1995]. Researchers have found many common patterns that seem to occur across many different domains in which users have cues (such as words) and follow paths (such as hyperlinks). Previous work has shown that information foraging theory applies to program maintenance [Lawrance 2007], and it seems also likely that it would apply to exploring an API using its code and documentation, which would allow previous research to be leveraged to better understand API usability. The focus of the study in Chapter 6 is specifically about searching for a starting class and methods in classes. Results from the previous information foraging theory research alone do not provide sufficient guidance for design.

2.2. Evaluation Techniques

Though not usually applied to programming tools or APIs, the field of HCI has many rich evaluation techniques for evaluating the usability of user interfaces. We base our techniques on several of these. In our lab studies described in Chapters 4 through 7 we use techniques like the think-aloud protocol, in which users are encouraged to verbalize their thought process, to better understand their use of APIs. When performing a case study at SAP (Chapter 7), we used user-centered and participatory design techniques to involve users in the creation of a revised API.

2.3. Programming Tools and Documentation Research

The tools described in Chapters 8 through 10 of this thesis were inspired by the studies in Chapters 4 through 7, and also by existing programming tools and by the limitations of these tools.

The studies showed that it is useful to display example code, as is done by Mica (Chapter 8) and Jadeite (Chapter 9). The Strathcona system [Homes 2005] finds and recommends source code examples based on an IDE's current context. While it does not find examples from the Internet or allow explicit queries, it addresses the problem of discovering how to use APIs to perform a desired task from examples. Because its example search is implicit and based on the currently written code statements, it is unable to help in situations where programmers do not have any starting point. The tools I have designed help programmers with the stage of finding the first few methods from which they could then make use of a tool like Strathcona.

Another source of inspiration for our tools were tools that take advantage of programmers' aggregate use patterns to automatically infer which parts of code are relevant to other parts of code. IDE tools such as Team Tracks [DeLine 2005] help programmers with how to use the internal APIs of large projects based on other programmers' IDE usage. For example, if programmers in a team often look at the `invalidate()` method after looking at the `paint()` method, then `invalidate()` will be automatically suggested when a new team member looks at `paint()`. While the learning task that this tool addresses is similar to that of the tools in this thesis, it is more suited to learning private code, about which there might be no information on the Internet, while the tools in this thesis are more suited to public APIs or open source projects large enough to have Internet discussion sites. We adapt techniques from tools like Team Tracks by using co-occurrence in web results to detect related classes and methods for the tools in this thesis.

In our search tool Mica, presented in Chapter 8, we analyze which API keywords are most correlated with the search results. The techniques used in Mica for finding keywords that are correlated with the top results of a query are similar to techniques used for query expansion [Baeza-Yeats 1999] in information retrieval systems. While this is often used only for document retrieval, in interactive search systems the expanded terms may be shown to the user. The sidebar in Mica system differs from these systems in that it filters based on programming relevance and the primary use of the terms is user understanding.

There are several search engines that search specifically for code[*][†][‡]. However, these search only a limited repository of known good code. They are unsuitable for solving the vocabulary problem [Furnas 1987] we observe programmers encounter with APIs in Chapter 8, because they do not search the informal forums and other pages that help programmers use naïve terminology to find the correct terminology. They are also limited in their use in finding examples even when programmers have a

[*] JExamples, http://www.jexamples.com

[†] Hoogle, http://www-users.cs.york.ac.uk/~ndm/hoogle/

[‡] Google Code Search, http://www.google.com/codesearch

method name to start from, because they have such small repositories and because the repositories lack the textual descriptions that let programmers find methods used with a particular intent.

2.3.1. Mining API Information from Repositories

All three of the tools in this thesis attempt to extract and aggregate knowledge about how APIs are used. There has been much recent research on how to mine useful information from large repositories of source code [Kagdi 2007]. My tools differ from most of this work in that they use code snippets from standard webpages, found using Google and Yahoo, rather than a CVS repository or source-code-specific search engine. I chose this approach for two main reasons. First, for breadth: none of the code repositories or code search engines I have seen is as comprehensive in the variety of examples they contain as what is indexed by Google and Yahoo. Second, using web search engines allow the tools in this thesis to be representative of common usage. Many code search engines, in comparison, are heavily affected by a relatively few very large open-source applications, whose usage of any given API is not always representative of how a typical programmer might use it. However, for a different, possible private, API, it is possible that a different approach would work better.

A few API search systems like Assieme [Hoffmann 2007] also use webpage snippets as source data. One of the main differences between Jadeite (Chapter 9) and these systems is that Jadeite presents a Javadoc-like hierarchical browsing interface, rather than a search interface. Searching and browsing interfaces both have their advantages, and can also be used together. However, we chose in Jadeite to focus on trying to create the best browsing based interface, in part because this let us do our analysis offline, allowing us to analyze more data while avoiding any latency issues during use. Furthermore, browsing compliments search interfaces by helping users find the right terminology to search for.

One technique explored in Chapters 9 and 10 is that of determining and displaying which API classes and method are *popular*. Recent repository mining work [Holmes 2008] has used method popularity data to recommend the most popular parts of an API. The font sizes in Jadeite and Apatite are similarly motivated, though different in presentation style (font sizes) and context (lists of classes in standard API documentation).

2.4. SUMMARY

Despite the wealth of opinions on how to create APIs in this body of related work, there is little evidence to show how one can create usable APIs. And though many useful tools can sometimes be of assistance in using APIs, they do not yet solve all of programmers' problems. From other HCI research, we can find many techniques that have been applied to studying graphical user-interfaces. These can usefully be applied to studying APIs, which are one of the primary user interfaces used by programmers. But which parts of API design can we study that would be most useful to informing the design of many new APIs? The next chapter presents our analysis of the different considerations of API design, from which we will select of specific areas to study. The studies in turn provide the inspiration and evidence for a collection of new programming tools.

3.

API Design Decisions[*]

Before we discuss the studies of APIs, we first need to discuss what the different types of API design decisions are, which matter, and why.

[*] Portions of this chapter previously appeared in [Stylos 2006b].

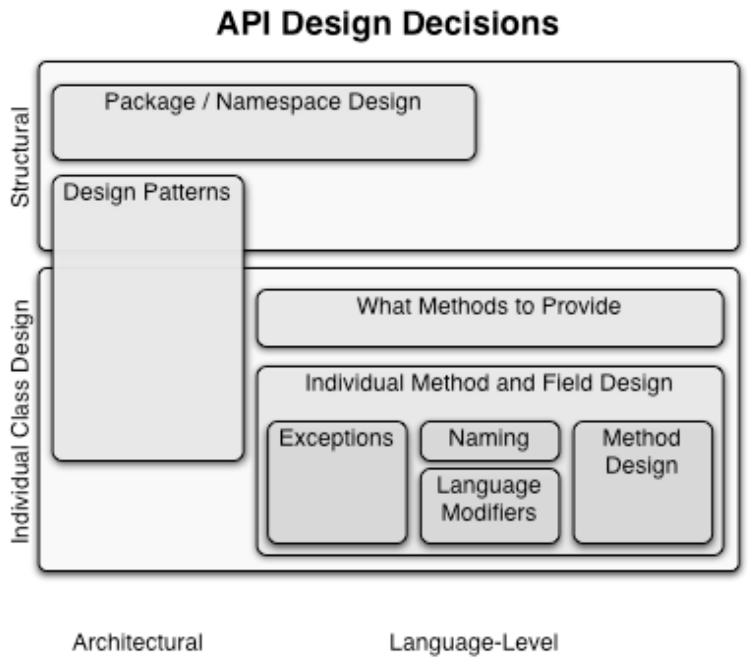


Figure 3.1. An overview of API design decisions relevant to object-oriented languages.

3.1. Space of API Design Decisions

This section maps out the space of API design decisions. We start by looking where these decisions fit in with other development decisions – such as those related to tool and documentation design. We then discuss the organization we chose for the space and then look at specific recommendations in the space. An outline of the design space is shown in Figure 3.1.

3.1.1. Development Decisions

When designing APIs, decisions not directly relating to the APIs themselves are also relevant. For example, alternative solutions to an API usability problem might be to change the API, change the documentation, provide more example code, or if possible, to change the development tool. For the first half of this thesis we focus on the design decisions that directly relate to the resulting APIs themselves. In the second half of the thesis we investigate decisions made in tools and documentation (Figure 3.2).

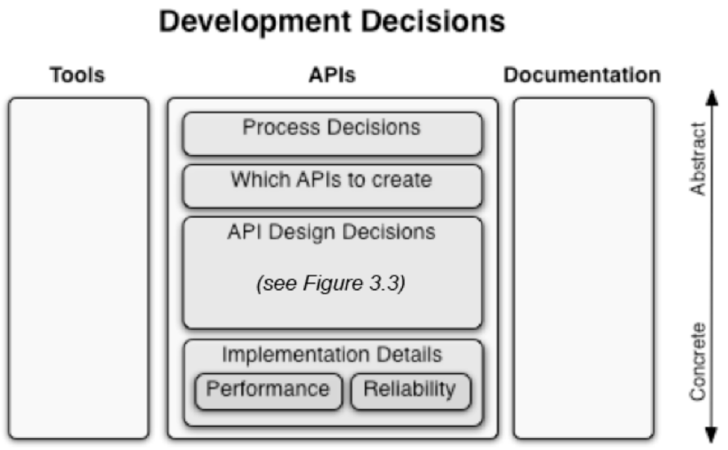


Figure 3.2. How the API design decisions we consider fit into the broader space of design decisions.

Tool design decisions are relatively separable from API decisions; however, tool decisions are still relevant to APIs. For example, tool features like code-completion in the code editor can change the way API users explore APIs.

Documentation decisions are more closely related to APIs, with documentation being important to and closely linked with APIs. However, these decisions are still separable, and often made by different groups of people. Chapters 9 and 10 talk about ideas for improving API documentation.

API designers and organizations make design decisions related to the *process* of designing APIs in addition to those about the APIs themselves. We separate these from decisions relating directly to APIs. Similarly, we separate questions that relate to deciding which API to create – for example whether to create a networking API instead of an XML API – from the questions of how to design APIs for a particular topic. Finally, we separate the implementation details that are hidden by an API's design.

3.1.2. Mapping the Design Space

To build the design space described in this section, we first started with the API recommendations by experts [Bloch 2001][Cwalina 2005]. These are arguably the most comprehensive lists of API design decisions, and using these as our basis

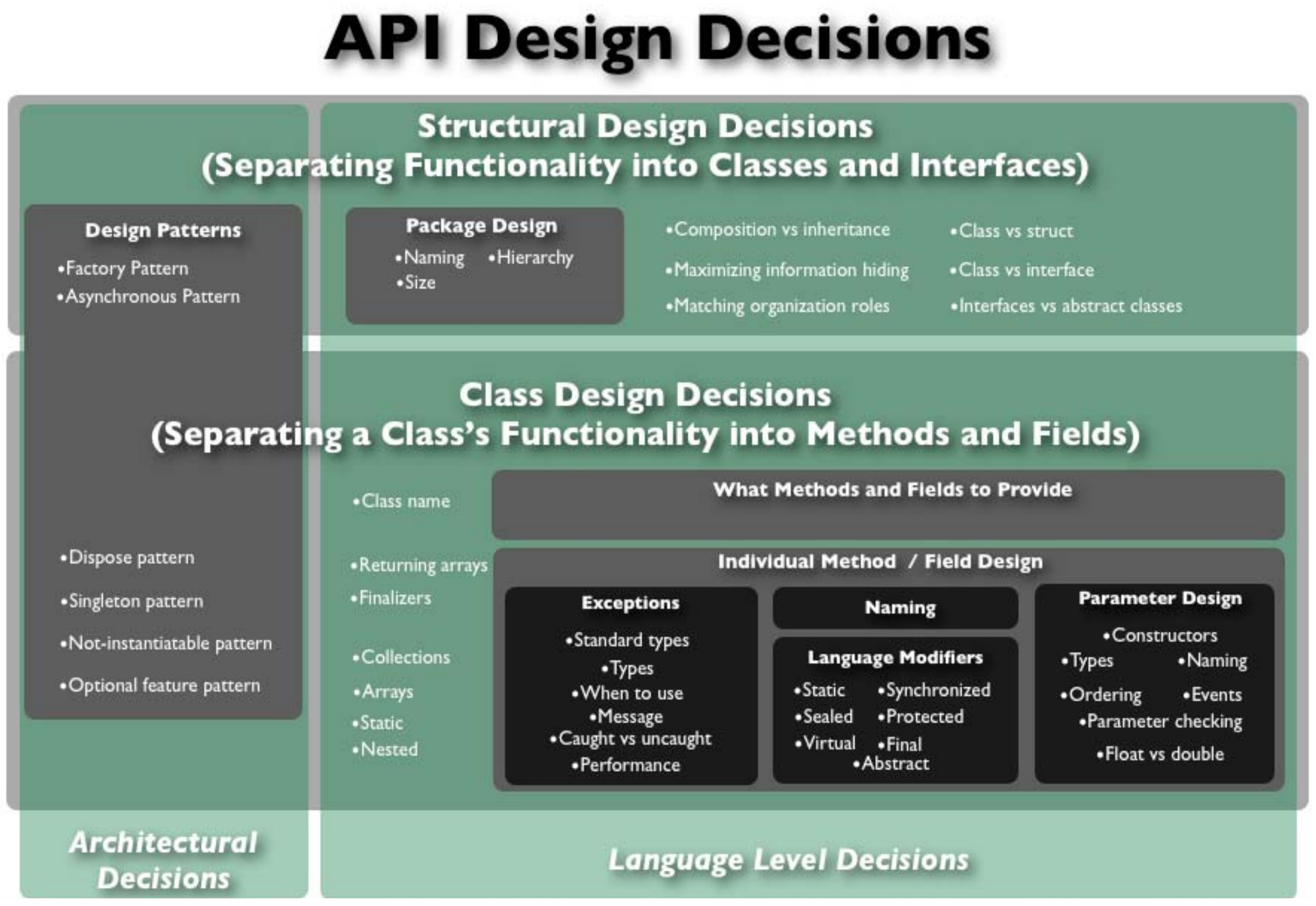


Figure 3.3. A map of the space of API decisions, with specific recommendation topics from API experts included as bullet-

ensures that the general shape of our space reflects the sort of decisions that API designers care about today. These sources cover a broad range of topics, having collected all of the recommendations from several years of API development. However, there is no guarantee that they are complete. This is reflected by the fact that not all of the topics in these books overlap, though many do.

To help create a more complete design space, we consulted the language specifications for Java and C# and identified all of the language-level features relevant to API design.

To generate a more comprehensive list of the architectural-level API decisions, we reviewed the literature on object-oriented system design (e.g., [Wirfs-Brock 2003]). These sources contained many architectural patterns that might be, but rarely are, used in API design. It is likely that the tradeoffs of using some of these designs in public APIs, which are used more often then they are designed, would differ significantly from the design of large-scale object-oriented systems, which are maintained more often than they are reused. It remains unclear whether the relative inattention given to architectural decisions in API design is because the field is new or because these types of complicated architectures are fundamentally unsuited for APIs (for example, because they may be less usable).

The API design decisions that we consider in this chapter correspond closely to the decisions that the API recommendation sources discuss. However, some of the recommendations from these sources pertain to process or documentation decisions and are therefore outside the scope of those considered here.

3.1.3. DIMENSIONS OF THE API DESIGN SPACE

One of the principal challenges in mapping the space of API design decisions is that of grouping and separating different decisions. A challenging aspect of this task is to identify which categories and dimensions of decisions are most fundamental to API design, rather than being a result of the language design. After several iterations, we chose two different categories for the map.

The first category is the decisions relating to which classes (and interfaces) to provide – the overall class structure – versus decisions internal to a specific class, such as what methods and properties to provide in that class. The “inter-class” or "structural" decisions are shown on top of Figure 3.3, and "specific class design" or decisions internal to a particular class are shown on the bottom.

The second category we use is that of specific programming language features versus "architectural" features of an API. For example, a language-level decision might be whether or not to make a class "static", or a method "synchronized," while an architectural decision might be whether or not use the "factory" design pattern for object creation. We use the term "architectural" here broadly to refer to any decision at a higher level than a particular language feature. The architectural decisions are shown on the left of Figure 3.3 and the language-level decisions are shown on the right. An important distinction between these two types of API design decisions is that there are a fixed number of language features, but a potentially infinite number of architectural decisions. Despite this, the majority of API recommendations and discussions referred to specific language-level features and not architectural decisions.

3.1.4. Specific API Design Decisions

While so far, this chapter has focused on categories and organization of API design decisions, we now examine the specific decisions and recommendations made by API design experts. We use the topics of these recommendations to fill in the API design decision space outlined in Figure 3.1, creating the more detailed map shown in Figure 3.4. Each bullet point in this map represents a topic on which designers have published recommendations. Multiple recommendations (and types of recommendations) can exist for a single topic. The most common type of recommendation is the situations in which to use a particular pattern or language-feature – such as when to make a method "protected." However, some topics, in particular naming, are relevant to all classes and methods and are not optional.

The bullets in Figure 3.4 contain short descriptions of API design topics. For longer descriptions, and the particular recommendations that they correspond to, see [Bloch 2001] and [Cwalina 2005].

3.1.5. API Design Controversies

Certain API design decisions are more contentious than others. We list here several topics of debate as found on online forums. We summarize the topic of debate but not specifics of the opposing sides, their rationales, or merits.

- Java Exceptions: Whether to use checked versus unchecked exceptions.
- Returning null versus throwing an exception.
- Returning null versus returning an empty object (i.e., an empty string).
- Returning error codes versus throwing exceptions.
- Naming: using “Hungarian” notation.
- Naming: using namespaces to disambiguate name collisions.
- Naming: how to name an updated version of an old class or method.

An interesting property of these points of contention is that for most, each side has little or no data to support their beliefs, even though most of the claims made in the debates are based on testable hypotheses. For example, on returning null verses returning an empty object, a hypothesis of the pro-null side is that returning null will cause users of the API to be more aware of error conditions. Another hypothesis is that this greater awareness will lead to more aware programmers who create code with fewer bugs. Both of these hypotheses can be tested. Side-by-side usability comparisons can help shed actual user data on the issues that have thus far have mostly been limited to ideological debates.

3.2. Importance of API Design Decisions

Because there are so many API design decisions, it is useful to be able to prioritize them. In general, an API design decision might be said to be “important” if it has a large impact on at least one stakeholder. This section identifies different dimensions in which one API design decision might be more “important” than another. Because there are different dimensions, there is no “absolute” importance of an API decision and comparing design decisions requires making trade-offs between the different aspects of importance. Table 3.2 summarizes these different metrics.

Design frequency: How often this decision comes up when designing APIs. For example, an API designer might frequently have to make naming decisions and only rarely have to decide which asynchronous execution pattern to provide.

Design difficulty: How likely API designers are to make the decision sub-optimally. Some decisions, such as consistently naming setter and getter methods, have strong existing recommendations, while others, such as when to use exceptions, are more contentious and not consistently applied by API designers.

Use frequency: How often API users are directly affected by a decision. For example, API users might frequently have to initialize a new object using a constructor, but only rarely have to write special error-handling code.

Use difficulty: How costly a sub-optimal decision is for an API user. For example, using the “static” modifier contrary to an API user’s expectations might have less of an effect than changing a method’s access protection.

	API Designers	API Users
Frequent	Designers make this decision often	Users are often affected by this decision
Difficult	Designers do not always make this decision correctly	Users are severely affected by this decision

Table 3.2. Different ways that API decisions can be important for designers and users of APIs.

As researchers, we are especially interested in finding decisions that API designers currently make sub-optimally (which might reveal flaws in conventional wisdom) and that affect API users either frequently or severely.

3.3. Summary

This chapter makes a number of contributions:

- A map describing the space of API design decisions, showing how large the space is and also that it can be grouped in useful ways.
- A discussion of the different ways API design decisions can be “important”.

These contributions helped guide the choice of which API design decisions to study, which are described in the next three chapters, and which tools to create, which are discussed in Chapters 7 - 10.

4.

USABILITY STUDIES OF APIS: CONSTRUCTORS VS CREATE-SET-CALL[*]

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

Create-Set-Call

vs.

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

Required constructor

Figure 4.1. The “create-set-call” construction pattern and required constructor parameters. We compared the usability of these two construction options.

The first study of API design decisions we performed was that of the “create-set-call” pattern of object construction, shown in Figure 4.1. This study was performed during an internship with Steven Clarke at Microsoft, and was motivated by his studies of specific APIs, described in Chapter 2, and was performed in part to help inform the .NET Framework designers.

[*] Joint work with Steven Clarke at Microsoft. Portions of this chapter previously appeared in [Stylos 2007a].

Studies of the usability of specific APIs proved effective at Microsft, but because they create so many APIs, it is not always possible to run user studies on each API designed. This is an issue for smaller organizations as well, which may not have the resources to run user studies of their APIs at all. This chapter describes the first study that instead of testing the usability of a specific API, tests the usability of a design choice that is common to many APIs. By better understanding API design choices, we can better inform the design of all APIs.

In object-oriented languages like the .NET languages and Java, one of the most common API design choices involves what object constructors to provide. These constructors are also some of the most commonly encountered parts of an API by programmers, who have to figure out whether or how to construct each object before they use that object. There are two common design choices: provide only constructors that require certain objects (a "required constructor"). This option has the benefit of enforcing certain invariants at the expense of flexibility. An alternative design, "create-set-call," allows objects to be created and then initialized. Examples of these two are shown in Figure 4.1.

To get a fuller understanding of when to use each in API design, we compared the two approaches in a user study of thirty professional programmers from three distinct programming persona (see Section 1.4.1). The programmers performed several tasks; some these tasks involved APIs with required constructors and other tasks used create-set-call APIs. Some of the tasks involved code creation, and others involved debugging or reading of code.

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

4.1. METHOD

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

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

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

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

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

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

4.1.1. Participant Recruitment

To recruit participants of the personas described in the introduction, we used the following prescreening guidelines:

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

We recruited participants who had registered with the Microsoft Usability Research website: http://microsoft.com/usability. In compensation for their time participants were given two vouchers each redeemable for a software or hardware item at the Microsoft Store, such as Visual Studio, Windows XP or a Microsoft keyboard.

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

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

4.1.2. STUDY ENVIRONMENT

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

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

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

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

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

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

4.1.3. Task 1: Notepad Programming

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

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

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

4.1.4. Task 1-B: File API Design

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

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

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

4.1.5. Task 2: Files and Emails

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

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

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

4.1.6. Task 3: Domain-Independent Classes

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

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

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

4.1.7. TASK 4: MESSAGE QUEUE DEBUGGING

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

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

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

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

4.1.8. TASK 5: OPTIONAL CONSTRUCTORS

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

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

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

4.1.9. Task 6: Reading Code on Paper

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

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

4.1.10. Post-Study Interviews

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

4.2. RESULTS

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

4.2.1. COMMON PARTICIPANT BEHAVIOR

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

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

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

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

4.2.2. Task 1 Results: Notepad Programming

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

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

4.2.3. Task 1-B Results: File API Design

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

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

4.2.4. Task 2 Results: Files and Emails

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

4.2.5. Task 3 Results: Domain-Independent Classes

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

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

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

4.2.6. Task 4 Results: Message Queue Debugging

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

4.2.7. Task 5 Results: Optional Constructors

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

4.2.8. Task 6 Results: Reading Code on Paper

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

4.2.9. Post-Study Interview Results

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

- Initialization flexibility: By allowing objects to be created before all the property values are known, create-set-call allows objects to be created in one place and initialized someone else, possibly in another class or package. This was a common justification given by pragmatic programmers.
- Less restrictive: In general, APIs should let their consumers decide how to do things, and not force one way over another.
- Consistency: Most APIs have default constructors, and so people will expect them. This reason was given by two programmers who created APIs that were used by other members of their teams.
- More control: Several systematic programmers cited the fact that create-set-call let them attempt to set each property individually and deal with any errors that might come up using return-codes, while constructors only allowed for exceptions.

4.3. Discussion

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

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

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

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

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

4.4. Model of Participants' Strategies

To better understand the underlying causes of opportunistic and pragmatic programmers' greater effectiveness with create-set-call we analyzed videos of participants' work and created a model of the participants' strategies for creating and using new objects. This model is represented graphically in Figure 4.2 and described in more detail below.

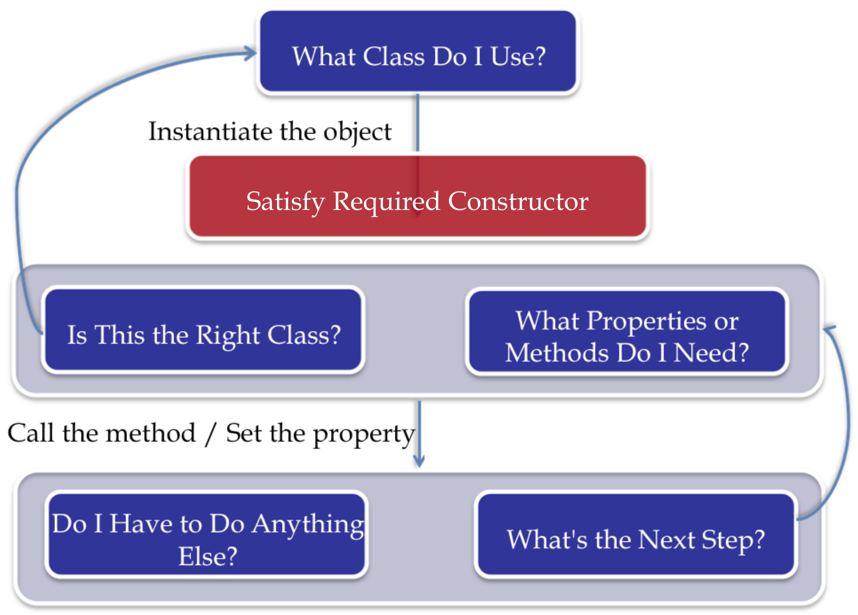

What Class Do I Use?
Instantiate the object
Satisfy Required Constructor
Is This the Right Class?
What Properties or Methods Do I Need?
Call the method / Set the property
Do I Have to Do Anything Else?
What's the Next Step?

Figure 4.2. When constructors were required, the IDE indicated a compiler error, leading users to interrupt their exploration to satisfy the required constructor.

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

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

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

When classes used the create-set-call pattern, the exploration of an instance's properties and methods directly followed finding a candidate class. However, in the case of classes with required constructors, participants were forced to satisfy the required constructor (the second box from the top in Figure 4.2) – usually by figuring out what the compiler error was, then recursively trying to instantiate class for each of the required parameters – before they could finish deciding if the class was even the one they wanted.

By requiring more effort at such an early stage of object exploration, required constructors created, in terms of the cognitive dimensions [Green 1996], a larger *work-step unit*, and greater *premature commitment*. Required constructors theoretically decreased diffuseness of the code, however we did not see an increase in readability as a result. In addition, participants were often annoyed by the unexpected interruption of their exploration, and simply wanted the current construction problem to go away so they could continue their task of finding the right object.

4.5. LIMITATIONS

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

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

The consistency of the results we saw across three different programming languages suggest that the results generalize to other object oriented languages as well.

However, differing syntax, for example named constructor parameters in languages like Objective C, may offer additional variables that need to be taken into account.

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

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

4.6. Summary

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

This study offers evidence that API usability can be a significant barrier for programmers, but that despite the challenges facing API consumers and creators, it is possible to create APIs that are highly usable by a broad range of programmers. Some of this recent success in creating usable APIs is due to running studies of specific APIs: users of a target population perform realistic tasks with an early version of an API [Clarke 2004]. However, this approach is difficult for smaller organizations to apply, requiring resources for a user study, and expensive for larger organizations, which might produce thousands of different APIs a year.

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

The findings of this study were:

- Contrary to API designers' expectations, each type of programmer preferred and was more successful with APIs that used the create-set-call pattern rather than required constructors.
- Required constructors were slower and less preferred in part because of the premature commitment they required and their larger work-step unit.
- A model (in section 4.4) describes the work flow of how programmers explored APIs through code-completion and helps visualize why required constructors interrupted their normal work style.

5.

Usability Studies of APIs: Factory Pattern vs Constructors[*]

```
Factory factory = Factory.getDefault();
Product product = factory.create(args);
```

Factories

vs.

```
Product product = new Product(args);
```

Constructors

Figure 5.1. The abstract factory and factory method patterns versus constructors. We compared the usability of these construction options.

In this chapter we consider the usability implications of one of the best-known object-oriented design patterns: the factory pattern [Gamma 1995]. Following the success of the study in the previous chapter, we chose to examine another API design decision relating to object construction, this time examining classes that used the factory design pattern rather than a standard constructor. Examples of code for

[*] Joint work with Brian Ellis. Portions of this chapter previously appeared [Ellis 2007].

these two options are shown in Figure 5.1. This study shows that creating objects from factories 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.

5.1. 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) [Gamma 1995]. 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 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 5.2.

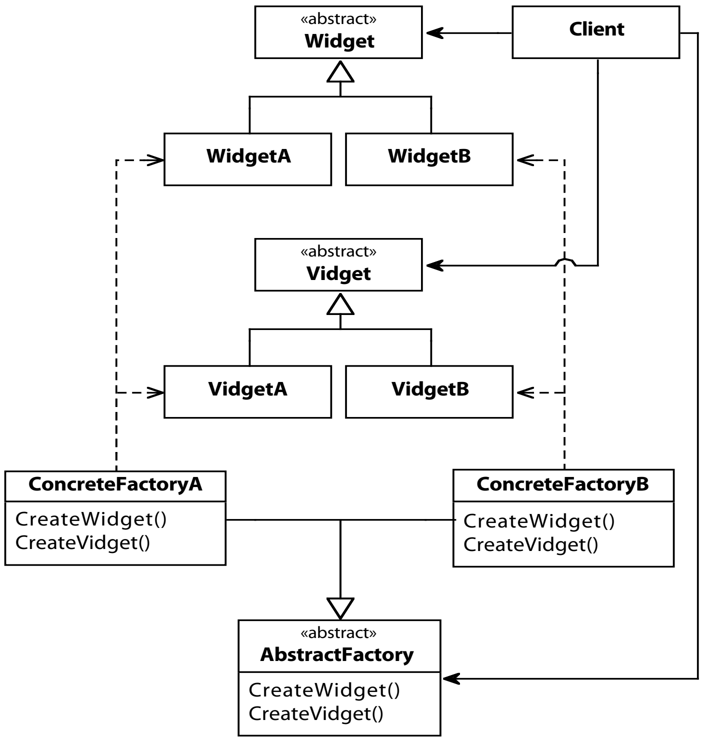


Figure 5.2. The abstract factory pattern in UML.

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 in Figure 5.2, the code to do this might look something like this:

```
AbstractFactory f = AbstractFactory.getDefault();

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

The "factory method" pattern is related but simpler: like the abstract factory pattern, the goal of the factory method pattern is to allow 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. (In [Gamma 1995] the factory method pattern always uses a non-static method; however, many factory methods in Java and other APIs use static methods, and we include those in our discussion of factories.) 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 [Gamma 1995] 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 as shown above.

5.1.1. ADVANTAGES

Gamma, et al. described in some detail both the benefits and liabilities of the abstract factory and factory method pattern as they saw them [Gamma 1995]. 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. [Gamma 1995] 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. Another problem 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, 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.

5.1.2. Uses in Practice

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 [Florijn 1997]. By convention, however, factory classes often end with the word "`Factory`" — using this simple metric, the Microsoft .NET 2.0 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. More prolifically, the Java 1.5 SE API contains at least 61 factory classes and interfaces (out of 3,279 total). 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[†]. 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.

5.1.3. Alternatives

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.

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

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[‡]. 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 [Astrachan 1998]. Like a factory, the parent class depends directly upon its children, but no special factory class or factory method 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 is much simpler, however.

A class cluster could be used to implement the example shown in Figure 5.2; 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();

    }

}
```

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

```
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. Note also that there would no longer be two parallel class hierarchies, one of products and the other of factories — another advantage over factories. The design does require at least the allocation of new instances of `Widgets`, however it can reuse instances of `WidgetA` and `WidgetB`.

5.2. METHOD

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 strove to present the factory pattern in as many contexts as possible to account for any bias toward or against factories in one particular domain (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.

We crafted a series of five Java programming tasks (outlined in following sections) that explored 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 project. Participants were also given access to the Sun Java 1.5 SE API specification[§], which contains documentation for every public class and method in the Java API.

Participants were selected from a pool of applicants at Carnegie Mellon University 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 the inevitable computer science students. Twelve participants were selected, with programming experience ranging from one to twenty-two years. Eight had experience programming in a professional capacity, eight were students in a computer- or electronics-related major, and two were non-technical students. All were males between 18 and 35 years of age.

Participants were randomly put into the factory or constructor conditions for tasks with 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.

A 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

[§] Java 2 Platform API Specification, http://java.sun.com/j2se/1.5.0/docs/api/index.html correlated with time to task completion, which also includes such activities as researching the documentation. Due to enormous individual differences, completion time is easiest to compare when measured within subjects; hence, presenting both a factory and a constructor version of an API were used instead of separate conditions in two out of the three relevant 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.

5.2.1. TASK 1: NOTEPAD PROGRAMMING

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, specifically the presence or absence of the factory pattern.

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 addition of this last parameter makes the task a good candidate for the use of a factory pattern by suggesting two subtypes of email whose implementation might be hidden by a factory.

5.2.2. TASK 2: EMAIL

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.

5.2.3. Task 3: Thingies

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.

5.2.4. Task 4: PIUtils

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 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.

5.2.5. Task 5: Sockets

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 by calling a factory method on it. The `MulticastSocket`, on the other hand, is a concrete subclass of `Socket` and has several public constructors.

5.3. Results

5.3.1. Task 1: Notepad Programming Results

All twelve participants used a constructor call in their implementation of the method in the first task, using Notepad to write their code. 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.

5.3.2. Task 2: Email Results

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.

5.3.3. Task 3: Thingies Results

Of the twelve participants, two 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 for each participant 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 so that the Wilcoxon Signed Ranks test was determined to be the most applicable 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 5.3 summarizes the times for the Thingies and other timed tasks.

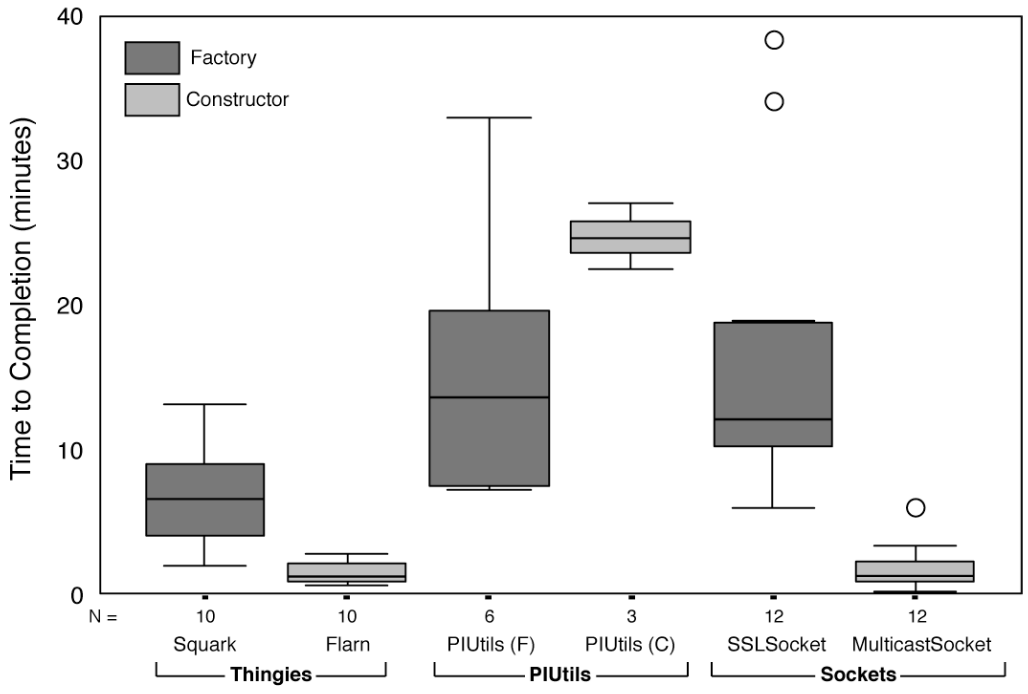


Figure 5.3. The time it took participants to complete each task using a standard box-and-whisker diagram. Circles represent outliers (data points more than three standard deviations from the mean).

5.3.4. TASK 4: PIUTILS RESULTS

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. Nevertheless, the question of the factory pattern's ease of debugging relative to constructors might be an avenue for future work.

5.3.5. Task 5: Sockets Results

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 were due to a failure 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 ($p = 0.005$). Although these times reflect scores rated by a single experimenter, we feel that any errors scoring the task would not be sufficient to challenge the significance of this result.

5.4. 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", in that they expected that to be the way to create objects, and that was the first technique they tried.

As previously mentioned, there exist patterns such as class clusters that can perform the same role in an API as a factory, but without the enormous cost in usability. 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. If, indeed, all or most of the benefits of factories can be addressed by an alternative solution that does not incur such a huge usability penalty, the results of this study suggest that such an alternative would be generally preferable to a factory in most, if not all, cases.

5.4.1. 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”.

The abstract nature of the factory seems to add extra confusion for users. In the Sockets task, every participant was at least briefly waylaid by the fact that `SSLSocket` was an abstract class with no public constructors and no known subclasses, but was nonetheless expected to be instantiated somehow. One participant expressed this confusion thus:

> “SSLSocket is an abstract class and I’m just trying to find what extends this class.
> But it won’t give me any clue.”

Indeed, participants experienced a very strong bias toward trying to find a public subclass of `SSLSocket` rather than looking for ways of obtaining one indirectly. This was due, at least in part, to a particularly unhelpful convention in 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, alternately, 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.

5.4.2. 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 shadowed the method in the parent factory class, `SocketFactory`, and 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 `SSLSocketFactory`s. 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). 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 uttered similar expressions of 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.

5.4.3. INTERVIEWS

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 [Nielsen 1993]. 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 reject this notion, and feel that results that show significant negative impacts on programmer's real ability to use APIs should override such uninformed opinions.

5.5. SUMMARY

Our study finds that the factory pattern universally erodes the usability of APIs in which they are used. There are alternatives with better usability, such as class clusters, which can be used in many situations in which a factory might 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 people's productivity should be avoided. When factories *are* used, we hope that extra care will be taken with the documentation to help users more than current factory documentation, and that new programming tools make it easier to use these APIs.

The study provides:

- Evidence that the use of factories, common to real APIs, instead of factories creates a significant usability barrier to the use of these APIs.
- Evidence that, when not presented with any public constructor, programmers frequently assume that they have the wrong class or need to create a new subtype, *not* that they need to use a factory.
- Evidence that, even when looking for a factory, *finding* a factory is often a difficult and time consuming task in large APIs.

- Evidence that, even after finding the correct factory, *correctly using* a factory (which themselves often lack constructors and need to be instantiated using factory methods) is also difficult.

This study serves as motivation for some of the features in the API documentation tools Jadeite and Apatite, described in chapters 9 and 10, including automatically showing examples of how to create objects.

6.

USABILITY STUDIES OF APIS: OBJECT DESIGN COMPARISONS*

```
mailServer.send(mailMessage);
```

vs.

```
mailMessage.send(mailServer);
```

Figure 6.1. Sample code using two different APIs. APIs that produce similar looking code can be remarkably different in terms of learnability. We compare APIs and show that programmers find the same starting classes and are significantly faster combining multiple objects when the other class they need is referened by a method on the starting class.

This chapter describes a study examining method placement — which class a method belongs to — in APIs that require the use of multiple objects. This study uses techniques from the previous chapters to answer the question of whether and how to separate functionality in an API into one or multiple objects. The study was motivated by our previous observations that combining multiple objects is a challenging part of using APIs, see [Ko 2004] and Chapters 4 and 5. We later refined this observation by noticing that programmers seemed to have particular trouble using APIs in which the object they needed was not referenced by any of the

* Portions of this chapter previously appeared in [Stylos 2008b].

methods on the class they started with. For example, users sending a mail message started with the `MailMessage` class and had a difficult time finding the `MailServer` class (the top code example in Figure 6.1). We hypothesized that: (1) For common tasks, most programmers look for and find the same class to begin their API explorations; (2) Programmers explore a class by examining its methods, and the classes referenced by these methods; and (3) because of the previous two hypotheses, programmers would be significantly faster if the classes programmers gravitate toward as starting points reference other needed classes in at least one of their methods. We designed our user study to test these three hypotheses.

This is of practical interest because in real world APIs like Java's JDK and Microsoft's .NET, it frequently seems to be the case that the classes one needs are not referenced by the classes with which one starts. It has implications not only for API design, but also how to design effective programming tools and documentation for current APIs. Our previous research has shown that the programming language, the development tools, the API documentation, and the APIs themselves all impact programmers' use of APIs.

To test the three hypotheses, we created two different versions of three different APIs. Two of the APIs were based on real APIs in which the class we thought was the most logical starting point did not reference other, needed helping classes. The third task was designed to be domain-independent, and factor out programmers' experience with any real domain or API. We then had ten programmers perform each of the tasks; they were randomly assigned different versions of each API.

In summary, we found that, for the tasks we selected, programmers did indeed gravitate toward the same starting classes, use the methods of this starting class to explore the API, and were significantly faster — between 2 and 11 times faster at the part of the task requiring combining the objects — using the APIs where the starting class contained methods referencing the helper classes (rather than the reverse). Because of the varied nature of our tasks, and because the strategies and work styles exhibited by our participants are consistent with those we have seen in earlier studies, we feel confident that these results will generalize to different APIs as well.

6.1. Other API Design Considerations

This study focuses on the usability considerations of API design (and specifically learnability and discoverability considerations). However, there are other API design considerations such as those relating to performance, implementation and architecture (Chapter 1). In this section we discuss how method placement affects some of these other API design goals.

Our general finding is that it is better for methods to be on the class that the user starts from (e.g., on `mailMessage` in Figure 6.1). However, in some cases, methods might not be placed on the most discoverable class so as to preserve information hiding. For example, it might be desirable for the `MailMessage` class not to know about the existence of a `MailServer` class. This information hiding might be more important in cases where the two classes are on different abstraction layers within the API, to prevent a lower level from knowing about a higher level.

In some cases it might not be desirable to place a method on an interface or on an abstract class. For example, if `MailMessage` is an interface, placing the `send()` method in `MailMessage` would require additional, possibly duplicated, code for all implementing classes, and miss an opportunity for reuse.

Placing a method on two or more different classes — for example giving both `MailMessage` and `MailServer` a `send()` method — has the additional disadvantages of increasing the size of the API and the implementation.

An API design must weigh these and other trade-offs and come to solutions that are appropriate for particular APIs, audiences, and use cases. The purpose of our research is to provide additional data that can inform API design decisions by providing a focused usability evaluation.

In cases where the API design implications of the usability evaluation might not be appropriate, the usability observations can still be used to inform tool and documentation design.

Programming language design also impacts method placement. In some object-oriented languages, methods are not necessarily "owned" by one class but can be equally associated with all of the classes in its signature [Chambers 1992]. This would seem to remove the asymmetric discovery barrier created by current languages' method ownership. However, this also comes at the expense of potentially making it harder to find the methods that are currently owned by only one class by filling up the documentation and code-completion menus with many potentially less-relevant methods. Currently, method ownership is a useful (though imperfect) clue about which methods are most relevant to a class.

6.2. METHOD

To test our hypotheses, we selected real-world tasks from real APIs in which multiple objects were required. However, we distilled down the tasks to be small enough to be feasible to implement in about half an hour with no prior knowledge. We also included an intentionally domain-independent task.

6.2.1. STUDY OVERVIEW

Our study involved ten programmers each performing three small programming tasks. In addition, for two of the tasks, they were asked to first write pseudocode for how they would expect to solve the task (before looking at any real APIs). This allowed us to capture the programmers' expectations about the task independent from the actual APIs. During the programming stage we used the think aloud protocol to capture more information about what programmers were looking for and what their assumptions were.

Participants were given the following study instructions:

> This study involves using Java APIs to perform a series of small programming tasks. We are studying the APIs, not you.
>
> In some tasks, you will be asked to first use a text editor to write the code that you would expect to write to solve the task. Then you will be asked to use Eclipse to write a small program that performs the specified task using the specific APIs (unless otherwise specified). After 30 minutes on each task, you will be asked to continue to the next task so that we can collect observations about as many tasks as possible.

We used screen capturing software to record the contents of the screen and programmers' think-aloud verbalizations. By asking programmers which subtasks they were working on when it was not clear, we were able to use the recorded videos to measure how much time participants spent on different aspects of the tasks.

For the pseudocode writing step, programmers were given only a text editor. For the programming steps programmers were presented with an Eclipse IDE environment and a Firefox browser with the appropriate Javadocs. (Because they were using modified APIs, we told them — if asked — that they should not use other internet resources to find sample code. However, most participants did not ask.)

In the tasks, condition A represented the API closest to the real API (if applicable), in which the task required the use of an object that was not referenced by the class we expected to be found as a starting point. Condition B represented the "fixed" API, in which the class we expected would be used as a starting class did contain a method referencing the helper class. APIs in each condition were fully functional so that participants' programs could be compiled and would actually work.

To create the different conditions, we modified the source code of the original API implementations and used the modified APIs and implementations to generate new JAR libraries and Javadoc documentation. At the beginning of each task, we loaded an Eclipse project with a skeleton class and showed participants how to use the Firefox web browser to access the Javadoc pages for the task. We used Eclipse version 3.3.1 and Firefox version 2 on a MacBook Pro running OS X 10.5 with an external monitor, mouse and keyboard.

The order of the tasks was balanced to account for any learning effects. Participants were randomly assigned to conditions for each task, with the restriction that each participant was given at least one task in condition A and at least one task in condition B.

To test the hypothesis that programmers would find the same starting classes, we did not tell programmers which classes were required to complete the task. However, to limit the scope of the tasks and ensure that participants used the APIs we were interested in, we did tell participants which packages to use (these packages contained between 32 and 61 classes each).

6.2.2. PARTICIPANTS

We used on-campus posters and electronic message boards to advertise our study and get participants. We prescreened participants using an online survey that asked potential candidates about their programming experience and contained a small programming question to ensure sufficient knowledge of Java.

Our ten participants had between one and eleven years of Java experience, with a median of 3 years. All participants were male, and ranged in age from 19 to 26, with a median age of 23.

6.2.3. EMAIL TASK

The email task involved a slight modification of the Java Mail APIs[†]. In the actual API, a `Message` class must be sent using a static method on `Transport` class. In the modified condition B, we added a static `send()` method on the `MimeMessage` class as well.

The instructions for the email task were as follows:

> In EmailTask.txt, write pseudocode for how you would expect to send an email message. Now add code to the EmailTask.java file in the EmailTask Eclipse project to finish this task using the Java Mail APIs in the javax.mail.* packages.
>
> Send the email to ProgrammingStudy@gmail.com with whatever text you please. You may check if the email was received by logging into the ProgrammingStudy account with the password: ********.

The `javax.mail` package and its subpackages contained between 61 non-exception classes.

Programmers were given starter code that set up mail server information in the `Session` object. However, this starter code did not contain references to the `Message` or `Transport` classes. Programmers were given access to local Javadoc files for the Java Mail APIs.

The starter code was as follows:

```
   Properties props = new Properties();
```

[†] JavaMail APIs: http://java.sun.com/products/javamail/

```
props.put("mail.smtp.host", "localhost");

props.put("mail.from", "ProgrammingStudy@gmail.com");

Session session = Session.getInstance(props, null);
```

Possible solution code for the two conditions were as follows:

```
MimeMessage msg = new MimeMessage(session);

msg.setFrom("ProgrammingStudy@gmail.com");

msg.setRecipients(Message.RecipientType.TO, "ProgrammingStudy@gmail.com");

msg.setSubject("Test Subject");

msg.setText("Test message body");

Transport.send(msg);

    OR

msg.send();
```

6.2.4. WEB AUTHENTICATION TASK

The web task was based on a modified version of the Apache Axis2 API for web-services[‡]. In this API, username and password authentication are set by passing an `Authentication` class instance to the `Options` class. To simplify the study task, we extended and repackaged the actual Axis2 API to include a class capable of downloading the contents of webpages in a single operation. In the original API the `Authenticator` was set using a generic method on the `Options` class and a special string flag; we simplified this to a specific `setDefaultAuthenticator()` method to focus specifically on the decision to use the `Options` class.

The instructions for the web task were as follows:

[‡] Axis2 API: http://ws.apache.org/axis2/1_0/api/index.html

> In WebTask.txt, write pseudocode for how you would expect to show the html contents of the password protected page http://www.jsstylos.com/protected/test.html on the console. You may test access the page in a web-browser, using the username "username1" and password "password1". Now use the WebPageTask.java file in the WebTask Eclipse project to print out the contents of this webpage using the APIs in org.apache.axis2.transport.http.* packages.

The `org.apache.axis2.transport.http` package contained 32 classes, at most three of which were needed to complete the task.

Possible solution code for the two conditions were:

```
WebRequest webRequest = new WebRequest();

webRequest.setUrl("http://www.jsstylos.com/protected/test.html");

Authenticator authenticator = new Authenticator();

authenticator.setUsername("username1");

authenticator.setPassword("password1");

Options.setDefaultAuthenticator(authenticator);

    OR

webRequest.setDefaultAuthenticator(authenticator);


System.out.println(webRequest.getPageContents());
```

6.2.5. THINGIES TASK

The Thingies task was designed to test the API pattern outside of programmers' expectations from any particular domain. To this end we modified the names of a real API with nonsensical names as in previous studies (Chapters 4, 5). Unlike the other tasks, programmers were given a starting class and were not asked to write pseudocode. In condition B of the Thingies task, participants were required to find an object that could "bless" their `Foo` instance in a package of 53 nonsensically-named classes.

The instructions were as follows:

> Use the ThingiesTask.java file in the ThingiesTask Eclipse project to write a program that successfully calls the runMe() method on a Foo object in the Thingies package.

Simply creating a new `Foo` object and calling the `runMe()` method resulted in a runtime exception saying that the instance of the `Foo` must be "blessed" before calling `runMe()`. In condition A there was a `bless` method which took as an argument an instance of a `Narn` object, making it necessary to instantiate a `Narn` object to bless the `Foo` and successfully call `runMe()`. In condition B the `Narn` class had a `bless` method, which took an instance of a `Foo` as an argument.

Possible solution code for the two conditions were:

```
Foo foo = new Foo();

Narn narn = new Narn();

narn.bless(foo);

   OR

foo.bless(narn);


foo.runMe();
```

6.3. Results

Our primary result was that, for each task, programmers were dramatically and significantly faster — between 2.4 and 11.2 times faster — at combining multiple objects in condition B, in which the class we anticipated as being used for exploration included a method referencing the required other class. To reduce variance, we factored out the time participants spent finding a starting class. The results for the three tasks are shown in Figure 6.2.

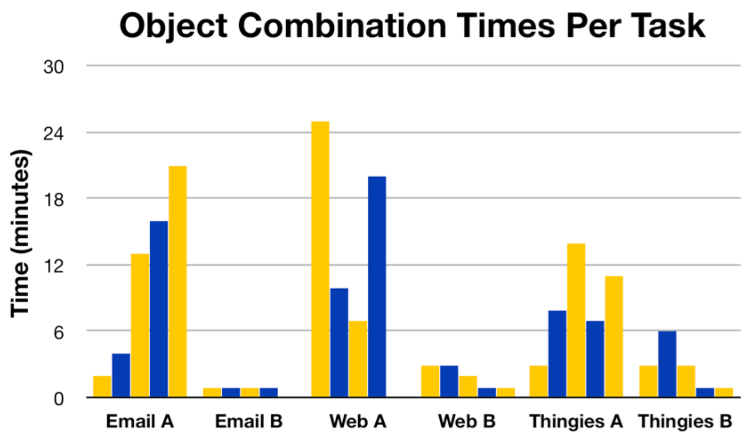


Figure 6.1. Each bar represents the object-combination time spent by a participant. In condition A tasks, a helper class contained a required method. In condition B, this method was placed on the main class. (Colors are just to make the bars easier to differentiate.)

In condition B, all programmers finished all tasks. Two of the five participants in the condition A email task and two of the five participants in the condition A web task failed to finish in 30 minutes; in the analysis, we used their time spent combining objects up until the time limit.

In the email task, participants spent an average of 11.2 minutes finding and using the `Transport` object to send the email message in condition A, and an average of 1 minute sending the email message in condition B.

In the web authentication task, participants spent an average of 15.2 minutes finding and using the appropriate authentication classes in condition A, compared to an average of 2 minutes in condition B.

In the thingies task participants spent an average of 6.8 minutes finding and using the `Narn` object to bless the `Foo` object in condition A compared to 2.8 minutes in condition B.

Because the timing data exhibited both ceiling and floor effects, we used the Wilcoxian Rank Sum method for computing statistic significance. We found statistically significant ($p < 0.05$) differences between conditions A and B for each of the three tasks.

We did not see a statistically significant effect of task order or individual participant programming experience on task completion times, showing that there was sufficient counter-balancing.

We also found evidence that, for the tasks we selected, programmers found and used the same classes as starting points. In the email task, all ten of our participants found and instantiated or read the documentation for the `Message` class before finding or reading the `Transport` documentation (no one started in the `Transport` class and then found the `Message` or `MimeMessage` later). In the web task, eight of our ten participants found and instantiated or read the documentation for the `WebRequest` class before examining or instantiating the `Authorization` or `Options` classes. This is true despite the fact that we did not tell programmers which classes to use. Along with the results from the participants' pseudocode, this suggests that the classes programmers find and use to explore the API are strongly influenced by programmers' expectations and the names of the classes in the API.

Based on the pseudocode written by our participants before seeing the actual APIs, all of our participants expected to be able to call a `send()` method on the same class they used to represent the email. Eight of the ten participants expected to be able to specify the username and password in the same object used to request the contents of the password-protected webpage.

An example of pseudocode one participant wrote for the email task was:

```
emailsender sr =new emailsender();

sr.setemailid(emailid);

sr.setsubject(subject);

sr.setserver(server)

sr.setmessage(message);

sr.send();
```

An example of pseudocode another participant wrote for the webtask was:

```
HTTPGrabber webGrab = new HTTPGrabber();

webGrab.setURL("http://www.jsstylos.com/protected/test.html");

webGrab.addHeader("username", "username1");
webGrab.addHeader("password", "password1");

String s = webGrab.fetchURL();
```

6.4. Discussion

The APIs in which the methods were on helper objects were harder to use because:

- Not finding an expected method, participants would sometimes question their (correct) choice of starting class;
- Programmers had to recognize that the use of an additional class was required;
- Programmers had to locate the additional class.

The common strategy programmers used to find a starting class was to browse the class list in the package documentation of the Javadocs. Based on the seeming relevance of a class name they would then visit the Javadoc for that class. In the class documentation, participants used the short textual summary and also the list of available methods to help determine if the class could help them solve the task. If it looked potentially relevant but did not seem to contain all of the necessary information, they would sometimes explore the class's interfaces or subclasses, but more commonly would go back to the package list to browse for another class. Several of our participants performed similar explorations using Eclipse's code completion as the primary method instead of the Javadocs. The Eclipse workspace was set up so that the Javadocs were linked with the jar files, so that the mouse-over tool-tip of a class or method would show the Javadoc documentation.

Most participants browsed the package list and class documentation from top to bottom, rather than using search or scanning to check if a particular name occurred in the alphabetical list. However, several of our participants used Firefox's in-page search function to look for specific keywords in the Javadocs. This was sometimes a successful strategy, even when class names did not exactly match the search term. For example, search for "send" in the mail documentation found the `SendFailedException` class, which referenced `Tranport.send(Message)` in its "See also" list. One participant chose to enable Firefox's "Highlight all" search parameter to visually reveal all the instances of his search terms on the page. It is possible that web documentation that more directly supports search — such as Microsoft's MSDN — would prompt programmers to make greater use of searching.

Although the Javadocs included a "Use" page for each class, which listed all of the classes that reference the selected class and so included the needed helper class for these tasks, only one of our participants ever looked at a Use page. To support tasks like the ones in this study one might argue for more prominent display of this information. However, for some classes, the Use page contained more than a hundred different references, which are all presumably relevant to some use of the class.

Most of the participants' pseudocode was of the following form:

```
Object obj = new Object();

obj.setProperty(value);

...

obj.callActionMethod();
```

Based on our previous studies, this seem to be a commonly expected form of API for Java and C# developers. Several of the unexpected difficulties we have observed programmers having are as a result of APIs not following this simple model, for example by not providing a default constructor (or any public constructor), requiring the combination of multiple objects, or by requiring the use of subclasses. It does not seem feasible to be able to provide such simple and high-level classes for every possible task — much of the rich expressiveness of APIs comes from being decomposed into multiple parts that can be assembled in new ways. However, this model would seem to be a standard of simplicity to which APIs might aspire for the most common tasks.

In terms of the cognitive dimensions [Green 1996][Clarke 2004], the results of our study can be seen to reveal barriers stemming from visibility and hidden dependencies. While not hidden in the resulting code, the dependencies between a class and a class that acts on it are effectively hidden by the tools and documentation as used by our participants. Programmers' difficulties finding related classes may also be seen as a challenge in progressive evaluation. Having coded an incomplete solution, the API offers little direct feedback on how to complete the task.

6.4.1. Email Task Discussion

The email task used APIs that were the least modified from the actual public APIs; in condition A, participants did use the real public APIs directly. Because of this, there were several additional API complexities that presented barriers for participants.

All of the participants in our study examined the abstract `Message` class before finding the concrete `MimeMessage` class. Most participants found the `Message` class by browsing the list of classes in the `javax.mail` package Javadocs. Many of the participants attempted to instantiate an instance of the `Message` class, even after recently viewing documentation stating that the class was abstract.

6.4.2. Web Authentication Task Discussion

In the web task, unlike in the other two tasks, neither the two main required classes — `WebRequest` and `Authenticator` — directly referenced the other. Instead, the `Options` class referenced the `Authenticator` class, and the `WebRequest` class implicitly used the `Options` class. Surprisingly, however, combining these two classes did not take participants significantly longer than the objects in the other two tasks. This might be in part because none of the 30 classes provided were obvious starting points, and participants reverted to a brute force examination of every class. We restricted participants to this package and chose this package size to ensure that the task would be feasible and focus on the APIs we designed, however this made the exploration simpler than in actual tasks. In a programmer's real work, without instructions specifying which package to use, searching all of the possible packages for relevant classes would likely be even more time-consuming.

6.4.3. Thingies Task Discussion

Surprisingly, participants were faster at recognizing the need for and finding the required helper class in the Thingies task, in which the classes did not have sensible names, than they were in the two other, more sensible tasks. The reason for this seemed to be that — lacking any semantic clues from the API names — programmers reverted to a comprehensive hunt for a class with a relevant looking method. In the other tasks, programmers spent more time trying to understand the classes, which ended up taking more time. However, this strategy only worked in the thingies task because the package was small enough that programmers could manually scan each class. In a larger and less bounded API, this strategy would likely be less effective.

6.5. LIMITATIONS

When studying API usability, it is important to identify the scope with which our results might be generalized and the potential limitations of our study.

The participants in our study were all PhD, masters, or undergraduate students. However because the work and exploration strategies exhibited by our subjects matched those observed in previous studies with more varied participants (Chapter 4), we feel that the results will generalize beyond this population, at least to other programmers exhibiting the common “pragmatic” and “opportunistic” work-styles (Chapter 1). Programmers in our study exhibited one of these two personas, characterized by bottom-up and learn-as-you-go coding techniques.

The three tasks in our study were smaller than most realistic programming tasks, so that we could test more tasks and to avoid extraneous task complications. There could be more complicated effects of method placement within larger and longer tasks, and it is not yet clear what the time impact would be on programmers doing their own tasks. Because of the similarities in work strategies we saw across our tasks, and because programmers often approach larger programming tasks by focusing on smaller subtasks, we feel that our results will generalize to different and larger tasks. Because of all of our tasks focused on creating or modifying code, we cannot say what the usability implications of method placement are on other tasks such as reading or debugging code.

During the study we had the competing goals of having programmers work in a realistic manner to get accurate timing information and to gain insight into programmers’ thoughts and assumptions while they worked. We chose to use the think-aloud protocol, which likely affected their times. However, because we used the same protocol in both conditions, we think that the relative time comparisons between the two conditions are still valid.

Because we used modified versions of real APIs, participants in our study were not able to use internet resources to find sample code, a common starting strategy when learning a new API. However, because it is often not easy or possible to find an appropriate sample for the right version of the right API, we feel that the exploration of APIs without sample code is still a useful indication of its usability.

The programming tasks we used were in Java, and the documentation we provided used the standard Javadoc format. A different documentation style might lead to different programmer behavior; for example, documentation that emphasized searching might make programmers less likely to browse classes, instead guessing relevant search terms. However, we expect that programmers would find similar starting points whether by searching or browsing, and have similar difficulties after they find these starting points.

6.6. IMPLICATIONS

Because we directly compared an API solution to the issue of method placement, the most direct implication from the results of our study is that changing the APIs would directly benefit programmers. However, the problems we observed our participants having with the original APIs could also be addressed in fixes to the documentation or the developer tools.

6.6.1. API DESIGN

One question when trying to address API method placement usability issues in an API is whether we can automatically identify potential problems. Identifying and fixing potential method placement usability issues across a large API would require empirical data on: the most common tasks for any given part of the API; the classes programmers select as starting points to try to accomplish these tasks; and the code the programmers end up writing. Given these three things it would be possible to approximate the exploration difficulty of a given task by using a graph of which classes reference which other classes in their type signature. API designers might have information about tasks, starting points, and sample code while designing an API. In these cases, it would be possible to manually inspect the sample solutions that some API designers recommend creating before the actual API [Bloch 2001] [Cwalina 2005] to identify potential discoverability barriers before an API is released, potentially improving method placement.

One additional issue highlighted by this study was the difficulty programmers have finding classes that are useful to begin exploration of an API. Placing methods on a main class still does not solve the problem of easily finding the starting class in the first place. As with other API usability issues, this could be addressed by changes to the API, documentation or tools. Existing API recommendations all point to class naming as a critical aspect of usability [Cwalina 2005], and our research confirms this. Based on the think-aloud, participants in our study seemed to primarily use the class names in the list of classes in a package to decide which classes to look at. Cwalina et al. recommend reserving the more general, recognizable class names for common and instantiatable classes [Cwalina 2005]. The classes in the email task are an example of how *not* to do this: the attractive name `Message` is used by the abstract class, while the instantiatable class is named the more obscure `MimeMessage`. In addition to using recognizable high-level class names for common classes, our study suggests that giving common classes names that start early in the alphabet when possible might also be helpful.

6.6.2. Documentation Design

An alternative solution that we explore in the coming chapters is to leave the APIs as they are and to change the documentation to help programmers more easily find related classes. One simple solution would be to use the @see Javadoc tag to reference more related classes. However, as revealed by the pseudocode written by our participants, programmers often expect to need only one class, and so a documentation solution would have to not only make it easy to find the related class but find an effective way of helping programmers realize that they need another class. Another potential solution would be to add more descriptive textual documentation. However, programmers in our studies often skim or completely skip over textual class documentation, choosing to refer to the list of methods and fields instead. Chapters 9 and 10 explore more ways to address this problem.

6.6.3. Tool Design

Development tools such as the Eclipse and Visual Studio IDEs could also do more to support programmers using multiple objects and finding methods on helper classes. Current IDE features like code-completion and class hierarchy browsers make it easy to see the methods on a given class but much harder to find other, related classes. Newer research tools like Strathcona [Holmes 2005] might help by showing programmers relevant example code that includes helper classes. However, these tools typically require large example repositories and are potentially more complicated and heavyweight than features like code-completion.

6.7. SUMMARY

This chapter presents results from a study showing that programmers were faster using APIs in which the classes from which they started their exploration included references to the other classes they needed. We hope that API designers will consider this knowledge along with their other API design goals to help create APIs that make it easier for programmers to perform common tasks. We hope also that the designers of programming environments and API documentation will use this observation to help create tools and documentation that help programmers more easily and simply find related classes necessary for common tasks using their natural exploration strategies. And we hope our series of studies will inspire others to study even more aspects of the usability of APIs, so that usability can be an important consideration for all future designs.

From this study we can conclude:

- APIs should be designed so that the most discoverable class can easily be used as an exploration point from which to discover other, helper classes.
- Programming tools and API documentation should make it easier to discover the existence and use of helper classes and methods.

Chapters 9 and 10 will describe the Jadeite and Apatite API documentation system, which take inspiration from the results of this study and offer multiple ways to mitigate problems with real APIs.

7.

Usability Studies of APIs: A Case Study at SAP[*]

[*] Joint work with Benjamin Graf, Daniela K. Busse, Carsten Ziegler, Ralf Ehret and Jan Karstens at SAP. Portions of this chapter previously appeared in [Stylos 2008a].

After publishing the studies described in chapters 4 and 5, I was invited to intern with researchers at SAP to help them study the usability of a specific API that they were developing. This experience provided an opportunity to test of the techniques we had developed on a specific API and, after having worked with Microsoft, gave me another opportunity to collaborate with API designers, giving me a better understanding of their perspective and letting me share the results of our research.

7.1. Introduction

As a market leader in business software, SAP employs roughly 18,000 software developers. One of the key elements of their productivity is being able to quickly and effectively reuse code that colleagues have already written. APIs explicitly allow for code to be reused by other developers; however previous work (Chapters 4, 5) and experience at SAP has shown that using APIs that do not meet the developers' specific requirements can often be tedious, difficult, or even impossible.

To better understand this problem and explore possible solutions, we examined a specific API — the SAP "Business Rules Framework Plus" API (BRFplus) implemented in the ABAP programming language. We used this API as the focus of a case study in which we developed and applied a process for studying and improving the usability of APIs. We chose to study the BRFplus API because of its importance to SAP (BRFplus recently replaced all 26 previous rules engines to provide one common and standardized business rules framework to all of SAP), and because some users had reported difficulty using it.

By encapsulating business rule functionality, the BRFplus API allows additional functionality to be specified by business users, who know the details of what the software should do, instead of software developers (see Section 7.2). This provides the potential for powerful customization of software at the hands of the business user, enabled by BRFplus. By making BRFplus easier to use, we hope to bring it to more areas of SAP, providing more flexibility and value to SAP's customers.

We did a user-centric design of an API wrapper (using input from stakeholder interviews, user requirements gathering sessions, and a pseudo-code study) and then performed a usability evaluation to assess its value in terms of helping application-level developers address their specific use cases more easily.

Our overall goals stretch beyond the BRFplus API. We hope to demonstrate within SAP the value in focusing on API usability. SAP has developed and applied user research for user interfaces and other areas, but — like most companies — has not previously examined the usability of APIs with a systematic user experience effort. Doing this will help SAP create APIs that are easier to use, improving the productivity of SAP's developers and customers. In doing so, we hope to advance the state of art in API usability and contribute our insights, results, and techniques back to the research community.

7.2. About Business Rules

Business rules [Ross 2003] allow end users (non-programmers) to specify software behaviors that would normally be specified at software development time by a developer. This allows software to be customized by users after the software is deployed. For example, business rules might be used to specify how much tax is computed on different items, since this varies by location. An end user might do this by using a graphical user interface (GUI) to specify that for "food items" the total tax is 5% of the base price, and "other items" are taxed at 7%, potentially providing more detailed specifications for food and other. This differs from "traditional" programming, where a software developer would write a function with an "if" or "case" statement to compute the tax, and the code would have to be recompiled when the logic changed.

Business rules can similarly be used to compute how large each employee's end-of-year bonus should be, or how to specify rules used during the approval process of expense reports. After being specified by a user, these rules can be automatically executed, like software. In this sense a business rules API and accompanying GUI provide an environment for end user programming.

7.3. Identifying User Requirements

The BRFplus API we examined allows for the creation and editing of business rules. The original API was primarily intended for internal SAP developers for creating platform level code – i.e. code that hundreds of SAP applications and solutions would be building on. This requires the API to give the developer a significant amount of control and transparency in using the API. Initial interviews with stakeholders and domain experts at SAP brought to light, however, that also more and more application developers were using the API directly for their coding tasks – and that these users were increasingly running into difficulties using the API, as it was not designed to meet their specific (high level) needs. We received consistent feedback from our initial stakeholder interviews that, while they had designed a powerful and flexible API (as required by platform users), many current users struggled to implement their simpler use-cases.

In terms of the cognitive dimensions [Green 1996], the interviews led us to suspect that there was a mismatch of abstraction level. The API was seen as providing low-level functionality (and was thus very flexible) while the specific class of application developer users had higher-level goals, prioritizing simplicity of use over control and transparency – their use cases were not as complex as to require significant amounts of granularity or flexibility, and the time spent in understanding and debugging the API's use was very limited compared to platform developers.

The difference in user roles can also be expressed by previously identified developer personas (Chapter 1) that describe different styles of programming with different strategies and goals. We used these personas to help understand different developer strategies in our research. The system level developers at SAP, who help write the platform that others rely on, tend to exhibit traits of the systematic programming persona, cautiously understanding the implications of each line of code. The application level developers tend to act more pragmatically, wanting to understand and tweak the code but also be productive. Consultants, under the most time pressure, often exhibit traits of the opportunistic programmers.

In order to validate these initial hypotheses based on our stakeholder interviews, we ran six 60-80 minute individual requirements gathering sessions with existing BRFplus users. In these sessions, we asked the developers to first explain the overall purpose of their code that used business rules, and then to explain which parts of the BRFplus API they used, and how. We also discussed how they might have wanted to use BRFplus but were not able to.

In our interviews the API users we talked to did indeed have relatively simple use cases, compared with the flexibility of BRFplus.

For example, one developer used BRFplus for Manufacturing Execution. The automatic generation of production orders as well as the guidance of material lots through a shop floor was to be governed by rules. For instance, a rule might specify that scheduling of certain work packages on the shop floor (e.g. those concerning Product X, or from Supplier Y) should follow a specific scheduling strategy – with those strategies themselves being sets of rules, or a formula). However, rather than describing the full rules set using BRFplus (which would be possible using a nested construct of tables and formula rules – one of the more complex features in BRFplus), the developer instead used BRFplus only to return a "strategy-code" that referred to a rule that he hard-coded outside of BRFplus. This made the business rules easy for business users to execute, with a loss of generality that was seen as acceptable for this use case. But it also limited the ability of the business user to specify rule changes at run-time, having to rely on the hard-coded values predefined by the developer.

This is an example of what we saw with a number of uses of the BRFplus API: While the overall scenarios often included computing values from formulas and triggering actions based on rule results, the users we interviewed moved this functionality outside of the rule system for simplicity, having a developer specify the formula or action rather than an having an end-user specify it in the rule itself.

7.4. REDESIGN STRATEGIES

In previous API usability research [Clarke 2004][McLellan 1998] (Chapters 4, 5), the results of API usability studies have been used to improve the API before its final release. The API we examined had already been (internally) released and used, and so the team had to continue to support it. We could not simply create a new version and ignore the old API, however we had several different options about what to do. For example we could adapt the BRFplus API in backwards-compatible ways (which would limit the changes we could make) or we could create a completely new API (at the cost of having to support both of them). Because of the specific abstraction level problem we had identified, we chose to design a "wrapper API," a higher-level API implemented on top of the original API. This would also enable programmers to choose which level of granularity they wanted to interact with the API on, the wrapper, or what was underneath. This technique was used by Microsoft's Foundation Classes, for example, which built on top of the existing Win32 APIs. Other scenarios might well call for other types of solutions, such as adding additional classes at the same level or even lower levels of abstraction.

7.5. USING PSEUDOCODE TO REVEAL USERS' EXPECTATIONS

Before designing a wrapper API we first wanted to learn how users thought, what (if any) mental models they had, and what terminology they used. To do this we designed a study in which existing and prospective users would write pseudocode against an imaginary business rule API using a simple text editor. We adapted and extended this technique from our previous work examining API design choices (Chapters 4, 5).

We contacted several users of the BRFplus as well as other developers who were knowledgeable about business rules, and conducted another round of six different one hour sessions. Because the volunteer participants willing to participate were mostly remote colleagues in different locations, we conducted all of our sessions remotely via telephone and Windows NetMeeting to see their screen. This setup allowed us to discuss privately by muting the speakerphone.

We emailed participants study instructions immediately before the study that briefly described our project and gave them a scenario to write pseudocode for. Participants were told they could write ABAP or Java-like pseudocode. During the study we asked participants to think aloud.

We asked participants to write code for the task of defining the rental car price for customers based on their age and the rental duration. We chose our example to be representative of the application-level use cases we saw in our interviews that would be easily understandable.

The pseudocode that the six participants came up with showed many similarities to each other. Some of the results were:

- Participants wrote code at a high level of abstraction, reinforcing what we had learned in earlier interviews. Participants wrote in a dozen lines what would take a hundred or so with the BRFplus.
- Participants tended to separate the structure of rules from the data in their code. For example, users would first specify that a number, “price”, should be associated with an age and a duration, and then set the specific values. In the current BRFplus model, there is no separation between rule structure and data.
- Most participants used tables to model the structure of their rules. This can possibly be put down to the fact that the most efficient data structure for handling sets in ABAP are internal tables.
- Participants omitted details such as locking data structures for thread safety, versioning and activation, and explicitly saving rules. These details were required to support more complicated scenarios, but could be handled automatically in the simpler cases.

7.6. Usability Evaluation

After we had designed and implemented a prototype version of the wrapper API we designed a think-aloud study to evaluate the wrapper design. Our goals were to find areas of improvement and to see if participants would be able to use the API at all without much documentation. We chose to provide almost no documentation because we wanted to avoid hiding unusable aspects of our design behind good documentation.

We based our API evaluation study design on the previous work on evaluating early API designs (Chapters 4, 5). In the study, participants wrote code that used the wrapper API to implement a series of up to three tasks, as time allowed. Task one involved creating new rules for how many vacation days different employees in different countries should get (e.g. German, full time employees get 30 days vacation a year). Task two involved storing these rules and then loading them to calculate the vacation days a new employee should get. Task three involved creating an additional rule involving ranges (e.g. German, full time employees who have worked less than one year get 20 days vacation). These tasks were a simplified version of a real use-case.

We then performed a third round of 60-90 minute sessions, this time to study the usability of the wrapper API. Three of these sessions took place in a usability lab with a one-way mirror separating observer and participant rooms. The other sessions took place remotely, using a phone and Netmeeting for screen sharing. We maintained a connection between the participant's computer and the observer's computer and used screen-capturing software to record the session on the observer's computer.

To better understand participants' behavior, we asked them to think out loud as they worked. A consequence of this is that the time they spent to complete a task could have been affected by speaking out loud. This was especially likely since few of our participants were native English speakers and we asked that they speak their thoughts in English (the only common language among the project members).

We gave participants brief written study instructions and documentation giving a one-to-two sentence description of each class in the wrapper API.

The high level results of the study were that five out of the six participants were able to finish the first task in 90 minutes, four out of six were able to finish the first two tasks, and three were able to finish all three tasks within 90 minutes. Because of the limited documentation and time, we felt that having most participants finish at least one task was a positive reflection of usability of the API. We also observed some common difficulties using our API; fixing these might allow more participants to finish all of the tasks.

We had initially considered performing a comparative study between the wrapper API and the original API. However, the BRFplus felt that the results of the wrapper API evaluation were so strong that using it was clearly faster and it was not worth having participants perform the same tasks with the original API, which the BRFplus team did not feel would be practical in 90 minutes. To get an upper bound on the usability of the original API, we had a developer of the BRFplus team with intimate knowledge of the API perform the same study tasks using the BRFplus instead of the wrapper API. The BRFplus developer was able to solve all of the tasks, but required 120 minutes, because of the additional details required to keep track of in the BRFplus as opposed to the wrapper API.

7.7. DESIGN ITERATION

Overall, participants were able to use the wrapper API easily. However our study revealed several, relatively minor, usability problems with the wrapper API design. Based on these observations, we created a revised version of the wrapper API. If time had permitted, we would have then run more study participants with the revised API to ensure that these changes solved the problems we observed and did not introduce other problems.

We had assumed that participants would be familiar with the "`range`" object in ABAP programming language. These are used to create rules with `ranges`, for example that employees who have worked between 1 and 2 years should get a certain number of days vacation. However, we found that while most participants had heard of it, few knew how to use this construct. Based on this, our revised API included a convenience function for adding rule ranges.

In our initial design we used two separate objects for creating rules and for using the rules to compute a value based on some input. We chose this because rules can contain ranges, while the concrete values used for rule instance processing cannot. However, participants had some difficulty understanding the distinction between these classes, so our revised wrapper uses the same class for both of these operations.

7.8. SUMMARY

We did a user-centric design of an API wrapper and then performed a usability evaluation to assess its value in terms of helping application-level developers address their specific use cases more easily.

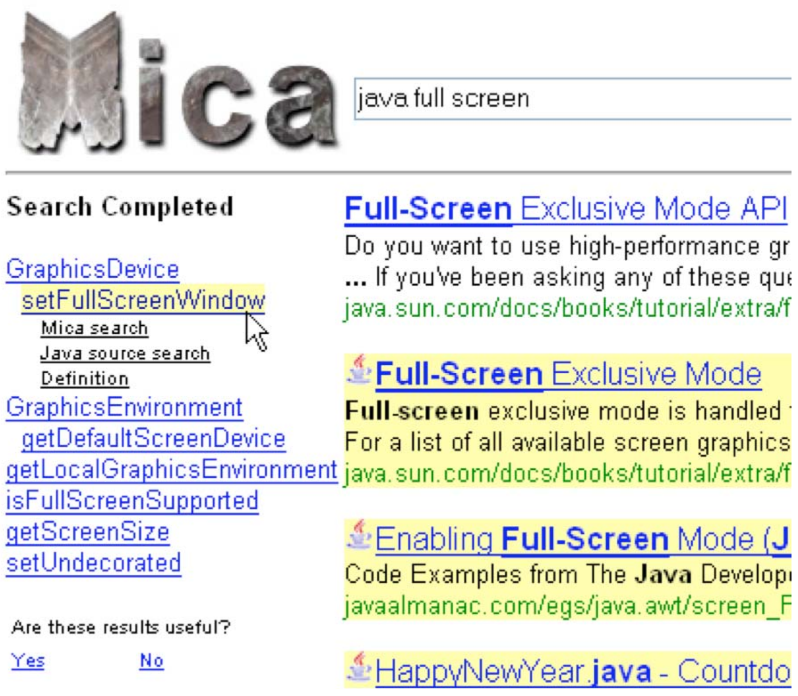
This work shows that:

- API usability is a problem in real APIs.
- The methodologies described in earlier chapters can be used to identify usability problems and solutions.
- Wrapper APIs are a powerful solution to some types of API usability problems.

This project showed that, while not trivial, an API can be investigated, designed and evaluated (in prototype form) in three months with a handful of people and the help of current and prospective users.

8.

PROGRAMMING TOOLS: MICA[*]



Mica

java full screen

Search Completed

GraphicsDevice
setFullScreenWindow
Mica search
Java source search
Definition
GraphicsEnvironment
getDefaultScreenDevice
getLocalGraphicsEnvironment
isFullScreenSupported
getScreenSize
setUndecorated

Are these results useful?
Yes No

Full-Screen Exclusive Mode API
Do you want to use high-performance gr
... If you've been asking any of these que
java.sun.com/docs/books/tutorial/extra/f

Full-Screen Exclusive Mode
Full-screen exclusive mode is handled
For a list of all available screen graphics
java.sun.com/docs/books/tutorial/extra/f

Enabling Full-Screen Mode (J
Code Examples from The **Java** Develop
javaalmanac.com/egs/java.awt/screen_F

HappyNewYear.java - Countdo

Figure 8.1. The Mica web application. Mica includes a keyword sidebar on the left, which is generated from web search results shown on the right. Search result pages are categorized by their content; the Java icon indicates that those results contain code.

Mica (Figure 8.1), the first tool we created to help programmers use APIs, was motivated by observations of programmers in which we observed programmers

[*] Portions of this chapter previously appeared in [Stylos 2006a].

using Google to overcome their difficulties learning the right vocabulary for an API. While not perfect, using Google sometimes was able to handle the large scale of APIs in a way that the other electronic forms of documentation were not. For example: a search for "time" on Microsoft's MSDN help returns more than 500 separate documentation items.

We performed a formative study, in which the programmers we observed found web search especially useful for APIs, and were often able to use web search to overcome each type of learning barrier. They were able to do so because of the wide range of documents that the search engines indexed – documentation, forum discussions, code snippets and the source code for full programs – and also because of the effective ranking algorithms that made the most useful and relevant information in very large collections most likely to be found. This helped programmers overcome selection barriers by finding the right terminology from their naïve phrasing. This worked because the search engines indexed forums and other pages on which people had described a problem or solution using the same, incorrect, terminology.

However, while search engines were a popular and often a valuable tool for programming help, programmers encountered a number of problems and inefficiencies using them. One challenge was that the documents that programmers spent their time looking at were often not relevant. Programmers would get frustrated when they scanned long result pages only to find that they did not contain the source code they were looking for, or that they contained only source code and not the higher level documentation they needed. Sometimes the results had nothing to do with programming at all. Even when a search did yield some relevant results, if the first few documents programmers browsed did not seem relevant, they would often give up and try another path. The less familiar programmers were with the domain, the less successful they were at predicting how relevant a document would be.

The difficulties that programmers experienced are not surprising given that web search engines were not designed to specifically support the programming task. While there are several simple improvements that could aid programmers, such more control over searching of punctuation marks and other programming syntax, substantially improving programmers' experience requires a new type of tool that uses knowledge of programmers' behavior to provide more effective online programming support.

Mica (Making Interfaces Clear and Accessible) is a prototype tool we designed to help programmers more effectively and efficiently use web search to learn how to use APIs (shown in Figure 8.1). It does this by providing the following cues about the information contained in the result pages: (1) Mica displays relevant API methods, class and field names that are contained in the search results. (2) Mica orders the field names based on their frequency and correlation with the web results, and by the structural containment of classes and methods. (3) Mica displays icons that indicate whether a page contains source code and whether it is an official documentation page. (4) Mica provides keyword-relevant result summaries on demand.

Mica is available online at: http://www.cs.cmu.edu/~mica.

8.1. MOTIVATING OBSERVATIONS OF PROGRAMMERS

Mica's design was motivated by observations of programmers using Internet resources, described in this section. We studied programming projects in various stages of creating new functionality and observed how the programmers used Internet resources to support their programming. In three case studies we found common patterns of usage, and different situations where the effectiveness of the tools broke down.

While the information seeking literature includes some studies of how programmers [Berglund 2002] and others [Choo 1999][Marchionini 1995] use internet resources, these mostly look at more formal documentation, with an emphasis on implications for documentation writers. We were interested in also looking at how programmers used informal resources, such as forum and Usenet posts and independent websites with sample applications.

The Information Foraging theory [Pirolli 1995] explains people's search behavior in terms of "information scent," cues that indicate how useful a path will be. We were interested in finding out which cues were most helpful to programmers in finding relevant information.

We observed three small programming projects in Java by computer science graduate students and a collection of screen-captures of Java programmers collected for another study [Ko 2005]. The projects involved creating a new GUI Java application, creating an Eclipse plug-in, and modifying an existing but unfamiliar open source application.

8.1.1. OBSERVED STEPS OF API LEARNING

In observing the programmers, we noticed several different stages of API learning and transitions between these stages (shown in Figure 8.2).

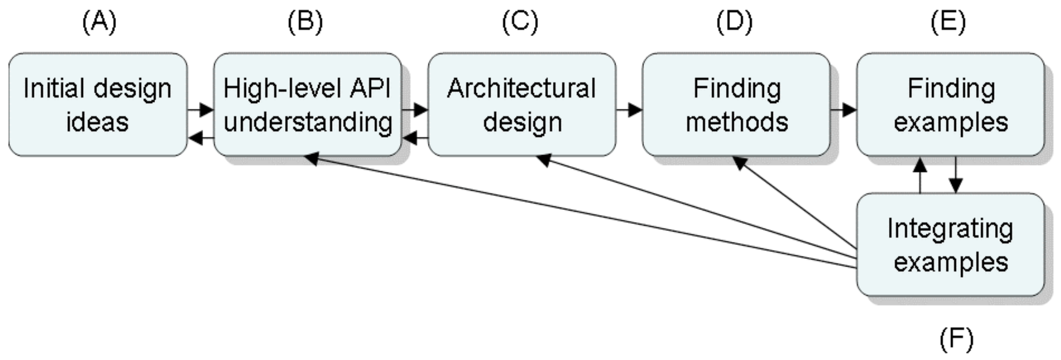


Figure 8.2. High-level programming activities observed in our study. Internet resources were used in steps (B), (D), (E) and (F); Mica is designed to help with these steps.

Each of the programmers started with an initial idea of what their application was to do (A). Only after getting an overview of the structure of the APIs they would use (B) did the programmers begin to design how they would implement their application (C). These initial design ideas would sometimes require further high-level API understanding (B). Once they had a high-level design, they looked for specific methods that could accomplish their task (D). Once they found the name of a method in an API, they searched to find out exactly how to use it, using documentation and example code (E). They then integrated this example code into their own program and see if it accomplished what they wanted (F). If it did not, they would find new examples of how to use the same methods (E), look for new methods (D), redesign their architecture (C) or look for different APIs (B).

8.1.2. INTERNET RESOURCES USED

Each of our observed programmers used Google as their primary resource for finding programming information on the Internet, with one also using Google Groups, Google's search engine for Usenet archives. The programmers found a variety of different types of resources using Google, including tutorial pages such as those on http://java.sun.com, documentation such as the Java SDK Javadoc pages, overviews and articles on software architectures, webpages with example programs, and forum posts with questions, answers and code snippets.

8.1.3. HIGH-LEVEL API UNDERSTANDING

Because the programmers we watched already had a good idea of what program they wanted to create (A), the first step we observed was that of getting an overview of which APIs they needed to use and those APIs' overall structure (B). Each programmer's first step in this task was to pose a general query to Google. For example, the programmer writing a Eclipse plug-in searched for "refactoring plugin eclipse". Google was effective at finding tutorials, high-level articles (such as those provided by IBM and Eclipse.com about Eclipse) and sample projects with source code and documentation that the programmers found useful.

One of the reasons that Google was effective at this task was because it was worked well even with the use of non-expert terminology. For example, the novice programmer creating a simple Java application was able to find information about creating windows and widgets using the search "creating a form in java," even though the word "form," which the programmer was familiar with from Visual Basic, is not often used in Java programming.

8.1.4. DISCOVERING WHICH API METHODS TO USE

When programmers had begun to understand the high-level elements of the APIs they were using (B), they then formed an idea of how they planned to implement their application (C). Their next action was to determine what specific classes and method calls they could use to accomplish specific tasks (D). Often this step was combined with the previous step (B) as the tutorial or article would include specific method references.

Programmers used different strategies in this step, including using Google with a description of the desired functionality and browsing the list of classes in the JDK's Javadoc documentation. Because Google indexes many documentation sources, including the JDK's Javadocs, searching with Google could often double as a search of the official documentation. In addition, when programmers found a potentially useful method, they would often look up its official documentation to verify that it did what they thought.

When programmers performed a web search based on a description of the desired result, they would open and scan some of the resulting pages. They looked for code or words that seemed like they might be method or class names – looking for cues such as a fixed width font or programming punctuation. Often they spent time looking at pages that were not related to programming or which did not have a specific programming solution. Even when pages contained relevant information, programmers would often have a hard time finding it on the page, and sometimes falsely conclude that it was not there.

8.1.5. Finding Examples

Once programmers had found a method they thought might be useful (D), they then looked for specific code examples of how to call the method (E). This was usually done by searching Google, or Google Groups, with the name of the specific method, but was also sometimes combined with the previous step (D), or the previous two steps (B, D) when the search results already included code samples.

The Javadoc documentation usually did not provide examples of code use. Examples were used to answer such questions as: "How do I instantiate an instance of this method's class?", "How do I get variables of the appropriate types to pass as arguments?" and "At what point in my code should I call this method?"

The examples the programmers found were occasionally in the form of complete programs with a few lines of interesting code, but more often small code snippets without an accompanying full project.

8.1.6. Observation Conclusions

The popularity and effectiveness of search engines like Google helped convince us that these are important sources of information for programming help tools. The search engines were effective often because of the wide variety of material they indexed, allowing for the successful use of incorrect terminology to find correct answers. Because of this, we chose to build Mica on top of Google rather than have it search its own index as several other tools do[†] [‡].

Where Google failed was in providing appropriate relevance cues for the information that programmers needed. Programmers wanted to find examples or documentation and placed heavy emphasis on the specific API terms that appeared, but Google's presentation of its results did not provide this information.

8.2. MICA DESIGN

We designed Mica to provide the cues that the programmers in our observations needed to more effectively use web search. Mica identifies specific relevant methods and class names by loading and analyzing the result pages of a search, identifying the code-related terms, using frequency-based heuristics to determine which names are likely to be the most relevant to the programmer's specific search, and displaying the results in a sidebar of the webpage that dynamically updates as result pages are loaded and processed on the server. When pages are loaded by the server, they are classified as official documentation or as containing source code, and results of these types are given appropriate icons to guide the programmer. Mica currently finds keywords contained in the Java APIs, but is designed for a range of languages and APIs.

8.2.1. SEARCH RESULTS

Mica uses the Google Web APIs[§] to generate its web search results. Doing so allows the same search options as Google, such as quoted and negated search terms, and the same quality of results. These search results are used both for generating the web search portion of the result page (right part of Figure 8.1) and for obtaining the addresses of the result pages that are loaded and processed by Mica to generate the keyword sidebar.

Mica also uses the spell-check portion of the Google Web APIs to detect likely misspellings and suggest corrections. While a programming specific spelling

[†] JExamples, http://www.jexamples.com

[‡] Hoogle, http://www.haskell.org/hoogle/

[§] The Google Web APIs, http://www.google.com/apis correction mechanism might offer some advantages, Google's system is based on its users' searches and word commonality and usually performs well for programming searches.

8.2.2. KEYWORD SIDEBAR

Keyword selection. The keywords Mica shows (in the left part of Figure 8.1) are selected from all the words related to programming that are contained in any of the ten web search result pages. We chose to examine only the first ten result pages so that we could show programmers where each of the words came from, and because in prototypes of Mica we found that looking at more pages did not significantly improve the quality of the results. Currently, keywords are considered to be related to programming if they match any method, class and interface name from the Java SDK libraries. The list of Java programming names was generated in advance from JavaDoc annotations in the Java source code. However, the ideas in Mica could be expanded to support more languages and APIs, and to implement keyword selection using other clues (such as fixed width fonts and other page formatting) to avoid needing a precomputed complete list of known keywords, and to avoid falsely recognizing keywords that are also common words when they are used outside of a code context.

Keyword ranking. Because the search result pages typically contain hundreds of programming keywords, one key contribution of Mica is how it ranks the keywords to determine which to display, and in what order.

Intuitively, keywords that occur frequently in the search results but infrequently globally (across the whole Internet) are the ones most relevant to a programmer's specific search. We measure search frequency by the number of unique search result pages that contain the keyword, and global frequency by using a precomputed table of how many pages in Google's index of more than 8 billion pages contain each keyword. This table was precomputed for all of the Java API keywords using Google's "Google Suggest" feature. However, a simple ranking metric of search frequency divided by global frequency did not yield good results in our experimentation. Globally uncommon keywords that happened to occur in one result would often rank highest. On the other hand, ranking metrics that more heavily weighted search result frequency tended to rank highest the common methods that were not specific to the search.

To compromise between these, we experimented with a threshold that filters out globally common keywords, and settled on a threshold value of 250,000. Words that occur on more than 250,000 of Google's more than 8 billion pages indexed are considered globally common, and are never suggested as answers in Mica's keyword sidebar. Using this threshold we found that ranking based on result frequency first and global infrequency second (to break ties) worked well. While a threshold has the disadvantage that common keywords are never suggested, we have found that to a surprising extent, the specific questions from programmers relate to tangible input or output and are answered by specific, relatively uncommon keywords. The very common keywords tend to be helper methods that are used across many different tasks, and specific tasks tend not to contain only common helper methods.

Keyword structure. In addition to ranking them based on frequency, the keywords that are in the same class are also grouped and indented beneath the enclosing class if the class is included in the results. For example, `GraphicsDevice` and `GraphicsEnvironment` are enclosing classes in Figure 8.1. To avoid hiding relevance, the higher of the rankings of a method and its enclosing class is used to determine the ranking of the aggregate result that contains the class and method(s).

Highlighting. When moused-over, the keywords in Mica's sidebar show programmers which results contain the keywords by highlighting their background (as shown in the bottom two web search results in Figure 8.1). Similarly, mousing-over search results highlights in the sidebar the keywords contained in that result page. The highlighting of keywords can be used to help more quickly reveal which of the displayed keywords are relevant given a search result that seems relevant. The highlighting of search results can also be used to quickly observe clusters of results that contain related solutions and conversely results that contain different solutions.

In addition to mouse-based highlighting, search results are displayed with a light grey background until they are processed by the server. Doing so gives the programmer feedback on the progress of the server and also helps reveal dead or unresponsive result pages that may not be worth clicking.

Keyword links. When a keyword is clicked, links for that term appear underneath it (the links under `setFullScreenWindow` in Figure 8.1).

The three links currently provided are: a new Mica query on that term, a new Mica query restricted to only Java source code files, and a link to the Javadoc definition of that class. These links are based on three commonly observed uses of keywords: formulating a new query with that term, looking for examples of that term, and looking up the official documentation for that keyword.

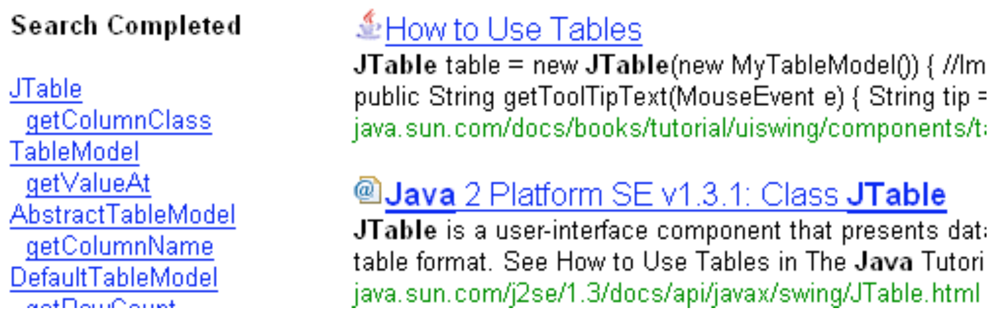
8.2.3. SUMMARY GENERATION

When a keyword is clicked, in addition to the appearance of new links, Mica generates new summaries for the search results that contain that keyword. The new summaries are displayed without affecting the rest of the page.

We designed this feature so that programmers could compare the different uses of a particular method without having to open up the result pages. A design challenge is selecting the context that makes a short summary most useful for a particular keyword. We currently begin a keyword-specific summary at the line with the first occurrence of the keyword on a page, this approach could be augmented by choosing which instance of a keyword to show and choosing how much text before the keyword to include.

8.2.4. ICONS FOR RESULT ATTRIBUTES

Mica displays icons next to search results to represent the type of content some pages contain. For results that contain source code, it displays a Java icon, and for official documentation, it displays a Javadoc icon (see Figure 8.3). The motivation for this feature came from observing programmers' use of Google.



Search Completed

JTable
getColumnClass
TableModel
getValueAt
AbstractTableModel
getColumnName
DefaultTableModel

How to Use Tables
JTable table = new **JTable**(new MyTableModel()) { //Im
public String getToolTipText(MouseEvent e) { String tip =
java.sun.com/docs/books/tutorial/uiswing/components/t

Java 2 Platform SE v1.3.1: Class **JTable**
JTable is a user-interface component that presents dat
table format. See How to Use Tables in The **Java** Tutori
java.sun.com/j2se/1.3/docs/api/javax/swing/JTable.html

Figure 8.3. Result icons for code and documentation.

When programmers wanted to find source code examples in our study (stage E in Figure 8.2), we observed them spending time opening and scanning result pages looking for code. For many searches, only two or three of the first ten results would contain code, and so this process was relatively slow and did not add to programmers' understanding.

To decide if a page contains source code, Mica currently uses a heuristic that if it contains two or more of the eight code keywords shown in the sidebar, it contains code. While Mica could be generalized to use a more sophisticated algorithm, such as the robust code recognition algorithm in [Rha 2005], this heuristic has worked surprisingly well. One reason for this is that because of the keyword ranking, the keywords are known to be globally uncommon, and so this helps avoid the detection of Java keywords that are also English words.

When programmers wanted to refer to the official documentation, we observed that they would often use Google to find the specific Javadoc reference page, even when the root documentation page was already bookmarked or open.

Mica decides if a page should be marked with the icon for official documentation by comparing the URL to Sun's API documentation site.

While Mica currently recognizes and displays an icon only for documentation and source code, it could be extended to recognize tutorials and forum discussions. Forum questions and answers might also be distinguished, but we feel it would be more useful combine these into one "discussion" category because in our observations, the questions themselves were often as useful as answers, and labeling them differently might unnecessarily discourage programmers from exploring them.

8.3. MICA IMPLEMENTATION

Mica is implemented as a collection of Java servlets that use the Google Web-APIs.

While creating a local index of content would have allowed much faster processing, we use the Google search results because of their high degree of relevance. Especially when helping programmers solve vocabulary problems, the effectiveness of any analysis will be limited by the quality of the initial rankings, and so dealing with a non-local index, as we do here, or trying to replicate the quality of Google's rankings is an important and difficult tradeoff for a programming web-search engine.

Because waiting for the result pages to download takes as long as thirty seconds, an architectural challenge was how to display keywords and result-type analysis dynamically, as each individual page is downloaded and processed.

The first implementation strategy that we tried was to use the `XMLHttpRequest` JavaScript object used by many recent dynamic webpages such as Google Maps and many different web-mail sites. In our current implementation however, we instead use a servlet that continuously appends JavaScript code to the HTML page that overwrites earlier results as the page loads. The two main advantages of this approach are that it is compatible with more browsers, including those that do not yet fully support the `XMLHttpRequest` object, and that because the end result is a single regular HTML page, browsers are better able to cache it. This is particularly an issue when using the back button to revisit a Mica query in between result pages.

8.4. USAGE LOGGING

We have made Mica available for public use and have advertised its presence on several Java-related newsgroups. We have used the logs of its usage to help guide future directions of the tool's implementation.

Mica logs the queries as well as out going result clicks and uses of the sidebar. To log outgoing links, which do not usually involve any communication with the server of the source page and so are not trivially loggable, we use JavaScript to trap all click events and record the ones that correspond to outgoing links.

So far Mica has logged some two hundred queries from a hundred unique IP addresses.

8.4.1. QUERY TYPES

One early finding was that while about half of the queries submitted to Mica appeared to involve a general topic or vocabulary search – for example, “load dll jar” or “date arithmetic,” most of the remaining searches were for a specific known Java method or class name, such as “`JSpinner`” or “`URLEncoder`”.

While we had observed programmers using known method names to search for documentation and examples, we had not expected the percentage of such searches to be so high. The usage of Mica in this way helped motivate the documentation recognition and links that provide further exploration from keywords.

Table 8.1 shows a handful of the searches that external programmers have posed to Mica since it has been available.

We would like to know how helpful Mica was for the programmers who issued the queries, and provided feedback links (shown in Figure 8.1), however these were used very rarely, and so this question may only be answerable by a lab study.

Specific	General
toUpperCase	concatenating strings
thread	weak references
WeakHashMap	lazy loading and caching
urlencoder	awt events
DeferredOutputStream	regular expressions
JTable	date arithmetic
JSpinner	load dll jar

Table 8.1. A sample of the queries that programmers have posed to Mica.

8.4.2. Sidebar Usage

Users clicked the sidebar in roughly half of all searches. Since we expected much of the sidebar's usefulness to be been in the learning the terms themselves, which does not necessarily require explicit user action, these numbers are not surprising. Based on its initial usage we expanded the functionality offered by clicking the keywords in the sidebar, making it easier to use them as a basis for new queries and providing a direct link to the documentation for each term.

8.5. Summary

Motivated by our observations of how programmers use web searches to find API information, Mica is a tool that provides better information cues for programmers by extracting relevant information from web results and guiding programmers toward the results that will likely be most helpful for their current task.

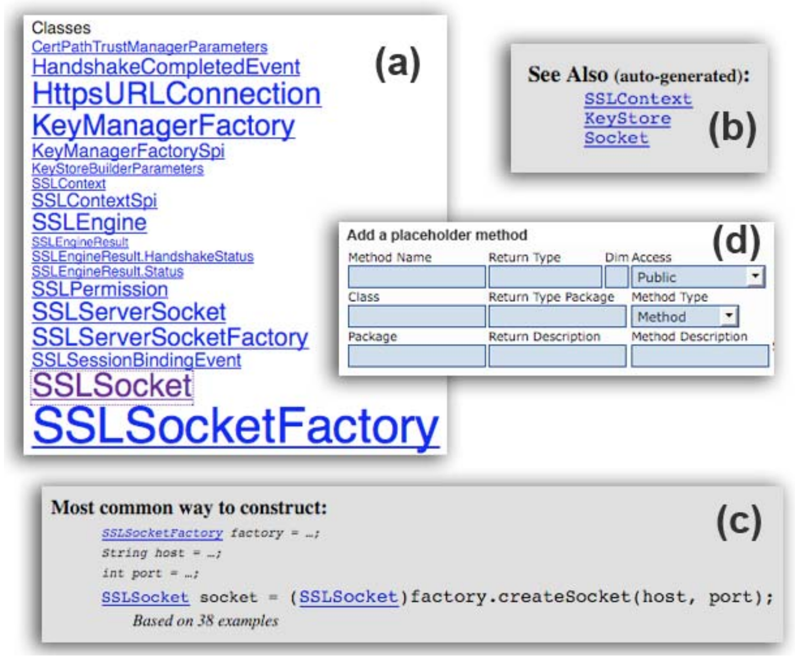
By focusing on programmers' needs and behaviors, Mica shows that tools can offer practical web search improvements for programmers.

The motivating study, design and implementation of Mica led to several contributions:

- Evidence that programmers used Google as one of their primary API exploration resources.
- Evidence that programmers spend time trying to determine which API keywords are common among the top search results, and which result pages contain code examples.
- A new search interface that makes this information of about common API keywords and pages with code examples more directly visible.

9.

PROGRAMMING TOOLS: JADEITE*



Classes
CertPathTrustManagerParameters
HandshakeCompletedEvent
HttpsURLConnection
KeyManagerFactory
KeyManagerFactorySpi
KeyStoreBuilderParameters
SSLContext
SSLContextSpi
SSLEngine
SSLEngineResult
SSLEngineResult.HandshakeStatus
SSLEngineResult.Status
SSLPermission
SSLServerSocket
SSLServerSocketFactory
SSLSessionBindingEvent
SSLSocket
SSLSocketFactory

See Also (auto-generated):
SSLContext
KeyStore
Socket

Add a placeholder method

Method Name	Return Type	Dim	Access
			Public
Class	**Return Type Package**		**Method Type**
			Method
Package	**Return Description**		**Method Description**

Most common way to construct:

```
SSLSocketFactory factory = …;
String host = …;
int port = …;
SSLSocket socket = (SSLSocket)factory.createSocket(host, port);
```

Based on 38 examples

Figure 9.1. Novel features of the Jadeite documentation system. Font sizes are varied based on usage data (a); related classes (b) and common methods of class construction (c) are automatically determined; and users can add new placeholder classes or methods (d) to stand in for expected parts of an API.

A growing body of evidence has made it clear that many APIs are difficult to use [Clarke 2004] (Chapters 1-8). This same research has also shown that not all of this

* Joint work with Zizhuang Yang and Andrew Faulring. Portions of this chapter previously appeared in [Stylos 2009a].

difficulty is intrinsic; APIs can be designed so that they are significantly easier to use. In many cases APIs can achieve a goal of being “self documenting” [Cwalina 2005], where users can learn the APIs simply by trying to use them.

However, this knowledge of how to design more usable APIs does little for the many widely used APIs that have already been released. In addition, there are important considerations other than usability that designers must take into account [Bloch 2001][Cwalina 2005], including performance and future extensibility, which can lead to designing harder-to-use APIs for legitimate reasons.

Different approaches for improving the usability of existing APIs (written in existing programming languages) include: creating wrapper APIs (Chapter 7), changing the integrated development environment (IDE), and changing the API documentation (this and the following chapter). Because previous observations showed that many Java programmers rely heavily [Forward 2002] on Javadoc-based documentation [Kramer 1999], we have been exploring ways that API documentation can be used to improve the usability of existing APIs. This chapter presents Jadeite (Figure 9.1), a prototype documentation system that embodies these ideas. Jadeite stands for: Java API Documentation with Extra Information Tacked-on for Emphasis. Jadeite is a system for displaying API documentation that uses other programmers’ previous API usage to make common tasks easier. Jadeite’s features are motivated by the specific problems observed in previous user studies (Chapters 4-7).

The contributions of this research include new documentation techniques for focusing users’ attention on what is most likely to be relevant and for adding useful extra information in a controlled way, which are shown to be extremely effective (e.g., a factor of 3 faster). We also contribute new techniques for using Google to mine the vast information of the web to augment the information to be displayed to users, and these techniques are likely to be useful for many other areas besides API documentation.

Jadeite is available online at: http://www.cs.cmu.edu/~jadeite.

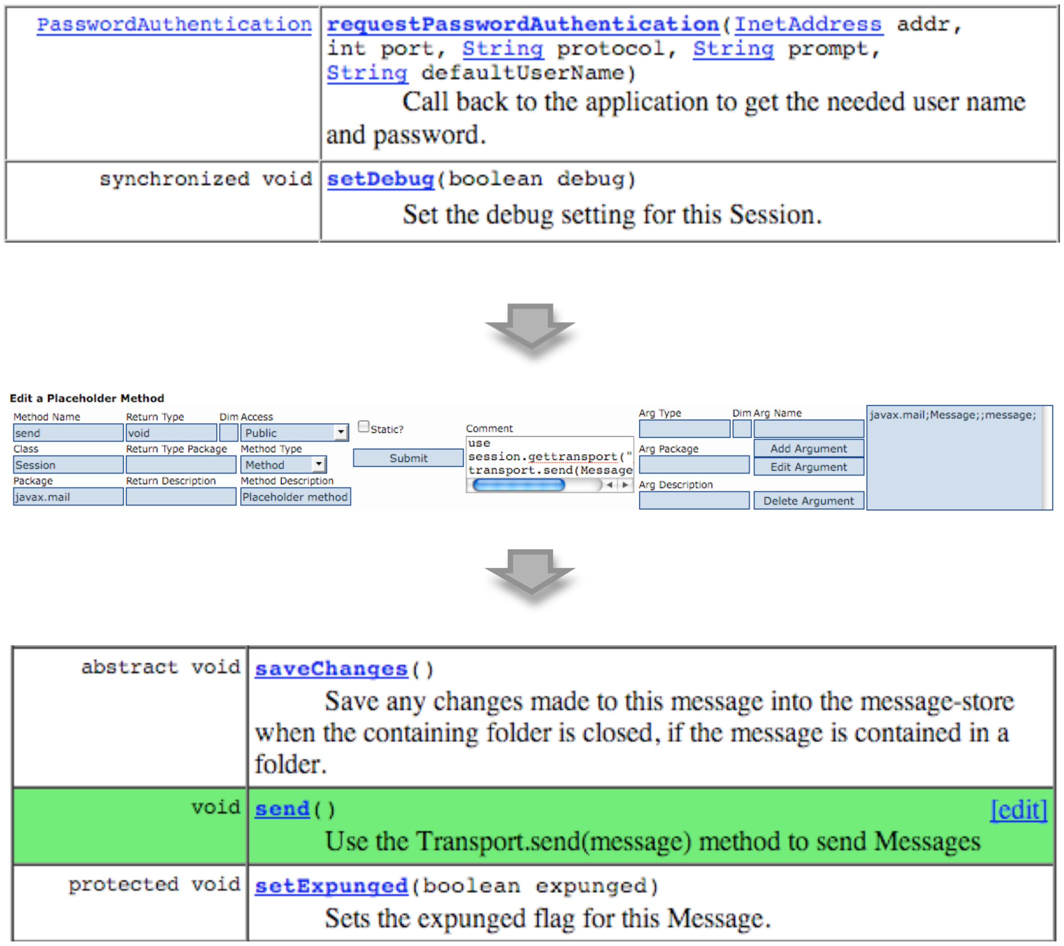
9.1. PLACEHOLDERS

9.1.1. PLACEHOLDER DESIGN

Typical API documentation lists the classes and methods that exist in an API. The idea behind our API "placeholders" is that the documentation should also list the classes and methods that programmers *expect* to exist, and these placeholders should contain forward references to the actual parts of the APIs that should be used instead.

The motivation for this feature comes from observing programmers become frustrated with APIs that did not contain the expected classes and methods. For example, a programmer might reasonably wonder why Java's `Message` and `MimeMessage` classes lack a `send()` method, why classes like `SSLSocket` lack a public constructor, or why the `File` class lacks `read()` and `write()` methods (Chapter 4). Even when there are valid reasons for omitting expected parts of an API, we conjectured that the simplest and most effective way to explain these is by including placeholders in the context of the actual API documentation, where they would appear if they actually existed.

Displaying these placeholders alongside the documentation for the actual API is a key aspect of this idea. Otherwise users would be required to prematurely decide where to look, in the actual API documentation or a separate site, before knowing whether the particular class or method they wanted existed.



PasswordAuthentication	requestPasswordAuthentication(InetAddress addr, int port, String protocol, String prompt, String defaultUserName) Call back to the application to get the needed user name and password.
synchronized void	setDebug(boolean debug) Set the debug setting for this Session.

Edit a Placeholder Method

Method Name	send
Return Type	void
Dim	
Access	Public
Class	Session
Return Type Package	
Method Type	Method
Package	javax.mail
Return Description	
Method Description	Placeholder method
Static?	☐
Comment	use session.gettransport(" transport.send(Message
Arg Type	
Dim	
Arg Name	
Arg Package	
Arg Description	

javax.mail;Message;;message;

abstract void	saveChanges() Save any changes made to this message into the message-store when the containing folder is closed, if the message is contained in a folder.
void	send() [edit] Use the Transport.send(message) method to send Messages
protected void	setExpunged(boolean expunged) Sets the expunged flag for this Message.

Figure 9.2. Before, during and after adding a placeholder send (message) method to the Session class.

One of the primary goals of the placeholder design was to provide a *scalable* way for programmers to edit and add to API documentation. One goal of the design was to work with many different users and edits. Since methods are displayed for browsing concisely by signature, with additional details available when clicked on, it is practical to browse classes with dozens of methods, and adding a few more placeholder methods will not appreciably increase the size of what users must investigate. In contrast, viewing dozens of separate examples or dozens of paragraphs of textual documentation would take much longer.

An API designer might intentionally seed an API with placeholders for the classes and methods they considered including but chose not to. Programmers trying to use the API might later add other placeholders for operations that the original designers never thought of. Other programmers, or the same programmers once they figure out a solution, can then annotate any of the placeholders with replacement code explaining how to accomplish the desired functionality with the available APIs. Programmers might add placeholders for the benefit of others or so that they themselves do not need to re-learn the API when returning to it in the future.

We mark a method as a placeholder by displaying it in the method summary list with a green background, adding “Edit” links in the summary and description, and by displaying “This is a placeholder method” in the description. We wanted to avoid any possible confusion of placeholder methods with actual methods, while still displaying them in the same part of the documentation. Placeholders are currently authored using a form interface (middle of Figure 9.2), but a WYSIWYG editor is possible.

When desired functionality is known not to be possible (or practical) with a given API, a programmer can create a placeholder that is marked as not possible. For example, users of Google’s SOAP APIs might want to perform an image search, but that is not possible with those APIs. One programmer could create a searchImages(query) method, an another could add an annotation explaining that image searching is not supported.

Placeholders also provide a low-maintenance way of providing redundancy in an API. While providing multiple, redundant names or designs would often be unwieldy and inelegant in an actual API, creating placeholders for these reasonable alternatives provides a mechanism for matching the expectations of many different users while keeping the API simpler.

Unlike the other features described below (which take advantage of aggregate information currently available from large corpora), placeholders are based on the idea of community collaboration and evolution, and are an interactive way for users to make the documentation system more useful than when they started. Similar to a wiki, we imagine that sufficient use will evolve the documentation into a more comprehensive and useful state.

9.1.2. Placeholder Implementation

Jadeite is based on the Javadoc documentation system, in part because this is the standard form of documentation for Java APIs that many programmers are used to. The freely available tool to generate Javadocs contains a mechanism for customizability in the form of "doclets[†]", Java classes that enable programmers to generate customized Javadocs. We use a custom doclet to generate a database that is then used by a PHP script to generate documentation that looks similar to Javadoc. Using a web scripting language allows us to more easily create documentation that is dynamic and interactive, instead of being limited to static html. One disadvantage of this approach was that it required reimplementating most of the functionality already offered in Javadoc. To reduce this burden, we took advantage of Javadoc's source file parsing by using a doclet to generate a SQL database that our PHP front-end uses. This approach allows us to generate new documentation for any API for which standard Javadocs can be generated.

Placeholder classes and methods are added to the database by the PHP front-end and stored alongside the actual APIs with an additional placeholder flag. Because they are stored alongside the actual API, Jadeite includes placeholders in the rest of the documentation, for example by including a placeholder class in the list of all known subclasses of its superclass, or all known implementing classes of any interface it implements.

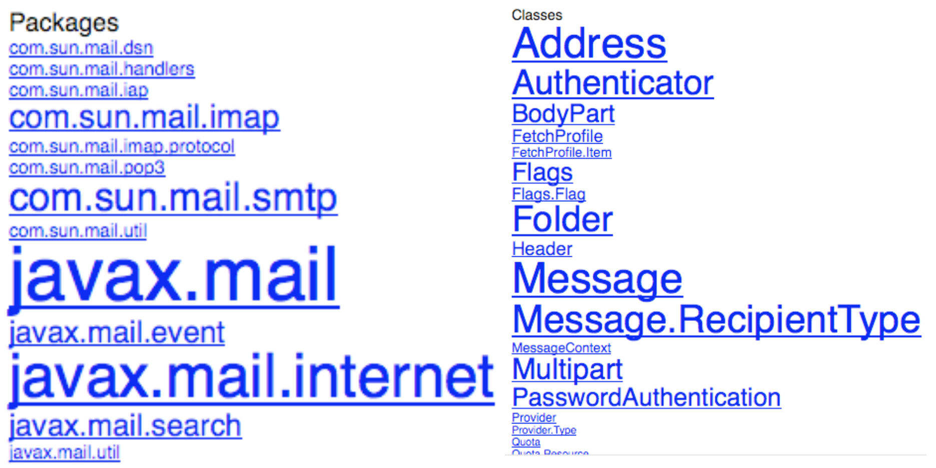
9.2. Font Sizing

9.2.1. Font Sizing Design

In our studies we observed programmers had difficulty finding the classes they wanted (Chapter 6), and in the process they would spend time examining and trying to understand classes that few people ever use (as evidenced by the rarity of example code and references to these classes on the internet). However from the documentation it can be difficult or impossible to tell which classes are the common classes that most people use and which classes are only used rarely.

Our goal was to come up with a design that would highlight the most commonly used classes within the context of the complete documentation, while still showing all of the classes. In our observations of programmers using documentation in which classes were sorted by popularity, instead of alphabetically by name, this greatly annoyed users, who could no longer find a class even if they already knew its name. Because of these observations, we wanted to keep the existing alphabetical list.

[†] Doclets: http://java.sun.com/j2se/1.4.2/docs/tooldocs/javadoc/overview.html



Packages
com.sun.mail.dsn
com.sun.mail.handlers
com.sun.mail.iap
com.sun.mail.imap
com.sun.mail.imap.protocol
com.sun.mail.pop3
com.sun.mail.smtp
com.sun.mail.util
javax.mail
javax.mail.event
javax.mail.internet
javax.mail.search
javax.mail.util

Classes
Address
Authenticator
BodyPart
FetchProfile
FetchProfile.Item
Flags
Flags.Flag
Folder
Header
Message
Message.RecipientType
MessageContext
Multipart
PasswordAuthentication
Provider
Provider.Type
Quota

Figure 9.3. Jadeite displays varied font sizes based on Google hits, shown for the Java Mail packages (left) and the javax.mail classes (right).

The popularity-based font size design we created (see Figure 9.3) is inspired by tag clouds [Halvey 2007], which usually display a similar list vertically across several lines, and are often generated from chat logs.

9.2.2. FONT SIZING IMPLEMENTATION

We compute font sizes based on the number of Google hits for each class and package. We compute this offline, as a batch process, by using the Google API to search for the fully qualified class name e.g., "`java.lang.Object`" and recording the number of hits returned. The frequencies of classes in the Java 6 APIs roughly follow a power law distribution (see Figure 9.4) from the most frequent `java.lang.Object` (with 3,530,000 hits) to the least frequent `java.awt.peer.SystemTrayPeer` (17 hits).

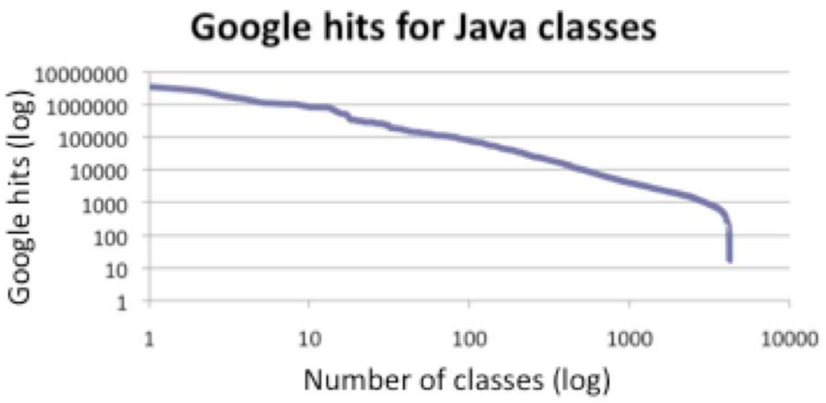


Figure 9.4. The power law distribution of Google hit counts for fully qualified Java class names, on a log-log scale.

Tag clouds generally use either linear or logarithmic weighting schemes. In linear weighting, the most popular element is assigned a predefined maximum font size, and the least popular element is similarly assigned a minimum font size. Linear interpolation is used to calculate the font size of each element, so something halfway between the least and most popular classes will get the halfway between the minimum and maximum font sizes. Logarithmic weighting uses logarithmic interpolation instead.

Because of the observed power law distribution of Java class popularity, using a logarithmic scale for computing font sizes yields a roughly even distribution of font sizes, while using a linear scale results in a few classes with large font sizes and many small classes. We chose to use logarithmic weighting on the list of all classes in the API, so that the list was generally readable, but chose to use linear weighting when listing the classes in a single package. This is because most packages seem to actually have relatively few commonly used classes. Using logarithmic weighting would give above average prominence to almost half of the classes in a package, while we thought much fewer would be usefully highlighted.

We currently compute font sizes for packages, classes and interfaces. When computing font sizes for a list of classes within a single package, we use the relative popularity of a class (or interface) within that particular package (as opposed to throughout the entire API). This makes it difficult to tell from a package list if a class is globally popular (though the font size of its package name gives a hint to this), but has the advantage that there is always a range of font sizes within the class listings of a package, as opposed to a list of classes in uniformly large or small font sizes, as would otherwise happen with popular or unpopular packages.

One of the main advantages of using Google is that the corpus searched is so large (billions of pages, more than 400 million with the word "Java"). It has the disadvantage, however, that it can be ambiguous whether a word refers to a specific Java class or not. We chose to measure popularity by the fully qualified class name (e.g. "`java.io.File`"), because this avoided a problem where class names that were also common English words (for example "`File`" would otherwise get inaccurately high hits, even when including the package name as another search term in the query). Using fully qualified class names also has problems, though; some classes are more commonly referred to fully qualified than others. In particular, Exception classes are frequently referred to fully qualified to avoid an extra import statement. To deal with this, we ignore exceptions when computing font sizes and impose a limit to the maximum size of an Exception (about two-thirds of the maximum font size). A few particular classes are also very frequently referred to fully qualified, such as `java.lang.Object` and `java.lang.String`. These dominate the lists even when using logarithmic weighting. To solve this problem, we ignore the top 0.05% most common classes when computing other classes' font sizes. These very common classes are still displayed at the maximum font size. (Selectively ignoring values means that these classes might otherwise be assigned sizes *greater* than the normal maximum font size, however we limit these to the normal maximum font size.)

Jadeite computes the popularity of individual class methods using the same technique as for classes and packages, however we do not currently display this information. One reason for this is that methods are not currently displayed in the same simple list that packages and classes are, making it less obvious which font sizes to change. Design ideas for this could be explored in the future.

9.3. CONSTRUCTION EXAMPLES

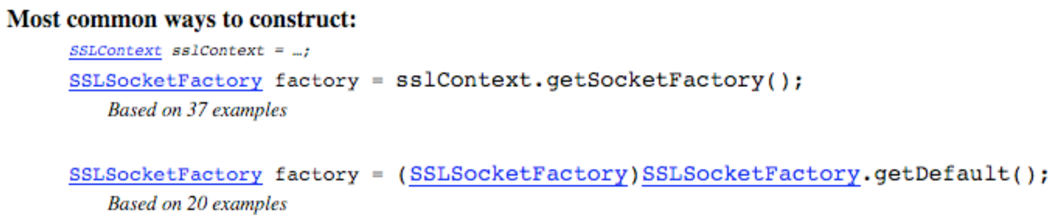
9.3.1. CONSTRUCTION EXAMPLES DESIGN

From the pseudocode that participants wrote in our previous studies and from their think-aloud comments (Chapter 4) showed that nearly all of the our users expected all objects to be constructed using a constructor (and usually by a default –

parameter-less – constructor). When presented with classes that needed to be constructed without a constructor, the first – and sometimes insurmountable – barrier was in realizing that something other than such a constructor was needed.

Providing this initial realization was one of the main goals of our design of the construction-examples feature. For this reason we chose to place the construction-example snippet near the very top of the class documentation page, just below the inheritance hierarchy (see Figure 9.5). In addition to trying to solve the usability problem of the Factory pattern (Chapter 5), we were also motivated by difficulties we observed programmers had with abstract classes and interfaces. In our studies, where programmers would often not realize a class was abstract (or that it was actually an interface) until after they had written code that tried to construct it.

Another goal was to provide short, understandable snippets that users could copy and paste into their programs. In initial prototype displayed only a single line of example code. However, in order to annotate the types of the variables and keep it on a single line we had to use non-standard Java syntax. We quickly realized, however, that a more readable snippet was required for users, and so we display the snippet on multiple lines, using an additional line for each of the instance variables that are used as a factory or parameter (see Figure 9.5). This lets us use standard Java syntax for defining class instances.


Most common ways to construct:

```
SSLContext sslContext = …;
SSLSocketFactory factory = sslContext.getSocketFactory();
```
Based on 37 examples

```
SSLSocketFactory factory = (SSLSocketFactory)SSLSocketFactory.getDefault();
```
Based on 20 examples

Figure 9.5. Based on example code found on Google result pages, Jadeite shows the most common ways to construct an instance of the SSLSocketFactory class.

One aspect of the design we considered was how large of a construction example snippet to display. While a class instance is usually instantiated in only a single line, this line sometimes uses parameters or factories that themselves have complicated construction patterns. Some classes also have post-construction initialization methods that need to be called before using the object. We chose to display only a single line with the addition of partial lines for each of the instance variables used in the construction example, but chose not to recursively try to include code to instantiate each of these variables, since sometimes this chain would be very large. (An exception is values that are used inside the main construction example without being assigned to a temporary value, for example a constant like "localhost" or 8080.) We display an ellipsis after the variable declaration, to represent that some instantiation of these variables is needed but not shown. Users can see how to instantiate each of these variables, if they need to, by clicking the class name link and seeing the most common construction patterns for that particular class. One disadvantage of this approach is that it loses the specific context of how the classes are used together. For example, suppose a factory is used to create a product class. Showing how to create the factory on its own page means that users will see the most common overall way to construct the factory, which might not be the same as the way the factory is usually constructed when using that particular product. So far, this does not seem to be a practical limitation for the Java APIs we have looked at, however.

When classes can be created in multiple ways, one question we had is how many different construction examples to show. In our initial development, almost all of the classes we examined were nearly always constructed in one particular way, so we chose to usually display only the most common way of construction, and display the two most common ways in the few cases that there were more than one common way. We currently display two different construction examples if the second most popular construction has more than 50% of the number of different source examples as the most popular construction pattern (except if there are 20 or fewer total examples for that construction pattern).

9.3.2. Construction Examples Implementation

The examples are constructed by examining the sample code contained on the top 500 Google results for a search using the fully qualified name of the class. Within these pages we look for code construction examples that match a regular expression for variable declarations and assignments (*ClassName variableName* = *expression*;). For each of the variables used in each construction examples, we try to figure out the type of the variable by looking for variable or parameter declarations. For example, an earlier line that contained: "`SSLSocketFactory factory = `" would tell us that the factory in the construction example was of type SSLSocketFactory. For each variable type and explicit class reference, we then try to determine which package it was from. In the event that there are multiple classes with the same name in different packages (for example `java.util.List` and `java.awt.List`), we guess the package that is closest to the package of the class for which we are looking for construction examples (choosing `java.awt.List` in construction examples of classes in `java.awt` or subpackages).

After recording all of these construction examples, we aggregate all of the examples that have the same type signature, ignoring whitespace and variable names. For each variable we determine the most common variable name and use this and all of the variable types we were able to determine to create a construction example signature. For example:

```
( javax.net.ssl.SSLSocket ) {javax.net.ssl.SSLSocketFactory:factory} .
createSocket ( {String:host} , {int:port}
```

We chose to use Google as the corpus again because the other large corpora we examined seemed to be less comprehensive and more biased by the inclusion of a few large projects (such as Apache), whose use of code did not seem representative of average use. Code-focused search engines also tended to return more results from the implementation of the desired API (for example from the source code that implements the Java Swing APIs), rather than example code that used the API.

Using Google has the disadvantage that many of the examples are incomplete. This requires Jadeite to perform more work on the back end to identify valid source code and to recover type information. However, we think that the improvement in results is worth this extra effort.

9.4. RELATED CLASSES

9.4.1. RELATED CLASSES DESIGN

Our object design study (Chapter 6) showed that people had difficulty finding needed methods on other classes, and difficulty even realizing that other classes were needed.

Javadoc already contains a “see also” section with a manually created list of related classes (and occasionally related methods), but in our studies, we found these annotations to be inadequately sparse. One of our goals was to automatically generate richer data about related classes.

One design question was how many related classes to show. One approach would be to use a relevance threshold, so that classes with many strongly related classes would show more related classes than a class not strongly related other class. Even doing this, however, would require calibration so that a certain number of classes were displayed on average. Our current design displays a fixed number of related classes for each class, in part to explore the question of how many different classes are useful to display. Our initial choice of five classes seemed too long, making the classes in the middle of the list less noticeable, so Jadeite currently displays only three.

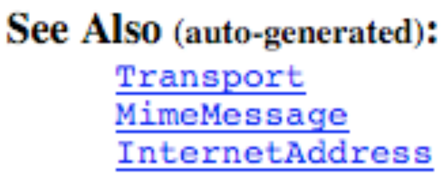

See Also (auto-generated):
Transport
MimeMessage
InternetAddress

Figure 9.6. Jadeite displays related classes automatically based on co-occurrence on Google result pages. Shown are the related classes for the Message class in the Java Mail API.

While currently Javadoc displays related classes in a comma-separated list on the same line, we chose to display each class on a separate line. We did this to make the second and third classes more visible and to avoid giving as much emphasis to initial class as in the single line approach.

One fundamental question we had in computing which related classes to display, was whether to display the classes that were most commonly used in conjunction with the current class or instead the classes that were most closely correlated with the current class. The second approach takes into account how (un)common each class is overall. For example, one might worry that in the first, naïve, approach, the `System` class would be most related to everything, since so much example code uses `System.out.println()`, while the second approach might return obscure classes that are occasionally used with the target class and nowhere else. Mica (Chapter 8) takes the second approach, attempting a careful balance between displaying too many classes that are globally popular versus displaying classes that are not at all popular but closely correlated since they are rarely used anywhere else. Our current design uses the first, simple, approach, with the exception that it explicitly filters out `java.lang.System`. So far we have been surprised by the effectiveness of this approach, which tends to find related classes in the same package, instead of finding globally popular classes in other packages.

9.4.2. RELATED CLASSES IMPLEMENTATION

Jadeite finds related classes by examining the Google result pages using a query of the fully qualified class name. From these results we count the number of references to other classes, and find those that co-occur the most. This is similar to a technique used by Mica (Chapter 8). However, unlike Mica, which examined the results in real-time based on a user-supplied search query, we compute related classes offline and so are able to examine more result pages – the first 500, compared with Mica's 10 result pages.

9.5. STUDY

We performed a user study to test if and how much Jadeite's web interface helped programmers compared with standard Javadoc.

9.5.1. METHOD

We repeated two tasks from prior work since they proved to be examples of difficult tasks. These first two tasks, creating an `SSLSocket` (which required a factory) and sending an email (which required the use of multiple abstract classes and a helper `Transport` object) have been previously described (Chapters 5, 6). We added a third task to test how programmers would be affected by our tool when they were performing a comparatively uncommon task, to test if Jadeite's features get in the way of finding the necessary information. In the third task, we asked participants to take an input like www.google.com and return an output like "66.2.10.162", using the package `java.net`. We used this wording to avoid mentioning terms like IP address, URL, or DNS lookup, which might have biased their exploration. We chose this task because none of Jadeite's features were helpful: the font sizes of classes in the package were dominated by the `URL` class, not the `InetAddress` class that needed to be used; the construction example for the InetAddress used the local host, and not an arbitrary host name in the form of a string; and `InetAddress` was not suggested as a related class to `URL`, the most common class in the package. We wanted to make sure that, in the (hopefully uncommon) case when users had to do something different than Jadeite suggested, Jadeite would at least not cause new problems.

We used identical recruiting and study setup from the previous studies (Chapters 5, 6), so that the earlier data can serve as the control condition. We ended up with 7 participants, all current students, with between 1 and 4 years of Java experience (an average of 2 years). All were very familiar with Javadoc. Participants in the Jadeite condition were told that they would be using new documentation, and were given a brief, one-minute overview of the new features.

We focused our study primarily on the effect of the three automated analyses, without placeholders, and then on the user interface for adding placeholders. The first five participants performed each task using Jadeite without any placeholders (and without the user interface for adding new placeholders), and then, after they had finished all of the tasks, they were told about placeholders and asked to add any they felt would have been helpful. The last two participants performed the tasks with placeholders turned on (though still no UI for adding new ones), and saw all of the placeholders that the previous 5 participants had added.

We also asked participants to fill out a survey at the end of the study, in which we had them rank how helpful they thought each feature and the documentation overall on a 7-point Likert scale, and asked them whether they preferred standard Javadoc or the new documentation.

9.5.2. Results

To test the effect of Jadeite, we measured the time taken to perform several specific parts of the tasks. Measuring these parts helps reduce the effect of overall task variance due to each participant's programming style and also helps separate the effects of different features on participants' success.

On the Email task we measured the time to find the `Message` class. Every participant found the documentation for this class before writing successful code. We also measured the time it took participants to find the `MimeMessage` class, which was the needed subclass of the abstract `Message` class.

The times compared here are the first five Jadeite participants compared with the control condition participants, run in the previous studies.

Participants using Jadeite were approximately 3 times faster at finding `Message`, in an average 4 minutes versus 12 minutes in the control condition. For this and the other times we used the Wilcoxon Rank-Sum test (which does not assume normality) to test statistical significance, and found p to be < 0.05.

Participants were also about 3 times faster at finding `MimeMessage`, 5 minutes for Jadeite participants versus 18 minutes in the control ($p < 0.05$).

In the `SSLSockets` task we measured the time it took participants to find the `SSLSocketFactory` class, which was needed to construct the `SSLSocket`. Participants were about 2.5 times faster at finding `SSLSocketFactory` in the Jadeite condition, spending an average of 7 minutes versus 17 for the control condition ($p < 0.05$).

After testing one condition of the `InetAddress` task (there being no previous study to compare it to, unlike the other tasks), we felt from our subjective observations that participants were not slowed down by any of the Jadeite features, even when they did not suggest the right answer for the task. Because running a second condition for just this task would have required twice as many participants, with the expected result being no statistical differences, we did not run a second, control condition, and make no claims about the relative effectiveness of participants for this task.

On the survey, participants ranked Jadeite at 6.3 out of 7, where 7 was very helpful and 1 was very unhelpful (see Figure 9.7). Placeholders were ranked 6.3, font sizing 5.9, construction examples 6.7, and related classes were ranked 5.3. All seven participants preferred Jadeite to the standard Javadocs. After the study, two of the participants asked (without prompting) to be emailed if we released Jadeite to the public (which we now have done).

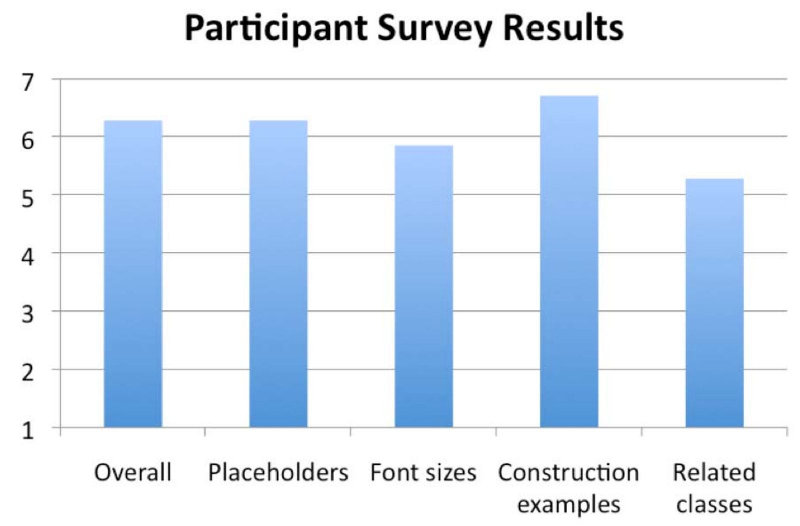


Figure 9.7. How helpful participants thought Jadeite and its features were, with 7 being very helpful.

9.5.3. Discussion

Based on our observations of which features participants used, the faster times finding the `Message` class can mainly be attributed to the font sizes, and the faster times on the `MimeMessage` and `SSLSocketFactory` can mainly be attributed to the construction examples.

Subjectively, placeholders seemed to help the final two users a great deal, though this might have been in part because all of the available placeholders were relevant. The long-term usefulness of placeholders will have to be tested as more, varied placeholders are added over time.

The reaction to the font sizes by the participants seemed initially neutral, but grew more positive with use. In contrast, participants were immediately happy with the construction examples. No one seemed to notice the related classes. One reason for not noticing the related classes could be its position below the paragraphs of textual documentation. We chose this location because it is where Javadoc normally displays related classes, but it seems to be a visual dead zone.

Based on watching participants add new placeholders, we confirmed that our existing form-based interface (see Figure 9.2b) was too complicated, and participants had difficulty determining the purpose of each textbox. From these results we have increased the priority of creating in-place web interface for adding placeholders on our list of planned improvements.

While our observations are consistent with the idea of programmers preferring to use example code, we had not previously realized just how powerful automatically selected example code could be, and how practical it was for inclusion on the documentation for each class or even each method. Consequently, we believe that finding new ways to add more example code is the most promising future direction, both in terms of programmer preference and effectiveness.

9.5.4. LIMITATIONS

Our study tested only a limited number of tasks, and focused on tasks that we knew to be problematic, to test if we had helped solve some of these problems. Based on our own usage of the tool, we think that it will be useful for many more APIs and tasks as well.

The participants in our study were unfamiliar with the APIs they used in the tasks. We expect that the magnitude of the benefit would be diminished when participants are already familiar with the APIs, since they will already have an idea about which classes they might need and how to use them. We expect that the features in Jadeite will still be useful to jog the memory of programmers who return to unfamiliar parts of the APIs. Also, since APIs are so large, we think using new parts of APIs is an inevitable and important task.

Participants in our study may have been biased to use our features by their visual novelty, or by the fact that we briefly pointed out the new features as part of the tutorial the subjects ran at the beginning of the study. However, given the disuse of the related classes, it is also possible that they were biased against using the new features by their familiarity with standard Javadoc. We considered not including the 1-minute overview where we pointed out the new features, but felt that this would hinge the results on their visual prominence, and we wanted a realistic design that would be practical and usable as a long-term solution, not something that was artificially eye-catching to ensure that programmers noticed it on their first use.

While our techniques work well on the APIs we have tried so far, we expect that other APIs with fewer users or comprehensive code repositories might benefit more from other implementation strategies to find construction examples, related classes and compute popularity.

Using our techniques, APIs that change while keeping the same package and class names will retrieve out of date examples. However, these types of major breaking changes to existing APIs have been rare, with functionality usually being offered in the form of new classes or methods.

Jadeite is based off of Javadoc because that is the standard documentation style for Java, but the features would generalize to other object-oriented languages like C#, though they might be designed slightly differently to better match the existing style of the documentation. Generalizing to non-object-oriented languages would require greater changes – the construction examples might not make sense, but the idea of placeholders and popularity-based sizing and highlighting would generalize.

9.6. SUMMARY

The approach taken in this work, of studying the user's real problems, creating tools to solve those problems, and performing user studies to evaluate the results, proved very successful, and resulted in new designs that may benefit many different kinds of documentation. We showed that information about programmers' API usage, whether it is mined from Google or code repositories, or explicitly annotated by programmers, can improve existing API documentation. Jadeite demonstrated how this data can be used to make it easier to find starting classes, figure out how to construct objects, and find the right helper objects. We hope that lowering these barriers will help make programming easier and more accessible to more people.

The contributions from the design, implementation and evaluation of Jadeite include:

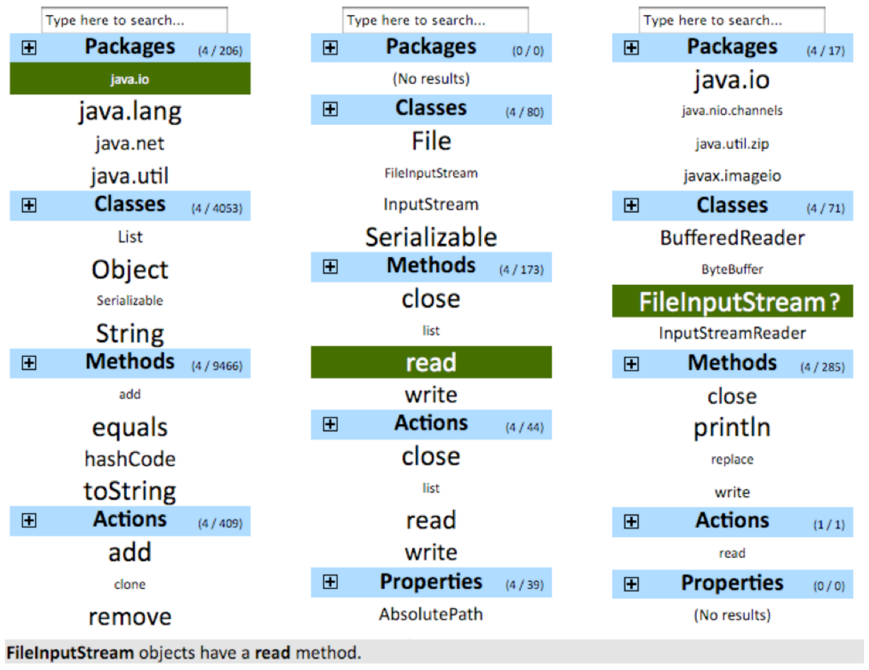
- Techniques for exploiting usage data on Google to improve usability.
- The concept of user-created "placeholders" that are displayed alongside actual API documentation.
- A user-interface that uses usage data to more prominently show popular classes.

- Techniques for extracting code construction examples from a large and potentially imprecise repository of webpages.
- A user-interface that displays common construction patterns alongside the rest of the class documentation.
- Techniques for automatically determining related classes using a web repository.
- Evidence that these features can make common API tasks easier.

Overall, Jadeite embodied much of the knowledge that we gained through our prior user studies, along with several new ideas of how API documentation might be better.

10.

PROGRAMMING TOOLS: APATITE*



FileInputStream objects have a **read** method.

Figure 10.1. Using Apatite, programmers can browse APIs by packages, classes, methods, actions or properties. Here a programmer discovers that FileInputStream is a class for reading files.

In our previous observations of programmers (Chapter 8) we saw that using Google to learn about APIs can be a popular and powerful, though imperfect, strategy. The

* Portions of this chapter were submitted for publication in [Stylos 2009b].

web contains many relevant programming discussions and examples, but it also contains many irrelevant and unhelpful pages.

Another popular strategy for learning Java APIs is to explore the API's Javadoc documentation. The Javadocs display a hierarchy of all of the packages, classes and methods of an API, but usually provide little cues besides the alphabetically listed package and class names to help users decide which classes to investigate. In user studies of different APIs and API design decisions, we observed that these lists were often insufficient for helping guide programmers to the relevant classes for their task, and that they would spend considerable time examining obscure, rarely-used classes. Our observations of programmers using existing API documentation [Beaton 2008][Jeong 2009] suggested that the structure of these hierarchies were insufficient when programmers knew a desired action but not which class it might belong to.

For example, programmers attempting to send an email often started by looking for a `send()` method, and would only later find a `MailMessage` class that could be sent, and later still find the `Transport` class used to perform the sending. By organizing the API in terms of a single Package → Class → Method hierarchy, current Java documentation does not support explorations where users start with a desired action.

An inspiration for Apatite was the associative browsing tool Feldspar (Chau 2008). Feldspar provided an interface to browse personal information like email, contacts, and events based on association rather than search. For example, using Feldspar you could find people mentioned in an email about an event last year, without needing to remember any of the specifics or enter any text queries. The multi-column browsing interface we used in Apatite is inspired by Feldspar. However, to work effectively with API data, Apatite differs from Feldspar in several key ways. In Feldspar, the associations are used to progressively narrow the set of possible results until a particular person, email or event has been located. In Apatite, each selection can potentially increase the number of relevant items, letting users browse a path through the API rather than constructing a set of constraints.

This chapter presents Apatite (Figure 10.1), a new exploration tool for APIs that lets programmers browse APIs starting from actions, properties, methods, classes or packages, and finding other items by association. Apatite uses the popularity of each individual API element as well as statistics about how often different API elements are used together to display the most relevant results.

Apatite stands for: **A**ssociative **P**erusing of **A**PIs **T**hat **I**dentifies **T**argets **E**asily.

Apatite is available online at: http://www.cs.cmu.edu/~apatite.

10.1. User Interface

10.1.1. Example use

Suppose that a Java programmer wishes to read some data from the disk. Using the standard Javadoc, or even Jadeite, the programmer would have to figure out which class to investigate. Because there are more than three thousand classes in the base APIs, it is likely that the programmer would want to first select one of the 166 packages to narrow down the investigation. Fortunately, `java.io`, the package relevant to reading and writing data, is near the top of this alphabetical list. The `java.io` package contains 78 classes and interfaces. The most relevant-seeming name is the `File` class. The `File` class documentation contains 47 methods, constructors and fields (not counting those inherited from `java.lang.Object`). After inspecting these, a programmer would learn that *none* of these methods support file reading, so they must return to guess which of the other 77 classes or interfaces in the package might do what they want.

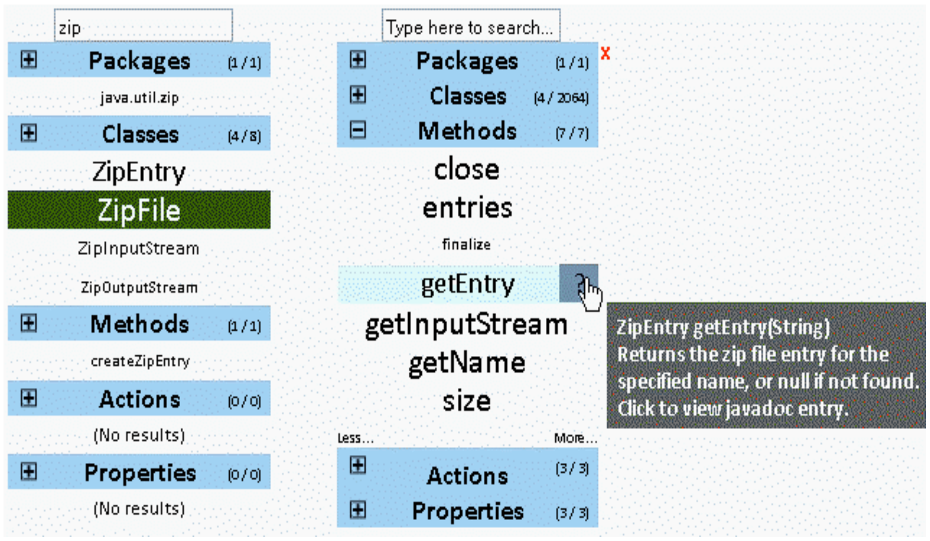
Now let us see how a programmer would use Apatite to answer this question. The initial Apatite screen shows the top four entries for each of packages, classes, methods, actions and properties (see Figure 10.1). Because it is popular, `java.io` is displayed in the packages section of this initial list. When a programmer clicks on `java.io`, the most relevant classes, methods, actions, properties and other packages are shown in the next column. `Read` appears in this list of most popular methods related to `java.io`. When clicked on, `read` shows a third column of relevant items. In the class section of this list are `BufferedReader`, `ByteBuffer`, `FileInputStream` and `InputStreamReader`. When hovered over, a question mark displaying information about the item and linking to the documentation is shown for each of these classes, as shown in Figure 10.2. Which class is the right answer? It turns out there are multiple ways of reading files, most of which involve multiple classes, but `BufferedReader` and `FileInputStream` are two popular classes for reading files. Clicking on the question mark next to each class links to the Javadoc documentation for that class. A list of class or method names is not always enough for programmers to know whether they have what they want or not, but even when it is not, Apatite provides a quick mechanism for generating relevant information about which classes or methods they might want to investigate further.

10.1.2. FEATURES

Apatite's interface includes several features in addition to those supporting the core interactions as described above.

A search box at the top of each column lets users quickly filter the names to only show those that include the entered text. This allows a form of search while still staying within the same view of the documentation. In the example from the previous section, a user might choose to enter "read" into the search field of the first column rather than selecting a package. In addition to being filtered, the sizes of the displayed elements are recalculated based on the comparative weights of the remaining items (avoiding the case when all of the font sizes for all the remaining items are too small to see any relative differences).

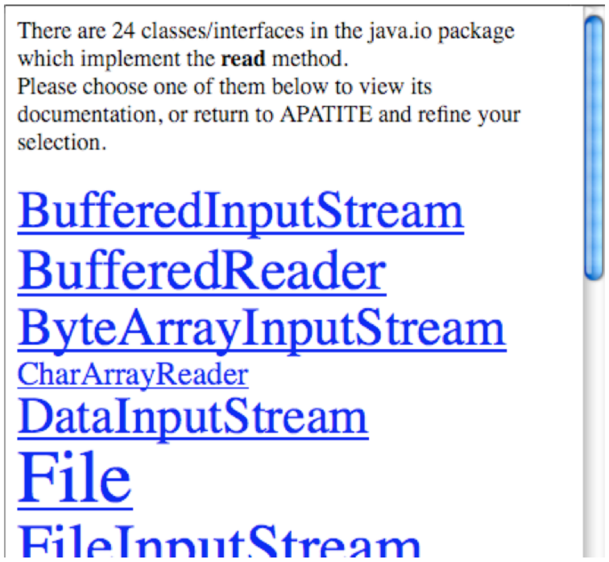
To support browsing beyond the most common four items Apatite supports expanding a section by clicking on the [+] in the section title. This causes the other sections to collapse and displays the top 15 items in the expanded section, with recalculated font sizes. Once a section is expanded, "more…" and "less…" links appear at the bottom of the section, letting users double or half the number of items shown in the list as many times as they want (see Figure 10.2, second column). Because Apatite supports browsing from members of any API section to any other section, we decided that it is important that initially each section is partially expanded to show the top few items.



ZipEntry getEntry(String)
Returns the zip file entry for the specified name, or null if not found.
Click to view javadoc entry.

Figure 10.2. Apatite supports keyword searching, expanding each section, and shows hover-text details.

All of the lists are always kept alphabetically sorted because we observed in earlier studies that when lists of API items were not sorted, users would become frustrated and not be able to quickly find items on the list again after glancing elsewhere.



There are 24 classes/interfaces in the java.io package which implement the **read** method.
Please choose one of them below to view its documentation, or return to APATITE and refine your selection.

BufferedInputStream
BufferedReader
ByteArrayInputStream
CharArrayReader
DataInputStream
File
FileInputStream

Figure 10.3. When a user clicks to see documentation for an item with multiple instantiations, Apatite shows a page linking to the different classes.

To learn more about a particular item in the API, in addition to its associations, Apatite displays mouse-over text, a status bar and links to the documentation. When a user hovers over a particular item, a [?] link appears to the right of the item.

Hovering over the [?] displays information such as the return type and method parameters (see Figure 10.2). If an item represents multiple classes or methods, such as a method that is included in multiple classes that the user has not yet chosen between, then the hover text displays the number of different items with that name. Clicking on the [?] link goes directly to the documentation if the item in unique, or to an intermediate page (see Figure 10.3) in a new tab showing links to all of the different items when there is more than one.

The status bar, shown in the bottom of Figure 10.1, describes the association between the item the mouse is over and the previous item, for example whether a class contains a method or is the type that is returned by the method. This information is also shown in mouse-over text.

Apatite provides the options of *actions* and *properties* in addition to packages, classes and methods. We chose these abstractions because in our motivating observations [Beaton 2008][Jeong 2009], we watched programmers often start with the verbs, or action words, and often did not yet know the right nouns or objects. In Java, these verbs are usually implemented in the form of methods. However, many methods are simply getter and setter methods for what would be properties in other object-oriented languages like C#. We use properties to separate out these getter and setter methods from the methods that represent actual actions.

10.2. IMPLEMENTATION

Apatite is implemented as a web-accessible application. All of the necessary data is stored on the server in a MySQL database and served on-demand via AJAX-style requests. Our primary motivation in using web-based technology was to create a tool that the general public could easily use without needing to install any additional software.

10.2.1. POPULATING THE INITIAL COLUMN

The first column that Apatite displays is based on how frequently each API item is used by programmers. Popularity information for each item comes from data collected for Jadeite. In the Jadeite data set, the popularity for each package, class and method is based on the number of hits returned by a Google search for each item. For example, the weight for the `InputStream` class is computed based on the number of results returned for the query: `"java.io" +InputStream`. These results were computed ahead of time as a batch process and are cached in Apatite's database.

On Apatite's initial page, the four most popular items in each category are retrieved and displayed alphabetically. Popularity data for API items tends to follow a power law, so within each category a logarithmic function is applied to each item's Google hits to derive a corresponding font size:

$$Size = 100 \times \frac{\log w - \log w_{\min}}{\log w_{\max} - \log w_{\min}}$$

where w is the number of hits for each item, $w_{\min}$ is the number of hits for the most popular item in the result set, and $w_{\max}$ is the number of hits for the least popular item the result set. When a category is expanded to show more items, they are once again sorted alphabetically, and their font sizes are re-computed using the new $w_{\max}$ and $w_{\min}$ values.

10.2.2. Populating Additional Columns

When the user clicks on an item, a new column appears consisting of items that are associated with the user's selection. These associations have been pre-computed using various methods (described below). Each item in this associated column also has a numeric score that represents how strong the association is; this is an analogue to the number of Google hits used in the first column. With this data, Apatite displays the new column using the same process that it uses on the initial column: the strongest associations in each category are retrieved, sorted alphabetically, and then sized using the logarithm of their score, as described above.

The process of computing these associations and their weights depends on the nature of the association. The heuristics described below were developed by exploring various possible techniques experimentally and choosing ones that seemed to yield the most logical associations, and by using the user feedback in our user tests to help tune the association heuristics.

For associating a **method with other methods**, we collected a new set of popularity data that indicates how often methods are used in conjunction with one another. To determine the metric for associating Method A with Method B, we counted the number of times the text `.methodB(` appears in the first 100 Yahoo search results for `methodA`. (We used Yahoo for our search results here, rather than Google's as in Mica and most of Jadeite, because Yahoo's daily limit of searches was higher than Google's.) The result is an approximation of how often the methods are used together in actual code. We analyzed the search results rather than just measuring the results for a new search for both `.methodA(` and `.methodB(` because Yahoo's search does not allow limiting results to those that include specific punctuation like periods and parentheses.

For associating a **method with classes**, we included three different types of relationships: classes that *implement* the selected method, classes that are *returned by* the selected method, and classes that are *arguments* to the selected method. For each of these, the association weight is the sum of the number of Google hits over all implementations that meet the association's criteria. For example, the weight for associating the read method with the `ByteBuffer` class is the total number of Google hits for all implementations of read which take a `ByteBuffer` object as an argument. If a certain class is associated with a method in more than one way, we use the maximum weight of these associations.

For associating a **class with other classes**, we take the sum of the association weights between Class A's methods and Class B's methods, multiplied by the logarithm of Class B's overall popularity. Using the logarithm of the popularity in this case avoids overemphasizing extremely popular classes like Object.

The remaining associations (**package to class**, **class to package**, **class to method**, etc.) can all be characterized as having an encapsulation relationship (e.g. `BufferedReader` is contained in the `java.io` package). For these, we assign weights by summing the Google popularity data over all implementations that satisfy the encapsulation condition. For example, to generate associations between the read method and various packages, we first group all of the implementations of read by their containing package, and then sum the popularity of each package's separate implementations.

To perform **multi-column associations** (when there are more than two columns present), we combine the weights of the associations from each of the previous column pairs, progressively discounting the weights of earlier columns so that the results are most heavily influenced by the most recently selected item.

10.2.3. DETERMINING ACTIONS AND PROPERTIES

As a first approach to determining the actions and properties, we found methods that represent actions and properties by looking at the method names. To recognize methods that represent actions, we used a dictionary to look up the part of speech of the first "word" in each camel-case identifier and identified methods that begin with verbs. For instance, `createGraphics`, `createImage`, and `createStatement` all are associated with the action `create`. Similarly, words and phrases that are preceded by the standard prefixes "get," "set," "is," and "has" are considered to be properties. E.g., `getName`, `hasName`, `isName`, and `setName` are all examples of methods that are associated with the `name` property.

This technique works surprisingly well. One motivation for only looking at identifiers to determine actions and properties was to keep the interface consistent with the vocabulary of the language; by browsing the popular actions of various packages and classes, users can familiarize themselves with common word choices. However, it does fail in some cases where English words can serve as both verbs or adjectives. "List", for example, appears as a verb in the `listFiles` method and as an adjective in the `listIterator` method. For now, these cases are manually removed from the database.

10.3. EVALUATION

Throughout Apatite's development, we performed a number of user studies to gauge the tool's potential and discover which aspects were in need of improvement.

10.3.1. EARLY PILOT STUDIES

We ran pilot studies often during the design process to discover how users might use Apatite and what features might help them find results faster. We made several modifications based on this feedback, including adding the ability to perform a text search on each set of results and inserting [+] and [-] symbols to category titles to make their expandability and collapsibility easier to discover.

Overall, these early trials garnered a positive reaction from users. After the design had been refined numerous times, we carried out a more formal user study that aimed to compare Apatite's performance to that of existing methods for searching through API documentation.

10.3.2. METHOD

The primary study investigated how well Apatite helps typical programmers accomplish difficult tasks using unfamiliar API's. We used a within and between-subjects approach, with two main conditions: for some tasks, participants were restricted to using Apatite, while for other tasks, participants were allowed to use any alternative internet-based method, such as the standard Java API Javadoc and/or Google. We did not place any restrictions on the control group in order to measure how programmers actually explore unfamiliar APIs.

The study was administered to ten participants, who each had between one-half and five years of Java programming experience. There were four tasks total, each with six or seven sub-tasks. Each participant performed two tasks with Apatite and two tasks with standard Google and Javadoc. The order of the tasks and conditions were counter balanced to account for any learning effects.

After reading each sub-task, a timer would begin, and the participant would begin searching for the answer. Upon locating an answer, he or she would check with the study administrator; if the proposed solution was incorrect, the timer would continue until a correct answer was found. To discourage guessing, participants were instructed to be confident their answers were correct before checking with the administrator. Participants were allowed up to 11 minutes for each sub-task, or 15 minutes for each question set, whichever expired first. Exceeding the time limit on a given question set was marked as a failure.

The question sets focused on tasks that, although they are common tasks based on our popularity data, they seemed to have particularly confusing or complicated implementations in the Java API. The four sets were comprised of "Network Programming", "Querying a Database", "Popup Windows", and "File Input/Output."

The sub-tasks within each set were designed to assess a variety of levels of comprehension. Each set contained a number of simple, direct tasks, such as "Which method can we use to open a connection to a URL?" Other questions tested higher-level API fluency, such as "Which classes are commonly used to represent sockets? Please list three." The purpose of this diversity was to observe Apatite's performance in a variety of circumstances in order to learn when and how it is most useful.

10.3.3. Results

Though our study was not extensive enough to draw quantitatively significant results, there was an evident trend indicating Apatite's effectiveness. On average, participants answered three of the four question sets faster using Apatite than using Google/Javadoc. The "File Input/Output" question proved too difficult for most participants in either condition to complete in the allotted time.

Apatite had the biggest advantage over traditional approaches when the user had a good idea of what method or action he or she wanted to use but did not know which class contained it. For example, when asked "Which class and method can we use to execute a SQL database query?", most non-Apatite users first found their way to the `SQLData` class, which is actually a rarely-used class that cannot perform the task required. From there, some browsed through the `java.sql` package manually, while others tried a variety of Google searches – but even when they found source code examples, it took a notable amount of time for most users to find the `executeQuery` method in the Statement class in the `java.sql` package. On the other hand, many Apatite users searched for "execute" or "query" and found the `executeQuery` method as a top result. Those who used Apatite to first investigate classes with the word "SQL" observed that they did not have an execute action, which led them to search for ones that did. A similar effect was observed for the question, "Which method can we use to open a connection to a URL?" Apatite users usually searched for the keyword "URL" and used the popularity-based ranking to quickly identify the correct class (`URLConnection`), while the control group usually wandered around Javadoc before landing on the correct page.

Even after participants in the control group had found the correct class, often they would simply overlook the method they were looking for. Apatite's font size variations provided an advantage in this regard as well.

Apatite was also effective when users needed to discover higher-level information about a particular API. After answering some questions about popup menus, participants were asked to provide some alternative objects that a UI designer might want to use to show information in a similar manner. Javadoc/Google users who were unfamiliar with this portion of the API (most of them) struggled with this question; most resorted to scanning through the entire `java.awt` package, looking for promising class names. On the other hand, most Apatite users were able to complete the question by clicking the show method (which had been located during a previous sub-task) to discover similar classes, such as `Dialog`, `JOptionPane`, and `Window`.

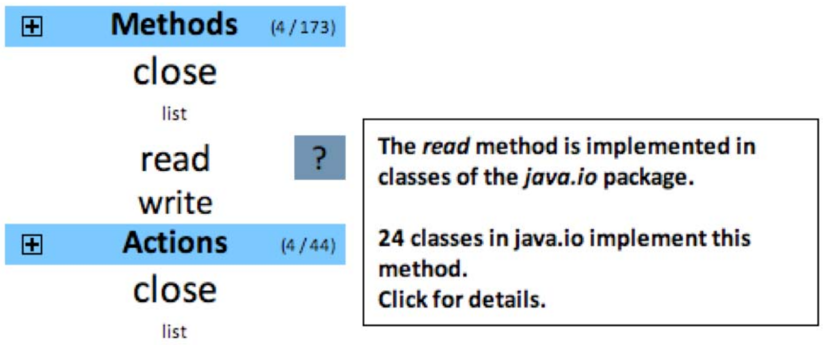
Apatite's effectiveness was most evident when dealing with relatively common methods. Since Apatite initially displays only popular methods until a user expands a category or clicks "More", participants had an easier time finding answers in these cases. This advantage faded when searching for less-common methods; if users did not know which words to search for, they typically had to browse through all of the methods in Apatite. This matched what users did when not using Apatite, using Google to find a specific class's documentation, but then simply scanning through the Javadoc to find the appropriate method.

Overall, eight of the ten participants felt Apatite was an effective tool for browsing the Java API. Most requested to be notified when it was made public and felt that it would become significantly more useful after some experience with using it over a longer period of time. Several commented that Apatite would have very been useful during introductory programming courses.

Negative comments mostly centered on the fact that it was difficult for some users to change their search strategies after using traditional methods for so long. Participants requested support for more robust text searching, providing associations based on inheritance relationships, and making association descriptions more noticeable.

10.3.4. Implications

Several features have been, or will be, added or modified based on feedback from the user study. In the original interface, lists of items had "More" buttons at the bottom which, when clicked, increased the number of items being displayed by 15. However, multiple users assumed that this button was designed to display all of the possible results. This created problems when users were searching for uncommon methods. We have made the button text clearer as a result and have added an additional "See All" option.



The *read* method is implemented in classes of the *java.io* package.

24 classes in java.io implement this method.
Click for details.

Figure 10.4. The hover-text that describes the active item was designed based on user feedback.

A more fundamental issue arose regarding how users find more information about the various items that Apatite displays. Sometimes, participants located the method that they wanted to use but had trouble determining which class contained it; they often forgot that containing classes can be viewed by clicking the method to view an additional column. Based on these observations we redesigned the text shown when an item is hovered over to be more descriptive.

10.4. Limitations

Our approach of computing popularity and associations between different API items relies on there being a large corpus that is representative of how people actually use an API, and names that can be somewhat disambiguated from common English words. For popular APIs like the base Java APIs, we think this works well, but it does not work for some other APIs, like those that are new, have too few users, or do not have sufficiently unique names. For example, the Android APIs share some but not all of the package and class names with Java, causing Android's statistics to be inaccurate when computed with our current approach. It remains to be seen whether any alternative approach, such as using source code or explicit hand weights, will be feasible in these cases.

Apatite attempts to distinguish between methods that represent actions and those that provide access to properties. The current implementation does this imperfectly and it is not clear if there is any practical approach to doing this accurately for large APIs. Showing separate sections for methods and actions, as Apatite does currently, has the disadvantage that the two lists often overlap, reducing the number of unique items that are shown. However, only showing actions and properties (and not methods) could present problems if (as is currently the case) these are not perfectly separated.

Compared to other resources like Google and Javadoc, Apatite has comparative strengths and weaknesses. Based on our experience with it, Apatite is especially good in the following cases:

- When you are not sure what class to start with.
- When you want to see what are the common classes and objects for part of an API.
- When you have a particular class or method and need to find something else to use with it.

Apatite is less useful when:

- You need to do something obscure or unusual.
- When you already know all of the classes and methods to use, and just need the details.

10.5. Future Work

In the next version of Apatite, currently under development, we plan to use a database of synonyms we are gathering that gives weighted associations between English words with each method, class and package. Using this database, we plan to enable richer search interactions, so that for example a user can type in "transparent" and see a method named `getAlpha()`. We also plan to use this database to populate the "actions" section with verbs that do not actually appear as part of the method names, allowing such alternatives as "write" for methods like `println`.

Embedding Apatite inside of a programming development environment like Eclipse could help programmers explore APIs without having to leave their code view. This would also allow the tool to use context about what code has already been written to help decide which items to display.

We have initially targeted Apatite for users who already know how to program. However, it might also be of use to those who are just learning how to program. A modified version for that audience might use nouns and verbs as the primary sections and help learners get to example code and tutorials in addition to the standard documentation pages.

We think that Apatite could be a useful tool in helping API designers create new APIs, by helping them see the commonly used parts of current APIs and by helping them think ahead of time about which classes and methods will likely be used together.

In addition to our research extensions to Apatite, we envision other new research directions inspired by our experiences. The approach of letting users browse information by association was useful in Feldspar (Chau 2008) and Apatite, and could be a useful technique in other domains as well. This approach could allow users to explore many different types of large data sets starting from different directions.

10.6. SUMMARY

Apatite demonstrates new interaction techniques for browsing APIs by association. Early user tests helped guide our design of the tool and suggest that these features offer advantages over existing tools. Apatite contains some novel features, and makes several contributions:

- A UI for visualizing the popularity of API packages, classes and methods.
- A UI for browsing APIs by action, and properties rather than just the normal package and class hierarchy.
- Techniques for finding and visualizing related classes and methods using co-occurrence in web searches.

Together these features give programmers a new view of APIs, and seem to help solve some of the difficulties using APIs observed in earlier chapters.

11.

LIMITATIONS AND FUTURE WORK

This chapter describes the limitations of this work and describes ideas for how this research could be extended by future work.

11.1. LIMITATIONS

The studies and tools in this thesis each have their own sets of limitations, as described in their respective chapters. Limitations across the approaches taken in this thesis are described here.

It may not be feasible to preemptively test the usability of every API decision or every API. However when difficult, contentious or potentially costly decisions arise, this research shows that there is an objective and effective way of deciding what to do.

This work does not present a complete and final solution to the problem of making usable APIs. But it does provide evidence that we need such APIs, and it provides a methodology for how to create them, shows how we can provide tools that enable programmers to more effectively use the APIs we already have, and answers a few questions about how to create them.

11.1.1. LIMITATIONS OF API STUDIES

One potential limitation of the results of the studies in this thesis is that they might be a product of the current education, programming tools and experiences of programmers and might not generalize to conditions in the future. However, we observed consistent results across different types of programmers with different educations and backgrounds, which suggests these results are generalizable. In addition, many of our observations are based on work styles rather than what programmers have been taught or trained to expect by existing languages, and because of this, I would expect at least some of the results to generalize to entirely different environments. One piece of evidence for this is that in some cases programmers' expectations are consistently contrary to all of the real APIs they had used before. For example, nearly all programmers expected to be able to create an empty `File` object without specifying a path, something no major I/O API allows, but programmers expectations were stronger than their repeated experiences.

11.1.2. Limitations of Tools

Our programming tools were able to leverage much of the large amount of information accessible through Google and Yahoo's search APIs. However they were somewhat constrained by their use of these as their window into the internet. A company might have the resources to use a cache of a much larger portion of the internet as the source for similar API tools. This larger source of data would likely be able to create even richer, more responsive and more useful tools.

We evaluated our tools with lab studies to verify that they actually solved the problems they were designed for. However, the true usefulness of these tools can only be shown with long term use by many programmers. While outside the scope of this thesis, we have released many of these tools publicly on the internet and hope to learn more about these tools based on their continued use by real programmers.

11.2. Future Work

The work presented in this thesis is one small piece in what I hope will become a burgeoning field. Ideas for future research are listed here in three different sub-sections. First are the major topics, from which I expect there to be multiple research contributions that would be suitable for a PhD topic. Next are the smaller research projects, which I think would be appropriate for a new PhD student or bright masters or undergrad student, and which I expect would yield a paper-worthy research contribution. Last, I list the engineering extensions that could be added to the work I have already done. These are cases where the solution is clear, but not yet implemented. While the research contribution of these items is less clear, they would be important to consider for someone looking to build and deploy a system similar to the ones in this thesis, and these features are likely to have an important effect on the effectiveness and adoption of any deployed tool.

11.2.1. Large Research Projects

By focusing on API decisions and tools to make APIs easier to use, I intentionally left several avenues of approach as outside my scope. However, many of these are likely to offer interesting results.

Designing **a programming language with API usability** in mind could be a potentially fruitful path to improving API usability. My research focused on the existing languages like Java and C#, but another researcher starting at the programming language level could use the insights from my research to help design a better programming language and better APIs on top of it. For example, the factory study showed that using factories to create objects, while it offered architectural advantages, came at the expense of discoverability. A new programming language might reduce the restrictions imposed on object constructors so that they might return existing instances (without imposing any new allocation costs) or return subtypes, neither of which is allowed by most current object-oriented languages. A programming language designer might also consider intentionally imposing limitations on the types of APIs that can be created, for the sake of simplicity and usability. For example, a language could require that every object contain a default (parameterless) constructor. Language features like aspects might make it easier to combine the different pieces of code required to use complicated frameworks [Fairbanks 2008]. Pane's work [Pane 2001] provides some insights into how usability can be considered a primary goal of programming language design, though his language was designed for children rather than professional programmers.

Related to the task of developing a programming language for API usability is that of designing an end-to-end compiler, IDE and runtime system that focused on **intelligent and understandable error messages**, while coding, debugging, and at runtime. Difficult to understand — and frequently misunderstood — compiler errors and runtime exceptions are a major source of programmer difficulty (Chapter 4) and fixing them correctly would require redesigning the entire end-to-end system to ensure that the IDE passed the right information to the compiler, the compiler stored the right information for the runtime, and that the runtime monitored the right information and correlated it with the compile-time information to generate the most sensible messages. For example, by keeping track of how code changes over time, a compiler could recognize calls to methods that have been changed or renamed, allowing it to generate not only a more sensible error message ("Method foo1 is now named foo2", rather than "No method named foo1") but also an actionable fix suggestion. By keeping track during runtime which methods in which threads set fields, the runtime system could say not just "Null pointer" but point to the thread interleaving that caused the value to be set to null or not be initialized in time. A first step of this project might involve doing a study to categorize the types of problems programmers have when reading (or not) and understanding error and exception messages. The resulting system of this project would be large and difficult to build, but could have a huge potential impact on the improvement in productivity it would offer programmers.

Designing **an IDE specifically for API consumption** would be another large but potentially profitable endeavor. While my research looked into how to improve code-completion for an IDE, and other research has examined how to add support for code examples (e.g. [Homes 2005]), someone designing an IDE from scratch would have many interesting opportunities to support API usage in the IDE user interface. Different strategies for doing this include displaying usage and API-relevant external cues, tracking where code comes from (when it is copied or inspired by webpages, for example) and where it goes (pasted or retyped), and providing different methods of displaying code (rather than just plain or syntax-highlighted text) that make patterns of API calls more apparent. Structuring code in snippets rather than files [Ko 2006] could help programmers isolate and reuse patterns of API calls. Providing support for programmers to explore multiple code alternatives or to run code snippets without creating and saving new projects or files might make it easier for programmers to try out code examples found on the web. An IDE might allow programmers to provide visual and textual cues about the intent, evolution and normal execution of code, by allowing authors to easily mark code only used in exceptional cases or initialization and making this less visible compared to the "mainline" code that best captures the normal case of an algorithm. Many of these ideas could be added on to an existing IDE, in the form of an Eclipse plugin for example, but their true power might only come from taking them into consideration during the entire design of an IDE.

Another topic that I did not cover in my research is the role of **education for learning APIs**. While this is sometime suggested as a means to avoid or ignore glaring usability problems with an API ("fix the users"), education doubtlessly plays an important role in how programmers use APIs. Computer science education provides opportunities not only for teaching specific APIs and API patterns but also teaching strategies for exploring APIs and debugging code that uses APIs.

While my tool Jadeite makes some use of explicit user feedback, I think there are many more opportunities for **"crowd-sourcing" with APIs**. Such a topic would likely require a significant number of actual users, but the would-be users are out there and no existing tool is filling this need. Though there are programming forums, there has yet to be a venue for programmers to ask and answer questions that takes advantage of crowd-sourcing techniques for percolating popular questions to the top, using moderation and meta-moderation to track and encourage helpful questions and answers, and intentionally combine randomness with its quality ranking system to avoid simply reinforcing the existing ratings.

Another potentially fruitful area of future research is the creation of meaningful **metrics for the usability** of different APIs. Currently programmers or managers deciding between different APIs or off-the-shelf components have few mechanisms for measuring or reasoning about the comparative usability of the different APIs without investing the time to try each. Benchmarks for usability, analogous to current benchmarks for speed or battery usage, could be a powerful tool for API consumers if these benchmarks accurately reflected the APIs' usability. Whether these benchmarks would have to be tailored to specific domains and whether they could be automatically run or require user studies would need to be determined.

11.2.2. SMALLER RESEARCH PROJECTS

By extending or running user studies on my existing tools, a student might make a valuable research contribution within a limited time frame.

The tools in this thesis could be more useful for more types of APIs if they used **additional sources of usage data**. The popularity of private, recently created or rarely used APIs is not captured well by the current approaches, but might benefit from new approaches such as using sample code, unit tests or the implementation of the API itself to approximate the relative importance of different parts of the API. Techniques for letting API designers manually specify the expected popularity of each piece of an API might also be useful in these cases.There is currently a lack of specific advice for API designers on how to choose the best **names** for the classes and methods in an API, despite indications that these names are very important to an API's usability. Based on observations from studies done by our research group [Beaton 2008][Jeong 2009], we think that there are general effects such as name length, word order and a name's relative position in an alphabetical list that affect API usability. With web search being a popular tool for finding and discovering APIs, uniqueness may also be an important property of good API names. Studies that generated specific advice for how to choose names in an API could help API designers more easily create more usable APIs.

The studies and tools in this thesis focused on different programming languages (Java, C#, C++, and Visual Basic .NET), but all of these are object-oriented. Studying the usability of APIs in non-object-oriented programming languages might reveal other interesting results. The libraries provided in spreadsheets and other statistical and mathematical programming environments are also forms of APIs that might contain unique usability characteristics because of the specialized nature of the environments, tasks and users.

Chapter 3 presented an overview of the design space of API design decisions. However, there is still work that could be done in coming up with more specific design decisions and different ways of organizing them. There is also more work that could be done in determining which of these is important and running user studies to help inform these design decisions.

11.2.3. Engineering Projects

As the tools created for this thesis are all research prototypes, they can all be extended in interesting and useful ways. Most of these extensions are mentioned in the limitations and future work sections of the relevant chapters, however some extensions common to multiple tools are listed here.

One class of such projects would involve integrating two or more of the existing tools. For example, the Apatite browsing interface or Mica search interface could be combined with the Jadeite, Javadoc-style documentation.

Each of the existing tools could also be better integrated with an IDE like Eclipse. The user interfaces for the tools would need to be adjusted to fit well alongside the existing code editor.

11.3. SUMMARY

This thesis describes examples of how to run usability studies on API patterns and how to design documentation and tools that take advantage of knowledge gained in these studies to make API easier to use. However, there is still much work to be done in studying API usability and creating tools to help use APIs. This chapter summarized ideas for how to extend the work in this thesis and other fruitful related research directions.

12. CONCLUSIONS

Most of the previous chapters focus on specific studies or tools. But when taken as a whole, our experience performing these studies and creating these tools provide general insights about API design and software engineering, and advice for API designers, tool and documentation creators, API usability researchers and API users.

12.1. PROGRAMMERS' STRATEGIES USING APIS

One of these insights involves our observations of the habits, strategies and work styles used by programmers today to learn and explore unfamiliar APIs.

One of our first observations of programmers' work habits is that Google is becoming a central tool for finding and learning about APIs. Despite all of the websites designed specifically for exploring code and APIs, Google's basic web search proved to be much more popular with programmers, beating out Google's own Code Search site. Why is Google's web search so much more popular than sites tailored specifically for APIs? One primary reason seems to be that by focusing on source code rather than text webpages most of the other sites are not nearly as comprehensive as Google, and do not include the discussion around code snippets. These sites also do not seem to be as representative of common uses of APIs. Websites that use code repositories as their source data tend to contain a few very large open source projects, such as those from the Apache collection of projects, in an effort to contain more code and be larger and more comprehensive. But this often comes at the expense of how representative the code returned is compared to typical programmers' API use. These observations motivated the tools created in this thesis to use Google as their repository for examples and data, to build on top of its size and ability to find representative uses of APIs.

12.2. There Is a Mismatch Between API Developers and Users

One way this research informs software engineering is by telling us that there *is* a mismatch between the APIs that are out there and the programmers who are trying to use them. The fact that there is so consistently a mismatch tells us that API design is hard and that, without user input or feedback, it is rarely done right. Sharing this research with the software engineering and human-computer interaction communities elicited a near universal recognition of the problem, and many specific horror stories about using APIs.

There are API design considerations other than usability, and these are sometimes at odds with usability. Being aware of potential usability problems and their magnitude is the first step toward making an informed decision about tradeoffs between these considerations. The studies of API design decisions in this thesis aim to better inform these tradeoffs. The specific contributions in this area are recapped in the Contributions section near the end of this chapter.

12.3. Advice About What to Do About It

This thesis brings knowledge of problems with current APIs, but it also contains general and specific advice about what API designers, tool and documentation creators, and API users can do about it.

12.3.1. Advice for API Designers

The first step for API designers is to realize that, as described above, there *is* a problem for many users of current APIs.

This thesis provides a methodology for testing hypotheses about which choices would make APIs more usable. This approach bases decisions about API design on data collected from real users, rather than designers' individual (and often differing) opinions. API designers should take a user-centered approach throughout the API design process to catch potential usability problems early and ensure that they are actually fixed before the API is released.

When creating APIs, API designers should beware of architectural cleverness. Programmers are taught to be clever, which can be good when implementing an API, but bad when creating an interface that others need to use. API designs that vary from the norm — even in ways that seem reasonable or preferable to the API designers — can cause usability problems, as the studies in this thesis show.

This thesis also offers some specific, practical advice about how to design APIs, based on these studies of API decisions. These are included in the contributions below in Section 12.4.

12.3.2. Advice for Tool Builders and API Documentation Creators

The concrete tools described in this thesis embody several general ideas for new directions that tool and documentation creators can take.

The documentation sites in this thesis use rich data sets and user interaction to blur the line between documentation and programming tools. We would like to see this become a trend, and to see API documentation become richer and more interactive, moving from static textual documentation to a dynamic, interactive, living entity. Allowing users to contribute to the API documentation in a structured manner, like Jadeite supports through its placeholders, is one powerful mechanism for moving documentation in this direction.

Programming examples have long been considered to be important for documentation, but are still often not included, perhaps because of the difficulty in manually creating examples for every class or method. Our experience with Jadeite shows that useful examples that are representative of a class's common use can be automatically located. Documentation creators should explore other ways to include automatically located examples in the cases when they are unable to manually create examples for every part of an API. Our experiences show that Google's web search is one valuable source for examples.

Traditionally, programming tools and API documentation have provided no explicit indications of relative importance or popularity of the pieces of an API. Tool and documentation designers should consider making this information more explicit and visible. Our tools show that doing so provides a powerful tool for weeding out the unimportant parts of large APIs and helping API users find what they want more quickly.

12.3.3. Advice for API Usability Researchers

Based on our experience with designing and running studies of API design decisions, we can give advice to other API usability researchers, who intend to work on research to improve the usability of APIs.

As a new research direction that fits between the existing fields of software engineering and HCI, one important consideration for API usability researchers is that of who their audience is. Research performed to inform an engineering team will be different from research designed to inform the software engineering or HCI research communities, both in terms of how the research is conducted and how it is presented. For software engineering research audiences, the first hurdle in presenting the work is to justify the general approach of user-centered research. For HCI research audiences, the first hurdle is to motivate the existence and importance of the problem to the parts of the HCI community that do not have a programming background and have never heard of APIs. The programming personas developed at Microsoft (see Chapter 1), while useful for classifying users and understanding study results, have been a source of some contention within the research communities, which can see the personas as not having been empirically validated and use this as a source of objection to the rest of the results if they are based on persona classification.

For both research audiences, special care should be taken to ensure reproducible, statistically significant numerical results; however, this limits the scope of what one can study. For more pragmatic audiences, such as API designers, one can study larger and more ambitious APIs and API design decisions, however this may rely on subjective interpretation to understand of the results.

When deciding *what* to study, usability researchers should gather as much information as possible to determine which of the many potential API design decisions would be most usefully studied. There are many specific design decisions listed in Chapter 3, however it is not sufficient to simply pick one and study it if one wants useful and interesting results. Each of the studies in this thesis relied on intuition based on previous observations of programmer, either from Steven Clarke's prior studies or from the other studies in this thesis. The studies in this thesis offer some intuition for additional studies, as mentioned in the Future Work, but other sources of programmer observations such as contextual inquiries of one's target audience may also provide inspiration and intuition for useful study topics.

Perhaps the most important piece of advice for those considering research in this area is that this is a rich and relatively unexplored research area that is likely be the source of many more interesting and valuable results. We hope that more software engineering and HCI researchers and industry practitioners will help explore and grow this field of research.

12.3.4. Advice for API Users

In *The Design of Everyday Things* [Norman 1988], Don Norman argues that people should not blame themselves for the poor design of the physical objects they use, but that they should demand better designs. From this thesis we can recommend that API consumers should act the same way. It is not programmers' fault that the APIs they consume are difficult to use, and they deserve, and should demand, more usable APIs.

The studies in this thesis showed dramatically different results in usability as a result of relatively small changes in the API. API users should consider usability an important criterion when choosing and discussing with their team which APIs and components to use.

However, when stuck with the APIs that do exist, we can recommend some techniques for successfully using them. Many of the most difficult-to-solve problems we observed programmers encountering with APIs involved the programmers not recognizing an assumption that they made about the API, for example that a particular class or method existed, or that the one class they were examining would be sufficient to complete the task. Ideally, APIs would be designed to match these assumptions, if they are commonly held, but API users might be able to make their own lives easier by becoming more self aware of all of the assumptions they make, and by attempting to consider alternatives to their assumptions when the API does not seem to match them. When considering alternatives, it may help to be able to think from the API designer's perspective, and imagine the different ways that such an API *could* be implemented.

For example, when figuring out how to create instances of new objects, API users should remember required constructors, factory methods and factory objects. Protected constructors do not always mean that subclassing is required, or that this is even a good idea.

12.4. Contributions

This thesis makes a number of knowledge (Table 12.1) and technical (Table 12.2) contributions. These are listed throughout the thesis, and are collected here.

Across all studies	Techniques for eliciting programmers' expectations about APIs, including using paper programming.
Mapping the API Design Decision Space (Chapter 3)	A map of the space of API design decisions.
	A discussion of the different ways API design decisions can be "important", including how the severity and commonality of their impact on programmers and how often and well they are made by designers.
Create-Set-Call vs. Required Constructors (Chapter 4)	Contrary to API designers' expectations, programmers preferred and were more successful with APIs that used the create-set-call pattern rather than required constructors.
	Required constructors were slower and less preferred in part because of the premature commitment they required and their larger work-step unit.
	A model (in Section 4.4) describes the work flow of how programmers explored APIs through code-completion and helps visualize why required constructors interrupted their normal work style.
Constructors vs. Factories (Chapter 5)	Programmers expect, prefer and are more successful with constructors over factories.
	When a class provides only protected constructors, and has no public subclasses, programmers often try creating their own subclass rather than looking for a factory.
	Programmers have difficulty finding factories, even when they know the factories exist.
	Programmers have difficulty using factories after they find them.
	Difficulty with any of these steps can cause programmers to falsely question their previous assumptions.
Object Design Decisions (Chapter 6)	APIs should be designed so that the most discoverable class can easily be used as an exploration point from which to discover other, helper classes.
	Programming tools and API documentation should make it easier to discover the existence and use of helper classes and methods.
	Programmers are more likely to investigate classes that appear higher up on alphabetical lists.
SAP Case Study (Chapter 7)	API usability is a problem in real APIs.
	The methodologies described in earlier chapters can be used to identify usability problems and solutions in real, large scale APIs.

	Wrapper APIs are a powerful solution to some API usability problems.
Mica Study (Chapter 8)	Evidence that programmers used Google as one of their primary API exploration resources.
	Evidence that even “rock star” programmers have a difficult time using APIs.
	A model of how programmers use Google and other internet resources to find information on APIs, going from high-level information to low-level specifics.
	Evidence that programmers have difficulty determining which API keywords are common among the top search results, and which result pages contain code examples.

Table 12.1. Knowledge contributions from studies of APIs.

These knowledge contributions helped motivate and suggest design ideas for the tools described in Chapters 8 - 10. The technical contributions of these tools are described in the following table (12.2).

Across all tools	Techniques for measuring the popularity of packages, classes and methods using the size of Google results.
Mica (Chapter 8)	Techniques for determining the Java classes and methods most correlated with the Google search results.
	Identifying results that have code examples and results that are official documentation pages.
	A new search interface that makes this information for common API keywords and pages with code examples more directly visible.
Jadeite (Chapter 9)	Tag-cloud-inspired techniques for visualizing popularity of packages and classes.
	Techniques for finding and extracting example code from Google results.
	Techniques for reconstructing type information from incomplete example code snippets.
	Techniques for aggregating examples of similar type signatures and selecting representative variable names.
	An interface technique for letting users create “placeholder” classes and methods and displaying these alongside the actual API.
	Techniques for finding related classes using co-occurrence in Google results.
Apatite (Chapter 10)	A UI for visualizing the popularity of API packages, classes and methods.
	A UI for browsing APIs by action and properties rather than just the normal package and class hierarchy.
	Techniques for finding related classes and methods using co-occurrence in web search results.

Table 12.2. Technical contributions across three tools to help programmers use APIs.

12.5. Conclusions

This thesis took a user-centered approach to developing more usable APIs, API documentation and tools. Throughout our experiences, we found along every step of the way that the same principles that applied to traditional HCI applied to programming and to APIs as well. Programming with APIs is a human activity, and this thesis provides examples of how treating it as such can uncover specific problems not previously recognized by API designers and guide solutions that offer substantial improvements.

The state of programming is in flux. Programming has shifted from being an activity of raw creation to an activity of reuse, and APIs are the primary mechanism for this reuse. However, programmers are impeded by lack of usability in current APIs. We hope that the tools and techniques presented in this thesis will help unleash some of the potential power that reusable programming offers, making programmers more productive and programming a more accessible activity.

BIBLIOGRAPHY

Aksit M., Marcelloni F., and Tekinerdogan B. (2000). Developing Object-Oriented Frameworks Using Domain Models. *ACM Computing Surveys*. 32, 1es. March 2000.

Astrachan O., Mitchener G., Berry G., and Cox L. (1998). Design Patterns: An Essential Component of the CS Curricula. *ACM SIGCSE Bulletin*. 30, 1, March 1998. 153-160.

Baeza-Yeats R. and Ribeiro-Neto B. (1999). Modern Information Retrieval. Addison-Wesley, Reading, MA.

Beaton J., Jeong S. Y., Xie Y., Stylos J., and Myers B. A. (2008). Usability Challenges for Enterprise Service-Oriented Architecture APIs. *2008 IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC). (Herrsching am Ammersee, Germany, September 15-18 2008). 193-196.

Berglund E. (2002). Library Communication Among Programmers Worldwide. Ph.D Thesis, Linköping University.

Bierhoff K. (2008). Checking API Protocol Compliance in Java. *OOPSLA '08 Companion*. (Nashville, TN, USA, October 2008). To appear.

Bore C. and Bore S. (2005). Profiling software API usability for consumer electronics. *Consumer Electronics*.

Chambers C. (1992). Object-Oriented Multi-Methods in Cecil. *European Conference on Object-Oriented Programming*. 33-56.

Chau D. H. and Myers B. (2008). What to Do When Search Fails: Finding Information by Association. Proceedings CHI'08: Human Factors in Computing Systems. (Florence, Italy, April 5-10, 2008.) 999-1008.

Clarke S. (2004). Measuring API Usability. *Dr. Dobbs Journal*, May 2004, S6-S9.

Clarke S. (2005). Describing and Measuring API Usability with the Cognitive Dimensions. *Cognitive Dimensions of Notations 10th Anniversary Workshop*.

Clarke S. (2007). What is an End User Software Engineer? *End User Software Engineering*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany.

Choo C. W., Detlor B., and Turbull D. (1999). Information Seeking on the Web: An Integrated Model of Browsing and Searching. *Proceedings of the ASIS Annual Meeting*. (Washington DC).

DeLine R., Czerwinski M., and Robertson G. (2005). Easing Program Comprehension by Sharing Navigation Data. *IEEE Symposium on Visual Languages & Human-Centered Computing* (VL/HCC), September, 241-248.

Eisenstadt M., and Peelle H. A. (1983). APL Learning Bugs. *APL Conference*, Washington D.C., 11-16.

Ellis B., Stylos J., and Myers B. (2007). The Factory Pattern in API Design: A Usability Evaluation. *International Conference on Software Engineering*. (Minneapolis, MN, USA, May 20-26, 2007). 302-312.

Evans E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

Fairbanks G., Scherlis W., and Garlan D. (2006). Design Fragments Make Using Frameworks Easier. *ACM SIGPLAN Conference on Object Oriented Programs, Systems, Languages, and Applications*. (Portland, OR, USA). 22-27.

Florijn G., Meijers M., and van Winsen P. (1997). Tool Support for Object-Oriented Patterns. *European Conference on Object-Oriented Programming*. 472-495.

Forward A. and Lethbridge T. C. (2002). The Relevance of Software Documentation, Tools and Technologies: A Survey. *Document Engineering*. 26-33.

Furnas G. W., Landauer T. K., Gomez L. M.., and Dumais S. T. (1987). The Vocabulary Problem in Human-System Communication. *Communications of the ACM*. 30(11). 964-971.

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

Gopinathan M. and Rajamani S. K. (2008). Enforcing Object Protocols by Combining Static and Runtime Analysis. *Conference on Object Oriented Programs, Systems, Languages, and Applications*. (Nashville, TN, USA). To appear.

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

Greenberg S. and Buxton B. (2008). Usability Evaluation Considered Harmful (Some of the Time). *Human Factors in Computing Systems*. 111-120.

Halvey M. J. and Keane M. T. (2007). An Assessment of Tag Presentation Techniques. *International Conference on World Wide Web.* 1313-1314.

Hoffmann R., Fogarty J., and Weld D. S. (2007). Assieme: Finding and Leveraging Implicit Reference in a Web Search Interface for Programmers. *User Interface Software and Technology*. 13-22.

Holmes R. and Murphy G. C. (2005). Using structural context to recommend source code examples. *Proceedings of the International Conference on Software Engineering* (St. Louis, MO, USA, May 15-21, 2005). 117-125.

Holmes R. and Walker R. J. (2008). A Newbie's Guide to Eclipse APIs. *International Working Conference on Mining Software Repositories*. 149-152.

Jeong S. Y, Xie Y., Beaton J., Myers B. A., Stylos J., Ehret R., Karstens J., Efeoglu A., and Busse D. K. (2009). Improving Documentation for eSOA APIs Through User Studies. *Second International Symposium on End User Development* (IS-EUD), (March 2-4, 2009. Siegen, Germany.) 86-105.

Jones S. P., Blackwell A., and Burnett M. (2003). A User-Centred Approach to Functions in Excel. *SIGPLAN Notices*. 38, 9 (Sep. 2003), 165-176.

Kagdi H., Collard M. L., and Maletic J. I. (2007). A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution. *Journal of Software Maintenance and Evolution: Research and Practice*. 19, 2. 77-131.

Knuth D. (1989). The Errors of TeX. *Software: Practice and Experience*. 19, 7, 607-685.

Ko A. J. (2003). A Contextual Inquiry of Expert Programmers in an Event-Based Programming Environment. *ACM Conference on Human Factors in Computing*, Fort Lauderdale, FL, April 8-10, 1036-1037.

Ko A. J., Myers B. A., and Aung H. H. (2004). Six Learning Barriers in End-User Programming Systems. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, 199–206.

Ko A. J., Aung H. H., and Myers B. A. (2005a). Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks, *International Conference on Software Engineering*, St. Louis, Missouri, 126-135.

Ko A. J. and Myers B. A. (2005b). A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages and Computing*, 16(1-2), 41-84.

Lawrance J., Bellamy R., and Burnett M. (2007). Scents in Programs: Does Information Foraging Theory Apply to Program Maintenance? *IEEE Symposium on Visual Languages and Human-Centric Computing*. (Coeur d'Alene, ID, USA, September 23-27, 2007). 15-22.

Liblit B., Begel A., and Sweetser E. (2006). Cognitive Perspectives on the Role of Naming in Computer Programs. *Psychology of Programming Interest Group Workshop*. (Brighton, UK, September 2006). 53-67.

Kramer D. (1999). API Documentation from Source Code Comments: A Case Study of Javadoc. *International Working Conference on Computer Documentation*. 147-153.

Mandelin D., Xu L., Bodík R., and Kimelman D. (2005). Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA, June 12 - 15, 2005). 48-61.

Marchionini G. (1995). Information Seeking in Electronic Environments. Cambridge University Press, New York, NY.

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

Myers B. A., Pane J. F. and Ko A. (2004). Natural Programming Languages and Environments. *Communications of the ACM. (special issue on End-User Development)*. Sept, 2004, vol. 47, no. 9. 47-52.

Nielsen J. (1993). Usability Engineering. Academic Press, Boston, MA.

Norman D. (1988). The Design of Everyday Things. Doubleday. New York, NY.

Pane J. (2002). A Programming System for Children that is Designed for Usability. Ph.D. Thesis, Carnegie Mellon University, Computer Science Department, CMU-CS-02-127, Pittsburgh, PA, May 3, 2002.

Pirolli P. and Card S. (1995). Information Foraging in Information Access Environments. *Human Factors in Computing*. (Denver, CO, USA). 51-58.

Pruitt J. and Grundin J. (2003). Personas: Practice and Theory. ACM Press, New York, NY.

Rha P. (2005). Detecting and Parsing Embedded Lightweight Structures. Masters thesis, Massachusetts Institute of Technology.

Ross R. G. (2003). Principles of the Business Rules Approach. Addison-Wesley Longman, Amsterdam, February 2003.

Rosson M. B. and Alpert S. K. (1990). The Cognitive Consequences of Object Oriented Design. *Human-Computer Interaction*. 5. 345-379.

Scaffidi C., Shaw M., and Myers B. (2005). Estimating the Number of End Users and End User Programmers. *IEEE Symposium on Visual Languages and Human-Centric Computing*. (Dallas, TX, USA, September 20-24, 2005). 207-214.

Stylos J. (2005). Designing a Programming Terminology Aid. *IEEE Symposium on Visual Languages and Human-Centric Computing*. (Dallas, TX, USA, September 20-24, 2005). 347-348.

Stylos J. and Myers B. A. (2006a). Mica: A Programming Web-Search Aid. *IEEE Symposium on Visual Languages and Human-Centric Computing*. (Brighton, UK, September 4-8, 2006). 195-202.

Stylos J., Clarke S., and Myers B. A. (2006b). Comparing API Design Choices with Usability Studies: A Case Study and Future Directions. *Psychology of Programming Interest Group*. (Brighton, UK, September 7-8, 2006). 131-139.

Stylos J. and Clarke S. (2007a). Usability Implications of Requiring Parameters in Objects' Constructors. *International Conference on Software Engineering*. (Minneapolis, MN, USA, May 20-26, 2007). 529-539.

Stylos J. and Myers B. A. (2007b). Mapping the Space of API Design Decisions. *IEEE Symposium on Visual Languages and Human-Centric Computing*. (Coeur d'Alene, ID, USA, September 23-27, 2007). 50-57.

Stylos J., Busse D., Graf B., Ziegler C., Ehret R., and Karstens J. (2008a). Making Code Easier to Reuse: API Design for Improved Usability. *IEEE Symposium on Visual Languages and Human-Centric Computing*. (Herrsching am Ammersee, Germany, September 15-18, 2008). 189-192.

Stylos J. and Myers B. A. (2008b). The Implications of Method Placement on API Learnability. *ACM SIGSOFT Symposium on Foundations of Software Engineering*. (Atlanta, GA, USA, November 9-14, 2008). To appear.

Stylos J. and Myers B. A. (2009a). Improving API Documentation Using API Usage Information. *Human Factors in Computing Systems*. (Boston, MA, USA, April 4-19, 2009).

Stylos J., Eisenberg D. S., Myers B. A. (2009b). Apatite: Associative Browsing of APIs. Submitted for publication.

Vokác M., Tichy W., Sjøberg D. I. K., Arisholm E. and Aldrin M. (2004). A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns: A Replication in a Real Programming Environment. *Empirical Software Engineering*. v9. 3. 149-195.

Wirfs-Brock R. and McKean A. (2003). Object Design: Roles, Responsibilities, and Collaborations. Addison-Wesley, Boston, MA.