

Mapping the Space of API Design Decisions

Jeffrey Stylos and Brad Myers
 Carnegie Mellon University
 {jsstylos, bam}@cs.cmu.edu

Abstract

When creating new application programming interfaces (APIs), designers must make many decisions. These decisions affect the quality of the resulting APIs in terms of performance (such as speed and memory usage), power (expressiveness, extensibility and evolvability) and usability (learnability, productivity and error prevention). Experienced API designers have written recommendations on how to design APIs, offering their opinions on various API design decisions. Additionally, empirical research has begun to measure the usability tradeoffs of specific design decisions. While previous work has offered specific suggestions, there has not been a clear description of the design space of all possible API design decisions, or the quality attributes that these decisions affect. This paper puts existing API design recommendations into context by mapping out the space of API design decisions and API quality attributes.

1. Introduction

Using application programming interfaces (APIs) is becoming a larger part of programming. There are more APIs than ever, and their size is growing. Large APIs like Microsoft's .NET Framework or the Java APIs have grown to thousands of classes with tens of thousands of methods, and grow larger with each successive release.

One motivation for APIs is to improve programmers' productivity by enabling the reuse of more code instead of writing it from scratch. However, having large APIs means that the usability of the APIs can be a significant barrier to programmers' productivity. In some cases, deciding how to correctly use APIs can be *more* work than writing code from scratch. However there are pressures to use the APIs anyway. Other motivations for using APIs include using UI elements that are consistent with those in other applications, and because encapsulation forces resources to be only accessible through a particular API. In observations of

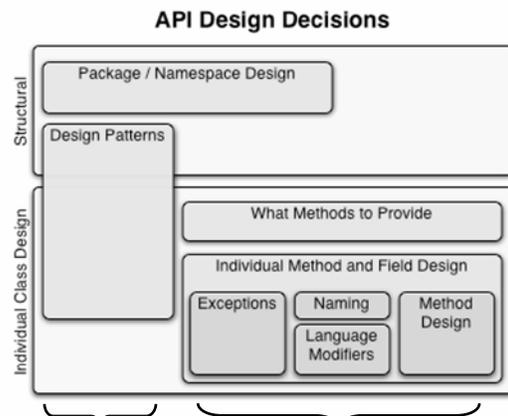


Figure 1. An overview of API design decisions relevant to object-oriented languages.

programmers, our research group has found API usability to be a significant problem for all programmers, from novices learning to program [10] to experts programming professionally [15].

To address the growing problem of API usability, some researchers are trying to design more usable APIs. Microsoft has run usability tests as part of the development of specific APIs, and has demonstrated that more usable APIs can improve programmers' productivity [3][4]. This research is promising, but expensive. Modern APIs are too large to test every feature, and so it is impractical to apply this method to each API.

Recent research aims to make APIs more usable by providing general API design recommendations that can inform the design of many APIs [6][14][16]. This research complements books written by expert API designers in which they offer their opinions on API design that have been formed by years of experience [1][5]. These research studies and expert guidelines help create better APIs by informing API developers about the tradeoffs of making different API design decisions.

Because the space of design decisions for APIs is still a relatively unexplored research area, it has re-

mained unclear how big of a space it is, and what the space looks like. This paper maps out a space of API design decisions relevant to object-oriented languages such as Java and C#. It does this by combining the topics that API experts have discussed with the list of language constructs from the programming language definitions. An overview of this API design decision space is shown in Figure 1, and this paper adds specific recommendations to the map of this space. To help define and understand this space, this paper additionally addresses the issues of what an API is, who the stakeholders are, what quality attributes they want APIs to have, and what makes different API decisions important.

These contributions provide a more solid basis for the designs of future APIs, and for API usability research in general.

2. What Is an API?

Roughly speaking, an API is a collection of existing code that other programmers can call to help accomplish programming tasks. Usually the APIs are available only in compiled form, as an interface.

APIs are created and used for several reasons. First, they can save programmers time by providing functionality that a programmer could create but which would be faster to reuse. Second, APIs provide information hiding so that, even within a single application, implementation details can be changed without affecting code that uses the API. Third, APIs can provide the application end-users with a set of consistent interaction elements, so that, for example, the color-chooser in one application acts the same as the color-chooser in another. Finally, many APIs provide access to functionality that is not easily achievable without the APIs. In the case of device drivers, operating system APIs and others, programmers wishing to programmatically interact with an existing device or piece of software must use the provided APIs because the implementation details are intentionally hidden.

Different programmers and organizations use different terminology to refer to APIs and related concepts like frameworks, libraries, toolkits, and development kits (see Table 1). These terms are not well distinguished, and are used inconsistently. However by looking at the aggregate of how they have been used most often, one can pick out connotations that vary between terms:

Libraries are small, and usually not object-oriented. They are usually stand alone (not being the sole method of interacting with an application or device). Examples include a math library and the “standard” library in C. A counter-example is the MSDN

“library” (which refers to a collection of documentation). “Class libraries” can refer to the compiled binaries of an API or application.

Frameworks are large, and usually object-oriented. They often provide an entirely new way of writing a program, and it is generally difficult to only partially use a framework. Examples include the .NET framework, the Eclipse framework, and the Microsoft Foundation Classes. Python creator Guido van Rossum says that frameworks can be more haphazard while libraries require more thought, but this distinction does not seem to match others’ usage of the terms [12].

Development kits are often attached to a language or device. The term can also refer to a distribution including a compiler, framework and possibly runtime application. Examples include the Java Development Kit (also called the JDK), and the Microsoft .NET Software Development Kit (which includes the .NET Framework, compilers, and the .NET runtime).

Toolkit is sometimes used for the collection of software that provides the user interface widgets. Sometimes, this term is used synonymously with development kit. Examples include the Macintosh Toolbox, the GIMP Toolkit, and the Google Web Toolkit.

API is used to describe both large and small “interfaces.” It is also sometimes used to describe interfaces between two components in the same program. It is possibly the most general of the terms in this section, so it is the term that is used in the rest of this document. We include libraries, frameworks, development kits, and toolkits when using the term APIs. The term can refer to either the abstract notion of the interfaces, the distribution suitable for use when programming (binaries with definition files), or the implementation source of the APIs. Some examples are the Win32 APIs, and the Google Maps APIs.

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

Table 1. Different API-related terms and examples for each. In this paper, we use “APIs” to refer to all of these together.

2.1. What Is Not an API?

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

A 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.

3. Who are the Stakeholders of an API?

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.

A commonly acknowledged property about APIs is that they need to be appropriate for their audience [1][5]; 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 [13]. "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" [8][11][2]. 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 [2]. 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.

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 use of standard widgets.

4. API Recommendation Sources

There are many scattered sources of API recommendations in print and online.

4.1. Expert Opinions

The two most comprehensive sources are currently the full-length books published on the subject by Joshua Bloch from Sun Microsystems (now at Google) [1] and Krzysztof Cwalina from Microsoft [5]. Each book presents a collection of guidelines that have been developed over several years and through the creation of such widespread APIs as the Java Development Kit and the .NET base libraries, respectively. The books are informed not just by the authors' experience but the experience of their companies.

4.2. Comparative Lab Studies

Our research group is beginning to create a new source of recommendations based on comparative studies and empirical data [16][6]. So far, we have performed studies that shed greater light on the trade-offs involved in requiring constructor arguments [16] and using the “factory” [7] design-pattern [6].

4.3. Informal Online Discussions

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 that the API guideline books tend to for a given decision.

4.4. Object-Oriented Systems Research

Finally, research on how to build object-oriented systems contains many design recommendations and tradeoffs. Many of these are relevant to the design of APIs, though it is likely that the tradeoffs for these different object-oriented architectures differ for APIs – which have a focus on reuse – than for internal implementations of software – which have a focus on maintainability.

5. What Makes an API Good?

The API recommendation sources discussed above refer to many different *qualities* that are desirable for APIs, though none attempts to enumerate them all. Sometimes the quality attributes are synonyms of each other, or else one is a specific case of another. We combine all of the mentioned attributes here, combining synonyms and forming a hierarchy of attributes. Figure 2 includes a summary of these attributes and the stakeholders most affected by each.

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.

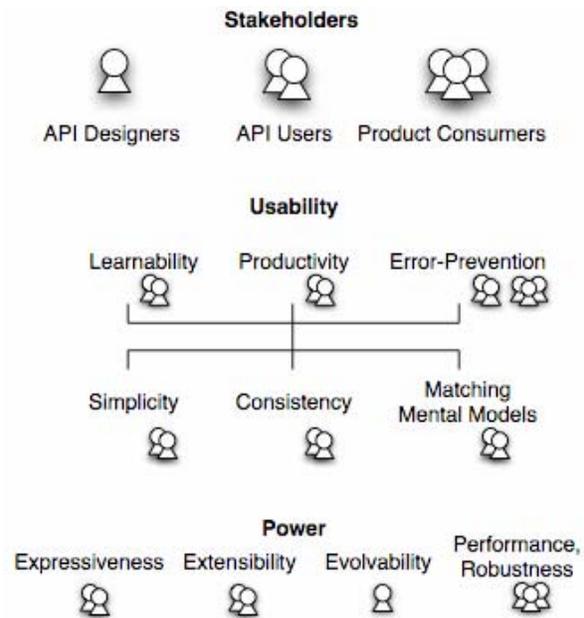


Figure 2. Quality attributes of APIs, and the stakeholders most affected by each quality.

5.1. API Adoption Issues

While the attributes above individually affect whether and how users adopt an API, there are also higher level issues that affect adoption. For example, one phenomenon is that of “death by 1000 paper cuts”: even if they do not have large problems, APIs can be unusable because of many small ones, such as multiple minor lapses in consistency or simplicity. On the other hand, a positive phenomenon is that of a “self-documenting” API: this is an API with enough positive usability attributes that users can figure it out as they go, and rarely if ever have to refer to the documentation.

Corporate policies may also affect API adoption. For example, some of the developers in one study [16] reported that, should they become stuck trying to use an API, their managers would ask them to switch to

another API or write all of the code from scratch, reducing features if necessary.

6. The 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.

6.1. 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 most of this paper, we focus only on the design decisions that directly relate to the resulting APIs themselves. However, we briefly mention the related decisions here to put these design decisions in context and to describe the space that we are *not* currently exploring (Figure 3).

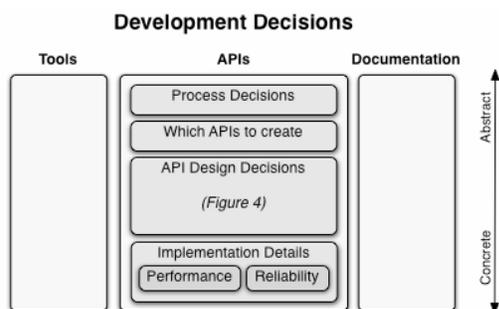


Figure 3. How the API design decisions we consider fit into the broader space of design decisions.

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.

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.

6.2. Mapping the Design Space

To build the design space described in this section, we first started with the API recommendations by experts [1][5]. These are arguably the most comprehensive lists of API design decisions, and using these as our basis 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., [17]). 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 than 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 paper 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 we consider here.

6.3. 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 our 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 4, 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 4 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.

6.4. Specific API Design Decisions

While so far, this paper 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 1, creating the more detailed map shown in Figure 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 4 contain short descriptions of API design topics. For longer descriptions, and the particular recommendations that they correspond to, see [1] and [5].

6.5. 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

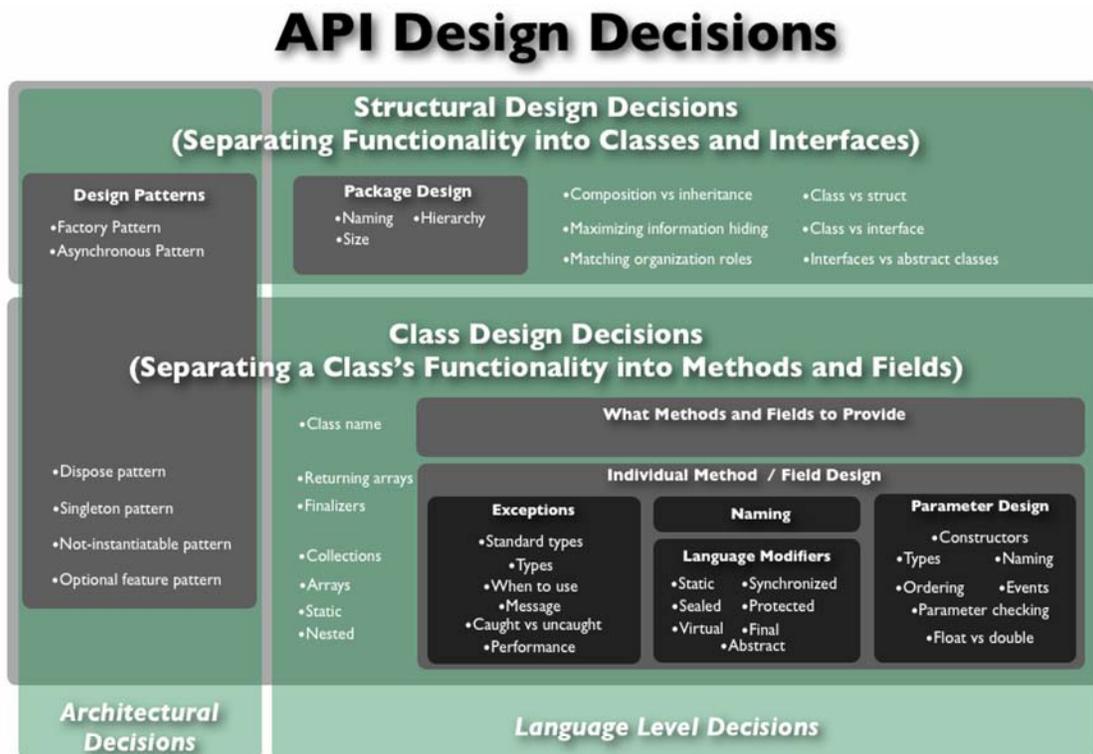


Figure 4. A map of the space of API decisions, with specific recommendation topics from API experts included as bullet-points.

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 of “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 versus 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.

7. What Makes an API Design Decision Important?

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 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 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.

8. Discussion

Perhaps the most important point made by our map of the API design space is that the space is very large. While previous literature has compiled lists of individual recommendations, the overall picture of how many design decisions there are has not before been clear. From a research perspective, this space is virtually unexplored, with many tradeoffs to be measured. An implication of this, and the size of the space, is that we need ways of prioritizing which design decisions are most important to different stakeholders in different situations. Section 7 discusses ways of prioritizing these design decisions.

9. Future Directions of API Design

The nature and importance of APIs have evolved over the past few decades, and there are interesting questions as to how they might continue to evolve.

As programming languages evolve, so will the specifics of the API design space. For example, the new “generics” feature in Java 1.5 provides a new set of decisions on how and when to use Java generics in Java APIs. Will the overall structure of the API design space change with each new programming language paradigm? Will new ways of exposing modules and services, such as natural-language based approaches,

change the *types* of decisions that are made? This remains to be seen. However, the space described in this paper covers the API design decisions made in existing languages.

Will the nature of API design become more or less centralized? The past decade has seen a few frameworks, such as the .NET libraries and the Java JDK, grow in size and importance, filling a role that used to be filled by a more dispersed collection of libraries from multiple sources. However, in addition to this apparent centralization, the internet and programming-based web search engines have increased the prominence of small third party code, examples and APIs. This *decentralization* of information has enabled tools such as Strathcona [9], which takes advantage of a dispersed collection of example code to make a single centralized framework (Eclipse) more usable. Tools like this provide one vision of the future in which a large collection of users indirectly help improve the usability of a few widely used frameworks.

10. Conclusion

This paper maps the space of API design decisions to help better understand the burgeoning and relatively unexplored field of API usability. By revealing how many decisions there are, we have greater motivation for focusing our efforts on finding the decisions that will be most worthwhile to study, and we provide different metrics for deciding which these are. We hope the definitions, design-space and metrics provided here will be of use to future API usability researchers.

11. Acknowledgements

We wish to thank George Fairbanks, Uri Dekel and Justin Weisz for their comments on drafts of this paper. This research was partially supported by NSF grant IIS-0329090 and the EUSES Consortium via NSF grant ITR-0325273. Opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect those of the NSF.

12. References

[1] Bloch, J. *Effective Java Programming Language Guide*, Addison-Wesley, Boston, MA, 2001.

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

[3] Clarke, S. Describing and Measuring API Usability with the Cognitive Dimensions. *Cognitive Dimensions of Notations 10th Anniversary Workshop*. http://www.cl.cam.ac.uk/afb21/CognitiveDimensions/workshop2005/Clarke_position_paper.pdf. 2005.

[4] Clarke, S. Measuring API Usability. *Dr. Dobbs Journal*, May 2004, pp S6-S9. 2004.

[5] Cwalina, K. and Abrams, B. *Framework Design Guidelines*. Addison-Wesley, Upper Saddle River, NJ, 2005.

[6] Ellis, B., Stylos, J. and Myers, B. The Factory Pattern in API Design: A Usability Evaluation. *International Conference on Software Engineering* (To appear). 2007.

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

[8] Grudin, J. and Pruitt, J. Personas, Participatory Design and Product Development: An Infrastructure for Engagement. *Proceedings of PDC 2002*. 144-161.

[9] Holmes, R. and Murphy, G. C. 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.

[10] Ko, A., Myers, B. and Aung, H. Six Learning Barriers in End-User Programming Systems. *IEEE Symposium on Visual Languages and Human-Centric Computing* (Rome, Italy, September 26-29, 2004). 199-206. 2004.

[11] Pruitt, J., and Grudin, J. *Personas: practice and theory*. ACM Press, New York, NY, 2003.

[12] Rossum, G. Library or Framework. <http://www.artima.com/weblogs/viewpost.jsp?thread=152104>. March 14, 2006.

[13] Scaffidi, C., Shaw, M., and Myers, B. Estimating the Numbers of End Users and End User Programmers. *Visual Languages and Human-Centric Computing*. 207-214, 2005.

[14] Stylos, J., Clarke, S. and Myers, B. Comparing API Design Choices with Usability Studies: A Case Study and Future Directions. *Psychology of Programming Interest Group*. 2006.

[15] Stylos, J. and Myers, B. Mica: A Web-Search Tool for Finding API Components and Examples. *Visual Languages and Human-Centric Computing*. 195-202. 2006.

[16] Stylos, J. and Clarke, S. Usability Implications of Requiring Parameters in Objects' Constructors. *International Conference on Software Engineering* (To appear). 2007.

[17] Wirfs-Brock, R. and McKean, A. *Object Design: Roles Responsibilities, and Collaborations*. Addison-Wesley, Boston, MA, 2003.