Crossroads
The ACM Student Magazine

Why Are APIs Difficult to Learn and Use?

By ***Christopher Scaffidi***

Introduction

An Application Programming Interface (API) comprises reusable functionality accessible through parameterized commands or functions. Examples include the Java and .NET class libraries, the C++ standard template library, and the system calls exposed by operating systems. An API serves as a foundation for creating applications by saving programmers the time necessary to code basic functionality from scratch.

In a textbook-perfect world, using APIs to build an application would be a simple three-step process:

1. Gather requirements from clients and other stakeholders.
2. Map requirements to API components and functions.
3. Type in "glue code" to combine the APIs' functionality into the required application.

Unfortunately, real software development with APIs is not so simple. In fact, programmers experience significant challenges when using APIs for many categories of functionalities: networking, databases, web applications, web services, graphics, user interfaces, text processing, and so forth.

Learning and using APIs can be difficult for reasons stemming from the very nature of software. For example, due to its high ductility, software can evolve quickly, which means that APIs can rapidly become outdated. In addition, software is intangible--unlike physical machines, software cannot be "opened up" to reveal how it works, so when learning an API, programmers cannot call on many of their tactile-visual skills.

Finally, software complexity results not only from inherent factors but also from a great variety of external factors: "Are the right DLLs installed? What are the right build parameters? Is this unofficial API documentation on a random web site trustworthy?"

Moreover, learning and using APIs can be difficult for reasons stemming from the culture around software. For example, there is incredible schedule pressure due to a desire to be first to market, so designers lack the time for creating as much API documentation as might be desired. However, the users of the APIs are under similar pressure, so they are often unwilling to invest time in wandering through poorly tested and documented APIs. Thus, universal schedule pressure creates a tension: API designers can never supply as much documentation as users demand.

This article focuses on four specific challenges related to learning and using APIs. The discussion of each challenge includes an outline of strategies that API users employ for dealing with that challenge, as well as strategies that API designers employ for helping their respective API users. Although most programmers learn these strategies over the course of many years, perhaps they could learn some of these strategies before they enter the real world.

Challenge #1: inadequate documentation

Programmers get help with APIs from a variety of sources, each with particular strengths and weaknesses.

- Printed tutorial books like *Professional C++* [**13**] incorporate a gentle learning curve. Their weakness is their relative lack of searchability (even with an index), and they may not exist for newer APIs.
- Online tutorial books like "SQLite Documentation" [**8**] often offer better searchability without sacrificing the gentle learning curve. However, they may not be subject to the same level of editorial control as a printed book and may contain numerous errors.
- Online API reference documentation like JavaDoc is ubiquitous. One weakness is that finding an up-to-date document can take a long time. In addition, programmers can have a difficult time determining which functions are optional and which are required in order to achieve complex effects.
- "Googling for code" often proves helpful for open source APIs, since many developers have used these APIs and published their code on the web. If a set of API functions looks promising, but the proper sequence of calls is unclear, then it

can be helpful to type function names into Google to retrieve examples where people have successfully navigated the API. The weakness is that code samples alone reveal little of the semantics behind functions.

- The final source of help is human contact through email, newsgroups, or direct verbal communication. This permits asking follow-up questions, and experts can tailor their instructions to a specific problem instance. The main weakness is that such people may have time to answer only a few questions by email each week.

In summary, although each piece of documentation provides some help, each piece also leaves some uncertainty about how to use an API. That is, documentation gives part of the picture, such as a list of available commands, but it may leave other parts foggy, such as which commands are optional.

Strategies relevant to incomplete documentation

Recurring Themes

The following themes recur in the strategies discussed in this article:

- *Make the problem smaller.* The problem is to map requirements to a new application. The first step is usually to divide this problem into smaller parts, each of which can be conquered and composed into a larger solution. "Make the problem smaller" applies to other strategies, as well, such as consciously ignoring part of the problem because a solution to that part already exists.
- *Accept approximate results when necessary.* Software engineers accept and implement

easy requirements that satisfy most of the clients' needs and then push back on less important requirements that would take longer to implement. The result is an application that is almost ideal from the customer's standpoint, and still affordable.

- *Pick an approach that is efficient on average.* Sometimes a programming strategy quickly leads from requirements to application; sometimes a strategy does not work so well. Many programmers use strategies that usually work well but might take a terribly long time on occasion.

Strategies of API users

Although students and young developers might refer to only one source of documentation for an API, they eventually learn to seek several overlapping sources of documentation, which helps reveal subtleties in the API's usage. Each piece of documentation can reduce the uncertainty of which API function to use, *making the problem smaller*.

In addition, experienced API users recognize when it is acceptable to use API functions whose semantics are only *approximately* understood. Not every application runs a pacemaker! Sometimes it is better to write code that might contain a bug, and then fix the bug if it appears, than it is to try to produce a perfect program. For example, if a script needs to initialize a new application's database from a legacy system, then it is acceptable for the script to use somewhat imperfectly understood API functions, since the script only needs to run correctly once.

Strategies of API designers

API designers use a variety of strategies to facilitate the strategies of API users. For example, designers usually provide several forms of documentation so that users can employ the first strategy above.

Perhaps the most powerful documentation strategy for API designers is to provide "Hello World" examples. The "Hello World" term originated in reference to examples of how to print "Hello, World!" to the console; the first such example can be found in [**9**]. The term more generally can encompass any example that gives painstakingly precise instructions for how to accomplish a specific individual task using an API.

This sort of example includes the source code to meet a single requirement, making it utterly transparent which functions are required for one very simple API usage scenario; by implication, all other API functions are optional in the context of that scenario. The example also includes step-by-step instructions for building and executing the example.

A "Hello World" example leads the user directly to the code needed for one API use case. Of course, there are many such use cases, and the API designer probably cannot provide a "Hello World" example for all of them--so in the *worst case*, the API user might still fall back on guessing games. Yet if the API designer writes "Hello World" examples for the most common API use cases, then in the *average case*, the API user may not need to guess much at all.

Finally, note that users must have an obvious, easy way to look up the version number of the installed API so that they can find the right documentation for their version of the API. Astonishingly, many so-called tutorials lack the most basic console-printing "Hello World" example, and some APIs lack obvious version numbers.

Challenge #2: insufficient orthogonality

Using one portion of the API may cause another portion to function differently. In this sense, the internal couplings within the API implementation prevent users from conceptually separating usage of one API function from usage of another API function.

For example, in many database APIs, using SQL to change a schema's structure has the side effect of committing any currently pending transaction. Thus, the functionality

for creating, altering, and dropping tables is not orthogonal to the functionality for starting, committing, and rolling back transactions. This nonorthogonality essentially limits the range of applications that programmers can easily create without much extra code complexity.

In general, nonorthogonality can result from any coupled state between different portions of the API. Broadly speaking, such state could be stored in a variety of places--files, main memory, BIOS, remote servers, and so forth.

Because nonorthogonality can occur in so many ways, it is very easy for API designers to generate accidental nonorthogonality. Unfortunately, in a truly worst-case situation, such nonorthogonality can make it difficult to predict how using one portion of an API could affect the behavior of other portions of the API.

Strategies relevant to insufficient orthogonality

Strategies of API users

At times, it makes perfect sense not to worry about API nonorthogonality. For small applications that will never use more than a handful of API functions, virtually all couplings involve functions that will never be used. In this case, programmers just grab whatever API function seems suitable for the task.

In fact, that is the *average case* for some organizations, and this allows developers to perfectly achieve the application's requirements without having to think too hard. Of course, if the application continues to grow, then the API's internal couplings might eventually become problematic, forcing major refactoring in order to continue producing optimal application behavior.

However, many companies do not have the luxury of ignoring an APIs' nonorthogonality. So when programmers review one another's code before check-in, they often ask whether it is necessary to use various portions of APIs. Skilled software developers are keenly aware that using an API constrains future usage of the API, so they need a strategy to help evaluate whether future development is grossly hampered by current usage of the API functions.

In general, to make such an analysis tractable, programmers do not evaluate API functions individually, but instead, look at groups of API functions that seem intercoupled as a single identifiable unit. For example, rather than individually

evaluating each function for launching threads, it makes more sense to evaluate a group of basic functions for launching and synchronizing threads.

Experienced programmers do not use a group of API functions unless the group passes two tests, based on experimentation and documentation. First, programmers must feel that they understand and can live with the internal couplings within the group of functions. Second, they must believe that this group does not couple to other vital groups of functions in any way that would inhibit future development or break an existing piece of the application. If the group of functions does not pass muster, then the programmer might keep "shopping" for an appropriate function or might code an equivalent function from scratch.

Such a strategy is only *approximately* optimal, since some intergroup coupling will still slip in occasionally. This occurs largely because experimentation and documentation reveals most but not all of an API's internal couplings.

Strategies of API designers

The job of an API designer is to facilitate the API user's strategies as described above. However, a certain amount of nonorthogonality is unavoidable, since the nonorthogonality can be what makes the API useful in the first place. Hence, designers of successful APIs put the nonorthogonality only in natural places--within the same function group, so to speak. For example, it makes sense that a function for terminating threads cannot be called until after the corresponding function for creating threads is called; such nonorthogonality is natural and appropriate.

In other words, creating only the natural intra-API couplings and preventing spurious couplings limit the number of couplings to *keep the problem as small as possible* for API users. A wide variety of reference materials exist for coaching API designers on the tactics of achieving this [**1**] [**2**] [**4**] [**10**]. Many of these tactics hinge on understanding the requirements that prospective API users face, and the next section will pick up this line of thought.

Finally, designers of successful APIs document nonorthogonality so that users are not surprised and can make reasoned decisions about which API functions to use.

Challenge #3: inappropriate abstractions

APIs sometimes provide exactly the right abstraction to meet application requirements. For example, XUL cleanly abstracts localization strings into an XML file that can conveniently vary by deployment locale [**11**], which makes it exceptionally easy to satisfy localization requirements.

At the other extreme, APIs may completely lack some valuable abstractions. For example, C++ lacks a function for determining whether a string ends with certain characters (whereas string instances in Java have an `endsWith(String)` method). At a higher level of abstraction, many cryptography packages lack a simple specific function for decrypting a source file to generate a destination file. At a still higher level, most platforms lack a component to detect, download, and install upgrades automatically. All three of these examples represent abstractions that programmers must code for themselves.

Sometimes the mismatch between requirements and API abstraction is more subtle. For example, XUL provides a tree widget visually similar to that of Windows Explorer; this tree almost perfectly matches a user interface requirement common to many applications. Unfortunately, the platform only ships with one tree view, which requires all data to be loaded into memory; if an application has millions of rows, then it becomes necessary to implement a custom view in order to conserve memory.

There are a variety of reasons why a programmer might have to settle for an API with inappropriate abstractions. Perhaps no better API exists. Perhaps the company lacks money to license a better API. Or perhaps the API has become part of the company's infrastructure and cannot easily be changed. In any such case, choosing an API with inappropriate abstractions makes the programmer's job more difficult.

Strategies relevant to inappropriate abstractions

Strategies of API users

One characteristic of successful programmers is a well-developed ability to imagine and implement new abstractions. When a given API fails to satisfy a particular requirement, such programmers at least have a supply of appropriate abstractions conceptually available.

In fact, successful programmers maintain personal toolboxes of already-implemented, ready-to-reuse abstractions. This is a time-honored strategy: Alan Perlis cited the

value of such "modularity" and "standard parts" as early as NATO's 1968 software engineering conference [12], and Fred Brooks insisted that every software development team should have a "toolsmith" responsible for creating "specialized utilities, catalogued procedures, and macro libraries" [3]. In effect, this strategy *makes the problem smaller,* since the programmer can check off requirements satisfied by personal toolkit functions in addition to requirements satisfied by API functions.

Incidentally, this strategy also makes the programmer's job more difficult in some ways. First, it demands consideration of whether investing in a reusable abstraction will pay off in the future. Second, for some programmers, the existence of a personal toolkit can prove to be a powerful "switching cost" when adopting new platforms where old toolkits might not apply, and this can limit future employment opportunities. However, despite these caveats, cultivation of reusable abstraction implementations has proven to be a valuable strategy for many programmers coping with the inadequacies of an API.

A corollary strategy for API users is to create documentation and unit tests for their reusable functionality. The documentation is essential for helping other members of a team reuse the functionality. For large multiperson projects, this investment can pay off even before the current application ships, but it may benefit other projects as well. Of course, the documentation strategies mentioned earlier for API designers also apply here.

Unit tests are valuable because upgrading the API implementation may cause a function that previously met a requirement to stop functioning properly. At some companies, policy dictates that engineers must write unit tests for every new class. This policy can pay off quickly when using open source APIs, which can evolve quickly and with little warning.

Strategies of API designers

API designers can do much to ensure that the API's exposed abstractions match requirements of concern to prospective users. This helps make the "programming world" more closely resemble the "problem world" [7].

Of course, it is naive to think that the API designer can provide abstractions for every single user's application requirement. On the other hand, it is laziness if API designers do not spend time studying prospective users or do not try to match abstractions to

the most commonly occurring requirements. Matching abstractions to as many requirements as possible helps to *make the problem smaller* for API users, who can check off more requirements using API functionalities.

Three factors govern this strategy of producing functionality that suits API users' application requirements.

First, each abstraction should be deliberate, rather than mimicking idiosyncrasies in the underlying primitives used to implement the API. For example, the author recently needed some cryptography functions for signing messages and verifying the signature on the receiving end. The underlying API contained hundreds of "primitive" functions, but none of them exactly fit the requirements. Searching the web for a C++ wrapper with "sign this" and "verify this" operations only turned up wrappers that simply carried over the API's complexities to C++ syntax. The only option was to implement a wrapper that cleanly exposed the desired functionality, neatly hiding the complex calls on the underlying API by providing only a handful of methods, each of which exactly suited a clear requirement.

Second, API designers should consider using a facade pattern [**5**] to provide shortcuts for frequently needed sequences of operations. For example, libcurl offers the so-called "easy interface" for http downloads, in addition to a gorier (and more powerful) nuts-and-bolts API [**14**]. This represents a modified facade pattern, since the facade does not completely hide the full API, but rather provides shortcuts. Interestingly, the API designers did not provide the easy facade in the initial version of libcurl; they added it after they saw what sequences of operations occurred most frequently in users' code.

Finally, part of why APIs seem inappropriate for requirements is because it can be very difficult to find the right API functions. This is more than a documentation problem, as it often comes about because API designers fail to keep "conceptual chunks" reasonably sized. For example, classes should not have an excessive number of methods, and inheritance hierarchies should not contain more than a few levels. Anything larger is too difficult for humans to grasp [**1**]. Bloated packages, classes, and method signatures exemplify "bad smells" and call for redesign [**4**].

Challenge #4: incompatible assumptions

All too often, the assumptions built into one essential API conflict with the assumptions built into another essential API. Such conflict makes the API user's job more difficult.

As a simple example, using web services demands a multi-threaded architecture. Otherwise, the application becomes unresponsive during network downloads. On the other hand, if an API's components live in single-thread apartments, then threads from outside the apartment cannot call methods on those components; the thread doing downloads must pass data through a thread-safe buffer to another thread responsible for calling methods on the components. This exemplifies a control model conflict--one type of architectural mismatch, which is a general class of problems discussed in more detail in [**6**].

In addition, as the author's team discovered on a recent C++ project, APIs might conflict not only with one another but also with the software development process.

These conflicts resulted from a series of constraints. First, using the simple object access protocol (SOAP) to retrieve data from web services uses character sequences to contain data values. On the other hand, C++ uses objects' typed member fields to contain data values. Thus, a conversion becomes necessary when populating the fields with data from a web service. (This constitutes a data model form of architectural mismatch [**6**].) Fortunately, tools exist for generating code that automatically performs this conversion, and the team selected gSOAP [**15**] to generate the code for the application. Unfortunately, as it turned out, gSOAP generates code that uses underscores and capitalization in ways that conflict with the company's coding guidelines. (Additional conflicts resulted from other sets of constraints.)

These conflicts became manifest during readability reviews by other employees. Not surprisingly, the reviewers complained bitterly about the code's poor adherence to the coding guidelines. After many weeks of negotiation, the reviewers decided to overlook some perceived problems, but they still demanded various changes that were not compatible with the API. In short, conflicting assumptions between the APIs and the development process can create quite a hassle for API users.

Strategies relevant to incompatible assumptions

Strategies of API users

In light of the challenge of evaluating and navigating the conflicting assumptions of APIs, most API users seem to take whatever API is handy and build from there. Later, as necessary, they incorporate additional APIs. If adding a certain API seems too

difficult, then they skip it and "shop" for an API that fits better. In the *average* case, such a strategy will allow the application to grow without a huge amount of the brute force hacking alluded to above.

In the *worst* case, however, the application might evolve to the point where there are no relevant APIs available that do not badly conflict with APIs already in use. At that point, the developers may resort to nasty hacks. Alternatively, they can back out an API and replace it with another; of course, this task is eased significantly if the previous API was hidden behind an encapsulating layer within the application's architecture.

As a complement to this "greedy" strategy, the team can *make the problem smaller* by standing firm on as few process-related constraints as possible. Yet this does not imply that teams should never use ceremonious processes! Instead, the team should be somewhat flexible on process. For example, trying to force-fit coding guidelines to code that calls an API is sometimes impossible. Such a conundrum calls for careful trade-offs, and it could turn out that a team should eschew certain APIs. In other words, the decision should result from a careful evaluation of the options, rather than an obdurate "we don't do that here."

Strategies of API designers

API designers can *make the problem smaller* by minimizing API dependencies. For example, if one portion of an application can be implemented using an API that is already in use by another portion of the application, then there is a strong disincentive to bringing another API into use. Using unnecessary, redundant APIs can add another source of bugs and can increase download size unnecessarily.

In addition, through the strategy of providing graceful degradation, API designers can reduce the cost associated with a conflict of API assumptions. This makes it easier for API users to live with *slightly sub-optimal* applications. For example, because of some details about how Linux manages threads, it appeared at one point in a recent project that the author could not use Linux, the SQLite database API [**8**], and multi-threaded access to web services all at the same time. Rather than create a nasty hack or punt the API, the team took advantage of the fact that web services can be used in a single-threaded fashion. This hurt the application's responsiveness, but this cost was reasonable, considering that few users had Linux. (Later, a build process tweak became apparent that enabled using multi-threaded downloads again.) In short, a little

bit of one quality attribute was expended to paper over a bit of architectural mismatch.

Summary

Summary of main points

This article has presented four reasons why using an API is often difficult. Fortunately, API users have a variety of strategies that make the job of using an API more tractable. API designers use a complementary set of strategies in order to make their API users' lives easier. In general, three themes recur throughout these strategies:

- Make the problem smaller.
- Accept approximate results when necessary.
- Pick an approach that is efficient on average.

Themes like these are typical of an engineering discipline, and they undoubtedly appear in other strategies for similar API-related challenges. As software engineering continues to mature, themes like these will hopefully diffuse throughout the discipline's textbooks and curricula so that new programmers will be better equipped for facing the APIs of the real world.

References

1

Bloch, J. *Effective Java*. Addison-Wesley, Boston, MA, 2001.

2

Bloch, J. "How to Design a Good API and Why it Matters," keynote in *Library-Centric Software Design Workshop*, 16 October 2005, <**http://lcsd05.cs.tamu.edu/slides/keynote.pdf**> (28 November 2005).

3

Brooks, F. *The Mythical Man-Month: Essays in Software Engineering*. Addison-Wesley, Boston, MA, 1995.

4

Fowler, M., et. al. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Boston, MA, 1999.

5

Gamma, E., et. al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, 1995.

6

Garlan, D., Allen, R., and Ockerbloom, J. Architectural Mismatch Or Why It's

Hard To Build Systems Out of Existing Parts. In *ICSE '95: Proceedings of the 17th International Conference on Software Engineering*, ACM Press, 1995, pp. 179-185.

7

Green, T. Cognitive Dimensions of Notations, In *Proceedings of the Fifth Conference of the British Computer Society*, British Computer Society Human Computer Interaction Specialist Group, 1990, pp. 443-460.

8

Hipp, D., et. al. "SQLite Documentation", version 3.2.x March-June 2005, <**http://www.sqlite.org/docs.html**> (28 November 2005).

9

Kernighan, B., and Ritchie, D. *The C Programming Language*, Prentice Hall PTR, 1988.

10

Meyers, S. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, Addison-Wesley, Boston, NJ, 2005.

11

Mozilla community. "XULPlanet.com", *Semi-official XUL online documentation*, June 2005, <**http://www.xulplanet.com/**> (28 November 2005).

12

Software Engineering: Report on a conference sponsored by the NATO Science Committee (Naur, P., Randell, B., eds.), IEEE Press, Piscataway, NJ, 1968.

13

Solter, N., and Kleper, S. *Professional C++*, Wiley Publishing, Hoboken, NJ, 2005.

14

Stenberg, D., et. al. "libcurl- what makes it special", *Official libcurl online documentation*, version 7.1.0, May 2005. <**http://curl.haxx.se/libcurl/features.html#docs**> (28 November 2005).

15

van Engelen, R. "gSOAP: SOAP C++ Web Services," *Online gSOAP documentation*, June 2005, <**http://www.cs.fsu.edu/~engelen/soap.html**> (28 November 2005).

Biography: Christopher Scaffidi is currently a software engineering graduate student at Carnegie Mellon University. Prior to entering graduate school, he worked as a web application developer and a Java programmer. His most recent applications involved open source APIs, including XUL, OpenSSL, and libcurl. Home page: **http://www.cs.cmu.edu/~cscaffid/**.