APIs with an Appetite

Time for everyone's favorite subject again: API design. This one just doesn't get old, does it? Well, OK, maybe it does, but leave it to Kode Vicious to inject some fresh insight into this age-old programming challenge. This month KV turns the spotlight on the delicate art of API sizing. Have more API questions? Send them, along with any other coding queries, to kv@acmqueue.com. You'll be glad you did.

Dear KV,

This may sound funny to you, but one of my co-workers recently called one of my designs "fat." My project is to define a set of database APIs that will be used by all kinds of different front-end Web services to store and retrieve data. The problem is that a one-size-fits-all approach can't work because each customer of the system has different needs. Some are storing images, some are storing text, sound, video, and just about anything else you can imagine. In the current design each type of data has its own specific set of APIs to store, search, retrieve, and manipulate its own type of data. If I constrain the API too much, then some group won't be able to use it and will build its own local API. That means we will lose the advantages of having a central store for all different types of data. As it stands now, there are about 500 different API calls in the library. Is that too fat?

API Diet Plan

Dear ADP,

Public ridicule is never funny, except, well, when it is. "Is my API too fat?" is definitely a new one on ol' KV, which is no small feat.

The fact is, APIs can run the gamut from one do-it-all function to libraries of thousands of very specific

Got a question for Kode Vicious? E-mail him at kv@acmqueue.com—if you dare! And if your letter appears in print, he may even send you a Queue coffee mug, if he's in the mood. And oh yeah, we edit letters for content, style, and for your own good!

A koder with attitude, KV ANSWERS

YOUR QUESTIONS.

MISS MANNERS HE AIN'T.

functions. As with many things in software, there is a spectrum, and picking the right place on the dial is not easy. The short not-

so-helpful answer to your question is that you want "just enough APIs but not too many." Let me try to be a bit more helpful than that.

On one end of the API spectrum is the single do-it-all function that is all things to all people. Using a dizzying variety of parameters, all of which can be various types of data, it is possible with this one magic API to do everything in the system. In C the function usually looks something like this:

```
void *do_as_i_say(...)
```

The ellipses (those three dots) indicate that the function takes a variable set of arguments and that it returns a pointer to void, thereby allowing the function to return almost any kind of structure.

While such a function is almost infinitely variable, it doesn't really give the caller any idea of what can be done with it without reading the inside of the function itself. Functions are supposed to act as interfaces to complex pieces of software; by hiding that complexity, they make the underlying software easier to use.

At the opposite end of the spectrum are the fat APIs, which may or may not be what you yourself have written. Now I am hoping—and if I were religious I might even be praying—that you didn't design something like the following atrocity:

```
int store_jpeg_image(string name, image data);
image get_jpeg_image(string name);
int delete_jpeg_image(string name);
int store_gif_image(string name, image data);
image get_gif_image(string name);
int delete_gif_image(string name);
int store_tiff_image(string name, image data);
image get_tiff_image(string name);
int delete_tiff_image(string name);
```

```
int store_text(string name, string data);
string get_text(string name);
int delete_text(string name);
...
```

and on and on for 500 APIs. If you did this, then, yes, your API is not only fat but is actually morbidly obese. It is time to staple its stomach and put it on a diet of very thin soup.

There are several problems with the fat API I've outlined here, and they plague all such similar designs. The first problem is that certain things, which ought to be passed as arguments, are encoded into the function name itself. Images are often handled in similar ways, and any nits that happen between them can and should be handled within the function you're providing. All those APIs with tiff, gif, and jpeg can be replaced by a `store_image`, `get_image`, and `delete_image` set of APIs, and the type of image can be encoded as an argument. For those

The classic, and perhaps now cliché, example of a good API is the Unix `open`, `close`, `read`, `write`, `ioctl` set of system calls...

of you laughing right now, stop! I have seen APIs like this and they do not make me laugh. They are probably responsible for the inordinate amount of teeth grinding my dentist has recently warned me about.

The next problem with this set of APIs is that they are very likely going to be implemented by a set of copy/paste operations. That is, the programmer is going to write one set first—say, the one for GIF files—and then will copy and paste that code to handle the other file types. The dangers of copy/paste were outlined in a previous KV (October 2005), but I'll point it out again here: When code is copied and pasted thoughtlessly, bugs are replicated and then are harder to fix. If there is a bug in the GIF handling, it will be propagated to all the other file types, and then when the “junior bug fix-it guy” comes along and fixes the GIF code, the remaining bugs will not

get fixed. Keeping common code in a common place is good common sense.

Lastly, this set of APIs is almost mind-numbing to read and remember. The programmer using this library has to remember the names of all the different things they might store, instead of simply remembering that there is a store operation and then looking up in the documentation (wait, there is documentation, right?) any particular special needs of the data type they are storing. Human short-term memory has, on average, seven places to store things. If you're going to ask people to remember something extra, you better have a very good reason for doing so or they'll soon forget... uh, what was I saying?

The classic, and perhaps now cliché, example of a good API is the Unix `open`, `close`, `read`, `write`, `ioctl` set of system calls for performing file I/O. Unix cheated, in a way, by saying that all files were just streams of bytes without structure, but nonetheless it stands as a good example because with those five APIs most programmers can do most of the things they need to do with file I/O.

Perhaps the most important, and least recognized, API in the Unix file I/O set is the `ioctl` call. This call operates as the universal escape clause, the function anyone can use—in some ways like the `do_what_i_say` API I showed earlier—to pass data from a program down into the bowels of the system, in order to do something the original designers did not think to provide. It is omitting this type of call from an API set that causes people to overspecify and create a function for everything that could ever happen. Overspecification leads to libraries of hundreds of APIs, of which maybe only 10 are needed and 100 are ever used.

API design is an iterative process. First you provide what you think the users need, as well as an escape valve such as `ioctl`. After a while you see which functions people are providing via the escape valve and you standardize those in your library as full APIs. Repeat until done.

KV

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who has made San Francisco his home since 1990.

© 2007 ACM 1542-7730/07/0300 \$5.00