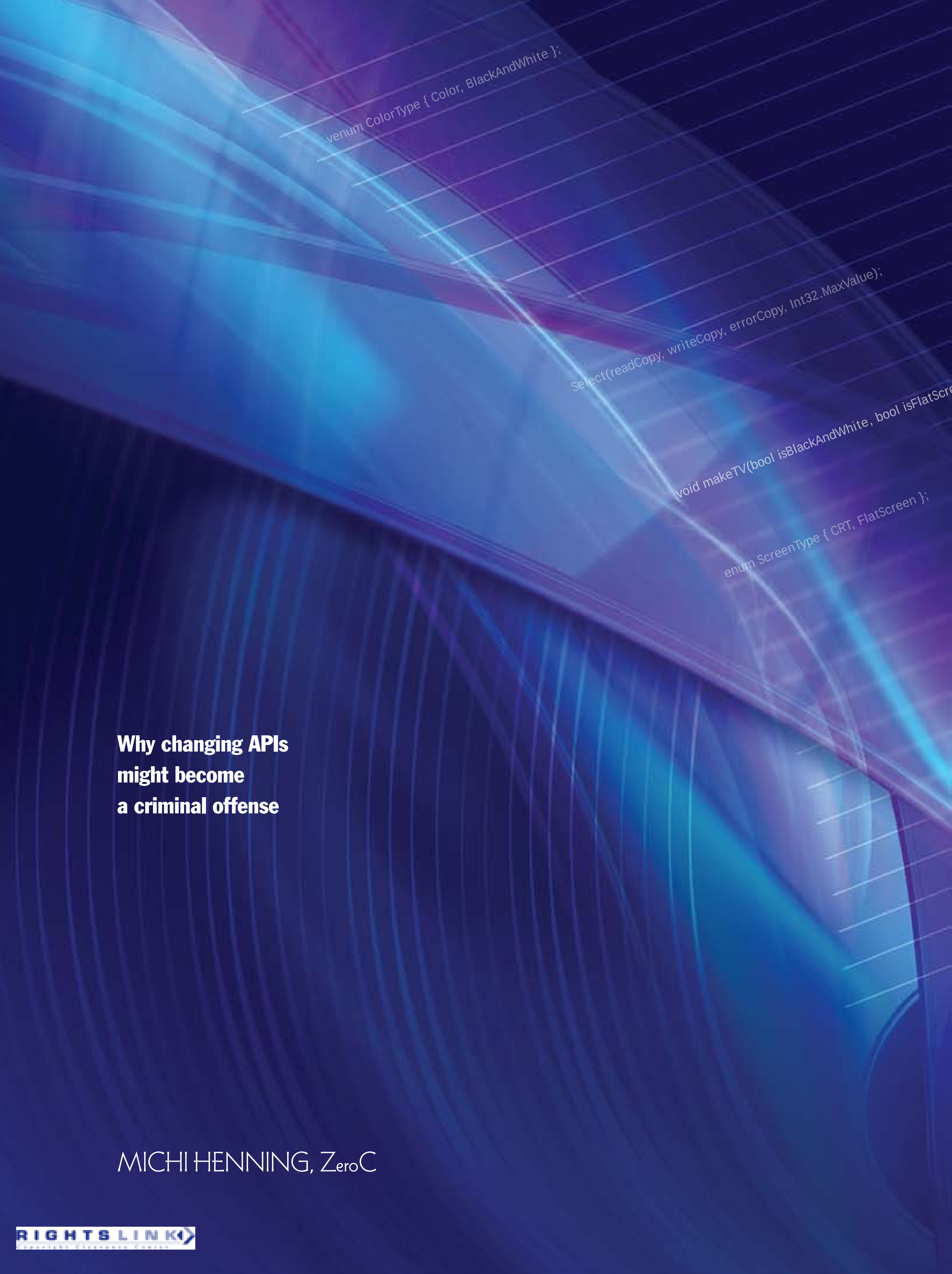

venum ColorType { Color, BlackAndWhite };
Select(readCopy, writeCopy, errorCopy, Int32.MaxValue);
void makeTV(bool isBlackAndWhite, bool isFlatScre
enum ScreenType { CRT, FlatScreen };

Why changing APIs might become a criminal offense

MICHI HENNING, ZeroC

API Design Matters

After more than 25 years as a software engineer, I still find myself underestimating the time it will take to complete a particular programming task. Sometimes, the resulting schedule slip is caused by my own shortcomings: as I dig into a problem, I simply discover that it is a lot harder than I initially thought, so the problem takes longer to solve—such is life as a programmer. Just as often I know exactly what I want to achieve and how to achieve it, but it still takes far longer than anticipated. When that happens, it is usually because I am struggling with an API that seems to do its level best to throw rocks in my path and make my life difficult. What I find telling is that, after 25 years of progress in software engineering, this still happens. Worse, recent APIs implemented in modern programming languages make the same mistakes as their two-decade-old counterparts written in C. There seems to be something elusive about API design that, despite many years of progress, we have yet to master.

GOOD APIS ARE HARD

We all recognize a good API when we get to use one. Good APIs are a joy to use. They work without friction and almost disappear from sight: the right call for a particular job is available at just the right time, can be found and memorized easily, is well documented, has an interface that is intuitive to use, and deals correctly with boundary conditions.

So, why are there so many bad APIs around? The prime reason is that, for every way to design an API correctly, there are usually dozens of ways to design it incorrectly. Simply put, it is very easy to create a bad API and rather difficult to create a good one. Even minor and quite innocent design flaws have a tendency to get magnified out of all proportion because APIs are provided once, but are called many times. If a design flaw results in awkward or inefficient code, the resulting problems show up at every point the API is called. In addition, separate design flaws that in isolation are minor can interact with each other in surprisingly damaging ways and quickly lead to a huge amount of collateral damage.

BAD APIS ARE EASY

Before I go on, let me show you by example how seemingly innocuous design choices can have far-reaching

API Design Matters

ramifications. This example, which I came across in my day-to-day work, nicely illustrates the consequences of bad design. (Literally hundreds of similar examples can be found in virtually every platform; my intent is not to single out .NET in particular.)

Figure 1 shows the interface to the .NET socket Select() function in C#. The call accepts three lists of sockets that are to be monitored: a list of sockets to check for readability, a list of sockets to check for writeability, and a list of sockets to check for errors. A typical use of Select() is in servers that accept incoming requests from multiple clients; the server calls Select() in a loop and, in each iteration of the loop, deals with whatever sockets are ready before calling Select() again. This loop looks something like the one shown in figure 1.

The first observation is that Select() overwrites its arguments: the lists that are passed into the call are replaced with lists that contain only those sockets that are ready. As a rule, however, the set of sockets to be monitored changes only rarely, and the most common case is that the server passes the same lists in each iteration. Because Select() overwrites its arguments, the caller must make a copy of each list before passing it to Select(). This is inconvenient and does not scale well: servers frequently need to monitor hundreds of sockets so, on each iteration, the code has to copy the lists before calling Select(). The cost of doing this is considerable.

A second observation is that, almost always, the list of sockets to monitor for errors is simply the union of the sockets to monitor for reading and writing. (It is rare that the caller wants to monitor a socket only for error conditions, but not for readability or writeability.) If a server monitors 100 sockets each for reading and writing, it ends up copying 300 list elements on each iteration: 100 each for the read, write, and error lists. If the sockets monitored for reading are not the same as the ones monitored for writing, but overlap for some sockets, constructing the

FIG 1

The .NET socket Select() function in C#

```
public static void Select(IList checkRead, IList checkWrite,
                          IList checkError, int microseconds);
{

//Server code
int timeout = ...;
ArrayList readList = ...; // Sockets to monitor for reading.
ArrayList writeList = ...; // Sockets to monitor for writing.
ArrayList errorList; // Sockets to monitor for errors.
while(!done)
{
   SocketList readTmp = readList.Clone();
   SocketList writeTmp = writeList.Clone();
   SocketList errorTmp = readList.Clone();
   Select(readTmp, writeTmp, errorTmp, timeout);
   for(int i = 0; i < readTmp.Count; ++i) {
      // Deal with each socket that is ready for reading...
   }
   for(int i = 0; i < writeTmp.Count; ++i) {
      // Deal with each socket that is ready for writing...
   }
   for(int i = 0; i < errorTmp.Count; ++i) {
       // Deal with each socket that encountered an error...
   }
   if(readTmp.Count == 0 &&
      writeTmp.Count == 0 &&
      errorTmp.Count == 0) {
       // No sockets are ready...
   }
}
```

error list gets harder because of the need to avoid placing the same socket more than once on the error list (or even more inefficient, if such duplicates are accepted).

Yet another observation is that Select() accepts a time-out value in microseconds: if no socket becomes ready within the specified time-out, Select() returns. Note, however, that the function has a void return type—that is, it does not indicate on return whether any sockets are ready. To determine whether any sockets are ready, the caller must test the length of all three lists; no socket is ready only if all three lists have zero length. If the caller happens to be interested in this case, it has to write a rather awkward test. Worse, Select() clobbers the caller's arguments if it times out and no socket is ready: the caller needs to make a copy of the three lists on each iteration even if nothing happens!

The documentation for Select() in .NET 1.1 states this about the time-out parameter: "The time to wait for a response, in microseconds." It offers no further explanation of the meaning of this parameter. Of course, the question immediately arises, "How do I wait indefinitely?" Seeing that .NET Select() is just a thin wrapper around the underlying Win32 API, the caller is likely to assume that a negative time-out value indicates that Select() should wait forever. A quick experiment, however, confirms that any time-out value that is equal to or less than zero is taken to mean "return immediately if no socket is ready." (This problem has been fixed in the .NET 2.0 version of Select().) To wait "forever," the best thing the caller can do is pass Int.MaxValue (2^{31}-1). That turns out to be a little over 35 minutes, which is nowhere near "forever." Moreover, how should Select() be used if a time-out is required that is not infinite, but longer than 35 minutes?

When I first came across this problem, I thought, "Well, this is unfortunate, but not a big deal. I'll simply write a wrapper for Select() that transparently restarts the call if it times out after 35 minutes. Then I change all calls to Select() in the code to call that wrapper instead."

FIG 2

The doSelect() function

```
public void doSelect(IList checkRead, IList checkWrite,
                     IList checkError, int milliseconds)
{
  ArrayList readCopy;   // Copies of the three parameters because
  ArrayList writeCopy;  // Select() clobbers them.
  ArrayList errorCopy;

  if (milliseconds <= 0) {
    // Simulate waiting forever.
    do {
      // Make copy of the three lists here...

      Select(readCopy, writeCopy, errorCopy, Int32.MaxValue);
    } while ((readCopy == null || readCopy.Count == 0) &&
             (writeCopy == null || writeCopy.Count == 0) &&
             (errorCopy == null || errorCopy.Count == 0));
  } else {
    // Deal with non-infinite timouts.
    while ((milliseconds > Int32.MaxValue / 1000) &&
           readCopy == null || readCopy.Count == 0) &&
           writeCopy == null || writeCopy.Count == 0) &&
           errorCopy == null || errorCopy.Count == 0)) {

      // Make a copy of the three lists here...

      Select(readCopy, writeCopy, errorCopy,
             (Int32.MaxValue / 1000) * 1000);
      milliseconds -= Int32.MaxValue / 1000;
    }
    if ((readCopy == null || readCopy.Count == 0) &&
       (writeCopy == null || writeCopy.Count == 0) &&
       (errorCopy == null || errorCopy == 0)) {
      Select(checkRead, checkWrite, checkError, milliseconds * 1000);
    }
  }
  // Copy the three lists back into the original parameters here...
}
```

API Design Matters

So, let's take a look at creating this drop-in replacement, called doSelect(), shown in figure 2. The signature (prototype) of the call is the same as for the normal Select(), but we want to ensure that negative time-out values cause it to wait forever and that it is possible to wait for more than 35 minutes. Using a granularity of milliseconds for the time-out allows a time-out of a little more than 24 days, which I will assume is sufficient.

Note the terminating condition of the do-loop in the code in figure 2: it is necessary to check the length of all three lists because Select() does not indicate whether it returned because of a time-out or because a socket is ready. Moreover, if the caller is not interested in using one or two of the three lists, it can pass either null or an empty list. This forces the code to use the awkward test to control the loop because, when Select() returns, one or two of the three lists may be null (if the caller passed null) or may be not null, but empty.

The problem here is that there are two legal parameter values for one and the same thing: both null and an empty list indicate that the caller is not interested in monitoring one of the passed lists. In itself, this is not a big deal but, if I want to reuse Select() as in the preceding code, it turns out to be rather inconvenient.

The second part of the code, which deals with restarting Select() for time-outs greater than 35 minutes, also gets rather complex, both because of the awkward test needed to detect whether a time-out has indeed occurred and because of the need to deal with the case in which milliseconds * 1000 does not divide Int.MaxValue without leaving a remainder.

We are not finished yet: the preceding code still contains comments in place of copying the input parameters and copying the results back into those parameters. One would think that this is easy: simply call a Clone() method, as one would do in Java. Unlike Java, however, .NET's type Object (which is the ultimate base type of all types) does not provide a Clone method; instead, for a type to be cloneable, it must explicitly derive from an ICloneable interface. The formal parameter type of the lists passed to Select() is IList, which is an interface and, therefore, abstract: I cannot instantiate things of type IList, only things derived from IList. The problem is that IList does *not* derive from ICloneable, so there is no convenient way to copy an IList, except by explicitly iterating over the list contents and doing the job element by element. Similarly, there is no method on IList that would allow it to be easily overwritten with the contents of another list (which is necessary to copy the results back into the parameters before doSelect() returns). Again, the only way to achieve this is to iterate and copy the elements one at a time.

Another problem with Select() is that it accepts *lists* of sockets. Lists allow the same socket to appear more than once in each list, but doing so doesn't make sense:

conceptually, what is passed are *sets* of sockets. So, why does Select() use lists? The answer is simple: the .NET collection classes do not include a set abstraction. Using IList to model a set is unfortunate: it creates a semantic problem because lists allow duplicates. (The behavior of Select() in the presence of duplicates is anybody's guess because it is not documented; checking against the actual behavior of the implementation is not all that useful because, in the absence of documentation, the behavior can change without warning.) Using IList to model a set is also detrimental in other ways: when a connection closes, the server must remove the corresponding socket from its lists. Doing so requires the server either to perform a linear search (which does not scale well) or to maintain the lists in sorted order so it can use a split search (which is more work). This is a good example of how design flaws have a tendency to spread and cause collateral damage: an oversight in one API causes grief in an unrelated API.

I will spare you the details of how to complete the wrapper code. Suffice it to say that the supposedly simple wrapper I set out to write, by the time I had added parameter copying, error handling, and a few comments, ran to nearly 100 lines of fairly complex code. All this because of a few seemingly minor design flaws:

- Select() overwrites its arguments.
- Select() does not provide a simple indicator that would allow the caller to distinguish a return because of a time-out from a return because a socket is ready.
- Select() does not allow a time-out longer than 35 minutes.
- Select() uses lists instead of sets of sockets.

Here is what Select() could look like instead:

```
public static int
Select(ISet checkRead, ISet checkWrite,
        Timespan seconds,
        out ISet readable, out ISet writeable,
        out ISet error);
```

With this version, the caller provides sets to monitor sockets for reading and writing, but no error set: sockets in both the read set and the write set are automatically monitored for errors. The time-out is provided as a Timespan (a type provided by .NET) that has resolution down to 100 nanoseconds, a range of more than 10 million days, and can be negative (or null) to cover the "wait forever" case. Instead of overwriting its arguments, this version returns the sockets that are ready for reading, writing, and have encountered an error as separate sets, and it returns the number of sockets that are ready or zero, in which case the call returned because the time-out was reached. With this simple change, the usability problems disappear and, because the caller no longer needs to copy the arguments, the code is far more efficient as well.

There are many other ways to fix the problems with Select() (such as the approach used by epoll()). The point of this example is not to come up with the ultimate version of Select(), but to demonstrate how a small number of minor oversights can quickly add up to create code that is messy, hard to maintain, error prone, and inefficient. With a slightly better interface to Select(), none of the code I outlined here would be necessary, and I (and probably many other programmers) would have saved considerable time and effort.

THE COST OF POOR APIS

The consequences of poor API design are numerous and serious. Poor APIs are difficult to program with and often require additional code to be written, as in the preceding example. If nothing else, this additional code makes programs larger and less efficient because each line of unnecessary code increases working set size and reduces CPU cache hits. Moreover, as in the preceding example, poor design can lead to inherently inefficient code by forcing unnecessary data copies. (Another popular design flaw—namely, throwing exceptions for expected outcomes—also causes inefficiencies because catching and handling exceptions is almost always slower than testing a return value.)

The effects of poor APIs, however, go far beyond inefficient code: poor APIs are harder to understand and more difficult to work with than good ones. In other words, programmers take longer to write code against poor APIs than against good ones, so poor APIs directly lead to increased development cost. Poor APIs often require not only extra code, but also more complex code that provides more places where bugs can hide. The cost is increased testing effort and increased likelihood for bugs to go undetected until the software is deployed in the field, when the cost of fixing bugs is highest.

Much of software development is about creating abstractions, and APIs are the visible interfaces to these abstractions. Abstractions reduce complexity because they throw away irrelevant detail and retain only the information that is necessary for a particular job. Abstractions do not exist in isolation; rather, we layer abstractions on top of each other. Application code calls higher-level libraries that, in turn, are often implemented by calling on the services provided by lower-level libraries that, in turn, call on the services provided by the system call interface of an operating system. This hierarchy of abstraction layers is an immensely powerful and useful concept. Without it, software as we know it could not exist because programmers would be completely overwhelmed by complexity.

The lower in the abstraction hierarchy an API defect occurs, the more serious are the consequences. If I mis-design a function in my own code, the only person affected is me, because I am the only caller of the function. If I mis-design a function in one of our project libraries, potentially all of my colleagues suffer. If I mis-design a function in a widely published library, potentially tens of thousands of programmers suffer.

Of course, end users also suffer. The suffering can take many forms, but the cumulative cost is invariably high. For example, if Microsoft Word contains a bug that causes

API Design Matters

it to crash occasionally because of a mis-designed API, thousands or hundreds of thousands of end users lose valuable time. Similarly, consider the numerous security holes in countless applications and system software that, ultimately, are caused by unsafe I/O and string manipulation functions in the standard C library (such as `scanf()` and `strcpy()`). The effects of these poorly designed APIs are still with us more than 30 years after they were created, and the cumulative cost of the design defects easily runs to many billions of dollars.

Perversely, layering of abstractions is often used to trivialize the impact of a bad API: "It doesn't matter—we can just write a wrapper to hide the problems." This argument could not be more wrong because it ignores the cost of doing so. First, even the most efficient wrapper adds some cost in terms of memory and execution speed (and wrappers are often far from efficient). Second, for a widely used API, the wrapper will be written thousands of times, whereas getting the API right in the first place needs to be done only once. Third, more often than not, the wrapper creates its own set of problems: the .NET `Select()` function is a wrapper around the underlying C function; the .NET version first fails to fix the poor interface of the original, and then adds its own share of problems by omitting the return value, getting the time-out wrong, and passing lists instead of sets. So, while creating a wrapper can help to make bad APIs more usable, that does not mean that bad APIs do not matter: two wrongs don't make a right, and unnecessary wrappers just lead to bloatware.

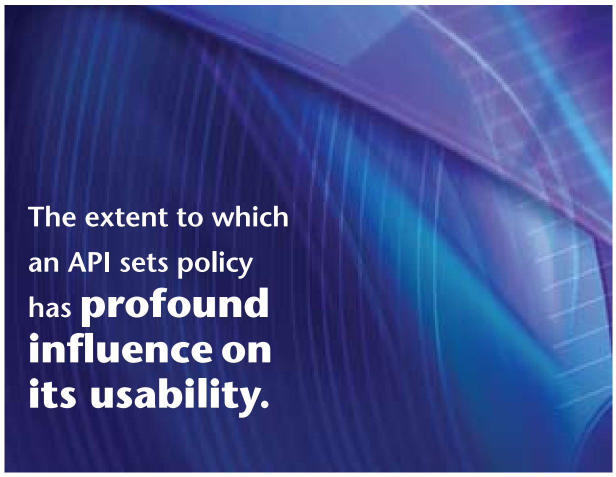
HOW TO DO BETTER

There are a few guidelines to use when designing an API. These are not surefire ways to guarantee success, but being aware of these guidelines and taking them explicitly into account during design makes it much more likely that the result will turn out to be usable. The list is necessarily incomplete—doing the topic justice would require a large book. Nevertheless, here are a few of my favorite things to think about when creating an API.

An API must provide sufficient functionality for the caller to achieve its task. This seems obvious: an API that provides insufficient functionality is not complete. As illustrated by the inability of `Select()` to wait more than 35 minutes, however, such insufficiency can go undetected. It pays to go through a checklist of functionality during the design and ask, "Have I missed anything?"

An API should be minimal, without imposing undue inconvenience on the caller. This guideline simply says "smaller is better." The fewer types, functions, and parameters an API uses, the easier it is to learn, remember, and use correctly. This minimalism is important. Many APIs end up as a kitchen sink of convenience functions that can be composed of other, more fundamental functions. (The C++ standard string class with its more than 100 member functions is an example. After many years of programming in C++, I still find myself unable to use standard strings for anything nontrivial without consulting the manual.) The qualification of this guideline, *without imposing undue inconvenience on the caller*, is important because it draws attention to real-world use cases. To design an API well, the designer must have an understanding of the environment in which the API will be used and design to that environment. Whether or not to provide a nonfundamental convenience function depends on how often the designer anticipates that function will be needed. If the function will be used frequently, it is worth adding; if it is used only occasionally, the added complexity is unlikely to be worth the rare gain in convenience.

The Unix kernel violates this guideline with `wait()`, `waitpid()`, `wait3()`, and `wait4()`. The `wait4()` function is


The extent to which an API sets policy has **profound influence on its usability.**

sufficient because it can be used to implement the functionality of the first three. There is also waitid(), which could almost, but not quite, be implemented in terms of wait4(). The caller has to read the documentation for all five functions in order to work out which one to use. It would be simpler and easier for the caller to have a single combined function instead. This is also an example of how concerns about backward compatibility erode APIs over time: the API accumulates crud that, eventually, does more damage than the good it ever did by remaining backward compatible. (And the sordid history of stumbling design remains for all the world to see.)

APIs cannot be designed without an understanding of their context. Consider a class that provides access to a set of name–value pairs of strings, such as environment variables:

```
class NVPairs {
    public string lookup(string name);
    // ...
}
```

The lookup method provides access to the value stored by the named variable. Obviously, if a variable with the given name is set, the function returns its value. How should the function behave if the variable is not set? There are several options:

- Throw a VariableNotSet exception.
- Return null.
- Return the empty string.

Throwing an exception is appropriate if the designer anticipates that looking for a variable that isn't there is not a common case and likely to indicate something that the caller would treat as an error. If so, throwing an exception is exactly the right thing because exceptions force the caller to deal with the error. On the other hand, the caller may look up a variable and, if it is not set, substitute a default value. If so, throwing an exception is exactly the wrong thing because handling an exception breaks the normal flow of control and is more difficult than testing for a null or empty return value.

Assuming that we decide not to throw an exception if a variable is not set, two obvious choices indicate that a lookup failed: return null or the empty string. Which one is correct? Again, the answer depends on the anticipated use cases. Returning null allows the caller to distinguish a variable that is not set at all from a variable that is set to the empty string, whereas returning the empty string for variables that are not set makes it impossible to distinguish a variable that was never set from a variable that was explicitly set to the empty string. Returning null is necessary if it is deemed important to be able to make this distinction; but, if the distinction is not important, it is better to return the empty string and never return null.

General-purpose APIs should be "policy-free;" special-purpose APIs should be "policy-rich." In the preceding guideline, I mentioned that correct design of an API depends on its context. This leads to a more fundamental design issue—namely, that APIs inevitably dictate policy: an API performs optimally only if the caller's use of the API is in agreement with the designer's anticipated use cases. Conversely, the designer of an API cannot help but dictate to the caller a particular set of semantics and a particular style of programming. It is important for designers to be aware of this: the extent to which an API sets policy has profound influence on its usability.

If little is known about the context in which an API is going to be used, the designer has little choice but to keep all options open and allow the API to be as widely applicable as possible. In the preceding lookup example, this calls for returning null for variables that are not set, because that choice allows the caller to layer its own policy on top of the API; with a few extra lines of code, the caller can treat lookup of a nonexistent variable as a hard error, substitute a default value, or treat unset and empty variables as equivalent. This generality, however, comes at a price for those callers who do not need the flexibility because it makes it harder for the caller to treat lookup of a nonexistent variable as an error.

This design tension is present in almost every API—the line between what should and should not be an error is very fine, and placing the line incorrectly quickly causes major pain. The more that is known about the context of an API, the more "fascist" the API can become—that is, the more policy it can set. Doing so is doing a favor to the caller because it catches errors that otherwise would go undetected. With careful design of types and parameters, errors can often be caught at compile time instead of being delayed until runtime. Making the effort to do this is worthwhile because every error caught at compile time is one less bug that can incur extra cost during testing or in the field.

The Select() API fails this guideline because, by overwriting its arguments, it sets a policy that is in direct conflict with the most common use case. Similarly, the .NET Receive() API commits this crime for nonblocking sockets: it throws an exception if the call worked but no data is ready, and it returns zero without an exception if the connection is lost. This is the precise opposite of what the caller needs, and it is sobering to look at the mess of control flow this causes for the caller.

Sometimes, the design tension cannot be resolved despite the best efforts of the designer. This is often the case when little can be known about context because an API is low-level or must, by its nature, work in many different contexts (as is the case for general-purpose collection classes, for example). In this case, the strategy pattern can often be used to good effect. It allows the caller to supply a policy (for example, in the form of a caller-provided comparison function that is used to maintain ordered collections) and so keeps the design open. Depending on the programming language, caller-provided policies can be implemented with callbacks, virtual functions, delegates, or template parameters (among others). If the API provides sensible defaults, such externalized policies can lead to more flexibility without compromising usability and clarity. (Be careful, though, not to "pass the buck," as described later in this article.)

APIs should be designed from the perspective of the caller. When a programmer is given the job of creating an API, he or she is usually immediately in problem-solving mode: What data structures and algorithms do I need for the job, and what input and output parameters are necessary to get it done? It's all downhill from there: the implementer is focused on solving the problem, and the concerns of the caller are quickly forgotten. Here is a typical example of this:

```
makeTV(false, true);
```

This evidently is a function call that creates a TV. But what is the meaning of the parameters? Compare with the following:

```
makeTV(Color, FlatScreen);
```

The second version is much more readable *to the caller*: even without reading the manual, it is obvious that the call creates a color flat-screen TV. To the implementer, however, the first version is just as usable:

```
void makeTV(bool isBlackAndWhite,
            bool isFlatScreen)
{ /* ... */ }
```

The implementer gets nicely named variables that indicate whether the TV is black and white or color, and whether it has a flat screen or a conventional one, but that information is lost to the caller. The second version requires the implementer to do more work—namely, to add enum definitions and change the function signature:

```
enum ColorType { Color, BlackAndWhite };
enum ScreenType { CRT, FlatScreen };
void makeTV(ColorType col, ScreenType st);
```

This alternative definition requires the implementer to think about the problem in terms of the caller. However, the implementer is preoccupied with getting the TV created, so there is little room in the implementer's mind for worrying about somebody else's problems.

A great way to get usable APIs is to let the *customer* (namely, the caller) write the function signature, and to give that signature to a programmer to implement. This step alone eliminates at least half of poor APIs: too often, the implementers of APIs never use their own creations, with disastrous consequences for usability. Moreover, an API is not about programming, data structures, or algorithms—an API is a *user* interface, just as much as a GUI is. The user at the using end of the API is a programmer—that is, a human being. Even though we tend to think of APIs as machine interfaces, they are not: they are *human*–machine interfaces.

What should drive the design of APIs is not the needs of the implementer. After all, the implementer needs to implement the API only once, but the callers of the API need to call it hundreds or thousands of times. This means that good APIs are designed with the needs of the caller in mind, even if that makes the implementer's job more complicated.

Good APIs don't pass the buck. There are many ways to "pass the buck" when designing an API. A favorite way is to be afraid of setting policy: "Well, the caller might want to do this or that, and I can't be sure which, so I'll make it configurable." The typical outcome of this approach is functions that take five or ten parameters. Because the designer does not have the spine to set policy and be clear about what the API should and should not do, the API ends up with far more complexity than necessary. This approach also violates minimalism and the principle of "I should not pay for what I don't use": if a function has ten parameters, five of which are irrelevant for the majority of use cases, callers pay the price of supplying ten parameters every time they make a call, even when they could not care less about the functionality provided by the extra five parameters. A good API is clear about what it wants to achieve and what it does not want to achieve, and is not afraid to be up-front about it. The resulting simplicity usually amply repays the minor loss of functionality, especially if the API has well-chosen fundamental operations that can easily be composed into more complex ones.

Another way of passing the buck is to sacrifice usability on the altar of efficiency. For example, the CORBA C++ mapping requires callers to fastidiously keep track of memory allocation and deallocation responsibilities; the result is an API that makes it incredibly easy to corrupt memory. When benchmarking the mapping, it turns out to be quite fast because it avoids many memory allocations and deallocations. The performance gain, however, is an illusion because, instead of the API doing the dirty work, it makes the caller responsible for doing the dirty work—overall, the same number of memory allocations takes place regardless. In other words, a safer API could be provided with zero runtime overhead. By benchmarking only the work done inside the API (instead of the overall work done by both caller and API), the designers can claim to have created a better-performing API, even though the performance advantage is due only to selective accounting.

The original C version of select() exhibits the same approach:

```
int select(int nfds, fd_set *readfds,
           fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

Like the .NET version, the C version also overwrites its arguments. This again reflects the needs of the implementer rather than the caller: it is easier and more efficient to clobber the arguments than to allocate separate output arrays of file descriptors, and it avoids the problems of how to deallocate the output arrays again. All this really does, however, is shift the burden from implementer to caller—at a net efficiency gain of zero.

The Unix kernel also is not without blemish and passes the buck occasionally: many a programmer has cursed the decision to allow some system calls to be interrupted, forcing programmers to deal explicitly with EINTR and restart interrupted system calls manually, instead of having the kernel do this transparently.

Passing the buck can take many different forms, the details of which vary greatly from API to API. The key questions for the designer are: Is there anything I could reasonably do for the caller I am not doing? If so, do I have valid reasons for not doing it? Explicitly asking these questions makes design the result of a conscious process and discourages "design by accident."

APIs should be documented before they are implemented. A big problem with API documentation is that it is usually written after the API is implemented, and often written by the implementer. The implementer, however, is mentally contaminated by the implementation and will have a tendency simply to write down what he or she has done. This often leads to incomplete documentation because the implementer is too familiar with the API and assumes that some things are obvious when they are not. Worse, it often leads to APIs that miss important use cases entirely. On the other hand, if the caller (not the implementer) writes the documentation, the caller can approach the problem from a "this is what I need" perspective, unburdened by implementation concerns. This makes it more likely that the API addresses the needs of the caller and prevents many design flaws from arising in the first place.

Of course, the caller may ask for something that turns out to be unreasonable from an implementation perspective. Caller and implementer can then iterate over the design until they reach agreement. That way, neither caller nor implementation concerns are neglected.

Once documented and implemented, the API should be tried out by someone unfamiliar with it. Initially, that person should check how much of the API can be understood without looking at the documentation. If an API can be used without documentation, chances are that it is good: a self-documenting API is the best kind of API there is. While test driving the API and its documentation, the user is likely to ask important "what if" questions: What if the third parameter is null? Is that legal? What if I want to wait indefinitely for a socket to become ready? Can I do that? These questions often pinpoint design flaws, and a cross-check with the documentation will confirm whether the questions have answers and whether the answers are reasonable.

Make sure that documentation is *complete*, particularly with respect to error behavior. The behavior of an API when things go wrong is as much a part of the formal

API Design Matters

contract as when things go right. Does the documentation say whether the API maintains the strong exception guarantee? Does it detail the state of out and in-out parameters in case of an error? Does it detail any side effects that may linger after an error has occurred? Does it provide enough information for the caller to make sense of an error? (Throwing a `DidntWork` exception from all socket operations just doesn't cut it!) Programmers *do* need to know how an API behaves when something goes wrong, and they *do* need to get detailed error information they can process programmatically. (Human-readable error messages are nice for diagnostics and debugging, but not nice if they are the only things available—there is nothing worse than having to write a parser for error strings just so I can control the flow of my program.)

Unit and system testing also have an impact on APIs because they can expose things that no one thought of earlier. Test results can help improve the documentation and, therefore, the API. (Yes, the documentation *is* part of the API.)

The worst person to write documentation is the implementer, and the worst time to write documentation is after implementation. Doing so greatly increases the chance that interface, implementation, and documentation will *all* have problems.

Good APIs are ergonomic. Ergonomics is a major field of study in its own right, and probably one of the hardest parts of API design to pin down. Much has been written about this topic in the form of style guides that define naming conventions, code layout, documentation style, and so on. Beyond mere style issues though, achieving good ergonomics is hard because it raises complex cognitive and psychological issues. Programmers are humans and are not created with cookie cutters, so an API that seems fine to one programmer can be perceived as only so-so by another.

Especially for large and complex APIs, a major part of ergonomics relates to consistency. For example, an API is easier to use if its functions always place parameters of a particular type in the same order. Similarly, APIs are easier to use if they establish naming themes that group related functions together with a particular naming style. The same is true for APIs that establish simple and uniform conventions for related tasks and that use uniform error handling.

Consistency is important because not only does it make things easier to use and memorize, but it also enables transference of learning: having learned a part of an API, the caller also has learned much of the remainder of the API and so experiences minimal friction. Transference is important not only within APIs but also across APIs—the more concepts APIs can adopt from each other, the easier it becomes to master all of them. (The Unix standard I/O library violates this idea in a number of places. For example, the `read()` and `write()` system calls place the file descriptor first, but the standard library I/O calls, such as `fgets()` and `fputs()`, place the stream pointer last, except for `fscanf()` and `fprintf()`, which place it first. This lack of parallelism is jarring to many people.)

Good ergonomics and getting an API to "feel" right require a lot of expertise because the designer has to juggle numerous and often conflicting demands. Finding the correct tradeoff among these demands is the hallmark of good design.

API CHANGE REQUIRES CULTURAL CHANGE

I am convinced that it is possible to do better when it comes to API design. Apart from the nitty-gritty technical issues, I believe that we need to address a number of cultural issues to get on top of the API problem. What we need is not only technical wisdom, but also a change in the way we teach and practice software engineering.

EDUCATION

Back in the late '70s and early '80s, when I was cutting my teeth as a programmer and getting my degree, much of the emphasis in a budding programmer's education was on data structures and algorithms. They were the bread and butter of programming, and a good understanding of data structures such as lists, balanced trees, and hash tables was essential, as was a good understanding of common algorithms and their performance tradeoffs. These were also the days when system libraries provided only the most basic functions, such as simple I/O and string manipulation; higher-level functions such as `bsearch()` and `qsort()` were the exception rather than the rule. This meant that it was *de rigueur* for a competent programmer to know how to write various data structures and manipulate them efficiently.

We have moved on considerably since then. Virtually every major development platform today comes with libraries full of pre-canned data structures and algorithms. In fact, these days if I catch a programmer writing a linked list, that person had better have a very good reason for doing so instead of using an implementation provided by a system library.

Similarly, in the '70s and '80s, if I wanted to create software, I had to write pretty much everything from scratch: if I needed encryption, I wrote it from scratch; if I needed compression, I wrote it from scratch; if I needed inter-process communication, I wrote it from scratch. All this has changed dramatically with the open source movement. Today, open source is available for almost every imaginable kind of reusable functionality. As a result, the process of creating software has changed considerably: instead of *creating* functionality, much of today's software engineering is about *integrating* existing functionality or about repackaging it in some way. To put it differently: API design today is much more important than it was 20 years ago, not only because we are designing more APIs, but also because these APIs tend to provide access to much richer and more complex functionality than they used to.

Looking at the curriculum of many universities, it seems that this shift in emphasis has gone largely unnoticed. In my days as an undergraduate, no one ever bothered to explain how to decide whether something should be a return value or an out parameter, how to choose between raising an exception and returning an error code, or how to decide if it might be appropriate for a function to modify its arguments. Little seems to have changed since then: my son, who is currently working toward a software engineering degree at the same university where I earned my degree, tells me that still no one bothers to explain these things. Little wonder then that we see so many poorly designed APIs: it is not reasonable to expect programmers to be good at something they have never been taught.

Yet, good API design, even though complex, is something that *can* be taught. If undergraduates can learn how to write hash tables, they can also learn when it is appropriate to throw an exception as opposed to returning an error code, and they can learn to distinguish a poor API from a good one. What is needed is recognition of the importance of the topic; much of the research and wisdom are available already—all we need to do is pass them on.

CAREER PATH

I am 47, and I write code. Looking around me, I realize how unusual this is: in my company, all of my programming colleagues are younger than I and, when I look at former programming colleagues, most of them no longer write code; instead, they have moved on to different positions (such as project manager) or have left the industry entirely. I see this trend everywhere in the software industry: older programmers are rare, quite often because no career path exists for them beyond a certain point. I recall how much effort it took me to resist a forced "promotion" into a management position at a former company—I ended up staying a programmer, but was told that future pay increases were pretty much out of the question if I was unwilling to move into management.

There is also a belief that older programmers "lose the edge" and don't cut it anymore. That belief is mistaken, in my opinion: older programmers may not burn as much midnight oil as younger ones, but that's not because they are old, but because they get the job done without having to stay up past midnight.

This loss of older programmers is unfortunate, particularly when it comes to API design. While good API design can be learned, there is no substitute for experience. Many good APIs were created by programmers who had to suffer under a bad one and then decided to redo the job, but properly this time. It takes time and a healthy dose of "once burned, twice shy" to gather the expertise that is necessary to do better. Unfortunately, the industry trend is to promote precisely its most experienced people away from programming, just when they could put their accumulated expertise to good use.

Another trend is for companies to promote their best programmers to designer or system architect. Typically, these programmers are farmed out to various projects as consultants, with the aim of ensuring that the project takes off on the right track and avoids mistakes it might make without the wisdom of the consultants. The intent of this practice is laudable, but the outcome is usually sobering: because the consultants are so valuable, having given their advice, they are moved to the next project long before implementation is finished, let alone testing and delivery. By the time the consultants have moved on, any problems with their earlier sage advice are no longer their problems, but the problems of a project they have long since left behind. In other words, the consultants never get to live through the consequences of their own design decisions, which is a perfect way to breed them into incompetence. The way to keep designers sharp and honest is to make them eat their own dog food. Any pro-

API Design Matters

cess that deprives designers of that feedback is ultimately doomed to failure.

EXTERNAL CONTROLS

Years ago, I was working on a large development project that, for contractual reasons, was forced into an operating-system upgrade during a critical phase shortly before a delivery deadline. After the upgrade, the previously working system started behaving strangely and occasionally produced random and inexplicable failures. The process of tracking down the problem took nearly two days, during which a large team of programmers was mostly twiddling its thumbs. Ultimately, the cause turned out to be a change in the behavior of awk's index() function. Once we identified the problem, the fix was trivial—we simply installed the previous version of awk. The point is that a minor change in the semantics of a minor part of an API had cost the project thousands of dollars, and the change was the result of a side effect of a programmer fixing an unrelated bug.

This anecdote hints at a problem we will increasingly have to face in the future. With the ever-growing importance of computing, there are APIs whose correct functioning is important almost beyond description. For example, consider the importance of APIs such as the Unix system call interface, the C library, Win32, or OpenSSL. Any change in interface or semantics of these APIs incurs an enormous economic cost and can introduce vulnerabilities. It is irresponsible to allow a single company (let alone a single developer) to make changes to such critical APIs without external controls.

As an analogy, a building contractor cannot simply try out a new concrete mixture to see how well it performs. To use a new concrete mixture, a lengthy testing and approval process must be followed, and failure to follow that process incurs criminal penalties. At least for mission-critical APIs, a similar process is necessary, as a matter of self-defense: if a substantial fraction of the world's economy depends on the safety and correct functioning of certain APIs, it stands to reason that any changes to these APIs should be carefully monitored.

Whether such controls should take the form of legislation and criminal penalties is debatable. Legislation would likely introduce an entirely new set of problems, such as stifling innovation and making software more expensive. (The ongoing legal battle between Microsoft and the European Union is a case in point.) I see a real danger of just such a scenario occurring. Up to now, we have been lucky, and the damage caused by malware such as worms has been relatively minor. We won't be lucky forever: the first worm to exploit an API flaw to wipe out more than 10 percent of the world's PCs would cause economic and human damage on such a scale that legislators *would* be kicked into action. If that were to happen, we would likely swap one set of problems for another one that is worse.

What are the alternatives to legislation? The open source community has shown the way for many years: open peer review of APIs and implementations has proven an extremely effective way to ferret out design flaws, inefficiencies, and security holes. This process avoids the problems associated with legislation, catches many flaws before an API is widely used, and makes it more likely that, when a zero-day defect is discovered, it is fixed and a patch distributed promptly.

In the future, we will likely see a combination of both tighter legislative controls and more open peer review. Finding the right balance between the two is crucial to the future of computing and our economy. API design truly matters—but we had better realize it before events run away with things and remove any choice. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

MICHI HENNING (michi@zeroc.com) is chief scientist of ZeroC. From 1995 to 2002, he worked on CORBA as a member of the Object Management Group's architecture board and as an ORB implementer, consultant, and trainer. With Steve Vinoski, he wrote *Advanced CORBA Programming with C++* (Addison-Wesley, 1999). Since joining ZeroC, he has worked on the design and implementation of Ice, ZeroC's next-generation middleware, and in 2003 co-authored "Distributed Programming with Ice." He holds an honors degree in computer science from the University of Queensland, Australia.