Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

The Most Important Design Guideline?

Scott Meyers

The activity of "design" includes many things, but certainly one of the most important aspects is interface specification. Interfaces determine which aspects of a component are accessible and to whom; they thus determine encapsulation. Interfaces specify what functionality (data, properties, methods, and so forth) is available to clients. Interfaces reflect how a system is broken down into its constituent components.

Interfaces are everywhere. They're the "I" in GUI and API, but they're much more pervasive than that. Classes and structs have interfaces; functions and methods have interfaces; templates and namespaces have interfaces; subsystems and modules have interfaces; libraries and applications have interfaces. Regardless of your role in the development of a software system, it almost certainly involves some interface design, so it's helpful to have some heuristics that indicate when you're doing it well—or poorly. Over time, I've come to the conclusion that the most important general interface design guideline is this:

> *Make interfaces easy to use correctly and hard to use incorrectly.*

This guideline leads to a conclusion that some developers find unsettling.

Interface designers must take responsibility

Let's make the reasonable assumption that your clients—the people using your interfaces—are trying to do a good job. They're smart, they're motivated, they're conscientious. They're willing to read some documentation to help them understand the system they're using. They *want* things to behave correctly.

That being the case, if they make a mistake when using your interface, *it's your fault*. We're assuming they're doing their best—they *want* to succeed. If they fail, it's because you let them. So, if somebody uses your interface incorrectly, either they're working hard at it (less likely) or your interface allowed them to do something easy that was not correct (more likely). This puts the shoe on the foot not used to wearing it: it means that responsibility for interface usage errors belongs to the *interface designer*, not the interface user.

In a perfect world, adherence to this guideline would all but guarantee correct program behavior. In such a world, programs that wouldn't do the right thing wouldn't compile, and programs that compiled would almost certainly do the right thing. At the human-computer interface level, commands that wouldn't do the right thing would be rejected, and commands that were accepted would almost certainly do the right thing. Alas, our world isn't perfect, but the interfaces used in most software systems can be significantly improved with relatively little effort.

 0740-7459/04/$20.00 © 2004 IEEE

```
struct Day { int d; };        // thin wrappers for Day and Year
struct Year { int y; };

class Month {
public:
  static const Month Jan;     // a fixed set of immutable
  static const Month Feb;     // Month objects
  ...
  static const Month Dec;
private:
  explicit Month(int);
};

class Date {                                        // revised (safer,
public:                                             // more flexible)
  explicit Date(Day d, Month m, Year y);            // interface
  explicit Date(Month m, Day d, Year y);
  explicit Date(Year y, Month m, Day d);
  ...
};
```

Figure 1. Design for date specification that's easy to use correctly and hard to use incorrectly.

Improving your interfaces

Consider a (C++) class for representing dates in time and how its constructor might be declared:

```
class Date {
public:
  explicit Date(int month,
                int day,
                int year);
};
```

This is a classic example of an interface that's easy to use incorrectly. Because all three parameters are the same type, callers can easily mix up the order, an error that's especially likely given that different cultures have different ordering conventions for a date's month, day, and year. Furthermore, the interface allows for nonsense data to be passed in, for example, a value of –29 for a month.

Creating separate types for days, months, and years can eliminate the ordering errors, and creating a fixed set of immutable `Month` objects can essentially eliminate the possibility of specifying invalid months. See Figure 1 for an example of this approach.

Figure 1 points out two important aspects to designing interfaces that obey the guideline. First, interface designers must train themselves to try to imagine all (reasonable) ways in which their interfaces could be used incorrectly. Second, they must find ways to prevent such errors from occurring.

Perhaps the most widely applicable approach to preventing errors is to define new types for use in the interface, in this case, `Day`, `Month`, and `Year`. It's best if such types exhibit the usual characteristics of good type design, including proper encapsulation and well-designed interfaces, but this example demonstrates that even introducing thin wrappers such as `Day` and `Year` can prevent some kinds of errors in date specification.

Responsibility for interface usage errors belongs to the interface designer, not the interface user.

A second commonly useful approach to preventing errors is to eliminate the possibility of clients creating invalid values. This approach applies when we know the universe of possible values in advance. In the date-specification example we just saw, I know that there are only 12 valid months, so I created a `Month` class with a private constructor, thus preventing the creation of `Month` values other than the 12 specific constant objects offered by the class. An alternative means to a similar end would be to use an `enum`, but, at least in C++, `enums` are less type-safe than classes, because the line between `enums` and `ints` isn't as distinct as we might wish.

In addition to introducing new types to the revised `Date` interface, I also added new constructors to the design. The `Day`, `Month`, and `Year` types make the interface harder to use incorrectly, but without the `Date` constructor overloads, the result is also harder to use correctly. Good interfaces support as many forms of correct use as possible while simultaneously thwarting as many incorrect forms as possible. Both efforts are necessary. One without the other won't suffice.

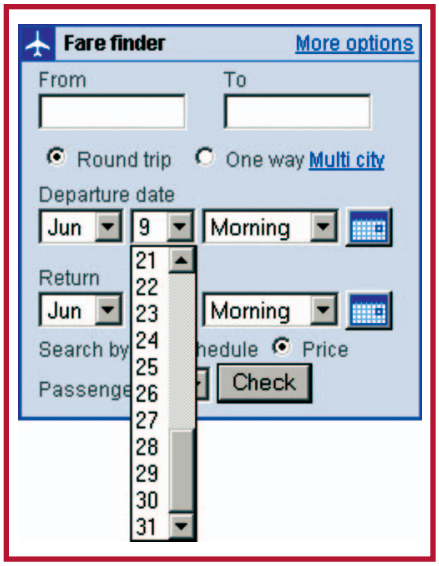

Figure 2. Drop-down box allowing specification of an invalid date.

Forcing users of an interface to choose from a set of guaranteed-valid choices is often good design, but it's not a panacea. Consider Figure 2, which shows how drop-down boxes for day, month, and year at the United Airlines' Web site still allow users to specify an invalid date (such as June 31). This is an example of an interface that appears to conform to the guideline, but doesn't, because it lulls the user into a feeling that mistakes are impossible. That is, it's easy to use incorrectly.

Note also that restricting users to choosing from a set of guaranteed-valid choices doesn't necessarily guarantee that the resulting data will be correct. The most constrained GUI drop-down boxes (or class constructors) in the world can't keep me from specifying August 27 when what I really meant was July 27.

In fact, forcing users to specify information via this kind of interface might actually *increase* the chances of specifying invalid data. Many GUI forms (in both applications and at Web sites) use drop-down boxes for specifying a state, for example, and my experience has been that I inadvertently specify the wrong state *much* more frequently than I mistype my state's two-letter abbreviation. If my experience is at all typical (and anecdotal evidence suggests that it is), that indicates that a drop-down box for this information is inferior to a simple text input box when considering which interface is easier to use correctly and harder to use incorrectly. It's important not to lose sight of this goal lest we confuse means and ends. The goal is an interface that's easy to use correctly and hard to use incorrectly. An approach that's often helpful in achieving this is to restrict the available input values, but sometimes that approach can be counterproductive.

Another example of an easy-to-misuse interface is one where a function returns a resource that the caller is responsible for releasing. Even languages with garbage collection exhibit this problem, because memory isn't the only resource. Consider the example in Figure 3. Here, the interface presented by the `getResource` method is a resource leak waiting to happen. All it takes is a client who forgets to call `release` when they are supposed to. The C++ approach to this problem would be to put the resource-releasing code (possibly as part of a reference-counting scheme) in `Resource`'s destructor. Callers of `getResource` could then forget about resource management, because it would be automatic.

```
class Resource {
public:
  Resource();
  void release();

  static                          // caller must call
    Resource getResource();       // release on the
                                  // returned object
};
```

Figure 3. Another example of an easy-to-misuse interface, where the function returns a resource that the caller is responsible for releasing.

Unfortunately, languages such as Java and the .NET languages don't offer destructors or their equivalent, and the idioms that address resource issues such as this (`finally` or `using` blocks, for example) put the onus on clients to remember to use the idioms. But interfaces that rely on clients remembering to do something are easy to use incorrectly.

In situations like this, good interface designers fall back on simple encapsulation: if something is tricky or error-prone and there's no way to get around it, they hide the tricky or error-prone code as much as possible, so as few people as possible have to deal with it. For example, `getResource` might be declared `private` or `protected` so that the easy-to-use-incorrectly interface is accessible to relatively few clients. In addition, `Resource` might be outfitted with debugging capabilities so that situations in which objects that are leaked or that have unusually long lifetimes (suggesting an overly late call to `release`) are easy to identify.

Adhering to the guideline that interfaces should be easy to use correctly and hard to use incorrectly leads to systems that are both more usable and more likely to be used correctly. That's why it's the most important general design guideline in my arsenal. To employ it, designers need to train themselves to anticipate what clients might reasonably like to do, and then facilitate that activity. They also must anticipate what clients might incorrectly do, and prevent that activity. Above all, it requires remembering that when an interface is used incorrectly, the fault is that of the interface designer, not the interface user.

Scott Meyers is an independent consultant on software development. He authored *Effective C++, More Effective C++, and Effective STL*, is consulting editor for Addison-Wesley's Effective Software Development Series, and serves on advisory boards for *Software Development* magazine and several start-up companies. Contact him at smeyers@aristeia.com; www.aristeia.com.