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

# Separating User Interface Code

Martin Fowler

The first program I wrote on a salary was scientific calculation software in Fortran. As I was writing, I noticed that the code running the primitive menu system differed in style from the code carrying out the calculations. So I separated the routines for these tasks, which paid off when I was asked to create higher-level tasks that did several of the individual menu steps. I could just write a routine that called the calculation routines directly without involving the menus.

Thus, I learned for myself a design principle that's served me well in software development: Keep your user interface code separate from everything else. It's a simple rule, embodied into more than one application framework, but it's often not followed, which causes quite a bit of trouble.

## Stating the separation

Any code that does anything with a user interface should only involve user interface code. It can take input from the user and display information, but it should not manipulate the information other than to format it for display. A clearly separated piece of code—separate routines, modules, or classes (based on your language's organizing structure)—should do calculations, validations, or communications should be done by a clearly separated piece of code. For the rest of the article, I'll refer to the user interface code as *presentation code* and the other code as *domain code*.

When separating the presentation from the domain, make sure that no part of the domain code makes any reference to the presentation code. So, if you write an application with a WIMP (windows, icons, mouse, and pointer) GUI, you should be able to write a command line interface that does everything that you can do through the WIMP interface—without copying any code from the WIMP into the command line.

## Why do this?

Following this principle leads to several good results. First, this presentation code separates the code into different areas of complexity. Any successful presentation requires a fair bit of programming, and the complexity inherent in that presentation differs in style from the domain with which you work. Often it uses libraries that are only relevant to that presentation. A clear separation lets you concentrate on each aspect of the problem separately—and one complicated thing at a time is enough. It also lets different people work on the separate pieces, which is useful when people want to hone more specialized skills.

Making the domain independent of the presentation also lets you support multiple presentations on the same domain code, as suggested by the WIMP versus command line example, and also by writing a higher-level Fortran routine. Multiple presentations

0740-7459/01/$10.00 © 2001 IEEE

are the reality of software. Domain code is usually easy to port from platform to platform, but presentation code is more tightly coupled to operating system or hardware. Even without porting, it's common to find that demands to changes in the presentation occur with a different rhythm than changes in the domain functionality.

Pulling away the domain code also makes it easier to spot—and avoid—duplication in domain code. Different screens often require similar validation logic, but when it's hidden among all the screen handling, it's difficult to spot. I remember a case where an application needed to change its date validation. The application had two parts that used different languages. One part had date validation copied over its date widgets and required over 100 edits. The other did its validation in a single date class and required just a single change. At the time, this was hyped as part of the massive productivity gain you could get with object-oriented software—but the former non-object system could have received the same benefit by having a single date validation routine. The separation yielded the benefit.

Presentations, particularly WIMPs and browser-based presentations, can be very difficult to test. While tools exist that capture mouse clicks, the resulting macros are very tricky to maintain. Driving tests through direct calls to routines is far easier. Separating the domain code makes it much easier to test. Testability is often ignored as a criteria for good design, but a hard-to-test application becomes very difficult to modify.

## The difficulties

So why don't programmers separate their code? Much of the reason lies in tools that make it hard to maintain the separation. In addition, the examples for those tools don't reveal the price for ignoring the separation.

In the last decade or so, the biggest presentation tool has been the family of platforms for developing WIMP interfaces: Visual Basic, Delphi, Powerbuilder, and the like. These tools were designed for putting WIMP interfaces onto SQL databases and were very successful. The key to their success was data-aware widgets, such as a pop-up menu bound to a SQL query. Such tools are very powerful, letting you quickly build a WIMP interface that operates on a database, but the tools don't provide any place to extract the domain code. If straight updates to data and view are all you do, then this is not a problem. Even as a certified object-bigot, I always recommend these kinds of tools for these kinds of applications. However, once domain logic gets complicated, it becomes hard to see how to separate it.

This problem became particularly obvious as the industry moved to Web interfaces. If the domain logic is stuck inside a WIMP interface, it's not possible to use it from a Web browser.

However, the Web interfaces often encourage the same problems. Now we have server page technologies that let you embed code into HTML. As a way of laying out how generated information appears on the page, this makes plenty of sense. The structure starts breaking down when the code inside the server page is more complicated. As soon as code starts making calculations, running queries, or doing validations, it runs into that same trap of mixing presentation with domain code. To avoid this, make a separate module that contains the domain code and only make simple calls from the server page to that module. For a simple set of pages, there is an overhead (although I would call it a small one), but as the set gets more complicated, the value of the separation grows.

This same principle, of course, is at the heart of using XML. I built my Web site, www.martinfowler.com, by writing XML and converting it to HTML. It lets me concentrate on the structure of what I was writing in one place, so I could think about its formatting later (not that I do any fancy formatting.) Those who use content-oriented styles in word processors are doing much the same thing. I've reached the point where that kind of separation seems natural. I've had to become a bit of an XSLT whiz—and the tools for that aren't even adolescent yet.

The general principle here is that of separating concerns, but I find such a general principle hard to explain and follow. After all, what concerns should you separate? Presentation and domain are two separable concerns I've found straightforward to explain—although that principle isn't always easy to follow. I think it's a key principle in well-engineered software. If we ever see engineering codes for software, I'd bet that separation of presentation and domain will be in there somewhere.

**Pulling away the domain code also makes it easier to spot—and avoid—duplication in domain code.**

**Martin Fowler** is the chief scientist for ThoughtWorks, an Internet systems delivery and consulting company. For a decade, he was an independent consultant pioneering the use of objects in developing business information systems. He's worked with technologies including Smalltalk, C++, object and relational databases, and Enterprise Java with domains including leasing, payroll, derivatives trading, and health care. He is particularly known for his work in patterns, UML, lightweight methodologies, and refactoring. He has written four books: *Analysis Patterns, Refactoring, Planning Extreme Programming,* and *UML Distilled.* Contact him at fowler@acm.org.