Beyond the Black Box: Open Implementation

Enapsulation, informally known as black-box abstraction, is a widely known and accepted principle. However, many practitioners find themselves violating it in order to achieve performance requirements in a practical manner. This gap between theory and practice must be filled. Open implementation is a controversial new approach that claims to do just that. The ideas that follow will spark further debate on black-box abstraction (a debate that is taking place, in part, on the Internet). I welcome your responses on this column and the subject at large.

— Tomoo Matsubara

SOFTWARE HAS TRADITIONALLY BEEN constructed according to the principle that a module should expose its functionality but hide its implementation. This principle, informally known as black-box abstraction, is a basic tenet of software design, underlying our approaches to portability, reuse, and many other important issues in computing.

Although black-box abstraction has many attractive qualities, exposing only the functionality of a module can lead to serious performance difficulties during reuse. These problems have led a new community of researchers to reconsider the question of what a module should expose to clients. The process revealed that a module can be more useful and ultimately more reusable — if it allows clients to control its implementation strategy. The new design principle is called open implementation.

A number of cutting-edge tools can be seen as open implementations, including programming languages, distributed computing systems, databases, GUI toolkits, object-oriented frameworks, and even simple data structures. The open-implementation principle can help us to better understand those systems, as well as to design future systems that better suit client needs.

ABSTRACTION IN ENGINEERING. The fundamental issue in engineering is controlling complexity. The systems we build are so complex that we cannot hope to comprehend their full complexity at any one time. Instead, as engineers, we use abstraction and decomposition, which allow us to selectively focus our attention, yet retain an appropriate understanding of the whole system.

But abstraction and decomposition are very general notions that do not explicitly say how to decompose and how to abstract. What parts should we break a system into? What should we abstract away or keep explicit at the interface to each part? We need more focused principles to guide system decomposition and interface design. Black-box abstraction is one such principle. As Figure 1 shows, it helps us decompose a system into functional modules, with interfaces that abstract away implementation and keep only functionality explicit.

The promise of the black-box principle is to simplify client code by ensuring that it can't be "caught up in the implementation details" of underlying modules. It also enables reusability by decreasing the dependence between modules and their clients.

WHEN THE CLIENT KNOWS BEST. The reason black-box abstraction doesn't always work is that in some cases, the best implementation strategy...

Continued on p. 10

Figure 1. A black-box abstraction presents a single interface — denoted by the thick blue line — that exposes functionality but hides implementation.
Continued from p. 8

for a module can't be determined unless the implementor knows just how a module will be used. In other words, the client often knows best how the module should be implemented. Black-box abstraction forces the implementor to decide early on what the implementation will be and then locks that decision into the black box. This results in conflicts when the implementor makes a choice that a client can't tolerate. These conflicts become more likely as more clients try to use a module. That is why the open-implementation principle is so critical to extensive reusability.

For example, consider a window system and one particular client: the display portion of a spreadsheet application. According to the black-box principle, the window system's interface would expose the functionality of windowing (sharing of the screen), display, mouse-tracking, and so on. It would hide the data structures it uses to store the window system's state, how mouse tracking is implemented, and so on.

Continuing with the black-box principle, it should be easy to implement the spreadsheet on top of a clean, powerful window system. The spreadsheet's needs are simple: a rectangular array of boxes in which to display and in which the user can click the mouse. This is exactly the functionality a window system provides, so the simplest way to code the spreadsheet would be to use one window for each cell. This approach takes advantage of the window system's black-box interface to cleanly express what is desired, and it makes maximal reuse of the existing window-system implementation. Figure 2 shows the resulting program.

This is the quintessence of the black-box principle. The code is simple and clear, and it can be read without having to know anything about the underlying implementation.

Yet few experienced programmers would be surprised to learn that this implementation doesn't quite work. Or rather, it might work, but its performance may be so bad as to render it useless. Why? Because the window-system implementation may not be tuned for this kind of use. When writing the window system, the implementor is faced with a number of performance tradeoffs. If, for example, it is assumed that 25 to 50 windows is a typical number for an application to use, instead of 10,000, a heavy-weight window-representation strategy would likely be used. Once that strategy is locked into the window system's black box, the spreadsheet developer can't use the window system in this simple way.

Client programmers who are confronted with such conflicts end up having to "code around" the problem. This commonly takes one of two forms, hematomas of duplication and coding between the lines, illustrated in Figure 3. In the spreadsheet case, the programmer would likely end up writing their own "little window system" that could draw boxes on the screen, display in them, and handle mouse events—all with the proper performance tradeoffs. This is a hematoma of duplication.

In other cases, the client programmer can write code in a contorted way to get better performance. A classic example is in the use of virtual memory. In a program that allocates a number of objects, there is often a "natural" way to order that allocation. But when there are so many objects that paging behavior becomes critical, client programmers often rewrite the application to allocate the objects in a different order. This has the indirect effect of controlling their layout on physical pages and thereby improving performance. The client programmer is coding between the lines to control an issue that the interface claimed to hide.

**OPEN IMPLEMENTATION.** The analysis of a number of domains in these terms—including programming languages, distributed computing, databases, GUI toolkits, object-oriented frameworks, and simple data structures—leads to the conclusion that:

*It is impossible to hide all implementation issues behind a module interface. Some of these issues are crucial implementation-strategy decisions that will inevitably bias the performance of the resulting implementation. Module implementations must somehow be opened up to allow clients control over these issues as well.*

The question remains: How can we give clients control
over implementation-strategy decisions in an elegant way, without causing more problems than we're trying to solve? Put another way, how can we fix the problems of the black-box principle without giving up what is right about it?

In the window-system case, an elegant approach might take the form of pragmas. In the code below, the TiledParent pragma tells the window system to use an implementation strategy that is biased towards a large number of tiled subwindows.

```plaintext
main := mkwindow(root, 0, 0, 1000, 1000) (TiledParent)
for i = 1 to 100
    for j = 1 to 100
        mkwindow(main, 10, 10, i*10, j*10)
end
end
```

**SEPARATION OF CONTROL.** This example demonstrates an essential property of well-designed open implementations. As Figure 4 shows, the design clearly separates the primary interface through which the client requests windowing functionality and the secondary (but coupled) meta-interface through which the client tunes the implementation underlying the primary interface. When reading this code, the programmer can focus on the basic functionality and ignore the tuning — in this case, by simply ignoring the TiledParent pragma.

The goal of an open implementation is to allow the client programmer to:
- use the module’s primary functionality alone when the default implementation is adequate;
- control the module’s implementation-strategy decisions when necessary; and
- deal with functionality and implementation strategy decisions in largely separate ways.

Achieving this kind of separation has been the focus of much open-implementation research. A primary focus has been the concept of computational reflection, which explores how modules can provide interfaces for examining and adjusting themselves — that is, how modules can provide meta-interfaces. Although this may sound somewhat esoteric, it is exactly what open implementations need — an interface for controlling the implementation strategy that sits behind the primary interface.

Open implementation is a growing area of interest, and there are many issues to explore. Issues currently getting active attention include:
- How to design appropriately abstract meta-interfaces that give clients control over implementation strategy decisions without drowning them in implementation details.
- How to decide what implementation strategy decisions to expose.
- Implementation technologies to support open implementation.
- Finding more examples of existing ad-hoc open implementations and studying them to learn more about the approach.

---

**Figure 4.** An open implementation presents two interfaces. The primary interface provides the functionality and the meta-interface allows the client to adjust the implementation strategy decisions that underlie the primary interface.

It isn't possible to discuss any of these here, but a longer version of this article, as well as pointers to other work and articles, is available at http://www.parc.xerox.com/oi. I invite you to visit those pages and join in the discussion and development of this new principle of software design.

Gregor Kiczales is head of the embedded-computation area at Xerox Palo Alto Research Center. His research is focused on fundamental issues in software architecture. He can be reached at gregor@parc.xerox.com.