Missing in Action: Information Hiding

Prospecting for programmer's gold.

REWARD for lost software-engineering concept. Responds to the name "information hiding." Last seen in Canada in the late 1970s. Sometimes answers to "encapsulation," "modularity," or "abstraction." If found, please call 555-HIDE.

INFORMATION HIDING IS ONE OF SOFTWARE engineering's seminal design ideas. So what's happened to it? Most of the structured and object-oriented design books I checked recently list "information hiding" in their indexes, but few give it more than a passing acknowledgment. This slight is akin to the response that Michael Stipe, leader of the rock group R.E.M., gave when asked to describe the Beatles' influence on his music. He said he doubted that he had ever listened to an entire Beatles album. They are irrelevant, he said, "elevator music."

As a musician and composer, Stipe has missed something by not listening to the Beatles. As software designers and implementors, some of us have missed something by not thoroughly acquainting ourselves with information hiding.

OUT OF THE DARK. Information hiding first came to public attention in David Parnas's 1972 paper, "On the Criteria to Be Used in Decomposing Systems Into Modules" (*Communications of the ACM*, Dec. 1972). Information hiding is characterized by the idea of "secrets" — design and implementation decisions that a software developer hides from the rest of a program. It is part of the foundation of both structured and object-oriented design. In structured design, information hiding produces "black boxes"; in object-oriented design, it gives rise to the concepts of encapsulation and modularity, and is associated with abstraction. However, information hiding doesn't require or depend on any particular design methodology, and you can use it with any design approach.

Frederick Brooks, in the 20th Anniversary edition of *The Mythical Man-Month* (Addison-Wesley, 1995), concludes that his criticism of information hiding was one of the few errors in the book's first edition: "Parnas was right, and I was wrong about information hiding," he proclaims. In 1987, Barry Boehm reported that information hiding was a powerful technique for eliminating rework and that it was particularly effective during software evolution ("Improving Software Productivity," *Computer*, Sept. 1987). As incremental, evolutionary development styles become more popular, the value of information hiding can only increase.

DESIGN SECRETS. Suppose you have a program in which each object is supposed to have a unique ID stored in a member variable called ID. One design approach would be to use integers for the IDs and store the highest ID assigned in a global variable called `MaxID`. In each place that a new object is allocated, perhaps in each object's constructor, you could simply use the statement `ID = ++MaxID`. (This is a C-language statement that increments the value of `MaxID` by 1 and assigns the new value to `ID`.) That would guarantee a unique ID, and it would add the absolute minimum of code in each place an object is created. What could go wrong with that?

Information hiding is characterized by the idea of "secrets."

A lot of things. What if you want to reserve ranges of IDs for special purposes? What if you want to reuse the IDs of objects that have been destroyed? What if you want to add an assertion that fires when you allocate more IDs than the maximum number you've anticipated? If you allocated IDs by spreading `ID = ++MaxID` statements throughout your program, you'd have to change the code associated with every one of those statements.

The way new IDs are created is a design decision that you should hide. If you use the phrase `++MaxID` throughout your program, you expose

Continued on page 127

Editor:
Steve McConnell
Phantom Lake Engineering
PO Box 6922
Bellevue, WA 98008
smcconn@aol.com

0740-7459/96/$05.00 © 1996 IEEE

Continued from page 128

the fact that a new ID is created by incrementing `MaxID`. If, instead, you put the statement `ID = NewID()` throughout your program, you hide the information about how new IDs are created. Inside the `NewID()` function, you might still have just one line of code — `return ( ++MaxID )` or its equivalent — but if you later decide to reserve certain ranges of IDs for special purposes or to reuse old IDs, you could make those changes within the `NewID()` function itself without touching dozens or hundreds of `ID = NewID()` statements. And, no matter how complicated the revisions inside `NewID()` might become, they wouldn't affect any other part of the program.

Now suppose you discover that you need to change the ID type from an integer to a string. If you've spread variable declarations like `int` ID throughout your program, your use of the `NewID()` function won't help. You'll still have to go through your program and make dozens or hundreds of changes.

In this case, the design decision to hide is the ID's type. You could simply declare your IDs to be of `IDTYPE`, a user-defined type that resolves to `int`, rather than directly declaring them to be of type int. Once again, hiding a design decision makes a huge difference in the amount of code affected by a change.

SPARE CHANGES. To use information hiding, you begin by listing the design secrets that you want to hide. As the example suggested, the most common kind of secret is a design decision that you think might change. You then separate each design secret by assigning it to its own class, subroutine, or other design unit. Next you isolate (encapsulate) each secret so that if it does change, the change doesn't affect the rest of the program.

Some of the design areas that are most likely to change are specific to individual projects, but you will run into others again and again, such as

- hardware dependencies for display screens, printers, plotters, communications devices, disk drives, tapes, sound, and so on;
- input and output formats, both machine and end-user readable;
- nonstandard language features and library routines;
- difficult design and implementation areas, especially areas that might be developed poorly and require redesign or reimplementation;
- complex data structures, data structures that are used by more than one class, or data structures you haven't designed to your satisfaction;

Asking what needs to be hidden supports good design decisions at all levels.

- complex logic, which is almost as likely to change as complex data structures;
- global variables, which are probably never truly needed, but which always benefit from being hidden behind access routines;
- data-size constraints such as array declarations and loop limits; and
- business rules such as the laws, regulations, policies, and procedures that are embedded into a computer system.

HEURISTIC VALUE. Aside from providing support for structured and object-oriented design, information hiding has a unique heuristic power: the ability to inspire effective design solutions.

Although object design provides the heuristic power of modeling the world in objects, in the example above, object thinking wouldn't help you avoid declaring the ID as an `int` instead of an `IDTYPE`. The object designer would ask, "Should an ID be treated as an object?" Depending on the project's coding standards, a "Yes" answer might mean that the designer has to create interface and implementation source-code files for the ID class; write a constructor, destructor, copy operator, and assignment operator; document it all; have it all reviewed; and place it under configuration control. Unless the designer is exceptionally motivated, he'll decide that creating a whole class just for an ID isn't worth it and will use `int`s instead.

Note what just happened. A useful design alternative — that of simply hiding the ID's data type — was not even considered. If, instead, the designer had asked, "What about the ID should be hidden?" he might well have decided to hide its type behind a simple type declaration that substitutes `IDTYPE` for `int`. The difference between object design and information hiding in this example is more subtle than a clash of explicit rules and regulations. Object design would approve of this design decision as much as information hiding would. Rather, the difference is one of heuristics: Thinking about information hiding inspires and promotes design decisions that thinking about objects does not.

WHAT TO HIDE? Information hiding can also be useful in designing a class's public interface. The gap between theory and practice in class design is wide. Among many class designers, the decision about what to put into a class's public interface amounts to deciding what interface would be the easiest to write code to — which usually results in exposing as much of the class as possible. From what I've seen, most programmers would rather expose all of a class's private data than write 10 extra lines of code to keep the secrets intact. Asking, "What does this class need to hide?" cuts to the heart of the interface-design issue. If you can put a function or data into the class's public interface without compromising its secrets, do. Otherwise, don't.

Asking what needs to be hidden supports good design decisions at all levels. It promotes the use of named constants instead of literals at the implementation level. It helps in creating good subroutine and parameter names inside classes. It guides decisions about class and subsystem decompositions and interconnections at the system level. Get into the habit of asking, "What should I hide?" You'll be surprised at how many difficult design decisions vanish before your eyes. ◆