Open Implementation Design Guidelines

Gregor Kiczales, John Lamping,
Cristina Videira Lopes,
Chris Maeda, Anurag Mendhekar
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304 U.S.A
gregor@parc.xerox.com

Gail Murphy
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver B.C. Canada V6T 1Z4
murphy@cs.ubc.ca

ABSTRACT
Designing reusable software modules can be extremely difficult. The design must be balanced between being general enough to address the needs of a wide range of clients and being focused enough to truly satisfy the requirements of each specific client. One area where it can be particularly difficult to strike this balance is in the implementation strategy of the module. The problem is that general-purpose implementation strategies, tuned for a wide range of clients, aren't necessarily optimal for each specific client—this is especially an issue for modules that are intended to be reusable and yet provide high-performance.

An examination of existing software systems shows that an increasingly important technique for handling this problem is to design the module's interface in such a way that the client can assist or participate in the selection of the module's implementation strategy. We call this approach *open implementation*.

When designing the interface to a module that allows its clients some control over its implementation strategy, it is important to retain, as much as possible, the advantages of traditional closed implementation modules. This paper explores issues in the design of interfaces to open implementation modules. We identify key design choices, and present guidelines for deciding which choices are likely to work best in particular situations.

Keywords
open implementation, software design, software reuse

INTRODUCTION
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.

Black-box abstraction has many attractive qualities—amortized development costs, localization of change, etc. Exposing only the functionality of a module in its interface, however, can sometimes lead to performance difficulties when the module gets reused. It has been observed that in such cases, clients "code around" the problem either by re-implementing an appropriate version of the module or by using existing modules in contorted ways [5, 6]. In either case, many of the goals that motivated creating the module in the first place are not actually realized.

Many recent systems address this problem by having modules that allow client control of their implementation strategy [7, 8, 9, 10, 11, 12,]. We say that these modules have open implementations.

The open implementation approach works by somewhat shifting the black-box guidelines for module design. Whereas black-box modules hide all aspects of their implementation, open implementation modules allow clients some control over selection of their implementation strategy, while still hiding many true details of their implementation. In doing this, open implementation module designs strive for an appropriate balance between preserving the kind of opacity black-box modules have, and providing the kind of performance tailorability some clients require.

A number of existing systems have open implementation style interfaces, but thus far, there has been no systematic study of open implementation design, and as a result, designers of these systems have had little or no general guidance to assist them. This paper addresses this need by examining a series of specific modules with open implementations, including designs taken from published systems and toy designs that illustrate specific issues. The designs serve to illustrate important concepts, guidelines, and tradeoffs. They also provide concrete instances to study and use as idioms in future designs.

This paper is specifically focused on the design of interfaces to modules with an open implementation. While the implementation techniques that support these interfaces are crucial, they are beyond the scope of this paper.[1] Neither does this paper focus on the general motivation for open implementation—that can be found in [13, 14, 6, 15, 16]—instead we operate from the premise that some modules can benefit from the open implementation approach, and focus on issues in the design of their interfaces.

A BASE CASE

Before we begin an exploration of open implementation interface designs, it is necessary to provide a basis for the terms *module* and *interface*. We use these terms in a similar fashion to [17] where a module represents a work assignment, and an interface is the set of assumptions a client programmer using the module may make about its behavior.[2] The modules subject to an open implementation are conceived in the same manner as any other module, namely by the application of the information hiding principle [18]. According to this principle, modules are selected to localize and hide design decisions.

The following interface design for a simple set module will be used as an illustrative example throughout the paper. This black-box interface presents only the functionality of the set module and hides all implementation issues behind the interface. It will serve as a comparison point for subsequent open implementation designs for interfaces to set modules. We are using the set module throughout to help make the differences between the designs more clear. But not all of the designs we present will be appropriate for a module as simple as this. These will noted explicitly.

[1] Many of the implementation techniques are straightforward, and will be apparent simply from looking at the interface design. Others are more subtle, and involve recently developed techniques in language and system implementation [1, 2, 4]. There is, as yet, no unified presentation of these techniques; a separate paper describing this is in preparation.

[2] In this paper, we are concerned with guidelines on the selection and form of the interface to an open implementation module. Issues related to the specification of an interface are outside the scope of this work.

Set Module Interface Design A

This is the simple "black-box" design. It has the usual procedures for creating sets, adding and removing elements from sets, and mapping over the elements of a set. The calling interface to the module might look something like:

```
makeSet()
insert(item, set)
delete(item, set)
isIn(item, set)
map(function, state, set)[3]
```

Interface design A is attractive in its simplicity. In addition, it adheres to the five characteristics of quality interface designs outlined in [19]. That is, the interface is *consistent* (e.g., the set parameter is consistently passed as the last argument), *essential* (e.g., each service is offered in only one way), *general* (e.g., a set may be used for only insertions, or both insertions and deletions), *minimal* (e.g., each function provides one operation), and *opaque* (e.g., the interface hides the "secret" around which the module has been defined).

It is, however, inherently difficult to develop an implementation of this interface that will please a large range of prospective clients. This difficulty arises because determining the best implementation strategy for a set depends on knowing what is going to be done with it. How many elements will it have? How often will new elements be inserted? Will existing elements be deleted? How often? How often will the other set operations be called? All of these factors are important in determining how to implement a set. This is why there are so many different implementation strategies for sets. The libg++ library [20], for example, has eleven variants of set, including linked lists, B-trees and hash tables, to name a few. But with design A, the set module implementor has little basis for selecting which implementation strategy to use—the interface makes it difficult for the set module to know what a specific client's usage pattern will be. This is, in short, an appropriate case for an open implementation design.

SEPARATION OF USE FROM IMPLEMENTATION STRATEGY CONTROL

The following design addresses the difficulty of developing a reusable implementation of design A by providing clients limited control over the selection of the module's implementation strategy.

Set Module Interface Design B

In this design, the interface is the same as in design A, except that now **`makeSet`** can optionally be called with an argument that describes the client's pattern of use. The intent is that the set

[3] The **`map`** procedure calls the function on every element of the set, passing it both the element and the state block. This design makes it possible to "simulate a closure."

module implementation can examine this description and select an appropriate specialized implementation strategy tuned for that pattern of use. The optional usage parameter is a string in a simple declarative language that supports the encoding of information such as the size of the set and the relative frequency with which the various operations are called.

```
makeSet(usage)
makeSet()
insert(item, set)
delete(item, set)
isIn(item, set)
map(function, state, set)
```

The following example calls to **makeSet** show how the usage parameter works:

```
makeSet("n=10000,
         insert=lo,
         delete=lo,
         isIn=hi")
makeSet("n=5,
         insert=hi,
         delete=hi")
```

In this design, the opacity criteria have been relaxed somewhat from design A. Whereas design A kept the implementation entirely "secret," design B admits to clients that selecting the implementation strategy is an important issue, and that understanding how the set will be used can help in that selection. But note that most of the secrets remain hidden. The client does not know what the actual implementation strategies are, and they certainly do not know any of the details about how those strategies are implemented.

We begin with a few simple observations about this new interface design:

- It is only a small change from interface design A. The **makeSet** procedure now accepts an optional argument; all the other procedures are unchanged.
- The client's use of the new functionality is optional. It is still possible to call **makeSet** with no arguments, which will leave the set module free to choose a default general-purpose implementation strategy, much as it would have in design A.
- The client's use of the new functionality has an inherently well-bounded effect. The implementation strategy control associated with a given call to **makeSet** affects only the sets created by that call. This makes it possible for some sets to use the new functionality and others not, and for different sets that use the new functionality to do so in different ways to get different implementation strategies.
- The new part of the interface can be seen as being relatively orthogonal to the original interface. The new part supports client control of implementation strategy, whereas the old part supports actually using sets.

The last observation means that set module interface design B effectively splits client code into two kinds: most of the client code simply uses the set module's functionality, while the parameter to **makeSet** is involved in controlling the set module's implementation strategy.

This important property is in fact the subject of the first design guideline—***open implementation module interfaces should support a clear separation between client code that uses the module's functionality (use code) and client code that controls the module's implementation strategy (ISC code) .***

A clear separation between client use code and ISC code is important because it helps to preserve the advantages of black-box modules. It helps the client programmer selectively focus their attention on either the way their code uses the module's functionality, or the way their code controls the module's implementation strategy. When focusing on the use code, the client programmer is effectively working with a black-box interface to the module.

Design B does a good job in this respect; the client programmer simply has to selectively ignore the paramater passed to **makeSet** in order to focus on use code. It would even be easy to build an automatic tool that could hide the ISC code when the programmer wanted to ignore it.

In working with this guideline, what is most important is the effective separation the client programmer has to work with, as manifested in their code. This goal can be supported by use/ISC separation in the interface, but it is separation in the client code that is the real benefit.

In addition to having a clear separation between client use and ISC code, ***open implementation module interfaces should be designed to make the ISC code optional, make the ISC code easy to disable, and support alternative ISC codes for one piece of use code.*** These additional guidelines provide further support for the development of clients of open implementation modules. They enable clients to first be developed with a focus on getting the functionality right, by leaving out ISC code. They assist performance debugging, by selectively turning parts of the ISC code on and off. They facilitate porting, by allowing different ISC code for different environments. They support division of expertise, since use code can be written by a person (or group) with one expertise and ISC code can later be written by a person (or group) with another expertise.

One example of a system with clear use/ISC code separation is High-Performance Fortran (HPF) [21], a Fortran extension intended to support efficient data parallel programming. One of HPF's principal components is a set of declarations that allows programmers to assist the compiler

(and the runtime system) in determining strategies for distributing arrays across multiple processors. In our terminology, these declarations are ISC code. Clear use/ISC separation is achieved by embedding the declarations into what would be comments in a Fortran-90 program. An example of the use of this mechanism is:

```
REAL A(1000,1000), B(998,998)
!HPF$ ALIGN B(I,J) WITH A(I+1,J+1)
```

where the first line is use code that declares two large arrays and the second line is ISC code saying how to lay out the elements of the arrays with respect to each other.

Scoring the HPF interface design against the use/ISC separation guidelines:

- The use/ISC code separation is clear—the ISC code can easily be ignored by the client programmer or hidden by a tool.
- The ISC code is optional—either HPF or Fortran-90 compilers will compile an HPF program without the ISC code.
- The ISC code is easy to disable—a very simple tool can strip it out of a program before passing that program on to the compiler.
- HPF doesn't directly support multiple ISC codes for one use code, but it is easy to build a tool that does do so, for example by further extending the syntax to mark each line of ISC code with the platform for which it is intended, and then using a pre-processor to strip out inappropriate lines before passing the code off to the HPF compiler.

These properties translate into direct benefits to HPF programmers. Programs can be developed focusing on just the use code. The ISC code can be added later during tuning, possibly by different programmers. Even after the ISC code has been added, the use code is internally complete and executable on its own, so that evolution can be accomplished by first adjusting and testing the use code, and then making any needed adjustments to the ISC code.

An example that doesn't do quite as good a job on use/ISC separation is the libg++ library [20], a large library of C++ classes and other building blocks, that includes a set module with an open implementation. But in this design, ISC code is mandatory at set construction, requiring client programmers to always think about the set module's implementation strategy, even in the many cases where a general-purpose strategy would be sufficient. The result is that too many of the benefits of the black-box interface are lost. This also means there is no way to tell from reading the client code whether a particular piece of ISC code was well thought out, or was merely intended to be a default. This makes the code harder to reason about and maintain.

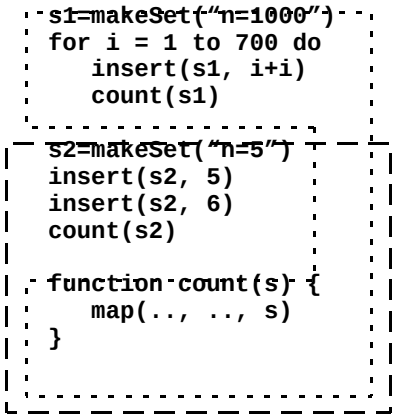

Figure 1: Scope control in Design B

The work described in [10] improves on the libg++ design in several ways, one of which is to provide a more clear use/ISC separation.

SCOPE CONTROL

An important observation about design B is that any given piece of ISC code affects the implementation of only some sets—just those sets created by the **`makeSet`** the ISC code appears in. This important point is the focus of the next design guideline—***open implementation module interfaces should be designed to allow the scope of influence of ISC code to be controlled in a way that is both natural and sufficiently fine-grained.***

Like use/ISC separation, the motivation for this guideline is to help the client programmer understand their program, in this case by making it easier for them to reason about the effect of the ISC code they write. The programmer's reasoning is directly facilitated when the scope of influence of ISC code is natural and fine-grained.

Design B does a good job of meeting this guideline. The ISC code on a specific call to **`makeSet`** affects only those sets returned from that call (and all the set operations on them). It is natural for the client programmer to think in terms of sets created by a given call to **`makeSet`**. This granularity is sufficiently fine grained for the programmer to reason easily about the effect of any piece of ISC code.

Figure 1 shows the effect of design B's scope control from the client programmer's perspective. It shows a number of lines of use code, and two pieces of ISC code, the strings **`"n=1000"`** and **`"n=5"`**. The dashed lines indicate what parts of the use code are in the scope of influence of each piece of ISC code. Note that the **`count`** function, and the call to **`map`** inside it are in both scopes, since it can be passed sets with either kind of implementation.

Choosing the Scope Control

While the importance of natural and fine-grained ISC code scope control is easy to state, designing an appropriate scope control for an interface can be a subtle problem.

Coming up with the design involves considering how and why the client is going to want to control the implementation strategy, and making sure that the design gives clients a fine-enough granularity to work with, without being overly difficult to implement or use. This section presents some alternative scope controls, to illustrate some of the considerations that come into play.

As an alternative scope control for design B, consider a design where the client could only control the implementation strategy on a per-application basis. This might be done with a declaration associated with the makefile for the application, that affected all the sets used by that application. This scope control would not be fine-enough grained, because it is reasonable to expect that an application will want to use sets more than once, and do so in different ways, and thus want different implementations strategies. This alternative design would thus be not much more useful than a closed implementation of sets.

As another example consider file systems that allow the client to control their pre-fetching and caching strategy [22]. These systems tend to provide this control on a per stream basis.[4] A per-file basis would be too coarse a granularity, because it would cause problems if two different clients opened the same file but wanted different implementation strategies. Similarly, ISC scope control on a per-process basis would be too coarse, since it is reasonable to expect that a system running in one process might want to open different streams with different implementation strategies.

While it is important to have sufficiently fine-grained scope control, there is a tension in that the more fine-grained it gets, the harder it can be both to use and to implement. For example, if a file system allowed the client to control the pre-fetching strategy on a per-byte basis—every call to **`readByte`** could control the pre-fetching that happened with that call—it would undoubtedly be more powerful than on a per-stream basis, but it could be more cumbersome to use and difficult to implement. (Implementation technology capable of supporting such a design does exist however [3].)

There are, however, cases where very coarse ISC scope control has proven useful. Consider for example the BLAS libraries [23] for matrix routines. There are different library implementations customized for different hardware architectures. The library is linked in when execution starts, and affects *all* the matrix arithmetic in the application, but in this case that is an appropriate granularity.

In summary, natural and fine-grained scope control complements clear use/ISC separation. A clear use/ISC separation divides the client code into use code and ISC code. Natural and fine-grained ISC code scope control partitions the client code into parts depending on what ISC code affects them.

SUBJECT MATTER

While design B does address the original need for client control of implementation strategy, the way in which it does so has a few potential weaknesses:

- If a client programmer mis-describes the behavior of their program they may wind up with an implementation strategy that is worse than the default.
- Even if the client programmer properly describes the behavior of their program, they have no guarantee that they will get an implementation strategy that is optimal for their purposes. An implementation of design B might not include an implementation strategy that is optimal for every usage profile a client might describe in a call to **`makeSet`**.

In essence, design B allows the client to say more about its behavior, but leaves the client unsure about the effect this will have on the module's implementation strategy. Addressing this uncertainty is the motivation for the next design.

Set Module Interface Design C

This design for the set module interface is identical to design B except for the optional argument to **`makeSet`**. In this design the client programmer has the option to explicitly specify one of a fixed list of implementation strategies for the new set. The fixed list is: **`BTree, LinkedList, HashTable`**.

```
makeSet(strategy)
makeSet()
insert(item, set)
delete(item, set)
isIn(item, set)
map(function, state, set)
```

Two example calls to **`makeSet`** are:

```
makeSet("LinkedList")

makeSet("HashTable")
```

First we note some of the ways that design C is similar to design B:

- It has similar use/ISC separation, i.e. a parameter of a procedure in the use interface.
- It has similar scope control, i.e. a given piece of ISC code affects only operations on sets returned by that call to **`makeSet`**.

But designs B and C differ in an important respect, having to do with the nature of the ISC code in clients of each. To capture this difference, we introduce a concept called the ***ISC code subject matter*** of an open implementation mod-

[4] By stream we mean the result of opening the file, that is a handle to the file that can be used to read/or write bytes.

Subject Matter	Client ISC Code	Example
client program's behavior	`n=10000,insert=hi,delete=lo,isIn=hi`	Design B
performance requirements the module must meet at its interface	`bandwith=10000`	Network Quality of Service [24]
module implementation strategy	`HashTable`	Design C

Table 1: Subject matter and Style of ISC Code

ule's interface design. We use this term to refer to the explicit subject of the ISC code.

In design B, the ISC code subject matter is the client program's behavior. In design C it is the module's implementation strategy. This distinction may appear somewhat subtle, since, after all, both designs allow the client to affect the module's implementation strategy. And pieces of ISC code from designs B and C can have the same intent, even though they have different subject matter, i.e. **`"n=1000, insert=lo, delete=lo, isIn=hi"`** and **`"HashTable"`**. The difference is in what the ISC code is *explicitly* about: the client program's behavior in design B vs. is the module's implementation strategy in design C.

There is a third important possibility for ISC code subject matter—performance requirements the module must meet at its interface. While this subject matter may not be appropriate for the interface to a set module, it is useful in other cases.[5] One example of open implementation modules with this ISC code subject matter is network protocol interfaces that allow clients to request a particular quality of service [24]. Such guarantees are critical for applications, such as audio- and video-conferencing, that send real-time data streams over a network.

The three possibilities for ISC code subject matter are summarized in Table 1.

Tradeoffs

Choosing the ISC code subject matter is a key decision in the design of the interface to an open implementation module. The ISC code subject matter has a significant effect on how easy the module will be to design, specify and implement, as well as how well it will work for its clients.

Making the ISC code subject matter be the client's behavior feels like it should be easier for the client programmer, since all they have to do is figure out the behavior of their program and let the module do the rest. But this isn't always the case. It can often be easier for a client programmer to simply name a well-known implementation strategy that they know will be appropriate. Further, this can give the client programmer more certainty that their ISC code will have the effect they desire. This is why the libg++ set library has module implementation strategy as its subject matter, not client program behavior. (It is more like design C than design B.)

On the other hand, having the ISC code subject matter be the module's implementation strategy opens the door to potential problems if the client programmer chooses an inappropriate strategy. We are all familiar with the fact that good C compilers ignore register declarations because programmers almost always use them incorrectly. So the interface designer should only make this choice for ISC code subject matter when there is a reasonable chance that the client programmer will be able to choose correctly.

And, while having the subject matter be the performance requirements at the interface seems like a happy compromise, it is not always the best choice either. There are many cases where it is easier for the client programmer to speak in terms of one of the other subject matters.

One rule of thumb for selecting ISC code subject matter is based on seeing the process of selecting implementation strategy as a series of analysis steps: Given the client use code, how does it use the interface? Given a client with that usage pattern, what performance properties does it require? Given those performance requirements, what implementation strategy will best satisfy them? This process is illustrated in Figure 2.

[5] The libg++ set library uses the module's implementation strategy as its ISC code subject matter. (It is like design C in that sense.) But, the documentation of the different strategies (**`XPSets`**, **`OXPSets`**, **`SLSets`** etc.) itself includes a description of each strategy's order of complexity (i.e. [a O(n)], [f O(n)], [d O(n)]… for **`XPSets`**), so it describes itself in terms of performance properties at the module's interface.

[6] If there is one implementation strategy that is appropriate for all clients, there is no need for an open implementation.

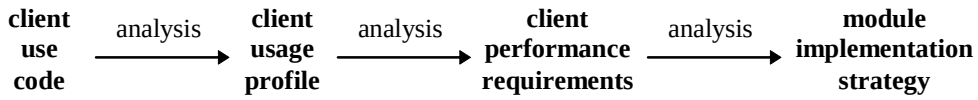

Figure 2: Analysis steps in the process of selecting implementation strategy

Seeing the process that way, the guideline is: ***Pick the first subject matter along the process of Figure 2 for which all of the following criteria hold:***

- ***It is possible to build an automatic mechanism that completes the chain of reasoning from that point onwards to get an optimal implementation strategy.***
- ***It is easy to design an interface to express the subject matter at that point.***
- ***It would be easy for the client programmer to use that interface to express that subject matter. This includes both figuring out what to say and how to say it.***

Note that this guideline also provides a way of knowing when not to use an open implementation. An open implementation is not needed when all of the steps of the above inference process can be handled automatically to arrive at an optimal implementation strategy.

One example of an appropriate choice of ISC code subject matter is the `inline` declaration found in many programming languages, including C and Common Lisp. This declaration allows the programmer to name an implementation strategy for procedure calling. It comes at the end of the inference process above, and so the programmer has a clear sense of what its effect will be.

A corresponding example of inappropriate choice of ISC code subject matter is the speed/space/safety declarations found in Common Lisp [25]. These declarations don't have a clear subject matter; it isn't clear where they fall in the inference process above, and programmers don't have a clear sense of what their effect will be.

Implementation Details Must be Hidden

Design C further relaxes the original secrets around which Design A was defined. Now, the existence of a fixed set of implementation strategies is no longer secret. But notice that the true details of each strategies implementation is still hidden. There is still plenty of information hiding across the interface between the client and the implementation. This can be stated in a design guideline: ***Open omplementation module interfaces should be designed to pass only essential implementation strategy information.*** The three subject matters are different ways of encoding the essential information.

STYLE OF THE ISC CODE

While design C addresses the lack of guarantees in design B, both designs are limited to whatever set of implementation strategies is provided by the module. This makes them both vulnerable to the implementation not being flexible enough for a wider range of clients. This motivates yet another design.

Set Module Interface Design D

In this design, the use interface is exactly the same as in design C. But this design not only allows client programmers to choose from a fixed set of default implementation strategy, but also allows them to provide entirely new implementation strategies for the set module. The client provides these strategies in the form of an entirely new implementation of the set functionality, packaged up as a subclass of the class **`Set`**. (In this paper we use the mechanism of object-oriented programming to capture this kind of design, but other mechanisms like callbacks or dispatching procedures could be used just as well.)

The following example illustrates the use of interface design D:

In use file

```
makeSet("mySet")
```

In ISC file

```
class mySet (Set) {
  method insert…
  method delete…
  method isIn…
  method map…}
```

Design D is similar to design C in many ways:

- It has the same scope control.
- It has similar use/ISC code separation. The key difference in design D is that client ISC code includes not only the code inside the arguments to **`makeSet`**, but also the code that defines any new implementation strategies for sets.
- The ISC code subject matter in this design is the implementation strategy of the module. But in this design, the ISC code takes two different forms. The part inside the arguments to **`makeSet`** is just like in design C, but the part that defines new subclasses of Set is different.

To capture this difference between the declarative ISC code in designs B and C and the programmatic ISC code that in design D, we introduce a new concept, the ***style of the ISC code***.

Declarative style ISC code is simple, but its power is limited to the forms of declarations supported by the interface. This limitation can be problematic when a client has needs that fall out of the purview of these declarations. An interface that supports *programmatic* ISC code addresses this limitation by allowing the client to write ISC code in the form of a small program.

In design D, the set primitives **`insert`**, **`delete`**, **`isIn`** etc. will invoke the client's programmatic ISC code when one of the client-defined implementations is requested. Errors in this ISC code will cause errors seen by the use code. So, unlike the situation in the earlier designs, ISC code has the potential of breaking the use functionality of the interface.

The programmatic style of interface thus can lead to less robust designs. For this reason, it should only be used in cases, such as this one, where otherwise the client would be forced to "code around" the performance deficiency of the module. The use of programmatic ISC code puts a premium on having the right scope control, so as to restrict the consequences of bad programmatic ISC code to those places where it is requested. So, for example, if a buggy backing store is given to the Mach external pager, the whole operating system does not come crashing down. Only the process requesting that backing store is affected.

THE DESIGN SPACE

Figure 4, on the next page, summarizes these four design approaches. It illustrates the progressively deeper involvement of the client in the implementation in the successive styles. The right style to use for a particular module is the one that lets client get as involved in implementation strategy as they need to, without having to get more involved.

Layering

Not only can different clients of a module need different implementation strategies, different clients of a module may also be better served by different interface design styles. Fortunately, this can be accommodated.

Notice that interface design D subsumes both design C and design A. That is, a client of design D has three choices regarding control of the set module's implementation strategy:

1. They can specify no ISC code and get the default implementation strategy.
2. They can choose from the list of the built-in strategies.
3. They can provide a new strategy.

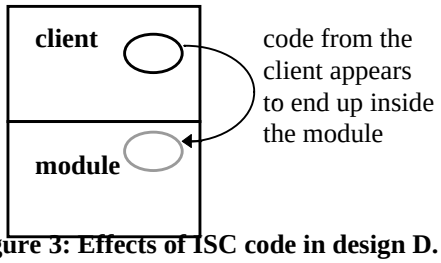

Figure 3: Effects of ISC code in design D.

We say that design D is a *layered* interface design.[7] In this design the client can get into the implementation strategy selection process at three different levels. In fact, the first two levels of the above layering have been implicitly present since design B, stemming from the guideline that ISC code should be optional.

Many existing open implementation modules have layering in this sense. The file system mentioned above is one example, that closely parallels design D. The client can do nothing, in which case they get a default pre-fetching policy, or they can choose from a small set of built-in policies, or they can write programmatic ISC code to define a new policy.

A layered interface design aims at exploiting a version of the 90/10 rule. The idea is that 90% of the clients can use the default strategy, the remaining 10% will need to write some ISC code. 90% of that 10% can select from among the built-in strategies, and only the final 1% (but probably a very important 1%) have to provide an entirely new strategy.

Layering is not an end in itself, but a technique to address what might otherwise seem like an irresolvable trade-off. In particular, layering is a way to design an interface that has the robustness and ease of use of declarative ISC code, while at the same time having the power of programmatic ISC code. The guideline is: ***When there is a simple interface that can describe strategies that will satisfy a significant fraction of clients, but it is impractical to accommodate all important strategies in that interface, then the interfaces should be layered.***

[7] Layered interface designs refer to the structure of the interface, not to the underlying software structure. A layered interface design might or might not be implemented by a layered software architecture.

Interface Style and *example*	How Strategy is Selected	Tradeoffs	When it is Appropriate
Style A – No implementation strategy control interface	Module selects implementation strategies by observing client's use of the Black-Box Interface.	Same as Black-Box Abstraction.	One implementation strategy will satisfy all clients. Or the module can determine a good strategy by itself.
Style B – Client provides declarative information about its usage pattern. *"sequential file scan."*	Module selects strategy by matching usage pattern information from client to the best available strategy.	Client provided information about its usage pattern doesn't constrain the implementation. Difficult for client to know how it is influencing module strategy.	It is easy to choose an effective implementation strategy if the client behavior is known.
Style C – Client specifies the implementation strategy the module should use. *"LRU cache management"*	Module adopts the strategy specified by client.	Easy to specify exact strategy. However, client might be uninformed or wrong about best strategy to use.	There are a few candidate implementation strategies, but it is difficult to choose among them automatically.
Style D – Client provides the implementation strategy to use. *an object that implements a custom strategy on top of the cache management protocol*	Module adopts the strategy provided by client.	Easy to specify exact strategy. However, designing module to support replaceable strategies might be difficult. For client, building a new strategy implementation might be expensive.	It is not feasible for the module to implement all implementation strategies that clients might need.

Figure 4: open implementation interface styles.

OTHER DESIGNS

The range of design approaches presented here are suitable for a large class of open implementations. But there is no room here to cover all the approaches. Two notable omissions are: an approach, particularly used in some open operating systems, that allows incremental definition of new strategies; approaches for allocating shared resources. These other approaches will be explored in future work.

CONCLUSION

Open implementation is appropriate for reusable modules that have clients with a wide range of different performance requirements. Open implementation is based on reworking the opacity guidelines for traditional black-box modules. In open implementation, modules allow their clients to participate in their implementation strategy, but still hide many aspects of their implementation details. Open implementation requires new design guidelines to augment the existing ones for black-box modules. This paper provides an initial set of such guidelines and issues having to do with:

- Clear use/ISC client code separation
- Natural and fine-grained ISC code scope control
- Selection of appropriate ISC code subject matter
- Selection of appropriate ISC code style
- Incrementality in the ISC interface
- Use of layering to balance ease of use and power

ACKNOWLEDGMENTS

We would like to thank the people who have contributed directly to this paper: Art Lee, Rob DeLine, John Irwin, Jean-Marc Loingtier, and Marvin Theimer.

BIBLIOGRAPHY

1. Kiczales, G., J.d. Riveres, and D.G. Bobrow, *The Art of the Metaobject Protocol*. 1991: MIT Press.
2. Chambers, C. and D. Ungar. *Making Pure Object-Oriented Languages Practical*. in *OOPSLA '91 Proceedings; SIGPLAN Notices*. 1991. Phoenix, AZ.
3. Pu, C. and H. Massalin, *An Overview of The Synthesis Operating System*. 1989: Columbia University.
4. Chiba, S. *A Metaobject Protocol for C++*. in *OOPSLA '95 Conference Proceedings Object-Oriented Programming Systems, Languages, and Applications*. 1995. Austin: ACM Press.

5. Stonebraker, M., *Operating System Support for Database Management.* Communications of the ACM, 1981. **24**(7): p. 412-418.

6. Kiczales, G. *Towards a New Model of Abstraction in Software Engineering*. in *Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*. 1992. Tokyo, Japan.

7. Young, M.W., *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.* Vol. Technical report CMU-CS-89-202. 1989: Carnegie Mellon University, Computer Science Department.

8. Hamilton, G. and P. Kougiouris, *The Spring Nucleus: A Microkernel for Objects*. 1993: Sun Microsystems Laboratories, Inc.

9. Yokote, Y. *The Apertos Reflective Operating System: The Concept and its Implementation*. in *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*. 1992.

10. Lortz, V.B. and K.G. Shin. *Combining Contracts and Exemplar-Based Programming for Class Hiding and Customization*. in *Object-Oriented Programming Systems, Languages, and Applications*. 1994. Portland, Oregon: ACM Press.

11. Maeda, C. and B.N. Bershad. *Service without Servers*. in *Fourth Workshop on Workstation Operating Systems*. 1993: IEEE Computer Society Technical Committee on Operating Systems and Application Environments, IEEE Computer Society Press.

12. Anderson, T.E. and others, *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, 1992. **10**(1): p. 53-79.

13. Shaw, M. and W.A. Wulf, *Towards Relaxing Assumptions in Languages and Their Implementations.* SIGPLAN Notices, 1980. **15**(3): p. 45-61.

14. Heninger Britton, K., R.A. Parker, and D.L. Parnas. *A Procedure for Designing Abstract Interfaces for Device Interface Modules*. in *5th International Conference on Software Engineering*. 1981: IEEE Computer Society Press.

15. Kiczales, G., *Beyond the Black Box: Open Implementation.* IEEE Software, 1996. **13**(1): p. 8--11.

16. *Open Implementation Home Page*, Xerox Palo AltoResearch Center, http://www.parc.xerox.com/oi.

17. Parnas, D.L. and P.C. Clements, *A Rational Design Process: How and Why to Fake It.* IEEE Transactions on Software Engineering and Methodology, 1986. **SE-12**(2): p. 251--257.

18. Parnas, D.L., *On the Criteria to be Used in Decomposing Systems into Modules.* Communications of the ACM, 1972. **15**(12): p. 1053-1058.

19. Hoffman, D., *On Criteria For Module Interfaces.* IEEE Transactions on Software Engineering and Methodology, 1990. **16**(5): p. 537--542.

20. Gnu, *Lib G++ Documentation*, http://www.delorie.com/gnu/docs.

21. Steele Jr., G.L., *High Performance Fortran: Status Report.* ACM SIGPlan Notices, 1993. **28**(1).

22. Patterson, R.H. and et al., *A Status Report on Research in Transparent Informed Prefetching*, in *ACM Operating Systems Review*. 1993. p. 21-34.

23. Dongarra, J.J., *et al.*, *An Extended Set of Fortran Basic Linear Algebra Subprograms.* ACM Transactions on Mathematical Software, 1988. **14**: p. 1--17.

24. Zhang, L., *et al.*, *RSVP: A New Resource ReSerVation Protocol.* IEEE Network, 1993(September).

25. Steele Jr., G.L., *Common Lisp the Language*. Second ed. 1990: Digital Press. 1029.