Open Implementation Analysis and Design

Chris Maeda
Xerox PARC
maeda@parc.xerox.com

Arthur Lee
Korea University
alee@psl.korea.ac.kr

Gail Murphy
University of British Columbia
murphy@cs.ubc.ca

Gregor Kiczales
Xerox PARC
gregor@parc.xerox.com

ABSTRACT

This paper describes a methodology for designing Open Implementations -- software modules that can adapt or change their internals to accommodate the needs of different clients. Analysis techniques are used for capturing domain knowledge, user requirements, and domain properties that influence the module's eventual implementation. Design techniques are used for determining and refining the interfaces by which clients control the modules implementation strategies. The methodology has evolved over the past two years in several pilot projects.

Keywords: Software Design Methodology, Software Reuse, Open Implementation, Framework, Metaobject Protocol

INTRODUCTION

This paper describes Open Implementation Analysis and Design (OIA/D), a method for designing and implementing reusable software modules. An Open Implementation (OI) of a software module exposes facets of its internal operation to client control in a principled way [Kic91, Kic96]. The key assumption behind Open Implementation is that software modules can be more reusable if they can be designed to accommodate a range of implementation strategies. Since no one implementation strategy is adequate for all clients, the module should support several implementation strategies and allow clients to help select the strategy actually used.

The Open Implementation approach has been applied in a wide variety of domains, such as operating systems [Mae96], graphical user interfaces [Rao91], object-oriented programming systems [Kic91a], and (within Xerox) distributed document systems. Although the approach provides many benefits, it is difficult for software engineers to apply because it requires them to design both modules that can effectively accommodate multiple implementation strategies and interfaces that permit clients to effectively use the modules. This paper addresses these difficulties by describing a *method* that enables software engineers to design effective Open Implementations in a disciplined way. This OIA/D method is currently the only defined approach available to help engineers apply the concepts of Open Implementation. The method is a result of our previous work in designing and building metaobject protocols [Kic91a, Mae96, Rao91], and is currently in use in two pilot projects within Xerox.

In this paper, we provide an overview of Open Implementation by comparing the approach to other methods of achieving reusable modules and by outlining the difficulties faced by a software engineer designing an Open Implementation. We then present an overview of the OIA/D method, followed by a detailed description of the analysis and design phases of the method. An example of the development of a block cache manager for a file management system is used to illustrate the concepts of the method. For further detail on the example, the reader is referred to Maeda [Mae96].

RELATED WORK

Software reuse has long been a desired goal to reduce the time and cost of developing and maintaining software systems. Goguen has written that "[s]uccessful software reuse depends upon the following tasks being sufficiently easy:

- finding old parts that are close enough to what you need,
- understanding those parts,
- getting them to do what you need now, and
- putting them all together correctly" [p. 160, Gog89].

The Open Implementation approach primarily helps engineers address the third task listed above by enabling engineers to build software modules[1] that are configurable, and therefore usable (or reusable), by a broad range of clients. The approach thus enables code reuse.

Various mechanisms have been proposed to enable software engineers to perform code reuse, including, among others, program transformation [BD77, Bal81] and component generation approaches [DFSS89, BSST93]. Unlike most existing transformation and generation approaches to help engineers configure existing software parts, the Open Implementation approach does not require a specialized language, generator, or environment. Rather, the concept is that a single module, often created with an existing programming language, can support multiple clients. For example, Lortz and Shin describe how Open

[1] Similar to the definition given by Parnas [Par72], the term module in this paper is used to refer to a work assignment.

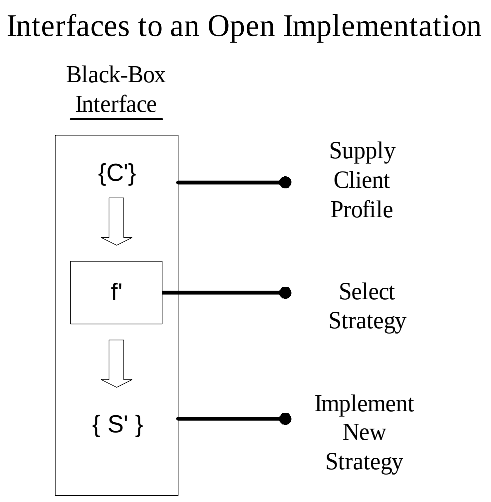

Figure 1: An OI module may provide multiple interfaces for the client to influence the internals of the module.

Implementations may be used to create customized server objects in a real-time database system in C++ [LS94]. In their approach, clients can express the properties and behavior they desired from a server as a string parameter passed to a server factory:[2]

```
MdartsArray<int> parts_list (“parts_list”,
“persistent; range_checked; sparse;
size=1000”);
```

The server factory object parses the second parameter to determine the appropriate class of server to create and provide. Passing a parameter with client information is just one way of providing an Open Implementation. The range of mechanisms that may be used to realize an Open Implementation module, and the ramifications of the various mechanisms, are described further by Kiczales and colleagues [KLLM96].

In comparison to existing software development methods, such as structured design [YC79], Booch [Boo91] , OMT [RBPFL91] and Shlaer-Mellor [MS88], that are intended to help a software engineer create a software *system* from a set of requirements, the OIA/D method focuses on helping an engineer design a specific, or small set, of software *modules* that are intended to be used as part of several systems. Instead of being a replacement for existing software development, then, the OIA/D method is a companion method. An engineer may use an existing---typically object-oriented---method to help model a problem domain and eventually implement the system. The engineer may then later apply the OIA/D method to help reanalyze and redesign modules that are found to be potentially reusable.

THE OI PROBLEM

The goals of any Open Implementation (OI) are to ensure that suitable implementation strategies are available for a range of clients, to ensure that the appropriate strategy may be selected for or by a client, and to ensure that the benefits associated with black-box abstraction are not unreasonably compromised. The designers and builders of an OI must take the client’s perspective and determine:

- the facets of client behavior that are relevant to the OI module,
- means of determining those facets of client behavior,
- the strategies to be implemented in the OI module, and
- a means of selecting a strategy for a particular client behavior profile.

The ease with which each of these issues can be determined has a strong effect on both the design of the interfaces and on the internals of the OI module. For instance, if it is hard to measure client behavior, then the OI module might need an interface that lets the client supply explicit information about its behavior. On the other hand, if it is difficult to determine the right implementation strategy to implement, the OI module might provide an interface that lets clients specify new implementation strategies.

Any OI module implements a set of implementation strategies (we call this set S) and must select the appropriate strategy for each client usage profile (from the set C of all possible client usage profiles). We model an OI as a function f: C $\rightarrow$ S that selects a strategy from S for each client usage profile from C.

There are two complications with this model that the engineer must resolve. First, since the space of possible user behavior is infinite, the engineer must determine a subset C’ of discernible client behaviors which covers some region of C and must partition the subspace of all behaviors into the elements of C’. Second, since the space of possible implementation strategies S is also very large, the engineer must also determine a subset S’ of implementation strategies that will be supported by the module. The designer must then provide mechanisms in the OI module to enable the selection of the implementation strategy from S’ that is best suited to the current user profile from C’. Returning to the model above, the OI module must support the function f’: C’ $\rightarrow$ S’.

Figure 1 shows the different kinds of interfaces that an OI might provide to allow clients to control its implementation strategies.[3] The Black-Box interface in Figure 1 is how clients access the functionality of the module. Every OI must provide a Black-Box interface with a default implementation strategy. By observing the client’s behavior at the Black-Box interface, an OI might select an appropriate implementation strategy for the client’s usage profile. Sometimes, however, selecting an appropriate strategy based on observations of client behavior may not be enough. For these cases, the OI may provide an optional interface, the Supply Client Profile interface in Figure 1, to permit the client to provide information about its future behavior. Since it is often difficult for clients to know detailed information about their own behavior, the OI may also provide an optional interface, the Select Strategy interface, that allows clients to simply select or describe the appropriate strategy. Other times, it is not possible for the OI to provide the appropriate range of strategies. In these cases, an optional interface may be provided

[2] This example is from Lortz and Shin [p. 457, LS94].

[3] We describe the access to the module as separate interfaces to emphasize that the interfaces are not, in some sense, equal. A client need not understand or use all interfaces that may be provided to an OI module. This use of multiple interfaces is described in further detail by Kiczales [Kic96].

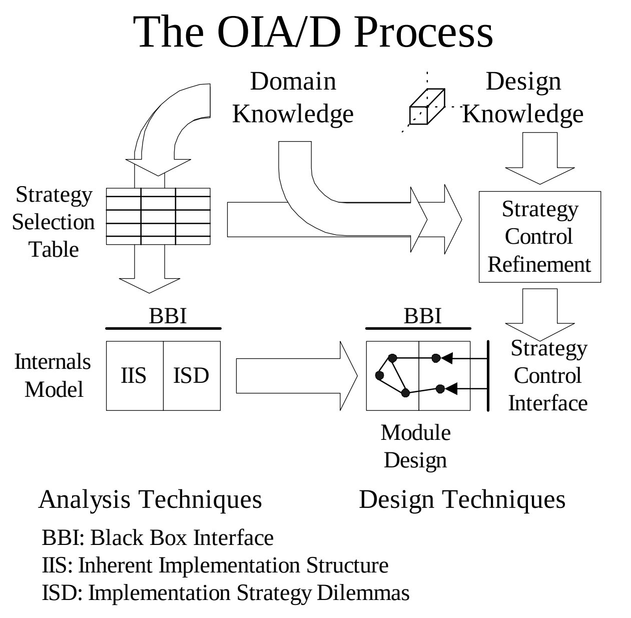

Figure 2: The analysis and design techniques of the OIA/D method.

that allows clients to describe or provide code that implements the appropriate strategy. This interface is labeled as the Implement New Strategy interface in Figure 1.

OIA/D OVERVIEW

The OIA/D method consists of two phases for determining the facets of a module's internals that need to be exposed to client control, and how those facets may be controlled in a disciplined way. As shown in Figure 2, the analysis phase consists of techniques for building a model of the module's internals (Internals Model), and for determining the set of usage profiles and strategies that will be supported by the OI (Strategy Selection Table). The design phase consists of techniques to transform the internals model into a design, and to apply the usage profile and strategy information to define interfaces by which clients can control the module's internals. Throughout the process, the techniques rely on a corpus of domain knowledge that describes how the module will be used and how clients would like the module to behave.

In the remainder of this paper we present the OIA/D method by describing the analysis and design phases. Although we describe each phase sequentially, the real-world application of these techniques is generally iterative. We illustrate the techniques comprising the method by describing their use in developing a block cache manager for the file management service of an operating system.

ANALYSIS

The method's analysis phase has two goals. The first goal is to understand what the module is intended to provide to clients. The second goal is to understand the clients desired use of the module from two points of view: understanding the range of client usage profiles that will be accommodated by the OI, and knowing the implementation strategy that is most suited for each usage profile. The first step toward meeting these goals is to acquire the appropriate domain knowledge.

Acquiring Domain Knowledge

To design an effective OI, the designer must have two kinds of domain knowledge: an understanding of the service provided by the module and an understanding of how different clients use the service. While effective techniques for capturing domain knowledge are essential to the successful use of OIA/D, the formality of the techniques necessary for capturing domain knowledge depends on the size and experience of the design team. We have found that an informal approach is often sufficient when working with a small team whose members are experienced in both the domain and the method. In this approach, the team simply relies on their knowledge of the problem domain without formally capturing the knowledge. For instance, in the file system example we use in this paper, the OI designer relied on his knowledge of the literature on the implementation strategies [Den71, Lef89, Cus93] and performance bottlenecks [Pat88, Sto81] of file management systems. The designer also examined the source code of several different clients of file systems to determine how the clients used the system and where there was a mismatch between the needs of each application and the services provided by the file system.

With a larger team, or a team whose members are inexperienced in the method, more formal approaches are generally needed. In this situation, one approach that we have found useful is to imagine conversations between the clients of the module and its implementers. (Real conversations with the real clients may also be used.) The topics of the conversations are the behavior of the clients, what the clients would like the module to do in different cases, and how the clients would like the module to behave.

We use this technique to capture domain knowledge by writing down the sentences of these conversations. For example, we captured domain knowledge in one of our pilot projects by asking the design team what the software should do but does not, and how the clients would most naturally communicate the right behavior to the module. The responses were used later in the analysis and design phases to understand what parts of the software module needed to be opened up and what form the control interfaces should take.

Modeling the Module

Trying to understand the internals of a module before the module is implemented or even designed may sound peculiar. However, we have found that most modules have what we call an inherent internal structure because of fundamental properties of the solution domain. (Recall that the OIA/D method is used once modules are selected using other software development techniques---most often OOA/D techniques, so we are dealing with modules in the solution, rather than the problem, domain.) For example, performance-critical applications that make heavy use of files must inherently deal with file caching issues because of the performance mismatch between disks and microprocessors in today's hardware architectures. These fundamental properties

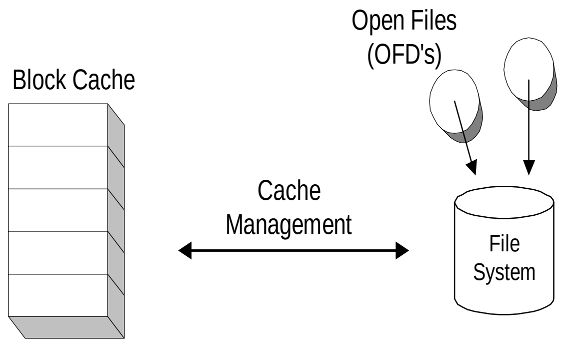

Figure 3: The inherent structure of a file management system.

of the solution domain of the module have a profound effect on the implementation of the module. Early recognition of these properties can help an engineer design an appropriate and effective OI of the module.

To construct a model of the module, we use the domain knowledge to determine three facets of the module: the Black-Box Interface (BBI), the Inherent Implementation Structure (IIS), and the Implementation Strategy Dilemmas (ISD). The Black-Box Interface is the simplest abstract interface to the module that captures the module's functionality independent of the implementation. The Inherent Implementation Structure consists of the internal components and structure common to any implementation of the Black-Box interface given the solution space. The Implementation Strategy Dilemmas are decisions that require knowledge of client usage patterns in order to be made optimally. These three facets can be thought of as partitioning the old notion of a software module as a black box into three parts.

Returning to the file system example, our designer determines that the Black-Box abstraction of a file is quite simple – a file may be modeled as a persistent, named, and ordered sequence of bytes. The Black-Box interface to the abstraction is also straightforward. Files may be `open`ed and closed. When a file is opened, the client provides the name of a file and receives an open file descriptor (ofd) in return. An ofd is conceptually a tuple consisting of a file and a numerical offset which is the client's current position in the file. Several operations are defined on ofd's: `read`, `write`, `seek`, `offset`, and `close`. Read returns one or more bytes of the file starting at the offset. Write stores one or more bytes at the file starting at the offset. Seek changes the current position. Offset returns the current position. Close frees the ofd.

Understanding the inherent implementation structure is more difficult. From his knowledge of the literature, the designer knows that file management implementations have an inherent solution structure that is due to the characteristics of the underlying hardware (Figure 3). Since files are persistent, they must be stored on mass-storage devices like magnetic disks, optical disks, or flash memory. Mass-storage devices tend to be one or more orders of magnitude slower than RAM and CPU in terms of both latency and throughput. As a result, there are two standard techniques for implementing file management services [Den71]:

- Store data in blocks to amortize the fixed cost of an I/O operation over many bytes.
- Keep blocks in an in-memory cache to avoid and/or mask the cost of I/O operations.

The concept of blocks and caches are used to form the inherent implementation structure.

Finally, the designer must determine the implementation strategy dilemmas. Again based on his domain knowledge, the designer knows that the implementation strategy dilemmas in a file management system center around how the block buffers in the cache are allocated among several concurrent clients and how each client's allocation is managed. The different file access behaviors of clients mean that a given cache management strategy will often have excellent performance for one client and dismal performance for another. In addition, the fact that several clients must share the block cache concurrently means that a given strategy might work well when the system is lightly loaded but might work poorly under heavy load.

Since the block allocation module manages a physical resource (block buffers in physical memory), the block allocation module must be implemented as part of the operating system kernel in order to preserve system integrity. The cache management module may be implemented by each client because it manages blocks that have already been allocated to the client by the block allocation module.

Together, the BBI, IIS, and ISD provide a model of the module, including both its interface and its internal structure. In the design phase, we will see how this model is used to derive the initial design of the module. The file system example described in this section relied on implicit domain knowledge. If, instead, the domain knowledge is in the form of statements from conversations, each statement can be analyzed to determine if it provides information about the Black Box Interface to the module, the Inherent Implementation Structure of the module, or the Implementation Strategy Dilemmas of the module.

Analyzing Usage Profiles

The model of the module the designer creates starts to elicit the client behaviors (C) and implementation strategies (S) that the OI might support. To build the OI, though, the designer must still determine the relevant subset C' of user behaviors, the relevant subset S' of implementation strategies, and the mapping from elements of C' to elements of S' (f') that the OI will realize

Similar to the development of the model of the module, a designer determines this information by applying domain knowledge. Specifically, the designer uses the domain knowledge to construct a Strategy Selection Table. This table has a set of entries for each Implementation Strategy Dilemma identified in the model. Figure 4 shows the Strategy Selection Table for the file system example. For each dilemma, the designer has determined the possible ways to resolve the dilemma and which clients will like this decision. For instance, the designer has recorded that the "Cache Fetching" dilemma has three possible strategies and that the fetch on demand strategy is useful for program executable files (the first line of the table). When the Strategy Selection Table is complete, the "Who likes it column" provides the subset of relevant user behavior (C'), the "Decision" column provides the subset of strategies (S'), and the

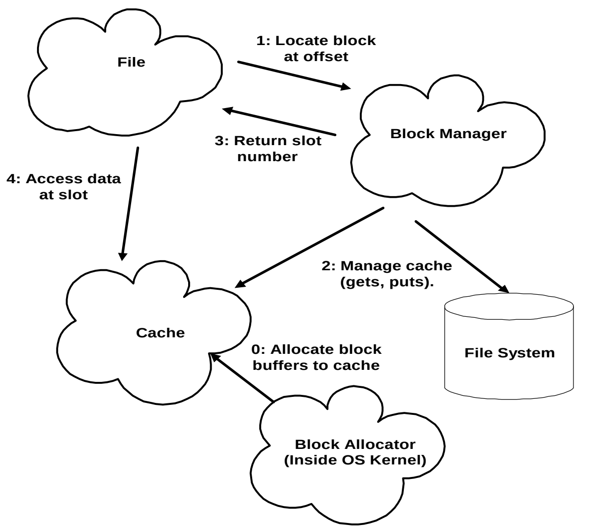
File
1: Locate block at offset
Block Manager
3: Return slot number
4: Access data at slot
2: Manage cache (gets, puts).
Cache
0: Allocate block buffers to cache
File System
Block Allocator (Inside OS Kernel)

Figure 4: An object diagram for the file management system. The numbers show the order in which objects are invoked.

two columns together provide the mapping from user profiles to strategies (f').

We have found that a group brainstorming activity is a useful way of constructing the Strategy Selection Table. Keeping the contents of the table updated as discussion proceeds lets it serve as a kind of organizational memory for the discussion. Furthermore, recording this information in tabular format makes it easy to use the information during the design phase.

DESIGN

As a result of the analysis phase, a designer produces two things: the Internals Model and the Strategy Selection Table. The Internals Model captures the abstract functionality of the module, the inherent domain properties that determine how it is implemented, the design decisions that require information about client behavior in order to implement well. The Strategy Selection Table tells the designer how "open" the module must be; that is, the range of user behaviors that it must support and the range of strategies that must be implemented to support this user behavior.

In this section we discuss techniques for designing the internals and interfaces to the OI. We start by describing techniques for transforming the Internals Model into a preliminary design. Then, we describe how to design the interfaces that control the module internals.

From Model to Design

A variety of design techniques could be used to transform the Internals Model into a module design. Generally, we perform this transformation by mapping each element of the solution structure into an object in the design. Furthermore, we map each Implementation Strategy Dilemma into an object in the design so that changes to an implementation strategy will be localized in an object.

In the case of the file system, it is relatively straightforward to derive a design from the solution structure described earlier that consisted of a blocked file system on persistent storage and an in-memory block cache. The designer simply transforms each element of the solution structure and each implementation

Dilemma	Strategy	Who likes it
Cache Fetching	Fetch "on demand" – when a desired block is not in the cache.	Program executable files that have locality due to loops and unpredictable reference patterns due to branching.
	Sequential read-ahead – fetch next blocks in file when a given block is referenced.	Programs that read and write data files, for example most command line utility programs.
	Custom – client provides information about its future reference pattern that is used to select blocks for prefetching.	Programs with well-known reference patterns that appear random to the operating system, such as table indices in relational databases.
Cache Flushing	Flush on fetch – flush a block when another block is about to be fetched. The flushed block is recycled to hold the fetched block.	Executables and data files that are reused by several processes. This strategy enables these files to remain cached over multiple program invocations.
	Flush on finish – flush a block when a program has finished with it.	Good for data files that are unlikely to be reused for when a program needs to limit the size of its cache footprint.
	Custom strategy – client says when it needs blocks to be flushed.	Databases and other programs that need to guarantee persistence.
Cache Allocation	Best effort – give cache blocks to currently running program.	Programs with small working set size – performance degrades linearly with cache allocation.
	Guarantees – provide strict cache allocation. Clients may negotiate cache allocation sizes.	Programs like databases with larger working sets that can tailor their algorithms to their working set size.

Figure 5: The Strategy Selection Table for the cache management subsystem of file management system.

Interface Style	Strategy Selection	Tradeoffs
Style A – No custom control interface	Module selects implementation strategies by observing client's use of the Black-Box Interface.	Same as Black-Box Abstraction.
Style B – Client provides declarative information about its usage pattern. For example, when opening a file, the client can provide a description of its behavior, such as "sequential file scan," that the module can use to select strategies.	Module selects strategy by matching usage pattern information from client to the best available strategy.	Client specifying information about its usage pattern doesn't constrain the implementation. Difficult for client to know how it is influencing module internals.
Style C – Client specifies the implementation strategy the module should use, such as "LRU cache management" for the file cache example.	Module adopts the strategy specified by client.	Easy to select strategy. However, client might be uninformed or wrong about best strategy to use.
Style D – Client provides the implementation strategy to use. For example, client might provide a pointer to an object that implements some cache management protocol when opening a file. Subsumes Style C if module provides predefined strategies.	Module adopts the strategy provided by client.	Easy to select strategy. Engineering module to support replaceable strategies might be difficult. For client, engineering a new strategy implementation might be expensive.

Figure 7: Open Implementation interface styles (from [KLLMM96]).

strategy into an object such that the design consists of five objects: the file system, ofd's, cache, block allocation strategy, and block management strategy. Figure 4 shows a design diagram, based on a Booch object diagram [Boo91] of these objects and their interactions.

The Block Allocation module, which is inside the operating system and encapsulates the Cache Allocation strategy, allocates block buffers to the cache from the physical memory pool. Once a cache has been allocated buffers, the Block Manager, which encapsulates the Cache Fetch and Flush strategies, determines how these buffers are managed. The interaction in Figure 4 shows what happens when the client reads an open file through an ofd object.

Other transformation strategies may also be applicable to create the initial design from the Internals Model, such as the fine-grained module composition approach described by VanHilst and Notkin [VN96], but have not yet been investigated.

Designing the Interfaces

Once we have the initial module design, the missing pieces are the control interfaces that allow clients to control implementation strategies in use. The Black-Box interface developed as part of the Internals Model presents only the functionality of a module, hiding all implementation issues. The engineer is now faced with the design of additional Strategy Control interfaces to provide the necessary control. The design of these interfaces proceeds as two steps. First, the engineer selects appropriate styles of interfaces given the strategies outlined in the Strategy Selection Table. This step is largely independent of the domain. Second, the engineer refines the interfaces based on the domain information acquired earlier.

Selecting an Interface Style

Kiczales and colleagues have outlined a number of interface styles available to realize an OI [KLLMM96]. These interface styles are summarized briefly in Figure 7. Each of these styles has different associated costs and benefits. To ease the selection of an appropriate style, we have determined a domain-independent mapping from properties of the OI to these interface styles. Figure 6 shows the design space defined by the properties of the OI. The three axes of the design space are the ease or difficulty of determining the set of client usage profiles C', the set of implementation strategies S', and the function f' mapping usage profiles to strategies. We have divided the design space into eight regions which roughly correspond to whether each of these questions is easy or hard. As shown in Figure 8, each region has a certain interface style that is most suited for it. The data in Figure 8 can be summarized by the following decision rule:

```
If determining S' is hard, use Style D
else if determining f' is hard, use Style C
else if determining C' is hard, use Style B
else use Style A.
```

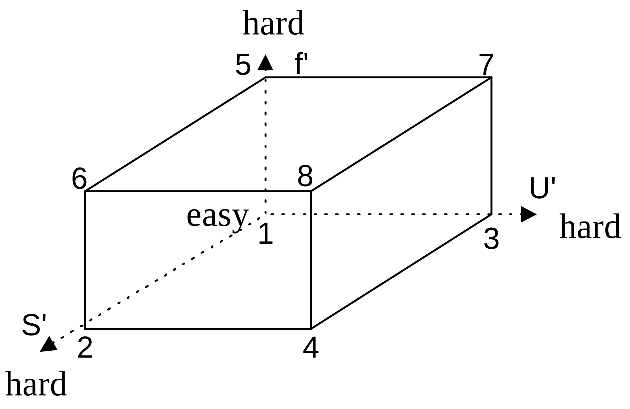

Figure 6: The difficulty of determining the sets C' and S', and the function f' defines the design space of the OI module. Each region of the design space (numbered 1 through 8) has a certain interface style that works best.

Region #	C’	S’	f’	Interface Style
1	Easy	Easy	Easy	Style A – OI is not necessary; module can simply observe client behavior and dynamically adjust strategy to suit.
2	Easy	Hard	Easy	Style D – it is difficult to choose which strategies to implement but easy to determine client behavior and map it to the appropriate strategy. An OI might let clients dynamically extend the range of strategies implemented. Since selecting the strategy is easy, clients can handle it.
3	Hard	Easy	Easy	Style B – it is hard to measure the client usage profiles although it is easy to determine the strategy once the usage profile is known. Clients should supply the usage profile which can easily be used to decide the strategy. Style C is also possible if the set of possible strategies is relatively static.
4	Hard	Hard	Easy	Style D – it is hard to determine the client usage profiles and the set of possible strategies. With Style D, the client can specify the strategy to use and can also provide a new strategy if the current set is inadequate.
5	Easy	Easy	Hard	Style C – it is easy to determine client usage profiles and the set of possible strategies. However, the difficulty of selecting a strategy for a given usage profile requires the client to specify the right strategy. Measure and decide strategies.
6	Easy	Hard	Hard	Style D – while it is easy to determine the set of client usage profiles, it is difficult to determine the set of implementation strategies and to select a strategy for a given usage profile. Style D allows the client to specify a strategy or to supply a new strategy if necessary.
7	Hard	Easy	Hard	Style C – client should select strategy since it is difficult to determine the client usage profile and to select the right strategy given the usage profile.
8	Hard	Hard	Hard	Style D – client tells module what to do.

Figure 8: Determining an Interface Style for each region of the design space.

An engineer starts to apply the mapping by determining whether it is ‘easy’ or ‘hard’ to determine the provided client usage profiles (C’), whether it is ‘easy’ or ‘hard’ to determine the set of provided implementation strategies (S’), and whether it is easy or hard to find a solution to the function f’: C’ $\rightarrow$ S’. The answers to these questions determine the region of design space in Figure 6 which dictates the best interface strategy as shown in Figure 8.

To return the file management example we described previously, the lifetime of an operating system (10-20 years) spans several generations of computer hardware (1-2 years per generation). Each new generation is more powerful than the last (due to Moore’s Law) and enables a new and more powerful set of applications previously unforeseen. As a result, it is difficult for the designer to determine the sets C’ and S’ and the function f’ because doing so would require predicting implementation strategies suitable for the behavior of applications several years in the future. The file management system is very much at region 8 of the design space in Figure 6.

The file management example uses Interface Style D for the Block Manager module. Interface Styles A, B, or C are used for the Block Allocation module because the Block Allocation module is part of the operating system (runs in the operating system kernel) and is shared by all instances of the Block Manager. Since the Block Allocation module cannot run unprotected client code without compromising the integrity of the operating system, Style D cannot be used.

Refining the Custom Control

Using the Interface Style Selection Table of Figure 8, the engineer can choose the appropriate **kind** of Custom Control interfaces for the OI. But, the engineer must still select a particular style and design the contents of the interface. We have developed a set of techniques to help an engineer with this design that is based on the treatment of the interface(s) as a language; each use of the interface corresponds to a statement in the language.

An engineer starts applying the technique by taking the conversations between the module implementer and the module clients from the domain knowledge acquisition stage and re-expressing each statement as a statement in the language of the Custom Control Interface. The engineer then iteratively refines the Interface by making changes to the language and re-expressing each statement in the new language until the desired expressiveness in the language is reached. The statements are analyzed using the framework described in Figure 9. Each statement is analyzed to determine its Subject, Vocabulary, Scope, and Binding Time. Changes to the Interface language are made by changing one of these features and are evaluated by re-expressing each statement in the new language to see if they are more natural.

For example, imagine two different interface languages for controlling the disk caching behavior of a file system. In the first language, the vocabulary is in terms of the Black-Box abstraction of a file while the second language deals with the underlying blocks that make up the cache. In the first language, the client might make a statement like, “I’m going to read this file from

Language Feature	Description
Subject	Whether the statement describes the client’s or the implementation’s behavior.
Vocabulary	Whether the words in the statement are in terms of Black-Box Interface or of the implementation structure?
Scope	What elements of the implementation does the statement apply to? (For example, a single instance of the module versus all instances of the module.)
Binding Time	Can the statement be made at design time, compile time, link time, or run time. This feature helps determines whether static or dynamic implementation mechanisms can be used.

Figure 9: Elements of the Interface Languages.

beginning to end.” In the second language, the same statement would be expressed as, “When I access block x of this file, the next block I access will be block x+1.” The second language has more expressive power because it can be used to express complex access patterns. The first language is less powerful but probably easier to use because certain access patterns, such as “beginning to end,” are easy to express. The C’ and S’ subsets determined in the analysis phase can tell whether the candidate interface language is sufficiently powerful to express all the elements of C’ and S’, or whether it needs further refinement.To see how the refinement techniques work, let us examine some statements that client programmers would say. For the file system example, two of those statements could be:

1. “I am going to scan this file from beginning to end.”
2. “I am going to repeatedly access the first block and randomly the others.”

While the subject of both statements is the behavior of the client, Statement 1 uses the vocabulary of the Black-Box Interface language (i.e. “file” and “beginning to end”) while Statement 2 uses the vocabulary of the Inherent Implementation Structure (i.e. “block”). We would like both statements to have identical language features; in this case, a common vocabulary. Let us try to rephrase the first statement in the vocabulary of the second statement.

1’. “I am going to read the blocks in order.”

This translation was easy and natural as long as “order” is a concept in the vocabulary of the IIS. On the other hand, it is difficult to translate Statement 2 into the vocabulary of Statement 1 because there is no concept of “block” in the language of the Black-Box Interface. Our goal is to arrive at a common language for expressing all of the statements clearly and easily. This common language defines the syntax and semantics of the Custom Control Interface between the OI module and its clients. In this example, we will end up with statements 1’ and 2 and use this language for the Custom Control Interface.

STATUS AND CONCLUSIONS

The OIA/D method is a way of designing high-performance, highly reusable software modules using existing technology. The method applies once a module has been selected and targeted for reuse, and thus is compatible with existing software development methods. The analysis phase of the OIA/D method consists of techniques for constructing a model of the module’s internals and interfaces, and for understanding the user profiles and implementation strategies that should be supported. The design phase consists of techniques for transforming the analysis module into a design and for designing the interfaces that allow clients to control the module’s internals.

The method is still young and its techniques are still evolving. In particular, we would like to have a better understanding of the relation between the regions of the design space and the interface styles most for suited for each. Also, the cache management example showed that a certain style of interface was not suitable when shared resources were being managed. This may be an additional dimension in the design space that needs to be explored.

To date we have used the method in Xerox for pilot projects in distributed document services. Our experiences show that OIA/D is useful for recognizing the design decisions that must be left to client control early in the lifecycle. Additional studies of the methodology are underway at Korea University.

REFERENCES

[Bal81] Balzer, R. Transformational implementation: An example. *IEEE Transactions on Software Engineering*, SE-7(10):3-14, 1981.

[BD77] Burstall, R. and Darlington, J. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery* 24 (1):44-67, 1977.

[Boo91] Booch, G. Object-oriented design with applications, Benjamin/Cummings, 1991.

[BSST93] Batory, D., Singhal, V., Sirkin, M. and Thomas, J. Scaleable software libraries. *In Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1993.

[Cus93] Custer, H. Inside Windows NT, Microsoft Press, 1993.

[Den71] Denning, P. J. Third Generation Computer Systems. Computing Surveys 3(4):175-216, 1971.

[DFSS89] Dubinsky, E. Freudenberger, S., Schonberg, E. and Schwartz, J.T. Reusability of design for large software systems: An experiment with the SETL optimizer. In Software Reusability: Concepts and Models (Volume 1), edited by T. Biggerstaff and A. Perlis, chapter 11, ACM Press, New York, 1989.

[JF88] Johnson, R.E. and Foote, B. Designing reusable classes. *Journal of Object-Oriented Programming*, 1 (2): 22-25, 1988.

[Gog84] Goguen, J. Parameterized programming. *IEEE Transactions on Software Engineering* SE-10(5):528-543, 1984.

[Gog89] Goguen, J. Principles of parameterized programming. In Software Reusability: Concepts and Models (Volume 1), edited by T. Biggerstaff and A. Perlis, chapter 7, ACM Press, New York, 1989.

[Kic91] Kiczales, G. Towards a new model of abstraction in software engineering. *In Proceedings of the 1991 International Workshop on Object Oriented in Operating Systems*, IEEE Computer Society Press, Los Alamitos, CA, p. 127-8.

[Kic91a] Kiczales, G., des Rivières, J., and Bobrow, D. G. The Art of the Metaobject Protocol, MIT Press, 1991.

[Kic96] Kiczales, G. Beyond the black box: Open Implementation. *IEEE Software*, 13(1): 8,10-11, 1996.

[KLLMM96] Kiczales, G., Lamping, J., Lopes, C.V., Mendhekar, A. and Murphy, G. Open implementation design guidelines. Submitted to the *19th International Conference on Software Engineering*, 1996.

[Lef89] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. The Design and Implementation of the 4.3BSD UNIX Operating System, Addison-Wesley, 1989.

[LS94] Lortz, V.B and Shin, K.G. Combining contracts and exemplar-based programming for class hiding and customization. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, New York, 1994, p. 453-467.

[Mae96] Maeda, C. "A Metaobject Protocol for Controlling File Cache Management." Proceedings of ISOTAS'96 (International Symposium on Advanced Technologies for Object Software), Kanazawa, Japan, 1996.

[MS88] Mellor, S.J. and Shlaer, S. Object oriented systems analysis: modeling the world in data, Prentice Hall, 1988.

[Par72] Parnas, D. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053-1058, 1972.

[Pat88] Patterson, D., Gibson, G., Katz, R. A Case for Redundant Arrays of Inexpensive Disks (RAID). In Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD), 109-116, 1988.

[Rao91] Rao, R. Implementational Reflection in Silica. In Proceedings of European Conference on Object-Oriented Programming (ECOOP) (1991), P. America, ed., vol. 512 of Lecture Notes in Computer Science, Springer-Verlag, 251-267.

[RBPEL91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. Object-oriented modeling and design, Prentice Hall, 1991.

[Sto81] Stonebraker, M. Operating System Support for Database Management. Communications of the ACM, 24(7):412-418, 1981.

[VN96] VanHilst, M. and Notkin, D. Decoupling change from design. To appear in *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Engineering*, October 1996.

[YC79] Yourdon, E. and Constantine, L.L. Structured design: Fundamentals of a Discipline of Computer Program and System Design Prentice Hall, Englewood Cliffs, N.J.,