Managing Abstraction-Induced Complexity

David Keppel
UWCSE TR 93-06-02

June 11, 1993

Abstract

Abstraction reduces the apparent complexity of an implementation by hiding all but "the most relevant" details. However, no interface is suitable for all the users of the implementation for exactly the same reason: each user has a slightly different view of what is "most relevant." Thus, although abstractions can reduce complexity, working around their limitations can introduce other complexities. Some abstractions are designed to minimize the added complexity. This paper examines five common abstraction models and uses common examples to contrast the tradeoffs presented by each model.

1 Introduction

"You can have any two: fast, clean, robust." – Programmer's Adage

Abstraction reduces the apparent complexity of an implementation by hiding all but "the most relevant" details. However, no interface is suitable for all the users of the implementation for exactly the same reason: each user has a slightly different view of what is "most relevant". Thus, although abstractions can reduce complexity, working around their limitations can introduce other complexities.

This paper presents interfaces and implementations as a spectrum choices. Several interfaces are selected as idealized models; the paper illustrates each model using examples of existing systems. It then discusses some of the needs that led to each abstraction, and also considers some limitations of each approach.

Author's address: Department of Computer Science and Engineering; University of Washington, FR-35; Seattle, Washington 98195. Electronic mail address: "pardo@cs.washington.edu".

In this paper, an *interface* is an abstraction or model of a *service* that performs useful work. A *client* is a piece of code that calls upon the service to perform some part of the client's task.

An interface has three jobs. The first is to present behavior that the client can use to solve its problem. The second is to hide enough details that the client finds it easier to use the service than to solve the problem directly; that is, the "goodness" of an interface is the extent to which it simplifies the client [Par79]. The third is to hide details that are likely to change [Par72].

Designing interfaces is tricky because of conflicting demands: The abstraction should be specific enough that it is simple, but also general enough that many clients can use it. An ideal interface hides all implementation details, so the client can focus on getting the job done instead of worrying about how it gets done. Yet the underlying implementation should suit all clients, and each client can place different demands on each feature of the service.

Thus, while abstractions can simplify clients, they can also induce complexity by forcing clients to worry about certain details. Some details of this abstraction-induced complexity are discussed below.

1.1 Abstraction-Induced Complexity

The interface and implementation should ideally provide good performance, conceptual cleanliness, robustness, and portability. Typically, however, systems that are clear and general have inadequate performance [Lam84]. Although system designers are ultimately interested in solving a broad range of problems, performance is often improved at the expense of generality. The emphasis in this paper is interface designs that achieve good performance without sacrificing generality.

Since different clients have differing demands, performance problems cannot generally be solved by simple tuning: optimizations for one client are typically pessimizations for some other client. Thus, performance problems are often dealt with using one of three strategies [Kic92]:

- FIXED: The service provides a good interface and an implementation that is tuned for some subset of the clients. Clients that suffer bad performance are forced to work around the performance restrictions of the service. These workarounds complicate the client by forcing it to reimplement major parts of the service, or force it to take advantage of nonportable implementation details.

- PROLIFIC: The service attempts to solve performance problems by supplying many interfaces, each specialized for particular client needs. For example, programming languages can provide many kinds of arrays: dense, sparse, integer-indexed, and so on; operating systems can try to support many kinds of scheduling.

  However, this approach bloats both the interface and the implementation. Programmers have trouble first in understanding the verbose interface, and then in deciding which parts of the interface they should use. Service implementors have problems writing and debugging the implementation. Essentially, complexity arises because the service is promising many fixed implementations, one to solve each problem.

  Another difficulty is that supplying many special-purpose interfaces still doesn't solve the general problem. Ultimately, some client needs to use the service in a way for which there is no special interface. The service can be updated each time the need arises, but at great cost to both the service maintainer and the programmer writing the client. Alternatively, the service can supply an additional general-purpose interface, and clients that can't use the special interfaces can fall back on the general interface. Yet it is problems with the general-purpose interface that led to the plethora of special interfaces. Although *some* clients use the general code only rarely, other clients surely have it as a common case, and those clients suffer for it. Likewise, since the general interface is used only rarely, it is typically poorly-tuned and unreliable [Fat85].

- META-CONTROL: A third strategy uses two interfaces. Usually the client is concerned with the *primary* interface that performs operations for the client. Sometimes the client also makes use of a *meta-control* interface that leaves the primary interface unchanged but changes the way the service operations are implemented.

  Using a meta-control interface helps separate the implementation issues from the interface issues. Instead of using many special-purpose interfaces, one for each implementation, there are just two: the primary interface, which is the same for all clients, plus the meta-control interface, which allows each client to pass information to the service in order to get the desired implementation.

  In general, clients supply some code to perform tuning; thus meta-control interfaces introduce some complexity. Different meta-control interfaces provide different capabilities, and, therefore, introduce different kinds of complexity.

Abstractions can reduce the complexity of a client, yet the same abstractions can introduce complexities of their own, with different abstractions introducing different complexities. Prolific interfaces are complex and provide generality through a low-performance interface, so this paper focuses on tradeoffs among fixed and meta-control designs.

1.2 Outline

The paper introduces and compares five interface models, and shows tradeoffs among them. The models are presented as points in a spectrum of choices, with each model solving some problems but creating others. The goal is to help service designers understand particular tradeoffs involved in developing clean, robust and portable interfaces that meet the performance demands of a wide variety of clients.

Although this paper examines common interface models, there are several important issues that are not addressed: First, the paper does not provide exact rules for choosing an interface model to solve a particular problem. Second, the paper does not discuss higher-order interfaces; for example, it may be possible to partially-evaluate [Par91] richer interfaces in various ways in order to produce each of the interfaces discussed here. Third, this paper discusses performance tuning but does not explore alternative ways of improving the service's performance, such as

using approximate or probabalistically correct algorithms. Finally, this paper does not address the evolution of interfaces as new features are needed.

The remainder of this paper is organized is follows: Section 2 briefly introduces each model. Section 3 gives concrete examples of each model and discusses tradeoffs between them. Section 4 discusses related issues, such as interfaces that use hybrid models, protection, and composition of services. Finally, Section 5 summarizes the important features of each interface.

2 Models

This paper presents five interface models, all in common use. The goal of this paper is to compare them and see how the abstractions overlap, and to examine how each does and doesn't solve certain kinds of problems. The models, briefly, are:

- FIXED: the abstraction promises it will solve certain problems, but makes no promises about how it will do it. The programmer generally assumes a single implementation for all uses. Alternatively, the abstraction promises key implementation features will be the same across all implementations of the service.

- ADAPTIVE: The abstraction promises that it will figure out how to use a "good" implementation, even if differing client demands imply drastically different implementations of the interface.

- ADJUSTABLE: The abstraction includes a meta-control "tuning knob" by which the client can pass usage hints to the services.

- OPEN: The abstraction includes a reflective [Smi84] mechanism by which the client "injects" new code in to the implementation [Kic92]. The injected code reimplements key details of the service so they are optimized for the client.

- INCOMPLETE: The meta-control and primary interfaces are merged. The service provides only a part of the implementation. The client provides both tuning and missing parts of the service.

Each of these designs is similar to that of its neighbors. Thus, each tends to solve problems in ways that are similar to its neighbors, and each tends to have limitations similar to those of its neighbors. The remainder of this section examines some of the typical ways in which the interfaces differ.

The adaptive model differs from the fixed model in that the adaptive system promises to examine particular circumstances and select a good implementation, even if there is no single good implementation for all circumstances. In a sense, a fixed model is a degenerate adaptive model that refuses to adapt.

An adjustable system allows the client to give the service hints about what implementation is best. Typically, hints are either selected from a fixed set of choices or are represented by a value along an axis of tuning.

Open systems are like adjustable ones, as both use a meta-control interface to tune the implementation. An open interface exposes parts of the implementation and the client uses a general-purpose programming language to perform tuning. Tuning is accomplished by reimplementing core operations to best suit the client. Note that adjustable systems export interfaces that tune only part of the implementation; in much the same way, open systems export only some of the implementation details.

Incomplete systems take openness a step further: they embed the meta-control interface in the primary interface. Thus, they *require* the client to implement core operations. The service implements parts of operations that are suitable for all clients, and the client supplies whatever the service cannot do well. The programmer's model is strongly different from the open model, because core operations are viewed as parts of the client code instead of being seen as additions to the service. However, in another sense, the client-supplied core operations exist soley to satisfy the particular service and thus are not distinct from operations that are "injected" into an open system.

Although the interfaces above are presented as a continuum, they are perhaps best viewed as a spiral, where an incomplete implementation is adjacent to a fixed implementation. The part of the incomplete implementation that *is* provided is usually fixed. However, the incomplete implementation provides a simpler (less complete) set of primitive operations than are provided by a higher-level fixed interface.

3 Common Examples

All of the above interfaces are in common use. This section provides examples of each and describes some

of the problems that they solve well and some that they solve poorly.

This section uses five example services from three areas: arrays and procedure calls from programming languages; virtual memory and thread scheduling from operating systems; and machine branch instructions.

The remainder of this section has one subsection for each interface model. Each subsection is organized as follows: First, the subsection briefly describes the subsection's model. Next, the model is elaborated with each of the five example services. Then, the characteristics of the model are discussed: what problems it solves well, and what are some of the limitations of the approach. Finally, each subsection ends with a comparison to the model presented in the preceding subsection.

3.1 Fixed

Fixed interfaces provide the same abstraction to all clients. To the extent that the implementation is "good enough", fixed interfaces work well because the interface is simple and consistent.

Programming languages often provide a single fixed abstraction for arrays and procedures. For example, C provides arrays that are indexed by small integer values and that are space efficient if the array is used densely [KR88]. Clients that want e.g., sparse arrays either suffer poor space efficiency or must build sparse arrays using other language constructs. Straightforward implementations of procedure call and return are simple and effective, but small procedures suffer a substantial procedure call overhead.

Operating systems generally provide abstractions for memory and threads of control. For example, *virtual memory* (*VM*) can be viewed as being resident in physical memory over the lifetime of (one invocation of) the client, even though it may occasionally be copied out to secondary storage and later copied back. Thread abstractions provide a way of starting and stopping units of parallel execution. Fixed implementations such as FIFO (first in, first out) page replacement and thread scheduling work well as long as the client makes sparse use of the abstraction: where memory is used modestly or when rescheduling is rare. They work poorly, however, when a processes' VM approaches the physical memory size, or when the application is sensitive to scheduling policies.

Architectures provide an abstract interface on top of the underlying machine's state. Fixed abstractions allow clients to perform branches without worrying about implementation details of how the processor pipeline is updated. Performance is typically good if branches are rare or the pipeline is short, but performance suffers with deep pipelines and frequent branches [YP92].

Fixed abstractions are generally simple to implement and use. They also provide control over just the operations that the client wants performed. Fixed implementation are also efficient when the implementation is a good match to the client's use of the service. However, fixed abstractions usually fail either by limiting the operations to those with a consistently good implementation, or by hiding important performance details [Kic92].

3.2 Adaptive

Adaptive systems choose an implementation based on the way the client uses the service. That is, different clients make different kinds of requests, and the kinds of requests communicate implicit information that a service can use to perform tuning. Adaptive and fixed interfaces generally have the same *program* interface, but have a different *programmer* interface because some of the self-tuning (or self-modifying) nature of the implementation is discussed in the documentation. The distinction is important because the programmer may perform certain operations as implicit hints to the service or may avoid certain operations because the programmer interface warns that it may be unable to adapt to them.

Programming languages often provide adaptive interfaces by allowing a programmer to use fixed abstractions for arrays and procedures, while promising that the translator can perform optimizations that yield efficient implementations. Most programming language implementations discover optimizations statically [ASU88], but some systems collect information based on runtime use of the constructs [DB77, GW78, DS84, CW86, CU91].

Operating systems provide VM and thread abstractions that, at some level, promise to divide machine resources between various parts of a client in a way that reflects the current needs of the client. The usual VM abstraction, for instance, promises fixed-sized *pages* and promises that if memory is used sparsely, only the pages that are used will count towards the client's use of memory [Sun90]. Most thread abstractions provided by OS kernels adapt to the thread's use of resources, e.g., boosting CPU priority if the

thread is mostly I/O bound [PS83].

Architectures provide adaptive branch prediction, where the hardware keeps information about each branch instruction. When a given instruction is executed, the predictor attempts to update pipeline state in a way that reflects common use of the branch instruction [HP90, YP92].

Adaptive systems generally perform well to the extent that they can deduce the needs of the client. However, adaptive systems are only as good as the tuning information that they deduce, and they can lag behind when they rely on past behavior to predict future needs. An adaptive system that adapts poorly may perform worse than a fixed implementation. Adaptive systems can also perform poorly when the cost of adaptation is more than the overhead cost of using a simpler fixed implementation [KEH91].

3.3 Adjustable

Adjustable systems are like adaptive ones that have the tuning mechanism exposed in a meta-control interface. Where the adaptive system deduces and predicts client behavior, an adjustable system simply lets the client tell it what is going to happen.

Programming languages often provide tuning knobs e.g., to trade between implementations that are complex but space-efficient and those that are algorithmically efficient but waste space. These programmer directives are supplied because the programmer may have information the compiler cannot deduce about whether arrays are full or sparse or whether certain procedure calls are common or rare. Tuning is often performed using command line flags (`-finline-functions`), using magic comments (`#pragma`[1] `inline`), or using reserved keywords (`inline`) [Sta89].

Operating systems have tuning and scheduling knobs, again because the programmer may have information the OS cannot deduce. For example, the programmer may indicate one memory region is used sequentially, while another is used randomly, while yet a third is used in bursts (e.g., `madvise` and `vadvise` [Sun90]). Or, a programmer may know either that the client is composed of nearly-independent tasks, or that tasks cooperate closely and thus benefit from gang scheduling.

[1] The word "pragma" is derived from "pragmatic". The usual use is as an allowance for cases where the service performs the desired operation but where performance, storage layout, or other such "pragmatic" concerns require the implementation to be adjusted some way.

Architectures often reserve branch opcode bits that the programmer or compiler can use to tell the architecture whether or not a branch is typically taken [HP90, Sit93]. The compiler or programmer may have that information when the branch is generated because a certain branch corresponds either to a part of a test for an unusual (e.g., error) case or a part of a test for termination of a loop.

Adjustable systems can improve over adaptive systems because the client often knows things that are hard for the adaptive system to infer. Adjustable systems are particularly successful where there are a few good choices that cover most of the cases (e.g., branch is or isn't usually taken) or where important information can be summarized with a few values (e.g., arrays usually have 1,000 elements and only about 10 elements are used).

Adjustable systems, however, can suffer compared to adaptive systems for a variety of reasons. First, the adjustable system forces the client to compute and communicate tuning information in a timely way; the adaptive one simply deduces information when it is needed. Clients of adjustable systems may be punished with bad implementations if they lie or fail to be diligent in providing good tuning information. Second, adaptive systems can deduce implementation-level information, where clients manipulate abstractions and thus necessarily pass abstracted information. Third, tuning information is selected from a fixed set of choices, which limits what information the client can provide. If better tuning information is needed, an adaptive service can be changed transparently, but an adjustable system may require client changes. Finally, it may be difficult to develop a simple interface that can identify particular circumstances, e.g., that the first 9 elements of an array are almost always used and that only one of the succeeding 991 is typically used. Conversely, it may be easy to develop an interface for any single situation, but hard to develop one that is both general and simpler than the (pre-existing and familiar) general-purpose programming language used by an open system.

3.4 Open

Where an adjustable system is tuned with flags and scalar values, an open system is tuned using code that is "injected" in to the service to achieve the desired behavior. Thus, where an adjustable system allows a client to communicate only limited tuning information, an open system allows more general control by

performing tuning with a general-purpose programming language.

An open translator for a program has a default implementation for, say, dense arrays indexed by small integers. A client wanting sparse arrays uses the meta-control interface to modify the compiler to implement sparse array allocation and indexing [KdRB82, Kic92]. In a like way, the translator can be augmented with code that examines procedure calls to check for application-specific information that indicates how each procedure call should be implemented [Rod91].

Open VM systems are rare because it is hard to safely migrate code from the user space in to the kernel; the issues are discussed further in Section 4.2. An open threads package can provide a primary interface to create, schedule, and run threads [BLL88], with a meta-control interface that e.g., lets the client reimplement synchronization objects in order to change the way in which threads behave when they block [BLLW88].

Even architectures can support an open interface, by allowing either user-programmable microcode or dynamically-configurable hardware [JF91, Sut91, AS93]. Branch predication can then use application state such as values in general processor registers, surrounding instructions, etc.

Open systems are most successful where it is hard to infer good behavior or define a simple language for tuning. An open system can take advantage of an existing general-purpose programming language[2] to specify how a close-but-not-quite implementation can be made "just right." Further, a good meta-control interface is designed so that "injected" code only needs to worry about the most relevant details. For example, the client of a translator can control array allocation and the mapping from indicies to storage locations, but can ignore details such as the internal parser representation.

Open systems, however, can suffer some of the same problems as adjustable systems: Tuning information is expressed in terms of the abstractions visible to the client instead of implementation-level information. Further, adjustable and open systems that fail to export the "right" tuning controls are effectively as rigid as fixed systems.

Open systems also suffer some additional problems compared to adjustable systems: First, the injected code specifies not only *what* is to be done, but also *how*. Second, tuning is restricted by the client's view: the client's view may be restricted by protection and security constraints [PS83] or because some information is unavailable e.g., when the client is compiled [KEH91]. Second, the client provides code that is specific to the service's implementation, which can cause portability problems. Third, since arbitrary code can be injected in to the service, it can have arbitrary side-effects. Although this problem exists with other models, open systems increase the number of ways the client can interact with the service and, therefore, the number of ways that problems can be introduced. Finally, an open system must restrict how much internal detail is exposed. If all details are exposed, manipulating the meta-control interface is both as flexible and as complex as changing source code.

[2] Open systems can thus be sub-classified as *declaratively* or *imperitively* open, depending on the tuning language.

3.5 Incomplete

The incomplete model recognizes that a given service cannot perform the desired operations effectively and, in effect, instead provides "tools" with which the client builds the desired service. In a sense, the client executes on behalf of the service. Incomplete services often either split one operation into several, so the client can run code between the service operations, or provide a callback mechanism so the service can call user code when it needs. An incomplete implementation is similar to an open system, but the meta-control is merged with the primary interface. The programmer's view is that the client-supplied code remains client code and is not "injected" in to the service.

Incomplete array and procedure implementations could be implemented by having the client register callback procedures that are called when the translator tries to process arrays or procedures. Or, the programming language provides lower-level operations such as pointers, pointer arithmetic and jumps, and the programmer implements the various kinds of arrays and procedure calls by hand.

User-level VM pagers can be implemented by having the application register a callback routine that is invoked when the program needs to reduce its working set size. Alternatively, the application can perform memory management and I/O explicitly. An incomplete threads package implementation can provide just the basic operations to initialize threads and switch between them, leaving allocation and scheduling to the client [Kep93].

Architectures use branch delay slots to expose underlying pipeline state. On a branch, the hardware updates the program counter, but the client is responsible for updating the pipeline state [HP90].

An incomplete service provides "ultimate" flexibility. It does so by providing a simple interface[3] and then both allowing and forcing the client to implement things the service doesn't do well. Incomplete services are particularly successful when only a small part of the service is heavily dependent on the client, and when there is a small (conceptual and performance) cost for executing client code on behalf of the service. Incomplete systems improve over open interfaces by including tuning information cleanly in the primary interface. Merging the interfaces helps for two reasons: First, the interface never exposes implementation details, so client-supplied code cannot have effects that reach back in to the service. Second, all implementations use the same external interface, so implementation-dependent meta-control code is avoided.

Incomplete systems suffer several disadvantages compared to open systems. First, the client must always deal with tuning issues. Moreover, the tuning and primary interfaces are merged, which makes it hard to separate and change just one. Second, executing client code on behalf of the service may require the use of some awkward mechanism. In the worst case, it is simpler for the client to simply reimplement the service. Finally, the client always pays because the incomplete system offers a lower-level service exactly because it gets its utility *by failing to provide part of the service the client desired.* In short, an incomplete implementation solves interface problems by providing implementations only for operations that it can do well.

4 Other Issues

4.1 Client/Service Contracts

Tuning interfaces give the client a way to change some part of the runtime behavior (e.g., performance) while leaving other parts of the interface unchanged (e.g., indexing in to an array accesses the same values, whatever the implementation). Generally, then, primary interfaces provides a service and the tuning interface maintains the same primary abstraction while providing an implementation that provides the service efficiently [Kic92].

The implementation can, in fact, be tuned to such a degree that it no longer works for arbitrary inputs. However, any client that tunes a service to that degree is making an implicit contract that it will not take advantage of the changes. For example, a client may specialize an array implementation so that it works only for indicies that are small integers. The client is then promising that it will not try to use strings, floating point numbers, and so on as indices.

A related issue is that different interfaces on a given service may actually provide different services. For example, a VM implementation that signals the client on page faults can be used to implement various kinds of page-based operations such as distributed shared VM [AL91]. These operations are not always available with other VM interfaces.

4.2 Protection and Security Boundaries

Operating system kernels are a particularly tricky area for open systems because the client cannot inject arbitrary code to run e.g., at full kernel privilege. The problem is that arbitrary code could compromise security. Checking arbitrary code to ensure it is safe requires a solution to the halting problem [Hop79], and is thus impossible for general-purpose programming languages.

One solution is to provide a limited language that can be checked easily. For example, a client can pass the kernel a limited program that tells just the types of values to be moved across a communication channel [TL93]. A simpler language is easier to check but less expressive. The simplest languages are simply tuning parameters.

A second solution is to cross protection boundaries each time the injected code is invoked. However, this limits the ways that injected code can affect the service; fine-grained changes require frequent protection boundary crossings. Thus, the service and the injected code cannot be finely intermingled.

A more common approach is to use the incomplete implementation technique to push operations out of the kernel and into client space. However, the service must ensure that system security remains intact. Once an operation has been pushed out of the kernel, any of the interface structuring techniques can be applied safely. For example, to handle paging [MA90, HC92] or scheduling [MVZ91, ABLL91].

[3] The interface may itself be fixed, adaptive, adjustable, open, or incomplete.

4.3 Hybrid Interfaces

The interfaces presented in Section 2 can be combined in various ways to create hybrid interfaces. For example, many systems are both adaptive and adjustable: Compilers may inline functions using both internal heuristics and programmer directives [Sta89]; VM systems often respond both to dynamic system behavior and to hints provided by applications [Sun90]; architectures often perform branch prediction using both adaptive hardware and tuning bits embedded in the opcodes [Sit93].

4.4 Level of Representation

A problem with meta-control interfaces is that they communicate incomplete information. One issue is that the client deals in abstractions and presents tuning information in terms of those abstractions. For example, the client may communicate tuning hints to the VM system, telling it that two arrays (abstractions) are used in different ways, without being aware that the implementation uses overlapped storage for the arrays.

A second issue is that the client presents information that represents a limited view of the world. The client can provide good information about behavior that has a detailed representation in the client (although extracting a summary may be hard). However, the client has a poor view of the rest of the system. For example, the client may have good information about its own VM behavior, but can have trouble cooperating with other applications because the client does not naturally deal with information about other applications. Clients can gather better information about the rest of the system, but in doing so must perform extra work that does not contribute directly to solving the problem at hand. Finally, information about e.g., another task's VM behavior may be protected information that is available to the kernel but not to applications.

An important but somewhat subtle point is that although adaptive systems sometimes have worse information than adjustable systems, they sometimes have better information than is available to the programmer. This is because adaptive systems may be able to see details that are invisible to the programmer — because abstractions hide them or because they are not nominally things that the client needs to be concerned with. For example, a compiler may be able to perform inlining on a call-site by call-site basis, even when the programmer cannot; an operating system can adapt both to an individual program and to the implicit interaction with other programs in the system; an adaptive branch prediction scheme can take advantage of both dynamic program behavior and of details of the processor implementation.

4.5 Layered Interfaces

Abstractions make it easy to build systems by layering together components. However, the same abstractions that lead to easy layering may also ultimately lead to problems. Each layer encodes some decisions, so many layers encode many decisions. If any single layer makes a decision that is bad for some client, then the whole layered abstraction will fail for that client. Similarly, if each layer makes some compromises, the compromises can add up, leaving a system that is useless to the client.

The tradeoff is usually one of generality vs. efficiency [Lam84]. If each layer compromises some performance, the final system has many performance compromises. If each layer compromises some generality, the final system has limited usefulness.

Tunable interfaces can help, but care is needed. If the top layer of the abstraction is responsible for tuning all layers below it, then the top-level tuning interface is the union of all tuning features provided by lower-level layers. Alternatively, the top-level interface can provide an abstraction of the lower-level tuning interfaces, but with a corresponding loss of expressiveness when the client communicates through the abstracted top-level tuning interface to the lower-level implementations.

Alternatively, the client can tune individual low-level components and then instantiate the high-level service with the pre-tuned components. This approach can be effective because tuning is associated directly with the component being tuned. However, the client now must manipulate services it does not use directly, and the client does not always know how higher-level services use the lower-level components, and, thus, the client cannot always tune components appropriately.

4.6 Binding Times

Another aspect of interfaces is the time at which tuning is performed. Generally, tuning proceeds in three distinct steps: First, tuning information is bound in to the client. Second, the client passes tuning infor-

mation to the service. Finally, the service performs tuning.

For example, a client may always use some memory region in a certain way, and VM tuning information for that region is compiled in to the client when it is built. When the client starts executing, it passes hints to the VM system. Later, as the client makes use of the memory region, the VM system uses the tuning information to tune the VM behavior for that region.

In general, tuning information can be derived, communicated, and used at nearly any time. For example, adaptive systems need not wait until program runtime to take advantage of tuning information. Adaptation can take place before program runtime because code can be executed incrementally as information becomes available [Par91]. For example, arrays are a service. Array indexing must generally be delayed until runtime. However, if the compiler can deduce information about array indicies, it can perform adaptive tuning and produce optimized code at compile time. However, if basic array behavior depends on runtime data values, basic tuning information is best collected at runtime [DB77, GW78, CW86, CU91, KEH91].

Implementation details may also affect binding times. For example, it isn't generally possible to statically tune between a client and a dynamically-linked service, since different invocations of the client may use different implementations of the service. Likewise, VM services can use static information from the client, but they must also respond to dynamic interactions with other system processes.

4.7 Conflicts of Interest

When several clients share a service, or when one client uses a service several ways, the service may be presented with several conflicting tuning demands. There are three general ways of dealing with conflicts: provide a single compromised service, provide several duplicates of the service, each with a specialized implementation, or progressively transform the service to use an implementation that best suits the way it is being used at any given moment.

A compromised service can still improve over a fixed service, since there may be improvements that can be shared by all clients or all distinct uses within a single client.

Multiple implementations may cost space, and may also introduce implicit client/service contracts (§4.1) that require a client to invoke the right instance of a service, since each instance may be too specialized to deal with a generic invocation.

Progressive transformation can eliminate some client/service contracts, since there is only one instance of the service at any time. However, reconfiguration costs both the update of the service itself, and also of any auxiliary structures associated with the service.

5 Summary

Abstractions are good because they hide all but "the most important details" of a service. Yet interfaces are hard to design because each client has a different view of what is "most important." An abstraction that fails to export the right details can require clients to work around the abstraction, introducing complexities that exist soley because of the abstraction. This paper presents five models of interfaces and implementations: fixed, adaptive, adjustable, open, and incomplete. The models are presented as a continuum of choices, with no single model best; each has advantages, and each has problems. The contribution of this paper is to introduce and compare the models, showing tradeoffs between them.

Fixed systems are typically simple and are efficient if the client and service are well-matched. Adaptive systems improve over fixed ones by keeping a simple interface and also allowing for a variety of implementations. However, adaptive systems may adapt poorly and the adaptation mechanism may be expensive. Adjustable systems can eliminate the guesswork and some of the overhead of adaptive systems, but push the tuning burden on to the client, with tuning only as good as the information provided by the client. Open systems give the client a better tuning language, so good information can be communicated effectively. However, client tuning errors may have pervasive and subtle effects, and implementation details of the service are exposed in the client. Incomplete systems maintain client and service separation by moving tuning information into the primary interface. However, the merging can force clients to implement parts of the very operations that the library was supposed to provide.

6 Acknowledgements

Thanks to Gregor Kiczales for discussing these ideas and convincing me they were important enough to write down. Thanks also to Virgil Bourassa, Mike Dixon, John Lamping, Dylan McNamee, David Notkin and Kevin Sullivan for detailed discussions and reviews of earlier versions of this paper, and also Robert Bedichek and Craig Chambers for comments on earlier versions. This work was supported by NSF PYI Award #MIP-9058-439.

References

[ABLL91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP-13)*, page 95, October 1991.

[AL91] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 96–107, April 1991.

[AS93] Peter M. Athanas and Harvey F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, pages 11–18, March 1993.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[BLL88] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A System for Object-Oriented Parallel Programming. *Software–Practice and Experience*, 18(8):713–732, 1988.

[BLLW88] Brian N. Bershad, Edward D. Lazowska, Henry M. Levy, and David B. Wagner. An Open Environment for Building Parallel Programming Systems. Technical Report UWCSE 88-01-03, University of Washington, January 1988.

[CU91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. *OOPSLA '91 Proceedings; SIGPLAN Notices*, 26(11):1–15, November 1991.

[CW86] Thomas W. Christopher and Ralph W. Wallace. Compiling Optimized Array Operations at Run-Time. *APL 86 Conferance Proceedings*, pages 136–141, July 1986.

[DB77] Eric J. Van Dyke and Kenneth A. Van Bree. A Dynamic Incremental Compiler for an Interpretive Language. *Hewlett-Packard Journal*, pages 17–24, July 1977.

[DS84] Peter Deutsch and Alan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. *11th Annual Symposium on Principles of Programming Languages (POPL-11)*, pages 297–302, January 1984.

[Fat85] Richard Fateman. Personal communication, 1985.

[GW78] Leo J. Guibas and Douglas K. Wyatt. Compilation and Delayed Evaluation in APL. *Fifth Annual ACM Symposium on Principles of Programming Languages (POPL-5)*, pages 1–8, 1978.

[HC92] Kieran Harty and David R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. *Proceedings of the Fifth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 187–197, 1992.

[Hop79] John E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[HP90] John L. Hennesy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[JF91] Charles Johnson and David L. Fox. The Silicon Palimpsest – A Programming Model for Electrically Reconfigurable Processors. Proceedings of the Second and Third Annual SigFORTH Workshops March 1991.

[KdRB82] Gregor Kiczales, Jim des Riveres and Daniel G. Bobrow. The Art of the Metaobject Protocol. The MIT Press, 1991.

[KEH91] David Keppel, Susan J. Eggers, and Robert R. Henry. A Case for Runtime Code Generation. Technical Report UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, November 1991.

[Kep93] David Keppel. QuickThreads: A Threads Building Core. Technical Report UWCSE 93-05-06, University of Washington, May 1993.

[Kic92] Gregor Kiczales. Towards a New Model of Abstraction in Software Engineering. *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pages 1–11, November 1992.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[Lam84] Butler W. Lampson. Hints for Computer System Design. *IEEE Software*, 1(1):11–28, January 1984.

[Rod91] Luis H. Rodriguez, Jr. Coarse-Grained Parallelism using Metaobject Protocols. Master's thesis, Massachusetts Institute of Technology, 1991.

[MA90] Dylan McNamee and Katherine Armstrong. Extending the Mach External Pager Interface to Allow User-Level Page Replacement Policies. Technical Report UWCSE 90-09-05, University of Washington, September 1990.

[MVZ91] Cathy McCann, Raj Vaswani, and John Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors. Technical Report UWCSE 90-03-02, University of Washington, March 1990 (Revised February 1991).

[Par72] David L. Parnas. On The Criteria To Be Used in Decomposing Systems Into Modules. *Communications of the ACM*, 5(12), December 1972.

[Par79] David L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(2), March 1979.

[Par91] Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation. *SIGPLAN Notices*, 26(9), September 1991.

[PS83] J. Peterson and A. Silberschatz. *Operating System Concepts*. Addison Wesley, 1983.

[Sit93] Richard L. Sites. Alpha AXP Architecture. *Communications of the ACM (CACM)*, 36(2), February 1993.

[Smi84] Brian Cantwell Smith. Reflection and semantics in Lisp. *11th Annual ACM Symposium on Principles of Programming Languages (POPL-11)*, page 23, January 1984.

[Sta89] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Cambridge, Massachusetts, 12 September 1989.

[Sun90] SunOS Reference Manual. Technical Report Part Number 800-1751-10, Sun Microsystems, 2550 Garcia Avenue, Mountain View, California, 1990.

[Sut91] Ivan Sutherland. Personal Communciation, March 1991.

[TL93] Chandramohan A. Thekkath and Henry M. Levy. Limits to Low Latency Communication on High-Speed Networks. *ACM Transactions on Computer Systems*, 11(2), May 1993.

[YP92] Tse-Yu Yeh and Yale N. Patt. A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. *Proceedings of Micro-25 (0-8186-3175-9/92, IEEE)*, 1992.