Appears in IMSA'92 Proceedings (Workshop on Reflection and Meta-level Architectures).

Towards a New Model of Abstraction in the Engineering of Software

Gregor Kiczales
Xerox Palo Alto Research Center*

We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for... [The mathematician] need not be idle; there are many operations he can carry out with these symbols, without ever having to look at the things they stand for.
Hermann Weyl, "The Mathematical Way of Thinking"
(This appears at the beginning of the *Building Abstractions With Data* chapter of "Structure and Interpretation of Computer Programs" by Harold Abelson and Gerald Jay Sussman.)

This is an abridged version of a longer paper in preparation. The eventual goal is to present, to those outside of the reflection and meta-level architectures community, the intuitions surrounding open implementations and the use of meta-level architectures, particularly metaobject protocols, to achieve them.

The view of abstraction on which software engineering is based does not support the reality of practice: it suggests that abstractions hide their implementation, whereas the evidence is that this is not generally possible. This discrepancy between our basic conceptual foundations and practice appears to be at the heart of a number of portability and complexity problems.

Work on metaobject protocols suggests a new view, in which abstractions do expose their implementations, but do so in a way that makes a principled division between the functionality they provide and the underlying implementation. By resolving the discrepancy with practice, this new view appears to lead to simpler programs. It also has the potential to resolve important outstanding problems surround reuse, software building blocks, and high-level programming languages.

Abstraction In Action

I want to start by talking about the current view of abstraction in software engineering: how we use it, what the principles are, what the terminology is and what it does for us. Rather than attempting any sort of formal definition, I will just use an example. I will talk about the implementation of a familiar system, using familiar terms of abstraction, with the goal of getting the terminology I am going to use out on the table.

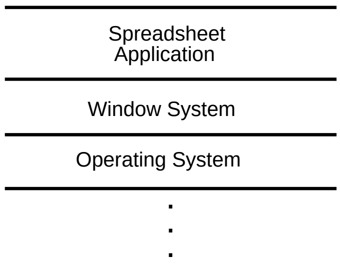


Figure 1: The layers of abstraction in the display portion of a spreadsheet application.

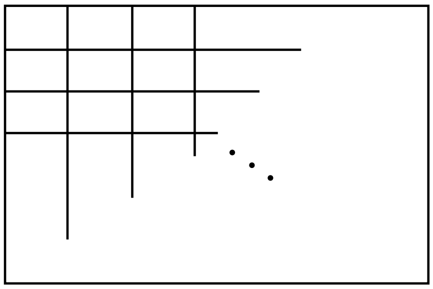
Consider the display portion of a spreadsheet application. In practice, the implementation would be based on "layers of abstraction" as shown in Figure 1. The spreadsheet would be implemented on top of a window system, which would in turn be implemented on top of an operating system and so on down (not very far) to the machine.

The horizontal lines in the figure are commonly called "abstraction barriers," "abstractions" or "interfaces." Each provides useful functionality while hiding "implementation details" from the client above.[1] To the degree that an abstraction provides powerful, composable functionality, and is free of implementation issues, we call it "clean" or "elegant." In the particular case of the window system, the abstraction would provide the ability to make windows, arrange them on the screen, display in them, track the mouse etc. Issues such as how the windows are represented in memory and how the mouse is tracked would be hidden as implementation details.

There seem to be (at least) three basic principles underlying our view of abstraction:

*3333 Coyote Hill Rd., Palo Alto, CA 94304; (415)812-4888; Gregor@parc.xerox.com.

[1] In this paper, the terms *client* and *application* are used to refer to a piece of software that makes use of some lower-level software; i.e. the spreadsheet is a client of the window system.



```
for i = 1 to 100
 for j = 1 to 100
  mkwindow(100, 100, i*100, j*100)
 end
end
```

Figure 2: A spreadsheet looks like a rectangular array of cells. The simplest way to implement it is to use one window for each cell.

- The first, and most important, has to do with management of complexity. In this sense, abstraction is a primary concept in all engineering disciplines and is, in fact, a basic property of how people approach the world. We simply can't cope with the full complexity of what goes on around us, so we have to find models or approximations that capture the salient features we need to address at a given time, and gloss over issues not of immediate concern.
- Second, is a convention that a primary place to draw an abstraction boundary is between those aspects of a system's behavior that are particular to a particular implementation vs. those aspects of its behavior that common across all implementations.
- Third, is a sense that not only is the kind of abstraction boundary that arises from the second principle useful, it is in fact the *only* one it appropriate to give to clients. That is, we believe that issues of an interface's implementation are not of concern to, and should be completely hidden from, clients.

(Note that the first of these is so basic that it rarely, at least in our field, gets explicit attention. But arguably, what our informal notions of elegance, cleanliness, and orthogonality are about is the degree to which an abstraction includes those issues which are important without including any that are not.)

Layered on top of these three principles are our goals of portability, reusability and in fact the whole concept of system software. The idea has been that by taking commonly useful, "basement-level," functionality — memory allocators, file systems, window systems, databases, programming languages etc. — giving it a general-purpose interface, and isolating the client from the implementation, we could make it possible for a wide range of clients to use the abstraction without caring about the implementation. Portability stems in particular from isolating the client from implementation details; this makes it possible to have other implementations of the abstraction which the client code can be ported to. Reuse stems in particular from making the abstraction general-purpose; the more general it is, the wider a variety of clients that can use it.

In line with this story, it should be an easy matter to implement a spreadsheet on top of a clean, powerful window system. What is needed is just a rectangular array of cells; we need to be able to display and type in each cell independently; and we need to know when the mouse is clicked over a cell. Since this is exactly the functionality a window system provides, the simplest way to code the spreadsheet is to use one window for each cell. This takes advantage of the high-level window system abstraction to cleanly express what is desired, and makes maximal reuse of the existing window system code. A program written in this fashion is shown in Figure 2.

This is abstraction at its best. The code is simple, clear, and we can read it without having to know anything about the inner workings of the underlying implementation. Abstraction here is doing just what our small minds need: making it possible for us to think about important properties of our program — its behavior — without having to think about the entirety of the machinations the underlying hardware is having to perform to get it to run.

As wonderful as this may sound, few experienced programmers would be surprised if this code didn't quite work. That is, it might work, but its performance might be so bad as to render it, in any practical sense, worthless. This can happen if the window system implementation is not tuned for this kind of use. As part of writing the window system, the implementor is faced with a number of tradeoffs, in the face of which they must make decisions. No matter what they do, the window system will end up tuned for some applications and against others. In this case, the implementor might have assumed that 25 to 50 windows was a more typical number for an application to use than $10,000$. They might also have assumed that the typical configuration of windows would have an irregular, rather than highly-regularized, geometry. Implementation decisions based on these assumptions, once made, become locked away behind the abstraction barrier as implementation details.

We are all familiar with this sort of situation, and probably have a good sense of how we would respond. But, stepping back and looking through it carefully is fruitful. There are several points to notice: (i) While the simple program in Figure 2 may not perform adequately, its intended behavior is perfectly clear. In other words, the window system abstraction itself is adequate for expressing the behavior the client programmer is after. (ii) The fact that the implementation will fail to provide adequate performance is nowhere evident in the client code. That is, the window system abstraction is not, in and of itself, betraying these properties of the implementation. (It's also likely to be the case that this performance property can't be gleaned from reading the window system documentation.) (iii) So, predicting and/or understanding the performance properties of this program can only be done with knowledge of internal aspects of the window system implementation — the so-called "hidden implementation details." (iii) Finally, it is relatively easy to imagine an implementation of the window system in which this code would perform adequately. Moreover, such an implementation might not be all that different from the existing one.[2]

What is clear then is that there is a basic discrepancy between our existing view of abstraction and the reality of day-to-day programming. We say that we design clean, powerful abstractions that hide their implementation, and then use those abstractions, without thinking about their implementation, to build higher-level functionality. But, the reality is that the implementation cannot always be hidden, its performance characteristics can show through in important ways. In fact, the client programmer is well aware of them, and is limited by them just as they are by the abstraction itself.

Looking ahead, the idea underlying the new abstraction framework will be to try and preserve what is good and essential about our existing abstraction framework — essentially the first two bulleted principles — while seeking to address the conflict between the third basic principle and the reality of practice. In doing this, the strategy will be to try and take advantage of the fact that very often, as in this example, our abstractions themselves are sufficiently expressive and our implementations may only be deficient in small ways. What we will end up doing is "opening up the implementation," but doing so in a principled way, so that the client doesn't have to be confronted with implementation issues all the time, and, moreover, can address some implementation issues without having to address them all.

Outline of the Paper

The rest of this paper expands this basic argument for open implementations. First, the consequences of the deficiency in our current abstraction framework are discussed, using both the window system example and an example from high-level programming languages. The application of metaobject protocol technology to these problems is discussed, and the new model of abstraction, drawn out from the intuitions underlying the metaobject protocol work, is presented. Given the new model, it is possible to identify a wide range of other work in the software engineering community which not only seems to confirm the intuition that the old model of abstraction is invalid but which in fact seems to be headed in the same direction as the framework presented in this paper. Finally there is a discussion of what future work might be required as part of continuing to develop this new abstraction framework.

[2] The issue is whether a window is a large structure, which locally caches derived properties, or whether it is a small structure, which continually recomputes derived properties from its parent (i.e. does a window know its position, or does it have to ask its parent). In the latter approach, a great deal of memory could be saved on the cell windows. Each could be as small as a word, or even take up no storage at all in more radical architectures. In addition to saving memory, certain operations could be supported more efficiently. For example, to tell which cell the mouse was over, the main window could, because of the regular geometry of the cells, do simple arithmetic rather than using the more general mechanism of polling all the cell windows.

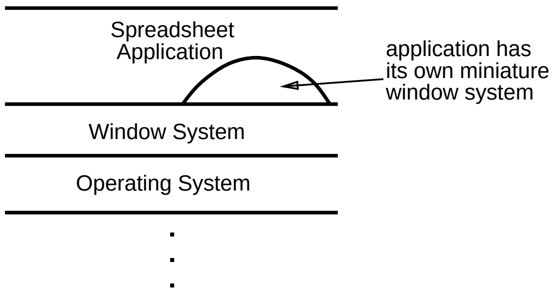


Figure 3: The spreadsheet application after being revised around the performance problems of the window system. The reimplementation of functionality which could not be reused from the window system appears as a 'hematoma' in the application. Each such hematoma increases the size of the application. In addition, the rest of the application can get more complex when it is rewritten to use the new functionality.

The Origins of Complexity and Portability Problems

Cases like the spreadsheet application, where an abstraction itself is adequate for the client's needs but the implementation shows through and is in some way deficient are common. The machinations the client programmer is forced into by these situations make their code more complex and less portable. These machinations fall into two general categories: (i) Reimplementation of the required functionality, in the application itself, with more appropriate performance tradeoffs; and (ii) coding "between the lines."

Reimplementation of functionality is what would mostly likely happen in this case. The spreadsheet programmer would end up writing their own "little window system," that could draw boxes on the screen, display in them, and handle mouse events. Reimplementing this way would allow the the programmer to ensure that the performance properties met their particular needs. As suggested by Figure 3, reimplementing part of the underlying functionality this way increases the size of the application, and, therefore, the total amount of code the programmer must be responsible for.

In addition to making the application strictly larger, reimplementation of underlying functionality can also cause the rest of the application — the code that simply uses the reimplemented functionlity — to become more complex. This happens if for some reason the newly implemented functionality cannot be used as elegantly as the original underlying functionality. This in turn can happen if, for any reason, the programmer cannout manage to slide the new implementation in under the old interface.

Once the programmer is forced away from being able to use the old interface, and into the problem of designing one of their own, its quite likely they won't do as good a job. Simply put, the application programmer doesn't have the time (even if they do have the interest) to design the new interface as cleanly as might be nice.

(As an aside, its worth point out that even if the interface ends up being just as (or more) elegant, one of the primary purposes of high-level standardization — to be able to easily read each other's code — has been defeated.)

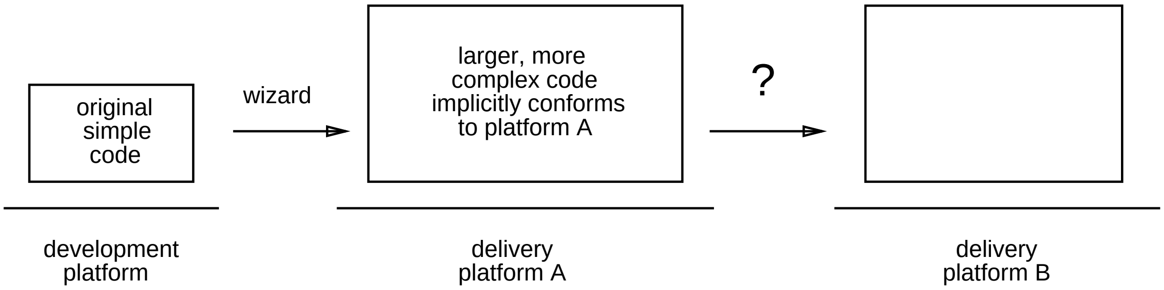


Figure 4: When an application is originally written on a fast machine, the code can start out being simple. To port the code to a delivery platform a wizard — someone who understands the inner workings of the delivery platform — is brought in to tune the code. The application gets larger and more complex, and above all it becomes implicitly adapted to the delivery platform. It is then even more difficult to move it to another platform.

Coding between the lines is happens when the application programmer writes their code in a particularly contorted way in order to get better performance. A classic example is in the use of virtual memory. In a program that allocates a number of objects, there is often a order to allocating those objects that is "natural" to the program. But, if there get to be a lot of objects, and paging behavior becomes critical, people will often rewrite the application to "allocate the objects close to each other" and thereby get better performance. This is coding between the lines because although the documented virtual memory abstraction makes no mention about the physical locality of objects, the programmer manages to contort their code enough to "speak to" the inside of the implementation and get the performance they want.

When programmers are forced into these situations, their applications become unduly complex and, more importantly, even less portable. It is easiest to see how this happens by starting with a hypothetical prototype implementation, coded on a machine that was fast enough that the programmer was not forced into these sins, and then looking at what happens as the application is moved to a delivery platform. (In reality, code is usually "optimized" when it is first written, but this simpler case makes what happens more clear.)

The original implementation is simple, clear and makes the greatest re-use of the underlying abstractions (i.e. the simple spreadsheet implementation). But, when it comes time to move it to the delivery platform, a number of performance problems come up that must be solved. A wizard is brought in, and through tricks like those mentioned above, manages to improve the performance of the application. Effectively, the wizard *convolves* the original simple code with their knowledge of inner workings of the delivery platform.[3] (The term convolves is chosen to suggest that, as a result of the convolution, properties of the code which had been well localized become duplicated and spread out.) In the process, the code becomes more complex and *implicitly* conformant to the delivery platform. When it comes time to move it to another platform, the code is more difficult to work with, and because of the implicit conformance, it is difficult to tell just why things are the way they are. This is shown in Figure 4.

[3] Note that putting it this way explains why the informal term "wizard" refers to someone who not only is good at working with a given abstraction (i.e. a window system), but who is also intimately familiar with the *inner workings* of the implementation. Simply put, the wizard is someone who specializes at doing what our traditional abstraction story says should never happen.

High-Level Languages

> I found a large number of programs perform poorly because of the language's tendency to hide "what is going on" with the misguided intention of "not bothering the programmer with details."
> *N. Wirth, "On the Design of Programming Languages," [29]*

I want to look next at the domain of high-level programming languages, where the reflection and meta-level architectures community has done a lot of work to address these kinds of problems. First, I will show, using the Common Lisp Object System (CLOS) [10, 25], how the same sorts of problems can come up. I will then show how those problems are addressed by the CLOS Metaobject Protocol (CLOS MOP) [4, 11, 12]. From there, it will be possible to generalize and present the new model of abstraction.

Consider the following CLOS class definitions:

```
(defclass position ()
     (x y))

(defclass person ()
     (name age address ...))
```

The class `position` might be part of a graphics application, where the instances are used to represent the position of the mouse as it moves. The class defines two slots, `x` and `y`.[4] The behavior of the application is such that there will be a very large number of instances, both slots will be used in every instance and access to those slots should be as fast as possible.

The second definition, `person`, might come from a knowledge representation system, where the instances are being used as frames. In this case, the class defines a thousand slots, corresponding to the many properties of people which might be known. As with the class `position`, the behavior

[4] *Slot* is the CLOS term for the fields of an instance.

of the application means that a couple of things are known: there will be a very large number of instances; but in any given instance only a few slots will actually be used.

Clearly, the ideal instance implementation strategy is different for the two classes. For `position`, an array-like strategy would be ideal; it provides compact storage of instances, and rapid access to the `x` and `y` slots. For `person`, a hash-table like strategy would be more appropriate, since it isn't worth allocating space for a slot until it is known that it will be used. This makes access slower, but it is a worthwhile tradeoff given a large number of instances.

What is most likely to be the case, in a run-of-the-mill CLOS implementation sans MOP,[5] is that the implementor will have chosen the array-like strategy. The prospective author of the `person` class will find themselves in a situation very much like that of the spreadsheet implementor above: While the CLOS language abstraction itself is perfectly adequate to express the behavior they desire, supposedly hidden properties of the implementation — the instance representation strategy — are critically getting in the way.

Metaobject Protocols

In this abridged version of the paper, this section is elided, since it would be redundant for IMSA'92 Workshop attendees.

For the eventual audience of this paper, the goal of this section will be to sketch the mechanics of metaobject protocols, and to show how, by careful design, a metaobject protocol can be used to allow the user to control critical aspects of the language implementation strategy, without overwhelming them with what truely are implementation details.

This section will also discuss, more briefly, how metaobjects protocols can be used to provide the user control over the semantics, or behavior of a language.

In addition to the CLOS Metaobject Protocol, other MOPs and reflective languages which might be discussed in this section include TELOS [18], ABCL/R2 [17], 3-KRS[16], Anibus [21, 22], Sartor [2] and Ploy [26].

A New Model of Abstraction

In the metaobject protocol approach, the client ends up writing two programs: a base-language program and an (optional) meta-language program. The base-language program expresses, the desired behavior of the client program, in terms of the functionality provided by the underlying system. The meta-language program can customize particular aspects of the underlying system's implementation so that it better meets the needs of the base-language program.

What begins to emerge is a "dual-interface" picture something like that shown in Figure 5. A high-level system (i.e. CLOS) presents two coupled interfaces: base- and meta-level. The base-level interface looks like the traditional interface any such system would present. It provides access to the system's functionality in a way that the application programmer can make productive use of and which does not betray implementation issues. The client programmer can work with it without having to think about the underlying implementation details.

[5] At this point all CLOS vendors I know of have plans to provide a metaobject protocol. So, a CLOS implementation sans MOP is more of a rhetorical tool than a reality.

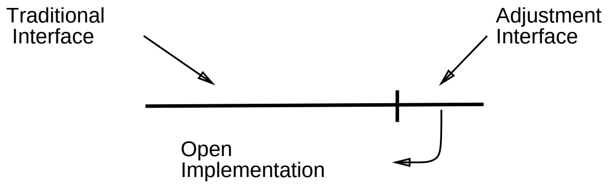


Figure 5: The dual-interface framework supports the notion of an open implementation. The client first writes a base-program, and then, if necessary, writes a meta-program to customize the underlying implementation to meet the base-program's needs. The curved arrow under the meta-level interface is intended to remind us that it provides access to what have traditionally been internal properties of the implementation.

But, for those cases where the underlying implementation is not adequate, the client has a more reasonable recourse. The meta-level interface provides them with the control they need to step in and customize the implementation to better suit their needs. That is, by owning up to the fact that users needs access to implementation issues (i.e. instance implementation strategy), and providing an explicit interface for doing so, the metaobject protocol approach manages to retain what is good about the first two principles of abstraction.

It is much too early to attempt to provide a complete account of dual interface abstractions, how to design them, how to use them or what technologies can be used to support them. But, based on experience with metaobject protocols and other recent reflective and meta-level architectures, some basic comments can be made.

First off, it appears that the design of base-level interfaces can be done using existing skills. As mentioned above, we have become quite good at designing interfaces that do not themselves betray the implementation. We should be able to make base-level interfaces even more clean because we will now have a principled place to put implementation issues that the client must have access to — the meta-level interface.

Mastering the design of meta-level interfaces, and, importantly, the coupling between base- and meta-level interfaces is going to take a great deal more work. But we can enumerate four preliminary, and closely interrelated, design principles: scope control, conceptual separation, incrementality and robustness.

- *Scope control* means that when the programmer uses the meta-level interface to customize the implementation, they should be given appropriate control over the scope of the specialization. One can imagine various kinds of scope control. In the CLOS example above, the programmer wants to be able to say both that they only want to affect the instance representation strategy, and that only want certain classes (i.e. `person`) to be affected. Other classes, particularly classes that are part of other applications, should not be affected. The window system case is analogous; some windows should use the implementation tuned for spreadsheets whereas others should use the default implementation.
- *Conceptual separation* means that it should be possible

to use the meta-level interface to customize particular aspects of the implementation without having to understand the entire meta-level interface. So, for example, the client programmer who wants to customize the instance implementation strategy shouldn't also have to be concerned with the method dispatch mechanism. This of course is difficult, since implementation issues can sometimes have surprisingly far-reaching effects. The challenge, as discussed in [14], is to come up with a sufficiently fine-grained model of the implementation.

- *Incrementality* means that the client who decides to customize some aspect of the implementation tradeoffs wants to do just that: *customize those properties.* They don't want to have to take total responsibility for the implementation and they don't want to end up having to write a whole new implementation from scratch. It must be possible for them to say just what it is they want to have be different, and then automatically reuse the rest of the implementation. This is the salient difference between the more recent reflective systems (CommonLoops, 3-KRS and beyond) and the original 3-Lisp work: by using object-oriented techniques, it has been possible to support the incremental definition of new implementations (interpreters, runtimes etc.) using subclass specialization. (More is said about object-oriented techniques later in the paper.)
- *Robustness*[6] simply means that bugs in a client's metaprogram should have appropriately limited effect on the rest of the system. To date, much of the work in the reflection and metaobject protocols community has provided only limited robustness, either by checking the results of functional protocols, or absorbing it from the underlying runtime in imperative[7] protocols. But these approaches significantly restrict the power of the protocol. In more recent work, we are beginning to explore the use of more declarative protocols, combined with partial evaluation techniques to recover the performance loss [2]. This remains a major open problem.

These four principles are not entirely orthogonal. Take for example, support for defining a new instance implementation strategy in the CLOS MOP. While it is easy to say that it does well on each of the first three, it is difficult to point to particular parts of the CLOS Metaobject Protocol design and say "Scope control comes from here and incrementality comes from here." Instead, they all seem to be intertwined; they all have to do with various kinds of "locality."

In fact, much of the recent work on reflective systems can be seen as experiments with locality. Group-wide reflection, one metaobject per object languages, metaobjects on a per-class basis, reifying the generic function rather than letting the class handle method dispatch — all of these provide different kinds of locality control [6, 8, 9, 16, 17, 27, 28] (as well as many of the other papers appearing in this workshop). What is clear is that there is no one right or most elegant metaobject structure, each has relative costs and advantages, and we need to keep experimenting to learn about how to handle locality this way. (There is more to say about the subject of locality as the paper progresses.)

It is also possible to make a basic comment about the way the designer of a dual-interface abstraction — or any open implementation — works: iteratively. They start with a traditional abstraction (i.e. a window system or CLOS), and gradually add a meta-level interface as it becomes clear what kinds of ways a close implementation can cause problems for the users. Moreover, it isn't a good idea to try and make the first version of a new kind of system open in this sense. Opening the implementation critically depends on understanding not just one implementation the clients might want, but also the various kinds of variability around that point they might want. In this mode of working, user bug-reports and complaints about previous versions of the system take on an important value. We can look for places where users complained that they wanted to do X, but the implementation didn't support it; the idea is to add enough control in the meta-level interface to make it possible to customize the implementation enough to make X viable. (In fact, in work on the CLOS Metaobject Protocol, we spent a lot of time thinking about these kinds of bug reports.)

Another way of thinking about the design of meta-level interfaces can be found in a 1980 paper by Mary Shaw and Wm. Wulf [24], in which they present an interesting (and prescient) intuition about the situation: "Traditionally, the designers and implementors of programming languages have made a number of decisions about the nature and representation of language features that... are unnecessarily preemptive." By preemptive, they mean a decision, on the part of the implementor (or the language designer), that preempts the programmer from being able to use a language feature in a way that otherwise appears natural. (A specific example they give has to do with the choice of representation of arrays.) Their paper is focused primarily on programming language implementations, but the notion of preemption is a powerful one to work with when thinking about any kind of meta-level interface. It suggests that anytime we find ourselves saying "well, I'll implement this feature a particular way because I think *most* users will do X," we should immediately think about the other users, the ones whose options we are about to preempt, and how, using a meta-level interface, we might allow them to customize things so they can do other than X.

A Recap

At this point, it is possible to give a capsule summary of the argument so far:

> In practice, high-level abstractions often cannot hide their implementations — the performance characteristics show through, the user is aware of them, and would be well-served by being able to control them. This happens because making any concrete implementation of a high-level system requires coming to terms with a number of tradeoffs. It simply isn't possible to provide a single, fixed, closed implementation of such a system that is "good enough" that all prospective users will be happy with it. In other words, the third principle of abstraction presented above appears to be invalid, at least in actual practice.

[6] This term is somewhat problematic, as it has particular technical meaning in some communities. Later in the paper, it will become clear that what is needed is a term that in some sense spans (at least) all of safety, reliability and security.

[7] In [12] we used the term *procedural* instead of *imperative*.

> Work on metaobject protocols and other meta-level architectures suggests a new abstraction framework that better addresses the need for open implementations. Under this framework the abstraction presented by a system is divided into two parts: one that provides functionality in a traditional way and another that provides control over the internal implementation strategies supporting that functionality. This approach retains the first two principles of the old abstraction framework, dropping only the third.

Looking At Other Work

With this summarization in mind, it becomes possible to look for other areas where open implementations and dual interface abstractions could be particularly advantageous. In doing so, what we are trying to assess is how much of the argument presented above applies in domains other than high-level programming languages. Clearly we would expect the basic argument for open implementations to move across — after all, we started with a window-system not a programming language. On the other hand, we may or may not expect the concept of metaobject protocols (or at least our current notion of them) to move to memory systems or schedulers. And in between those two levels are the crucial intermediary notions of locality, reflection, meta, and object-oriented programming. By looking at other examples, we hope to get a better sense of the overall picture and where each of these important concepts fits in it.

We are looking for systems of more than modest functionality, yet where performance is an issue. The whole category of system software — operating systems, window systems, database systems, RPC mechanisms etc. — is a natural place to look. The abstractions have been well-honed over the years, there is tremendous understanding of the different kinds of implementation strategies that can be useful and, because these systems underlie everything else we build, the potential payoff of increased understanding of their nature is large.

It turns out that not only does work in these areas appear to support the basic argument for open implementations, but in fact there appears to be a lot of work already going on that is driving in similar directions.

Programming Languages

A number of programming language projects have discovered that attempting to give their users a black-box abstraction with a single fixed implementation does not work. In some sense, compiler pragmas were the first example of this — they can be thought of as open implementations with a "declarative" meta-level interface.

In Hermes [Hermes book], several of the built-in data structures come with a small collection of different implementations. This, like pragmas, is a step in the direction of open implementations — several implementations is after all more than one, and letting the user choose is a step in the direction of openness. But, it does not completely solve the problem because there is no reason to believe that some prospective users will not want an implementation that is different from any of the ones provided. The designers of Hermes are aware of this limitation, it is just that their concern for robustness (safety in particular) has so far prevented them from adopting the more powerful reflective or metaobject protocol techniques [Yemeni, private conversation]. One possibility might be to add an internal metaobject protocol, which the designers could use to quickly provide clients with newly requested implementations, but which would not be documented to normal users.

As discussed by Rodriguez [22], the same sort of situation can be seen in languages for parallel programming. A key problem in this domain is that a compiler that attempts to automatically choose program's parallelization is often unable to do so optimally. Having recognized this problem, this community has developed architectures that allow the programmer to step in, in various ways, and direct the parallelization [3, 5, 7, 13, 15, 30]. These systems bear varying degrees of resemblance to explicit meta-level architectures, with one key difference being that they have not (yet) adopted the use of object-oriented techniques to organize the meta-level.

At least one language has gone farther, to have what is clearly a metaobject protocol, the only difference being that they don't use the terminology we do. Joshua is a rule-based inference system developed at Symbolics [23]. Because Joshua is such a high-level language, its default implementation can perform quite poorly on some examples. By allowing the client to step in and customize the inference mechanism to better suit the particular example, they sometimes get substantial performance improvements [Shrobe, private conversation].

Operating Systems

The operating system community long ago began to push up against the boundaries of the traditional black-box abstraction framework. Very early on, virtual memory systems provided limited meta-level interfaces that allowed clients to influence what page-replacement strategy was used (i.e. the Unix `madvise` facility). More recently, there has been a move, starting with systems like the Mach external pager, from the declarative approach to an approach more like that of metaobject protocols. Specifically, they are using object-oriented and imperative techniques to organize the meta-level.

Using this more powerful imperative approach, there has been similar work opening up thread packages and load-balancing mechanisms [1]. In fact, people associated with this work have, more recently, been explicitly questioning the validity of the traditional closed-implementation notion of system software in many of the same ways discussed in this paper.[Anderson, talk at PARC] (Within the reflection community, there is of course the Muse work at Sony, which has been explicitly addressing these issues for some time [31].)

In the operating system community, where there is a great deal of emphasis on reliability, the architectures have been interestingly different than in the metaobject protocol community. They have done a much better job of achieving robustness. The various efforts at reducing the size of the kernel are largely driven by a desire to make as much of the traditional operating system functionality user-replaceable. On the other hand, even though there is no apparent trade-off between robustness and incrementality, they have done much less well at providing incrementality.

Other Systems

Looking at other kinds of systems software turns up similar kinds of work, although perhaps not as aggressively open as in the operating system community. There are interesting things to be said about databases, RPC mechanisms and document processing systems. In fact, the spreadsheet example presented in this paper was drawn from work at PARC which explicitly addressed the applicability of metaobject protocol ideas to the window system domain [19, 20].

Future Work

Changing something as fundamental as our underlying conception of abstraction is not going to be a small task. All of our current design principles, conventions, tools, techniques, documentation principles, programming languages and more rest on the more fundamental notion of abstraction. This section provides a short sampling of what might need to be done, ranging from the relatively straightforward — assuring ourselves that the need for open implementations and a corresponding revision of our abstraction framework is in fact genuine — to the more far reaching — working out the ramifications of this revision, and what it will take to get it to work.

Much of what needs to be done involves looking at basic concepts in software engineering practice, to see how they depend on the old model of abstraction and how they might need to be revised. This includes issues like portability, software building blocks and top-down programming.

Complexity and Portability Revisited

A primary issue to be addressed has to do with what the consequences of the open implementation argument is for portability and complexity. One of the comments I often hear, when I talk about the metaobject protocol work, is that opening up implementations in this way will cause client code to be more complex and create portability problems. The goal is of course very much the opposite: to make code simpler and improve portability. But, these ideas makes people nervous; it is important that the meta-level architectures community be able to address their concerns carefully.

The criticism from skeptics is: (i) You are allowing the client to muck with implementation issues that used to be hidden. (ii) This will result in code that is more complex, and wedded to features specific to the implementation. (iii) This will make the code more difficult to work with and less portable.

The counterargument is: (i) Clients already are aware of the implementation issues, it is just that we have been trying to pretend that wasn't the case. That is the whole thrust of the first part of the paper. (ii) We believe that client code will be simpler, because it will be able to reuse more of the underlying functionality. There won't be hematomas and other complexities that currently result from performance problems in the library functionality. It is also important to understand that the meta-level interface is not implementation-specific. It applies to all implementations of the system. What is implementation-specific is the default implementation. So, the meta-program, since it is a customization of the default implementation, may end up depending on properties of the implementation for which it is written but: (a) programs already are implementation-specific; (b) in the new framework this dependence will be more explicit since it will be isolated to the meta-program; and (c) if there is less code to work with it will be easier to work with no matter what.

Higher-Level Building Blocks

The concept of open implementations has significant ramifications on our concepts of what kinds of building blocks it might be possible to work with in the future. Learning how to make clean, powerful open implementations should result in being able to build and work with higher-level building blocks, which should in turn result in simpler application programs. This expectation is based on the belief that what has kept us from being able to successfully develop very high level libraries has been our inability to provide (closed) implementations that pleased enough users.

The programming language domain is perhaps the place where it is most clear that a large part of what has kept us at a low-level is the closed implementation framework. High-level languages have enjoyed limited success in large part due to performance problems. We haven't been able to get good enough performance out of higher-level languages because we haven't been able to write compilers that are "smart enough" to satisfy all the users. But, the open implementation idea fundamentally acknowledges that if a language is more than modestly high-level, it simply isn't possible to build a closed compiler that is smart enough. We must instead open the compiler up so that the programmer, who knows a great deal about how they want their program to be compiled, can step in and help.

This restraining force on high-level languages is particularly evident in the earlier quote from Wirth. Essentially, his argument is that since it isn't possible to properly implement high-level functionality (using a closed implementation), the language should be restricted to providing only low-level functionality. The question now is whether open implementations and the dual interface abstraction framework make it possible to make truly high-level languages with good performance. Experiments need to be done with a variety of such languages.

Top Down Programming vs. Reuse

In the previously mentioned paper by Shaw & Wulf they make the claim that top-down programming is fundamentally at odds with reusable code libraries and even the notion of system software. Their argument, as I understand it, is that a reusable library essentially blocks, at the abstraction boundary, the downward flow of design decisions, preventing those decisions from leaking into the library's implementation as we would like.

Their argument is essentially compatible with the one presented in this paper. From the dual interface abstraction point of view, the conflict is not between top-down programming and reusable code; it is between top down programming and *closed implementations* of reusable code. This leads to another way of thinking about open implementations, complementary to the dual interface model. The idea is that reusable code should be like a sponge: It provides basic functionality (the base-level interface), basic structure (the default implementation) but also allows the user to "pour in" important customizations from above to "firm it up."

Work needs to be done to go back and look at top-down programming and the conflict Shaw & Wulf mention to see how it informs the open implementation and dual interface abstraction frameworks.

Multiple Open Layers

This view of top-down programming makes it clear that opening an implementation only to the client immediately above is not enough. We need to do better than that; all layers need to be open to all layers above them. So, for example, when an application is written on top of a high-level language, which itself sits on top of a virtual memory system, the application code needs to be able to control not just how the language uses the memory it is allocated, but also how that virtual memory system allocates that memory.

Work needs to be done to develop this ability to push down, through multiple levels of abstraction this way.

Open Behavior

The discussion in this paper begins to provide an explanation of part of the problem metaobject protocols are solving — specifically, the need for open implementations. But a clear lesson from the metaobject protocol work is that users can also take productive advantage of being able to customize the *semantics* (or behavior) of systems they are building on top of.

Work needs to be done to integrate the need for open behavior, and the way that meta-level architectures provide it, into the argument presented in this paper and into any new abstraction framework that is developed.

Mastering Locality

The dual interface framework is similar to the way in which one might expect the conversation between the human provider and client of a system to talk. Much of the time they would just talk about the functionality that would be provided. At other times they would "go meta" and talk about how the functionality was going to be used and crucial performance issues.

And it is by making this analogy with the discussion between humans that we can get some insight into the problems that we will face in really trying to get this to work: very often, the concepts that are most natural to use at the meta-level cross-cut those provided at the base-level. What it seems we want to be able to do is to allow the user to use natural base-level concepts *and* natural meta-level concepts — as if they were the x and y axes of a plane — to get at just what it is in the implementation they want to affect. The problem is that the "points" in the plane spanned by these two axes are not necessarily easy to localize in an implementation.

Take, as an example, the user of a Lisp-like language who wants to control the tagging strategy for certain objects within a certain part of their program. It's quite natural for them to say something like: "Use immediate tagging for fixnums and positions, tag rectangles and lines in the pointer, and tag everything else in the actual object representations." But, it would be surprising to find an existing compiler in which making this change was easy, much less one that could be persuaded to have just part of a program work this way. (Getting such a compiler architecture is the thrust of the work reported in [14].)

We are, in essence, trying to find a way to provide two effective[8] views of a system through cross-cutting "localities." Getting this to work, in the general case, appears to be quite difficult; aside from crystallizing it as a problem, there isn't much to say about it at this time.

One strategy — the one that has been prevalent in existing meta-level architectures — is to make the problem easier by delaying the implementation of strategy selection until run-time or thereabouts. So, for example, the existing metaobject protocols address only those issues which do not need to be handled in a compile-time fashion. The various systems that address distribution, concurrency and real-time [other papers in this proceedings] are also addressing problems which are amenable to architectures with runtime dispatch.

An important point is that this problem, of having to handle two cross-cutting localities, isn't due to the dual-interface framework. It is a fundamental problem, it has always been there and it will always be there. The structure of complex systems is such that it is natural for people to make this jump from one locality to another, and we have to find a way to support that. All the dual-interface framework does is: (i) make it more clear that this problem needs to be solved, and (ii), give one particular organization to the relation between the two different localities. Of course, looking at the problem this way makes it clear that we may well want more than two, cross-cutting, effective interfaces to a system — the *dual* interface framework may quickly become the multi-interface framework.

Summary

It runs deep in our field that we consider ourselves to be based on mathematics. This leads us to try and take many of our basic notions from mathematics. The fact that Abelson and Sussman would quote Weyl the way they do is evidence of this.

But, while this appeal to mathematics for conceptual foundations may be attractive, it is, at least in the case of abstraction, risky. There is a deep difference between what we do and what mathematicians do. The "abstractions" we manipulate are not, in point of fact, abstract. They are backed by real pieces of code, running on real machines, consuming real energy and taking up real space. To attempt to completely ignore the underlying implementation is like trying to completely ignore the laws of physics; it may be tempting but it won't get us very far.

Instead, what is possible is to temporarily *set aside* concern for some (or even all) of the laws of physics. This is what the dual interface model does: In the base-level interface we set physics aside, and focus on what behavior we want to build; in the meta-level interface we respect physics by making sure that the underlying implementation efficiently supports what we are doing. Because the two are separate, we can work with one without the other, in accordance with the primary purpose of abstraction, which is to give us a handle on complexity. But, because the two are coupled, we have an effective handle on the underlying implementation when we need it. I like to call this kind of abstraction, in which we sometimes elide, but never ignore the underlying implementation "physically correct computing."

[8] *Effective* means essentially the same thing that "causally connected" did in Smith's earlier work.

This is also like what the mechanical engineers call modeling, where they take multiple independent models of a system, each of which highlights certain properties and sets others aside. Of course a mechanical engineer's models aren't effective, and we would like ours to be — that is a fundamental difference in what we do and is why we can't borrow directly from them. But, it is the case that we are engineers not mathematicians. We would do better to look to other engineering disciplines, and not solely to mathematics, for our principles of abstraction.

This is, I think, the real contribution of the argument in this paper: Because we are engineers, not mathematicians, we must respect the laws of physics — we cannot hope to completely ignore the underlying implementation. The particular details of the dual interface model, the notion that two interfaces are enough, the role of object-oriented programming, the notion of meta; all of these are inherently approximate. What will remain, in the long term, is the intuition of physically correct computing and the requirement that we build open implementations.

Acknowledgments

I would like to thank Hal Abelson, J. Michael Ashley, Alan Bawden, Danny Bobrow, John Seely Brown, Jim des Rivières, Mike Dixon, John Lamping, Ramana Rao, Jonathan Rees, Luis Rodriguez, Erik Ruf, Brian Cantwell Smith, Marvin Theimer and Brent Welch for countless hours of discussion working out the ideas in this paper.

For their comments and feedback on earlier drafts of this paper itself, I would like to thank J. Michael Ashley, Danny Bobrow, Jim des Rivières, Mike Dixon, John Lamping and Luis Rodriguez.

References

[1] Anderson, T., Lazowska, E., and Levy, H. The performance implications of thread management alternatives for shared memory multiprocessors. In *IEEE Transactions on Computers, 38(12)* (1989), IEEE, pp. 1631–1644.

[2] Ashley, J. Open compilers. To appear in forthcoming PARC Technical Report., August 1992.

[3] Berlin, A. Partial evaluation applied to numerical computation. In *Lisp and Functional Programming Conference* (1990).

[4] Bobrow, D., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. Commonloops: Merging Lisp and object-oriented programming. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)* (Nov 1986), ACM.

[5] Chen, M., il Choo, Y., and Li, J. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing 2*, 2 (October 1988), 171–207.

[6] Cointe, P. Metaclasses are first class: The ObjVlisp model. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Orlando, FL* (1987), pp. 156–167.

[7] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[8] Ichisugi, Y., Matsuoka, S., Watanabe, T., and Yonezawa, A. An object-oriented concurrent reflective architecture for distributed computing environment. In *8th Conference Proceedings, Japan Society for Software Science and Technology* (Sept. 1991). (in Japanese).

[9] Ishikawa, Y., and Okamura, H. A new reflective architecture: AL–1 approach. In *Proceedings of the OOPSLA Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming* (1991).

[10] Keene, S. E. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[11] Kiczales, G. Metaobject protocols — why we want them and what else they can do. In *Object-Oriented Programming: The CLOS Perspective*, A. Paepcke, Ed. MIT Press, 1992.

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

[13] Lam, M. S., and Rinard, M. C. Coarse-grain parallel programming in Jade. In *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1991), pp. 94–105.

[14] Lamping, J., Kiczales, G., Rodriguez Jr., L. H., and Ruf, E. An architecture for an open compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures* (1992). Also to appear in forthcoming PARC Technical Report.

[15] Lucassen, J. M. Types and effects: Towards the integration of functional and imperative programming. Tech. Rep. MIT/LCS/TR-408, MIT, August 1987.

[16] Maes, P. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (1987), pp. 147–155.

[17] Matsuoka, S., Watanabe, T., and Yonezawa, A. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *European Conference on Object Oriented Programming* (1991), pp. 231–250.

[18] *The EuLisp Definition*, April 1992. Draft.

[19] Rao, R. Implementational reflection in Silica. In *Informal Proceedings of ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming* (Oct. 1990).

[20] 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, pp. 251–267.

[21] Rodriguez Jr., L. H. Coarse-grained parallelism using metaobject protocols. Master's thesis, Massachusetts Institute of Technology, 1991.

[22] Rodriguez Jr., L. H. Towards a better understanding of compile-time mops for parallelizing compilers. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures* (1992). Also to appear in forthcoming PARC Technical Report.

[23] Rowley, S., Shrobe, H., and Cassels, R. Joshua: Uniform access to heterogeneous knowledge structures or Why Joshua is better than conniving or planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence* (1987), pp. 48–58.

[24] Shaw, M., and Wulf, W. A. Towards relaxing assumptions in languages and their implementations. In *SIGPLAN Notices 15, 3* (1980), pp. 45–51.

[25] Steele, G. L. *Common Lisp: The Language (second edition)*. Digital Press, 1990.

[26] Vahdat, A. The design of a metaobject protocol controlling the behavior of a scheme interpreter. To appear in forthcoming PARC Technical Report., August 1992.

[27] Watanabe, T., and Yonezawa, A. An actor-based metalevel architecture for group-wide reflection. In *Informal Proceedings of ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming* (Oct. 1990). (Extended Abstract of [28]).

[28] Watanabe, T., and Yonezawa, A. An actor-based metalevel architecture for group-wide reflection. In *Proceedings of REX School/Workshop on Foundations of Object-Oriented Languages (REX/FOOL), Noordwijkerhout, the Netherlands, May, 1990* (1991), J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds., no. 489 in Lecture Notes in Computer Science, Springer Verlag, pp. 405–425.

[29] Wirth, N. On the design of programming languages. In *Information Processing 74* (1974).

[30] Yang, J. A., and il Choo, Y. Meta-crystal – a metalanguage for parallel-program optimization. Tech. Rep. YALEU/DCS/TR-786, Yale University, April 1990.

[31] Yokote, Y., Teraoka, F., and Tokoro, M. A reflective architecture for an object-oriented distributed operating system. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)* (July 1989). (also available as a technical report SCSL-TR-89-001, Sony Computer Science Laboratory Inc.).