(Appears in the Proceedings of IWOOOS'93)

# Operating Systems: Why Object-Oriented?

Gregor Kiczales, John Lamping*
Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304

## Abstract

*The implementor of an operating system service faces inherent dilemmas in choosing implementation strategies. A new approach to operating system design is emerging in which, rather than attempting to hide these dilemmas, they are exposed to clients, using a meta-protocols. Object-oriented programming techniques play an important role in enabling meta-protocols to expose these dilemmas to the client in a principled and modular fashion.*

## 1 Introduction

A fundamental change is taking place in the operating systems community: more and more issues that were once considered to be under the sole control of the operating system implementor are now being placed under the control of client programmers.[1] The OO-OS community has been at the forefront of this trend.

In Mach, the external pager allows clients to replace the paging mechanism [Y+87]. More recent work, [MA90] and [KLVA93], allows client replacement of the paging policy as well; Scheduler Activations[A+92] share the job of scheduling between clients and the system; Apertos allows these and other aspects of operating system implementation to be client-controlled [Yok92, YTT89, YTY+89, YTM+91]. Object-oriented operating systems under development provide these and other kinds of control as well [HK93, KN93, NKM93].

What is going on? Why are all these people abandoning the traditional notion of hiding the implementation of these services? Have they lost their senses? Or is there some coherent underlying theme that explains this trend? Why are object-oriented techniques so tied up with this? Why do some people use terms like "meta" and "reflective" when they talk about these systems?

We have developed a simple analysis that provides perspective on the situation. We argue that: (i) there are inherent dilemmas faced by the designers and implementors of operating system services; (ii) the recent trend reflects a move towards exposing these dilemmas to clients, rather than attempting to resolve them in the service implementation; and (iii) that object-oriented techniques play an important enabling role in exposing the dilemmas in a *principled and modular fashion*. We argue the first two of these points in more detail in [Kic92] and [KLM+93]. In this paper, we summarize those points and discuss the third.

We will examine some of the designs mentioned above in terms of this analysis. By giving us a common and fundamental way of understanding these different efforts, the analysis makes it possible to see parallels with work in other parts of computer science, including programming languages, databases, parallel and distributed computing. We believe the resulting knowledge sharing and perspective can lead to cleaner, simpler and more powerful operating system interfaces.

## 2 Mapping Dilemmas

The question to be answered then is: "Why are we being forced to expose implementation issues we once tried to hide, and is it possible to give a crisp categorization of the issues we need to expose?" The answer is intuitively accessible, but somewhat surprising when given explicit voice:

> It isn't possible to hide all implementation issues behind an abstraction barrier because not all of them are "just details," some are instead *crucial implementation strategy issues for which the resolution will invariably bias the performance of the resulting implementation.*

*gregor,lamping@parc.xerox.com

[1] In this paper, the term *client* is used to refer to a program or system that makes use of or calls another. The called system is referred to as a *service*.

We call these issues *mapping dilemmas* and the decisions on how to resolve them *mapping decisions*. When a particular client of a service performs poorly because the implementation embodies an innappropriate mapping decision, we call it a *mapping conflict*.

### 2.1 Examples

The explanatory power of these terms can be seen in a number of examples:

- **Virtual Memory** — In the case of virtual memory, the basic functionality being provided is particularly simple: a bunch of memory addresses that can be read or written. The mapping dilemmas have primarily to do with how to map virtual addresses to pages, and how to map those pages to physical memory at any given time. (This second concern breaks down into individual issues of page replacement policy, page ahead policy etc.)

  A classic example of mapping conflict in this domain is when a client, such as a database system or a garbage collector, does a "scan" of one portion of its memory, while doing random access to another portion. A virtual memory implementation based on an LRU is likely to perform well well for the second part of memory, but will be pessimal for the first part.

- **File Systems** — The functionality provided by file systems is somewhat more complex: a system of named "files" to which data can be read or written in a streaming or random access way. The mapping dilemmas in a file system are similar to those in virtual memory however: buffer size, cache management, read ahead, write-through etc.

  The common mapping conflicts are also similar to those in the virtual memory case. If, for example, a client that makes highly-random access to single addresses of a large number of files, and then doesn't read that location again for a long time, tries to run on a server that does extensive page-ahead and caching, the client's performance will probably not be good.

- **Network Protocols** — The basic functionality provided by a stream-oriented network protocol such as TCP is something like that of a file-system. First, there is a mechanism for getting a connection (essentially naming it), then the connection can be treated as a stream. The mapping dilemmas include such issues as buffer management and scheduling.

## 3 Meta-Protocols to Open the Implementation

Having established the motivation for opening up mapping decisions to clients, the question then becomes how to do it well. We want to avoid drowning clients in too many details, or overconstraining service implementors, or any other such problems. We must remember that the original reason for attempting to hide implementation issues was a good one: we wanted to simplify the client programmer's task by abstracting away issues that (we hoped) they didn't need to deal with. We have, over the years, developed a number of elegant OS abstractions, and we don't want to forfeit them or the skills that created them. What we would like is for our systems to become cleaner, not less so.

We argue that the original approach was an attempt to control complexity by exposing functionality and hiding implementation issues. The new approach, *based on separation rather than hiding*, is to control complexity by separating the client's access to the service's functionality from their access to the service's mapping decisions.[2]

The separation divides the interface that a service presents into two parts, called the base- and meta-interfaces (or protocols).[3] The base-interface is that ideal interface which a true black-box service would present: functionality only, no implementation issues. So, in the case of virtual memory, the base-interface is just read and write byte. In the case of a file system, it is something like open/close file and read/write byte. Clients of the service write a base-program that builds *on top* of the base-interface in the traditional way.

The meta-interface allows the client to control the mapping decisions underlying the implementation of the base-interface. Clients can use this interface to tune the implementation of the service to better suit their needs. For example, if there were a meta-interface to a virtual memory system, the client would write code to the base interface, reading and writing to a large address space; and then if the service's default paging implementation interacts poorly with the client's usage patterns, the client programmer could write code using the meta-interface to indicate a superior paging algorithm for the base code's behavior.

It is important to stress that while systems with an explicit meta-interface necessarily expose some implementation issues, they should nonetheless hide the true details of a service's implementation. Meta-interfaces should expose only mapping dilemmas. They should not expose minor

[2] This approach, of separating concerns when it is impossible to hide them, is a common strategy in other engineering disciplines.

[3] We are using the term 'meta' in the sense that the second interface or protocol is *about the implementation of the first*. That is, in this context, 'meta' means something similar to 'about'.

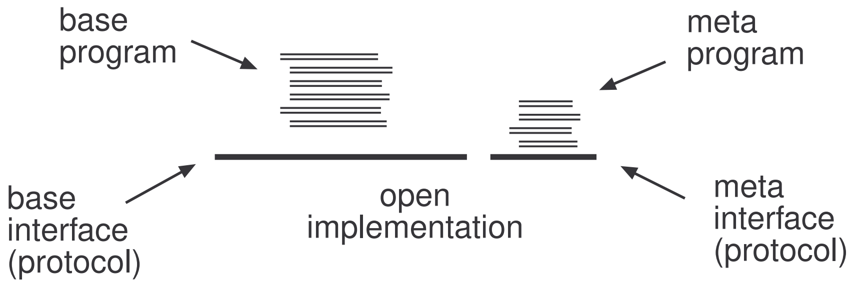


Figure 1: The meta-protocol framework. The client first writes a base-program, and then, if necessary, writes a meta-program to change any underlying mapping decisions that conflict with the base program's needs.

issues like variable names, and they should "hygienize" the mapping dilemmas as much as possible.

The goal in separating behavior and implementation is to allow the client programmer to: (i) Be able to control the service implementation when they need to; and (ii) when they need to do so, *be able to deal with behavior and implementation largely one at a time*. The goal is not to simply foist the responsibility for implementing the service onto the client. Mostly, the client programmer should just write in terms of the base-interface — they should not need to consider the meta-interface at all. But, for those performance-critical sections of their code that have mapping conflicts with the default service implementation, they can turn to the meta-interface and change the mapping decisions accordingly.

To meet these goals, we must design the base and meta interfaces in such a way as to make this switch in the client programmer's focus as easy possible. Clearly client programmers will need to use some knowledge of their base program when writing the meta program, but the goal is for them to not need to be emmeshed in its details. The idea is that they should be able to analyze their base program to discern what mapping decisions it needs, and then turn to the meta interface with only those mapping decisions in mind, not the full details of the base program.

### 3.1 More Fine-Grained Qualities

The principle of a base+meta interface provides an approach for separating different concerns, but more is needed to make it practical. In one sense, traditional black-box services have always had a crude meta-interface — if client programmers didn't like mapping decisions made inside a service implementation, they could write a new one entirely from scratch. (Sadly, this has been all too common. Database implementors, for example, have spent much of their energies coding around the virtual memory and file system services provided by operating systems.)

In architectures with explicit support for meta-level programming, we have found several qualities, beyond the essential base+meta split, to be vital to realizing its full potential:

- *Incrementality* — refers to the degree to which a client, when they want to change a *small* part of a service's implementation, can write a correspondingly small meta-program.

  One example of what this means for operating systems is that a client programmer, who wants a different page replacement policy, should be able to just describe the new policy, and not have to reimplement the mechanism as well.

- *Scope Control* — is the degree to which the effect of changes made through the meta-interface can be restricted to affect only some entities in (or parts of) the base-program.

  An example of what this means is that different parts of the same client should be able to have different page replacement policies. So, a database system can have a pager that handles both its 'scan' memory and its working memory well.

- *Interoperability* — is the degree to which parts of the base-program that use non-standard mapping decisions can freely interact with other parts of the base program.

  An example of what this means is that a client ought to be able to freely intermix addresses being paged under different policies, in a transparent (except for performance) way.

- *Robustness* — is the degree to which the implementation is graceful in the face of bugs in client meta-programs (and base-programs).

This is a traditional concern of operating systems — that different tasks are protected from each other — extended to the concept of meta-programming.

These are the properties that distinguish a meta-level architecture from a black-box service. Achieving them requires various techniques. Various kinds of protection facilities can provide robustness. Interoperability can be achieved via by having components interact only via standard interfaces. Finally, object orientation provides a way of achieving the first two qualities: scope control and incrementality.

We argue, in fact, that a principal role object orientation is playing in object oriented operating systems is to provide scope control and incrementality in a subcategory of meta-protocols, called metaobject protocols.

# 4 The Role of Object-Orientation

To see how object-oriented techniques achieve scope control and incrementality, first consider a *black-box* operating system with no client control over any aspect of how the box is implemented — mapping decisions or details. Such a system is depicted in the left hand side of Figure 2.

The first step towards making such a system more tailorable is an old an familiar one: allow the operating system to be tailored, at the time it is installed, to install different implementations of (or parameters to) various system services. This approach is depicted in the middle of Figure 2. It effectively opens up a dimension along which the system is partitioned, so that the implementation of parts of the system can be changed. In the figure, the points on the vertial axis correspond to the different services of the operating system that can be replaced.

This mechanism is powerful, but it is difficult (and potentially inefficient) to use. Using the new terms we have introduced, we can say quite succinctly what is wrong with this approach: It doesn't have very fine-grained scope control. *All* of the clients running on the operating system see the effects of *all* the changes made this way.

Object-oriented techniques make it possible to go one step further, by differentiating the operations of the box not only along a functional dimension, but also along a client dimension. The idea is to group the data that the box works on into objects, and to be able to specify a different mapping decision for a particular function and a particular equivalence class of objects.

This is shown in the right hand side of Figure 2. The objects, in this case might be different clients, or processes, or regions of memory. Intersections in the resulting "cartesian coordinates" are points of the systems behavior that are externally controllable. The circle at the intersection of two coordinates in the figure illustrates client "C" choosing a different pager.

The figure further illustrates the importance of carefully choosing the organization of the axes, because that determines the units of scope control and incrementality. The objects are the units of scope control, and the operations are the units of incrementality. In the figure, if the objects are individual clients, the scope control is quite coarse. If, on the other hand, they are individual regions of memory, the scope control is sharper.

The other axis, the units of functionality with respect to which mapping decisions can be changed, is equally important. A system that subdivides a pager into several sub-functions provides better incrementality. If all a client wants to change is one of the sub-pieces, they don't have to go to the trouble of reimplementing the rest of the service.

# 5 Examples

We are now in a position to examine some existing operating systems and show how the concepts we have presented can be used to explain and compare them. We claim that even though many of these systems were not originally conceived under these terms, the terms serve to explain them in a unified way.

Two clear examples of virtual memory systems with meta protocols are SunOS and the Mach External Pager. SunOS, with **madvise** and **mprotect** allows the client to choose among pre-defined implementation choices, rather than being allowed to write their own implementation. We call this kind of structure a *declarative* meta-protocol. Along the other dimension, the object dimension, the SunOS protocol gives fine grained control, in that processes can choose different strategies for different parts of their virtual memory.

The Mach External Pager provides equally fine control along the object dimension, in that one process can simultaneously map in various different memory objects, implemented by various pagers. The *imperative* meta-protocol it provides is more powerful since users are allowed to write their own page replacement mechanisms rather than choosing from among a fixed set. On the other hand, it provides worse incrementality, since the client must replace the entire paging mechanism to make even a small change.

The work by Krueger et. al.[KLVA93] does better with incrementality, since it decomposes pager functionality into several pieces and lets the user replace only those pieces that matter to them.

In general, there has been a trend in object oriented operating systems toward providing finer control along both dimensions, leading to more and more flexible meta-level

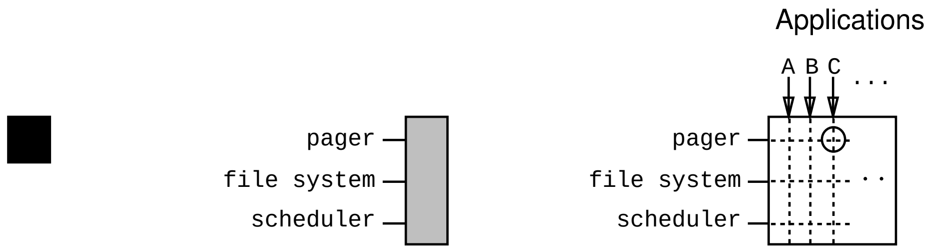


Figure 2: On the left, a black-box system. In the middle, a grey-box that allows replacement of certain internal handlers. On the right, object-oriented techniques provide cartesian coordinates, making it possible to replace the handler only for a certain category of object.

interfaces. The mapping dilemma framework provides a way of looking at object oriented operating systems in terms of how well they provide access to mapping choices in an incremental way, and with good scope control. These metrics, in turn, can suggest where the access might be improved.

## 6 Conclusion and Future Work

We have tried to show what problem it is the OO-OS community is solving, and what role OO is playing in those solutions. In particular, operating system implementors face mapping dilemmas: critical issues of how to implement the services operating systems provide. It is impossible for the OS designer to resolve these mapping dilemmas, because different applications have different requirements. The strategy that is evolving is for services to provide an explicit meta-interface that allows client control over mapping decisions. A major challenge in this approach is to find principled and elegant way of presenting the meta-functionality.

Object-Oriented programming techniques provide two critical capabilities toward this goal: scope control and incrementality. They do this by providing a kind of cartesian coordinates for targeting a fixed point inside a system from the outside.

But the real issues that are being attacked are mapping dilemmas. The point is that while we should cherish our abstractions, we also need to be able to talk about how they are implemented. Object orientation is a technique in service of that goal.

### Acknowledgements

A number of people, too numerous to mention, have contributed to the development of the concept of meta-protocols. We are grateful to the following people for helping us to see their application to (and existing use in) operating systems: Paul Dourish, David Keppel, Chris Maeda, Dylan McNamee, Gail Murphy, Mike Dixon, Marvin Theimer, Mark Weiser and Brent Welch.

## References

[A+92] Thomas E. Anderson et al. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[HK93] Graham Hamilton and Panos Kougiouris. The spring nucleus: A microkernel for objects. Technical Report SMLI TR-93-14, Sun Microsystems Laboratories, Inc., April 1993.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in software engineering. In Akinori Yonezawa and Brian Cantwell Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*, pages 1–11, Tokyo, Japan, November 1992.

[KLM+93] Gregor Kiczales, John Lamping, Chris Maeda, David Keppel, and Dylan McNamee. The need for customizable operating systems. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, October 1993.

[KLVA93] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for the development of application-specific virtual memory management. In *Proceedings of* *the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, October 1993.

[KN93] Yousef A. Khalidi and Michael N. Nelson. The spring virtual memory system. Technical Report SMLI TR-93-09, Sun Microsystems Laboratories, Inc., February 1993.

[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.

[NKM93] Michaell N. Nelson, Yousef A. Khalidi, and Peter W. Madany. The spring file system. Technical Report SMLI TR-93-10, Sun Microsystems Laboratories, Inc., February 1993.

[Y+87] Michael Young et al. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th Symposium on Operating Systems Principles*, 1987.

[Yok92] Yasuhiko Yokote. The apertos reflective operating system: The concept and its implementation. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 414–434, October 1992.

[YTM+91] Yasuhiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The Muse object architecture: A new operating system structuring concept. *Operating Systems Review*, 25(2), April 1991. (also available as a technical report SCSL-TR-91-002, Sony Computer Science Laboratory Inc.).

[YTT89] Yasuhiko Yokote, Fumio Teraoka, and Mario Tokoro. 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.).

[YTY+89] Yasuhiko Yokote, Fumio Teraoka, Masaki Yamada, Hiroshi Tezuka, and Mario Tokoro. The design and implementation of the Muse object-oriented distributed operating system. In *Proceedings of the First Conference on Technology of Object-Oriented Languages and Systems*, October 1989. (also available as a technical report SCSL-TR-89-010, Sony Computer Science Laboratory Inc.).