Inheritance in Unlikely Places:
Using Objects to Build Derived Implementations of Flat Interfaces

Michael B. Jones

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052, USA
Michael.Jones@cs.cmu.edu

Abstract

Traditional operating systems typically provide a number of flat, non-object oriented interfaces through which system services are provided to applications. Two such examples are the UNIX[1] system call interface and the X Window System[2] protocol. Typically such services are provided by large, monolithic implementations, with no provision for deriving related implementations through inheritance or other means.

Nonetheless, it is possible to provide flexible, object oriented views of otherwise flat, non-object oriented interfaces. Furthermore, these views can be used to build new implementations of these same flat interfaces in terms of other ones. This approach can bring the benefits of objects such as enhanced and/or application specific implementations of standard services to both building and using otherwise fixed, flat interfaces which for one reason or another cannot be changed.

My recent thesis work has taken this approach; other seemingly unrelated systems also use this paradigm. This paper presents this paradigm and some of the benefits derived from it, citing several real systems as examples.

1. Introduction

The ability to implement a given interface in terms of another implementation of the same interface is taken for granted by the object oriented programming community. Some of the benefits of this ability include being able to utilize enhanced and/or application specific implementations of nonetheless standard interfaces. Such an implementation of one instance of an interface in terms of another is typically accomplished through object oriented interfaces and the use of inheritance.

[1]UNIX is a trademark of UNIX System Laboratories, Inc.

[2]The X Window System is a trademark of Massachusetts Institute of Technology.

On the other hand, traditional operating systems typically provide one or more flat interfaces containing interface specific abstractions. For example, the UNIX interface presents abstractions such as pathnames, descriptors, files, directories, processes, etc.; the X Window System interface presents abstractions such as resource IDs, windows, pixmaps, input events, etc. While these interfaces typically contain a small number of abstractions with largely independent semantics, these abstractions are not presented to applications as objects. Thus, no clear means of inheritance for such interfaces is supported. However, recently several systems have been built which overcome these problems, allowing derived implementations of flat interfaces to be easily built in terms of existing implementations of these same interfaces. While these systems have typically been developed independently, the structure of the solutions arrived at is in many cases substantially similar.

In particular, objects are constructed which correspond to the abstractions present in the original interface, providing an object oriented view of the original interface. Such objects are typically part of a toolkit, providing different sets of objects and interfaces appropriate to different applications. Next, an implementation of the original interface is constructed in terms of the toolkit objects. Taken together, these components map the original flat interface onto an object oriented interface, which is in turn mapped back onto the original interface. This having been done, inheritance may be used in the usual fashion to construct derived implementations of the object oriented interface. When composed with the mappings from and to the original interface, these then become derived implementations of the original interface, accomplishing the desired result.

2. Parallel Structure in Two Different Problem Domains

This section illustrates the paradigm of using objects to build derived implementations of flat interfaces through two examples in two different problem domains. The first is in the domain of operating system interfaces. In particular, my thesis work [7] [8] [9] has demonstrated both the usefulness of and an effective method for constructing what I have termed system interface interposition agents. Interposition agents are user code which provide the system interface to normal clients of the system interface (e.g., application programs) and which are implemented in terms another instance of the same interface. Agents look to normal applications like the operating system, and to the operating system like a normal application. In other words, interposition agents are derived implementations of the operating system interface.

The second is in the domain of network window system interfaces. A number of systems have been built which provide the X Window System [14] server interface to X client programs, and which are themselves implemented as X client programs. Such programs are often known as X pseudo-servers [15]. Thus, X pseudo-servers are derived implementations of the X Window System interface.

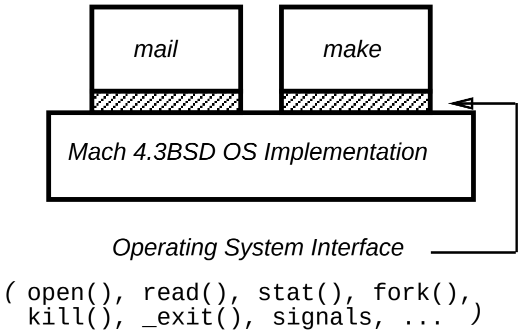

Figure 2-1: Operating system interface examples

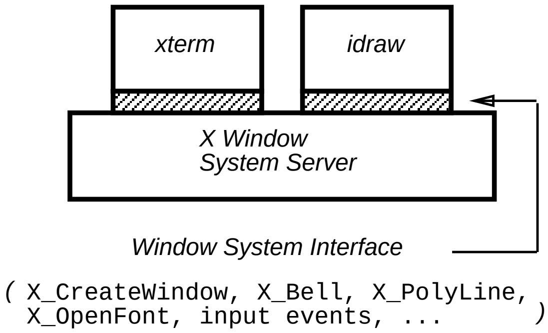

Figure 2-2: X Window System interface examples

To make these examples more concrete, figures 2-1 and 2-2 demonstrate use of their respective interfaces by client programs in the normal manner. Figures 2-3 and 2-4 expand these pictures to include the use of derived implementations of their respective interfaces. As well as providing examples of the use of derived implementations of flat interfaces in the abstract, figures 2-3 and 2-4 provide a few such examples of derived implementations which are clearly also useful in their own right.

Figure 2-3 contains two system interface interposition agents which have been motivated by software engineering needs of the Mach project [1]. The first provides for views of the filesystem namespace which may be customized to

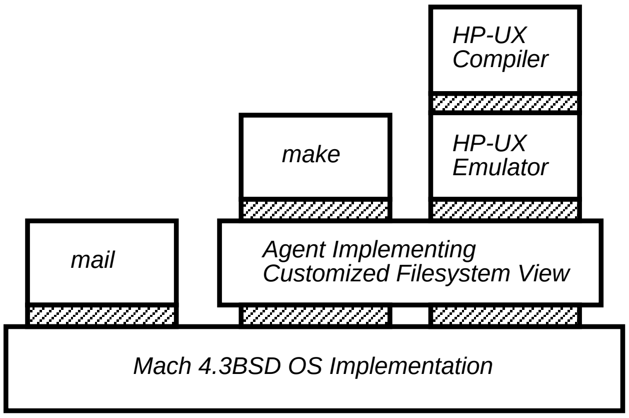

Figure 2-3: Operating system interface examples with derived implementations

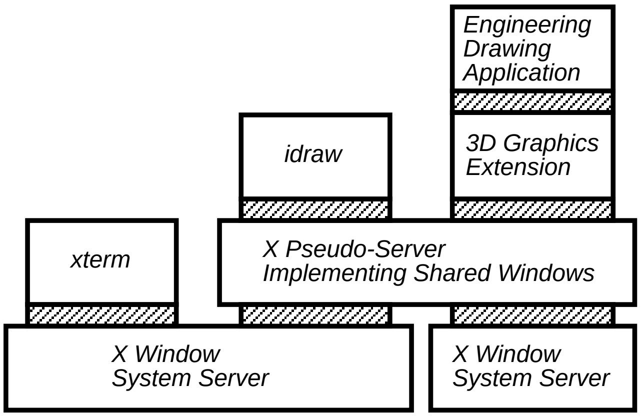

Figure 2-4: X Window System interface examples with derived implementations

suit application requirements. This agent might implement enhancements such as viewpaths and/or union directories [11] [6]. The second agent might provide an adapter between two variants of the UNIX interface. In particular, such an agent would make it possible for binaries from one UNIX variant to be run on a somewhat different one through the use of a derived and specialized implementation of the system interface.

Figure 2-4 contains two X Window System interface pseudo-servers which have been motivated by needs of other groups. The first provides a shared X server, allowing users on different physical displays to share logically combined displays or windows. The second pseudo-server provides an implementation of an X protocol extension as a layer on top of another X server which might not otherwise provide it. Such pseudo-servers potentially provide more flexibility for both experimentation and implementation of X server functions than would otherwise be present.

While the structure of the two figures is slightly different, this does not reflect any fundamental difference between the two problem domains. Rather, the differences are merely an artifact of the actual examples chosen.

Although not strictly parallel in structure, the examples were chosen because they demonstrate real solutions to actual problems.

3. Solution Structure

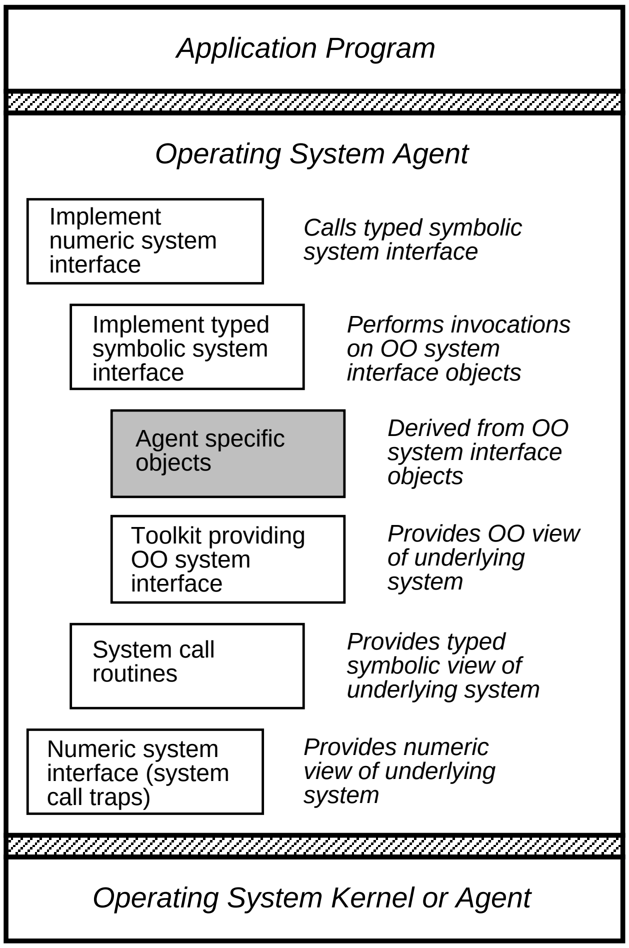

Figure 3-1: Structure of a system interface interposition agent

When an interface is presented to clients in an object-oriented manner derived implementations can be produced a in straight forward fashion through the use of inheritance. Indeed, examples of the success of this approach in both example domains exist [4] [13]. A means of doing so for the flat, non-object oriented interfaces prevalent in traditional operating systems is not as readily apparent.

Nonetheless, derived implementations of flat interfaces can be supported by transforming the flat interface into an object oriented interface, producing a derived version of the object oriented interface in the usual manner through inheritance, and then transforming the object oriented interface back into a client of the original flat interface. Once the superstructure has been constructed which is necessary to perform the transformations to and from the new object oriented interface, any number of potentially unrelated derived implementations of the flat interface can

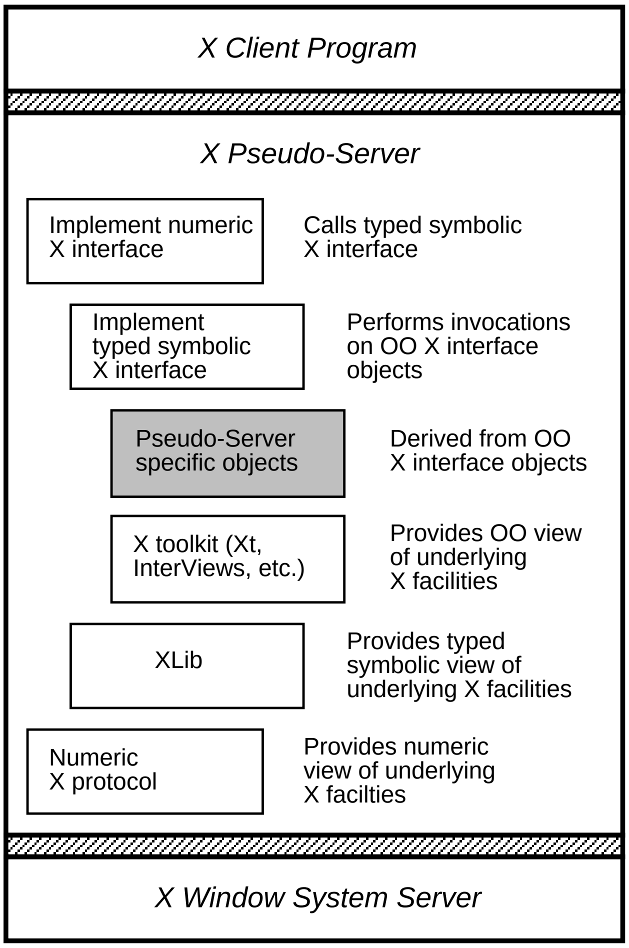

Figure 3-2: Structure of an X Window System pseudo-server

be constructed in terms of derived implementations of the object oriented interface. This key idea should be effective across a number of problem domains.

For instance, an object oriented toolkit providing objects corresponding to UNIX abstractions was constructed as part of my thesis work [9]. Multiple layers of objects exist which allow derived forms of such abstractions as system calls, descriptors, open objects, pathnames, directories, etc. to be implemented via inheritance. Similar toolkits can be provided for other interfaces.

To use this approach for a given flat interface, a toolkit providing object oriented views of that interface must be constructed. The structure of each such toolkit is highly interface-specific, and while the ideas underlying them are similar, the code for each is likely to be quite different. Each toolkit, and indeed, each derived implementation built with them, must be carefully constructed so as to preserve the semantics of the underlying interface. The good news is that once such a toolkit has been built for a given flat interface it can be reused to build many derived implementations of that interface.

Figures 3-1 and 3-2 demonstrate the application of this paradigm in the two example problem domains. In both figures, structures are presented which transform the flat interface into invocations on an object oriented interface, and which then implement this object oriented interface in terms of the original flat interface. Both figures present a general framework which can be specialized to produce any number of different derived implementations. The key to both illustrations is that specializations can be done through application specific objects which are derived from the object oriented interface objects. These application specific derived objects are indicated by the shaded box in each diagram. While the interfaces provided in the different problem domains differ widely, the same approach can be effective in both.

4. Applicability

The success of this approach hinges on the ability to cast an existing interface into a well-structured object oriented interface. I believe that this should be possible for existing interfaces which satisfy two key requirements:

- the interface contains a reasonably small number of abstractions;
- the behavior of these abstractions is largely independent.

Interfaces satisfying these properties should be amenable to this approach.

The two examples presented earlier, the UNIX system interface and the X Window System, both satisfy these properties. Not every interface does. As a glaring counterexample, consider libc taken as a whole, which satisfies neither. Libc contains a huge number of abstractions that are often implemented in terms of one another in interdependent and non-obvious ways.

5. Costs

One obvious potential cost of this approach is some performance loss. It is easy to incur additional overhead whenever more layers of abstraction are added to a system. On the other hand, the derived implementation presumedly provides some benefit to the applications which use it, or it wouldn't be used. In this sense, the performance costs are of a pay-per-use nature. While the benefit provided might be enhanced or modified functionality, it could also be an increase in performance. Such performance increases might be gained by capitalizing on application-specific knowledge of how the original interface is used. Indeed, one of the benefits of implementational reflection is the ability of an application to specialize and/or tune the implementations of services which it uses. Thus, in some cases, performance may actually be enhanced, despite the overheads caused by additional layers of abstraction.

The actual costs of interposing a derived implementation of an interface between normal clients and providers of the interface will depend upon the mechanism used. Some mechanisms can be used to selectively interpose on only those calls actually modified in the derived implementation, allowing the unmodified calls to incur no additional overhead. UNIX interposition agents constructed in the manner described in my thesis [9] have this property. Others, such as the use of an X pseudo-server, do not. Even for those calls which are modified, the costs need not be prohibitive. For instance, on a 25mhz i486, running Mach 2.5 (X144), system call interception takes only 30 µs. and calling the underlying implementation takes 37 µs., for a total additional overhead of 67 µs. This compares to 245 µs. to do an unintercepted *read()* of 1K on the same implementation.

Another potential cost of this approach is that of constructing software which is somewhat more general than strictly necessary to solve any particular problem requiring a changed implementation of a given interface. For instance, one competing approach would be to take the original implementation, and make changes to it. While this approach is time-honored and well understood, so are its drawbacks; it tends towards unstructured and undisciplined growth without bound. Furthermore, the result tends to be, if anything, harder to modify the next time. Yet, once the work has been done to build an object oriented structure supporting inheritance for a given interface, it can easily be reused to build many derived implementations. Thus, while the initial work to construct a changed implementation of a flat interface in terms of objects may be somewhat greater, the software engineering cost has the potential to actually be less in the long term.

This approach also provides another potential benefit. Derived implementations can be independently constructed and dynamically composed, providing more flexibility than simply changing the original implementation. In particular, there are situations where it is infeasible to modify the original implementation. In such cases, using a derived implementation may be the only choice.

6. Further Insights

The approach presented in this paper clearly builds upon techniques in common use in more traditional object oriented systems. Operating system agents utilize many of the techniques used by such object oriented operating systems as Choices [4]. Other systems such as the Mach Multi-Server [5] have used objects to provide flexible implementations of a flat interface, although do not do so in terms of an instance of the flat interface itself. Likewise, X pseudo-servers capitalize on the object oriented approaches taken by such X toolkits as Motif [12] and InterViews [3].

A more surprising and deeper result is that the ability to produce derived implementations of flat interfaces allows for implementational reflection[3] [13] [10] and introspection to be applied despite the often closed, monolithic nature of the usual implementations of these interfaces. Such derived implementations can expose relevant portions of their implementations, allowing for effective local and incremental adjustment. While reflection has been gaining

[3] Implementational reflection, as used in this paper, is the ability for a program to inspect, manipulate, and/or participate in the implementation of the facilities used by the program.

popularity through its use in object oriented languages such as CLOS [2] [16], object oriented window systems such as Silica [13] and object oriented operating systems such as Muse [17], its applicability to non-object oriented flat interfaces is a new result.

7. Conclusion

Traditional operating systems often provide system services through flat interfaces, which are typically implemented in a monolithic, closed manner. No provision is made for enhanced, restricted, or otherwise modified implementations of these same interfaces to coexist with the standard implementations.

In this paper, I have argued that a common paradigm can be used to produce derived implementations of a diverse class of flat interfaces such as those which provide system services. This provides a means of effectively ‘‘opening up’’ the implementations such system services, allowing for a multiplicity of derived implementations of such interfaces to be easily constructed. This approach makes it possible to bring many of the benefits of object oriented methodologies to domains where they were previously unavailable.

Acknowledgments

I’d like to extend a special thanks to Gregor Kiczales for the conversation which he insisted that we have during my recent visit to Xerox PARC. It was stimulating, thought provoking, and helped water the germ of an idea which became this paper. I’d also like to thank Daniel Duchamp and Chris Kent Kantarjiev for lending me their perspectives on X pseudo-servers.

This research was sponsored by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. government.

References

1. M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. Proc. Summer 1986 USENIX Technical Conference and Exhibition, June, 1986.

2. Bobrow, Daniel G., Linda G. Demichiel, Richard P. Gabriel, Sonya E. Kleene, Gregor Kiczales, and David A. Moon. *Common Lisp Object System Specification.* X3J13 Document Number 88-002R, 1988. Also appears as Chapter 28 of *Common Lisp, The Language*, Second Edition.

3. Paul Calder and Mark Linton. Glyphs: Flyweight Objects for User Interfaces. Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology, Oct, 1990, pp. 92-101.

4. Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. Proceedings of the USENIX C++ Workshop, Santa Fe, New Mexico, November, 1987, pp. 109-123.

5. Paulo Guedes and Daniel Julin. Object-Oriented Interfaces in the Mach 3.0 Multi-Server System. Proceedings of the Second International Workshop on Object-Orientation in Operating Systems, Palo Alto, October, 1991.

6. Hendricks, David. A Filesystem For Software Development. Summer Usenix Conference Proceedings, June, 1990, pp. 333-340.

7. Michael B. Jones. "A Toolkit for Interposing User Code at the System Interface". *IEEE Computer Society Technical Committee on Operating Systems and Applications Environments (TCOS) Technical Committee Newsletter 5*, 1 (Spring 1991), 26-31.

8. Michael B. Jones. Transparently Interposing User Code at the System Interface. Proceedings of the Second Workshop on Workstation Operating Systems, April, 1992.

9. Michael B. Jones. *Transparently Interposing User Code at the System Interface*. Ph.D. Th., Carnegie Mellon University, 1992. To be published Summer, 1992.

10. Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

11. Korn, David G. and Krell, Eduardo. "A New Dimension for the Unix File System". *Software - Practice and Experience 20*, S1 (Jun 1990), 19-34.

12. *OSF/MOTIF Manual.* Open Software Foundation, Cambridge, MA, 1989.

13. Ramana Rao. Implementational Reflection in Silica. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1991.

14. R. W. Scheifler and J. Gettys. "The X Window System". *ACM Transactions on Graphics 5*, 2 (1986).

15. Ian Smith and Elizabeth Mynatt. What You See Is What I Want: Experiences With The Virtual X Shared Window System. Tech. Rept. GT-GVU-91-33, Georgia Tech College of Computing, 1991.

16. Steele, Guy. *Common Lisp: The Language*, Second Edition. Digital Press, 1990.

17. Yashuhiko Yokote, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. Reflective Object Management in the Muse Operating System. Proceedings of the International Workshop on Object Orientation in Operating Systems, October, 1991, pp. 16-23.