Choices, Frameworks and Refinement*

Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris and Peter Madany
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801

Abstract

In this paper, we present a method for designing operating systems using object-oriented frameworks. A framework can be refined into subframeworks. Constraints specify the interactions between the subframeworks. We describe how we used object-oriented frameworks to design Choices, an object-oriented operating system.

1 Frameworks in an Object-Oriented Operating System

The design of *Choices*[1], an object-oriented operating system, comprises a hierarchy of frameworks. In the design, the concept of a framework subsumes the conventional organization of an operating system into layers. Frameworks not only allow the design of layers, but they also permit the construction of more complex structures. The use of frameworks permits design and code reuse and the consistent imposition of design constraints on all software, independent of the level at which it may be used.

The object-oriented operating system approach builds system software that models system resources and resource management as an organized collection of objects that encapsulate mechanisms, policies, algorithms, and data representations. A class defines a collection of objects that have identical behavior. Class hierarchies define relationships between classes that share common behavioral properties. Inheritance and inclusion polymorphism permit the methods of a concrete subclass to implement an invocation of a method on an abstract class. A framework of classes defines a design architecture that expresses the organization of an object-oriented implementation of a system. The framework can be refined into subframeworks, corresponding to the composition of a large complex system out of smaller interacting subsystems. A particular operating system implementation is just one of many possible ways that a framework for an operating system can be "instantiated."

Choices was designed from the beginning as an object-oriented operating system implemented in C++. The system runs stand-alone on the Sun SPARCstation II, Encore Multimax, Apple Macintosh IIx, IBM PS/2, and AT&T WGS-386. It supports distributed and shared memory multiprocessor applications, virtual memory, and has both conventional file systems and a persistent object store. The system has over 300 classes and 150,000 lines of source code.

In this paper we will describe how we have used the object-oriented notion of a framework in our work. *Choices* has the following frameworks: process management and exception handling, scheduling, synchronization, memory management, persistent storage, device management, name resolution, message passing, communication protocols, application interface, and instrumentation. We will discuss how particular frameworks in *Choices* have contributed to the organization of the system, techniques we have found helpful for building frameworks, and why frameworks are useful.

2 What is a Framework?

A framework is an architectural design for object-oriented systems. It describes the components of the system and the way they interact. In frameworks, classes define the components of the system. The interactions in the system are defined by constraints, inheritance, inclusion polymorphism, and informal rules of composition. *Choices* frameworks use class hierarchies to define single inheritance and C++ subtyping to express inclusion polymorphism. In practice, we

*This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grants NSG1471 and NAG 1-163.

have found that the design of a complex system such as an operating system is best defined as a framework that guides the design of subframeworks for subsystems. The subframeworks refine the operating system framework, as it applies to a specific subsystem.

The framework for the system provides generalized components and constraints to which the specialized subframeworks conform. The subframeworks introduce additional components and constraints and subclass some of the components of the framework. Recursively, these subframeworks may be refined to further frameworks. Frameworks simplify the construction of a family of related systems by providing an architectural design that has common components and interactions. An instance of a framework is a particular member of the family of systems.

Frameworks both support and augment the traditional layered approach that has been used to design operating systems. In both approaches the problem domain is divided into smaller domains. A layer represents an abstract machine that hides machine dependencies and provides new services. The abstract machine is presented as a set of subroutines. A framework introduces classes of components that encapsulate machine dependencies and define new services. A layer introduces an interface between implementations that is constrained by the set of calls that are defined.

A framework defines interfaces in the form of the public methods of abstract classes. It imposes restrictions on the implementation of an interface by the constraints it imposes. In the layered approach, the design of each layer is independent. Algorithms or data structures in one level may be similar to those in other levels, but the level approach to design has no way to express that similarity. Instead, a framework may have several different instantiations and implementations within a system; it may be reused. The constraints of a framework allow more complex interactions than between levels. The framework approach subsumes the layered approach because the basic properties of the layered approach can be modeled by frameworks. However, the framework approach also allows the constraints within a particular layer to be expressed. Finally, a framework can be defined in terms of abstract classes that are bound to specific concrete classes at run-time using inheritance and inclusion polymorphism. This provides the compile time independence that is exhibited by, for example, the application interface layer but also allows dynamic binding as, for example, is necessary to allow device drivers to be added or changed in a running system.

3 Choices Frameworks

The framework for *Choices* introduces abstract classes that represent the high-level concepts that are fundamental to an operating system. Subframeworks are defined that refine these concepts in the context of a subsystem of the operating system. The *Choices* framework provides the subframeworks with the design consistency constraints that are required to link them together into a system. The *Choices* framework consists of three abstract classes: *MemoryObject*, *Process*, and *Domain*.

In this paper, we describe the Persistent Storage, Device Management and Message Passing subframeworks. The virtual memory subframework has already been described by Johnson and Russo [5]. There are three stages to building a *Choices* subsystem. First, the abstract properties of the subsystem are collected. Next, a subframework is developed which is consistent with the *Choices* framework and which encodes the properties of the subsystem. This subframework defines classes and constraints. Finally, the subsystem is created as an implementation of the subframework, with concrete classes specializing the abstract classes of the framework and instances that conform to the constraints specified by the framework. The following diagram shows the development of a subsystem from abstract properties to a concrete implementation.

$$Abstract\ Properties \rightarrow subframework = \{classes, constraints\}$$

$$\rightarrow subsystem = \{instance\ of\ subframework\}$$

3.1 Choices subframeworks

Persistent Storage The *Choices* persistent storage framework[7] introduces a hierarchy of classes that can be combined to build both standard and customized storage systems. It is flexible enough to support both persistent storage systems and traditional file systems efficiently[8]. The framework abstracts five major properties of persistent storage systems:

- Persistent Storage,
- Storage Organization and Device Sharing,
- Naming,
- Data Structuring, and
- Application Interface.

The persistent storage framework divides a persistent storage system into three layers and is, therefore,

an example of a framework that subsumes a traditional layering structure. The top layer contains objects that present application interfaces, the middle layer contains objects that name files and structure the data within files, and the bottom layer contains objects that store and organize persistent data. The bottom layer can be further divided into several levels.

The *PersistentStore* class, which is a subclass of *MemoryObject*, abstracts persistent storage by defining persistent data stores which store and retrieve blocks of data using a random access method. Concrete *PersistentStores* are categorized by the two subclasses of *PersistentStore*: *Disk* and *File*. *Disks* encapsulate physical storage devices like hard disk drives, floppy disk drives, and RAM disks. They communicate with objects in the I/O subsystem. *Files* encapsulate logical storage devices like UNIX inodes and disk partitions. Each file has a source *PersistentStore* that supplies it with data from a lower level of the file system. *Files* provide a *window* into their source *PersistentStore*. The size of this window can be fixed or variable and can range from zero up to the size of the source *PersistentStore*. The window can be contiguous or divided into discontiguous regions of blocks.

The *PersistentObject* class defines persistent objects, which encapsulate and provide operations on the data managed by a persistent store. While a *PersistentStore* provides random access to an uninterpreted sequence of data, a *PersistentObject* interprets the data of a persistent store as having a format. Each *PersistentStore* has an associated *PersistentObject* class that provides a data abstraction and encapsulation of the persistent data in the store. At runtime, there is a one-to-one correspondence between an instance of a *PersistentStore* and its associated *PersistentObject*. The *PersistentStore* thus provides the underlying data for its associated *PersistentObject*. Subclasses of *PersistentObject* abstract the organization, naming, and data structuring properties of the persistent storage framework.

Storage is organized and storage devices are shared by dividing *PersistentStores* into nested levels of smaller *PersistentStores*. The *PersistentStoreContainer* class defines objects that divide a persistent store into an indexed collection of smaller stores (i.e. a collection of *Files*). *PersistentStoreContainers* that contain variable-length files, or ones that can create new files, use a *BlockAllocator* to manage the allocation of data blocks for the files.

Naming is orthogonal to storage organization. The *PersistentStoreDictionary* class defines objects that map symbolic names to the indices used by *PersistentStoreContainers*. While *Files* must be contained in exactly one container, they can be named by several dictionaries.

The framework incorporates three models for structuring the data within files: as arrays of bytes or words (defined by subclasses of *PersistentArray*), as collections of records (defined by subclasses of *RecordFile*), and as data structures encapsulated by persistent objects (defined by subclasses of *AutoloadPersistentObject*). The first model is suited to the C programming language and the UNIX and MS-DOS operating systems. The file system presents a random-access interface to sequences of bytes and imposes no additional structure. The second model fits programming languages like Cobol, PL/1, and Pascal and operating systems like VMS. The file system presents data as records that can correspond to the types of data structures of the language. The third model fits programming languages like C++ and object-oriented operating systems like *Choices*. The object storage system presents data as objects that are instances of user-defined subclasses of *AutoloadPersistentObject*.

The interfaces provided by the naming and data structuring classes are abstract enough to be used directly by application programs; but conventional file systems commonly define an additional layer of abstraction between files and application programs. The *FileSystemInterface* class provides this extra layer by organizing the dictionaries from various containers into a single hierarchy. Subclasses of the *RecordStream* class provide stream-oriented application interfaces for both *PersistentArrays* and *RecordFiles*.

Device Management The device management framework in *Choices* abstracts three properties of a typical I/O architecture:

- I/O Devices (*Device*)
- I/O Controllers (*DevicesController*)
- Addition and removal of devices, controllers and drivers(*DevicesManager*)

The device management framework[6] in *Choices* consists of two hierarchies and a DevicesManager class. One hierarchy is based on the abstract class *Device*. The other hierarchy is based on the abstract class *DevicesController*. An instance of a *Device* is constructed and bound to the NameServer when it is added to the system. Each *Device* acts as a server for components of other *Choices* frameworks. For instance, a DiskDevice acts as a server of classes of the file system framework. In turn, most of the *Devices* act as clients of

DevicesController objects. For instance, two DiskDevices representing disks attached to the same hardware controller act as clients of the same *DiskController*. The protocol of a *Device* depends on the physical device it represents. For instance, DiskDevices have `::read()` and `::write()` methods that transfer a number of disk blocks. SerialLineDevices have Input/Output methods to transfer strings of characters and control methods like `::setBaudRate()` to set control parameters.

Instances of *DevicesControllers* classes represent hardware controllers. A *DevicesController* acts as a server for possibly several *Devices*. A *DevicesController* is not visible to the user of a device. I/O operations should only be requested from a *Device*. The only other framework that interacts with a *DevicesController* is the Exception Handling framework. The interface between a *Device* and a *DevicesController* is a message interface based on *Command* objects. User requests on *Devices* cause the construction of one or more *Commands* which are then sent to a *DevicesController* object using the `::sendCommand()` method. A message interface between the *DevicesController* and the *Device* has two advantages. The first is that a *Device* can be reused with different *DevicesControllers*. For example, a *DiskDevice* can be used as the *Device* of a machine-dependent DiskController and a machine-independent SCSIDiskController. The second advantage of the message interface is that it does not force a *DevicesController* to have a specific interface that depends on its devices. The protocol of a *DevicesController* subclass can change without requiring a change to existing *Devices*. On the other hand, the message interface cannot be checked at compile-time to ensure consistency. We think that this is not a major problem, since the interface is internal to the framework.

The DevicesManager class is the third component of the framework. Each system has only one object of this class. When a *DevicesController* is loaded into the system it registers with the DevicesManager object. Hardware controllers and devices that are added to the system are also registered with the DevicesManager. The DevicesManager is informed of the addition and removal of hardware components by the system administrator or by the cooperation of hardware and machine-dependent software. The DevicesManager matches physical controllers with registered *DevicesControllers*. For each physical controller a "matching" *DevicesController* is instantiated. In addition, for each physical device a *Device* is constructed and returned when the method `DevicesController::attachDevice()` is invoked. The new *Device* is then bound to the NameServer.

Other frameworks use the Device Management framework with the help of classes that provide communication between the two frameworks. For instance, a Disk is a *PersistentStore* that does Input/Output using a DiskDevice. *Devices* are converted to objects in other hierarchies using the *Choices* conversion mechanism. Conversion is a term introduced in Smalltalk[2] and used for the collection classes. We generalize the conversion mechanism to apply to any class. We also combined the conversion mechanism with double dispatching[3] so that new interframework classes can be added to the system without changing existing classes inside the frameworks.

Message Passing This section describes a subframework for message passing designed to support hypercube applications on a shared memory machine. The abstract properties of message passing subframework can be categorized into five parts, with various options, as shown below.

- *Location* of Message system:
  - *User space*
  - *Kernel*
- *Message* Semantics and Process control:
  - *Hypercube*
- *Transport*:
  - *Process:* an independent process copies the message from source to destination.
  - *Buffered:* the receiver process incurs the overhead of message transfer.
- *Synchronization*:
  - *Semaphore*
  - *Spinlock*
- *Transfer* of the exchange of data implemented by:
  - *DoubleCopy:* the message is copied into and out of the kernel.
  - *SingleCopy:* the message is copied once from sender buffer to receiver buffer.
  - *PointerTransfer:* exchange of buffer pointers with no copy of data.

From the abstract properties we can derive a set of *abstract classes* and their associated hierarchies. The hierarchies may express some of the constraints in their names: other constraints are described in the

description of the interaction of the classes. These abstract classes are *MessageTransport*, *MessageSemantics*, *MessageSynchronization*, and *MessageTransfer*. Each of these have concrete subclasses that implement the abstractions defined by them. In the following paragraphs we will only describe a representative sample of concrete subclasses. The *KernelMessageSystem* and *UserMessageSystem* classes define, as well as implement, the location semantics. A more complete description of the message passing subframework can be found in Islam and Campbell [4].

The *MessageSemantics* class defines the semantics of message passing, such as whether asynchronous and synchronous message passing is supported. It also defines the process control semantics such as whether processes should be gang scheduled for the application.

The *MessageTransport* class defines whether a separate process will be delivering the message or whether the receiving process will incur the overhead of the transfer mechanism. The *MessageTransfer* class defines the management of message queues and how the data is actually moved between sender and receiver.

The *MessageSynchronization* class defines the message synchronization semantics. The concrete subclasses SpinLockBufferedMessageSynchronization, SemaphoreBufferedMessageSynchronization, SpinLockUserMessageSynchronization and SemaphoreProcessMessageSynchronization provide various implementations of this concept. The type of synchronization employed is used as a prefix to the concrete class. Since there is a restriction on how transport and synchronization may be mixed, the type of transport is also factored into the class: for example, the SpinLockBufferedMessageSynchronization class assumes a buffered transport mechanism with spin-locks for synchronization. MessageSingleTransfer, MessageDoubleTransfer, MessageUserTransfer and MessagePointerTransfer implement various schemes for moving a message from sender to receiver. For example, MessageDoubleTransfer copies a message from the sender's address space into the kernel and then from the kernel into the receiver's address space. Given the above framework it is possible to create a specific message passing system. For example, a collection of instances of the classes, KernelMessageSystem, BufferedMessageTransport, BufferedSpinLockMessageSynchronization, and MessageDoubleTransfer defines a hypercube style message passing system that is kernel based, lets the receiver process incur the cost of message transfer, provides process synchronization through spin-locks and copies the message into the kernel domain and then into the user address space (double copy semantics).

The message passing subframework makes reference to the Process framework class in its *MessageSemantics* abstract class. In addition, it references the MemoryObject framework class in its *MessageTransfer* abstract class.

The class hierarchies described above have been reached though iterative improvement. A concept was tried and when it did not work it was modified. One of the most important aspects of the design is *reuse*. For example, it is possible to combine a SpinLockMessageSynchronization class with either a MessageSingleTransfer to MessageDoubleTransfer. A earlier design merged the transfer and the synchronization hierarchies. This was clearly a mistake as the synchronization and transfer mechanisms are separate concepts. The old design forced the transfer mechanism to be replicated for both the semaphore and spin-lock modes of synchronization. Keeping these separate allows one instance of the transport mechanism to be used with several synchronization mechanisms.

4 Advantages of Frameworks

In this section, we describe some of the major advantages of using frameworks for designing an operating system. We demonstrate the advantages with examples from the *Choices* operating system.

- *Code Reuse* is normally achieved through the reuse of existing components and through polymorphism. With frameworks, code can also be reused through inheritance. The use of virtual functions in C++ for example, allows large bodies of code to be reused. In the persistent storage subframework, several abstract classes, including *PersistentStoreContainer* and *PersistentStoreDictionary*, implement all public operations. These operations are defined using several simple operations that subclasses must implement.

- *Design Reuse* is achieved in frameworks by reusing abstract concepts from one subframework in another framework. For example, the notion of MemoryObjects may be used in the persistent store subframework as well as in the virtual memory subframework. Frameworks allow this commonality to be described and reused.

- *Portability* is achieved in frameworks by separating machine-dependent parts of design from

the machine-independent parts. For example, an abstract class may have implementations of the machine-independent parts of a component, but machine-dependent parts will be specified by pure virtual functions that must be supplied by a subclass. For example, there is a CPU class that is machine-independent but it has concrete subclasses that are tailored to the SPARC, i386, NS32332, and MC68030 processors.

- *Rapid Prototyping* of different concepts is possible in frameworks because it supports code and design reuse. Code reuse and design reuse reduce coding time and design time, respectively. Once an abstract class has been built, it is only necessary to supply implementations of its pure virtual functions in a concrete class. For example, we were able to compare and contrast several message delivery mechanisms in the *Choices* message passing subframework.

- It is possible to customize for *performance*. For example, in the message passing subframework we allow synchronization through semaphores and spin-locks. For hypercube applications, the spin-lock version is a faster synchronization mechanism.

5 Techniques for building frameworks

In this section, we identify and describe some useful techniques for implementing frameworks. We provide example uses from *Choices*.

- *Abstract* classes provide generalized interfaces for *concrete* classes. *Concrete* classes are implementations of *abstract* classes. In *Choices*, the persistent storage framework involving PersistentStores, PersistentStoreContainers, and PersistentStoreDictionaries is used to introduce constraints on the partitioning of disks, provision of logical files, and implementation of file named.

- *Inclusion Polymorphism* refers to a subclass being a subtype of a superclass. This allows a subclass to be used wherever a superclass is expected. In *Choices*, all devices and device controllers are derived from a device-driver framework. Any device written to use an abstract controller interface may use any instantiation of a controller such as a SCSI bus controller. Further, given a request for a particular implementation of an I/O interface, the system is free to bind that request to any convenient implementation of the interface provided that the class of the object requested and class of the service offered satisfies the subtyping requirement.

- *Constraints* are descriptions of relationships between abstract classes of frameworks or the relationship between concrete and abstract classes within a framework. The use of constraints is most evident in how instances of concrete classes are combined. For example, in the message passing subframework a buffered message interface must be used with a buffered message delivery mechanism. It cannot be combined with a process message delivery mechanism.

- *Dynamic code loading* allows one to specify an abstract class when the system is designed and add a concrete descendant class of the abstract class at run-time. For example, a *Choices* device driver consists of a *DevicesController* class and a number of *Device* classes. A new device driver can be added to the system by loading the concrete subclasses of the *DevicesController* and the *Device* classes that form the device driver.

- *Delayed Binding* is the ability to determine dynamically the methods to which an object responds (often referred to as the signature of the object). In object-oriented systems this is not known until run-time. In C++, delayed binding is a result of using virtual functions. All abstract classes in *Choices* use virtual functions.

- *Conversion* allows objects to be changed at run-time into other objects. Conversion does not modify the original object; instead, a new one is created using the data of the old object. Subclasses of ProxiableObject implement the conversion process by responding to the `asA` message[7]. It takes an argument that may be the name of either a concrete or an abstract class and returns a reference either to an instance of the argument or an instance of a concrete subclass of the argument, respectively. The the `asA` method uses the `supports` method to ensure that the underlying data is compatible with the given class. For example, in the *Choices* device management subsystem a serial line can be converted to an input stream and an output stream. In the persistent storage system, a persistent store can be converted into a persistent object.

6 Conclusions

Our experience has shown that an object-oriented framework is an effective technique for designing a complex software system such as an operating system. In this paper, we have shown how complicated components of the operating system can be designed and the interfaces between the different components defined using frameworks. We also show how a framework for a system can be used to help design the subframeworks required for subsystems of the system. Parts of the framework are refined and specialized for the subframework. There are, however, critical parts of a framework that have only informal definition. In particular, we found that a suitable notation for expressing many of the informal constraints between components of a system is lacking. The relationships that can be expressed by the classes in C++ was insufficient to express all the constraints that accompanied the design of the *Choices* frameworks.

References

[1] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987.

[2] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, Reading, Massachusetts, 1989.

[3] Kurt J. Hebel and Ralph E. Johnson. Arithmetic and double dispatching in Smalltalk-80. *Journal of Object Oriented Programming*, March/April 1990.

[4] Nayeem Islam and Roy Campbell. The performance of message based applications on an object oriented operating system. Technical Report UIUCDCS-R-88-1455, University of Illinois Urbana-Champaign, April 1991.

[5] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS-R-91-1696, University of Illinois at Urbana-Champaign, May 1991.

[6] Panagiotis Kougiouris. A Device Management Framework for an Object-Oriented Operating System. Master's thesis, University of Illinois at Urbana-Champaign, August 1991.

[7] Peter W. Madany. An Object-Oriented Approach towards A General Model of File Systems. Technical Report UIUCDCS-R-90-1607, University of Illinois at Urbana-Champaign, Dec 1990.

[8] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference*, San Francisco, California, April 1990.