A Class Hierarchy for Building Stream-Oriented File Systems[1]

P. MADANY, R. CAMPBELL, V. RUSSO AND D. LEYENS

University of Illinois at Urbana-Champaign

Abstract

This paper describes the object-oriented design and implementation of a family of stream-oriented file systems under UNIX and under an object-oriented operating system called *Choices*. A class hierarchy provides an object-oriented taxonomy of the algorithms and data structures used in the design of this family. The family includes the System V file system, the 4.2 BSD file system, and the MS-DOS file system.

The class hierarchy has been developed by a series of experiments that are designed to lead to a framework for object-oriented file systems. The class hierarchy for stream-oriented file systems is the product of the second experiment in this series in which we revised a class hierarchy for UNIX-like file systems[MLRC88] to include the MS-DOS file system. We describe the hierarchy, how it evolved from the first experiment to the second, and review the lessons that we have learned from the two experiments.

1 INTRODUCTION

The *Choices* operating system architecture [CJR87, CRJ87, RJC88] is motivated by the difficulties of building portable and extensible operating systems for high-performance multiprocessor and uniprocessor computers. The solution we adopt to the organizational problems inherent in such systems is to design *Choices* as an object-oriented system. In addition, *Choices* provides application programs with an object-oriented system interface.[2] Associated with the operating system is an extensive class hierarchy that defines the interfaces and components of the system [RJC88, RC89].

Choices presents an object-oriented environment to applications. Object method invocation is used to invoke both the operating system services and the services provided by "server objects" running as applications on the system. *Choices* provides secure method invocation on server objects by using virtual memory protection mechanisms

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

[2] *Choices* is implemented in the C++[Str86] programming language. C++ provides a sufficiently efficient implementation of classes and inheritance to support operating system construction.

to restrict access to these objects. Many facilities implemented by such server objects would belong in a kernel of a more "traditional" operating system like UNIX. The file system is one such example of a traditional kernel service. In *Choices*, we have chosen to implement the file system as a collection of server objects; each object implements an independent component of the file system.

The file system is a major operating system subsystem. Following the design goals of *Choices*, we are building a spectrum of file systems to enhance the customization of the operating system family to applications. An application may use a customized file system that has components which are tailored to improve its performance, to optimize its utilization of storage, or to provide compatibility with other file systems. Stream-oriented file systems comprise a major category of file systems and form the basis for our initial work. In the future, we plan to examine record-oriented file systems, data bases and object-oriented file systems.

One *Choices* research goal is to demonstrate *code reuse* in systems software through object-oriented programming. Similar designs permit reuse of common algorithms and data structures in the various components and versions of operating systems. Class hierarchies and inheritance offer an excellent mechanism to achieve reuse because they help to organize related concepts and simplify code sharing. Although many existing file systems share design concepts, current software practices often lead to dissimilar implementations. For example, the 4.2 BSD and System V versions of the UNIX file system have many common design features yet their implementation differences require separate development and maintenance.

Another goal of *Choices* is to develop *object-oriented file systems* that users may extend and customize for their particular applications. The work we describe here is a milestone in this research. It is an object-oriented design and implementation study of the integration of several existing stream-oriented file systems into one class hierarchy with a set of abstract access protocols that may be used on any of the specific file systems. The resulting software provides a platform for the testing and development of applications while new file systems for *Choices* are being developed.

In this paper, we describe the results of a second experiment in designing object-oriented file systems. In the first experiment, we built a class hierarchy to represent UNIX-like file systems[MLRC88]. In this second experiment, we revised the class hierarchy to include the major components of stream-oriented file systems. The hierarchy supports the file system implementation of System V UNIX [Tho78], 4.2 BSD UNIX [MJLF84] and MS-DOS [Nor85] under *Choices*. We chose these three stream-oriented file systems based on their popularity and the range of data structures they use.

Our work has yielded significant results. First, the stream-oriented file systems are organized into a taxonomy of data structures and algorithms. Second, the systems

are portable and extensible. Third, it is possible to combine file system components from the various implementations to produce hybrid systems. For example, the efficient BSD disk allocation methods and larger block sizes can be combined with the System V directory structure to yield a system with higher throughput that is still compatible with user level code that relied on the System V directory structure.

The paper is organized as follows: Section 2 provides necessary background for the rest of the paper; Section 3 presents an overview of the layers of our solution; the next five sections describe each layer along with the abstract classes that define interfaces between them and the concrete subclasses that provide specific behaviors; Section 8 compares our solution to previous work; Section 9 discusses possible future directions; and Section 10 forms our conclusions.

2 BACKGROUND

Simplicity characterizes the design of the standard *System V file system*[Tho78] that is part of many commercially available UNIX operating systems. UNIX files are sequences of randomly accessible bytes accessed via a standard interface: **read**, **write**, and **lseek**. This interface conceals hardware device dependencies and hides block allocation and mapping. Because the operating system does not explicitly require record structures for files, the output of most UNIX tools can be the input of others. Nevertheless, any tool can impose a structure on a file. The random access feature allows even complex record structures to be imposed on specific files when needed.

Each file system is stored on a physical disk partition. A file system consists of a header, a disk block allocation table, an array of inodes that describe individual files, and the data blocks of those files. While file systems cannot span disk partitions, a single directory tree contains all the files on all the file systems. The directory tree hides individual disks and partitions from the user.

An *inode* is a data structure that describes and specifies the access rights to an individual file. Each inode contains the file's size, reference count, ownership, access rights, timestamps, and the set of disk blocks that hold the file's data. Within a UNIX system, each file is uniquely identified by a partition and inode array index number, called the *inumber*. *Directories* are sequences of records containing *name-inumber* pairs. Because directories contain inumbers instead of inodes, files can be in more than one directory at a time.

The System V file system's performance is marked by two characteristics: high disk space utilization and low CPU overhead per block transferred. However, there are some deficiencies in both performance and features that have been addressed by the design of the 4.2 BSD file system.

The *4.2 BSD file system*[MJLF84] and its subsequent 4.3 BSD revision are used in many academic institutions. It maintains the same basic interface as System V, but adds optimizations and extensions. The throughput of the System V file system is dominated by disk latency. To minimize this latency and thereby improve throughput, the BSD developers increased file block sizes and improved disk block allocation policies. To maintain the high disk space utilization of System V, BSD added the capability to fragment the last block in a file. To achieve the performance improvements, the BSD developers sacrificed the internal simplicity of the System V file system.

Two major features added by the BSD file system are symbolic links and long file names. Symbolic links allow users to create directory entries that refer to the files of different file systems. In System V, file names are restricted to 14 characters because of the fixed-size record structure used for directory entries. The BSD file system uses a variable-size record that allows file names up to 255 characters long.

The *MS-DOS file system*[Nor85] is part of another popular commercially available personal computer operating system. MS-DOS files are similar to UNIX files and support **read**, **write**, and **lseek**. The MS-DOS system also supports a UNIX-like tree-structured directory hierarchy.

Though an MS-DOS file system is superficially similar to UNIX, its internal structure is fundamentally different. MS-DOS does not use a UNIX-like inode array. Instead, directory entries contain most of the file's control information. This implementation technique restricts a file so that it may appear in only one directory. MS-DOS directory entries are fixed sizes. Thus, variable-sized file block mapping information cannot be kept with the other control information. Instead, all block mapping information for an MS-DOS file system is kept within a single data structure called a File Allocation Table (FAT). The organization of the FAT yields poor random access performance for large files. MS-DOS file sizes are recorded using (1) an end-of-file character, (2) the size in a file descriptor in the directory entry, and (3) a count field of the length of a file's FAT chain. UNIX uses a single mechanism. Though neither so elegant as the System V file system nor so powerful as the BSD file system, the MS-DOS file system survives well in the "hostile" environment of personal computers and floppy disks.

The following sections discuss the layers, components, and class hierarchy in our system and are followed by a comparison with our previous work and directions for future work.

3 FILE SYSTEM LAYERS

The *Choices* stream-oriented file system is organized into four conceptual layers as shown in Figure 1. Each layer provides an abstraction of the underlying physical storage devices used for file systems. The system is designed and programmed using object-

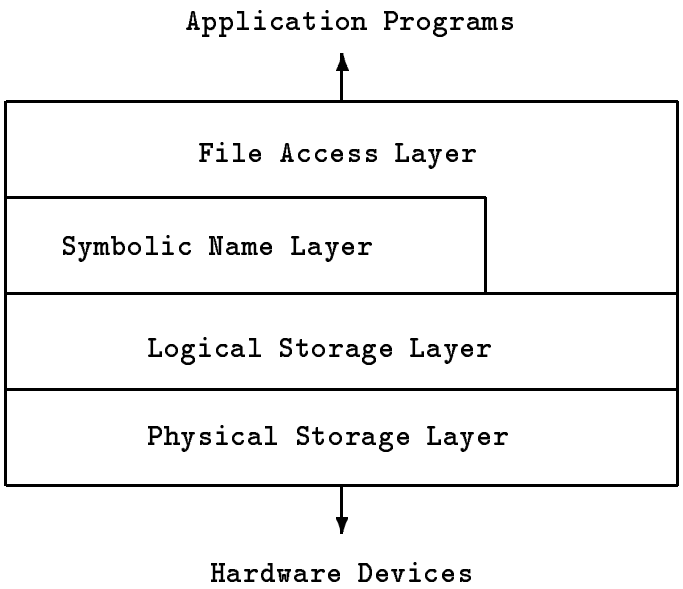

Figure 1: The *Choices* File System Layers.

oriented techniques. A class hierarchy represents the data structures and algorithms used in the layers. Abstract classes introduce the data access protocols that are needed between the layers, whereas concrete classes represent implementations of the protocols at different levels within the system and in different versions of the system.

The layers and class hierarchy correspond to orthogonal design issues. The layers simplify and organize the way in which file system services are provided[BS87]. Object-oriented design techniques[Sny81][Mey87] represent the layers, encapsulate design decisions and data structures, and organize the inheritance of common data structures and algorithms no matter in which layer they are used. Inheritance permits code reuse between the layers and simplifies maintenance. The file system has the following four layers:

- The *Physical Storage Layer* defines abstract protocols for accessing physical storage devices like disks and introduces concrete implementations of these protocols for specific devices. Access is defined in terms of the hardware device's physical blocks. The physical storage layer supports the partitioning of a hardware device into several physical stores, Each store supports an independent and possibly different file system.

- The *Logical Storage Layer* defines abstract protocols for accessing logical storage residing on a physical store. Access is defined in terms of an extensible sequence of blocks. Concrete implementations of these protocols support the particular mapping of logical storage onto a physical store required by a given file system. Multiple logical stores correspond to files and directories in the file system and

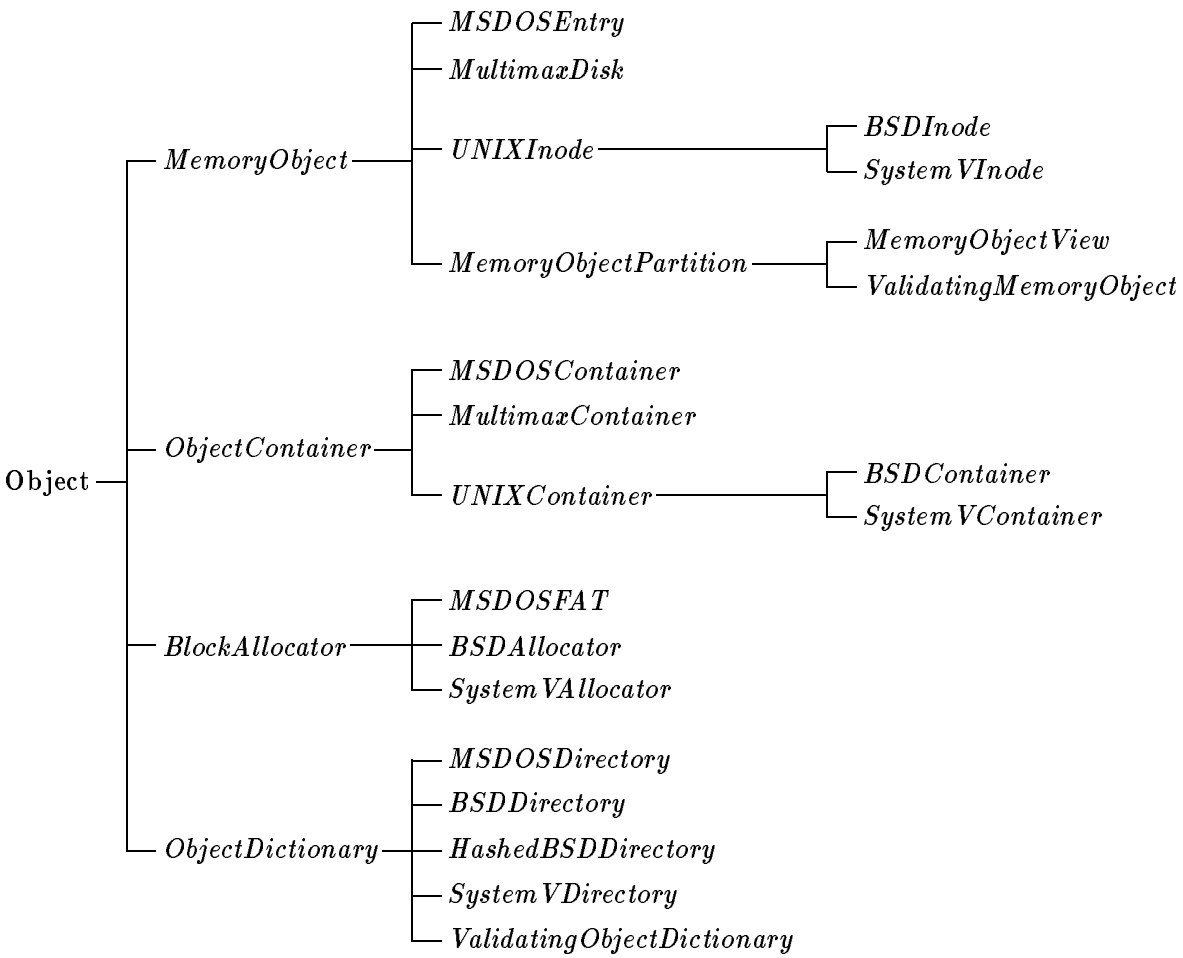

Figure 2: The *Choices* File System Class Hierarchy.

are mapped to a particular physical store. Physical storage is allocated and deallocated to support the access operations defined for logical stores.

- The *Symbolic Name Layer* associates a symbolic name with a logical store. The symbolic name is used to gain access to the contents of a logical store.
- The *File Access Layer* defines file access protocols for applications. It uses the symbolic name layer to provide applications with a unified naming scheme for referencing files in the file system. It defines a file by introducing a byte addressable stream-oriented access protocol for a logical store.

Six classes form the basis for the four layers implementing the stream-oriented file system.

The *FileSystemInterface* and *File* classes define a standard protocol that applications use to access the file system and individual files. Figure 2 shows the class hierarchy

for The *ObjectDictionary*, *ObjectContainer*, *BlockAllocator*, and *MemoryObject*. These are abstract classes defining protocols for accessing aggregates of named logical stores, managing groups of logical and physical stores, managing allocation of storage blocks, and accessing the stored data. Concrete subclasses specialize each of these four classes to implement System V, BSD 4.2, and MS-DOS file system behavior. Figure 3 shows how instances of these classes are organized into layers.

Instances of the file system classes represent access mechanisms to the *contents* of the file system. The instances are created dynamically when a request to access a file is made. Methods applied to the instances may create, delete or manipulate the contents of a file. The contents may be stored on secondary storage; in main memory; or, in a distributed system, on a different node of a network. The instances are removed when access to the contents of their respective files are no longer required. A file persists in storage even when there are no instances providing access to its contents.

In general, instances of all six classes are required in order to access a particular file. However, application programs interact directly only with a FileSystemInterface object and one or more FileStream objects. In turn, these objects interact with with instances of MemoryObjects, ObjectDictionaries, and ObjectContainers. MemoryObjects in the logical storage layer interact with BlockAllocators.

The following sections describe the component classes of each layer and the interfaces presented by instances of the component classes.

4 PHYSICAL STORAGE LAYER

The physical storage layer implements the file system device I/O techniques, and I/O scheduling and control[BS87]. Two classes of objects belong to this layer: *MemoryObjects* and *ObjectContainers*. MemoryObjects implement the abstraction of a physical store and provide an interface to and encapsulate the details of hardware devices such as magnetic disks, optical disks, random access memory, and networks of remote storage devices. Large storage devices are often partitioned[3] into smaller physical stores to provide more convenient file system management. An ObjectContainer provides an interface to and encapsulates the details of such hardware partitions.

4.1 MemoryObjects

The abstract class *MemoryObject* defines an access protocol for both physical and logical storage. Physical or logical storage is modelled as a sequence of identically sized units.[4] We will extend the use of MemoryObjects to logical storage in the next section.

[3] A partition is usually a contiguous section of a disk.

[4] To simplify implementations, the size of a unit is always an integer power of two.

The most important methods of class MemoryObject are **read** and **write**. These provide access to multiple contiguous units of storage. Objects that communicate with MemoryObjects via these methods must supply a unit number, the number of contiguous units, and a buffer address[RC89]. MemoryObjects also contain methods to report their *length* and to report the size of their units.[5]

The *Disk* subclasses of MemoryObject provide an abstract software interface to disk and controller hardware. Instances of the Disk subclasses use their corresponding disk's sector size as their unit size. Currently the only disks supported are those using an Encore Multimax disk controller. The *MultimaxDisk* class is a concrete Disk subclass that contains the controller specific code. In "traditional" operating systems, the methods of this class would be the disk driver routines.

To support storage device partitioning, the *MemoryObjectPartition* subclass provides a window into a storage device represented by an instance, usually of a Disk subclass, of MemoryObject. The offset and size of this window can range from zero up to the size of the underlying storage device. Since all currently implemented stream-oriented file systems support the clustering of disk sectors, a MemoryObjectPartition uses a clustering factor to convert between a partition's unit size and the disk's sector size. In general, several instances of a MemoryObjectPartition will be used to access a Disk. However, their windows cannot overlap.

4.2 ObjectContainer

An instance of the abstract class *ObjectContainer* manages the contents of a physical or logical store as a collection of stores. It facilitates access to these stores by creating and deleting instances of MemoryObject, ObjectContainer, and ObjectDictionary. In turn, each of these instances provides access to a member of the collection using that member's MemoryObject methods.[6] Both the physical and logical storage layers introduce subclasses of ObjectContainer.

The concrete class *MultimaxContainer* is a subclass of ObjectContainer in the physical storage layer and manages a physical storage MemoryObject like a Disk. It divides the physical store into *partitions*. Each partition is associated with a logical storage layer subclass of ObjectContainer that can be instantiated to provide access to the contents of the partition. In the current implementation, the logical storage may correspond to a BSD, System V, or MS-DOS file system. The MultimaxContainer methods create and delete instances of MemoryObjectPartition and logical storage layer ObjectContainers on demand. These instances provide access to the contents of a partition using the appropriate file system physical disk organization. The MultimaxContainer object maintains an indexed table of these instances. A partition table and logical storage

[5] MemoryObjects for logical storage also permit the length to be changed.

[6] In most cases, the member's MemoryObject will be in a lower layer than the instance.

type is stored in physical storage and the methods of MultimaxContainer check this information against requests to access a partition or logical storage to ensure consistency.

The **open** method of MultimaxContainer is is inherited from its superclass and has arguments that include an index and a partition or logical storage type. First, it creates an instance of MemoryObjectPartition that provides access to the partition corresponding to the index. Then, if the argument specifies a logical ObjectContainer type, it creates an instance of the appropriate ObjectContainer subclass to manage the partition and returns a reference to that instance. Otherwise, it returns a reference to the MemoryObjectPartition. The reference is also stored in the indexed table. Further calls to **open** with the same index and type cause the MultimaxContainer to invoke the **reference** method of the instance associated with that index and to return a reference to that instance. The reference method increments an internal reference count. Methods of the instance of MemoryObjectPartition or logical storage layer ObjectContainer are used to access the contents of the partition without further assistance from the MultimaxContainer methods.

When access to an instance of a MemoryObjectPartition or logical storage layer ObjectContainer is no longer required, the instance's **unreference** method is invoked. This method decrements the internal reference count for the instance. If the count reaches zero, the method invokes the **close** method of the MultimaxContainer that created it before deleting itself.

5 THE LOGICAL STORAGE LAYER

The logical storage layer implements the physical organization methods of a file system [BS87]. It defines access to logical stores that are extensible sequences of blocks. Logical stores may be created, deleted, read, and written. The logical storage layer maps access to the contents of a logical store onto access to the contents of its physical store. It allocates and deallocates the blocks of physical storage that are needed to support the creation, deletion, and writing of logical stores.

Following the methodology of object-oriented design, the logical and physical storage layers use the same abstract protocols for accessing the contents of storage; the protocols defined by the classes MemoryObject and ObjectContainer. The logical storage layer defines UNIXInode and MSDOSEntry as subclasses of MemoryObject and UNIXContainer and MSDOSContainer as subclasses of ObjectContainer. The UNIX classes are further subclassed for BSD and System V implementations. The layer defines the ValidatingMemoryObject as a subclass of MemoryObjectPartition to protect the contents of a logical store from incorrect access. The layer also introduces class BlockAllocator and its subclasses MSDOSFAT, BSDAllocator and SystemVAllocator to manage the physical storage block allocation needed by extensible storage objects.

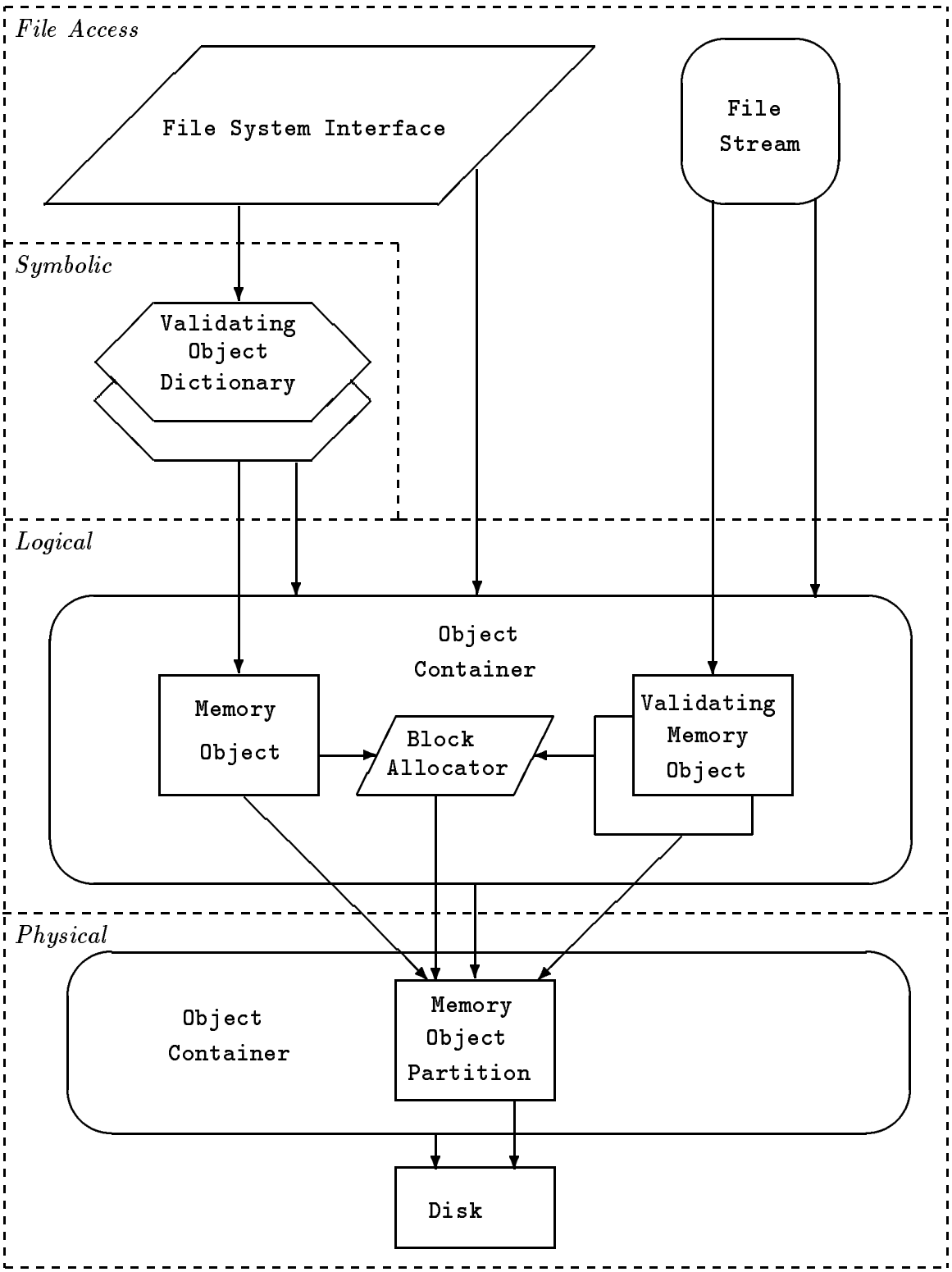

Figure 3: The *Choices* File System Framework

5.1 MemoryObjects

The logical storage layer defines logical stores for UNIX and MS-DOS files using the *UNIXInode* and *MSDOSEntry* subclasses of MemoryObject. These logical stores allow their length to be set both in units and bytes. Writing past the end of a logical store automatically increases its length up to the limitations of physical storage. The subclasses encapsulate file system version dependent code.

The UNIXInode class is subclassed into *BSDInode* and *SystemVInode* to represent the implementation differences between System V and 4.2 BSD files. *BSDInode* and *SystemVInode* inherit most of their code from UNIXInode. Both the **read** and **write** methods are implemented by UNIXInode. Even though the BSDInode class redefines **write**, it still calls UNIXInode's write method after performing some BSD-specific optimizations. Both subclasses define the mapping of inode structures onto physical storage blocks used by the read and write methods. To achieve this mapping, both subclasses set and retrieve direct and indirect block pointers.

The logical stores for MS-DOS files are defined by *MSDOSEntry*, a subclass of MemoryObject. Class MSDOSEntry inherits its interface from MemoryObject but encapsulates several MSDOS-specific details.

Multiple instances of FileStream in the file access layer may access a single logical store. Each FileStream has its own set of access rights for that store and the stores must be protected from incorrect access. ObjectContainers instantiate *ValidatingMemoryObjects* to implement an access right protection mechanism. Each instance of ValidatingMemoryObject checks FileStream requests against the permitted forms of access and delegates legal requests to the logical store.

For example, UNIX or MS-DOS files may be opened for reading only. Protection is provided by associating an instance of a *ValidatingMemoryObject* with the access to the contents of an open logical store. The instance delegates read accesses to the appropriate instance of an UNIXInode or MSDOSEntry.

5.2 ObjectContainers

The subclasses of ObjectContainer class in the logical storage layer manage collections of logical stores and maintain a table of indices, types, and references. Type information indicates the subclasses of ObjectContainer, MemoryObject, and ObjectDictionary that manage the logical stores that are mapped into a partition.

The ObjectContainer methods **open**, **create**, and **close** are implemented in the superclass and inherited by all subclasses. These methods are used to instantiate instances of the appropriate subclasses of ObjectContainer, MemoryObject, and ObjectDictionary.

The superclass is abstract and does not contain information about the physical storage mappings used by its subclasses or the subclasses of MemoryObject and ObjectDictionary. Thus, in order to be able to manipulate information in physical storage, the abstract class introduces the implementation specific protocols basicOpen, basicCreate, basicClose, basicDictionary, basicContainer, and basicRootId. When the abstract ObjectContainer methods open, create, and close need to manipulate information in physical storage using a mapping that depends upon the subclass of the instance, they invoke the appropriate method defined by the subclass using the implementation specific protocol. In this way, the ObjectContainer superclass methods can both be powerful and inherited, enhancing the degree of reuse within the file system.

For example, access to a BSD file is achieved in several steps. At boot time, the kernel creates a MultimaxContainer for each Disk. At mount time, MemoryObjectPartition and BSDContainer objects are instantiated by the open method of the MultimaxContainer. Subsequently, a BSD inode instance is created by the open method of the BSDContainer object. The BSDContainer object checks its index table to determine if the instance of BSDInode, subclass of MemoryObject, has already been instantiated. If not, it invokes its basicOpen method [7] to read the physical storage representing a BSD inode and to instantiate an appropriate object. The open method stores a reference to the BSDInode object in the index table and returns a reference to an instance of a ValidatingMemoryObject that delegates legal access requests to the BSD inode.

As with BSD and System V MemoryObject subclasses, much of the code for the version-specific container classes, *BSDContainer* and *SystemVContainer*, is shared via inheritance from the common superclass *UNIXContainer*.

Many of the methods of BSDContainer and SystemVContainer perform identical functions but use different data structures. In these cases, code and design can be shared as long as the differences can be hidden. For example, the readInode and writeInode methods use a mapping from inumbers to physical blocks. The mapInumber defines the protocol for this mapping in UNIXContainer. Each subclass then implements the method in a different way. This allows both the readInode and the writeInode methods to be implemented in UNIXContainer and to be inherited by the subclasses.

The MS-DOS container class *MSDOSContainer* implements code for data structures that have little in common with UNIX file systems yet it is analogous to its sibling class UNIXContainer. It does inherit the superclass ObjectContainer methods and this reuse simplifies maintenance.

[7] The virtual function feature of C++ assures that the open method will use the appropriate subclass implementation of basicOpen.

5.3 BlockAllocators

MemoryObject methods in the logical storage layer use the allocate and free methods defined by the abstract class *BlockAllocator* to manage the storage units of the physical storage layer. Allocate returns the index of a unit of physical storage that has been reserved for storing data and free releases a unit of storage that is no longer needed. The *SystemVAllocator* subclass uses a free list to manage storage units, the *BSDAllocator* subclass uses bitmaps, and the *MSDOSFAT* subclass uses a File Allocation Table.

6 SYMBOLIC NAME LAYER

The symbolic name layer supports the naming and aggregation of of the logical storage introduced by the logical storage layer. It corresponds to the directory layer in conventional file systems [BS87]. All objects in this layer are instances of subclasses of *ObjectDictionary*.

An ObjectDictionary is an aggregation facility for logical stores that maps a list of symbolic keys to the indices used by ObjectContainers. Within any dictionary, the keys must be unique, but several keys may map to the same index. ObjectDictionary is subclassed in order to store the keys and indices on an underlying logical store in a format that conforms to the directory structure of a particular file system. An instance of ObjectDictionary contains a reference to the instance of ObjectContainer that created it.

The lookup method takes a key and a type as arguments and, if the key is found, returns a reference to an instance of the appropriate class. It obtains this reference by invoking the open method on its ObjectContainer using the index that corresponds to the key. Two methods, create and add, allow objects to be added to dictionaries. The create method performs the same function as lookup for existing keys. If the key does not exist, however, the method returns the reference that it obtains by invoking the create method on its ObjectContainer. The add method takes a symbolic key and a reference to an instance of a ObjectContainer, ObjectDictionary, or MemoryObject as arguments. It converts the reference to the corresponding object index, by invoking the instance's idNumber method, and inserts the key and index into the dictionary.

Pairs of keys and object indices are deleted from a dictionary by the remove method. To prevent file loss and directory corruption, keys for ObjectDictionaries cannot be deleted with the remove method. Instead, each dictionary has a destroy method that will correctly dispose of an empty dictionary. The keys method returns a dictionary's entire set of symbolic keys in a single call.

As with the MemoryObject subclasses used in the logical storage layer, ObjectDictionary subclasses are divided into two categories, those that contain file system version specific code and those that provide protection.

The four subclasses *BSDDirectory*, *HashedBSDDirectory*, *SystemVDirectory*, and *MS-DOSDirectory* encapsulate data structures belonging to directories of the three implemented file systems. BSD directories use blocks with several variable-length entries, each containing pairs of filenames and inumbers. These entries can belong to a sequential list or a hash table. System V directories use lists of fixed length entries, each also containing filenames and inumbers. MS-DOS directories use lists of fixed length entries, each containing a file name and file control information.

Despite differences between the organization and contents of the directories, the **lookup**, **create**, and **add** methods for all four subclasses are inherited from ObjectDictionary. The definition of two protected[Str86] methods **find** and **insert** facilitates this inheritance. Since little code savings would result from sharing the other public methods **remove**, **destroy**, and **keys**, these methods are implemented in each subclass.

The special purpose subclass *ValidatingObjectDictionary* is similar to the MemoryObject subclass ValidatingMemoryObject and provides an interface that checks access rights before delegating requests to an instance of ObjectDictionary.

7 FILE ACCESS LAYER

In a *Choices* system, application programs interact with a FileSystemInterface object and with FileStream objects to gain access to logical stores and their contents.

7.1 FileSystemInterface

An instance of class *FileSystemInterface* provides an application program with a unified name space. It parses a *path name* into a list of symbolic names. Each symbolic name is interpreted, one after the other, by the instance of ObjectDictionary specified by the previous names in the path name. The FileSystemInterface has a *mount* method that takes a partition index and path name as an argument. It creates a mount table entry for the root dictionary of the logical ObjectContainer that corresponds to the partition index. The mount table maintains references to all ObjectContainers that have been instantiated as a result of the application's file system requests. For pathname resolution, the FileSystemInterface uses the mount table to organize the root dictionary of active ObjectContainers into a single tree. For convenience, it maintains a reference to the root of this tree and to a "current" dictionary.

The public methods of the FileSystemInterface are similar to several UNIX system calls including: **open**, **creat**, **link**, **unlink**, **mkdir**, **chdir**, and **stat**. These methods operate on and return references to instances of FileStream or ObjectDictionary classes.

7.2 FileStreams

Although the logical storage layer provides access to extensible sequences of data blocks, it does not provide the stream-oriented interface required by applications. The stream-oriented file systems we have implemented provide byte-addressability and the concept of a "current file position", i.e. the location within the file where the next read or write will occur. The file access layer in *Choices* introduces the *FileStream* class to provide these services.

Application programs can **read** and **write** sequentially or use random access. Each operation may read or write multiple bytes. Because the FileStream class implements the concept of a current file location pointer, it has a **seek** method to allow programs to position this pointer. The **read** and **write** methods also update this file pointer. Each instance of FileStream communicates with a corresponding logical store. Since MemoryObjects only provide a read/write block interface, FileStreams also manage the buffering of portions of the data of their associated logical store.

7.3 Memory Mapped Files

Direct access to data stored on secondary storage via FileStreams suffers from access latency and limited device throughput. The *Choices* virtual memory sub-system provides an alternate method of data access to applications by using physical memory to cache the data of a logical store. This cacheing does not change the functionality of a logical store, rather it allows data to be accessed directly by the instructions of the computer.

The class *MemoryObjectCache* provides an abstract protocol for mapping portions of a logical store (also called a logical memory[RC89]) into physical memory. Subclasses of the MemoryObjectCache class implement specific dynamic *cacheing* techniques.

8 COMPARISON WITH PREVIOUS WORK

A comparison of the current design with an earlier one[MLRC88] provides an interesting insight into the design and development evolution of an object-oriented system.

The major reasons for changing the design include:

1. incorporating a non-UNIX file system, i.e. the MS-DOS file system;
2. the desire to uncover and use more abstractions in the absence of a multiple inheritance feature in C++; and
3. several months of heavy use of the original class hierarchy.

In our earlier work, we implemented dictionaries using the directory concept of the UNIX file system. A directory (dictionary) was thus implemented as a specially structured file that maps filenames to inumbers. Consequently, we classified a directory as a subclass of class File, which was a subclass of MemoryObject. When we added support for MS-DOS directories, we found we needed a more abstract view of a directory (an ObjectDictionary). Not only did this new abstraction accommodate the MS-DOS style of directories, but it also allowed all concrete directory classes to share a single design. Furthermore, the ObjectDictionary class allows half of the public method code to be shared by all concrete subclasses.

After completing our original class hierarchy, we considered using multiple inheritance to combine ObjectDictionary and MemoryObject attributes from separate class hierarchies. By simply separating a directory into two components, the ObjectDictionary and the underlying MemoryObject, we eliminated the need for multiple inheritance without eliminating any possibilities for code sharing.

In our first attempt we also made the UNIXContainer class a subclass of MemoryObjectPartition, since each UNIXContainer controlled one partition of a disk. Before we separated this functionality into an ObjectContainer, a BlockAllocator, and an underlying MemoryObjectPartition, UNIXContainer objects were unnecessarily complex. Our current design easily accommodates both UNIX and MS-DOS subclasses of ObjectContainers and contains much common code. We also separated the block allocation functions from the ObjectContainer class and created an new abstract class BlockAllocator to encapsulate these functions.

Despite some of the major revisions we made to our original design, the first two file systems served our project well as an access mechanism for test programs and sample data and as a backing store for the memory-management system.

The addition of the abstract classes ObjectContainer, BlockAllocator, and ObjectDictionary changed the class hierarchy for the file system considerably. However, most of the original code was reused in the new design. This is a testimonial both to the quality of the original design and to the data encapsulation capabilities of object-oriented programming. We expect that the incorporation of more diverse file systems, to be discussed in the next section, and continued use of our newly designed class hierarchies will force continued evolution of our design.

9 FUTURE WORK

We plan to extend our library of stream-oriented file systems with record-oriented file systems, customizable file systems, and databases. The file system design shows that it is possible to build a class hierarchy with diverse components and designs. Record-oriented file systems and stream oriented file systems have many incompatibil-

ities [Dem88], but also share many features. Our next design step is to incorporate record-oriented file system structures into the hierarchy.

10 CONCLUSIONS

The *Choices* project is examining the role of object-oriented design and programming in the construction of system software. We have designed a new operating system and reimplemented several file systems successfully using these approaches. Throughout the implementation we have been impressed with the ease with which object-oriented design unites related concepts. This has had many major benefits, particularly in code development, debugging, and maintenance. In this paper, we discuss the design of a class hierarchy for three well-known, stream-oriented file systems.

This paper is a contribution to both the taxonomy of algorithms and data structures used in stream-oriented file systems and the evolution of class hierarchical, object-oriented file systems. In particular, we show that:

- a careful choice of the methods defined and inherited in a class hierarchy increases code and design sharing and reuse, even when implementing versions of a system that differ greatly.
- by restructuring our previously built hierarchy, not only were we able to incorporate a new file system, MS-DOS, but we were also able to discover and use more powerful abstractions.
- object-oriented code is reusable, even when major structural changes are made to the organization of code,
- object-oriented design promotes customization, since the library of file system components that we built allow hybrid file systems to be constructed that use particular components from different file systems.
- object-oriented code is portable and retargetable since all three file system implementations that we built are independent of UNIX and MS-DOS and can, potentially, be ported to many other operating systems besides *Choices* and UNIX.

To conclude, this paper describes a complete implementation of BSD, System V, and MS-DOS file systems as a portable package written in C++. Our next step is to add record-oriented features to the hierarchy and to build customizable file systems for *Choices* based on our experience of building these stream-oriented file systems.

REFERENCES

[BS87] Lubomir Bic and Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, 1987.

[CJR87] Roy Campbell, Gary Johnston, and Vincent Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[CRJ87] Roy Campbell, Vincent Russo, and Gary Johnston. The design of a multiprocessor operating system. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, 1987. Also Technical Report No. UIUCDCS–R–87–1388, Department of Computer Science, University of Illinois at Urbana-Champaign.

[Dem88] Richard A. Demers. Distrbuted Files for SAA. *IBM Systems Journal*, 27(3):348–361, 1988.

[Mey87] Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, pages 50–64, March 1987.

[MJLF84] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MLRC88] Peter W. Madany, Douglas E. Leyens, Vincent F. Russo, and Roy H. Campbell. A C++ Class Hierarchy for Building UNIX-Like File Systems. In *Proceedings of the USENIX C++ Conference*, Denver, CO, October 1988.

[Nor85] Peter Norton. *The Peter Norton Programmerś Guide to the IBM PC.* Microsoft Press, 1985.

[RC89] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Submitted to OOPSLA '89*, 1989. Also available as University of Illinois Technical Report.

[RJC88] Vince Russo, Gary Johnston, and Roy H. Campbell. Process Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '88*, San Diego, Ca., September 1988.

[Sny81] Lawrence Snyder. Using types and inheritance in object-oriented programming. *IEEE Transactions on Computers*, March 1981.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1986.

[Tho78] K. Thompson. Unix implementation. *Bell System Technical Journal*, 57(6):1931–1946, July 1978.