Organizing and Typing Persistent Objects within an Object-Oriented Framework*

Peter W. Madany and Roy H. Campbell
University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Avenue, Urbana, IL 61801 USA

Email:	madany@cs.uiuc.edu	roy@cs.uiuc.edu
Phone:	1-217-333-7937	1-217-333-0215
Fax:	1-217-333-3501	1-217-333-3501

Abstract

Conventional operating systems provide little or no direct support for the services required for an efficient persistent object system implementation. We have built a persistent object scheme using a customization and extension of an object-oriented operating system called *Choices*. *Choices* includes a framework for the storage of persistent data that is suited to the construction of both conventional file systems and persistent object systems. In this paper we describe three areas in which persistent object support differs from file system support: storage organization, storage management, and typing.

Persistent object systems must support various sizes of objects efficiently. *Customizable containers,* which are themselves persistent objects and can be nested, support a wide range of object sizes in *Choices*. Collections of persistent objects that are accessed as an aggregate and collections of light-weight persistent objects can be clustered in containers that are nested within containers for larger objects. Automated garbage collection schemes are added to storage management and have a major impact on persistent object applications. The *Choices* persistent

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

object store provides extensible sets of persistent object types. The store contains not only the data for persistent objects but also the names of the classes to which they belong and the code for the operations of the classes.

Besides presenting persistent object storage organization, storage management, and typing, this paper discusses how persistent objects are named and used within the *Choices* persistent data/file system framework.

1 Introduction

Efficient persistent object system implementations require services that are not usually found in conventional operating systems. We have built a persistent object scheme that is supported by the specialized services of a customizable and extensible object-oriented operating system called *Choices*. *Choices* includes a framework for the storage of persistent data[9] that is suited to the construction of both conventional file systems and persistent object systems. In a previous paper[5] we discussed *Choices* and how it provides security for persistent objects. In this paper we concentrate on three other areas in which persistent object support differs from file system support: storage organization, storage management, and typing.

Persistent objects incorporate the operating system notion of persistent data storage and the programming language notion of objects[4]. Many conventional operating systems store persistent data in files[2]. Persistent objects in *Choices* differ from regular data files in three fundamental ways:

- they provide a persistent data abstraction, encapsulate data, allow user defined operations, and impose a notion of type. Usually, the operations defined for regular data files are only read and write.

- their operations can store, retrieve, and dereference references to other persistent objects.
- invocation of an operation ensures that the encapsulated persistent data is retrieved. Both the retrieval and storage of data are automated and do not require explicit opens, reads, writes, and closes.

Providing the notion of a persistent object as an operating system service allows it to be used as a powerful concept with which to build other, both conventional and unconventional, services. The three features of a persistent object allow, for example, the convenient programming of a conventional file directory as a persistent object in *Choices*. Operations allow the contents of a directory to be listed in a readable format, to be searched for the location of a particular filename, or to be updated with the addition or removal of a filename.

Choices is not limited to a single application programming language. However, currently we have implemented persistence in *Choices* only for objects that are programmed using C++[16]. In addition, these persistent objects have the following restrictions:

- they must be instances of subclasses of the PersistentObject class, and
- they can store references only to other persistent objects.

Besides presenting persistent object storage organization, storage management, and typing, this paper discusses how persistent objects are named and used within the *Choices* persistent data/file system framework.

2 Persistent Data Storage Framework

The *Choices* file system framework contains a hierarchy of classes that can be combined to build both standard and customized storage systems. The framework is flexible enough to support both persistent storage systems and traditional file systems efficiently[14].

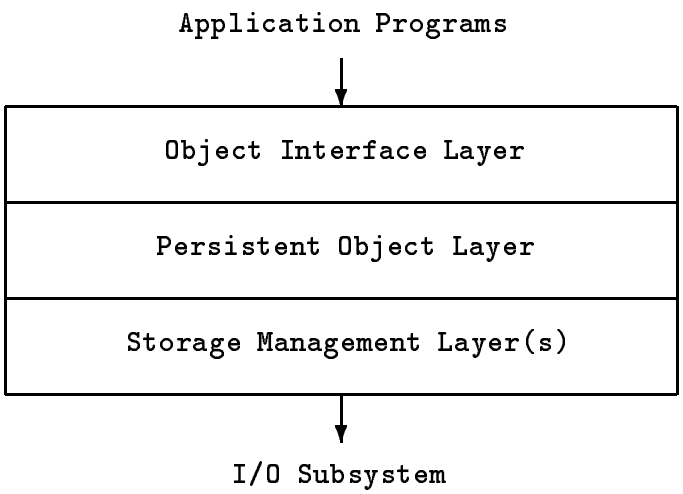

Figure 1: Layers of the File System Framework.

The framework divides a persistent data storage system into a minimum of three layers as shown in Figure 1. Each layer of the framework contains or uses objects that belong to one of two fundamental classes:

- the PersistentStore class defines persistent data stores, each of which store and retrieve persistent data using a random access method, and
- the PersistentObject class defines persistent objects, which encapsulate and provide operations on the data managed by a persistent store.

The PersistentStore provides random access to an uninterpreted sequence of data while the PersistentObject interprets the data of a persistent store as having a format.

Objects in the layers of the file system serve the following purposes:

- Objects in the *Storage Management Layers* access hardware devices like disk controllers or network interfaces in the I/O subsystem. Access is defined in terms of sequences of units, where a unit corresponds to the physical block size used by the hardware device. Objects in the higher *Storage Management Layers* manage the contents of the underlying physical persistent stores, interpreting the data in the stores as nested containers of logical stores. The Storage Management Layers contain PersistentStores and PersistentObjects that encapsulate storage organization and sharing.

- Objects in the *Persistent Object Layer* encapsulate the data within persistent stores and provide operations on the data. These operations include support for naming and data structuring. System programmers can extend the framework by designing persistent objects that support many types of operations and inter-object relationships. All objects in the Persistent Object Layer are PersistentObjects. These objects may be accessed by applications directly; however, application programs can also access them through objects in the Object Interface Layer.

- Objects in the *Object Interface Layer* define additional access protocols for both application and system programs. They use naming objects in the Persistent Object Layer to provide a unified name space for referencing objects in the system. They also provide higher level access protocols for the data encapsulated within the various types of files in the Persistent Object Layer. All objects in the Object Interface Layer refer to PersistentObjects or PersistentStores.

The PersistentStore class defines an access protocol for both physical storage devices and logical storage devices; each storage device is modeled as a sequence of identically sized blocks of persistent data.

The most important data access operations defined by PersistentStores are read and write. These operations retrieve or store one or more contiguous blocks of data, and they are used as primitives by other access methods. For example, clients of the file system may use the abstract protocols of a PersistentObject to access the persistent data of a PersistentStore.

Each PersistentStore has an associated PersistentObject class that provides a data abstraction and encapsulation of the persistent data in the store. At run-time, there is a one-to-one correspondence between an instance of a PersistentStore and its associated PersistentObject. The PersistentStore asA method returns a reference to the store's PersistentObject. If the PersistentObject has not yet been instantiated, the method creates the object by memory mapping the store's persistent data and using the data as part of the encapsulated data of the object. The PersistentStore thus provides the underlying data for its associated PersistentObject. PersistentObjects and their underlying PersistentStores provide the framework and foundation for the *Choices* file system and persistent object store.

In more detail, PersistentObjects and their underlying PersistentStores implement *object-oriented* access to persistent data. The asA method takes an argument which may be either a concrete or an abstract Class (see Section 5) and returns a reference either to an instance of the argument or an instance of a concrete subclass of the argument, respectively. The PersistentStore supports method checks to ensure that the store's associated PersistentObject class (and underlying data representation) is compatible with the class of the object whose reference is returned.

Several file system or persistent object store clients may access the same persistent data. To pro-

vide data consistency for concurrent updates to persistent data through the methods of a persistent object, the PersistentStore ensures that there is, at maximum, only one instance of its associated PersistentObject. Several PersistentObject subclasses use this data consistency provision to implement a main memory cache for frequently accessed persistent data. When a PersistentObject is no longer needed in primary memory, its finalization code calls the close operation on its underlying PersistentStore. A further asA request will create a new PersistentObject that uses the existing persistent data.

The concept of a PersistentStore is used both for physical and logical storage devices allowing reuse of code. All concrete PersistentStores (see Figure 3) belong to one of the following two subclasses:

- Disks encapsulate physical storage devices like hard disk drives, floppy disk drives, and RAM disks. Disks communicate with objects in the I/O subsystem.

- Files encapsulate logical storage devices like UNIX inodes and disk partitions. Files communicate with objects in a lower Storage Management Layer of the file system.

Several Disk subclasses of PersistentStore contain machine-specific code to interface with disks and controller hardware. In "conventional" operating systems, the methods of these classes would be the disk driver routines. The associated PersistentObject classes are used to structure the data on the disks into collections of partitions and collections of files. One example of such a PersistentObject is the PersistentStoreContainer that is used to contain files.

A File is a PersistentStore that is contained within a PersistentStoreContainer. Instead of containing hardware interface code like the disks, it has a source PersistentStore that supplies it with data from a lower layer of the file system. Files provide a *window* into their source PersistentStore.

The size of this window can be either fixed or variable and can range from zero up to the size of the source PersistentStore. The window can be either contiguous or divided into various discontiguous regions of blocks.

Recursively, the data managed by a File can have structure and can be organized into a collection of files. Again, in such cases a subclass of PersistentStoreContainer will be associated with the PersistentStore. In general, a PersistentStoreContainer manages the contents of a PersistentStore as an indexed collection of contained Files. Its methods create, manage, and delete contained Files.

PersistentStoreContainers can be nested to an arbitrary depth. The nesting supports the multiple Storage Management Layers of the framework and also permits user-defined file systems, archives and persistent stores. The PersistentStoreContainer associated with the lowest, physical device, layer divides a disk into a collection of partitions. Each partition is a PersistentStore in the next "logical file system" layer. A PersistentStoreContainer is associated with the PersistentStore in this layer and structures the data of a partition into a collection of files according to the specification of a particular file system. For example, a specific subclass of PersistentStoreContainer structures the data in a partition into a collection of inodes using the inode system format of System V. At higher levels, a file can have an associated PersistentStoreContainer that imposes a *tar* format on the data in a file and provides the abstraction that the file is a collection of tape archived files.

In this section we have described the two main components of the *Choices* file system and persistent object store; the PersistentStore and PersistentObject. Each PersistentStore has an associated PersistentObject which encapsulates the persistent data in the store and provides an abstraction of that data. Access to the PersistentObject conforms to an object-oriented approach using subclasses and inheritance. The PersistentStoreContainer is a particularly important PersistentObject abstraction that structures persistent data into a collection of independent PersistentStores.

3 Storage Organization

Many file systems are optimized to store many small files and a few large files. For example, in UNIX, the implementation uses disk blocks and indirect disk blocks. However, file systems that are organized to use one or more disk blocks for a file can only provide "heavy-weight" support for persistent objects and perform better when the objects are one or more disk blocks in size. Such file systems allow concurrent access to a small maximum number of files; some files may need to be closed to open other files. To program consistently with persistent objects, the size of the objects may range from a byte to the size of a large file. Persistent object systems must support smaller objects efficiently. The number of persistent objects active at any one time may be very large.

Choices supports a wide range of object sizes by using *nested, customizable* PersistentStoreContainers. A container divides a storage device into an indexed collection of PersistentStores, which hold the data for persistent objects. Containers are similar to the *segments* of the Comandos system[11] or the *file objects* of the EXODUS system[6], which both support the clustering of persistent objects for efficiency. Containers differ in that they fit within a general file system framework and they can be nested to an arbitrary depth. Light-weight persistent objects can be clustered in containers that are nested within containers for larger objects. The *Choices* class hierarchy (see Figure 2) defines containers that can be tailored to suit the objects they are intended to contain. A container may contain objects that have temporal locality; that is, if one of the objects is accessed it is likely that they all will be accessed. Alternatively, a container may contain objects of the same class. A large number of persistent objects can be active at any one time.

Table 1 shows how the overhead and minimum storage requirements vary for different persistent data store implementations. The overhead and requirements shown depend only upon the

Container Type	Object Overhead	Minimum Size of Stored Object	Directory Overhead
Class-Specific	0	size of object	4
General	10	1 block	4
MS-DOS	21	1 or 2 blocks	11
ar	44	size of object	16
System V	64	1 or 2 blocks	16
BSD 4.3	128	1 or 2 blocks	key length + 8
tar	412	1 block	100
AIX	512	0 blocks	key length + 8

Table 1: File System Overhead

requirements of the particular storage technique. In order to be disk format compatible with each storage technique, *Choices* imposes no additional storage requirements.

The least overhead is obtained for a container of persistent objects that are all of the same class. *Choices* memory maps the persistent data in each persistent object. Access is provided by the methods defined for the class. The collection stores the class of the objects stored in the container and hence there is zero extra storage overhead imposed per object. The size of the persistent store is exactly the size of the persistent object. The directory overhead to hold the logical name of the object is four bytes.

A container that holds objects of different classes, a general container, must store the size, location and class of each object. This imposes an overhead of ten extra bytes per object. Since each object may be of a different size, it is convenient to store the object in at least a block of a persistent store. The directory overhead is not increased.

The MS-DOS file system imposes the least overhead per file and directory of the various "stream-oriented" file systems we have implemented. The overhead records the protection and size of the file and the starting location of the file on disk. A non-empty file requires at least one cluster (one or two disk blocks) of persistent store where a block is the sector size of the disk. The eleven

bytes of directory information contain a two-part symbolic name. System V requires an overhead of 64 bytes per persistent store in which it stores "inode" information including protection and size information and pointers to the direct, single-indirect, double-indirect, and triple-indirect blocks that represent the file. It also uses at least one block to store a non-empty file. The directory information includes a fourteen character symbolic name and a two byte reference to the "inode" information. The AIX "inode" structure includes room for 384 bytes of data, therefore a minimal, non-empty file could use zero additional blocks of storage.

In this section, we have reviewed the requirements for a persistent store to support persistent objects and shown that most conventional file systems are inappropriate and impose too much overhead and too many limitations. However, existing file system implementation techniques can be used to provide storage for persistent objects. In *Choices*, we have exploited file system implementations by building an object-oriented framework for persistent storage that can be subclassed both to support efficient conventional file systems and efficient persistent objects.

4 Storage Management

The storage management issues for files systems and persistent object stores differ significantly because persistent objects can store references to other persistent objects. In practice, our *Choices* implementation required different garbage collection algorithms for persistent object containers and file containers. Nevertheless, the implementation allows all the different kinds of containers to be incorporated within the same storage organization framework. Each container is customized to support specific file system or persistent object storage management behavior.

Many file systems restrict inter-object references to directories, and they further restrict di-

rectories so as to disallow cyclic references. This avoidance of cyclic persistent data structures allows traditional file containers to use reference-counting or even simpler storage management techniques. An example of the use of reference-counting for storage management is a container of UNIX inodes[17]. UNIXContainers allow a regular file to have many directory entries but restrict a directory to have only one directory entry. The inode for each file and directory has a *link count*, which holds the number of directories that reference it. Application programs can use the `unlink` system call to remove a file from a directory. This call decrements the inode's link count. When the link count reaches zero, the inode is deleted.

An example of the use of a simpler storage management technique is a container for MSDOS files[12]. MSDOSContainers allow a file or a directory to have only one directory entry. Therefore no reference count is needed. Application programs can use the `unlink` system call to remove a file from a directory. In the case of an MSDOSContainer, this call simply deletes the file.

Unlike the restricted cases of UNIX and MSDOS containers, general persistent object containers must allow arbitrary inter-object references. Therefore, more sophisticated garbage collection techniques are needed for containers of persistent objects. Persistent object containers can be considered as heap managers for the persistent objects they contain. Since we chose to use the C++[16] model for objects we were not limited by the language to a specific kind of garbage collection. Instead, we can select an appropriate garbage collection scheme depending on the use of a particular container. Currently we have experimented with two schemes: the first one uses reference counting and the second one does not.

Both garbage collection schemes are based on the concept of *reachable* objects. A container of PersistentObjects keeps track of which objects belong to a set of *root* objects. An object is defined to be reachable if it is a root object or if there is a chain of references to it from a root object.

All objects that are not reachable should have their storage reclaimed. The PersistentObject class defines two operations that enable applications to control which objects belong to the root set:

- persist, which adds the object to the set of root objects if it was not already in the set, and
- desist, which removes the object from the set of root objects if it was in the set.

Our first scheme, which was described in [5], supported applications that have references and persistent objects which can be modeled by a directed acyclic graph (DAG). This restriction is reasonable for many applications that avoid circular lists like our first applications. However, in general the objects and references of many applications cannot be modeled by a DAG and the first scheme is too restrictive.

The second scheme uses a separate process that periodically locates and removes unreachable objects. This process resembles the UNIX **fsck** program or the MS-DOS **chkdsk** program. It differs from these programs in three ways:

- unreachable objects are considered garbage as opposed to being considered "lost" files that have been "found,"
- the process must be run periodically instead of being used to recover from error conditions, and
- the process must have access to the layout of all classes of objects, not just the layout of directories.

In brief, our studies of persistent object stores and references indicate that garbage collection algorithms for persistent object stores are different from those for file systems. In *Choices*, every persistent object is accessed from a container and this container can be customized to support

specific storage management behavior. We have implemented three different garbage collection techniques, one for files, one for persistent objects that conform to a DAG reference model, and one that scans storage for garbage. The current *Choices* design for persistent objects accommodates the different storage management techniques within the same framework of classes. However, it remains to be seen whether improved storage management techniques can be embedded within the same framework. Furthermore, we have not implemented any scheme that would support garbage collection when inter-object references are allowed to span container boundaries, but we feel an approach like that proposed in [15] would be feasible.

5 Typing

Most conventional file systems have a small, fixed set of file types. However, persistent object stores must support extensible sets of persistent object types. The PersistentStore implementation allows extensible sets of persistent object types by associating persistent data with a Class of PersistentObject. *Choices* implements run-time type-checking to ensure that persistent data is accessed using the appropriate persistent object abstraction. The checking is performed by the supports method of the PersistentStore. The type checking is based on the C++ style of associating type with the notion of class. Inclusion polymorphism or class inheritance is supported by the type checking mechanism so that a client of the persistent object store may access an instance of a new subclass of persistent object using the protocols defined for an abstract, older superclass. Thus, the typing mechanism supports extensions and specializations of existing classes of persistent objects.

Invocation of a persistent object method automatically and dynamically loads the code of the methods for the class of the object. Again, older applications may execute methods of new classes

of persistent object using only the preexisting abstract superclass definitions. Thus by using an object-oriented approach, the type system supports the late binding and dynamic loading of code for persistent objects. These mechanisms are again captured as specializations of classes within a class hierarchy that is organized as a framework.

Classes from the *Choices* PersistentObject (see Figure 2) and PersistentStore (see Figure 3) class hierarchies can be combined to build several standard file systems such as System V or MS-DOS. Such file system standards account for many of the subclasses of PersistentStore. For efficiency, these PersistentStores are specialized to support the fixed set of associated PersistentObject classes required to implement the standard. For example, the BSDInode class can store the data for: regular files (PersistentArrays), two types of directories (BSDDirectories and HashedBSDDirectories), and symbolic links (SymbolicLinks). Because the set of associated persistent object classes is small and fixed, an efficient representation of the classes can be hard-coded into the BSDInode class. However, the class hierarchies also support system and user-defined persistent objects. In this case it is not possible to hard-code the set of PersistentObject classes associated with each PersistentStore. Therefore, the PersistentStore hierarchy includes the GeneralNode class and the PersistentObject hierarchy includes the GeneralContainer class.

The implementation language of *Choices*, C++, does not support the dynamic loading of new classes at run-time. However, as the provider of a persistent object store, *Choices* must support new persistent objects and new persistent data stores. Together the GeneralContainer and GeneralNode classes support the storage, retrieval, and use of new, user-defined persistent objects and data. The classes support the dynamic loading of other classes, avoiding the necessity to recompile the persistent object store.

The dynamic loading of new subclasses of PersistentObject and PersistentStore is accomplished

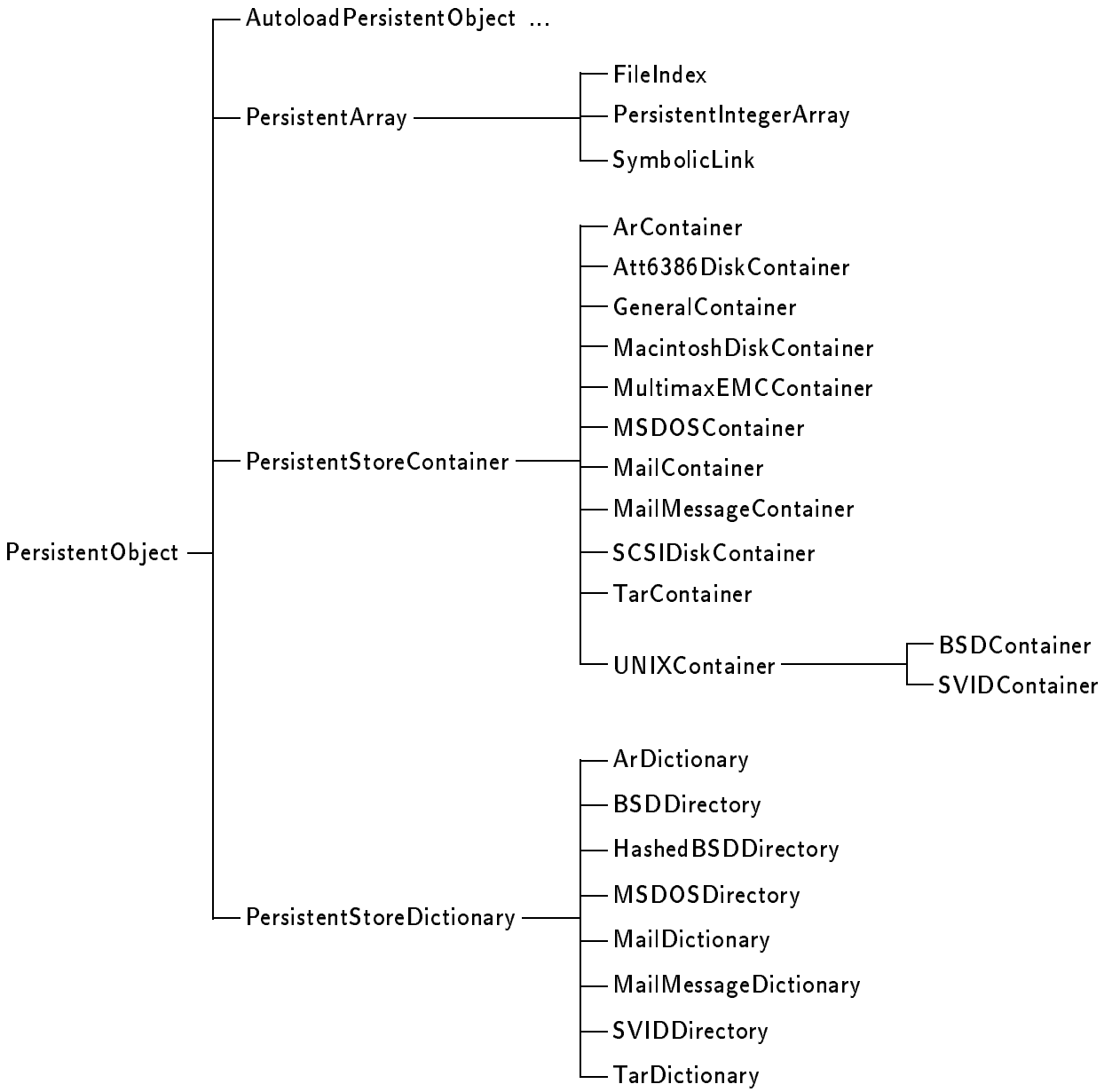

Figure 2: PersistentObject Class Hierarchy

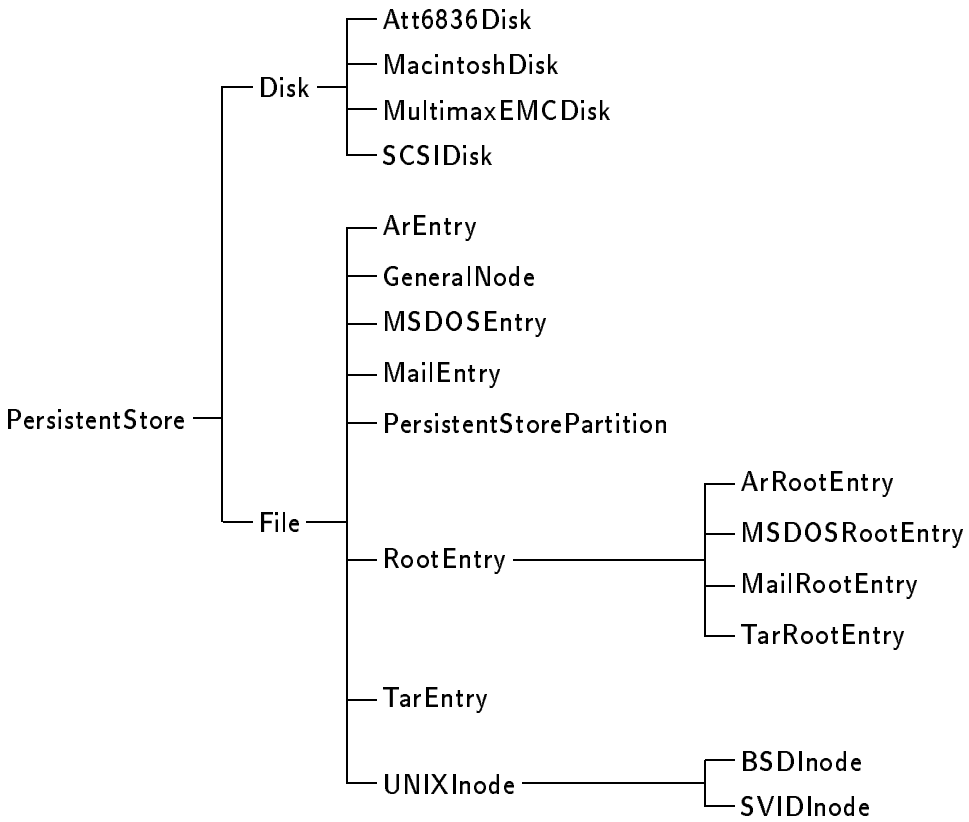

Figure 3: PersistentStore Class Hierarchy

using an object-oriented solution. In *Choices* we explicitly represent each class defined in the system as an object of class Class. Using object-oriented terminology, an instance of class Class is used to *reify* C++ class definitions. The GeneralContainer stores the name of the class of the PersistentObject that is associated with a GeneralNode as a character string. The GeneralNode refers to its associated PersistentObject class as an instance of class Class. The character string is converted to a reference to a Class by the lookup method of a kernel NameServer. The GeneralNode may invoke methods on the PersistentObject class using the protocol defined for class Class. This allows the GeneralContainer and GeneralNode classes to be compiled without using a specific class name for the associated PersistentObject, thus avoiding recompilation and relinking. For more information on first-class classes in C++, see [10, 8].

To support a dynamically extensible persistent object store, *Choices* must support the dynamic *loading* of code for new persistent objects. The new persistent object can be dynamically linked using the symbol table from previous loads and linkages. However, it is more difficult to modify the running system to invoke the constructor of the new object. Unfortunately, C++ constructors are *statically bound* at compile time and do not generate any useful symbolic linker and loader information. It is thus difficult to call the constructor from a running program like the persistent object store. Our solution to the dynamic invocation of the static C++ constructor involves indirection, using the invocation of an instance constructor method that is defined for objects of class Class. The GeneralNode may invoke the instance constructor method on its associated PersistentObject Class. The following steps are performed by this constructor method:

1. If the _constructor variable of the instance of Class is zero, the method creates a CodeLoader object and initializes the CodeLoader by passing it the name of the Class as an argument.

2. If the CodeLoader successfully loads the new class, the _constructor variable is assigned the address of a function that will invoke the static C++ constructor.

3. However, if the _constructor variable is still zero, the method fails by returning zero.

4. If the _constructor variable is non-zero, the method invokes the actual constructor. The return value of the constructor is also returned by the instance constructor method of the Class.

For more information on the operations performed by the CodeLoader and on adding code to running C++ systems, see [10, 7].

The organization of the containers of persistent objects is required to be space-efficient, see Section 3. Since there is no inherent limit on the length of Class names, storing one in the fixed-size structure of each GeneralNode is unacceptable. Instead, each GeneralContainer stores the names of the Classes of PersistentObjects that are associated with its GeneralNodes. This has several advantages:

- the Class names can be efficiently stored in a single buffer as null-terminated strings (so there is no need to impose an arbitrary limit on the length of Class names),

- if several GeneralNodes support the same Class, then the name only needs to be stored once, and

- each GeneralNode needs to store only an index (which requires only 1–4 bytes) into its container's list of Class names.

This section has discussed the typing of a persistent object store and the problems of dynamic code loading. In *Choices*, typing is accomplished by storing the name of the class of a persistent object in the container in which the object is stored. Invocation of a constructor for a new class

of persistent object is achieved using indirection. Objects of class Class represent the class of a persistent object and have an instance constructor method that may be used initialize the persistent object. Object-oriented techniques are used in *Choices* to simplify the programming of these mechanisms.

6 Naming

PersistentStores and PersistentObjects have both *logical* and *symbolic* names. The logical names of PersistentStores and PersistentObjects are unique identifiers, which are derived from the organization of nested PersistentStoreContainers:

- The method of identifying a PersistentStore depends on whether it is a Disk or a File.
  - A Disk is identified by a unique index within a computer system.
  - A File is identified by a pair comprising the identifier of its container and the File's index within its container.
- A PersistentObject is identified by the identifier of its underlying PersistentStore.

Within the Storage Management Layers of the framework, the file system identifies PersistentStores by their logical names, whereas higher layers and application programs identify PersistentStores by their symbolic names.

While logical names are based on the organization of nested PersistentStoreContainers, symbolic naming is orthogonal to organization. Therefore, although there is a single unique mapping of logical names to PersistentStores in a computer system, multiple symbolic name spaces can be mapped onto the set of logical names. Within each name space, multiple symbolic names can map to a single PersistentStore.

PersistentStores can be grouped into and given symbolic names by objects in both the Persistent Object and Object Interface Layers. In the Persistent Object Layer, PersistentStoreDictionaries, which are collections of <symbolic-key, logical-name> pairs, map symbolic names to the logical names of PersistentStores. Within any dictionary, the keys must be unique, but several keys may map to the same logical names. Using symbolic names, PersistentStores can be opened from, created in, added to, and removed from PersistentStoreDictionaries.

In the Object Interface Layer, FileSystemInterfaces use a set of PersistentStoreDictionaries to parse sequences of symbolic keys, called *pathnames*, and resolve them to logical names of PersistentObjects. FileSystemInterfaces can use various policies and mechanisms to unify the symbolic name spaces provided by PersistentStoreDictionaries. We have implemented a FileSystemInterface that provides the UNIX concept of root and current directories, mount tables, and symbolic links.

Since PersistentObjects can refer to other PersistentObjects directly, there is no requirement that they have symbolic names. It is advantageous for PersistentObjects that do not need to be identified by the users of applications to be given no symbolic names. In contrast, applications should be able to assign symbolic names to PersistentObjects both during and after their creation.

To assign a name to a PersistentObject as it is being created, applications can use the FileSystem-Interface::create or PersistentStoreDictionary::create operations. Both operations take two arguments, the name of the object to be created and the Class to which it should belong. They both check to see if the object name already exists, and if it does not, they invoke the PersistentStoreContainer::create operation and store the object name in a PersistentStoreDictionary. The major difference between the two operations is that FileSystemInterfaces use path names while PersistentStoreDictionaries use single-element file names.

To assign a name to a PersistentObject after it has been created, applications can use the

PersistentObject::bindTo operation. This operation takes a FileSystemInterface and an object name as arguments, and it invokes an operation on the FileSystemInterface that causes an entry to be added to a PersistentStoreDictionary that contains the object name and refers to the PersistentObject.

7 Programming with PersistentObjects

The utility of persistent objects is directly related to how closely their usage resembles the usage of standard programming language objects. Many persistent object systems extend the base language to allow the almost seemless addition of persistence[1, 4, 3, 13]. We chose instead to use the inherent extensibility of C++ and to use a set of support classes and programmer conventions. Since we also use these classes and conventions to manage standard heap objects within the *Choices* kernel, programming with PersistentObjects is nearly identical to programming other objects in *Choices*.

The two classes that make programming with PersistentObjects virtually transparent to application programmers are: PersistentObjectStar and AutoloadPersistentObject. PersistentObjectStar and its subclasses are part of a "smart pointer" hierarchy that mirrors the major class hierarchy of *Choices*, which is rooted in class Object. The entire hierarchy of ObjectStars provide a type-safe mechanism for managing the garbage collection of heap objects. We originally developed ObjectStars (called "References" in [5]) for our first persistent object implementation. We generalized them to support all heap-allocated objects[10], because we found using them preferable to the explicit programming of reference-counting methods. ObjectStars provide a pointer-like syntax using the operator $->$ overloading feature of C++.

The AutoloadPersistentObject class lets programmers develop persistent object classes without explicitly programming the storage and retrieval of persistent data. When the PersistentStore::asA

operation is invoked, the PersistentStore checks to see if the requested Class of persistent object is a subclass of AutoloadPersistentObject. If it is, then the store loads the data for the PersistentObject after calling its constructor. When an AutoloadPersistentObject is no longer needed in primary memory, its noRemainingReferences operation is automatically invoked and the object's persistent data is written back to its PersistentStore before the object is deactivated.

Because our framework supports the data format of various existing file system standards, and because these formats are not amenable to being directly memory-mapped, the default behavior for PersistentObjects is that it requires programmers to manually implement the memory mapping. If one is designing data structures using C++ classes, then automatic memory mapping is feasible, and this behavior can be inherited from AutoloadPersistentObject. In either case, the choice of whether a subclass of PersistentObject implements or inherits the memory mapping is transparent to users of instances of the class.

8 Summary

In this paper we show that, although conventional operating systems do not provide support for efficient persistent object system implementations, one can build both conventional file systems and persistent object systems within the same persistent storage framework. We also demonstrated how the differing requirements of file and object storage can be accommodated as specializations of the abstract classes within the framework.

Conventional storage organization, storage management, and typing are too inefficient, restrictive, and inflexible to be used as the basis for a general persistent storage system. We have built a general persistent storage framework that uses nested containers of persistent data stores. The

stores may be customized for the types and sizes of persistent objects that they contain. The framework permits the selection of various garbage collection schemes to match the allowed sets of inter-object references. The types of the persistent objects that can be stored in the system may be dynamically extended.

We also discuss the naming and use of persistent objects within the *Choices* operating system. Persistent objects are not required to have symbolic names, but applications can give objects symbolic names as needed, both during and after creation. A set of support classes and programmer conventions enables nearly transparent usage of persistent objects.

References

[1] R. Agrawal and N. H. Gehani. Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++. In *Second International Workshop on Database Programming Languages*, Oregon Coast, June 1989.

[2] Lubomir Bic and Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[3] Robert Bretl, David Maier, Allen Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams. The GemStone Data Management System. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 283–308. Addison-Wesley, Reading, Massachusetts, 1989.

[4] Alfred Leonard Brown. Persistent Object Stores. Technical Report Persistent Programming Report 71, Universities of St Andrews and Glasgow, October 1989.

[5] Roy H. Campbell and Peter W. Madany. Considerations of Persistence and Security in Choices, an Object-Oriented Operating System. In *International Workshop on Computer Architectures to Support Security and Persistence of Information*, pages 12.1–12.14, University of Bremen, Federal Republic of Germany, May 1990.

[6] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Storage Management for Objects in EXODUS. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 341–370. Addison Wesley, Reading, Massachusetts, 1989.

[7] Sean M. Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding New Code to a Running C++ Program. In *Proceedings of the USENIX C++ Conference*, pages 279–292, San Francisco, California, April 1990.

[8] John A. Interrante and Mark A. Linton. Run-time Access to Type Information in C++. In *Proceedings of the USENIX C++ Conference*, pages 233–240, San Francisco, California, April 1990.

[9] 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.

[10] Peter W. Madany, Roy H. Campbell, and Panogiotis Kougiouris. Experiences Building an Object-Oriented System in C++. In *Technology of Object-Oriented Languages and Systems Conference*, Paris, France, March 1991.

[11] José Alves Marques and Paulo Guedes. Extending the Operating System to Support an Object-Oriented Environment. In *Proceedings of OOPSLA '89*, pages 113–122, New Orleans,

Louisiana, September 1989.

[12] Peter Norton. *The Peter Norton Programmer's Guide to the IBM PC.* Microsoft Press, 1985.

[13] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The Design of the E Programming Language. Technical Report Computer Sciences 824, University of Wisconsin, Madison, February 1989.

[14] 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.

[15] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Technical Report 1320, INRIA, November 1990.

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

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