AN OBJECT-ORIENTED FRAMEWORK FOR FILE SYSTEMS

BY

PETER WILLIAM MADANY

B.A., Trinity Christian College, 1982
M.S., Illinois Institute of Technology, 1984

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

AN OBJECT-ORIENTED FRAMEWORK FOR FILE SYSTEMS

Peter William Madany, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1992
Roy H. Campbell, Advisor

File systems are essential components of general purpose operating systems and are highly visible to computer users. To satisfy the growing diversity of persistent storage needs presented by application and system programs, operating systems should support both standard and customizable file systems. To facilitate the design, construction, and use of these file systems, they should fit within a general and extensible model. Currently file systems are based on one of a few common models that are neither general nor extensible. This thesis shows that a simple, general, and extensible model can be constructed by presenting an object-oriented framework that contains a few powerful persistent storage abstractions. The framework supports the characterization, design, and construction of both common file systems and experimental file systems. The object-oriented approach was chosen to exploit the techniques of data encapsulation, data abstraction, inheritance, and polymorphism.

The framework is called the *Choices* file system framework. It contains two fundamental abstractions: the PersistentStore class defines objects that manage the storage and retrieval of raw persistent data, and the PersistentObject class defines objects that encapsulate and provide operations on the data managed by persistent stores. Persistent stores manage data access within the file system, while various kinds of persistent objects encapsulate the organization, naming, and structure of persistent data. The framework also includes file system access classes that provide convenient interfaces to persistent objects. Using subclasses of PersistentStore and PersistentObject and several application interface classes, I and other *Choices* project team members have built various file systems. These file systems include some stream-oriented file systems, a record-oriented file system, a distributed file system, a persistent object store, and several special-purpose file systems.

After introducing the problems of current file system models, defining terminology, and surveying related work, this thesis presents the *Choices* file system framework, discusses how the framework can be extended, and describes some of the file systems built using the framework.

To my son Jeremy

Acknowledgements

I thank my advisor, Professor Roy Campbell, for his suggestions, for analysis of my work on *Choices*, for his criticisms and clarifications of my descriptions of my work, and for obtaining funding for the *Choices* project. I thank Professor Ralph Johnson for teaching me much about object-oriented languages and object-oriented programming. Besides Roy and Ralph, I thank the other members of my committee: Daniel Reed, Dennis Mickunas, and Tony P. Ng.

I thank Vince Russo for suggesting that I develop a file system for *Choices*, for invaluable help in the design and requirements of the file system, and for doing much of the early work on *Choices*.

I also acknowledge the help I received from the students with whom I worked on the *Choices* project: Jeff Biesiadecki, John Coolidge, Dave Dykstra, Gadi Freedman, Bjorn Helgaas, Nayeem Islam, Gary Johnston, Panos Kougiouris, Doug Leyens, Lee Lup Yuen, Jeff Mantei, Gary Murakami, Amit Parghi, Anna Salzberg, Aamod Sane, Rajendra Singh, See-Mong Tan, Dan Weber, Lun Xiao, and Johnny Zweig.

Several people, including Mike North, Steve March, and Dave Raila, kept the wide variety of computers that I used up and running. I am grateful that they performed so many tasks that research assistants are usually expected to do.

I thank many of the non-academic staff members who work for the Department of Computer Science, especially Anda Harney.

I thank the National Science Foundation, NASA, and AT&T for generously supporting the projects I worked on while at the University of Illinois.

Finally, I owe my deepest gratitude to my wife, my parents, and my wife's grandmother.

Table of Contents

Chapter

1 Introduction 1

2 Background 4
- 2.1 File System Terms and Concepts 4
  - 2.1.1 Access Protocols 5
  - 2.1.2 Naming 6
  - 2.1.3 Operations 6
  - 2.1.4 Protection 7
  - 2.1.5 Attributes 8
  - 2.1.6 Other Terms 8
  - 2.1.7 Summary 9
- 2.2 Object-Oriented Programming Terms and Concepts 9
  - 2.2.1 Data Encapsulation 10
  - 2.2.2 Data Abstraction 10
  - 2.2.3 Interface and Code Sharing 10
  - 2.2.4 Polymorphism 11
  - 2.2.5 Design Sharing 11
  - 2.2.6 Summary 12

3 Related Work 13
- 3.1 Survey of File Systems 13
  - 3.1.1 Record File Systems 13
  - 3.1.2 Byte-Stream File Systems 14
  - 3.1.3 Object Storage Systems 15
  - 3.1.4 Distributed File Systems 18
  - 3.1.5 Summary 19
- 3.2 Approaches to File System Design 19
  - 3.2.1 File System Standards 20
  - 3.2.2 Extensible File Systems 20
  - 3.2.3 Software Frameworks 21
  - 3.2.4 Summary 25

4 File System Abstractions 27
- 4.1 UNIX File Systems 28
  - 4.1.1 The Application Interface Layer 28

4.1.2 The Persistent Object Layer . . . 32
4.1.3 The Storage Management Layer . . . 35
4.2 The MS-DOS File System . . . 37
4.2.1 The Application Interface Layer . . . 37
4.2.2 The Persistent Object Layer . . . 38
4.2.3 The Storage Management Layer . . . 39
4.3 File Types in the VMS File System . . . 40
4.3.1 Sequential Files . . . 41
4.3.2 Relative Files . . . 41
4.3.3 Indexed Files . . . 42
4.4 Summary . . . 43

5 File System Framework . . . 44
5.1 Examples of the Framework . . . 44
5.1.1 A Concrete Example . . . 46
5.1.2 An Abstract Example . . . 48
5.2 Components of the Framework . . . 51
5.2.1 Storage Devices . . . 52
5.2.2 Persistent Objects . . . 58
5.2.3 Storage Device Organization and Sharing . . . 61
5.2.4 Naming . . . 64
5.2.5 File Data Structures . . . 67
5.2.6 Application Interface Objects . . . 75
5.2.7 Protection . . . 81
5.3 Extensions to the Framework . . . 82
5.3.1 File Data Structures . . . 83
5.3.2 Application Interface Objects . . . 88
5.4 Constraints on the Framework . . . 89
5.5 Summary . . . 91

6 Building Concrete Subclasses . . . 93
6.1 Subclassing PersistentStores . . . 94
6.2 Subclassing PersistentObjects . . . 98
6.3 Subclassing PersistentStoreContainers . . . 99
6.4 Subclassing BlockAllocators . . . 103
6.5 Subclassing PersistentStoreDictionaries . . . 104
6.6 Subclassing Other Kinds of PersistentObjects . . . 107
6.7 Summary . . . 108

7 Performance . . . 110
7.1 File Access Operations . . . 111
7.2 File I/O Operations . . . 111
7.3 Summary . . . 113

8 Conclusions 114
8.1 Evolution of the Framework 114
8.1.1 UNIX-Like File Systems 114
8.1.2 Stream-Oriented File Systems 115
8.1.3 An Object-Oriented File System Framework 116
8.2 Software Engineering Benefits 117
8.2.1 Reusability 117
8.2.2 Portability 117
8.2.3 Maintainability 117
8.2.4 Customizability 118
8.2.5 Extensibility 118
8.3 Summary 119
8.4 Future Work 120

Appendix

A Source Code 121
A.1 Selected PersistentStore Operations 122
A.2 Selected Disk Operations 124
A.3 Selected File Operations 125
A.4 Selected PersistentStoreContainer Operations 128
A.5 Selected BlockAllocator Operations 131
A.6 Selected PersistentStoreDictionary Operations 132
A.7 Selected FileSystemInterface Operations 136

B Storage Management Examples 138
B.1 Standard Operating System Formats 138
B.1.1 UNIX Storage Management Systems 138
B.1.2 The MS-DOS Storage Management System 144
B.2 Standard Tool Formats 146
B.2.1 The Ar File System 146
B.2.2 The Tar File System 147
B.2.3 The Mail File System 149
B.2.4 The COFF File System 151
B.3 A Remote File System 153
B.4 A General File System 154
B.5 Summary 156

C Using a Persistent Object Store 158
C.1 The POLExpression Class 159
C.2 Subclasses of POLExpression 160
C.3 A Sample Program 162
C.4 Summary 163
C.5 Class Declarations 165

Bibliography 170

Vita 178

List of Tables

3.1 Approaches to File System Design 26

7.1 File Access Operation Measurements 111
7.2 Data Access Operation Measurements 112
7.3 File Copy Measurements 113

List of Figures

3.1 A Hierarchical File System 24

5.1 Layers of the File System Framework. 45
5.2 Concrete Example of File System Framework 47
5.3 Abstract File System Framework 49
5.4 File System Class Hierarchy 51
5.5 Class Declaration: PersistentStore 53
5.6 Class Declaration: Disk 56
5.7 Class Declaration: File 57
5.8 Class Declaration: PersistentObject 60
5.9 Persistent Object Class Hierarchy 60
5.10 Class Declaration: PersistentStoreContainer 62
5.11 Class Declaration: BlockAllocator 64
5.12 Class Declaration: PersistentStoreDictionary 66
5.13 Class Declaration: SymbolicLink 67
5.14 Persistent Array Class Hierarchy 69
5.15 Class Declaration: PersistentArray 70
5.16 Class Declaration: PersistentCharArray 70
5.17 Class Declaration: PersistentIntArray 71
5.18 Class Declaration: RecordFile 72
5.19 Class Declaration: FileIndex 73
5.20 Class Declaration: AutoloadPersistentObject 75
5.21 Class Declaration: MountAssociation 76
5.22 Class Declaration: MountTable 77
5.23 Class Declaration: FileSystemInterface 78
5.24 Class Declaration: RecordStream 80
5.25 Class Declaration: PersistentArrayStream 81
5.26 Record Structuring Class Hierarchy 83
5.27 Class Declaration: FixedRecordFile 84
5.28 Class Declaration: VariableRecordFile 85
5.29 Class Declaration: RelativeRecordFile 85
5.30 Class Declaration: IndexedRecordFile 87
5.31 Record Stream Class Hierarchy 88
5.32 Class Declaration: FixedRecordStream 88
5.33 Class Declaration: VariableRecordStream 89
5.34 Class Declaration: IndexedRecordStream 90

6.1 Persistent Store Class Hierarchy 94
6.2 Function Definition: ExampleFile::read 95
6.3 Function Definition: ExampleFile::write 96
6.4 Function Definition: ExampleConstructor 99
6.5 Persistent Store Container Class Hierarchy 100
6.6 Block Allocator Class Hierarchy 103
6.7 Persistent Store Dictionary Class Hierarchy 104

A.1 Function Definition: PersistentStore::asA 122
A.2 Function Definition: PersistentStore::copy 123
A.3 Function Definition: Disk::read 124
A.4 Function Definition: Disk::write 124
A.5 Function Definition: File::read 125
A.6 Function Definition: File::write 125
A.7 Function Definition: File::basicRead 126
A.8 Function Definition: File::basicWrite 127
A.9 Function Definition: PersistentStoreContainer::open 128
A.10 Function Definition: PersistentStoreContainer::create 129
A.11 Function Definition: PersistentStoreContainer::rootDictionary 129
A.12 Function Definition: PersistentStoreContainer::close 130
A.13 Function Definition: PersistentStoreContainer::synchronize 130
A.14 Function Definition: BlockAllocator::allocate 131
A.15 Function Definition: BlockAllocator::free 131
A.16 Function Definition: PersistentStoreDictionary::associations 132
A.17 Function Definition: PersistentStoreDictionary::keys 132
A.18 Function Definition: PersistentStoreDictionary::create 133
A.19 Function Definition: PersistentStoreDictionary::add 134
A.20 Function Definition: PersistentStoreDictionary::open 134
A.21 Function Definition: PersistentStoreDictionary::remove 135
A.22 Function Definition: FileSystemInterface::pathOpen 136
A.23 Function Definition: FileSystemInterface::findDictionary 136
A.24 Function Definition: FileSystemInteface::basicFindDictionary 137

B.1 Using Ar File System Classes within the Framework 148
B.2 Using Mail File System Classes within the Framework 150
B.3 Using COFF File System Classes within the Framework 152

C.1 Persistent Object Language Class Hierarchy 159
C.2 Example of User-defined Persistent Objects. 162
C.3 Persistent objects in a parse tree for `factorial(7)` 164
C.4 Class Declaration: POLExpression 165
C.5 Class Declaration: POLInteger 166
C.6 Class Declaration: POLString 166
C.7 Class Declaration: POLBuiltin 166
C.8 Class Declaration: POLExpressionList 167
C.9 Class Declaration: POLIfExpression 167
C.10 Class Declaration: POLLambda 168

C.11 Class Declaration: POLApply 168
C.12 Class Declaration: POLVariable 169

Chapter 1

Introduction

File systems are highly visible components of operating systems that store, retrieve, organize, name, structure, and protect a computer system's persistent data[BS88, PS85]. They determine a user's perspective of persistent data, allow effective utilization of storage devices, protect data from unauthorized access or destruction, and profoundly affect the performance of programs. As advances are made in the areas of computer architecture and programming languages, new demands are placed on file systems. These demands often lead to changes in the models that determine the construction and usage of file systems. Currently various models of file systems exist, including stream-oriented[Tho78, MJLF84, Nor85], record-oriented[Dem88, Sha91], and object-store[ABCM83, AG89a, MG89, SGM89]. However, rather than being general models of file systems, these models are closely tied to specific programming languages and environments; rather than being based on abstract frameworks, these models are usually defined by *concrete implementations*. The lack of generality in these models gives a narrow perspective of file systems and leads to incompatibilities between systems. Basing models on implementations makes them less flexible and less extensible. Together these deficiencies indicate the need for a model of file systems that has the following characteristics:

- simplicity — the model is composed of a small set of orthogonal abstractions,
- generality — existing file systems can be mapped to the model, and
- extensibility — new components can be easily added to the model.

In this thesis I show that one can construct a simple, general, extensible model of file systems by presenting an object-oriented framework that contains a few powerful persistent storage abstractions. I further show that system programmers can design and build a wide variety of file systems within this framework.

Thus the three major contributions of this thesis are:

- **A unified model of file systems.** The following types of file systems are all modeled using the same set of components: stream-oriented, record-oriented, and object-oriented. The model also supports customized and distributed file systems. This has several advantages that include supporting the classification of various types of file systems and providing more consistent application interfaces.

- **A practical framework for file system construction.** Not only can the framework presented in this thesis classify existing systems, but it can also be used to design and build efficient implementations of those systems. (All file systems described in Appendix B are built from the same classes of components.) Furthermore, it provides an excellent test-bed for experimenting with new combinations of file system algorithms and data structures.

- **A file system incorporating many popular interfaces and data structures.** Compatibility with existing standards greatly eases the initial use of new computer systems. The *Choices* file system framework provides two important kinds of compatibility. First, its application interface allows programmers to use familiar file system operations. Second, its internal data structures support the direct use of many common types of files and disk formats.

Thus, this thesis contributes to the fields of both *software engineering* and *operating systems.*

I chose an object-oriented approach to modeling file systems because of the benefits of the following object-oriented techniques: data encapsulation, data abstraction, inheritance, and polymorphism[Rus91]. *Data encapsulation* isolates an object's behavior from its representation. *Data abstraction* classifies and describes the behavior shared by a set of objects[GR83, Str86]. *Inheritance* allows classes to share structural and behavioral descriptions and supports hierarchies of classes[Weg87]. *Polymorphism* allows a particular operation to be applied to objects of various types[Dig86, Nie89].

To demonstrate the framework presented in this thesis, I have developed a portable, extensible file system prototype that includes a hierarchy of abstract and concrete file system classes. Besides demonstrating the characteristics of the framework, the prototype serves as the persistent storage system for the *Choices* family of operating systems, and it has been used as the basis of operating system class assignments and projects.

The remainder of this thesis is organized as follows. Chapter 2 provides background information and terminology. Chapter 3 surveys related file system models and approaches to file system designs. Chapter 4 presents abstractions found in some common file systems. Chapter 5 introduces the framework, its abstractions, and its constraints. Chapter 6 discusses extending the framework by building concrete subclasses. Chapter 7 gives some performance measurements of systems built using the framework. Chapter 8 draws conclusions about the work presented here and suggests how the framework can be used for further file system research. Appendix A provides pseudo-code for many of the operations discussed in Chapters 5 and 6. Appendix B describes the concrete subclasses of systems built using the framework. And Appendix C presents an example application of a persistent store built using the framework.

Chapter 2

Background

This chapter provides necessary background information for the rest of the thesis by defining terms and discussing concepts of file systems and object-oriented programming.

2.1 File System Terms and Concepts

Data is *persistent* if its lifetime exceeds a single execution of a program. A *file* is a collection of persistent data, called *contents*, and a set of *attributes*. A *file system* is the program that controls the access to and manipulation of sets of files. The term "file system" can also be used to refer to the storage occupied by a set of files[Tho78]. To avoid this ambiguous usage, I will use the term *file container* for a collection of files and their storage. Throughout this thesis, any physical storage device that provides random access to persistent data will be called a *disk*.

The primary purpose of a file system is to store and manage data effectively[BS88] and reliably[Tan86]. Various data storage media present widely different hardware interfaces; file systems present abstract interfaces that hide these differences. To support time-sharing and multi-tasking computers, file systems permit storage media sharing by organizing devices into flexible, dynamic sets of logical storage devices: files. Thus, from the perspective of application programs, files can be viewed as collections of persistent data, and, from the perspective of low-level system code, they can be viewed as logical storage devices. In an environment with file sharing, naming of files becomes crucial; thus file systems must provide mechanisms for attaching useful names to files. Likewise, file sharing by multiple users requires mechanisms that protect data from accidental or malicious corruption and from unauthorized access. Finally, the performance of a file system greatly affects the efficiency of the computer system that uses it as its primary data storage and management mechanism.

2.1.1 Access Protocols

Traditional file systems provide either a *record-oriented* or a *stream-oriented* interface that application programs use to access files.

Record-oriented file systems divide files into records. A *record* is the smallest datum that can be manipulated by a record-oriented file system. Records can have either a fixed or variable length. Access to records in a file can be organized in several ways[Dem88, Sha91]: *sequential* files allow access to consecutive records, *direct* files allow access to records by a record id number, *indexed* files allow access to records by an associated key value, and *hierarchical* files allow access to records in a tree organized by key values. Common implementations of these access methods include: ISAM[HS82], which provides both indexed and sequential access, and B-Trees, which are hierarchical files that have efficient indexing and support for sequential access[HS82].

Stream files can be viewed as sequential, direct, fixed-length record files with single-byte (or single-word) records. Stream-oriented systems are simple and have few access functions; record-oriented systems are more complex and have more, highly-parameterized access functions.

Besides traditional file systems, some operating systems provide object-oriented file systems, also called *object stores*[SGM89, MG89] or *object-oriented databases*[FAC$^+$89, KBC$^+$89], which present storage as a collection of *persistent* objects. These persistent objects resemble the objects provided by an object-oriented programming language, except that their lifetimes exceed a single execution of a program[AG89a, RCS89, ABCM83]. Object stores exhibit several features not found in traditional file systems: persistent objects can provide many operations unrelated to data storage and retrieval, they can be loaded into memory and written back to secondary storage without being explicitly opened or closed, and they can directly refer to other persistent objects.

2.1.2 Naming

A *file identifier* is a tuple of values[1] that uniquely names a file within a computer system. Files are given symbolic names by placing their identifiers in one or more dictionaries (also called directories). A *dictionary* is a mapping from symbolic names to file identifiers. Dictionaries themselves can be considered files.

Systems that allow multiple dictionaries to refer to a single file identifier (e.g. UNIX[Tho78]) use the number of these references, also called *links*, to determine the lifetime of a file. When a file has no links, its contents are deleted, and its storage is reclaimed.

Unlike object stores, which allow persistent objects to have direct inter-object references, traditional stream and record-oriented systems allow only dictionaries to refer directly to files.

2.1.3 Operations

To support necessary file manipulation and management, file systems provide operations [Dei84, Org87] that:

- create and name a new file or overwrite an existing file (*create*),
- destroy the attributes and contents of a file (*delete*),
- change a symbolic name referring to a file (*rename*),
- create another symbolic name referring to a file (*link*),
- remove a symbolic name referring to a file (*unlink*),
- gain access to a file (*open*),
- relinquish access to a file (*close*),
- inspect the attributes of a file,
- change the attributes of a file,
- locate a record within a file (*seek*),

[1] For example, an ordered pair of integers identifying a file container within a computer system and a file index within the container.

- retrieve one or more records of a file (*read*),
- alter one or more records of a file (*write* or *update*),
- add one or more records to the end of a file (*write* or *extend*), and
- iterate over the names in a dictionary.

These operations can be grouped into those that operate on whole files: create, delete, rename, link, unlink, open, close, inspect attributes, and change attributes, and those that control access to portions of a file's contents: seek, read, write, update, and extend. More complex file systems provide more operations, including those that insert and delete individual records, which operate only on files that have a record organization. Not all users should be allowed to perform all operations on all files; therefore, files and their contents must be protected.

2.1.4 Protection

File systems, except for primitive or single-user systems, protect files and their contents by requiring ownership of files and restricting access to various groups of users. These restrictions can be imposed using various concrete implementations of an abstract protection mechanism called an *access control matrix*[Tan86, Dei84]. In the access matrix model, programs execute within a *domain*, which is a set of pairs of *access rights* and *objects*. Each right allows programs within the domain to perform a single operation or set of operations on the associated object. For example, a domain could have the right to read one file, the right to change the attributes of another file, and the right to both read and write a third file. A complete access matrix contains a row for each domain and a column for each object. Each element in the matrix contains the rights of the domain for the object.

For space efficiency, access control matrices are commonly implemented using either *access control lists* or *capability lists*[Tan86]. An access control list represents a column of an access control matrix. Each list corresponds to an object and contains an element for each domain that has some rights to access the object. The elements of the access control list are the rights of the domain for the object. A capability list represents a row of an access control matrix. Each list corresponds to a domain and contains an element for each object that the domain has some rights to access. As with the elements of access control lists, the elements of the capability list are the rights of the domain for the object.

File systems often use a combination of access control lists and capability lists. Access control lists are ideal for protecting objects in secondary storage while they are not being used, whereas capability lists are more efficient for objects that are being accessed by one or more domains.

2.1.5 Attributes

File attributes describe a file and its contents. The following are common file system attributes:

- the unique file identifier,
- the amount of data (in bytes, blocks, or records),
- the type (e.g stream-file, record-file, dictionary, or object-class),
- data format (e.g. byte ordering, floating point format, record layout),
- ownership information,
- access permissions, and
- various operation timestamps.

Within this thesis, a file name stored in a dictionary will not be treated as a file attribute. Record-oriented systems maintain additional attributes for each file[Org87], and object-oriented systems allow users to specify additional file attributes[Flo88, CSG+88].

2.1.6 Other Terms

Systems designers employ various techniques to increase the performance, simplify the management, and increase the reliability of file systems. The following paragraphs give some of the common terminology for these techniques.

To optimize performance, file systems commonly provide a *buffer cache*[Bac86, NWO88], a section of primary memory that holds the most recently read or written data, or *memory-mapped* files[RC89], which use the virtual memory provided by an operating system to cache recently accessed data. Other systems cache disk tracks[Tan86] or read a whole file into memory when it is first accessed[HKM+88]. While buffer caches and memory-mapped files help overcome delays caused by mechanical parts of disks, these delays still dominate file I/O times. To reduce disk delays, file systems can use a technique called disk or file *striping*[DS89] in which consecutive blocks of a file are allocated on separate disks.

To ease file management within a group of computers, *distributed* file systems[SGK+85, RFH+86, HKM+88, WPE+83] allow computers to share files transparently. A *file server* is a computer that provides other computers, called *clients*, access to its files. To ensure file *consistency*, distributed file systems can support *immutable* files[GNS88], whose modification results in the creation of new files, or they can use various caching policies[NWO88].

2.1.7 Summary

File systems provide an abstract interface to support the storage, retrieval, organization, sharing, naming, and protection of persistent data. They can present various kinds of interfaces, including:

- unstructured data (stream-oriented),
- structured data (record-oriented), and
- encapsulated structured data (object-oriented).

Dictionaries map external, symbolic names to internal file identifiers. File systems support both operations that work on whole files and operations that work on a portion of a file's contents. Various mechanisms protect persistent data by limiting the set of operations a program is allowed to invoke on particular files. Besides storing the contents of a file, file systems store descriptions of a file, including its size, type, ownership, and access permissions. To improve performance, file systems cache recently accessed data, and to simplify management of data within a group of computers, file servers allow other computers to access their files transparently.

2.2 Object-Oriented Programming Terms and Concepts

Object-oriented systems exhibit several software engineering benefits through the use of several programming techniques. The benefits include the enhanced reusability, maintainability, portability, customizability, and extensibility of software[Rus91]. The techniques include *data encapsulation*, *data abstraction*, *interface and code sharing*, *polymorphism*, and *design sharing.*

2.2.1 Data Encapsulation

An *object* is a software entity that consists of a set of state data and a set of *operations* on the data[Weg87, Rus91]. Objects support *data encapsulation* by enforcing the use of operations to effect all state changes[Dig86, Nie89]. Data encapsulation helps software designers decompose systems into objects by insulating other objects from changes in the structure of the state data.

In Smalltalk[GR83], invoking an operation is called *sending a message* to an object, and the code invoked in response to a message is called a *method.* In C++[Str86], methods are called *member functions. Public* member functions define an object's external interface, while public and *private* member functions[2] define an object's behavior. An object's state is composed of its *instance variables*[Nie89], which are also called the *data members*[Str86] of the object.

2.2.2 Data Abstraction

A *class* is a template for building similar objects[Nie89]. A class specifies the instance variables and operations that all objects belonging to the class will have. Classes support *data abstraction* by describing the structure and behavior shared by a set of objects[GR83, Str86]. Classes allow programmers to extend the set of builtin types provided by the programming language. Classes differ from traditional user-defined type mechanisms because they support the notion of *sub-types*[Nie89]. A class defines a sub-type of a second class if instances of the first class can be used anywhere that instances of the second class could be used.

2.2.3 Interface and Code Sharing

Classes can belong to a *hierarchy* and share structural and behavioral descriptions using *inheritance.* Within a hierarchy a *subclass* can inherit from one or more *parent* classes, or *superclasses.* If subclasses have more than one superclass, the hierarchy uses *multiple inheritance*; otherwise, it uses *single inheritance*.

[2] Within this thesis, no distinction will be made between the *private* and *protected* keywords of C++.

Some classes serve only to specify an interface, i.e. a set of operations, that subclasses can share. These classes are called *abstract* classes, and they cannot have any instances[GR83]. Other classes implement the specified interface of an abstract superclass. These classes are called *concrete* classes. In practice, many classes specify both an interface for subclasses and provide an implementation of that interface.

Though the framework presented in this thesis uses classes, instances, instance variables, and subclassing, not all object oriented programming languages use these mechanisms. The Self Language[US87] achieves data encapsulation, data abstraction, interface sharing, and code sharing using a simpler set of mechanisms. In Self, classes are not distinguished from other objects; instead, any object can serve as a *prototype* for other objects. Furthermore, Self does not distinguish between accessing an instance variable and sending a message to an object. Inheritance from objects other than classes is often called *delegation*[Ste87, Nie89].

2.2.4 Polymorphism

Object-oriented languages provide *polymorphic* functions, which can be applied to objects of various types[Dig86, Nie89]. Polymorphism allows software to be more general (applicable to more kinds of data) and extensible (applicable to as yet unspecified data).

Polymorphism is supported by the *late binding* of function calls. The C++ mechanism that provides late binding is called a *virtual function* call[Str86]. Each instance of a class that defines or inherits virtual functions has a pointer to a virtual function table, called a *vtable*. When a virtual function is invoked on an object, the appropriate function is dispatched by retrieving the address of the function from the vtable.

2.2.5 Design Sharing

The most important form of software reuse is reusing the architecture or functional decomposition of systems within the same application domain[Deu89]. Abstract classes are reusable designs for components of a system[WBJ90], and frameworks are reusable designs for entire systems or subsystems.

A *framework* is a collection of abstract and concrete classes and a design for how to combine objects together to build a working system[JR91]. Such a working system, a set of cooperating objects that are an instantiation of a framework, is called an *ensemble*[JR91]. Frameworks describe how a system is decomposed[Deu89] and how its components interact. They allow not only the reuse of design and but also implementation[WBJ90].

Frameworks are similar to an architectural specification for a system, but polymorphism allows the components to belong to various related classes instead of being restricted to a single data type. Frameworks are useful for both building and characterizing systems.

Frameworks often start as single solutions to problems within specific domains. If they are carefully extended and revised, they can serve as solutions to many or most of the problems within their domains[JR91]. Thus they codify knowledge about a problem domain by describing the common aspects of the solutions[JR91]. By being designed for refinement, they support iterative research and development.

2.2.6 Summary

Advocates of object-oriented programming proclaim that the use of the five techniques discussed in this section leads to the following software engineering benefits[JR91, Rus91, Nie89]:

- Reusability—interfaces and code can be shared between classes in a hierarchy; also, designs of similar classes can be shared even when there is no inheritance relationship. Frameworks support the reuse of the designs of entire systems.
- Portability—data encapsulation helps to isolate machine-dependencies.
- Maintainability—the modularity provided by objects not only helps to isolate machine-dependencies, but also helps to keep changes in one module from affecting another.
- Customizability—systems can be tailored to suit the needs of a particular application without including features needed by other applications.
- Extensibility—new features can be added more easily to a system that has an appropriate set of abstractions and supports polymorphism.

Chapter 3

Related Work

This chapter is divided into two sections. The first section briefly surveys various types of file systems that are currently in use or being developed. The second section discusses approaches to system software design, especially file system design.

3.1 Survey of File Systems

This section surveys three categories of interfaces for file systems: record-oriented, byte-stream-oriented, and object-oriented. It also discusses the distribution of persistent data across several machines and alternative ways of utilizing storage media.

3.1.1 Record File Systems

In early computer systems, programs had to communicate directly with hardware I/O devices. To simplify the task of writing application programs, file systems, which provide abstract models of I/O devices, were developed to match models needed by programming languages[Dem88].

Many operating systems, including Digital Equipment Corporation's (DEC) VMS[Sha91] and International Business Machine's (IBM) System 38 and OS 400[Dem88], provide record-oriented file systems. The records in these systems correspond to programming language constructs like PL/1[Hug79] or C[KR78] structures or COBOL[SS79] or Pascal[JW75] records. These record-oriented systems all support many access modes, indices and record formats. The VMS system supports *sequential*, *relative*, and *indexed* file organizations, and sequential and random access modes. The VMS file system also supports multiple versions of a file with the same name.[1]

By supporting and optimizing commonly performed file manipulation operations and by matching the file system models used in popular programming languages, record-oriented systems can help to increase the productivity of both programmers and computer users. Nevertheless, record-oriented systems are often complex and lead to incompatibilities between programs using different file types. Some of these complexity and incompatibility problems, plus the rising popularity of programming languages like C[KR78], have led system developers to move towards stream-oriented file systems.

3.1.2 Byte-Stream File Systems

UNIX[Bac86] file systems are arguably the most important examples of stream-oriented file systems. Simplicity characterizes the standard *System V* UNIX file system[Tho78] model. UNIX files are sequences of bytes with random access and a simple interface: read, which retrieves a consecutive sequence of bytes; write, which stores a consecutive sequence of bytes; and lseek, which sets the current file offset. Because the operating system does not explicitly require record structures for files, the output from most UNIX tools can be the input to other UNIX tools. 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. Within each file container, files are named by a set of special files called directories, which are organized as a single hierarchy.[2]

Stream file systems are highly flexible and have few restrictions, yet they place an extra burden on the programmer. Being untyped and unstructured, stream files result in replication of common file manipulation code in application programs and missed optimizations that would take advantage of an internal file structure. Furthermore, untyped files do not identify the format of the data they contain, and this can cause problems in a distributed environment where byte ordering and floating point format can vary from machine to machine. Treating all files as byte-streams is analogous to treating all primary memory as sequences of bytes.

[1]See Section 4.3 for more information on the VMS file system.
[2]See Section 4.1 for more details on UNIX file systems.

3.1.3 Object Storage Systems

Instead of containing collections of unencapsulated data structures, object storage systems contain data encapsulated as objects of a particular programming language. Object storage systems have evolved from simple workspaces, to multiuser persistent object stores, to object-oriented extensions of conventional file systems, to sophisticated databases.

3.1.3.1 Workspaces

A *workspace* is a collection of persistent data that contains the complete state of a program between successive invocations. The individual data objects stored within a workspace are inaccessible outside the program that created the workspace. An early use of the concept of a workspace can be found in the COMIT string processing language[Sam69]. Other examples of systems that use workspaces include MIT Scheme, APL, and Smalltalk[GR83]. The workspace in Smalltalk is called an *image*.

Workspaces simply and conveniently support object-oriented programming language environments; however, they are designed to be single-user stores, not integrated with the rest of the data stored in a file system, and limited in the size and number of objects stored.

3.1.3.2 Persistent Object Stores

Persistent object stores remove the single-user and size limitations of workspaces. Though not as simple as workspaces, they are complete file systems, since they provide data sharing, naming, and protection.

The PS-algol programming language[ABCM83] provides a persistent heap for all data types, including first-class procedures, used within the language. To determine which objects are persistent, PS-algol uses the concept of a *database*, which is the transitive closure of all objects reachable from a specified object[Bro89]. PS-algol also supports *transactions* that allow multiple readers or a single writer to access a database. Other than the concepts of databases and transactions, persistence is transparent to PS-algol programs.

Data management in the Comandos project[MG89] is similar to that of the *Choices* project and is inspired by the Eden[PLNZ88], SOS[SGM89], and Clouds[PD88] projects. The Comandos Storage System comprises a set of containers, each of which contains a set of segments.

Commandos supports the following features for persistent objects: user defined types, clustering of objects within a segment, distributed object access, queries on objects, and management of classes. Comandos is designed for distributed workstation computing that exclusively uses persistent objects for data storage.

The EXODUS project[CDRS89] at the University of Wisconsin has built an object store for an extended version of C++ called E[RCS89]. See Section 3.2.3.2 for more details about EXODUS. EXODUS uses an untyped storage system. A group at AT&T Bell Laboratories is developing a similar object store for an extended version of C++ called O++[AG89b] as part of its Object Development Environment[AG89a]. The SOS operating system project[SGM89] has built a persistent object store for yet another extended version of C++. Like EXODUS, SOS used an untyped storage system.

The ET++ project[GWM89] supports object storage in files by converting the objects to a portable, textual form. This is similar to the "fileOut" technique used in Smalltalk to export objects from a workspace. While useful for simple prototypes, storing objects in textual form has several limitations, including poor performance and problems handling inter-object references.

A common feature of all the object stores presented here is that they are restricted to a specific programming language (or dialect) and model of data storage and management.

3.1.3.3 Object-Oriented File Systems

In contrast to persistent object stores, which are designed to support persistence in a particular programming language, the object-oriented file systems described below are hybrids of traditional file systems and object storage.

The Portable Common Tool Environment (PCTE)[Flo88], developed at the Software Engineering Research Centrum in the Netherlands, includes a simple object-oriented implementation of a UNIX file system. It extends the UNIX file system by allowing named attributes to be attached to classes of files. It also adds the concept of one-to-many links between files. These attributes and links can be inherited by subclasses. The *contents* of a PCTE file are accessed as a byte-stream, and the protection system in PCTE is exactly the same as in UNIX. The extended PCTE file system model is simple and partially extensible; however, the resulting model is neither general nor fully extensible, since it only presents a UNIX interface to a file's contents.

The Gypsy (more recently called BiiN) operating system[CSG+88] is an object-oriented extension of the UNIX operating system developed jointly by Siemens and Intel. It provides an extended UNIX file system built on object storage. Each object stored in the file system has a type, which can be a builtin type (like a regular file or a directory) or a user-defined type. User-defined types allow inter-object references and user-defined operations. BiiN also integrates version numbers with the naming of files, and it supports access-lists for file protection.

3.1.3.4 Object-Oriented Databases

The systems described in this section are prototypes that represent active research in the area of object-oriented data storage and management. Since they closely resemble object stores, their features are worth considering when building a general file system model.

Hewlett-Packard (HP) has built a prototype object-oriented Database Management System (DBMS), Iris, that is based on a relational storage manager[FAC+89]. Iris offers various interfaces including: Object SQL, which is an extension to conventional SQL; a "loosely-coupled" interface to LISP, which treats Iris as a set of objects; and a "tightly-coupled" LISP interface, which is based on CLOS. Iris supports "types" that are similar to classes in most object-oriented languages. Types are themselves objects, and they form a multiple inheritance hierarchy. Similar to C++, Iris uses special, efficient representations for primitive types like strings and numbers. The most novel aspect of Iris is the ability to add or remove types from existing objects.

The Microelectronics and Computer Technology Corporation (MCC) has developed an object-oriented database, ORION, with many advanced features[KBC+89]. These advanced features include: dynamic schema evolution, composite object clustering, version management, and multimedia information management. ORION also provides query optimization and transaction management. As a research prototype for LISP-based, single-user workstations, ORION serves as a vehicle for experimentation with several aspects of object-oriented data management.

In contrast to the LISP-based Iris and ORION systems, Gemstone is a Smalltalk-based, multi-user Object-Oriented DBMS developed by Servio Logic Corporation[BMO+89]. Unlike most Smalltalk systems, GemStone is designed specifically to support multiple, simultaneous users, large object spaces, and large objects. For efficiency the GemStone language, an extended Smalltalk dialect called OPAL, provides indices on "nonsequenceable" collections of similar objects. These indices are based on the *structure*, not the *behavior* of objects; this type of indexing seems to violate the data encapsulation of object-oriented programming.

VBASE[AH87] and ONTOS from Ontologics are other examples of object-oriented database management systems. VBASE combines an object-oriented language called COP and an object-oriented database. ONTOS, a newer product, is an object-oriented database for C++.

3.1.4 Distributed File Systems

A major trend in file system research is towards distributed file systems. The goal of these systems is to support an environment of networked computers and to provide better performance.

Sun Microsystems has developed a widely-used distributed file system, the Network File System[SGK+85] (NFS) that includes a stateless server and provides effective file system access to a small group of diskless workstations. As part of its UNIX System V Release 3, AT&T has developed a similar system, the Remote File System[RFH+86] (RFS) that uses a write-through cache to enhance data consistency.

Unlike NFS and RFS, which copy individual blocks on demand, the distributed Andrew File System[HKM+88] copies whole files on demand. It also assumes client machines have local secondary storage devices. Whereas NFS and RFS efficiently support relatively few clients, Andrew scales smoothly[HKM+88]. The Sprite File System[NWO88] is a distributed file system that caches data blocks on both client and server systems, provides the same level of data consistency as RFS, and scales as well as Andrew.

The Cedar file system from Xerox uses immutable shared (remote) files[GNS88] and mutable private (local) files. Like Andrew, Cedar uses whole file copies and local disks on each client. Cedar also provides file versions. The LOCUS operating system[WPE+83] supports a location-transparent distributed file system. The LOCUS file system extends the UNIX file system by supporting file *replication* to increase availability and performance.

All the file systems described in this section, except for Cedar, try to duplicate UNIX file system semantics and concentrate on implementation and performance. These distributed systems are presented in this section because any general model must describe characteristics of file systems such as distribution and data consistency.

3.1.5 Summary

Record-oriented file systems simplify the task of writing application programs by providing abstract models of I/O devices that match models needed by traditional programming languages. Complexity and incompatibility problems led system developers to move towards simple, stream-oriented file systems. Despite their simplicity, stream-oriented file systems place an extra burden on the programmer, since they result in replication of common file manipulation code. Object storage systems contain data encapsulated as objects instead of containing collections of unencapsulated data structures. They have evolved from simple workspaces to multiuser persistent object stores and sophisticated databases. Besides designing and refining file system access protocols, a trend in file system research is towards distribution, which simplifies data management in an environment of networked computers. Many distributed file systems duplicate the UNIX file system interface but provide different levels of performance and data consistency.

3.2 Approaches to File System Design

Different applications and computer architectures place different requirements on file systems. One could try to build a single file system to satisfy the needs of all current applications on all computer architectures. But such an effort would yield a large and overly complex system, and it would ultimately fail, since one cannot anticipate all the requirements of future applications. Instead, one can try several approaches to satisfy the wide variety of file system requirements:

1. allow individual operating systems to implement file systems that are appropriate for their intended applications and then provide a model for interoperability;
2. build flexible or extensible file systems;
3. provide a framework that simplifies the construction of standard and custom file systems.

The approach taken in this thesis combines items 2 and 3, since it provides a framework for building standard and custom file systems that includes flexible file systems, and it supports extensions at run-time.

3.2.1 File System Standards

The standardization efforts presented in this section help to identify file system abstractions and to enhance file system interoperability, but they do not attempt to model many major aspects of file systems, nor do they address the design and construction of file systems.

The International Standards Organization (ISO) has developed a File Transfer and Management (FTAM) Standard[Org87]. Currently it supports only a single, hierarchical model of file systems. In this version of the standard, all other types of file systems must be mapped into this model. Other models are planned for a future version of the standard; none of the planned models, however, is general.

IBM has also published file system standards called Distributed Data Management[Mac89]. DDM is designed to allow several different operating systems to share files. It uses a *few* standardized file models that closely match models found in popular high-level languages. One goal of DDM is to require at most $2n$ types of file conversions for n different types of file systems.

The Eden project at the University of Washington provided a heterogeneous, remote file system[PLNZ88] that makes remote file systems, which may provide different naming and access operations, available to other computers on a network. Eden maps all file systems into a single, simple file *access* model. Its model does not specify the structure of files, instead it specifies how application programs can access files. Applications can sequentially read and write variable-sized, typed records. If the records in a particular file are all the same size and type, then applications can randomly read and write the records.

3.2.2 Extensible File Systems

Though the application interface of the UNIX file system is flexible, its internal organization needed changes to incorporate new types of storage like network file systems. To solve this problem, the Eighth Edition UNIX[ATT85] (a research version of UNIX) file system contains a mechanism that integrates both network and special purpose file systems with the standard UNIX file system[Kil86]. Each file system that is supported by this mechanism is given a *type*. Also, each open file has a *generic inode*, which contains the type of the file, the type-independent information about the file, and a pointer to the type-dependent information about the file. The file system uses the type to select which implementation of the standard operations should be used. For example, there will be different read and write operations for local and remote files.

The addition of file system types and generic inodes proved to be successful for integrating both remote file systems and custom file system (for example, treating processes as files[Kil86]). Further, this mechanism was used as the basis for adding features to a commercial version of UNIX, System V Release 3[Bac86]. While this mechanism makes the UNIX file system extensible, it provides neither a way to extend the application interface nor a means to structure and reuse the internals of similar file systems.

As mentioned in Section 3.1.3.3, the BiiN operating system provides an extensible, object-oriented, UNIX-compatible file system[Bii88]. Besides supporting regular UNIX files and directories, programs can create and access networks of persistent objects that appear as conventional heap-allocated objects. The BiiN file system requires explicit *passivation* of objects to force them onto secondary storage, but it supports automatic object activation that resembles demand paging[PKD$^+$90].

The operations of the BiiN file system are polymorphic, since they can be applied to various types of files. The mechanism that supports these polymorphic operations, called *attribute calls*, resembles the virtual function tables used by C++ programs. Though BiiN presents an object-oriented interface to files, it does not make effective use of inheritance[CSG$^+$88]. BiiN also requires special hardware support (tag bits for pointers to objects) for object access and protection[PJC$^+$90].

In contrast to the file system types of Eighth Edition UNIX, which enable the integration of various types of storage management, BiiN supports types for individual files, which enable the extension of the file system's application interface. Neither system addresses the issue of simplifying file system construction.

3.2.3 Software Frameworks

Currently, many software solutions are invented to solve a particular problem and then reinvented by others who later try to solve similar problems. Many people, especially those who pay for these solutions, would like software developers to benefit from the knowledge gained from building earlier solutions. The users of this knowledge should be able to achieve their goals more quickly, efficiently, and reliably. *Software engineering* is a popular term applied to the simplification of various aspects of building large and complex software systems. In a recent paper[Sha90], Mary Shaw commented that engineering requires the codification of knowledge about a problem domain in the proper format, and that software engineering lacks proper formats for codifying system designs. Techniques like subroutine libraries do help software developers, but they offer little help for system designers, since they do not help identify the components of a system or the relationships between the components.

Nevertheless, there have been some successes in software engineering, for example, compiler construction[Sha90]. After many years of refining the practice of building compilers, researchers have developed common models and notations for describing compilers. Thus, when one starts to design a compiler, one can reuse a standard decomposition of a compiler[ASU86] (for example, a compiler contains several modules including a lexical analyzer, a parser, and a code generator), and one can also use standard notations to describe the grammars accepted by some of the components (for example, regular expressions for the lexical analyzer and context-free grammars for the parser). Software models do not have to be complete to be useful[Sha90]—the current state-of-the-art of compiler construction is the result of many years of research and development. Therefore, Shaw argues that software engineering can be the result of iterative cooperation between research and development.

3.2.3.1 Examples of Software Frameworks

Frameworks have been used successfully to support the construction of several user interfaces, including the Model/View/Controller framework of Smalltalk-80[Gol84] and the C++-based framework called ET++[GWM89], and graphical object editors[Vli90].

Frameworks can also be applied to operating system software. *Choices* is a family of object-oriented operating systems; it is designed as a hierarchy of frameworks[CRJ87]. Besides the file system framework described in this thesis, *Choices* contains frameworks for the design of several other subsystems, including: virtual memory[RC89, Rus91], process scheduling and synchronization, exception handling[RJC88, Rus91], networking[ZJ90], device management[Kou91], and message passing[IC91].

3.2.3.2 Examples of Storage Management Frameworks

The frameworks that are most closely related to the one described in this thesis support construction of database management systems.

The EXODUS project provides a toolkit for building DBMS's[CDRS89]. It is designed to enable rapid development of custom systems that support applications not well-suited to the relational model. EXODUS differs from extensible DBMS's in that it allows a *database implementor* to create a system customized for a particular application. EXODUS also differs from the other frameworks discussed in this chapter. Instead of providing modules that can be plugged together, it provides some flexible components (for example, a storage manager and an extended dialect of C++) and tools to generate other components (for example, a query optimizer generator).

At the University of Texas, researchers working on the GENESIS project analyzed the domain of the file management systems[BBR+89], which are the storage and retrieval subsystems for DBMS's. They observed that a mature software field, one that is well-understood and no longer evolving rapidly, can be standardized and can be described by generic architectures. These generic architectures can be built after studying existing systems, algorithms, and data structures. Once built, they provide a design for reusing their software components.

Though they used a different implementation language, C, and different terminology, the observations, techniques, and experiences of the GENESIS project resemble those described in this thesis.

One key observation is that many decompositions of a system are inappropriate for use as frameworks. For example, consider the decomposition of a file system shown in Figure 3.1, which was found in [BS88]. This example should not be used as a general framework for building file systems for several reasons, including:

- it does not identify the classes of components that compose a file system;
- changes to components in one layer affect components in other layers;
- it does not allow for file data encapsulated by any other types of objects besides directories; and
- it does not abstract systems that have more than one storage management layer.

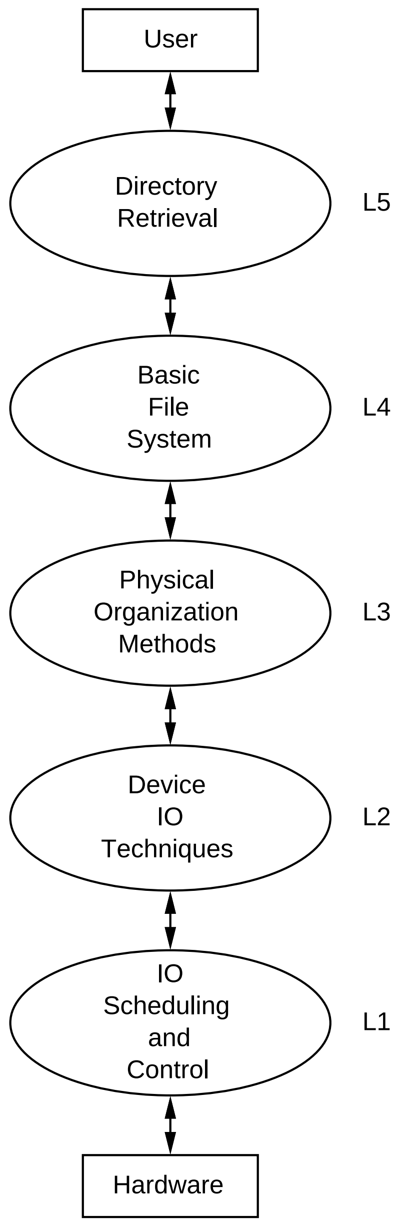

Figure 3.1: A Hierarchical File System

The framework shown in Figure 5.3 and presented in Chapter 5 addresses these deficiencies. The GENESIS team members used the following techniques:

- abstract classes (called *independently-identifiable objects*) capture the fundamental properties of a class of objects;
- many concrete subclasses (called *modules*) encapsulate the implementation details of existing systems; and
- polymorphism allows various types of modules to be used by the same functions.

Some experiences common to both the GENESIS and *Choices* projects include:

- the development of frameworks takes much longer than the development of just a single system;
- frameworks take much experience to develop;
- frameworks are the result of iterative design;
- frameworks allow one to quickly build custom systems; and
- frameworks do not provide solutions to problems within the domain, instead they enable the reuse of existing solutions and experimentation with proposed solutions.

3.2.4 Summary

One could choose between various approaches when designing a general and extensible file system model. Figure3.1 summarizes the benefits of the approaches described in this section:

- file system standards that provide a model for interoperability,
- flexible or extensible file systems,
- frameworks that simplify the construction of file systems.

File system standards like FTAM, DDM, and the Eden remote file system all achieve their goal of supporting interoperability, but they are not designed to solve the problems of the construction or extension of storage management and file system interfaces. Further, they only

Approach	Features					
	Storage Management		Application Interface		Interoperability	
	Extend	Construct	Extend	Construct	Sets of Files	Per File
Standards					●	
UNIX	●				●	
BiiN			●	○		
Genesis	●	●				
Exodus			●	●		
Choices	●	●	●	●	●	●

Table 3.1: Approaches to File System Design

allow interoperability between sets of files on different systems. They do not support different kinds of files stored in the same collection; for example, they do not allow both System V and BSD directories in the same disk partition.

Flexible file systems like the Eighth Edition UNIX file system support the extension of the storage management subsystem through the use of the generic inode mechanism, but they provide little support for constructing these extensions. Like the file system standards, they provide interoperability between different kinds of collections of files, but not different kinds of files within the same collection. Extensible file systems like the BiiN file system allow programmers to extend the file system interface, but they provide limited support for constructing these extensions.

Others have built frameworks that support both the extension and construction of either storage management systems (Genesis) or persistent storage interfaces (Exodus). The *Choices* file system framework, presented in this thesis, supports both the extension and construction of both storage management systems and persistent storage interfaces. Further, it provides interoperability of both different kinds of collections of files and individual files within the same collection.

Chapter 4

File System Abstractions

This chapter discusses some of the abstractions found in three popular file systems. The terms that I use are not the ones that the implementors of these systems used. Instead, I describe the file systems in *object-oriented* terms. Therefore, this chapter presents the kinds of objects that compose these systems, including their data structures and the operations that they support. By using the same technique to analyze the file systems and by describing them using a common set of terms, I lay the groundwork for showing how they can all be constructed within a single coherent framework in Chapter 5.

The first set of abstractions presented are those of the UNIX file system. Three variants of the UNIX file system have been implemented within the *Choices* file system framework because they are used by various computers to which *Choices* has been ported. Not all versions of the UNIX operating system support all the features discussed in this chapter. See Appendix B for details about the differences between them and for examples of how the framework captures their commonalities.

The second set of abstractions are those of the MS-DOS file system. Though many of its data structures are fundamentally different from UNIX file system data structures, many of the operations are similar. Therefore, one can build a model that incorporates both systems.

The third set of abstractions are the record-oriented file interfaces of the VMS operating system. These interfaces contain many operations that are not found in either UNIX or MS-DOS file systems. Nevertheless, the same model that describes stream-oriented and object-oriented file systems can easily describe the operations on VMS files.

4.1 UNIX File Systems

The UNIX file system presents an abstract interface that allows multiple users to access persistent data and many types of hardware devices[Bac86][1]. This support includes a hierarchical structure for directories, mechanisms for sharing files between users, and protection for files and directories. The UNIX file system contains several layers of abstractions. The upper layer, which I call the *application interface layer*, contains transient objects that present the external interface of the file system. The middle layer, which I call the *persistent object layer*, contains persistent objects that are stored within the file system. The bottom layer, which I call the *storage management layer,* contains storage devices and file control data structures. I will describe the objects and operations in each layer, starting with the application interface layer.

4.1.1 The Application Interface Layer

Application programs access the file system through a set of system calls. The systems calls can be classified by the type of object on which they operate: those that operate on the file system as a whole, those that operate on an individual open file, and those that operate on an individual open directory. This classification of system calls implies that three classes of objects compose the interface to UNIX applications:

1. a file system interface object,
2. a file stream, and
3. a directory stream.

4.1.1.1 File System Interface Operations

Most UNIX file system operations operate on a file system interface object. A single file system interface object operation can name and perform operations on all files and directories accessible to the computer. A directory is a special file that contains ordered pairs (also called *associations* or *entries* in this thesis) that map file names to internal file identifiers (called *inumbers*). Every directory contains at least two entries: one that maps the string "." to the inumber of the directory itself, and one that maps the string ".." to the inumber of its *parent* directory. Each

[1]In this analysis of the UNIX file system, I only discuss persistent data storage.

disk partition contains one root directory, which maps both "." and ".." to its own inumber. The UNIX file system presents a hierarchical naming structure by allowing directories to contain the names of both regular data files and other directories.

All file system interface operations take either one or two *pathname* arguments. A pathname is a sequence of zero or more directory names followed by a single file or directory name. The file system interface interprets each component of the pathname sequentially; it looks up each component in the directory specified by the previous component. Slash ("/") characters separate the components of pathnames. The set of directory names are called the *dirname*, and the last element of the pathname is called the *basename*. Pathnames can be either *absolute* or *relative*. Absolute pathnames begin with a "/" character, while relative pathnames begin with any other character. The file system interface looks up the first component of an absolute pathname in the *root* directory, while it looks up the first component of a relative pathname in the *current* directory. All processes in UNIX systems share the same root directory, but processes can either share the same current directory or have different ones.

All files and directories are *owned* by a single user and also belong to a *group* of users. For each file there are three types of *permissions*: read, write and execute. For each directory there are also three types of permissions: read, write and search. The UNIX operating system divides processes into three categories. The first category contains processes that belong to the owner of the file, the second category contains processes that belong to other members of the group to which the file belongs, and the third category contains all processes not in the first two groups. Both files and directories store nine flags that indicate which subset of the three permissions each of the three categories of processes has.

All file system operations could be designed to take pathnames as arguments, but it would be inefficient to use many such operations repeatedly on the same file. To make operations such as read and write more efficient, the UNIX file system keeps a table of open files. An entry in the file table refers to the file's control information, which has been loaded into primary memory. Each entry also contains a flag that indicates the access mode specified when the file was opened (either read-only, write-only, or read/write). For each process, the UNIX operating system keeps a table of references to file table entries. These tables are called *file descriptor tables*, and an integer index into these tables is called a *file descriptor*.

The file system interface supports five types of operations: file access and creation (open and creat), persistent changes to the name space (mkdir, rmdir, link, unlink, rename, symlink, and readlink), temporary changes to the name space (chdir, chroot, mount, and unmount), attribute access (stat and lstat), and protection (access, chmod, and chown).

The open operation allows programs to access the contents of a file through a file stream object. It takes two arguments, a pathname and the access mode (read, write, or read/write), and returns a file descriptor. The creat operation allows programs to create a new file and to access its contents through a file stream object. It takes two arguments, a pathname and the nine file permission flags, and returns a file descriptor that refers to a file opened for writes only.

The mkdir operation takes a single pathname argument and creates a directory that corresponds to the given name. The rmdir operation takes a single pathname argument and deletes the directory that it names. The link operation allows programs to create multiple names for a file. It takes two pathname arguments and adds an entry to the directory specified by the dirname of the second argument that maps the basename of the second argument to the inumber of the file specified by the first argument. The unlink operation allows programs to remove a file name. It takes a single pathname argument and removes an entry from the directory specified by the dirname of the argument that maps the basename of the argument to an inumber. The rename operation allows applications to change a name of a file. It takes two pathname arguments and atomically performs a link and an unlink operation. The symlink operation allows programs to store a pathname in a special file called a *symbolic link*. It takes two pathname arguments and adds an entry to the directory specified by the dirname of the second argument that maps the basename of the second argument to the inumber of a file that contains the first argument. The readlink operations allows programs to retrieve the contents of a symbolic link. Except for the readlink and lstat operations, all file system interface operations use the contents of symbolic names during pathname parsing. Operations that use symbolic names during pathname parsing replace the current component of the pathname with the contents of the symbolic link and then resume parsing the pathname.

The stat operation takes a single pathname argument and returns a data structure that contains all of the file's attributes. The lstat operation also takes a single pathname argument and returns a data structure that contains the file's attributes. If the basename of the argument corresponds to a symbolic link, however, it returns the attributes of the symbolic link instead of the attributes of the file named by the symbolic link.

The chdir operation changes the value of a process's current working directory, while the chroot operation changes the value of the root directory. The mount operation allows programs to create a temporary mapping from a directory on one disk to a directory on another. It stores these mappings in a *mount table*, which can be considered an auxiliary object that is used by the file system interface. It takes three arguments, the pathname of special file (for example, a disk partition), the pathname of the mount-point directory, and a flag that allows files to be mounted read-only. The unmount operation allows programs to destroy a temporary mapping from a directory on one disk to a directory on another. It takes one argument, the pathname of special file.

The access operation allows programs to determine which access rights they have for a given file or directory. It takes two arguments, a pathname and a set of access flags. The access flags allow the process that invokes the operation to check for any combination of file permissions (either read, write and execute for regular files, or read, write and searchfor directories). They also allow the process to check if the file exists. The chmod operation allows programs to change the access rights for a file. It also takes two arguments, a pathname and set of mode flags. The chown operation allows programs to change the ownership of a file. It takes three arguments, a pathname, a user name, and a group name.

Three variables compose the state of the file system interface: the root directory, the current directory, and the mount table.

4.1.1.2 File Stream Operations

A UNIX file is a random access sequence of bytes; the application interface to files, however, is called a *file stream.* The term file stream is appropriate because the data retrieval and storage operations, read and write, support sequential access as the default access mode. The operations on file streams include read, write, lseek, fstat, truncate, and close.

The read and write operations take three arguments: a file descriptor, a buffer location, and the number of bytes to be transferred. The read operation attempts to retrieve the specified number of bytes and place them in the buffer, and it returns the number of bytes successfully read. The write operation attempts to store the specified number of bytes and place them in the buffer, and it returns the number of bytes successfully written. Both operations use a *current file offset* to support sequential access. Programs that require random access to file streams can reposition the file offset by invoking the lseek operation, which takes the following three arguments: a file descriptor, a new file offset, and a operation code to indicate whether the new offset is relative to the beginning of the file, the current offset, or the end of the file.

The fstat operation retrieves the attributes of a file specified by a file descriptor argument and places them in a buffer specified by a buffer location argument. The truncate operation sets the size of a file specified by a file descriptor argument to a value specified by a number of bytes argument. The close operation informs the operating system that the program no longer needs the file specified by the file descriptor argument.

Each file stream (each entry in the open file table) has two variables: the value of the current file offset and a reference count, which indicates how many user file descriptor tables refer to it.

4.1.1.3 Directory Stream Operations

The UNIX file system supports many operations on directories, but application programs can directly invoke only one operation: getdents. The getdents operation allows programs to iterate over the file names in a directory.[2] It takes three arguments, a file descriptor, a buffer location, and the size of the buffer. The file descriptor must refer to an open file stream object that corresponds to a directory. The first invocation of getdents returns as many directory mappings as will fit in the buffer. The format of each mapping is a file system independent[3] data structure that contains both the symbolic name of the file and the file's inumber.

4.1.2 The Persistent Object Layer

Objects in the application interface layer create, access, and modify three types of persistent objects in the UNIX file system: regular files, which are structured as arrays of characters; directories, which map file names to inumbers; and symbolic links, which map an inumber to a path name. Thus, three classes of objects compose the persistent object layer of the UNIX file system:

[2]In early versions of UNIX, programs read directories as if they were regular data files and interpreted the directory data structure. This method was non-portable and violated the encapsulation of directory data.

[3]Some versions of UNIX support System V, 4.2 BSD, MS-DOS file systems, which all use different data structures for directory entries.

1. persistent arrays of characters,
2. directories, and
3. symbolic links.

The stat, lstat, and fstat operations return the attributes of any class of persistent object using the same data structure. Therefore, all classes of persistent objects share the same operation, called info, that returns a structure that describes the attributes of the object.

Each persistent object also has a state variable that refers to a block-oriented storage object, called an *inode*. Section 4.1.3 describes several classes of block-oriented storage objects, including inodes.

4.1.2.1 Persistent Character Array Operations

Regular files in UNIX resemble variable-sized arrays or characters with a lower bound of zero and an upper bound of one less than the size of the file. The atPut operation stores elements in the array. It three arguments: the starting element number, the number of consecutive elements to be stored, and a buffer containing the elements. If it stores some of the consecutive elements above the upper bound of the array, it automatically raises the upper bound. If the starting element is above the upper bound, all elements between the current upper bound and the starting element are set to zero. The at operation retrieves elements from the array. It takes three arguments: the starting element number, the number of consecutive elements to be retrieved, and a buffer to hold the elements. If some of the consecutive elements to be retrieved are above the upper bound of the array, then it retrieves only those up to the upper bound. Both operations return the number of elements that they successfully store or retrieve.

The size operation returns the current number of elements in the array. The setSize operation takes one argument and changes the upper bound to be equal to one less than the argument. If the new size is smaller than the old size, it discards elements of the array above the new upper bound. If the new size is larger than the old size, it sets elements of the array from the old upper bound up to the new upper bound to zero.

The persistent character array class is a simple class. The primary function of all its operations is to convert between the byte-oriented operations required by objects in the application interface layer and the block-oriented operations provided by objects in the storage management layer.

4.1.2.2 Symbolic Link Operations

The symlink operation on a file system interface stores a pathname as a sequence of characters in a symbolic link object. All operations on a file system interface can potentially retrieve the contents of a symbolic link (if one or more of the components of a pathname argument refer to symbolic links). Therefore, the symbolic link objects require operations to both store and retrieve a pathname. Because pathnames are stored and retrieved as sequences of characters, a symbolic link can have the same atPut and at operations as a persistent character array.

4.1.2.3 Directory Operations

As stated previously, directories map file names to internal file identifiers, called inumbers. The persistence of any regular file, symbolic link, or directory is determined by the number of directory entries that refer to its inumber. If that number (also called a *link count*) is greater than zero, then the object persists. If the number is zero, then the object is destroyed and any storage space that it used is reclaimed.

Each directory object has a state variable that refers to a container of inodes. The inumbers stored in a directory are indices into its container.

The open operation takes a file name as an argument and searches all entries for a match. If a match is found, the directory returns the result of invoking the open operation on its container using the inumber that corresponds to the file name as the argument. The create operation also takes a single file name argument. It first performs the same operation as open. If the search for the file name fails, it invokes the create operation on its container, which returns a newly initialized inode. It then stores an entry that contains the original argument as the file name and the inumber of the newly initialized inode. It also sets the link count of the corresponding inode to one.

The add operation takes a file name and an inumber as arguments and stores an entry if the file name is not already stored in the directory. It also increments the link count of the corresponding inode. The remove operation takes a single file name argument and searches all entries for a match. If a match is found, the directory removes the entry that contains the file name. It also decrements the link count of the corresponding inode.

4.1.3 The Storage Management Layer

Each object in the persistent object layer invokes operations on a corresponding inode in the storage management layer. Each directory also invokes operations on a container of inodes. The storage management layer is actually several layers, since inodes refer to disk *partitions*, which refer to *disks*.

4.1.3.1 Inode Operations

An inode has behavior and operations similar to a persistent character array. The difference between an inode and a persistent character array is that inodes are arrays of blocks instead of bytes. All blocks that belong to an inode are the same size, which is a multiple of the block size of the disk partition on which the inode is stored.

The read and write operations take the same arguments and perform the same functions as the corresponding operations on persistent character arrays. Each inode contains a table of mappings from their logical block numbers to the corresponding physical blocks of the disk partition on which the inode is stored. Both the read and write operations use this table of mappings. When a write operation stores data to a block that is currently unmapped, the inode invokes the allocate operation on a *block allocator object*.

The info operation returns the attributes that describe the inode. These attributes include the file's type (regular, directory, or symbolic link), owner, group, permission flags, size, and various time stamps.

The blockSize operation on an inode returns the size of its blocks. The numberOfBlocks operation returns the size of the inode in blocks. The numberOfBytes operation returns the size of the inode in bytes. Both the setNumberOfBlocks and the setNumberOfBytes operations reset the size of the file. If either operation truncates the inode, it also invokes the free operation on a block allocator object to free each logical block that the inode no longer needs.

When an inode is no longer referenced by any object in the persistent object layer, then it invokes the close operation on its container, passing itself as an argument.

4.1.3.2 Inode Container Operations

The control information for groups of inodes are stored together with their data blocks in an indexed collection. I call the object that encapsulates this information an *inode container*. Each container stores and retrieves data from the partition that also stores its inodes data blocks.

The open operation takes an inumber as an argument and returns the corresponding inode if it has been previously been created. The create operation searches for an unused inode, initializes it, and then returns it. The close operation checks to see if the link count of the inode is equal to zero. If the link count is zero, it sets the inode's size to zero and then marks the inode as unused.

4.1.3.3 Block Allocator Operations

All data blocks not used for inode and inode container control information can be used for either data blocks or logical block maps. These blocks are managed by a block allocator object that resembles a heap manager in many programming environments. The allocate operation takes a number of consecutive blocks as an argument. It searches for a cluster of blocks to satisfy the request and, if successful, returns the beginning block number of the cluster. The free operation takes two arguments, the beginning block number of a cluster of consecutive blocks and the size of the cluster in blocks. It makes the blocks available for future invocations of allocate.

4.1.3.4 Partition Operations

An inode container could divide an entire disk into a collection of inodes; however, UNIX systems traditionally partitioned disks into smaller sections to simplify system management[Bac86]. The mount command helps hide the partitioning of a disk from application programs. The four major operations of disk *partitions* (read, write, blockSize, and numberOfBlocks) perform exactly the same functions as the corresponding operations on inodes. Because partitions are contiguous sections of a disk, the mapping of logical partition blocks to physical disk blocks is simpler than the mapping of an inode's blocks to its partition's blocks. Furthermore, because the size of each partition is fixed, it does not need to refer to a block allocator, and it does not need to provide a setNumberOfBlocks or a setNumberOfBytes operation.

4.1.3.5 Partition Container Operations

Just as each inode belongs to a container that divides a partition into an indexed collection of inodes, partitions can be considered to belong to a container that divides a disk into an indexed collection of partitions. The open operation takes a partition number as an argument (also called a *minor device number*), and returns the corresponding partition.

4.1.3.6 Disk Operations

Ultimately, all persistent data must be stored on a physical storage device, usually a hard, floppy, or RAM disk. The read, write, blockSize, and numberOfBlocks operations all perform the same functions as the corresponding operations on partitions. The architecture of the UNIX operating system specifies that all disk drivers should present the same set of operations, regardless of the type of hardware.

4.2 The MS-DOS File System

This discussion of the MS-DOS file system compares and contrasts it with the UNIX file system presented in the previous section. The MS-DOS file system has many similarities with the UNIX file system, especially at the application interface layer. The differences between the two file systems primarily result from the fundamental differences in on-disk data structures and the single-user nature of MS-DOS.

4.2.1 The Application Interface Layer

Numerous MS-DOS system calls provide application programs access to three objects that are similar to the file system interface object, file stream, and directory stream of the UNIX file system.

4.2.1.1 File System Interface Operations

Like its UNIX counterpart, the MS-DOS file system interface object also uses hierarchical directories, and absolute and relative pathnames. Files do not have owners, nor do they belong to groups of users, but they can be marked read-only. Instead of having a single root directory for the entire system and a single current directory for each process, the file system interface stores a current directory for each disk drive or disk partition. It also keeps track of the current drive or partition.

The MS-DOS file system interface has operations that resemble the following UNIX operations: open, create, mkdir, rmdir, chdir, stat, and chmod. It also supports an operation not available on UNIX, which sets the current drive.

MS-DOS does not allow a file to be named by more than one directory; therefore, it does not support a link or unlink operation. It does, however, provide a delete operation instead of unlink. MS-DOS does not support symbolic links; therefore, it does not support a symlink, readlink, or lstat operation. MS-DOS does not have a single root directory for the entire system; therefore, it does not support a chroot operation. MS-DOS does not provide a mount table; therefore, it does not support a mount or unmount operation. MS-DOS does not store ownership information for files; therefore, it does not support a access or chown operation.

4.2.1.2 File Stream Operations

MS-DOS provides the operations on open files that allow application programs to read or write a single record sequentially, to read or write several consecutive records randomly, to reset the size of the file, to set the current file offset for sequential operations, to get the file's attributes, and to close the file. These operations give programs the same functionality as the read, write, lseek, fstat, truncate, and close operations on UNIX file streams.

4.2.1.3 Directory Stream Operations

As with the UNIX file system, the only operations on directories that are accessible to application programs support iterating over the names stored in a directory. Instead of providing a single getdents operation to support this feature, MS-DOS provides two operations. The first operation finds the first file name that matches a pattern specified in its argument. The second operation, which programs can invoke repeatedly, finds a subsequent file name that matches the pattern specified previously.

4.2.2 The Persistent Object Layer

MS-DOS supports only two types of persistent objects: regular files, which are structured as arrays of characters, and directories.

4.2.2.1 Persistent Character Array Operations

Regular files in the MS-DOS file system support the same operations as regular files in UNIX file system.

4.2.2.2 Directory Operations

Directories in the MS-DOS file system support operations that resemble the following UNIX operations: open, create, and remove. MS-DOS allows a file name to be named by only one directory; therefore, it does not support the add operation. Furthermore, MS-DOS does not need to store a link count for each file; therefore, the remove operation always deletes the file.

While the interface of MS-DOS directories resembles its UNIX counterpart, the internal data structures of MS-DOS and UNIX directories differ greatly. UNIX directories store only names and inumbers, whereas MS-DOS directories store both names and file control information. This control information includes the size and attributes of the file, a modification timestamp, and the number of the first data block of the file. Thus, MS-DOS keeps both storage management and naming information within the same object.

4.2.3 The Storage Management Layer

Some objects in the MS-DOS storage management layer closely resemble objects in the UNIX storage management layer, while others, like the *file allocation table* (FAT), have no counterpart in the UNIX file system.

4.2.3.1 MS-DOS File Operations

Each directory entry contains the control information for a file, which supports the same external operations as a UNIX inode. Since directory entries have a fixed size (32 bytes), they cannot contain a table that maps logical blocks to physical blocks. Instead, each entry stores only the location of the first logical block of the file. Each disk contains a single FAT, which stores the data that maps logical blocks to physical blocks

4.2.3.2 MS-DOS File Container Operations

File containers support the same operations as inode containers. Because each directory entry describes a file, the amount of data dedicated to file control information is variable, whereas inode containers have a fixed number of blocks dedicated for inodes. As a substitute for the inumbers used in UNIX to separate symbolic naming from the file identifiers needed by storage management objects, one could use the location of the first logical block of each file.

4.2.3.3 File Allocation Table Operations

The file allocation table stores a linked list of physical block numbers for each file. The mapBlock operation takes two arguments: a logical block number and a flag to indicate whether it should extend the linked list if the mapping for the logical block does not already exist. It the mapping already exists, it returns the physical block number. If the mapping does not exist and no extension is requested, it returns zero. Otherwise, it allocates a block of storage for each mapping that needs to be added to the end of the list. The freeBlocks operation takes a single argument, a logical block number. It truncates the elements of the list beyond the logical block number specified in the argument, and it frees the blocks no longer needed by the file.

4.2.3.4 Disk Operations

Disks support the same operations in MS-DOS as in UNIX. The tradition of partitioning disks is much less common in MS-DOS.

4.3 File Types in the VMS File System

Unlike UNIX and MS-DOS file systems, which present files as persistent arrays of characters, the Record Management Service (RMS) or the VMS file system[Sha91, Dei84] presents files as a collection of *records*.

VMS files can contain either of two major types of records: *fixed length* and *variable length*.[4] A file that contains fixed length records stores the size of its records as one of its attributes, in which case all records within the file have the same size. A file that contains variable length

[4]VMS actually supports three types of variable length records; see [Sha91] for details.

records either stores the size within each record or stores delimiting character strings at the end of each record.

Record files can have one of the following types: *sequential*, *relative*, and *indexed*. These types determine which operations a file supports and the behavior of those operations. All three types of files can contain either fixed or variable length records, but only sequential files can store records delimited by character strings.

4.3.1 Sequential Files

Sequential files are suitable for both tape drives and disk drives. Records can be read consecutively from the start of the file, and appended to the end of the file.

When a sequential file is first opened, the read operation returns the first record. Each subsequent call to read returns the next record. The read operation takes a buffer and the size of the buffer as arguments and returns either the size of the record that is read or an error code.

The write operation on a sequential file takes a buffer and the size of the data within the buffer as arguments and stores the data in a new record at the end of the file if the program has read the last record of the file. The write operation can also over-write an existing record if it is the same size as the one being written. It returns an error code if an attempt is made to over-write a record with a different size.

Many programs sequentially read or write the entire contents of a file, and sequential files adequately satisfy their requirements. Some programs, however, require random access to files.

4.3.2 Relative Files

Relative files are the simplest type of file supported by VMS that provides random access. They divide their storage into fixed length cells that either contain a record or are marked empty. Operations on relative files include a record number argument, which gives programs full control over the order in which they access the records of a file.

The read operation takes a record number, a buffer, and the size of the buffer as arguments and returns either the size of the record that is read or an error code. The error code indicates whether the corresponding cell is empty or the cell does not exist.

The write operation takes a record number, a buffer, and the size of the data within the buffer as arguments and stores the data if the corresponding cell is empty. If the cell does not exist, it creates a new cell and stores the data within it. An error code is returned to indicate that the corresponding cell is not empty.

The update operation takes the same arguments as the write operation, and it performs the same function, except that it over-writes the contents of a record within a non-empty cell.

The delete operation takes a record number as an argument, and it marks the corresponding cell empty.

Relative files support the random access features needed by many programs, but the only mechanism they provide for identifying records is by cell number. A more convenient way to access random records is by providing operations that identify records using symbolic names.

4.3.3 Indexed Files

Indexed files identify records using sets of symbolic *keys* called *indices*. An indexed file must have at least one index, called a *primary* index. The keys in the primary index, which are called primary keys, must be unique. Besides the primary index, files can also have up to 254 *alternate* indices. The keys in an alternate index do not need to be unique.

Operations on indexed files can access records either randomly and sequentially. Random access operations take a key and an index number as arguments. Sequential operations access records in the sorted order of a particular index.

The data for a particular key comes from a fixed location within each record, which can be described by its *offset* and *length* within the record. VMS allows an indexed file to be partitioned into *areas*. Files can store all data within a single area, or they can store records in one area and the keys in other areas. Application programs that store each index for a file in a separate area increase the locality of data accesses, since the data for an index will be stored close together. Therefore, these programs will perform better.

As mentioned earlier, both random and sequential interfaces for indexed files could be implemented by providing operations that accept keys and index numbers as arguments. Relative file and indexed files, however, share common behavior. Therefore, I implement the interface for indexed files by having it inherit the interface of relative files and augmenting this interface with two operations: keyToRecord and keyToRecordNext.

The keyToRecord operation takes an index number and a key as arguments. It returns the relative record number if the key is found within the specified index; otherwise, it returns an error code. The keyToRecordNext operation takes the same arguments as the keyToRecord operation, and it performs the same function. It also returns the value of the next key within the same index; therefore, it supports both random and sequential access.

The read operation takes the same arguments and performs the same function as the read operation on relative files.

The write operation takes the same arguments and performs the same function as the write operation on relative files. It also extracts the values for each of its keys and inserts them into the appropriate indices.

The update operation takes the same arguments and performs the same function as the update operation on relative files. It also removes the values of the old record's keys from each index. It then extracts the values for each of its own keys and inserts them into the appropriate indices.

The delete operation takes the same arguments and performs the same function as the delete operation on relative files. It also removes the values of the old record's keys from each index.

4.4 Summary

The UNIX and MS-DOS file systems support only two or three types of persistent objects: arrays of characters, directories, and symbolic links. This simplifies the storage management layers of both operating systems, but it also limits their application interfaces layers.

Besides support directories, VMS supports six combinations of record and file access types. These types of files enable VMS to present more features in its application interface, but they also require features not found in the storage management UNIX or MS-DOS. The goals of this thesis (generality, simplicity, and extensibility) require

- that a single model incorporate the types of persistent objects found in the byte-stream oriented systems like UNIX, the record-oriented systems like VMS, and the object-oriented systems like those mentioned in Section 3.1.3.3;
- that a few components model these features; and
- that more features can be added easily without changing the model.

Chapter 5

File System Framework

The file system framework contains a hierarchy of abstract and concrete classes that can be combined to build both standard and customized file systems. This chapter introduces the framework and categorizes and describes its abstract classes. Some classes provide interfaces to applications and other subsystems, some classes define objects that function as internal building blocks, and other classes serve both purposes. System programmers can use the framework to build new file systems, both conventional and experimental. Application programmers can use the interface classes to access and manipulate the persistent data stored within any file system built within this framework. The framework is also dynamically extensible, since it supports the addition of classes at run-time.

5.1 Examples of the Framework

There are two orthogonal ways to view the file system framework. The primary way is to categorize objects into two classes:

- the PersistentStore[1] class defines persistent data stores, each of which stores and retrieves blocks of persistent data using a random access method, and

[1]This thesis uses notational conventions similar to [Rus91]. Both the names of classes and operations are printed in sans serif font. Class names begin with capital letters, whereas operation names begin with lower case letters. Plural nouns indicate several instances of a class, and the indefinite article indicates a single instance of a class. For example, "a PersistentStore" is an instance of the PersistentStore class, and "PersistentStores" are instances of the PersistentStore class.

- the PersistentObject class defines persistent objects, which encapsulate and provide a better interface to the data managed by a persistent data store.

A PersistentStore provides random access to an uninterpreted sequence of blocks of data while a PersistentObject interprets the data as having a format. (The most essential objects belong to either of these two fundamental classes. Some objects in the framework, however, belong to neither class. These objects, described in Section 5.2.6 and Section 5.2.3.2, either augment the framework's application interface or enhance the structure and reuse of code.) For example, a UNIX inode is a PersistentStore, while a UNIX directory is a PersistentObject. A disk is a PersistentStore, but a table of descriptors for the files stored on a disk is a PersistentObject. All PersistentStores have the same interface, but the interfaces of different subclasses of PersistentObject differ greatly.

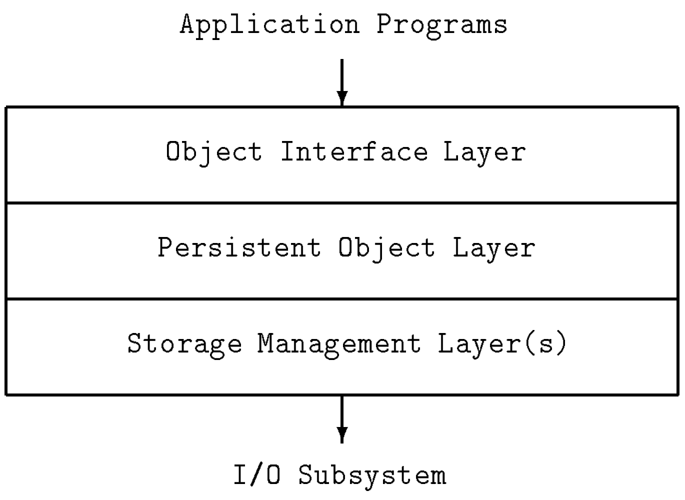

Figure 5.1: Layers of the File System Framework.

The second way to view the file system framework is that it divides a file system and its components into three layers, as shown in Figure 5.1. Objects in the layers of the file system serve the following purposes:

- Objects in the lowest *Storage Management Layer* access hardware devices in the I/O subsystem like disk controllers or network interfaces. Access is defined in terms of fixed-size sequences of blocks, where each block corresponds to a physical block used by the hardware device. Objects in the higher Storage Management Layers manage the contents of the underlying physical stores as nested containers of logical stores. Access is defined in terms of either fixed-size sequences of blocks (for example, disk partitions[2]), or extensible sequences of blocks (for example, UNIX inodes). The Storage Management Layers contain PersistentStores and PersistentObjects that encapsulate storage organization and sharing.

- Objects in the *Persistent Object Layer* encapsulate the data within lower layers and provide several kinds of operations on the data. These operations include support for naming and data structuring. Examples include directories of file names, persistent arrays of bytes, and collections of records. System programmers can extend the framework by designing persistent objects that support many other 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. One class of interface objects defines a per-process interface to the file system that keeps track of the current directory and provides a unified name space for referencing files in the system. Another set of classes defines byte- and record-stream protocols for the data encapsulated within the various types of file structures in the Persistent Object Layer. All objects in the Object Interface refer to PersistentObjects or PersistentStores in the lower layers.

5.1.1 A Concrete Example

To illustrate the kinds of objects that compose an instantiation of the framework, Figure 5.2 shows a simplified BSD UNIX file system built from the framework. The I/O subsystem provides objects in the Storage Management Layer an interface to direct-access block storage devices. In this example, the storage device is a SCSI controller with one or more disks attached. A SCSIDisk object represents one of these disks and provides a block-oriented read/write interface to the physical storage device. The SCSIDisk supports a SCSIContainer, which divides the disk into contiguous regions called Partitions. One of these Partitions uses the disk as its source

[2]A *partition* is a physically contiguous, fixed-size section of a disk.

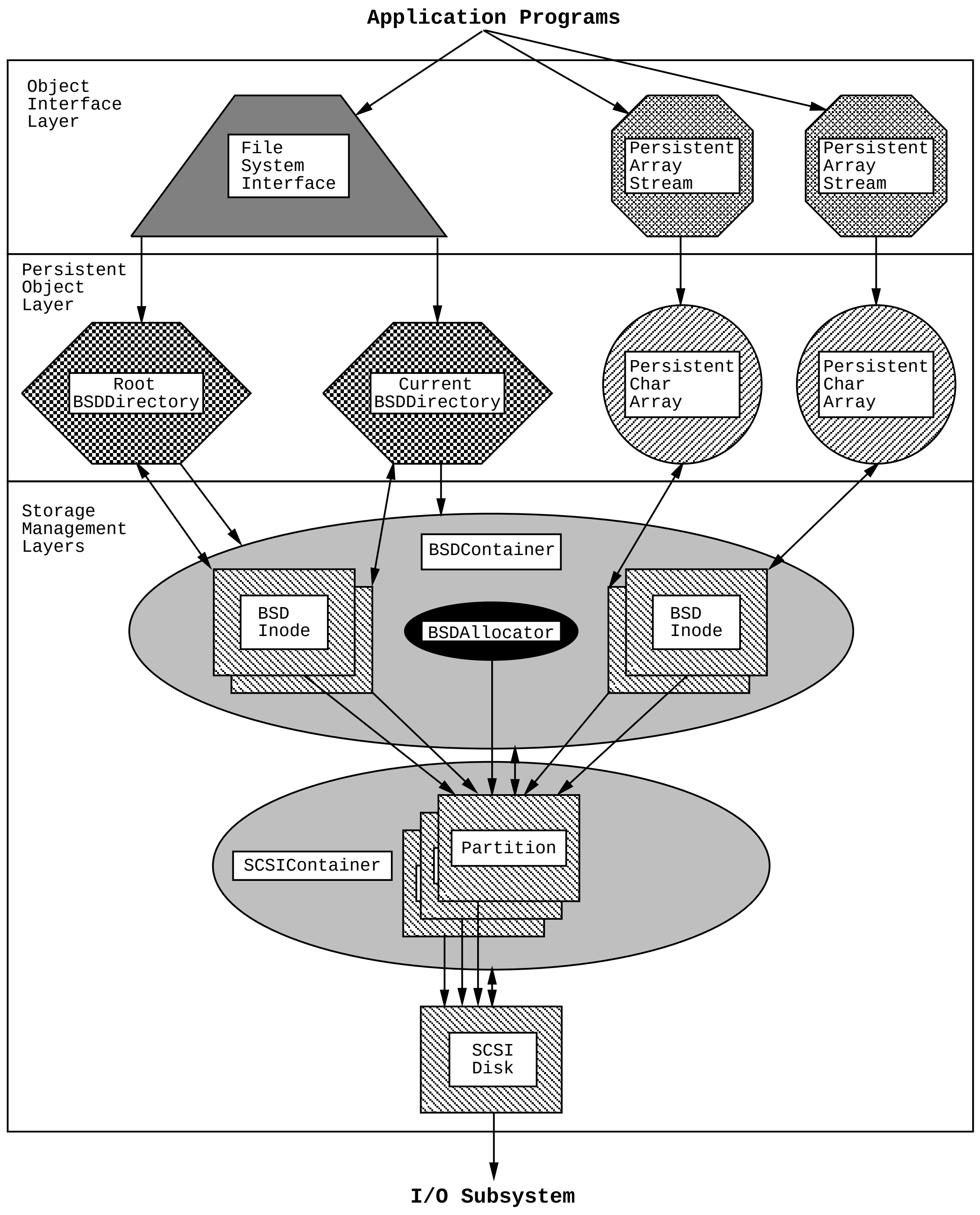

Figure 5.2: Concrete Example of File System Framework

of data and supports a BSDContainer, which organizes the data blocks of the partition as an indexed set of files called BSDInodes.

The BSDInodes use the partition as their source of data, and they use a BSDBlockAllocator to request additional blocks of the partition's storage. Each BSDInode stores the data for an object in the Persistent Object Layer, either a BSDDirectory or a PersistentCharArray. The BSDDirectories provide symbolic names for BSDInodes. The PersistentCharArrays provide access to regular files, which are sequences of bytes that are stored in their underlying BSDInodes.

In the Object Interface Layer, a FileSystemInterface parses *pathnames* into sequences of references to directories that start with either the root or current directory. It provides operations to open, create, and remove PersistentCharArrays, to create and remove directories, and to change the current or root directories.

PersistentArrayStreams provide byte-addressable, stream-oriented access to the data encapsulated within their corresponding PersistentCharArrays. PersistentArrayStreams support read and write operations, which perform the same function as the UNIX read and write system calls. They also support an operation similar to the UNIX lseek system call. PersistentArrayStreams convert the requests of application and system programs into the at and atPut operations supported by PersistentCharArrays.

PersistentCharArrays convert the at and atPut requests on their elements into the block-oriented read and write operations supported by BSDInodes. BSDInodes convert reads and writes of logical blocks into reads and writes of their partition's blocks. Partitions convert reads and writes of their blocks into reads and writes of their disk's physical blocks.

The set of PersistentArrayStream objects is similar to the *file table* used by the UNIX operating system, and the set of BSDInode objects is similar to the *inode table* used by the UNIX operating system[Bac86].

5.1.2 An Abstract Example

The framework abstracts more than just UNIX-like or stream-oriented file systems. As with any abstract framework[WBJ90], its elements can be combined to form various running systems. A more abstract view of the file system framework is shown in Figure 5.3, which uses the following key:

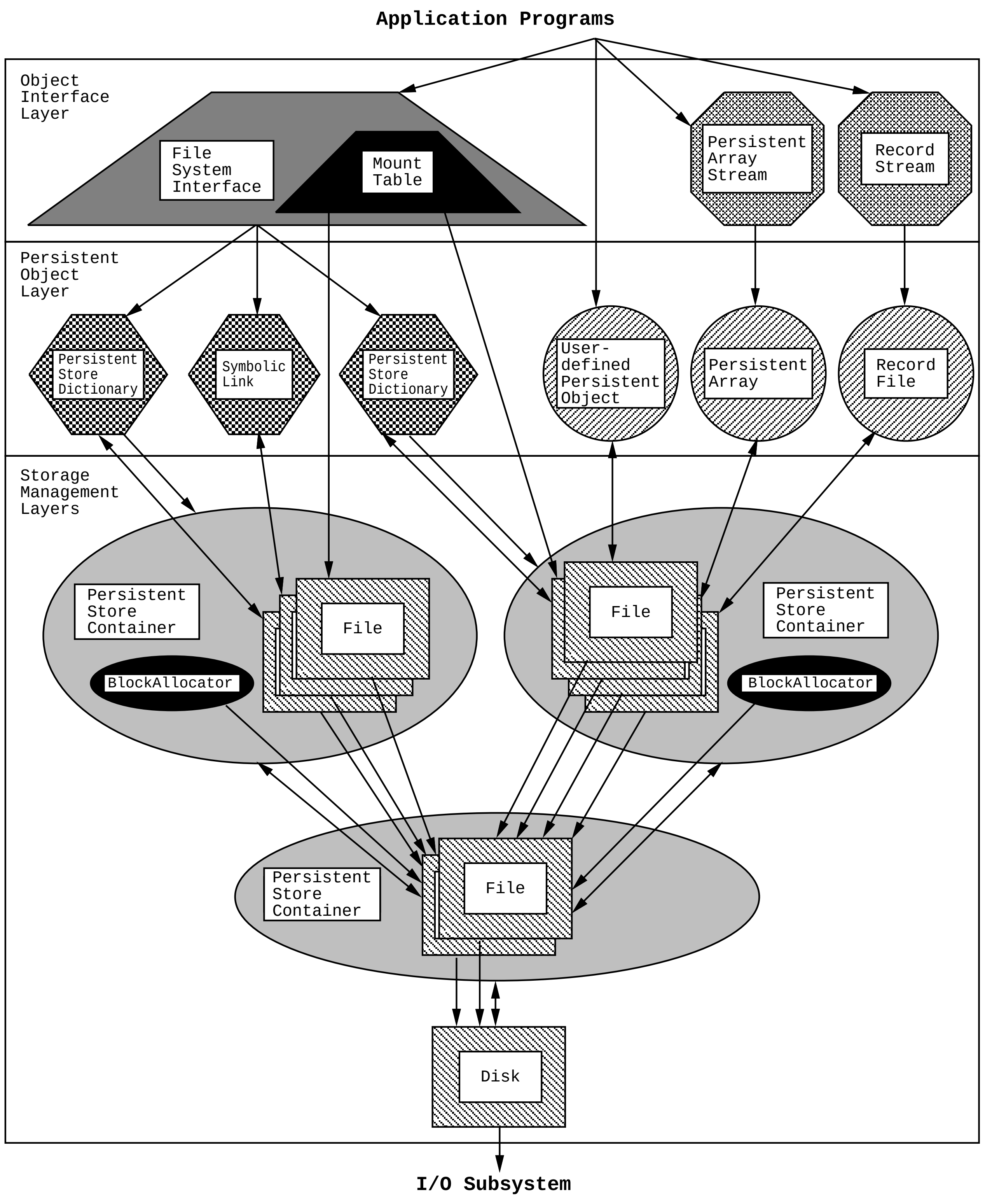

Figure 5.3: Abstract File System Framework

- rectangles represent physical and logical storage devices (either Disks or Files which are kinds of PersistentStores),
- ellipses represent objects that organize the sharing of both physical and logical storage devices (PersistentStoreContainers and BlockAllocators),
- hexagons represent persistent objects that name persistent stores and their corresponding persistent objects (PersistentStoreDictionaries and SymbolicLinks),
- circles represent persistent objects that structure the data within files (PersistentArrays, RecordFiles, and user-defined PersistentObjects),
- trapezoids represent interface objects that provide naming protocols (FileSystemInterfaces and MountTables), and
- octagons represent interface objects that provide external data access protocols for persistent objects (RecordStreams and PersistentArrayStreams).

A reference from one object to another is shown as a line with an arrow pointing towards the object being referenced. Each PersistentStore refers to its corresponding PersistentObject and vice versa, these relationships are shown as lines with arrows pointing towards both objects.

The I/O subsystem of the computer underlies the framework. One or more layers of PersistentStores fit above the I/O hardware. These stores can include various kinds of disks, disk partitions, and files. PersistentStoreContainers divide disks into sets of partitions or sets of files, and they also divide partitions into sets of files. Therefore, all but the lowest level PersistentStores are contained within a PersistentStoreContainer.

If a PersistentStoreContainer allows Files to be created and deleted or if it contains variable-sized Files, it uses a BlockAllocator to manage the data blocks of the next lower layer's PersistentStore. For example, a container of fixed-size disk partitions does not need a BlockAllocator, but a container of UNIX or MS-DOS files does.

Some Files in the highest Storage Management Layer support naming objects, while other Files support various types of RecordFiles, PersistentArrays, and user-defined PersistentObjects. The naming objects can be either directories or, if it is a BSD UNIX file system, symbolic links. RecordFiles support record-oriented file access, PersistentArrays support byte-stream-oriented file access, and user-defined PersistentObjects support object-oriented data access.

Application programs can refer directly to objects in the Persistent Object Layer, or they can use interface objects like FileSystemInterfaces and RecordStreams. For UNIX file systems, a MountTable enables the FileSystemInterface to unify the name spaces of separate Persistent-StoreContainers.

The interfaces of objects in this framework describe general *types* of objects. Systems programmer can build concrete classes that implement one of these interfaces. An object in the framework invokes operations on another object based on the type of the other object, not on its specific concrete class. Thus all operations are polymorphic, since they can be applied to various classes of objects. Since the framework is implemented in C++, types are represented by abstract classes. Figure 5.4 shows the top levels of the class hierarchy; more of the hierarchy is shown when the *italicized* classes are introduced.

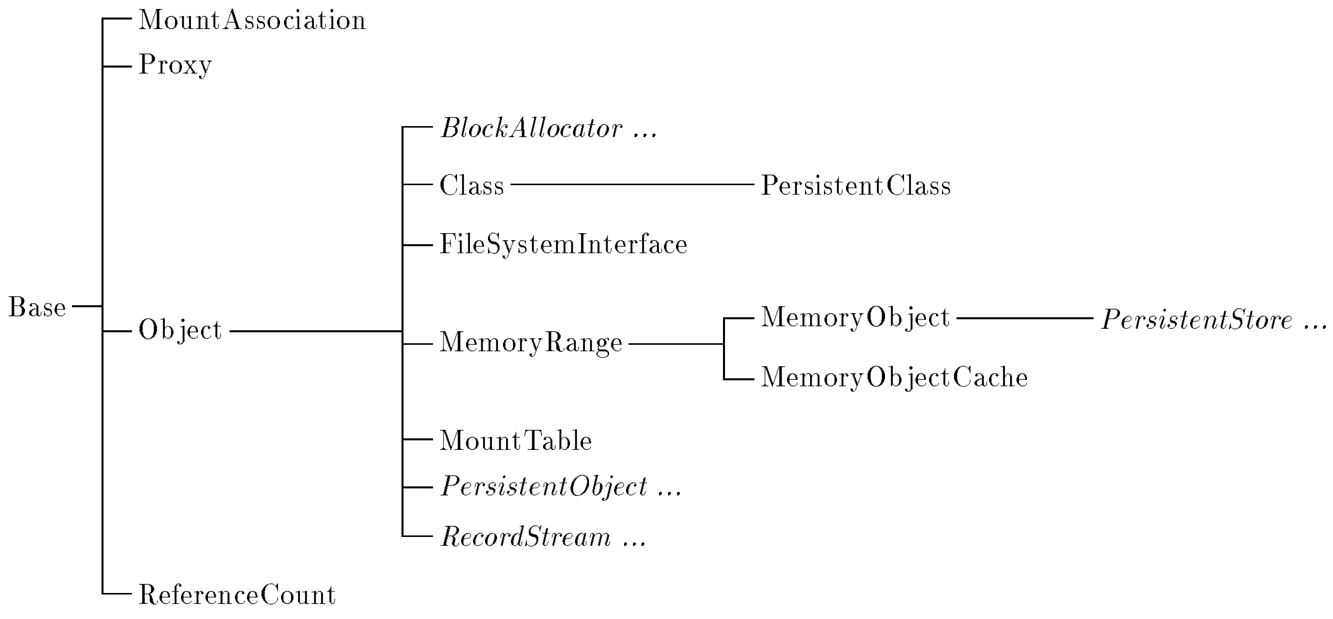

Figure 5.4: File System Class Hierarchy

5.2 Components of the Framework

This section introduces the major classes of the file system class hierarchy, which define the components of the framework. Most of the classes introduced in this section are abstract; a few classes, however, are concrete because they are general enough to satisfy the requirements of all file systems built using them.

5.2.1 Storage Devices

Regardless of how persistent data is organized, named, and structured, they must ultimately be stored on some physical persistent storage device. Because physical and logical storage devices share the same interface, they also share a single abstraction. The abstract PersistentStore class defines an access protocol for both physical storage devices (disks) and logical storage devices (files); each device is modeled as a sequence of identically sized blocks of data.

In the *Choices* virtual memory system[RC89], the MemoryObject class abstracts logical segments of memory. As a subclass of MemoryObject, PersistentStore abstracts both disks and files by adding the operations and state data described below. Figure 5.5 contains a simplified version of the C++ class declarations for PersistentStore, for MemoryObject, and for MemoryRange, which is the superclass of MemoryObject.[3]

Each PersistentStore inherits from class MemoryRange member variables that contain the number of data blocks and the size of each block. For efficiency, block sizes must be integer powers of two; therefore, block sizes are stored as $log_2(blockSize)$. Each PersistentStore has an id number (see Section 5.2.4), a reference to a PersistentObject that encapsulates its data, and, if it is mapped into virtual memory, a reference to a MemoryObjectCache (see Section 5.2.5).

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. Objects that communicate with PersistentStores using these operations must supply a starting block number, the number of contiguous blocks, and a buffer address[RC89].[4] Both operations return the number of blocks successfully read or written. The *Choices* virtual memory and file systems can both use subclasses of PersistentStores because they inherit the read/write interface from the MemoryObject class. The default implementations of read and write perform no data transfer and thus return zero; therefore, subclasses must implement one or both of these operations.

[3]All functions defined by all abstract classes in this framework are virtual functions; therefore, the `virtual` keyword has been elided from each function definition.

[4]To simplify the discussion of the read and write operations, buffer addresses will be given the type "`char *`" throughout this thesis. See [Rus91] for a detailed discussion of their actual type.

```
class MemoryRange : public Object {
public:
    MemoryRange( int numberOfBlocks, int log2BlockSize );
    int blockSize();
    int log2BlockSize();
    int numberOfBlocks();

protected:
    int _numberOfBlocks;
    int _log2BlockSize;
}

class MemoryObject : public MemoryRange {
public:
    MemoryObject( int numberOfBlocks, int log2BlockSize );
    int       read(  int start, int count, char * buffer );
    int       write( int start, int count, char * buffer );
    ErrorCode copy( MemoryObject * destination );
    int       setNumberOfBlocks( int );

protected:
    MemoryObjectCache * _cache;
}

class PersistentStore : public MemoryObject {
public:
    PersistentStore( int numberOfBlocks, int log2BlockSize, int idNumber );

    PersistentObject * asA( Class *, ErrorCode & );
    Class *  supports( Class * );
    void     close( PersistentObject * );

    int  recordSize();
    int  setRecordSize( int );
    int  numberOfRecords();
    int  setNumberOfRecords( int );
    int  idNumber();
    void info( FileInfo * )

protected:
    int                _idNumber;
    int                _numberOfRecords;
    PersistentObject * _persistentObject;
    Class *            _supports;
};
```

Figure 5.5: Class Declaration: PersistentStore

The copy operation on a store takes a destination store as an argument, reads each of the store's blocks, and writes the blocks to the destination store. Figure A.2 gives the general algorithm for copying PersistentStores; subclasses can overload the operation to improve its performance. The copy operation first calculates an appropriate buffer size for both reading and writing data. It then iterates over each set of blocks, reading them and writing them. Finally, it sets the size of the destination store, and returns an error code if the operation is not successful.

PersistentStores provide read/write access to raw data, but often a different interface is needed to satisfy the requirements of the clients of the file system. Some examples include: a container, which treats the data as a collection of files; a dictionary, which treats the data as a collection of file names; and a record file, which treats the data as a collection of records. Subclasses of PersistentObject define these and other customized interfaces to a PersistentStore's raw data. The PersistentObject class and its abstract subclasses provide operations that control the activation and deactivation of persistent objects, how these objects are mapped into memory, and how they are garbage collected. 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 operation returns a reference to the store's PersistentObject. If the PersistentObject has not yet been instantiated, the operation instantiates the object by mapping the store's persistent data into memory. The PersistentObject encapsulates this data as its state data. The PersistentStore thus provides the underlying data for its associated PersistentObject. PersistentObjects and their underlying PersistentStores provide the foundation for the *Choices* file system framework.

PersistentObjects and their underlying PersistentStores implement *object-oriented* access to persistent data. The asA operation takes an argument that may be either a concrete or an abstract Class[5] and returns a reference either to an instance of the argument or an instance of a concrete subclass of the argument, respectively. Figure A.1 gives the algorithm for the asA operation. The asA operation relies on the supports operation to perform the following steps:

[5]See [MCK91] for a description of first-class classes in *Choices*.

1. determine if the stored data structure is compatible with the requested Class or any of its subclasses, and

2. if the requested Class is compatible with the stored data structure:

   (a) return the requested Class if it is concrete,

   (b) otherwise, return the appropriate concrete subclass.

3. if the requested Class is incompatible with the stored data structure, return zero.

Several file system clients may access the same persistent data. To provide data consistency for concurrent updates to persistent data through the operations of a persistent object, the PersistentStore ensures that there is, at maximum, only one instance of its associated PersistentObject. A PersistentStore uses its instance variable, `_persistentObject`, to refer to its corresponding 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 to inform the PersistentStore that it is also no longer needed in primary memory. A further asA request will instantiate a new PersistentObject that uses the existing persistent data.

PersistentStores also provide operations to report the size of their blocks and records and to report and set their length in both blocks and records. Block and record sizes are given as numbers of bytes. In general, records may span blocks.

The concept of a PersistentStore is used both for physical and logical storage devices, allowing reuse of code. All concrete subclasses of PersistentStore, shown in Figure 6.1, belong to one of two categories represented by the following 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. Ultimately, the data read from and written to a File is also read from and written to a Disk.

5.2.1.1 Disks

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. Figure 5.6 contains a simplified version of the C++ declaration for class Disk. The read and write operations, shown in Figures A.3 and A.4, both convert data transfer requests into messages sent to the disk's controller using the doio operation. The classes of PersistentObjects associated with Disks structure the data on the disks into collections of partitions or collections of files. Disks are discussed in detail in [Kou91].

```
class Disk : public PersistentStore {
public:
    Disk( int numberOfBlocks, int log2BlockSize, int idNumber, int retries );

    int read(  int start, int count, char * buffer );
    int write( int start, int count, char * buffer );

    int partitions();

protected:
    int         _retries;
    Semaphore * _mutex;

    int doio( int start, int count, char * buffer, int operationCode );
};
```

Figure 5.6: Class Declaration: Disk

5.2.1.2 Files

A File is a PersistentStore that is contained within a PersistentStoreContainer (see Section 5.2.3). Instead of containing hardware interface code like Disks, it has a source PersistentStore that supplies it with data from a lower layer of the file system. For example, the BSDInodes in Figure 5.2 get their data from a Partition, which gets its data from a SCSIDisk. Both the BSDInodes and the Partition are Files. Figure 5.7 contains a simplified version of the C++ declaration for class File.

```
class File : public PersistentStore {
public:
    File( int numberOfBlocks, int log2BlockSize, int idNumber,
          PersistentStoreContainer *, PersistentStore * source,
          int offset = 0, Class * = 0 );

    int read(  int start, int count, char * buffer );
    int write( int start, int count, char * buffer );

    PersistentStoreContainer * container();

protected:
    PersistentStore *          _source;
    PersistentStoreContainer * _container;

    int _offset;
    int _log2BlockFactor;
    int _isBuffered;

    int basicRead(  int start, int count, char * buffer );
    int basicWrite( int start, int count, char * buffer );

    int blockFactor();
};
```

Figure 5.7: Class Declaration: File

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. Files that provide a contiguous window on their source use their `_offset` variable to hold the number of the first block of the window. Since many file systems support the *clustering* of disk sectors, a File uses a clustering factor (`_log2BlockFactor`) to convert between the block sizes of adjacent file system layers. If a file has a smaller block size than its source store, it must buffer reads and writes of the store.

The read and write operations of a File, shown in Figures A.5 and A.6, are forwarded to its source PersistentStore after the arguments are adjusted. The adjustment of arguments is performed by the basicRead and basicWrite operations, shown in Figures A.7 and A.8. Both operations handle two cases of relationships between the block sizes of the File and its source. If the File has a smaller block size, the blocks must be buffered and the data copied between the memory address given as the third parameter to either read or write and the buffer used to read or write from the source. If the block size of the File is greater than or equal to the source's, no buffering is needed. Instead, basicRead and basicWrite both multiply the starting block number and block count by the clustering factor. The block count returned by the source's read or write is adjusted by the clustering factor before being returned to the function that invoked the operation. Besides being multiplied by the clustering factor, the starting block number must also be mapped to its source's block number. Files that provide a contiguous window on their source can inherit the read and write operations from the File class. Other subclasses must calculate the block number argument using an appropriate function that they define to map the File's logical block numbers to its source's block numbers. See Chapter 6 for more information on subclassing Files.

5.2.2 Persistent Objects

Though PersistentStores provide an abstract interface to data storage devices, they do not provide the high level interfaces required by most application programs. For example, they do not allow programs to access files by name, nor do they support sequential access to individual bytes or records. PersistentObjects encapsulate the data made accessible by PersistentStores and provide a set of operations on that data. Subtypes of PersistentObject serve several purposes in the file system, including organization, naming, and file data structuring. These subtypes will be described in the following sections, after the PersistentObject class is presented.

A PersistentObject provides an encapsulation its underlying PersistentStore; therefore, it allows its clients to access the data of the PersistentStore only through the encapsulating interface. Nevertheless, a PersistentObject does allow its clients to access some attributes of its PersistentStore, by forwarding several operations to it. These operations include idNumber, numberOfRecords, recordSize, and info. Though it forwards some operations, does *not* forward read and write. Instead, a PersistentObject must define its own operations, which read and write the data of its PersistentStore. Sometimes these new operations leave most of the control to the client. For example, a PersistentArray treats the PersistentStore as an array, providing operations that clients can use to load and store elements. On the other hand, sometimes the interface is so different from that of the PersistentStore that it is hard to tell that a PersistentStore is involved. For example, a PersistentStoreContainer provide create, open, and close. All of these operations either read, write, both read and write the underlying PersistentStore.

All subclasses of PersistentObject inherit several of its operations. Figure 5.8 contains a simplified version of the C++ declaration for class PersistentObject. Each PersistentObject has at least two instance variables, one references its underlying PersistentStore, and the other is a flag that indicates whether the object has been modified since it was read from the PersistentStore. Figure 5.9 shows the top level of the PersistentObject class hierarchy; lower levels will be shown in the sections in which these classes are introduced.

The init operation provides a means to initialize a PersistentObject. Usually a C++ program uses a constructor to initialize an object, but PersistentObjects have their constructors called each time their data are retrieved from their PersistentStore. The init operation uses the UNIX-styled argument-count and argument-vector to give the function maximum flexibility. The noRemainingReferences operation is called only by the reference-counting code inherited from class Object. See [MCK91] for a description of reference-counting in *Choices*. This operation first calls the objects flush operation and then invokes close on its store to inform the store that it is no longer needed in primary memory. The flush operation converts all of an object's references to other PersistentObjects to their persistent format.

PersistentObject provides two operations to give applications control over the garbage collection of its various subclasses. The persist operation adds the object to the set of root objects

```
class PersistentObject : public Object {
public:
    PersistentObject( PersistentStore * source );

    int  init( int argc, char *argv[], FileSystemInterface * = 0 );
    void flush();

    int  persist();
    int  desist();

    PersistentObject * asA( Class *, ErrorCode & );
    int  copy( PersistentObject * );

    int  recordSize();
    int  numberOfRecords();

    void info( FileInfo * );
    int  idNumber();

protected:
    char               _isModified;
    PersistentStore *  _source;

    void noRemainingReferences();
};
```

Figure 5.8: Class Declaration: PersistentObject

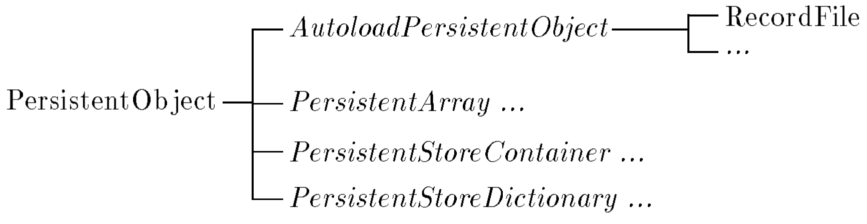
PersistentObject
AutoloadPersistentObject
RecordFile
...
PersistentArray ...
PersistentStoreContainer ...
PersistentStoreDictionary ...

Figure 5.9: Persistent Object Class Hierarchy

if it is not already in the set, and the desist operation removes the object from the set of root objects if it is in the set.

PersistentObjects respond to several operations by forwarding them to their underlying PersistentStores. These forwarded operations allow other objects to query a PersistentObject about the attributes of its underlying PersistentStore. They include:

- asA, which returns a reference to an object of the kind of the given Class if this PersistentObject's underlying store supports the Class,[6]
- copy, which copies the data from this PersistentObject's store to the underlying store of the given PersistentObject,
- recordSize, which returns the size of records stored if the data has a record structure or returns "one" if the data has no record structure,
- numberOfRecords, which returns the number of records stored if the data has a record structure or returns the number of bytes stored if the data has no record structure,
- info, which returns all attributes associated with the object, and
- idNumber, which returns the logical name of the object (see Section 5.2.4).

5.2.3 Storage Device Organization and Sharing

The data in a PersistentStore can be interpreted as a collection of Files. This organization makes it easier for users to share a device and enables users to store several logical collections of data on a single physical device.

5.2.3.1 PersistentStoreContainers

The abstract subclass of PersistentObject that supports the organization and sharing of storage devices is called PersistentStoreContainer. A PersistentStoreContainer divides the contents of a PersistentStore into an indexed collection of Files. It creates, makes accessible, and deletes these Files. PersistentStoreContainers can be nested to an arbitrary depth; this supports the multiple

[6]Usually the asA operation on PersistentObjects returns either the object itself or zero, since there is a one-to-one correspondence between the PersistentObject and its underlying PersistentStore.

Storage Management Layers of the framework (see Figure 5.3). The PersistentStoreContainer in the lowest layer divides a disk into several partitions, and the PersistentStoreContainer in the next layer subdivides partitions into logical storage for various types of files.

```
class PersistentStoreContainer : public PersistentObject {
public:
    PersistentStoreContainer( PersistentStore * source );

    File * open( int id, ErrorCode & );
    File * create( int id, Class *, ErrorCode & );
    void   close( File * );

    PersistentStoreDictionary * rootDictionary();
    void synchronize();

protected:
    Semaphore *      _mutex;
    BlockAllocator * _allocator;
    File **          _files;

    int    _numberOfFiles;
    int    _openFiles;

    File * basicOpen( int idNumber, ErrorCode & );
    File * basicCreate( int idNumber, Class *, ErrorCode & );
    void   basicClose( File * mo );

    int    basicRootId();
    void   basicSynchronize();
};
```

Figure 5.10: Class Declaration: PersistentStoreContainer

Figure 5.10 contains a simplified version of the C++ declaration for class PersistentStore-Container. A PersistentStoreContainer has a reference to an underlying PersistentStore and the `_modified` flag (both inherited from PersistentObject), a Semaphore that supports mutual exclusion, a BlockAllocator(see below), a table of references to all Files that are currently open, the total number of Files, and the number of Files currently open.

The major operations supported by PersistentStoreContainers are create, open, and close. The create operation, shown in Figure A.10, returns a newly created File. The open operation, shown in Figure A.9, takes a container-index as an argument and returns the corresponding File. The close operation, shown in Figure A.12, informs a PersistentStoreContainer that a currently open File is no longer being used by any other object in the system.

To support naming of contained objects, each PersistentStoreContainer has a rootDictionary operation, shown in Figure A.11, that returns a PersistentStoreDictionary (see Section 5.2.4) from which the contained Files can be reached. The synchronize operation, shown in Figure A.13, writes out all of a container's data structures that had been read from its source PersistentStore and then modified.

All five public operations allow subclasses to inherit and share mutual exclusion and open file table manipulation code. These operations are defined using the protected operations that subclasses must implement (see Chapter 6).

5.2.3.2 BlockAllocator

While a PersistentStoreContainer subdivides a PersistentStore, a BlockAllocator manages the allocation of its data blocks. In particular, it keeps track of which blocks are currently allocated and which blocks are free. Figure 5.11 contains a simplified version of the C++ declaration for class BlockAllocator. Subclasses encapsulate various mechanisms to manage block allocation, including free-lists or bit-maps. A BlockAllocator has a reference to its container's Persistent-Store, which is used to retrieve and store information about which blocks are allocated and free, and a Semaphore that supports mutual exclusion.

A BlockAllocator is not a PersistentObject, instead it is a component of a PersistentStore-Container. BlockAllocator is a separate class because it allows code reuse, since two or more subclasses of PersistentStoreContainer can use the same subclass of BlockAllocator.

Files whose size can change, e.g. those that represent variable-length files, use the allocate and free operations of BlockAllocators to request and release the blocks of storage. Allocate reserves a block of storage and returns its index, and free releases a block of storage that is no longer needed. Both public operations, shown in Figures A.14 and A.15, allow subclasses to inherit and share mutual exclusion code. These operations are defined using the protected operations that subclasses must implement (see Chapter 6).

```
class BlockAllocator : public Object {
public:
    BlockAllocator( PersistentStore * source );

    int  allocate( int & blocks, int hint, int numberOfBlocks );
    void free( int & blocks, int blockNumber, int numberOfBlocks );

protected:
    PersistentStore * _source;
    Semaphore *       _mutex;

    int  basicAllocate( int & blocks, int hint, int numberOfBlocks );
    void basicFree( int & blocks, int blockNumber, int numberOfBlocks );
};
```

Figure 5.11: Class Declaration: BlockAllocator

5.2.4 Naming

PersistentStores and PersistentObjects have both *logical* and *symbolic* names. The logical names of PersistentStores and PersistentObjects are unique and 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 shares the identifier of its underlying PersistentStore.[7]

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

[7]A PersistentObject can share the identifier of its underlying PersistentStore because there is a one-to-one correspondence between them.

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, a single PersistentStore can have many symbolic names.

5.2.4.1 PersistentStoreDictionaries

PersistentStores can be given symbolic names by and grouped into 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 name. An example of a PersistentStoreDictionary is a System V UNIX directory, which maps fixed-length symbolic keys to logical names called *inumbers*.

Using symbolic names, PersistentStores can be opened from, created in, added to, and removed from PersistentStoreDictionaries. Figure 5.12 contains a simplified version of the C++ declaration for class PersistentStoreDictionary.

A PersistentStoreDictionary has a reference to its underlying PersistentStore (inherited from PersistentObject), a Semaphore that supports mutual exclusion, a reference to a PersistentStore-Container that contains the Files to which its keys refer, and several variables that it uses to communicate with concrete subclasses.

The open operation, shown in Figure A.20, takes a key as an argument and returns the named PersistentStore, if the key is found. It obtains the PersistentStore by invoking the open operation on its PersistentStoreContainer using the id-number that corresponds to the key.

Two operations, create and add, shown in Figures A.18 and A.19, allow PersistentStores to be added to dictionaries. The create operation performs the same function as open for existing keys, except that it also checks to ensure that the store supports the given Class of PersistentObject. If the key does not exist, however, the operation creates and returns a new PersistentStore. The add operation takes a symbolic key and a PersistentStore as arguments. It then invokes the PersistentStore's idNumber operation and inserts the key and id-number into the dictionary.

The remove operation, shown in Figure A.21, deletes a mapping from a key to an id-number. The keys operation, shown in Figure A.17, returns the set of keys that is stored in the dictionary.

```
class PersistentStoreDictionary : public PersistentdObject {
public:
    PersistentStoreDictionary( PersistentStore * source );

    File * open( char * key, ErrorCode & );
    File * create( char * key, Class *, ErrorCode & );

    ErrorCode add( char * key, File * );
    ErrorCode remove( char * key );

    ErrorCode keys( char * buffer, int bufferSize, int & start );
    ErrorCode associations( char * buffer, int bufferSize, int & start );

protected:
    PersistentStoreContainer * _container;
    Semaphore *                _mutex;

    char * _buffer;
    int _size;

    int _idNumberOfKey;
    int _offset;
    int _emptyOffset;
    int _maxKeyLength;

    ErrorCode basicKeys( char * buffer, int bufferSize, int & start );
    ErrorCode basicAssociations( char * buffer, int bufferSize, int & start );

    int findKey( char * key, int neededSize );

    int insertAssociation( char * key, int keyLength, int idNumber );
    int clearAssociation();

    int associationSize( int keyLength );
    int isEmpty();
};
```

Figure 5.12: Class Declaration: PersistentStoreDictionary

The associations operation, shown in Figure A.16, returns the set of mappings from keys to id-numbers that is stored in the dictionary.

All six public operations allow subclasses to share not only mutual exclusion code, but also much of the algorithms for open, create, add, and remove. These operations are defined using the protected operations that subclasses must implement (see Chapter 6).

5.2.4.2 SymbolicLinks

The SymbolicLink class implements the BSD UNIX concept of a symbolic link, i.e. a mapping from a logical name to a pathname. Figure 5.13 contains a simplified version of the C++ declaration for class SymbolicLink. SymbolicLinks provide the at and atPut operations, both inherited from PersistentArray (see Figure 5.15), to retrieve and store their pathname.

```
class SymbolicLink : public PersistentCharArray {
public:
        SymbolicLink( PersistentStore * mo );
};
```

Figure 5.13: Class Declaration: SymbolicLink

The SymbolicLink class adds neither state nor operations to the definition of class Persistent-Array, nor does it redefine any inherited operations. The class is, nevertheless, important, since FileSystemInterfaces check each PersistentStore accessed during pathname parsing (see below) to see if it stores the data for a SymbolicLink. This check is performed by invoking the asA operation with the SymbolicLink class as an argument. In this case, the asA operation will succeed if and only if the store supports the SymbolicLink class.

5.2.5 File Data Structures

The framework incorporates three models for structuring the data accessed by applications:

- files that are structured as arrays of bytes or words,
- files that are structured as collections of records, and
- files that are data structures encapsulated by persistent objects.

In Chapter 2 these three methods of file access are called stream-oriented, record-oriented, and object-oriented, respectively.

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 on persistent data. Programs can cast the data that they read into whatever structures are needed. 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 in which the program that created them was written. The third model fits programming languages like C++ and object-oriented operating systems like *Choices*. The file system presents data as objects that are instances of user-defined subclasses of PersistentObject.

As with objects in the Storage Management Layer (Disks and Files), file data structuring objects either belong to primitive building-block classes or to composite classes that combine primitive objects to build more complex objects.

All classes of PersistentObjects discussed so far in this chapter are designed to abstract the internal components of a file system. The PersistentObjects discussed in this section encapsulate the data in PersistentStores by buffering the blocks of data and providing an interface that corresponds to objects commonly manipulated by application programs. Data can be buffered either by allocating buffers and transferring data between PersistentStores and the buffers or by mapping the PersistentStore into virtual memory.

Subclasses of the MemoryObjectCache[8] class provide virtual memory data caching for the file system[RC89, MCRL89, Rus91]. They can map any MemoryObject into virtual memory. By mapping a disk or a disk partition, a MemoryObjectCache effectively serves as a disk buffer cache. By mapping other PersistentStores, a MemoryObjectCache supports memory-mapped files or memory-mapped persistent objects.

Each memory-mapped PersistentStore has a single MemoryObjectCache that maintains physical memory management information associated with virtual memory. The information is kept in a machine-independent and virtual memory address independent form. This allows a PersistentStore to be mapped into multiple regions of a virtual address space.

[8]The design of MemoryObjectCaches is primarily the work of other members of the *Choices* project, including Roy Campbell, Gary Johnston, Ken MacGregor, Vincent Russo, and Aamod Sane.

MemoryObjectCaches also support distributed virtual memory[JC89]. They can be customized to support various page placement, page replacement, consistency, and coherency policies.

5.2.5.1 PersistentArrays

The PersistentArray subclass of PersistentObject abstracts various kinds of arrays of persistent data, which are shown in Figure 5.14. All subclasses of PersistentArray define two operations, at and atPut, which retrieve and store elements of the array. These subclasses serve two purposes in the framework: they structure the data in a file as an array of bytes or words, and they abstract the building-blocks of record-oriented files. Because they define arrays with either different element-types or index-types, they define different signatures (sets of argument types) for both operations; therefore, there is no benefit in defining the operations in their superclass. All subclasses inherit all their state variables from PersistentObject and PersistentArray. They also inherit buffer management operations from PersistentArray. Figure 5.15 contains a simplified version of the C++ declaration for class PersistentArray.

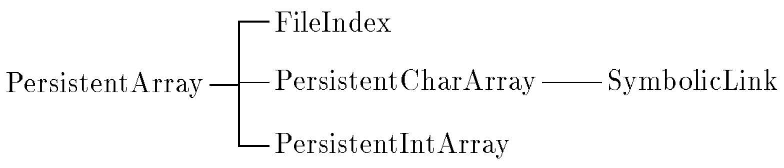

Figure 5.14: Persistent Array Class Hierarchy

A PersistentArray has a pointer to the data in the array. If the array is mapped into virtual memory, then the pointer stores the address where the data are mapped; otherwise, the pointer stores the address of a buffer allocated for the array. The size operation returns the number of elements currently in the array. The setSize operation sets the number of elements in the array to the value of its argument. If the new size is smaller than the current size, it truncates the array. The grow operation performs the following steps if and only if the array is not mapped into virtual memory: it allocates a new buffer large enough to store an array of the given size, copies the array's existing data to the buffer, and then frees the old buffer. The basicAt and basicAtPut operations support the at and atPut operations of subclasses by hiding the details

```
class PersistentArray : public PersistentdObject {
public:
    PersistentArray( PersistentStore * source );

    unsigned int size();
    unsigned int setSize();

protected:
    char * _arrayBuffer;

    void grow( int units );

    int basicAt(    unsigned int index, char * values, unsigned int count );
    int basicAtPut( unsigned int index, char * values, unsigned int count );
};
```

Figure 5.15: Class Declaration: PersistentArray

of buffer management. For example, the basicAtPut will invoke the grow operation if the index of the data to be stored exceeds the upper bound of the array.

5.2.5.2 PersistentCharArrays

The PersistentCharArray class abstracts files structured as arrays of bytes, for example, the regular data files of the UNIX operating system. Figure 5.16 contains a simplified version of the C++ declaration for class PersistentCharArray. PersistentCharArray defines at and atPut to retrieve and store consecutive elements of an array with a character element type and an integer index type. Both operations return the number of elements transferred.

```
class PersistentCharArray : public PersistentArray {
public:
    PersistentCharArray( PersistentStore * source );

    int at(    unsigned int index, char * values, unsigned int count );
    int atPut( unsigned int index, char * values, unsigned int count );
};
```

Figure 5.16: Class Declaration: PersistentCharArray

5.2.5.3 PersistentIntArrays

The PersistentIntArray class abstracts files structured as arrays of integers. Applications can use PersistentIntArrays for data files, but this class is primarily designed as a building-block for other file types. Figure 5.17 contains a simplified version of the C++ declaration for class PersistentIntArray.

```
class PersistentIntArray : public PersistentArray {
public:
    PersistentIntArray( PersistentStore * source );

    int init( int argc, char *argv[], FileSystemInterface * = 0 );

    int at(    unsigned int index );
    int atPut( unsigned int index, int value );
};
```

Figure 5.17: Class Declaration: PersistentIntArray

PersistentIntArray defines at and atPut to retrieve and store a single element of an array with an integer element type and an integer index type. The at operation returns the stored integer or zero if the index is out of range. The atPut operation also returns the stored integer. The init operation allows programs to specify the size of each element (either 1, 2, or 4 bytes). The default is the size of an integer in the C++ programming language.

5.2.5.4 RecordFiles

Traditionally the data in files are structured as collections of records. The RecordFile class abstracts files that are structured in this way. The concrete subclasses of RecordFile presented in Section 5.3 illustrate how PersistentArrays can be used as building blocks for more complex file structures. Figure 5.18 contains a simplified version of the C++ declaration for class RecordFile, which defines the interface for its subclasses.

The read and write operations allow single records to be retrieved from and stored to the file. The numberOfRecords operation returns the number of the file's last record. The minimumRecordSize and maximumRecordSize operations return the size of the smallest and largest records in the file, respectively.

```
class RecordFile : public AutoloadPersistentObject {
public:
    RecordFile( PersistentStore * source );

    void flush();

    int read(  unsigned int record, char * buffer,
               unsigned int size, ErrorCode & );
    int write( unsigned int record, char * buffer,
               unsigned int size, ErrorCode & );

    int numberOfRecords();

    int minimumRecordSize();
    int maximumRecordSize();

protected:
    PersistentCharArray * _data;

    unsigned int _minRecordSize;
    unsigned int _maxRecordSize;
};
```

Figure 5.18: Class Declaration: RecordFile

A RecordFile has a reference to a PersistentCharArray that encapsulates the unstructured data of the file, and variables to store the smallest and largest record sizes. The flush operation ensures that the reference to the PersistentCharArray is in a format suitable for persistent storage. The flush operation is normally invoked only when a PersistentObject is deactivated and its data are written back to its PersistentStore.

5.2.5.5 FileIndexes

The FileIndex class abstracts indices that map key values to record numbers. This class serves as a building-block for IndexedRecordFiles, which are presented in Section 5.3. Figure 5.19 contains a simplified version of the C++ declaration for class FileIndex.

```
class FileIndex : public PersistentArray {
public:
    FileIndex( PersistentStore * source );
    int init( int argc, char *argv[], FileSystemInterface * = 0 );

    int at(       char * key, ErrorCode & );
    int atPut(    char * key, int value, ErrorCode & );
    int atNext(   char * key, ErrorCode & );
    int atDelete( char * key, ErrorCode & );

protected:
    int findKey(   char * key, ErrorCode & );
    int insertKey( char * key, ErrorCode & );
};
```

Figure 5.19: Class Declaration: FileIndex

FileIndex defines at, atNext, and atPut to retrieve and store a single element of an array with an integer element type and a string index-type. The at operation returns the stored integer if the index is found; otherwise, an error code is returned. The atNext operation returns the stored integer if the index is found; otherwise, an error code is returned. It also returns the value of the next key. The atPut operation also returns the stored integer, if the key did not already exist; otherwise, an error code is returned. The atDelete operation removes the element of the array that corresponds to the key argument, if such an element exists; otherwise, it returns an error code. The init operation allows programs to specify the size of the keys.

The FileIndex class can be subclassed to give applications a choice of various algorithms to store and retrieve sets of keys efficiently. Currently the FileIndex class uses an insertion sort algorithm when storing keys and a binary search algorithm when retrieving keys. Subclasses can redefine findKey and insertKey operations to implement other algorithms.

5.2.5.6 User-defined Persistent Objects

Files structured as PersistentObjects differ from other 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 or at and atPut.
- their operations can store, retrieve, and dereference references to other persistent objects.
- invocation of an operation ensures that the encapsulated persistent data are 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 file name, or to be updated with the addition or removal of a file name.

Choices is not limited to a single application programming language. Currently, however, I have implemented persistence in *Choices* only for objects that are programmed using C++[Str86]. 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.

As long as system programmers do not violate the restrictions stated above, they can design the operations and structures of persistent objects as they would design other C++ objects. Nevertheless, there are a few operations that must be defined for concrete subclasses of PersistentObject; see Chapter 6 for information about these operations.

5.2.5.7 AutoloadPersistentObjects

Some kinds of PersistentObjects, for example PersistentStoreContainers, need complete control over how they cache their data or map it into memory. The classes that abstract these kinds of objects can define functions to manage their memory mapping; many subclasses of PersistentObject, however, do not need to define their own mapping functions. These subclasses can inherit memory mapping functions from the AutoloadPersistentObject class.

```
class AutoloadPersistentObject : public PersistentdObject {
public:
    AutoloadPersistentObject( PersistentStore * source );

protected:
    void noRemainingReferences();
};
```

Figure 5.20: Class Declaration: AutoloadPersistentObject

Figure 5.20 contains a simplified version of the C++ declaration for class AutoloadPersistentObject. Subclasses can inherit the noRemainingReferences operation, which automatically writes the persistent data of an object back to its PersistentStore when the object is no longer needed in primary memory. The PersistentStore::asA operation automatically loads the persistent data of an AutoloadPersistentObject into memory when the object is activated.

5.2.6 Application Interface Objects

The interfaces provided by the file-structuring PersistentObjects presented in the previous section are abstract enough to be used directly by application programs; but file systems commonly define an additional layer of abstraction between files and application programs.

For naming objects, the FileSystemInterface and MountTable classes provide this extra layer by organizing all other naming objects into a hierarchy of dictionaries. For regular files, subclasses of the RecordStream class, which are shown in Figure 5.31, provide a common application interface.

5.2.6.1 MountTables

A MountTable is a dynamic, associative, bidirectional function that maps PersistentStores called *mount-points* to PersistentStores called *mounted-stores*. Figure 5.22 contains a simplified version of the C++ declaration for class MountTable. A MountTable is implemented as a linked list of MountAssociations, which are ordered pairs of references to PersistentStores. Figure 5.21 contains a simplified version of the C++ declaration for class MountAssociation.

```
class MountAssociation : public Base {
public:
    MountAssociation( MountAssociation *, PersistentStore * mountPoint,
                                          PersistentStore * mountedStore );

    MountAssociation * lookupMounted( PersistentStore * );
    MountAssociation * lookupMountPoint( PersistentStore * );
    PersistentStore *  mountPoint();
    PersistentStore *  mounted();
    MountAssociation * next();

protected:
    MountAssociation * _next;
    MountAssociation * _previous;
    PersistentStore *  _mountPoint;
    PersistentStore *  _mounted;

    void setNext( MountAssociation * );
    void setPrevious( MountAssociation * );
};
```

Figure 5.21: Class Declaration: MountAssociation

MountTables provide the add operation to add a single mapping and the remove operation to remove a single mapping. The openMounted operation returns the mounted-store that corresponds to a given mount-point, and the openMountPoint operation returns the mount-point that corresponds to a given mounted-object. To make MountTables complete functions, all PersistentStores that are not explicitly mapped to other PersistentStores are implicitly mapped to themselves.

```
class MountTable : public Object {
public:
     MountTable();
     MountTable( MountTable * );

     int synchronize();

     int add( PersistentStore * mountPoint, PersistentStore * mountedStore );
     int remove( PersistentStore * );

     PersistentStore * openMounted( PersistentStore * );
     PersistentStore * openMountPoint( PersistentStore * );

protected:
      MountAssociations * _mountItems;
};
```

Figure 5.22: Class Declaration: MountTable

5.2.6.2 FileSystemInterface

A FileSystemInterface unifies the name-spaces provided by PersistentStoreDictionaries, SymbolicLinks, and MountTables by parsing sequences of symbolic keys, called *pathnames*, and resolving them to the logical names of PersistentObjects. Each symbolic name is interpreted sequentially by the instance of PersistentStoreDictionary specified by the pathname prefix composed of the previous symbolic names. A FileSystemInterface uses a MountTable to organize all dictionaries into a single tree. FileSystemInterfaces not only provide a unified naming interface for application programs, but they also ensure a consistent usage of features like mount tables and symbolic links. Furthermore, a FileSystemInterface can function as a name server by integrating the naming of devices or processes with the naming of files. For example, the UNIX file system contains device names in a directory called "/dev."

An example of a FileSystemInterface is the UNIX file system interface, which uses a root directory, a current directory, and a mount table to provide a unified name space for all files within a computer system. A FileSystemInterface that implements the BSD version of UNIX file naming would also use SymbolicLinks.

Figure 5.23 contains a simplified version of the C++ declaration for class FileSystemInterface. A FileSystemInterface has "root" and "current" dictionaries, a MountTable, and a Semaphore.

```
class FileSystemInterface : public Object {
public:
   FileSystemInterface( PersistentStoreDictionary * );
   FileSystemInterface( FileSystemInterface *, int deepCopyMountTable );

   Object *  open( char * path, Class * interfaceClass, ErrorCode &,
                       int flags = 0, int mode = 0, Class * objectClass = 0 );
   ErrorCode info( char *, FileInfo *, int followLinks );

   ErrorCode add( char * key, File * );
   ErrorCode link( char *, char *, int overwrite );
   ErrorCode unlink( char * );

   ErrorCode mount( char *, char * );
   ErrorCode unmount( char * );

   ErrorCode mkdir( char * );
   ErrorCode chdir( char * );
   ErrorCode chroot( char * );

   ErrorCode synchronize();

protected:
   PersistentStoreDictionary * _root;
   PersistentStoreDictionary * _current;
   MountTable *                _mountTable;
   Semaphore *                 _mutex;

   PersistentStoreDictionary * basicFindDictionary( char * path, char * & key,
                                           PersistentStoreDictionary *, int links );
   PersistentStoreDictionary * findDictionary( char * path, char * & key,
                                           PersistentStoreDictionary *, int links );

   PersistentStore * pathOpen( char * path, ErrorCode &,
                                     PersistentStoreDictionary *,
                                     int follow, int links );
   PersistentStore * pathCreate( char * path, Class *, ErrorCode & );

   PersistentStore * substituteLink( PersistentStoreDictionary *,
                                           PersistentStore *,
                                           ErrorCode &, int links );
   ErrorCode changeDictionary( PersistentStoreDictionary * &, char * path );
};
```

Figure 5.23: Class Declaration: FileSystemInterface

The Semaphore provides mutual exclusion for the operations that modify the other three state variables.

The public operations of the FileSystemInterface are similar to several UNIX system calls including:

- open, which returns a reference to the object that is named by the first argument and is an instance of the Class given by the second argument,
- info,[9] which returns status information about the object named by the first argument,
- link, which creates a symbolic name given in the second argument for the object named by the first argument,
- add,[10] which creates a symbolic name given in the first argument for the object referred to by the second argument,
- unlink, which deletes the given symbolic name for the object named by its argument,
- mount, which adds to the MountTable a MountAssociation referring to the object named by the first argument as the mount-point and the object named by the second argument as the mounted-store,
- unmount, which removes from the MountTable the MountAssociation that has the object named by the argument as the mounted-object,
- mkdir, which creates a directory and names it with the given symbolic name,
- chdir, which changes the current dictionary to the one named by the argument, and
- chroot, which changes the root dictionary to the one named by the argument.

These operations manipulate or return references to RecordStreams, PersistentObjects, or PersistentStores.

[9] The info operation corresponds to the `stat(2)` system call in the UNIX operating system.

[10] This add does not correspond to any system call in the UNIX operating system. It does, however, resemble the add operation of PersistentStoreDictionaries, except that it uses pathnames instead of single-element file names.

5.2.6.3 RecordStreams

RecordStreams provide the concept of a *current file position*, i.e. the location within the file where the next read or write will occur. Figure 5.24 contains a simplified version of the C++ declaration for class RecordStream. A RecordStream has a reference to a PersistentObject and a record number variable.

```
class RecordStream : public Object {
public:
        RecordStream( PersistentObject * po );
        int init( int argc, char *argv[] );

        int read(  char * buffer, int size, ErrorCode & );
        int write( char * buffer, int size, ErrorCode & );

        int recordNumber();
        int setRecordNumber( int delta, int mode );

        int numberOfRecords();
        int setNumberOfRecords( int count );
        int recordSize();
        int minimumRecordSize();
        int maximumRecordSize();

        void info( FileInfo * );

protected:
        int                _recordNumber;
        PersistentObject * _file;
};
```

Figure 5.24: Class Declaration: RecordStream

Because RecordStreams introduce the concept of a current file position, they support the setRecordNumber operation, which allows programs to reset the current position, and the recordNumber operation, which returns the current position. Application programs can read from and write to RecordStreams sequentially. The read and write operations also update the file position. Each instance of RecordStream gets data from or sends data to an underlying PersistentObject.

RecordStreams forward several operations directly to their PersistentObject, including: init, info, numberOfRecords, setNumberOfRecords, recordSize, minimumRecordSize, and maximumRecordSize.

5.2.6.4 PersistentArrayStreams

PersistentArrayStreams are RecordStreams that support the sequential reading and writing of PersistentArrays. Figure 5.25 contains a simplified version of the C++ declaration for class PersistentArrayStream.

```
class PersistentArrayStream : public RecordStream {
public:
    PersistentArrayStream( PersistentObject * source );

    int read(  char * buffer, int size, ErrorCode & );
    int write( char * buffer, int size, ErrorCode & );

protected:
    PersistentCharArray * _file;
};
```

Figure 5.25: Class Declaration: PersistentArrayStream

To support the UNIX concept of file streams, PersistentArrayStreams redefine the read and write so that they can transfer multiple consecutive bytes during a single invocation. Both operations return the number of bytes successfully transferred.

5.2.7 Protection

The file system provides three types of protection for persistent data:

1. all manipulation of persistent data must be performed by a PersistentObject,
2. access to a PersistentObject is restricted to processes executing within Domains that have established the right to access it, and
3. some PersistentObjects check permissions for each operation.

Processes use their Domain's list of Capabilities to access PersistentObjects. To be able to create a Capability to access a PersistentObject, the AuthenticationId[11] of the Domain must match one of the AuthenticationIds stored in the AccessList of each PersistentObject.

5.2.7.1 AuthenticationIds and AccessLists

The current implementation of my model simply uses the UNIX and MS-DOS protection models. In the UNIX model, PersistentStores have three access modes: read, write and execute, and they identify three sets of users: the owner, a group, and others. Nine flags specify which modes each set of users can enter. In the MS-DOS model, PersistentStores have two access modes: read and read-write, and they treat all users as the owners. A single flag specifies which modes users can enter.

5.2.7.2 Capabilities

In the *Choices* operating system, the capability mechanism is provided by the ObjectProxy class[Rus91]. ObjectProxies cannot be forged by applications and are protected by hardware, usually by the virtual memory hardware and the privileged instruction mode of the CPU.

5.3 Extensions to the Framework

The previous section describes the two fundamental classes of the framework (PersistentStore and PersistentObject), subclasses that support persistent data storage (Disk and File), subclasses that support storage organization, naming, and structuring (PersistentStoreContainer, PersistentStoreDictionary, PersistentArray, AutoloadPersistentObject and RecordFile), and application interface classes (FileSystemInterface and RecordStream). This section shows how the framework can be extended by building on primitive PersistentObjects such as subclasses of PersistentArrays. It describes how they can combined to build classes that structure data as collections of records. It also describes classes that provide interfaces for these extended types of persistent objects.

[11]An AuthenticationId identifies the user for whom processes within the Domain are executing.

5.3.1 File Data Structures

Several kinds of records and record files are presented in the discussion of the VMS operating system in Section 4.3. The framework currently contains four subclasses of the abstract RecordFile class, shown in Figure 5.26, that represent four combinations of record and file types: sequential fixed-length record files, sequential variable-length record files, relative fixed-length record files, and indexed fixed-length record files. These classes illustrate how the PersistentArrays can be used as building blocks for more complex file structures.

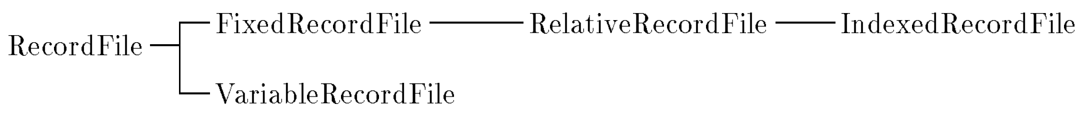

Figure 5.26: Record Structuring Class Hierarchy

5.3.1.1 FixedRecordFiles

The FixedRecordFile class abstracts sequential, fixed-length record files. Figure 5.27 contains a simplified version of the C++ declaration for class FixedRecordFile. The init operation allows programs to set the record size. The numberOfRecords operation returns the number of records stored in the file, which it calculates by dividing the size of the inherited PersistentCharArray by the recordSize.

The read operation retrieves the data that correspond to the record number argument, if the record exists (the record number is not past the end of the file); otherwise, it returns an error code. The write operation stores the data to be written that correspond to the record number argument, if the record number is at the end of the file; otherwise, it returns an error code.

5.3.1.2 VariableRecordFiles

The VariableRecordFile class abstracts sequential files with variable-length records. Figure 5.28 contains a simplified version of the C++ declaration for class VariableRecordFile. The init operation initializes the object's state, which consists of a PersistentIntArray that stores the offsets of

```
class FixedRecordFile : public RecordFile {
public:
    FixedRecordFile( PersistentStore * source );

    int init( int argc, char *argv[], FileSystemInterface * = 0 );

    int read(  unsigned int record, char * buffer,
               unsigned int size, ErrorCode & );
    int write( unsigned int record, char * buffer,
               unsigned int size, ErrorCode & );

    int numberOfRecords();
};
```

Figure 5.27: Class Declaration: FixedRecordFile

each record within the inherited PersistentCharArray. The numberOfRecords operation returns the number of records stored in the file, which is equal to the size of the PersistentIntArray.

The read operation retrieves the data that correspond to the record number argument, if the record exists (the record number is not past the end of the file); otherwise, it returns an error code. The write operation stores the data to be written that correspond to the record number argument, if the record number is at the end of the file; otherwise, it returns an error code.

5.3.1.3 RelativeRecordFiles

The RelativeRecordFile class specializes the FixedRecordFile class to abstract relative files. Figure 5.29 contains a simplified version of the C++ declaration for class RelativeRecordFile. The init operation initializes the object's state, which consists of a PersistentIntArray that stores the flags indicating whether each cell is full or empty. The PersistentIntArray is initialized to store 1-byte integers. Both full and empty cells are stored within the inherited PersistentCharArray. The numberOfRecords operation returns the number of cells in the file, which is equal to the size of the PersistentIntArray.

The read operation retrieves the data from the cell that corresponds to the record number argument, if the cell is not empty; otherwise, it returns an error code. The write operation stores the data to be written in the cell that corresponds to the record number argument, if

```
class VariableRecordFile : public RecordFile {
public:
    VariableRecordFile( PersistentStore * source );

    int init( int argc, char *argv[], FileSystemInterface * = 0 );
    void flush();

    int read(  unsigned int record, char * buffer,
               unsigned int size, ErrorCode & );
    int write( unsigned int record, char * buffer,
               unsigned int size, ErrorCode & );

    int numberOfRecords();

protected:
    PersistentIntArray * _control;
};
```

Figure 5.28: Class Declaration: VariableRecordFile

```
class RelativeRecordFile : public FixedRecordFile {
public:
    RelativeRecordFile( PersistentStore * source );

    int init( int argc, char *argv[], FileSystemInterface * = 0 );

    void flush();

    int read(  unsigned int record, char * buffer,
               unsigned int size, ErrorCode & );
    int write( unsigned int record, char * buffer,
               unsigned int size, ErrorCode & );

    int update(  unsigned int record, char * buffer,
                 unsigned int size, ErrorCode & );
    void delete( unsigned int record, ErrorCode & );

protected:
    PersistentIntArray * _control;
};
```

Figure 5.29: Class Declaration: RelativeRecordFile

the cell is empty; otherwise, it returns an error code. The update operation stores the data to be written in the cell that corresponds to the record number argument; otherwise, it returns an error code. The delete operation marks the the cell that corresponds to the record number argument as empty, if the cell is not empty; otherwise, it returns an error code.

5.3.1.4 IndexedRecordFiles

The IndexedRecordFile class specializes the RelativeRecordFile class to abstract indexed files. Figure 5.30 contains a simplified version of the C++ declaration for class IndexedRecordFile. The init operation initializes the object's state, which consists of a count of the number of indices the file has, an array of up to 255 FileIndexes that stores the keys for each record in the file, and arrays containing the starting location and lengths of keys within the file's records. IndexedRecordFiles must have at least one key, the *primary* key.

The read operation retrieves the data from the cell that corresponds to the record number argument, if the cell is not empty; otherwise, it returns an error code. The write operation stores the data to be written in the cell that corresponds to the record number argument, if the cell is empty; otherwise, it returns an error code. The update operation stores the data to be written in the cell that corresponds to the record number argument; otherwise, it returns an error code. Both the write and update operations also extract the values of the keys for the record being written and store them in the appropriate FileIndexes. The delete operation marks the the cell that corresponds to the record number argument as empty, if the cell is not empty; otherwise, it returns an error code. It also deletes the values of the keys for the record being deleted from the appropriate FileIndexes.

The keyToRecord operation returns the record number that corresponds to the given index number and key, if the key exists; otherwise, it returns an error code. The keyToRecordNext operation also returns the record number that corresponds to the given index number and key, if the key exists; otherwise, it returns an error code. It supports sequential access of the records in an indexed file by also returning the value of the next key in the same index. These operations use the at and atNext operations provided by the FileIndex class.

```
class IndexedRecordFile : public RelativeRecordFile {
public:
    IndexedRecordFile( PersistentStore * source );

    int init( int argc, char *argv[], FileSystemInterface * = 0 );
    void flush();

    int read(  unsigned int record, char * buffer,
               unsigned int size, ErrorCode & );
    int write( unsigned int record, char * buffer,
               unsigned int size, ErrorCode & );

    int update(  unsigned int record, char * buffer,
                 unsigned int size, ErrorCode & );
    void delete( unsigned int record, ErrorCode & );

    int keyToRecord(     int index, char * key, ErrorCode & );
    int keyToRecordNext( int index, char * key, ErrorCode & );

protected:
    unsigned int _indices;
    FileIndex *  _index[255];

    unsigned int  _start[255];
    unsigned int  _length[255];
};
```

Figure 5.30: Class Declaration: IndexedRecordFile

5.3.2 Application Interface Objects

Three subclasses of the RecordStream class, shown in Figure 5.31, provide interfaces for both sequential and random access to the classes of record files described above.

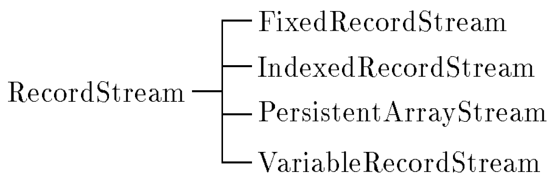

Figure 5.31: Record Stream Class Hierarchy

5.3.2.1 FixedRecordStreams

FixedRecordStreams are RecordStreams that support the sequential reading and writing of FixedRecordFiles. Figure 5.32 contains a simplified version of the C++ declaration for class FixedRecordStream.

```
class FixedRecordStream : public RecordStream {
public:
    FixedRecordStream( PersistentObject * source );

    int read( char * buffer, int size, ErrorCode & );
    int write( char * buffer, int size, ErrorCode & );

protected:
    FixedRecordFile * _file;
};
```

Figure 5.32: Class Declaration: FixedRecordStream

5.3.2.2 VariableRecordStreams

VariableRecordStreams are RecordStreams that support the sequential reading and writing of VariableRecordFiles. Figure 5.33 contains a simplified version of the C++ declaration for class VariableRecordStream.

```
class VariableRecordStream : public RecordStream {
public:
    VariableRecordStream( PersistentObject * source );

    int read( char * buffer, int size, ErrorCode & );
    int write( char * buffer, int size, ErrorCode & );

    int minimumRecordSize();
    int maximumRecordSize();

protected:
    VariableRecordFile * _file;
};
```

Figure 5.33: Class Declaration: VariableRecordStream

5.3.2.3 IndexedRecordStreams

IndexedRecordStreams are RecordStreams that support the sequential reading and writing of IndexedRecordFiles. Figure 5.34 contains a simplified version of the C++ declaration for class IndexedRecordStream.

The concept of a current file position for IndexedRecordStreams differs from that of other RecordStreams. An IndexedRecordFile can be read sequentially using the keys of any of its indices. The setIndexNumber allows an application to choose which FileIndex should be used for subsequent read invocations. The operation returns an error code if the file does not have an index that corresponds to the given argument number. The setNextKey allows an application to reset the file position by setting the value of the next key that will be looked up.

5.4 Constraints on the Framework

The framework presented in this chapter is both an abstract model of file systems and a practical design for file system construction. To build a working file system, one must combine components that belong to the following categories: physical storage devices, storage organization and sharing, logical storage devices, naming objects, file structuring objects, and application interfaces objects. If the set of classes in the framework includes the necessary concrete classes, the job of the file system designer is greatly simplified. Even if some concrete classes are miss-

```
class IndexedRecordStream : public RecordStream {
public:
    IndexedRecordStream( PersistentObject * source );

    int read(   char * buffer, int size, ErrorCode & );
    int write(  char * buffer, int size, ErrorCode & );
    int update( char * buffer, int size, ErrorCode & );
    void delete( ErrorCode & );

    ErrorCode setIndexNumber( int indexNumber );
    void      setNextKey( char * nextKey );

protected:
    char *             _nextKey;
    int                _indexNumber;
    IndexedRecordFile * _file;
};
```

Figure 5.34: Class Declaration: IndexedRecordStream

ing, the job of the file system designer is straightforward; see Chapter 6 for details on building concrete subclasses.

Besides the type declarations of the abstract classes' data structures and operations, the framework contains both external and internal *constraints* that limit how components can be combined. The external constraints are imposed by both software and hardware outside the file system, whereas the internal constraints are imposed by objects within the framework.

If one designs a file system to be stored on a SCSI disk, one must include an object using the SCSI protocols to communicate with the disk controller. If one designs a file system that uses the same disks as a BSD UNIX operating system, one must include objects that belong to BSD-specific subclasses of PersistentStoreContainer, File, and PersistentStoreDictionary. These concrete subclasses must correctly interpret and preserve the data structures defined by the BSD UNIX file system. If one designs a system to support application programs that require ISAM files, one must include objects that belong to application interface classes that provide indexed and sequential file access.

These examples illustrate three common external constraints:

- physical storage devices must conform to the protocols of controllers in the I/O subsystem;

- storage organization, storage sharing, and logical storage devices must be compatible with on-disk formats if compatibility with data stored by other operating systems is desired; and
- application interface objects must provide the operations required by the application programs that will be their clients.

If one includes IndexedRecordStreams in the file system, one must also include IndexedRecord-Files. Furthermore, IndexedRecordFiles require the inclusion of various kinds of PersistentArrays. If one build a file system that supports an extensible set of persistent objects types, one must include Files that can store these types. If one includes BSDContainers in the file system, one must include BSDInodes, since BSDContainers explicitly instantiate BSDInodes.

These examples illustrate three common internal constraints:

- persistent objects must provide the operations required by the application interface objects and by other persistent objects,
- storage management objects must provide the operations required by the persistent objects,
- objects that belong to some classes instantiate other objects that belong to specific classes.

5.5 Summary

This chapter describes the abstract classes in the file system framework. System programmers can construct concrete subclasses and combine them to build both conventional and experimental file systems. Application programmers can use the interfaces provided by the abstract classes to access the data stored within a file system built within the framework.

This chapter also categorizes the abstract classes into two groups:

- PersistentStores abstract persistent data storage devices, which store and retrieve blocks of data using a random access method, and
- PersistentObjects abstract objects that encapsulate and provide a better interface to the data managed by a persistent store.

Subclasses of PersistentStore, Disk and File, represent physical and logical storage devices, respectively. Subclasses of PersistentObject represent the following file system characteristics:

- storage device organization and sharing (PersistentStoreContainer),
- naming (PersistentStoreDictionary and SymbolicLink), and
- data structuring (PersistentArray, RecordFile, and AutoloadPersistentObject).

The framework is also designed to build layered systems. The lowest layers are Storage Management Layers, and they contain PersistentStores and PersistentStoreContainers. Above them is the Persistent Object Layer, which comprises naming and data structuring objects. The top layer is the Application Interface Layer, which contains objects that provide additional protocols for clients of the file system. The classes of objects in the top layer include RecordStreams and the FileSystemInterface class.

This chapter demonstrates how the framework supports the persistent object types of stream-oriented, record-oriented, and object-oriented file systems. There are many ways one could provide the necessary support, many of which would be neither simple nor extensible. The approach taken in this thesis, however, achieves both goals, because *the exact same features that support the persistent objects of object-oriented file systems also support the persistent objects of record-oriented file systems*. These features are:

1. each persistent object belongs to an extensible set of classes,
2. each class of persistent objects can define both data structures and a set of operations,
3. persistent objects are allowed to refer to other persistent objects using object identifiers instead of symbolic names,[12]
4. persistent objects can be automatically retrieved and stored without requiring application programs to explicitly open them.

[12]In the UNIX and MS-DOS file systems, only directories can refer to files using file identifiers.

Chapter 6

Building Concrete Subclasses

This chapter describes how one can extend the framework by adding concrete subclasses of the abstract classes presented in Chapter 5. This chapter also mentions several concrete subclasses that currently belong to the *Choices* class hierarchy. Examples of these classes are discussed in detail in Appendix B.

Abstract classes can be considered *single-class frameworks*[Deu89]. To build a concrete class within one of these frameworks, one must re-implement or *overload* some of the operations defined by the abstract superclass; therefore, this chapter concentrates on the operations that have been designed to be overloaded.[1] When feasible, abstract classes provide a default behavior for these operations, which can be inherited by subclasses.

One must also define instance variables and structures that conform to the layouts or formats defined by the hardware, operating system, or tool that the concrete class is designed to abstract. Because instance variables are easy to define for concrete classes and difficult to generalize, this chapter does not cover the definition of instance variables.

Using terminology from [Deu89], one can identify three sets of operations in a framework, including single-class frameworks. The first set of operations, called the "framework external interface," contains those used by classes outside the framework. The second set of operations, called the "framework internal interface," forms the interface between abstract classes and their subclasses. And the third set of operations, called the "resulting interface," contains the external interface operations plus any supplied by a concrete subclass.

[1]In C++, operations that are designed to be overloaded are declared as virtual functions in the abstract class's definition.

Most concrete subclasses discussed in this chapter provide resulting interfaces that are identical to their abstract classes' external interface. Thus they neither add new operations or remove inherited operations. One set of exceptions are the subclasses of the AutoloadPersistentObject class, which are intended to be new types of objects that add many new operations. This chapter concentrates on the internal interface of the single-class frameworks within the *Choices* file system framework.

6.1 Subclassing PersistentStores

The PersistentStore class has been subclassed to define several kinds of disks, partitions, files, and parts of files. Figure 6.1 shows the hierarchy rooted at PersistentStore. This section will primarily discuss the subclassing of Files. For more information on the subclassing of Disks, see [Kou91].

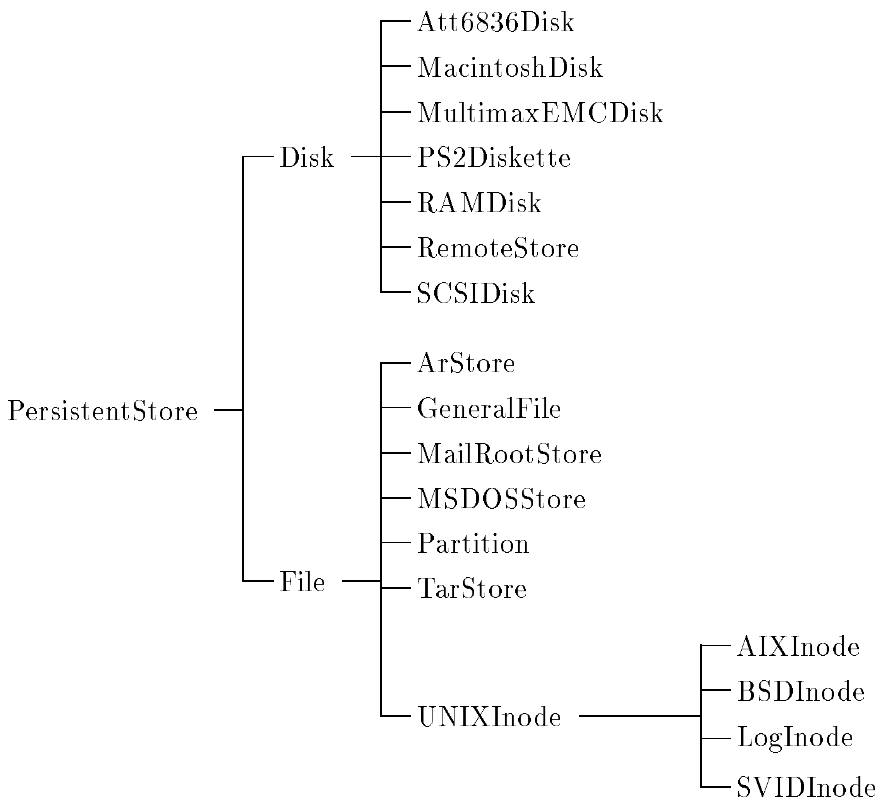

Figure 6.1: Persistent Store Class Hierarchy

Several kinds of File operations can be overloaded: data transfer (read, write, and copy), changing attributes (setRecordSize, setNumberOfRecords, and setNumberOfBlocks), and retrieving attributes (recordSize, numberOfRecords, and info). One can overload these operations to encapsulate a specific data structure, to implement a feature not supported by all Persistent-Stores, or to improve performance.

read and write Disk subclasses can inherit the implementations of read and write shown in Figures A.3 and A.4. A File subclass that abstracts fixed-length, contiguous windows of source PersistentStores can inherit the implementations of read and write shown in Figures A.5 and A.6. A File subclass that abstracts variable-length or discontiguous windows of source PersistentStores must overload read and write to perform the steps shown in Figures 6.2 and 6.3.

```
int
ExampleFile::read( unsigned int start, int count, char * buf )
{
    if( ( start + count ) > _numberOfBlocks ) count = _numberOfBlocks - start;

    int myBlock = start;
    for( int blocksRead = count; count > 0; count-- ) {
        int sourcesBlock = mapBlock( myBlock++, !AddMapping );

        if( sourcesBlock == 0 ) ByteZero( buf, blockSize() );
        else basicRead( sourcesBlock, 1, buf );

        buf += blockSize();
    }

    return( blocksRead );
}
```

Figure 6.2: Function Definition: ExampleFile::read

copy The MemoryObject class provides an implementation of the copy operation that can be inherited by all subclasses of PersistentStore. Some subclasses, however, overload copy to improve its performance. These subclasses allow instances to have empty blocks that do not correspond to blocks allocated from a lower layer (for example, UNIX file systems[Bac86]). They improve the performance of copy by not reading and writing empty, unallocated blocks.

```
int
ExampleFile::write( unsigned int start, int count, char * buf )
{
    if( ( start + count ) > _numberOfBlocks ) _numberOfBlocks = start + count;

    int myBlock = start;
    for( int blocksWritten = 0; count > 0; count-- ) {
        int sourcesBlock = mapBlock( myBlock++, AddMapping );

        if( sourcesBlock == 0 ) break;

        basicWrite( sourcesBlock, 1, buf );
        blocksWritten++;

        buf += blockSize();
    }

    return( blocksWritten );
}
```

Figure 6.3: Function Definition: ExampleFile::write

supports The PersistentStore class provides an implementation of the supports operation that is designed to be inherited by its subclasses. This implementation relies on subclasses to initialize the value of the `_supports` variable. A subclass can use one of the following methods to determine which Class of PersistentObject it supports:

- the class can be hard-coded to support a specific Class (for example, a particular Disk subclass usually supports a specific PersistentStoreContainer class),
- instances can search their persistent data to detect specific data structures (identified by "magic numbers") that correspond to a specific Class (for example, the Partition class searches certain blocks for magic numbers that would identify the format of its data),
- instances can map some class-specific data structure to the appropriate Class (for example, the `mode` bits of UNIX inodes), or
- instances can store the name of the Class that they support and look up the name in a kernel NameServer (for example, the GeneralFile class that is described in Section B.4).

recordSize Because most of the PersistentStores implemented within the framework do not implement the concept of records, i.e. they were designed to store data for PersistentCharArrays only, the default implementation of the recordSize operation returns one. A subclass that can store data for RecordFiles should define an instance variable to hold the value of the record size, and it should define the recordSize operation to return the value of this variable.

setRecordSize The default implementation of the setRecordSize operation returns the result of invoking recordSize. A subclass that can store data for RecordFiles should define the setRecord-Size operation to set the value of the record size instance variable to match the argument and return the value of the argument.

numberOfRecords The default implementation of the numberOfRecords operation returns the numberOfBlocks multiplied by the blockSize and then divided by the recordSize. A subclass that stores the number of records in an instance variable must overload the numberOfRecords operation to return the value of that variable.

setNumberOfRecords Subclasses of PersistentStore that do not allow the size of their instances to be changed can inherit the default implementation of the setNumberOfRecords operation, which returns the result of invoking numberOfRecords. Because the number of records (or bytes) is stored in data structures that differ from subclass to subclass, a subclass that allows the size of its instances to change must overload the setNumberOfRecords operation to perform the following steps:

1. store the argument in the appropriate instance variable,
2. calculate the number of blocks needed to store the new number of records,
3. invoke the setNumberOfBlocks operation, and
4. return the new number of records.

setNumberOfBlocks Subclasses of PersistentStore that do not allow the size of their instances to be changed can inherit the default implementation of the setNumberOfBlocks operation, which returns the result of invoking numberOfBlocks. (The numberOfBlocks operation, which returns the value of the `_numberOfBlocks` instance variable, is inherited from MemoryRange and should not be overloaded by any subclass.) A subclass that allows the size of its instances to change must overload the setNumberOfBlocks operation to perform the following steps:

1. if the file currently has more than the requested number of blocks,
   (a) free all blocks that are beyond the new end of the file,
   (b) store the requested number of blocks in the `_numberOfBlocks` instance variable.
2. if the file currently has less than the requested number of blocks,
   (a) allocate blocks for those that are beyond the current end of the file if needed,[2]
   (b) store the requested number of blocks in the `_numberOfBlocks` instance variable.
3. return the result of invoking the numberOfBlocks operation.

info The info operation returns the attributes of a PersistentStore and its corresponding PersistentObject. The attributes are returned through a structure that defines the union of the attributes supported by the currently implemented subclasses of PersistentStore. The default implementation returns only those attributes that are shared by all PersistentStores. A subclass that stores more attributes for its instances must overload the info operation to return the values of the additional attributes.

6.2 Subclassing PersistentObjects

constructors and destructors As with any C++ class, one usually defines a constructor and often defines a destructor for a subclass of PersistentObject. PersistentObjects, however, use these two operations differently than other objects. Because the constructor is called each time the persistent object is activated (instead of being called only when the object is created), the constructor should initialize persistent data only when the object is activated for the first time. Therefore, constructors and destructors primarily initialize and destroy transient, in-memory

[2]As mentioned above, some file systems, including UNIX file systems[Bac86], allow files to have empty blocks that do not correspond to blocks allocated from a lower layer. For such file systems, allocate calls are unnecessary.

data structures used by the object. The constructor for a subclass of PersistentObject should take a single argument, a reference to an underlying PersistentStore.

Constructor *Choices* supports the late binding of a class to its code and the dynamic loading of code for a user-defined class. This dynamic behavior is achieved by storing the address of each class's constructor in its corresponding Class object.[3] Because C++ does not allow programs to take the address of constructors, each subclass must define a function that encapsulates the invocation of its constructor. Programs can take the address of such a function; therefore, I call it an *addressable constructor*. An example of an addressable constructor is shown in Figure 6.4.

```
Object *
ExampleConstructor( PersistentStore * source )
{
    return( new Example( source ) );
}
```

Figure 6.4: Function Definition: ExampleConstructor

6.3 Subclassing PersistentStoreContainers

The PersistentStoreContainer class has been subclassed to define several different kinds of containers, including objects that divide disks, partitions, files, and parts of files into collections of Files. Figure 6.5 shows the hierarchy rooted at PersistentStoreContainer. Subclasses that partition disks (for example, MultimaxEMCContainer) and subclasses that subdivide specially formatted files (ArContainer, COFFContainer, MailContainer, and TarContainer) allow access to existing Files only. Subclasses designed to divide partitions into files (AIXContainer, BSDContainer, GeneralContainer, LogContainer, MSDOSContainer, and SVIDContainer) implement the full PersistentStoreContainer interface and support the creation of new files.

basicOpen The open operation is the only one defined by PersistentStoreContainer that allows programs to access its Files, and open relies on the subclass's implementation of basicOpen. Therefore, a subclass must overload basicOpen.

[3]See [MCK91] for a discussion of dynamic loading and Class objects in *Choices*.

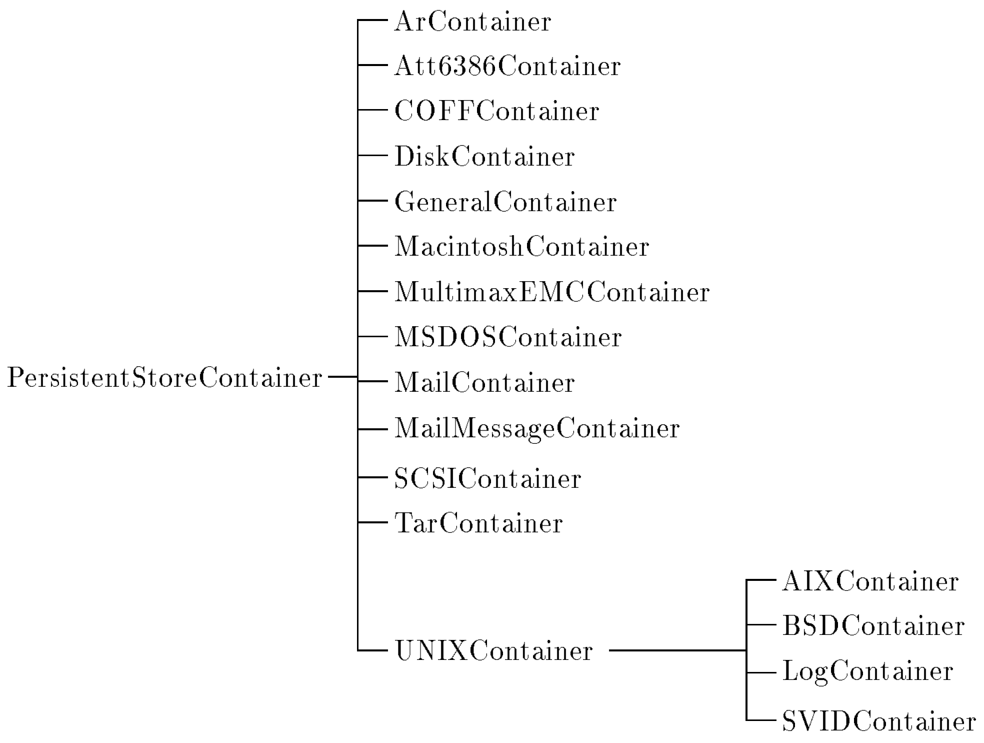

Figure 6.5: Persistent Store Container Class Hierarchy

The basicOpen operation should perform the following steps:

1. determine if the id-number argument corresponds to an existing file,
2. locate the attributes of the file,
3. determine which class the file belongs to,
4. call the constructor with the appropriate arguments for the file,
5. return the value returned by the constructor if steps 1–3 were successful,
6. otherwise, return an error code.

basicRootId The rootDictionary operation uses the basicRootId operation to identify which File stores the data for the container's root dictionary. PersistentStoreContainer defines basicRootId to return the id-number of the first File in the container. Therefore, a subclass needs to overload the basicRootId operation only if the appropriate File is *not* the first one in the container. The only step this operation needs to perform is to return the appropriate id-number to the caller.

About half the subclasses of PersistentStoreContainer overload basicRootId; the rest inherit the definition. One could initialize an instance variable to hold the id-number of the root dictionary in the subclass's constructor. But since the appropriate id-number is determined by the *class* of the container and not by individual *instances*, it is more space-efficient to implement basicRootId as an operation.

basicCreate The create operation, which creates new Files within a container, relies on the subclass's implementation of basicCreate. Therefore, a subclass that supports the creation of Files must overload basicCreate.

The basicCreate operation should perform the following steps:

1. determine if there is room for another file,
2. find space to store the attributes of the new file,
3. initialize the file's attributes,

4. call the constructor with the appropriate arguments for the file,

5. return the value returned by the constructor if steps 1–3 were successful,

6. otherwise, return an error code.

basicClose Files invoke their container's close operation when they are no longer needed in primary memory. The close operation relies on the implementation of basicClose in the PersistentStoreContainer class. This default implementation of basicClose simply deletes the File's instance variables from primary memory, without changing the file's attributes or checking to see if the file should be removed.

A subclass that supports the removal of Files or that allows Files to have mutable attributes should overload the basicClose operation to perform the following steps:

1. determine if the file should be removed from the container,

2. if the file should be removed, zero the file's attributes,

3. if the file's attributes changed, write the attributes to the underlying PersistentStore, and

4. delete the File's instance variables from primary memory.

basicSynchronize A PersistentStoreContainer that supports the creation or removal of Files or that supports Files with mutable attributes usually caches its state information and the attributes of its files. The synchronize operation keeps persistent data consistent with the cached data by calling basicSynchronize.

A subclass that caches state information should overload the basicSynchronize operation to perform the following steps:

1. if any state information has been modified since it was read from the underlying PersistentStore, write the modified information to the PersistentStore, and

2. mark the container unmodified.

6.4 Subclassing BlockAllocators

PersistentStoreContainers that support the creation and removal of Files usually contain Files whose size can change. Therefore, they use BlockAllocators to support the allocation and freeing of data blocks. Figure 6.6 shows the hierarchy rooted at BlockAllocator.

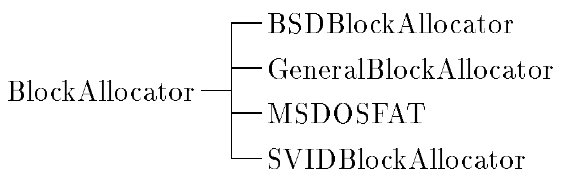

Figure 6.6: Block Allocator Class Hierarchy

Most concrete subclasses of PersistentStoreContainer that implement the basicCreate operation have corresponding concrete subclasses of BlockAllocator. However, the AIXContainer class uses the SVIDBlockAllocator class that was originally designed to be used with the SVIDContainer class.

Because the allocate and free operations rely on the subclass's implementations of basicAllocate and basicFree, a subclass must overload basicAllocate and basicFree.

basicAllocate The basicAllocate operation should perform the following steps:

1. determine if there is a block available for allocation,
2. find an available block,
3. update the data structure to reflect the allocation,
4. return the block number if steps 1–3 were successful,
5. otherwise, return zero.

basicFree The basicFree operation should perform the following steps:

1. determine if the block is currently allocated,
2. if the block has not been allocated, log this severe error,
3. otherwise, update the data structure to make the block available.

6.5 Subclassing PersistentStoreDictionaries

The PersistentStoreDictionary class has been subclassed to abstract the dictionaries of several specially formatted files and the directories of various standard file systems. Figure 6.7 shows the hierarchy rooted at PersistentStoreDictionary.

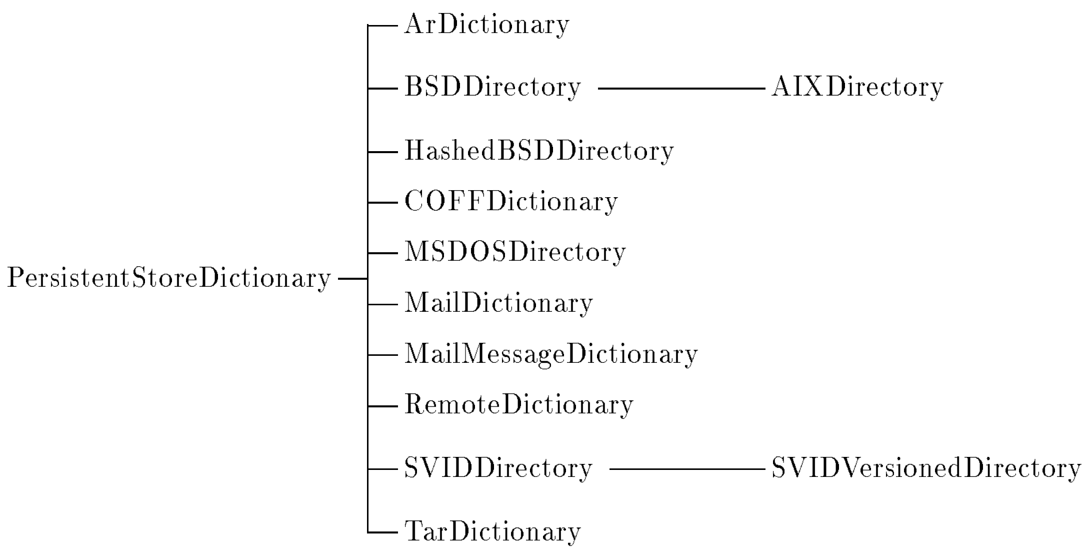

Figure 6.7: Persistent Store Dictionary Class Hierarchy

The relationship between the PersistentStoreDictionary class and its subclasses differs from the relationships between the PersistentStoreContainer and BlockAllocator classes and their subclasses. The operations of the latter classes communicate with the operations of their subclasses using only the arguments and return values of the subclasses' operations. Because some operations defined by subclasses of PersistentStoreDictionary (for example, findKey) need to return several values to the operations that invoke them, they use several instance variables to communicate with the operations that called them.

findKey The major PersistentStoreDictionary operations, open, create, add, and remove, all rely on the subclass's implementation of findKey. Because even read-only directories—those that do not implement the create, add, and remove operations—require the definition of findKey, a subclass must overload findKey. The open and remove operations use findKey only to find the location of a key within a dictionary. The create and add operations also use findKey to find space to insert the key if it is not already in the dictionary.

The findKey operation should perform the following steps:

1. search for the key within the dictionary,
2. set the `_offset` variable to the location of the key if found,
3. set the `_idNumberOfKey` variable to the id-number that corresponds to the key if found,
4. set the `_emptyOffset` variable to the location of space for the key if space was found and if the `_neededSize` argument was non-zero,
5. otherwise, set `_emptyOffset` to `-1`,
6. return one if the key was found,
7. otherwise, return zero.

basicKeys PersistentStoreDictionaries define the keys and associations operations to support programs that iterate over their elements (for example, the UNIX ls command). Because the keys and associations operations rely on the subclass's implementations of basicKeys and basicAssociations, a subclass should overload basicKeys and basicAssociations. Because the isEmpty operation also uses the basicKeys operation to determine if a PersistentStoreDictionary is storing any keys, a subclass must overload basicKeys.

The basicKeys operation extracts keys from the data structures used by a particular subclass of PersistentStoreDictionary and returns the keys in a buffer supplied by the invoker of the keys operation. It should perform the following steps:

1. copy as many keys as possible if the buffer address is non-zero,
2. count the number of keys if the buffer address is zero,
3. return the number of keys copied or counted if step 1 or 2 was successful,
4. otherwise, return an error code.

basicAssociations The basicAssociations operation extracts both keys and id-numbers from the data structures used by a particular subclass of PersistentStoreDictionary and returns them in a buffer supplied by the invoker of the associations operation. It should perform the following steps:

1. copy as many keys and id-numbers as possible to the given buffer,
2. return the number of keys copied if step 1 was successful,
3. otherwise, return an error code.

associationSize The create and add operations use associationSize to determine how large an association[4] would be for a given key size. A subclass that supports the create and add operations must overload associationSize to return the size of an association given the length of the key.

insertAssociation The create and add operations use the insertAssociation operation to store an association for a key that is not already in the dictionary. A subclass that supports the create or add operations must overload insertAssociation to perform the following steps:

1. create an association for the given key and id-number,
2. if the value of the `_emptyOffset` variable is `-1`, extend the dictionary to have enough space for the new association,
3. store the new association,
4. if the dictionary uses a more complex data structure than an array, update the rest of the data structure to be consistent with the addition of the new association, and
5. set the `_modified` flag to true.

[4]I call <key, id-number> pairs Associations, which is the Smalltalk terminology for the elements of a Dictionary. The elements of dictionaries can also be called *entries*.

clearAssociation The remove operation uses the clearAssociation operation to reset an association for a key that is to be removed from the dictionary. A subclass that supports the remove operation must overload clearAssociation to perform the following steps:

1. zero the storage for the old association,
2. if the dictionary uses a more complex data structure than an array, update the rest of the data structure to be consistent with the removal of the old association, and
3. set the `_modified` flag to true.

6.6 Subclassing Other Kinds of PersistentObjects

One can extend the PersistentObject class hierarchy by adding more than just subclasses of PersistentStoreContainer and PersistentStoreDictionary. The AutoloadPersistentObject, RecordFile, and PersistentArray classes discussed in Chapter 5 can also be subclassed. For example, the AutoloadPersistentObject hierarchy has been extended to include a set of classes that demonstrate the capabilities of PersistentObjects by implementing a simple programming language. Figure C.1 shows the hierarchy of language classes.

Programmers can define any new operations for subclasses of PersistentObject, but there are two inherited operations that they may need or want to redefine: init and flush.

init Because the constructor for a PersistentObject is called each time it is activated, and because the only argument for PersistentObject constructors should be the underlying PersistentStore, the init operation is intended to allow programs to initialize the persistent data of a PersistentObject. To support an arbitrary number and type of arguments, it uses the traditional UNIX/C argument-passing mechanism: `(int argc, char *argv[], FileSystemInterface *=0)`. The optional FileSystemInterface argument can be used if the object will interpret any of the elements of the argument vector as a pathname. If a FileSystemInterface is supplied as an argument, it can be used to interpret pathnames; otherwise, the process's default FileSystemInterface can be used.

flush *Choices* uses a set of classes and programmer conventions to support the memory management of active objects.[5] Currently, these classes and conventions use reference-counting for all Objects in the system, including active PersistentObjects.[6] References to PersistentObjects have both a transient and a persistent form. When a persistent object is active, its references to other objects can be in either form; however, when an object is inactive, its references must be in their persistent form.

If a class defines objects that do not refer to other objects, it can inherit the null flush operation from PersistentObject. Otherwise, the flush operation must be defined to convert all references of an object to their persistent format. This operation is called by the noRemainingReferences operation, which is inherited from the PersistentObject class.

6.7 Summary

The abstract classes in the framework were designed to be subclassed. These abstract classes include: Disk, File, PersistentObject, PersistentStoreContainer, BlockAllocator, and PersistentStoreDictionary. To allow the framework to be extended, these classes define public operations that all subclasses should support. This enables an instance of any concrete subclass to be used where an instance of the abstract class is needed.

Several techniques can be used to support the refinement of the behavior described by the operations of an abstract class:

- it can implement an operation with abstract code that relies on operations that must be implemented by subclasses;
- it can define a default behavior for an operation, which can optionally be reimplemented by subclasses; and
- it can forward an operation to an instance variable that is set by the constructor of the subclasses.

The *Choices* file system framework uses a combination of all the techniques listed here.

[5] See [MCK91] for a discussion of garbage collection in *Choices*.

[6] Storage of inactive PersistentObjects can be managed using either reference-counting or other forms of garbage collection.

Even when code cannot be reused through inheritance, the design for concrete subclasses can be shared. This chapter facilitates design sharing by specifying the intended behavior of operations that must be implemented by subclasses.

Chapter 7

Performance

Within a disk-based computer system, disk latencies dominate file operation times. To reduce these delays, file systems use various *caching* techniques, such as the *buffer cache* used in UNIX[Bac86] and the *memory-mapped files* used in *Choices*. A buffer cache allows the file system to keep copies of many of the most recently used disk blocks in physical memory. Since recently accessed blocks are more likely to be reused, the cache can greatly reduce the cost of reading and writing data blocks. The UNIX buffer cache is implemented in software. It uses a least-recently-used buffer replacement algorithm and hashing to map disk block numbers to buffer addresses. In contrast, *Choices* allows the file system to reuse the page replacement algorithms of its virtual memory management system. Instead of using a software mapping, *Choices* uses the virtual memory hardware to map requests for disk blocks to buffer addresses.

Because caching often speeds up file operations by a factor of ten, I measured operations both when the operation generated a cache *miss* and a cache *hit*. To make comparisons more significant, I used the same amount of physical memory, two megabytes, for caching disk blocks in both systems, and I tested the operations in the same order on each system. Also, because disk latencies vary from access to access, I repeated each test several times and report the mean value of each measurement and the 95% confidence interval for the mean.

Because the BSD file system is the most efficient of the systems that currently can be built from within the *Choices* file system framework, and because it uses the same on-disk data structures as Encore's version of UNIX, I have chosen to measure its performance.

File open, create and close in microseconds							
Operation	Cached	*Choices*			Encore UNIX		
Open existing file	NO	32173	±	62	28812	±	241
Open existing file	YES	4163	±	86	2722	±	14
Open currently open file	YES	2593	±	75	2067	±	86
Create new file	NO	29854	±	939	25546	±	1991
Close file	NO	72208	±	2257	80303	±	22233

Table 7.1: File Access Operation Measurements

7.1 File Access Operations

To use the file system, an application program must first gain access to files via open or create operations. Table 7.1 contains measurements of the time it takes to open existing files and create new files. The open operation uses both the current directory to convert a file name to an *inumber* and an in-core *inode* to convert the inumber to a reference to an open file object. *Choices* takes slightly longer than UNIX to open files, regardless of whether the disk block describing the file is cached. The reason *Choices* takes longer is that it builds a caching object, which maps the file into memory. This small amount of extra overhead for file opens is expected to be amortized over the entire time the file is open. The create operation is similar to the open operation; I chose to measure an uncached create only, since *Choices* flushes modified directory blocks to the disk after a file is created. Again, the creation time of a caching object accounts for the difference between creating files for *Choices* and for UNIX. I also measured the close operation for a file opened in *read-only* mode. The times are similar for both systems. The reason the close operation takes 70 to 80 milliseconds is that the in-core inode structure must be written back to the disk. Even if a file has not been modified, its inode structure will be modified, since the file access time is stored within the inode structure.

7.2 File I/O Operations

The most important operations on open files are read and write. Measurements of these operations are given in Table 7.2. Before blocks can be read or written, logical block numbers must be mapped to physical block numbers using the data stored in an inode structure. Inodes organize this block mapping information into a variable level tree. I measured I/O operations using both

Read, write, and lseek times in microseconds							
Operation	Cached	*Choices*			Encore UNIX		
Read block direct	NO	26803	±	420	33002	±	1275
Read block direct	YES	2524	±	106	3784	±	128
Read block indirect	NO	58841	±	4876	53457	±	769
Read block indirect	YES	2726	±	294	4358	±	219
Write block direct	YES	3752	±	207	3884	±	324
Write block indirect	YES	3168	±	23	4324	±	306
Lseek	—	111	±	5	194	±	6

Table 7.2: Data Access Operation Measurements

direct blocks and for single-indirect blocks.[1] All the I/O measurements reported in Table 7.2 are for reads or writes of 8192-byte *aligned* blocks. For cached read and write operations, *Choices* performs better, since it uses virtual memory hardware to map disk block number to buffer addresses. For uncached read and write operations, *Choices* and UNIX perform similarly, since both systems must perform disk I/O and update mapping information.

The lseek operation, which repositions the stream file location pointer, is essential for randomly accessed files. Table 7.2 also reports the overhead of the lseek operation. *Choices* performs lseeks faster primarily because it provides a more efficient system call mechanism[Rus91].

The interactions between various file system operations can often lead to unanticipated results. Therefore, I not only measured the times of individual operations, but I also measured the time of performing a common series of operations: copying an entire file. For this test I chose to copy a one megabyte file; I measured both the time to copy the data blocks from disk-to-disk and from cache-to-cache. Table 7.3 shows the results of these tests. For disk-to-disk copies, *Choices* performs slightly faster, largely owing to the efficiency of the *Choices* caching mechanism. For cache-to-cache copies, *Choices* takes less than half the time, again owing to the efficiency of the *Choices* caching mechanism. *Choices* also provides a single operation, copy, to copy an entire file. By avoiding the overhead of making many (256) system calls, *Choices* provides a substantially faster file copy mechanism.

[1]Double and triple-indirect blocks are seldom used in BSD file systems.

Copy one megabyte files in seconds			
Block size	Cached	*Choices*	Encore UNIX
8192	NO	8.167 ± 0.02	8.542 ± 0.11
8192	YES	.906 ± 0.02	2.019 ± 0.11
1048576	YES	.562 ± 0.03	—

Table 7.3: File Copy Measurements

7.3 Summary

A system constructed within an abstract framework using the object-oriented facilities of C++ could perform poorly. The data presented in this chapter, however, show that instead of yielding an unacceptably slow file system, the *Choices* file system prototype performs comparably to a commercial file system that runs on the same hardware and uses the same on-disk data structures.

Chapter 8

Conclusions

This chapter discusses the evolution of the *Choices* file system framework, mentions the benefits it derived from object-oriented techniques, summarizes the framework presented in this thesis, and presents opportunities for future research.

8.1 Evolution of the Framework

As stated in Chapter 3, frameworks are a result of iterative design. The *Choices* file system framework has been extended many times since it was first developed. Early in its development, several of these extensions required major reorganizations of the class hierarchy and modifications of abstract classes. Later, however, many new features were incorporated with little or no changes to the framework.

During its evolution, there were three major versions of the framework. Each version is briefly described in this section, including its major features, abstractions, improvements, and deficiencies. For a detailed description of the third version, see Chapter 5; for detailed descriptions of the previous two versions, see [MLRC88] and [MCRL89].

8.1.1 UNIX-Like File Systems

The first major version of the framework supported the design and construction of UNIX-like file systems. Members of the *Choices* project built object-oriented implementations of both the BSD and System V UNIX file systems. Since they shared much design, these two file systems contained many common abstractions. By encoding the common design elements as abstract classes, I developed the first version of the framework.

The key abstraction was the MemoryObject class, from which most classes in the framework inherited. Not only were the Disk, Partition, and BSDInode classes subclasses of MemoryObject, but so were the PersistentArrayStream, BSDContainer, and BSDDirectory classes. The FileSystemInterface class was similar to the current design. There were no concepts similar to PersistentObjects, PersistentArrays, or BlockAllocators.

This version of the framework was an improvement over the original, separate implementations of the two file systems, since it benefited from all the object-oriented techniques presented in Chapter 2 and summarized in this chapter. Nevertheless, it suffered from an overuse of inheritance, and it was not able to model the MS-DOS file system.

8.1.2 Stream-Oriented File Systems

The second major version of the framework supported the design and construction of many kinds of stream-oriented file systems. It was motivated by an effort to model the MS-DOS file system and to restrict the use of inheritance to express the *is-a* relationship.

Besides the MemoryObject class, the key abstractions were the PersistentStoreContainer and PersistentStoreDictionary classes. The BlockAllocator abstraction was also added. There were no concepts similar to PersistentArrays, and the abstractions currently encoded in the PersistentObject class were not well-defined.

The additional abstractions and improved usage of inheritance resulted in a greatly enhanced version of the framework. Not only did the framework successfully model UNIX and MS-DOS file systems, but it also was able to model several non-conventional file systems, described in Section B.2, without requiring any modifications.

Despite the improvements in the second version, it was still incapable of modeling object-oriented or record-oriented file systems, since it lacked a well-defined concept of a PersistentObject.

8.1.3 An Object-Oriented File System Framework

The current version of the framework supports the design and construction of stream-oriented, object-oriented, and record-oriented file systems. It was motivated by an effort to model a persistent object store and to be a dynamically extensible system.

The features of the MemoryObject class that related to persistent data were split into a subclass of MemoryObject, class PersistentStore. This enabled the *Choices* file system framework to be further refined without requiring changes to the *Choices* virtual memory framework, which also relied on the MemoryObject class as a key abstraction. The addition of the PersistentStore class to the framework allowed the requirement that the data managed by each PersistentStore must also be encapsulated by a PersistentObject.

The PersistentObject unified the data encapsulation attributes of the PersistentStoreContainer and PersistentStoreDictionary classes, which were introduced in the previous version of the framework. Together with the newly added PersistentArray abstraction, they categorized all the persistent objects found in stream-oriented file systems. Thus, all Disks and Files were encapsulated as objects which either contained other Files, mapped symbolic names to file identifiers, or were arrays of persistent data.

But the PersistentObject class more than just unified the concepts of stream-oriented file systems. It also enabled the support for a dynamically extensible set of Classes of PersistentObjects, an example of which is presented in Appendix C. The new types of PersistentObjects that could be added to the system included file structuring classes that encoded the abstractions of record-oriented file systems.

The file structuring and access classes presented in Section 5.3 demonstrate how the framework easily was extended to incorporate an entire category of file systems (record-oriented file systems). This extension used PersistentArrays as building blocks and resulted in only one change to an existing abstract class: the numberOfRecords operation was added to the PersistentStore class. This change was made to improve the performance of record-oriented files, not because it needed to provide basic functionality.

There are still many features of file systems which could be incorporated into the *Choices* file system model. Some of these include more types of access control, file and record locking, support for transactions, and various types of garbage collection for general persistent object stores.

8.2 Software Engineering Benefits

The *Choices* file system exhibits the traditional benefits of object-oriented programming, and it has served as an example that illustrates various attributes of object-oriented frameworks[OJ90, WBJ90].

8.2.1 Reusability

The classes in the framework reuse much code, many interfaces, and many designs. Classes such as PersistentStoreContainer, PersistentStoreDictionary, and UNIXInode provide most of the code for their subclasses. All the major abstract classes described in Chapter 5 provide interfaces for their subclasses. Even when code cannot be inherited by a subclass, the design can be reused as described in Chapter 6.

8.2.2 Portability

The prototype implementation of the framework is highly portable: it runs as part of the *Choices* kernel on various hardware platforms (for example, the Encore Multimax, the Apple Macintosh, the Sun Microsystems SPARC, and the IBM PS/2), as an application on versions of UNIX (SunOS, UMAX, and Cray UNICOS), and as a "public service object" in the ES-KIT system[CKH$^+$89]. Furthermore, it can be compiled using either the AT&T **cfront C++** translator or the Free Software Foundation **g++** compiler.

8.2.3 Maintainability

Maintainability is difficult to discuss in the context of a project that has had one primary developer. Nevertheless, I can identify two attributes of the framework that enhance the maintainability of systems built using it. The first attribute is that all data are encapsulated; thus, changes in the code for one class seldom affect the code of other classes. One major exception is that changes to a superclass can have drastic effects on its subclasses. The second attribute is that almost all sharable code is implemented only once, in a superclass of the classes that share it. This prevents code from being implemented more than once within the same system, with implementations becoming inconsistent over time.

8.2.4 Customizability

The design of the framework, aided by the implementation of first-class classes[MCK91], yielded highly customizable file systems. When making a member of the *Choices* operating system family, one can choose almost any combination of ten sets of storage management classes (AIX, BSD, SVID, MSDOS, General, Remote, Ar, Tar, COFF, and mail) simply by editing ten tokens in a *Choices* "Makefile." Because all persistent objects and application interface objects in the framework depend solely on the abstract interfaces provided by the storage management classes, changes in the set of supported concrete storage management subclasses just require relinking, not recompilation. Furthermore, most sets of classes not supported at link-time can be added at run-time (one notable exception is the COFF classes, which are reused by the dynamic code-loading mechanism[MCK91]).

8.2.5 Extensibility

A measure of the extensibility of a framework is how many changes it requires to incorporate new features. The *Choices* file system framework supported many extensions that did not require any changes to its abstract classes. This section contains examples of these extensions. Furthermore, Appendix C demonstrates how the application interface can be extended to support non-conventional file system applications.

Section B.2 presents the storage management classes of several custom file systems. These systems abstract files formatted by various UNIX tools as PersistentStoreContainers. Not only do these classes extend the framework without requiring any changes to its abstract classes, but they also extend the functionality of standard UNIX shell commands without requiring any changes to them.

Another extension was the implementation of a log-structured file system by two sets of students in an operating systems class.[1] Both sets of students demonstrated working prototypes of the file system proposed in [OD89]. The file system framework allowed them to concentrate

[1]The class, CS323, was taught at the University of Illinois in the Fall of 1989 by Professor Ralph Johnson.

on the novel aspects of the proposed system and to reuse much of the code that normally would need to be rewritten in conventional systems. In contrast, two years earlier a team of 12 students made less progress implementing the much simpler and better documented System V file system.

Students in another operating systems class[2] used the framework for two projects. The first involved adding VMS-style version numbers to System V directories, and the second involved implementing a simple remote file system. Again, neither project required any changes to the framework.

8.3 Summary

This thesis demonstrates that one can create a simple, general, and extensible model of file systems by presenting the *Choices* object-oriented framework for file systems. The framework contains two fundamental abstractions. The PersistentStore class defines an interface for all storage objects, from disks to disk partitions to files to sections of files. The data managed by each PersistentStore is encapsulated by a PersistentObject. PersistentStores manage data access within the file system, while various kinds of PersistentObjects encapsulate the organization, naming, and structure of persistent data.

The *Choices* file system framework supports the design and construction of both conventional and experimental file systems. It not only allows the addition of new kinds of PersistentStores and PersistentObjects, but it also simplifies the implementation of these new classes of objects. Furthermore, it provides interoperability of various kinds of files and collections of files.

Within the framework, storage is organized by nested PersistentStoreContainers, which divide a PersistentStore into a fixed or variable number of Files. If the number or the size of Files within a container is variable, then the container uses a BlockAllocator to manage its free blocks. Each class of BlockAllocator implements a form of storage management appropriate for the types of inter-object references allowed by the container. Naming is provided by PersistentStoreDictionaries, which map symbolic names to internal file identifiers.

[2] CS323 taught in the Fall of 1990 by Professor Roy Campbell.

The framework incorporates three models for structuring the data contained within files: files that contain unformatted arrays of bytes or words, files that contain collections of records, and files that are data structures encapsulated by persistent objects. To support persistent arrays, a PersistentStore must hold both the data and the file's *size*. To support collections of fixed size records, a PersistentStore must also hold the file's *record size*. To support the storage of persistent objects, a PersistentStore must also hold the file's *class*. Furthermore, the file classes must belong to an extensible set, any persistent object must be allowed to store references to other persistent objects within the same container, and some classes of persistent objects must be capable of being implicitly retrieved and stored. As shown in Chapter 5, a file system framework that supports these features for persistent objects also supports record-oriented file system features such as variable length records and indexed sequential access methods.

The framework includes concrete classes that implement the storage management of standard file systems, like UNIX and MS-DOS, and several custom file systems, like those that support the formats defined by **ar**, **tar**, **mail**, and COFF (see Appendix B). It also includes classes that provide byte-stream, record-oriented, and object-oriented interfaces to persistent data storage.

8.4 Future Work

There are many directions in which the research described in this thesis can be extended. The framework could be extended to incorporate more kinds of access control for files, to support record locking and transactions, and to implement various types of garbage collection for general persistent object stores. One could also use it as a test-bed for experimenting with network and distributed file systems, and to incorporate features of object-oriented database management systems.

One could also use the framework to develop custom file systems to support software development or to meet the needs of configuration management and software project management.

Finally, more work could be done in describing the framework itself. One of the aspects of frameworks that is hardest to specify is the constraints placed on the components[JR91]. Formal specification of the framework, using a language like Object-Z, may help describe framework attributes that cannot be expressed well in most programming languages.

Appendix A

Source Code

This appendix contains pseudo-code for selected operations of various abstract classes. Each operation presented here is written in C++ and is intended to illustrate the design or algorithm of the operation, but it is not intended to be the exact text of the *Choices* file system code. The reason for providing pseudo-code instead of the actual code is that the design should be easier to understand when some details are eliminated. These details include some performance enhancements and memory management code.

A.1 Selected PersistentStore Operations

```
// Return a PersistentObject encapsulating the data of this PersistentStore.
// Return zero if the Class of the data structure stored in this PersistentStore
// is not either aClass or any of its subclasses.
// Set status code to indicate success or reason for failure.
PersistentObject *
PersistentStore::asA( Class * aClass, ErrorCode & status )
{
   PersistentObject * persistentObject = 0;
   // Find the concrete subclass of aClass that describes
   // the encapsulated data of this PersistentStore
   Class * supportedClass = supports( aClass );
   if( supportedClass != 0 ) {
      // If the object has already been instantiated, skip instantiation
      if( _persistentObject == 0 ) {
         // Instantiate the object and store a reference to it for future calls.
         _persistentObject = supportedClass->constructor( this );
         // If this is an AutoLoadPersistentObject, load its data
         if( ( _persistentObject != 0 ) && ( _numberOfUnits > 0 ) &&
            _persistentObject->isKindOf(AutoloadPersistentObjectClass) ) {
            char * buffer = new char[_numberOfUnits * unitSize()];
            read( 0, _numberOfUnits, buffer );
            int baseSize = PersistentObjectClass->sizeOf();
            int bytesToCopy = supportedClass->sizeOf() - baseSize;
            char * persistentData = ((char *)_persistentObject) + baseSize;
            ByteCopy( buffer, persistentData, bytesToCopy );
            delete buffer;
         }
      }
      persistentObject = _persistentObject;
      status = Success;
   }
   if( persistentObject == 0 ) status = ProtocolMismatch;
   return( persistentObject );
}
```

Figure A.1: Function Definition: PersistentStore::asA

```
// This operation is inherited from the MemoryObject class.
// Copy the data of this MemoryObject to the destination MemoryObject
ErrorCode
MemoryObject::copy( MemoryObject * destination )
{
    int log2BlockDifference = _log2BlockSize - destination->log2BlockSize();

    int blocksPerRead = 1;
    int blocksPerWrite = 1;

    if     ( log2BlockDifference > 0 ) blocksPerWrite <<= log2BlockDifference;
    else if( log2BlockDifference < 0 ) blocksPerRead <<= -log2BlockDifference;

    char * buffer = new char[blocksPerRead * blockSize()];

    int blocksRead = 0;
    int blocksWritten = 0;
    ErrorCode errorCode = Success;

    while( blocksRead < _numberOfBlocks ) {
        int blocks = read( blocksRead, blocksPerRead, buffer );
        blocksRead += blocks;

        if( ( blocks < blocksPerRead ) &&
            ( blocksRead < _numberOfBlocks ) ) {
            errorCode = ReadError;
            break;
        }

        blocks = destination->write( blocksWritten, blocksPerWrite, buffer );
        blocksWritten += blocks;

        if( blocks != blocksPerWrite ) {
            errorCode = WriteError;
            break;
        }
    }

    delete buffer;

    destination->setNumberOfBlocks( _numberOfBlocks );

    return( errorCode );
}
```

Figure A.2: Function Definition: PersistentStore::copy

A.2 Selected Disk Operations

```
int
Disk::read( unsigned int start, int count, char * buf )
{
    _mutex->P();
    int blocksTransferred = doio( start, count, buf, DiskReadOperation ) ;
    _mutex->V();

    return( blocksTransferred );
}
```

Figure A.3: Function Definition: Disk::read

```
int
Disk::write( unsigned int start, int count, char * buf )
{
    _mutex->P();
    int blocksTransferred = doio( start, count, buf, DiskWriteOperation ) ;
    _mutex->V();

    return( blocksTransferred );
}
```

Figure A.4: Function Definition: Disk::write

A.3 Selected File Operations

```
int
File::read( unsigned int start, int count, char * buf )
{
    if( ( start + count ) > _numberOfBlocks ) {
        count = _numberOfBlocks - start;
    }

    return( basicRead( start, count, buf ) );
}
```

Figure A.5: Function Definition: File::read

```
int
File::write( unsigned int start, int count, char * buf )
{
    if( ( start + count ) > _numberOfBlocks ) {
        count = _numberOfBlocks - start;
    }

    return( basicWrite( start, count, buf ) );
}
```

Figure A.6: Function Definition: File::write

```
int
FileObject::basicRead( unsigned int start, int count, char * buffer )
{
   int blocksRead = 0;

   if( ( count > 0 ) && ( buffer != 0 ) ) {
      if( _isBuffered ) {
         int bytesToCopy = count * blockSize();
         int firstBlock = start + _offset;
         int byteOffset = firstBlock * blockSize() % _source->blockSize();
         int lastBlock = firstBlock + count - 1;
         firstBlock /= blockFactor();
         lastBlock  /= blockFactor();

         int blocksToRead = lastBlock - firstBlock + 1;
         char * blockBuffer = new char[blocksToRead * _source->blockSize()];

         blocksRead = _source->read( firstBlock, blocksToRead, blockBuffer );
         if( blocksRead < blocksToRead ) {
               blocksRead = ( blocksRead * blockFactor() ) -
                        ( byteOffset / blockSize() );
         }
         else blocksRead *= blockFactor();

         ByteCopy( blockBuffer + byteOffset, buffer, bytesToCopy );
         delete blockBuffer;
      }
      else {
         start *= blockFactor();
         start += _offset;
         count *= blockFactor();

         blocksRead = _source->read( start, count, buffer ) / blockFactor();
      }
   }

   return( blocksRead );
}
```

Figure A.7: Function Definition: File::basicRead

```
int
FileObject::basicWrite( unsigned int start, int count, char * buffer )
{
  int blocksWritten = 0;

  if( ( count > 0 ) && ( buffer != 0 ) ) {
    if( _isBuffered ) {
      int bytesToCopy = count * blockSize();
      int firstBlock = start + _offset;
      int byteOffset = firstBlock * blockSize() % _source->blockSize();
      int lastBlock = firstBlock + count - 1;
      firstBlock /= blockFactor();
      lastBlock  /= blockFactor();

      int blocksToWrite = lastBlock - firstBlock + 1;
      char * blockBuffer = new char[blocksToWrite * _source->blockSize()];

      _source->read( firstBlock, blocksToWrite, blockBuffer );
      ByteCopy( buffer, blockBuffer + byteOffset, bytesToCopy );
      blocksWritten = _source->write(firstBlock, blocksToWrite, blockBuffer);
      if( blocksWritten < blocksToWrite ) {
          blocksWritten = ( blocksWritten * blockFactor() ) -
                     ( byteOffset / blockSize() );
      }
      else blocksWritten *= blockFactor();

      delete blockBuffer;
    }
    else {
      start *= blockFactor();
      start += _offset;
      count *= blockFactor();

      blocksWritten = _source->write( start, count, buffer ) / blockFactor();
    }
  }

  return( blocksWritten );
}
```

Figure A.8: Function Definition: File::basicWrite

A.4 Selected PersistentStoreContainer Operations

```
File *
PersitentStoreContainer::open( int idNumber, ErrorCode & status )
{
    File * file = 0;

    if( ( 0 > idNumber) || ( idNumber >= _numberOfFiles ) ) {
        status = IdNumberOutOfRange;
    }
    else {
        _mutex->P();

        file = _files[idNumber];

        if( file == 0 ) {
            file = basicOpen( idNumber, status );
            if( file != 0 ) {
                _files[idNumber] = file;
                if ( _openFiles++ == 0 ) reference();
            }
        }
        else status = Success;

        _mutex->V();
    }

    return( file );
}
```

Figure A.9: Function Definition: PersistentStoreContainer::open

```
File *
PersitentStoreContainer::create( int hint, Class * aClass, ErrorCode& status )
{
    _mutex->P();

    File * file = basicCreate( hint, aClass, status );

    if( file != 0 ) {
        int idNumber = file->idNumber();
        _files[idNumber] = file;

        if ( _openFiles++ == 0 ) reference();
        _modified = 1;
    }

    _mutex->V();

    return( file );
}
```

Figure A.10: Function Definition: PersistentStoreContainer::create

```
PersitentStoreDictionary *
PersitentStoreContainer::rootDictionary()
{
    ErrorCode errorCode;
    File * file = basicOpen( basicRootId(), errorCode );

    if( file == 0 ) return( 0 );
    else return( file->asA( PersitentStoreDictionaryClass, errorCode ) );
}
```

Figure A.11: Function Definition: PersistentStoreContainer::rootDictionary

```
void
PersistentStoreContainer::close( File * file )
{
    int idNumber = file->idNumber();

    Assert( ( 0 <= idNumber ) && ( idNumber < _numberOfFiles ) );

    _mutex->P();

    basicClose( file );

    _files[idNumber] = 0;
    --_openFiles;

    int noOpenFiles = ( _openFiles == 0 );

    _mutex->V();

    if( noOpenFiles ) unreference();
}
```

Figure A.12: Function Definition: PersistentStoreContainer::close

```
void
PersitentStoreContainer::synchronize()
{
    _mutex->P();
    basicSynchronize();
    _mutex->V();
}
```

Figure A.13: Function Definition: PersistentStoreContainer::synchronize

A.5 Selected BlockAllocator Operations

```
int
BlockAllocator::allocate( int & blocks, int hint, int numberOfBlocks )
{
    _mutex->P();
    int blocksAllocated = basicAllocate( blocks, hint, numberOfBlocks );
    _mutex->V();

    return( blocksAllocated );
}
```

Figure A.14: Function Definition: BlockAllocator::allocate

```
void
BlockAllocator::free( int & blocks, int blockNumber, int numberOfBlocks )
{
    _mutex->P();
    basicFree( blocks, blockNumber, numberOfBlocks );
    _mutex->V();
}
```

Figure A.15: Function Definition: BlockAllocator::free

A.6 Selected PersistentStoreDictionary Operations

```
ErrorCode
PersitentStoreDictionary::associations( char * buffer, int size, int & count )
{
    ErrorCode status = 0;

    _mutex->P();

    if( !_source->access( AccessRead ) ) status = FileAccessViolation;
    else status = basicAssociations( buffer, size, count );

    _mutex->V();

    return( status );
}
```

Figure A.16: Function Definition: PersistentStoreDictionary::associations

```
ErrorCode
PersitentStoreDictionary::keys( char * buffer, int bufferSize, int & count )
{
    ErrorCode status = 0;

    _mutex->P();

    if( !_source->access( AccessRead ) ) status = FileAccessViolation;
    else status = basicKeys( buffer, bufferSize, count );

    _mutex->V();

    return( status );
}
```

Figure A.17: Function Definition: PersistentStoreDictionary::keys

```
PersistentStore *
PersitentStoreDictionary::create( char * key, Class * aClass,
                                  ErrorCode & status )
{
   PersistentStore * store = 0;
   int keyLength = strlen( key );

   _mutex->P();
   if      ( !_source->access( AccessSearch ) ) status = FileAccessViolation;
   else if( keyLength <= 0 )                    status = KeyTooSmall;
   else if( keyLength > _maxKeyLength )         status = KeyTooBig;
   else if( findKey( key, associationSize( keyLength ) ) ) {
      store = _container->open( _idNumberOfKey, status );
      if( store != 0 ) {
         Class * supportedClass = store->supports( aClass );
         if( supportedClass != 0 ) status = KeyExists;
         else {
            store = 0;
            status = ProtocolMismatch;
         }
      }
   }
   else if( !_source->access( AccessWrite ) )  status = FileAccessViolation;
   else {
      store = _container->create( hint(), aClass, status );
      if( store != 0 ) {
         Class * supportedClass = store->supports( aClass );
         if( supportedClass != 0 ) {
            status = insertAssociation( key, keyLength, store->idNumber() );
            if( status == 0 ) {
               if( supportedClass->isASubclassOf(PersistentStoreDictionaryClass) ) {
                  PersistentStoreDictionary * psd = store->asA( supportedClass );
                  status = psd->add( "..", _source );
               }
            }
            else store = 0;
         }
         else {
            store = 0;
            status = ProtocolMismatch;
         }
      }
   }
   _mutex->V();

   return( store );
}
```

Figure A.18: Function Definition: PersistentStoreDictionary::create

```
ErrorCode
PersitentStoreDictionary::add( char * key, PersistentStore * store )
{
    ErrorCode status;
    int keyLength = strlen( key );
    int associationLength = associationSize( keyLength );

    _mutex->P();

    if     ( keyLength <= 0 )                    status = KeyTooSmall;
    else if( keyLength > _maxKeyLength )         status = KeyTooBig;
    else if( !_source->access( AccessWrite ) )   status = FileAccessViolation;
    else if( findKey( key, associationLength ) ) status = KeyExists;
    else status = insertAssociation( key, keyLength, store->idNumber() );
    if( status == 0 ) store->links( 1 );

    _mutex->V();

    return( status );
}
```

Figure A.19: Function Definition: PersistentStoreDictionary::add

```
PersistentStore *
PersitentStoreDictionary::open( char * key, ErrorCode & status )
{
    _mutex->P();

    PersistentStore * store = 0;

    if     ( !_source->access( AccessSearch ) ) status = FileAccessViolation;
    else if( !findKey( key, 0 ) )               status = KeyNotFound;
    else store = _container->open( _idNumberOfKey, status );

    _mutex->V();

    return( store );
}
```

Figure A.20: Function Definition: PersistentStoreDictionary::open

```
ErrorCode
PersitentStoreDictionary::remove( char * key )
{
   int status = 0;

   _mutex->P();

   if     ( !_source->access( AccessWrite ) )  status = FileAccessViolation;
   else if( !_source->access( AccessSearch ) ) status = FileAccessViolation;
   else if( !findKey( key, 0 ) )               status = KeyNotFound;
   else {
      PersistentStore * store = _container->open( _idNumberOfKey, status );
      if( store != 0 ) {
         PersistentStoreDictionary * psd =
                        store->asA( PersistentStoreDictionaryClass, status );
         if( psd != 0 ) {
            if( psd == this ) status = DirectoryBusy;
            else status = psd->empty();
         }
         else status = Success;
         if( status == Success ) {
            status = clearAssociation();
            if( status == Success ) {
               store->links( -1 );
               if( psd != 0 ) {
                  store->links( -1 );
                  _source->links( -1 );
               }
            }
         }
      }
   }

   _mutex->V();

   return( status );
}
```

Figure A.21: Function Definition: PersistentStoreDictionary::remove

A.7 Selected FileSystemInterface Operations

```
PersistentStore *
FileSystemInterface::pathOpen( char * path, ErrorCode & status,
                               PersistentStoreDictionary * psd,
                               int follow, int links )
{
    PersistentStore * aStore = 0;
    status = Success;
    char * key;

    PersistentStoreDictionary * aDictionary = findDictionary( path, key,
                                                              psd, links );
    if( aDictionary == 0 ) status = InvalidPath;
    else {
        if( ( strcmp( key, "..") == 0 ) && ( aDictionary == _root ) ) {
                aStore = aDictionary->asA( PersistentStoreClass, status );
        }
        else if( *key == '\0' ) aStore = aDictionary->open( ".", status );
        else                    aStore = aDictionary->open( key, status );

        _mountTable->substitute( aStore );
        if( follow ) substituteLink( aDictionary, aStore, status, links );
    }
    return( aStore );
}
```

Figure A.22: Function Definition: FileSystemInterface::pathOpen

```
PersistentStoreDictionaryRef
FileSystemInterface::findDictionary( char * path, char * & key,
                                     PersistentStoreDictionary * aDictionary,
                                     int links )
{
    PersistentStoreDictionaryRef startDictionary = 0;
    if( path[0] == '/' ) startDictionary = _root;
    else startDictionary = aDictionary;
    return( basicFindDictionary( path, key, startDictionary, links ) );
}
```

Figure A.23: Function Definition: FileSystemInterface::findDictionary

```
PersistentStoreDictionaryRef
FileSystemInterface::basicFindDictionary( char * path, char * & key,
                                          PersistentStoreDictionary * start,
                                          int links )
{
    PersistentStoreDictionaryRef dictionary = start;
    for( ;; ) {
        while( *path == '/' ) path++;
        _mountTable->substituteUp( dictionary, path );

        key = path;
        char tmpKey[512];
        for( int i = 0; ( ( *path != '/' ) && ( *path != '\0' ) ); ) {
            tmpKey[i++] = *path++;
        }
        if( *path == '\0' ) break;
        tmpKey[i] = '\0';

        if( ( strcmp( tmpKey, "..") != 0 ) || ( dictionary != _root ) ) {
            int status = 0;
            PersistentStoreRef store = dictionary->open( tmpKey, status );
            if( store == 0 ) {
                dictionary = 0;
                break;
            }
            _mountTable->substitute( store );
            substituteLink( dictionary, store, status, links );
            PersistentPersistentStore * store = store;
            if( store == 0 ) break;
            dictionary = store->asA( PersistentStoreDictionaryClass, status );
            if( dictionary == 0 ) {
                PersistentStoreContainer * container = store->asA(
                                    PersistentStoreContainerClass, status );
                if( container == 0 ) break;
                else dictionary = container->rootDictionary();
            }
        }
    }
    _mountTable->substituteUp( dictionary, key );
    return( dictionary );
}
```

Figure A.24: Function Definition: FileSystemInteface::basicFindDictionary

Appendix B

Storage Management Examples

This appendix describes several file systems that have been built using the abstract classes presented in Chapter 5 and the subclassing techniques presented in Chapter 6. For each file system, I will give the requirements of the system followed by the concrete classes that satisfy them. These requirements usually take the form of data structures defined by another operating system, like UNIX or MS-DOS, or system programs, like **mail** or **tar**.

B.1 Standard Operating System Formats

Since the framework was first developed by implementing the file systems of three standard operating systems (System V UNIX, BSD UNIX, and MS-DOS), it is well-suited to building such systems. Besides describing these three systems, this section describes a fourth system (AIX).

B.1.1 UNIX Storage Management Systems

There are many versions of the UNIX operating system; there are also several versions of the UNIX file system. Implementations of three of the most common versions of the UNIX file system have been built within the *Choices* file system framework. Members of the *Choices* project chose to implement the three because each version was used by at least one of our target hosts. These versions also illustrate various types of code reuse within the framework.

I will first introduce the simplest system, System V, then the most complex, BSD, followed by the one that has similarities with both, AIX, and finally I will discuss abstract classes shared by all three.

B.1.1.1 The System V UNIX File System

The System V UNIX file system[Bac86, Tho78] divides each disk or partition into three sections: a header, called a *superblock*; a fixed-size array of file descriptors, called *inodes*; and the rest of the data blocks, which can be unallocated, hold file data, or hold block mapping information.

The superblock describes the general state of the disk or partition and includes the following fields: a *magic number*, a clustering factor, the total number of inodes, the number of free inodes, a cache of free inode numbers (*inumbers*), the total number of blocks, the number of free blocks, and a partial list of free block numbers. The rest of the free blocks are organized as a linked list.

Each element in the inode array holds the following information about the file that it describes: access permissions, ownership information, the type of the file (either *regular* or *directory*), the number of bytes in the file, access and modification times, block mapping information, and the number of *links* to the file.

The format of the block mapping information is designed to be space-efficient for both small and large files, but accessing blocks in large files is slower than in small files. Each inode stores the numbers of the first ten blocks in the file; these blocks are called *direct blocks*. If the file contains more than ten blocks, the inode can store the addresses of single, double, and triple *indirect blocks*. The single indirect block contains an array of data block numbers, the double indirect block contains an array of single indirect block numbers, and the triple indirect block contains an array of double indirect block numbers. The inode structure stores each block number as a 3-byte integer, which must be converted to a standard C++ integer before it can be used by the rest of the file system.

The System V file system names files using special files called *directories*, which contain lists of <name, inumber> pairs. Each name is a string of fourteen characters padded with null characters. Each pair "links" a symbolic name to a file. If an inode has a zero link count for its file, the file will be deleted and the inode cleared.

Besides naming a group of files and other directories, each directory assigns the name "." to itself and the name ".." to its *parent* directory. This organizes all directories in a partition as a tree. The root of the tree is called the *root directory*, and its corresponding file descriptor is always the second inode. While the directory names are organized as a tree, regular file names are a directed acyclic graph, since many directories can contain links to the same regular file.

The SVIDContainer class encapsulates the inode array data structures and the fields of the superblock that are unrelated to block allocation. It defines two operations readInode and writeInode that retrieve, store, and cache inodes. The basicOpen operation checks to ensure that the inode corresponding to the given inumber has a non-zero link count and then instantiates a SVIDInode with the appropriate arguments and returns a reference to it. The basicCreate operation checks to see if there are any free inumbers in the superblock cache; if there are none, it searches through the inode array and replenishes the cache. Once it has a free inumber, it initializes the corresponding inode and then instantiates a SVIDInode with the appropriate arguments and returns a reference to it. The basicClose operation checks the inode's link count; if it is zero, it clears the inode data structure so that it is free for reallocation. It also deletes the instance of SVIDInode. The basicSynchronize operation writes any modified superblock or inode information to the underlying PersistentStore. The basicRootId operation returns two, the inumber of the root directory.

The SVIDContainer constructor creates a SVIDBlockAllocator, which encapsulates the superblock's cache of free data block numbers and the linked list of free data blocks. The basicAllocate operation returns a free block number from the superblock cache and replenishes the cache when it is empty. The basicFree operation stores a previously allocated block's number in the superblock cache and writes out part of the cache when it is full.

The SVIDInode class encapsulates the System V inode data structure. Many of its operations, including numberOfRecords, setNumberOfRecords, and info, retrieve or store inode elements. It also defines the mapBlock operation, which takes two arguments. The first argument is the block number within the file, which it maps to a block number of the file's underlying PersistentStore. If the block number is not currently mapped, mapBlock can take one of two actions: it can either request a new block from the SVIDBlockAllocator or it can return a zero, indicating that the block is not mapped. The second argument indicates which action should be taken if the block number is not currently mapped. When the write operation calls mapBlock, it requests that a mapping be added if needed. This allows files to expand by writing data to additional blocks. When the read operation calls mapBlock, it requests that a mapping not be added because files may have unmapped blocks in UNIX file systems. When a program attempts to read such blocks, the read operation zeros the given buffer. A SVIDInode can store the data for either a PersistentCharArray or a SVIDDirectory.

The SVIDDirectory class encapsulates an array of fixed-length directory entries. Each entry includes a fourteen character file name and a two byte inumber. As entries are added to a SVIDDirectory, the array grows, but when entries are removed, the array does not shrink. The clearAssociation operation marks an entry as unused by setting the inumber field to zero. The findKey operation searches sequentially through the array of entries. If it is also looking for space to add a new key, it will return the location of the first entry that has a zero inumber. The associationSize operation returns the size of an entry, which is always 16, regardless of the size of the key to be stored.

B.1.1.2 The BSD UNIX File System

The BSD UNIX file system[MJLF84] retains the higher level organization of the System V file system, but changes all data structures and adds several features. To increase performance, each partition is divided into contiguous regions called *cylinder groups*, and some superblock information is spread between the groups. To increase reliability, the superblock is replicated in each cylinder group. The BSD file system uses bitmaps to store the state of both inode and block allocation.

BSD inodes cluster data blocks for files into groups of 8 contiguous blocks. This greatly increases the I/O performance, but uses too much disk space for the many small files that commonly exist in UNIX file systems. To improve disk space utilization, the last cluster of blocks in a small file can be a fragment that contains between 1 and 7 blocks. These fragments give the system both good performance and space utilization, but they do complicate the code that manages inodes. Besides block clusters and fragments, BSD inode structures differ from System V inodes in size (128 versus 64 bytes) and the way that they store block numbers (4 byte integers versus 3 byte integers).

BSD organizes directories as an array of blocks that each can contain several variable-length entries. Each entry contains four fields: the size of the entry, the size of the name, a file name, and an inumber. The size of an entry is always a multiple of four bytes. These entries can belong to one of several sequential lists, one list per block, or to a single hash table.

The BSD file system extends the naming model of the UNIX file system by adding *symbolic links*, which are files that contain a pathname. When a program attempts to open a symbolic link, the file system returns the file that is named by the path.

A BSDContainer performs the same functions as a SVIDContainer except that it handles header information that is spread between the superblock and several cylinder groups instead of being in a single superblock, encapsulates inode allocation bitmaps instead of a superblock cache of free inode numbers, and instantiates BSDInodes instead of SVIDInodes.

A BSDBlockAllocator performs the same functions as a SVIDBlockAllocator except that it handles header information that is spread between the superblock and several cylinder groups instead of being in a single superblock, encapsulates block allocation bitmaps instead of a superblock cache of free block numbers, and handles requests for contiguous clusters of blocks instead of just single blocks.

A BSDInode performs the same functions as a SVIDInode except that it encapsulates a different inode format, handles clusters and cluster fragments, and supports either a Persistent-CharArray, a BSDDirectory, a HashedBSDDirectory, or a SymbolicLink. A BSDInode can store the data for a symbolic link within the inode itself, if the size of the data is less than 60 bytes.

A BSDDirectory performs the same functions as a SVIDDirectory except that it encapsulates an array of blocks that contain sequential lists of variable-length entries. A HashedBSDDirectory also encapsulates arrays of blocks that contain variable-length entries, but all the entries are linked together in a single open hash table. The first block serves as the table of bucket headers for the bucket lists.

B.1.1.3 The AIX File System

The AIX file system resembles a file system with a System V free list, System V inodes, and BSD directories. Both the superblock and inode structures differ in layout and size between the AIX and System V file systems, yet many of the fields that they contain are identical. The two major differences between the AIX and System V inode structures are:

- an AIX inode contains a 384 byte buffer that it uses to store the data for a small file, and

- an AIX inode stores block numbers as four byte integers instead of the three byte integers.

AIX directories use the same variable-length entries as BSD directories, but the entries are always rounded to the nearest multiple of 16 bytes. Implementing an AIX file system within a framework that already contains components for System V and BSD file systems is greatly simplified through three types of reuse:

- using an existing class unmodified (for example, SVIDBlockAllocator),
- subclassing to encapsulate small differences in data structures (for example, AIXDirectory), and
- designing a new class based on the design of an existing class (for example, AIXContainer).

The AIXContainer class differs from the SVIDContainer class primarily in two ways: first, it encapsulates a slightly different superblock structure, and second, its open and create operations instantiate AIXInodes instead of SVIDInodes. Because of these differences are minor, the SVIDContainer class served as a guide for the implementation of the AIXContainer class.

The AIX free list uses the same data structure as the System V free list; therefore, an AIXContainer can use a SVIDBlockAllocator to manage block allocation and deallocation.

AIXInodes encapsulate AIX inode structures in the same way as SVIDInodes encapsulate System V inode structures, but they also have to add code to the read and write operations to handle files small enough to fit in the inode buffer.

The AIXDirectory is a subclass of BSDDirectory and inherits all operations except associationSize, which rounds the size of each directory entry up to the nearest multiple of 16 instead of 4.

B.1.1.4 Common UNIX Classes

Each of the three UNIX file systems described in this section contains four types of classes: PersistentStoreContainers, BlockAllocators, Files, and PersistentStoreDictionaries. Two file systems share a single class of BlockAllocator(System V and AIX), and two file systems share much of a class of PersistentStoreDictionary(BSD and AIX). But all three file systems have their own classes of PersistentStoreContainers and Files, even though all data structures contain similar information and all classes perform many similar operations. Therefore, I defined two classes that capture the common structure and behavior shared between all three systems: UNIXContainer and UNIXInode.

The UNIXContainer class implements the readInode and writeInode operations, which retrieve, store, and cache blocks of inode structures, for all three of its subclasses. To enable these operations to work even when all subclasses define different sizes for inode structures and possibly different block sizes, the subclasses must implement two simple operations: mapInumber and firstInumber. The mapInumber operation takes an inumber as an argument and returns the block number where the corresponding inode is stored. The firstInumber operation takes an inumber as an argument and returns the inumber of the first inode stored in the same block as the corresponding inode.

The UNIXInode class implements the read, write, copy and mapBlock operations. While all three subclasses inherit the implementations of read and write, two subclasses, BSDInode and AIXInode, overload both operations. These overloaded operations first check to see if the data are stored within the inode itself. If the data are stored within the inode, they handle the request; if not, they invoke the corresponding UNIXInode implementation of the operation. Since all three versions of the UNIX file system allow files to contain empty, unallocated blocks, the UNIXInode class implements the copy operation so that it does not copy empty blocks. To support the mapBlock operation, all subclasses implement operations to store and retrieve direct and indirect block pointers.

B.1.2 The MS-DOS Storage Management System

The MS-DOS file system[Nor85] resembles the UNIX file system in several ways:

- each file can store the data for either a regular file or a directory,
- the only type of PersistentObject supported by regular files is PersistentCharArray, and
- the directories within each PersistentStoreContainer are organized as a tree.

Despite these similarities, the internals of the MS-DOS file system differ fundamentally from UNIX file systems. MS-DOS divides a low-level PersistentStore into four fixed-sized sections: a header, called a boot sector, a File Allocation Table (FAT), a root directory and data blocks containing files and subdirectories.

The boot sector describes the general state of the underlying PersistentStore and includes the following fields: the size and format of the FAT, the size of the root directory, and the number of data blocks. MS-DOS stores most file control information within directory entries instead of inodes. This restricts files to being in exactly one directory.

Directories are arrays of entries with several fields that include: an 11 character file name, the type of the file (either regular or directory), the number of bytes in the file, the time of the last modification, and the number of the first data block in the file. The root directory has a fixed number of entries; there is no limit on the number of entries in subdirectories. As with a UNIX directory, an MS-DOS directory assigns the name "." to itself and the name ".." to its parent directory. Deleted entries are marked by zeroing the first character of the file name.

MS-DOS directory entries have a fixed size; therefore, variable-sized file block mapping information cannot be kept with the other control information. Instead, all block mapping (and block allocation) information for an MS-DOS file store is kept within a single data structure, the FAT. The FAT is an array that has an element, either a 12 or 16 bit integer, for each data block. The FAT stores a zero for each unallocated block, the next block number for each block that belongs to a file, and a negative one for the last block in a file. Thus it stores a chain of block numbers for each file, and files cannot have empty, unallocated blocks. The organization of the FAT also yields poor random access performance for large files.

An MSDOSContainer encapsulates the boot sector and the file description information stored in directory entries. The MSDOSContainer class uses a design similar to the UNIXContainer class. The basicOpen operation instantiates and returns either a File initialized to support an MSDOSDirectory, if the id-number is 1, or an MSDOSStore, if the id-number is greater than 1. The basicCreate operation finds and initializes a free directory entry and then instantiates and returns an MSDOSStore. The basicClose operation checks to see if the file has been deleted, and if it has, it clears the entry.

An MSDOSFAT encapsulates the File Allocation Table. Besides implementing the basicAllocate and basicFree operations, the MSDOSFAT class provides a mapBlock operation, which retrieves the *nth* element from a chain of block numbers, given n and the start of the chain. The MSDOSStore class uses the MSDOSFAT's mapBlock operation to implement its mapBlock operation.

Because the underlying data for the root directory is stored in a contiguous, fixed-size window onto the underlying PersistentStore, its data can be managed by an instance of the File class. The MSDOSStore class abstracts all MS-DOS files other than the root directory; it uses a design similar to the UNIXInode class.

Even though the root directory and subdirectories use different underlying PersistentStore classes, they are both abstracted by the same MSDOSDirectory class. An MSDOSDirectory encapsulates an array of fixed-length directory entries. As with UNIX directories, as entries are added to an MSDOSDirectory, the array grows, but when entries are removed, the array does not shrink. The clearAssociation operation marks an entry as unused by setting the first character of the file name to zero. The findKey operation searches sequentially through the array of entries. If it is also looking for space to add a new key, it will return the location of the first entry that has a file name with a null first character . The associationSize operation returns the size of an entry, which is always 32, regardless of the size of the key to be stored.

B.2 Standard Tool Formats

After successfully implementing the file systems of several standard operating systems, I attempted to build other kinds of file systems. The file formats defined by several standard UNIX tools, including **ar**, **tar**, **ld**, and **mail**, divide the data within a file into several components, organize these components so that they can be conveniently accessed, and provide names for them. Therefore, instead of accessing a file formatted by one of these tools as a PersistentCharArray, applications could access it as a PersistentStoreContainer, if the operating system included a file system customized for the particular file format. The *Choices* file system framework supports the design and implementation of such file systems.

B.2.1 The Ar File System

The UNIX **ar** (archive) command archives a collection of files by combining them into a single file. The format of the archive file includes an archive header, a header for each component file, the contents of each file, and a pad character at the end of the contents of each file that has an odd number of bytes. The archive header only contains a *magic string*. Each 60 character file header includes: a fifteen character file name, the file modification time, ownership information, access permissions, the size of the file, and a magic number. For portability between systems with different byte orders, all information in the header is stored as character strings.

An ArContainer encapsulates the archive header and file headers. The constructor for an ArContainer reads each file header and assigns id-numbers sequentially to each file stored within the archive, starting with the number 2. The root dictionary is assigned id-number 1. The basicOpen operation instantiates and returns either a File initialized to support an ArDictionary, if the id-number is 1, or an ArStore, if the id-number is greater than 1.

The ArStore class abstracts component files stored within an archive. Because each component file is stored in a contiguous, fixed-size window onto the underlying PersistentStore, the ArStore class inherits the read and write operations from the File class. The ArStore class implements operations that retrieve information from a file header, such as info and numberOfRecords. ArStores only support PersistentCharArrays. Since the contents of component files are not aligned on block boundaries, ArStores are examples of Files that buffer reads and writes.

An ArDictionary encapsulates the file names stored in the file headers in its underlying File and maps them to the id-numbers assigned by its corresponding ArContainer. Since the archive file does not allow nested dictionaries, there is only one ArDictionary per ArContainer. Figure B.1 shows an example that uses the Ar File System classes within the framework.

B.2.2 The Tar File System

The UNIX **tar** (tape archive) command also archives a collection of files by combining them into a single file. The format of the tape archive file includes a header for each component file and the contents of each file. Both the header and the contents of each file are aligned on 512-byte block boundaries. Each 512-character file header includes: a 100-character file name, the file modification time, ownership information, access permissions, the size of the file, the type of the file, a checksum, and a 100-character buffer that stores the file's data, if it is a symbolic link. For portability between systems with different byte orders, all information in the header is stored as character strings.

A TarContainer encapsulates the file headers. The constructor for an TarContainer reads each file header and assigns id-numbers sequentially to each file stored within the archive, starting with the number 2. The root dictionary is assigned id-number 1. The basicOpen operation

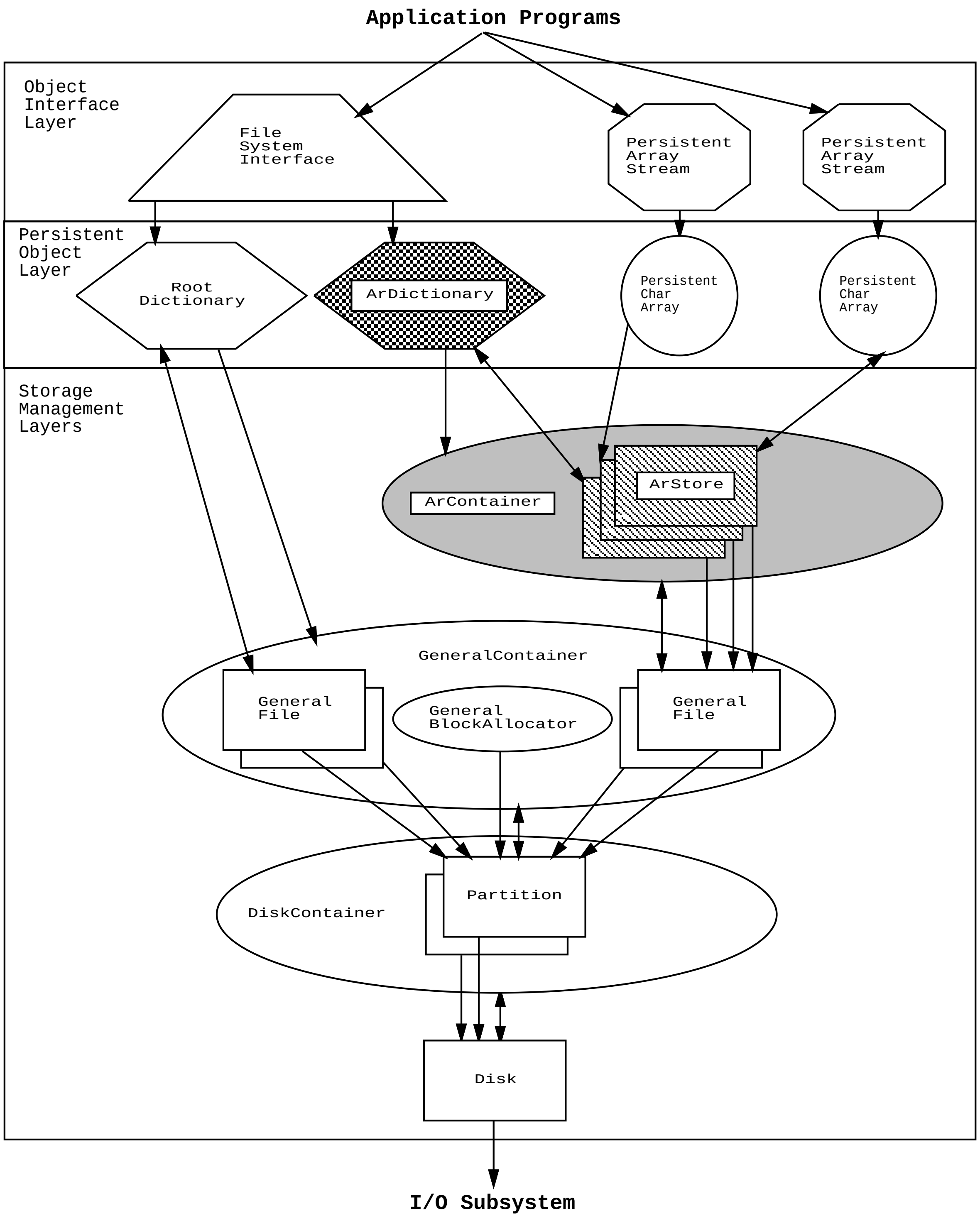

Figure B.1: Using Ar File System Classes within the Framework

instantiates and returns either a File initialized to support an TarDictionary, if the id-number is 1, or an TarStore, if the id-number is greater than 1.

The TarStore class abstracts component files stored within a tape archive. Because each component file is stored in a contiguous, fixed-size window onto the underlying PersistentStore, the TarStore class inherits the read and write operations from the File class. The TarStore class implements operations that retrieve information from a file header, such as info and numberOfRecords. TarStores only support PersistentCharArrays. Since the contents of component files are aligned on block boundaries, TarStores may not need to buffer reads and writes; therefore, retrieving data from a TarStore can be more efficient than retrieving data from an ArStore.

A TarDictionary encapsulates the file names stored in the file headers in its underlying File and maps them to the id-numbers assigned by its corresponding TarContainer. In the current implementation, there is only one TarDictionary per TarContainer.

B.2.3 The Mail File System

The UNIX **mail** command allows users to write, send, receive, read, store, retrieve, and delete electronic mail messages. The **mail** command formats a *mailbox* file into a sequence of variable-length messages. A line that begins with the string "From ", followed by the sender's name and the message's date, marks the beginning of each message. Each message comprises the beginning line, a series of *fields*, and a message *body*. A line that begins with a series of non-blank characters, followed by a colon, marks the beginning of each field. Most fields contain a single line, but fields can contain several lines, all but the first line must begin with a blank character. The message body begins with the first blank line in the message and ends with the end of the message.

A MailContainer encapsulates the structure of a mailbox file. The constructor for a MailContainer reads the entire mailbox and assigns id-numbers to each message, starting with the number 2. The root dictionary is assigned id-number 1. The basicOpen operation instantiates and returns either a MailRootStore, if the id-number is 1, or a File initialized to support a MailMessageContainer, if the id-number greater than 1.

The MailRootStore class defines objects that support MailDictionaries. A MailDictionary also encapsulates the structure of a mailbox file. It maps message numbers given as character

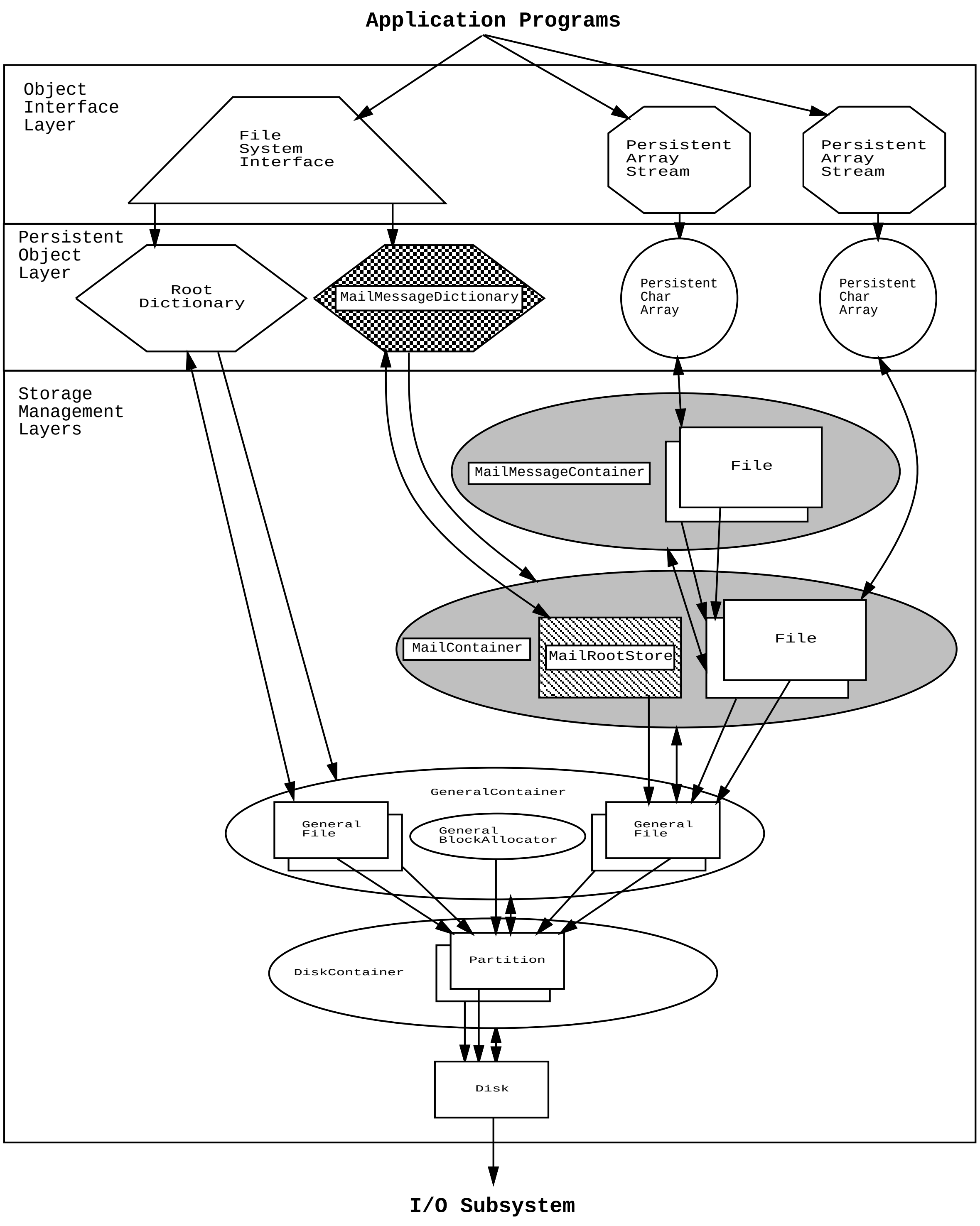

Figure B.2: Using Mail File System Classes within the Framework

strings into the id-numbers assigned by its corresponding MailContainer. (The mapping function converts the character string to an integer and then adds 1 to obtain the id-number.)

A MailMessageContainer encapsulates the structure of an individual message. The constructor for a MailMessageContainer reads the message and assigns id-numbers to each field, starting with the number 2. The root dictionary is assigned id-number 1, and the message body is assigned an id-number that is 2 plus the number of fields. The basicOpen operation instantiates and returns either a File initialized to support a MailMessageDictionary, if the id-number is 1, or a File initialized to support a PersistentCharArray, if the id-number greater than 1.

A MailMessageDictionary also encapsulates the structure of an individual message. It maps field names into the id-numbers assigned by its corresponding MailMessageContainer. Figure B.2 shows an example that uses the Mail File System classes within the framework.

B.2.4 The COFF File System

Several UNIX tools, including the **ld** (link editor for object files) command, use the Common Object File Format[Gir88] for binary object or executable files. The COFF format divides a file into the following sections: a file header, a system header, at least one data section, a symbol table, and a string table for symbols with names longer than eight characters. Each data section includes a header, the data, relocation information, and line numbers for debugging support. The system header, symbol table, and string table are optional. Line numbers within data sections are also optional.

A COFFContainer encapsulates the data structures of a COFF file. Its constructor reads the various headers in the file and assigns id-numbers to all components in the file, starting with the number 2. The root dictionary is assigned id-number 1. The basicOpen operation instantiates and returns either a File initialized to support a COFFDictionary, if the id-number is 1, or a File initialized to support a PersistentCharArray, if the id-number greater than 1.

A COFFDictionary encapsulates the sections names in the COFF file, and also provides names for all other components in the COFF file. Since the COFF format does not have a hierarchical naming structure, there is only one COFFDictionary per COFFContainer. Figure B.3 shows an example that uses the COFF File System classes within the framework.

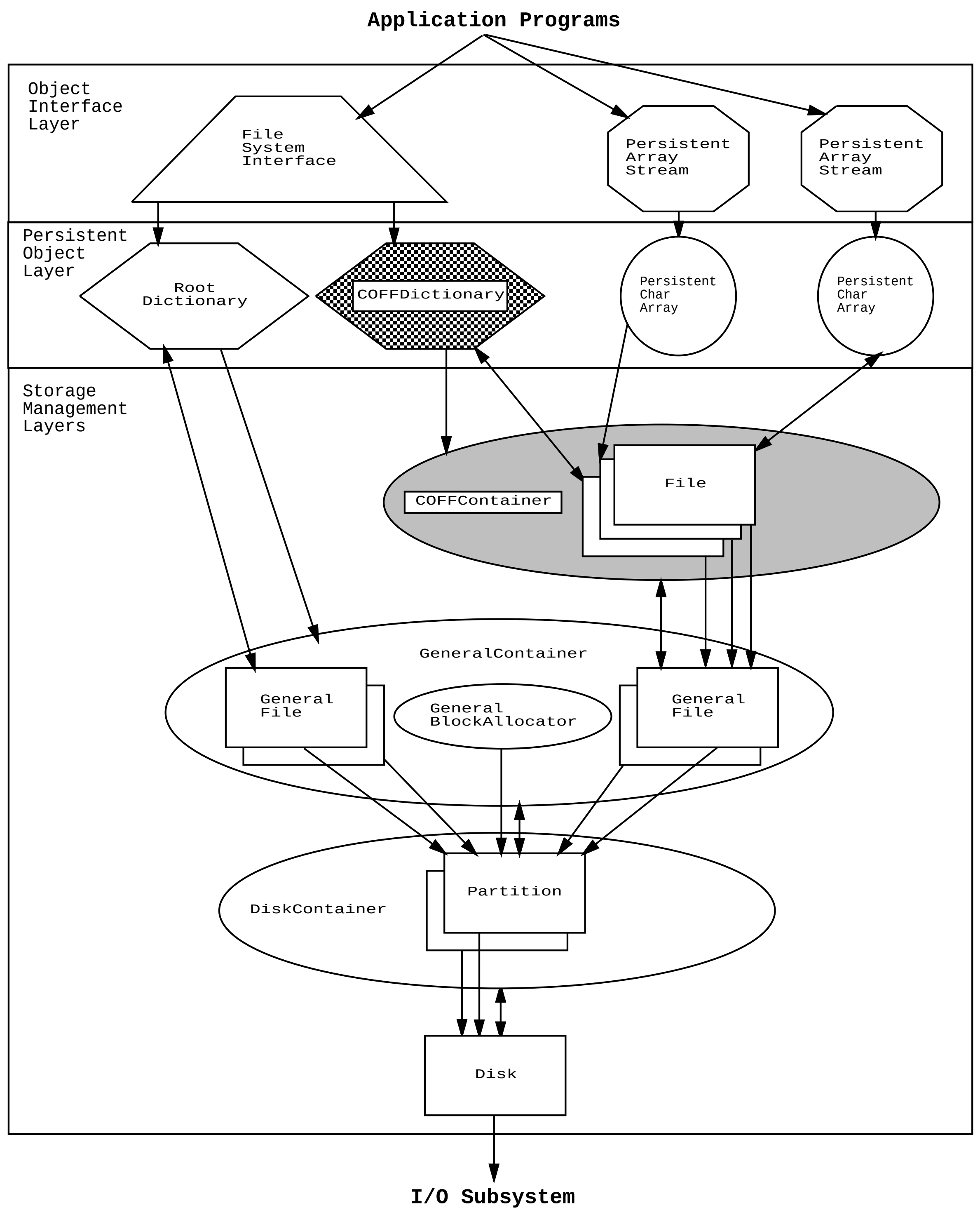

Figure B.3: Using COFF File System Classes within the Framework

B.3 A Remote File System

Besides supporting the construction of file systems that encapsulate the data structures of common systems or tools, the framework supports the construction of file systems that allow applications access to files stored on a remote computer. As an experiment to see how remote file systems fit within the framework, I designed two classes, RemoteStore and RemoteDictionary, that compose a remote file system. They make the files within a PersistentStoreContainer on a remote computer accessible on the local computer.

System or application programs can mount a PersistentStore that is located on another computer system on a local PersistentStore. If the mounted PersistentStore stores the data for the root dictionary of a remote PersistentStoreContainer, all of the contained files will be accessible. Both RemoteStores and RemoteDictionaries represent objects on the remote system, but the remote objects do not have to be active just because they have a corresponding object on the local system. As with Sun Microsystems' NFS file system, the state of the remote server system does not depend on the state of the local client system.

The implementation of the RemoteStore and RemoteDictionary classes differs from the implementation of all other subclasses of PersistentStore and PersistentStoreDictionary; instead of overloading the operations discussed in Chapter 6, they overload the public operations of their superclasses. Each operation sends a message to a server object on the remote system. The message includes an operation code, the id-number of the remote object, and the arguments. The server object invokes the corresponding operation on the remote object and returns the results. Implementing the public operations instead of protected ones has two advantages. First, the internals of the classes on remote systems could be different, but as long as the interface defined by the remote classes is the same, the operations will perform correctly. Second, the public operations could contain critical sections, which can be minimized if they send messages to remote systems. The state of both RemoteStores and RemoteDictionaries consists of the id-number of the corresponding remote object and the socket number used to communicate with the remote computer.

One operation that the RemoteStore class does not overload is asA, which can be inherited directly from PersistentStore, since the RemoteStore class overloads the supports operation. RemoteStores currently can support either a RemoteDictionary or a PersistentCharArray, so they can provide access to all file systems described in Section B.1 and most file systems described in Section B.2. (The **mail** file system uses nested PersistentStoreContainers, which are not currently supported by the remote file system, though such support could be easily added.) By supporting RemoteDictionaries, RemoteStores allow only a single PersistentObject to modify the data of a dictionary; thus they ensure data consistency for dictionaries. By supporting PersistentCharArrays, RemoteStores allow both local and remote PersistentObjects to modify the same underlying data; thus they do not ensure data consistency for regular files. These design decisions reflect the choices made by the designers of NFS.

Besides forwarding arguments to the corresponding dictionary on the remote computer, the open and create operations of RemoteDictionary instantiate a RemoteStore if the result from the remote computer indicates that the operation succeeded.

B.4 A General File System

All file systems that were presented in Sections B.1–B.3 structure files as arrays of bytes. This is natural since these file systems conform to standards defined by systems that all have the same simple file model. But the *Choices* file system framework incorporates three models for structuring the data within files, and, therefore, it needs classes that support record files and user-defined persistent objects. A set of three classes, GeneralContainer, GeneralBlockAllocator, and GeneralFile, provides a general instantiation of the Storage Management Layer, which supports all subclasses of PersistentObject.

Since one of my design goals was to simplify the implementation of the GeneralContainer class while adding support for all kinds of PersistentObjects, I chose to combine the simpler characteristics of both the System V and BSD UNIX file systems with a flexible file typing mechanism. Therefore, the general file system uses bit maps to store the state of both inode and block allocation, but it neither divides a disk into cylinder groups nor fragments data blocks for files. Like the System V file system, the general file system divides each disk or partition into three sections: a header, a fixed-size array of file descriptors, and data blocks, which can be unallocated, hold file data, or hold block mapping information.

The header information contains a superblock, a file descriptor allocation bit map, a block allocation bit map, and a list of class names. The superblock includes the following fields: a magic number, a clustering factor, the size of each file descriptor, the starting block numbers of both bit maps, the starting block number of the class name list, the starting block number of the file descriptors, the starting block number of the data blocks, the total number of file descriptors, the number of free file descriptors, the total number of blocks, and the number of free blocks. The class name list contains the names of the subclasses of PersistentObject that are supported by the contained Files.

Since there is no inherent limit on the length of class names, storing one in the fixed-size file descriptor is unacceptable. Instead, one buffer is used to store the names of all classes of PersistentObjects that are associated with the Files. 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 individual class names),
- if several files support the same class, the name only needs to be stored once, and
- each file descriptor needs to store only an index (which requires only one integer) into its container's list of class names.

Each file descriptor is a fixed-size data structure that resembles a System V inode, except that it stores block numbers the way BSD and AIX inodes do, as four-byte integers. Instead of hard-coding the type of the file using a few bits, the type of the file is stored as a one-integer index into the class name list. File descriptors also differ from UNIX inodes because they store a record size (for record-structured files and persistent arrays that have an element size other than one byte) and they do not store a reference (link) count. The general file system can store persistent objects that have arbitrary inter-object references; therefore, reference-counting is inadequate for reclaiming unreachable file descriptors. Thus, link counts are unnecessary.

The general file system does not define its own class of dictionary, since its Files can support any subclass of PersistentObject. Any set of subclasses of PersistentStoreDictionary can be used to name Files.

The GeneralContainer class encapsulates the class name list, the file descriptor array, and the fields of the superblock that are unrelated to block allocation. The basicOpen operation checks to ensure that the file descriptor that corresponds to the given id-number has a non-null class name index and then instantiates a GeneralFile with the appropriate arguments and returns a reference to it. The basicCreate operation searches the free file descriptor bit map. If it finds a free descriptor, it initializes it and then instantiates a GeneralFile with the appropriate arguments and returns a reference to it. The basicClose deletes the instance of GeneralFile, but it cannot make unreachable file descriptors available for reallocation. Instead, a separate garbage collection process is needed. The basicSynchronize operation writes any modified superblock or file descriptor information to the underlying PersistentStore. The basicRootId operation returns 1, the inumber of the root directory.

The GeneralContainer constructor creates a GeneralBlockAllocator, which encapsulates the block allocation bit map and implements the basicAllocate and basicFree operations.

A GeneralFile, which is a kind of UNIXInode, encapsulates an individual file descriptor. It performs the same functions as the SVIDInode, but instead of supporting just one kind of PersistentStoreDictionary or PersistentArray, it supports any kind of PersistentObject. The PersistentObjects presented in Appendix C are examples of objects that require the flexible type support of GeneralFiles.

B.5 Summary

This appendix describes sets of concrete classes that implement four different kinds of storage management and naming for file systems. These include the storage management and naming of several traditional stream-oriented file systems, file systems customized to support the data format of standard UNIX tools, a remote file system, and a general file system that supports all kinds of PersistentObjects. These sets of classes illustrate many of the benefits of the framework.

One benefit is various types of code reuse. Many storage management subsystems reuse classes without any modification. Examples include the File class used by all the systems in Section B.2, and the SVIDBlockAllocator class used by the AIX file system. Some classes inherit major operations from a common superclass and define simple operations that encapsulate minor differences in data structures. Examples of common superclasses are the UNIXInode and UNIXContainer classes. Almost all concrete classes share much of their design that they do not inherit with other concrete classes. Examples include all subclasses of the UNIXInode and UNIXContainer classes and all the container classes described in Section B.2.

Though much code and design is reused by the concrete classes described here, interface reuse is even more important. All subclasses of the PersistentObject class use the interface defined by the PersistentStore class, and the FileSystemInterface class uses the interfaces defined by the PersistentStoreContainer and PersistentStoreDictionary classes. Therefore, as long as the classes discussed in this appendix implement the operations defined by their abstract superclasses, the rest of the classes in the framework will be able to use them without modification.

Appendix C

Using a Persistent Object Store

To experiment with the programming of user-defined persistent objects, I decided to try to solve a well-known, non-trivial problem while using them instead of standard C++ objects. Therefore, I designed a set of classes that implement a simple programming language. I also decided to concentrate on the semantics of the language, not the syntax. Instead of using a parser, users of the language must create objects that represent a parse tree for the program and then invoke the execute operation on the root of the tree. The language, called Persistent Object Language (POL), defines eight different kinds of parse nodes, called POLExpressions:

- function applications,
- builtin functions,
- expression lists,
- if expressions,
- integers,
- lambdas (or function definitions),
- strings,
- and variables.

C.1 The POLExpression Class

The classes that define the various types of POLExpressions are arranged in the hierarchy shown in Figure C.1. At run-time, POL programs use nested PersistentStoreDictionaries as their stack of *activation records.*[1]

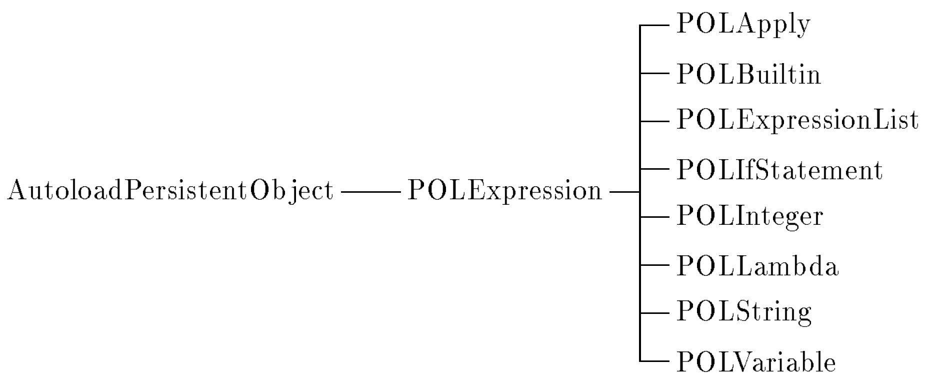

Figure C.1: Persistent Object Language Class Hierarchy

The POLExpression class defines the protocol (set of operations) that instances of subclasses use to communicate with each other. Figures C.4–C.12 contain simplified versions of the C++ declarations for POLExpression and its subclasses.

The init operation allows objects to initialize their persistent data. All subclasses provide their own implementations of init. The FileSystemInterface argument provides the environment in which objects can look up symbolic names of other objects. Subclasses that define instance variables that are references to other POLExpressions must also implement the flush operation.

The eval operation is intended to evaluate the portion of the parse tree rooted at the node on which it is invoked. It takes a FileSystemInterface as an argument and returns a reference to a POLExpression and an error code. The current directory of the FileSystemInterface represents the current activation record. Most subclass implementations recursively invoke eval on the nodes to which they refer, perform some calculation using the results of these recursive invocations, and then return the result of the calculation. Subclasses that abstract primitive objects, like

[1]See [ASU86] or [ASS85] for background on parse trees, activation records, and other programming language topics.

integers or strings, inherit POLExpression's implementation of eval, which just returns the object itself.

The printOn operation requests that an object print a human-readable representation of itself on the given OutputStream. All subclasses provide their own implementations of printOn.

Builtin boolean functions can invoke the isTrue operation on objects. The default implementation of isTrue returns true for all instances. Builtin arithmetic functions can invoke the asInt operation on objects. The default implementation of asInt returns 0 for all instances.

C.2 Subclasses of POLExpression

The POLInteger class abstracts a primitive POLExpression that stores an integer value. Its implementation of the asInt operation returns the value that it stores, and its implementation of the isTrue operation returns true if the value it stores is non-zero, otherwise it returns false. For better performance, POLInteger provides a setValue operation, which sets the stored value to the given argument and is more efficient than the init operation.

The POLString class abstracts a primitive POLExpression that stores a character string.

The POLExpressionList class abstracts lists of POLExpressions, which can be used to implement sequentially evaluated program expressions. Each POLExpressionList can store references to two POLExpressions: the first element of the list (`_car`) and the rest of the list (`_cdr`). The eval operation recursively evaluates `_car` and then `_cdr`, and it returns the result of evaluating `_cdr`.

The POLIfExpression class abstracts conditional evaluation of expressions. Each POLIf-Expression stores references to three POLExpressions. The eval operation first evaluates the `_condition` expression. If the condition evaluates to `true`, the operation returns the result of evaluating the `_true` expression; otherwise, it returns the result of evaluating the `_false` expression.

The POLApply class abstracts the application of functions to arguments. Each POLApply object stores the number of arguments and an array and references to the arguments. The eval operation performs the following steps:

1. invokes eval on each argument,
2. creates a dictionary in the current dictionary called "environment,"

3. changes the current dictionary to be the newly created one,
4. binds each argument to a name that is the character string representation of the argument number,
5. invokes eval on the function,
6. removes each arguments name from the current dictionary,
7. changes the current dictionary to be the parent,
8. removes the dictionary called “environment,” and
9. returns the result of invoking eval on the function.

The POLLambda class abstracts POL function definitions. Each POLLambda combines zero or more formal parameters with some POL code. The names of the formal parameters are stored as character strings, and the code is stored as a reference to a POLExpression. The eval operation performs the following steps:

1. rebinds each numerically named argument to its formal parameter name,
2. invokes eval on the code POLExpression,
3. removes each formal parameter name for the current dictionary,
4. returns the result of invoking eval on the code.

The POLBuiltin class abstracts several types of builtin functions: unary and binary arithmetic, unary and binary boolean, and output (with a variable number of arguments). (Many of these functions could be built from a small set of primitives, but then POL programs would run even slower.) The init operation stores the name of the builtin functions, which is usually the same as C++ operator used for that function (for example, `+, -, *, and /` for add, subtract, multiply, and divide). POLBuiltin uses names from the Pascal language for output functions (`write and writeln` instead of `printf`). The eval operation uses numerically named arguments instead of formal parameters. It performs the function that is indicated by the `_name` variable on these arguments.

The POLVariable class abstracts late bindings of object names to objects. A POLVariable can refer to another POLExpression by storing its name instead of directly referring to it. Instances of all other subclasses of POLExpression that refer to POLExpressions store direct references to those POLExpressions. When defining recursive functions in POL, one must refer to objects that have not yet been created. POLVariables solve the problem of "forward references," by deferring the binding of the reference until after the object has been created. The eval operation retrieves the current dictionary from the given FileSystemInterface. It then invokes the open operation on the dictionary, passing it the variable name. If the object is found in the dictionary, eval returns it; otherwise, it looks for the name in the parent dictionary. If needed, it will repeat this process until it reaches the root dictionary. If the name is not found in any dictionary from the current through the root dictionary, eval returns an error code.

```
> mkpo POLInteger 0 0
> mkpo POLInteger 1 1
> mkpo POLInteger 7 7
> mkpo POLBuiltin - -
> mkpo POLBuiltin "*" "*"
> mkpo POLBuiltin > >
> mkpo POLVariable N n
> mkpo POLVariable F factorial
> mkpo POLApply comparison > N 0
> mkpo POLApply subtraction - N 1
> mkpo POLApply recursion F subtraction
> mkpo POLApply multiplication "*" N recursion
> mkpo POLIfExpression condition comparison multiplication 1
> mkpo POLLambda factorial condition n
> mkpo POLApply factorialOf7 factorial 7
> factorialOf7
5040
>
```

Figure C.2: Example of User-defined Persistent Objects.

C.3 A Sample Program

The *Choices* system shell supports the **mkpo** command, which takes two or more arguments, creates a PersistentObject of the class named in the first argument, and binds it to the file named in the second argument. The **mkpo** command sends the rest of the arguments to the init operation of the newly created object. Then the object is automatically stored in the file system.

A sample shell program, shown in Figure C.2, demonstrates how one can create a parse tree of POLExpressions that represents the application of the factorial function to the number 7. Figure C.3 shows the parse tree. Each node in the tree contains the class and name of the object. For objects that contain a value in their instance data, the value is given. Inter-object references are represented as arrows from the object containing the reference to the object to which it refers.

When the shell parses the name of a command, it checks to see if the object is a binary executable, a shell script, or an executable persistent object. If it is an executable persistent object, the shell invokes the object's execute operation. In the case of these persistent language expressions, they will invoke the necessary methods on each other, with objects being loaded into memory and written back to disk automatically. After all operations have been performed, the root of the parse tree returns the correct answer.

C.4 Summary

This appendix demonstrates the flexibility and extensibility of the *Choices* file system framework. It supports not only the traditional features of file systems, such as data storage and retrieval, but also the automatic retrieval and execution of persistent objects.

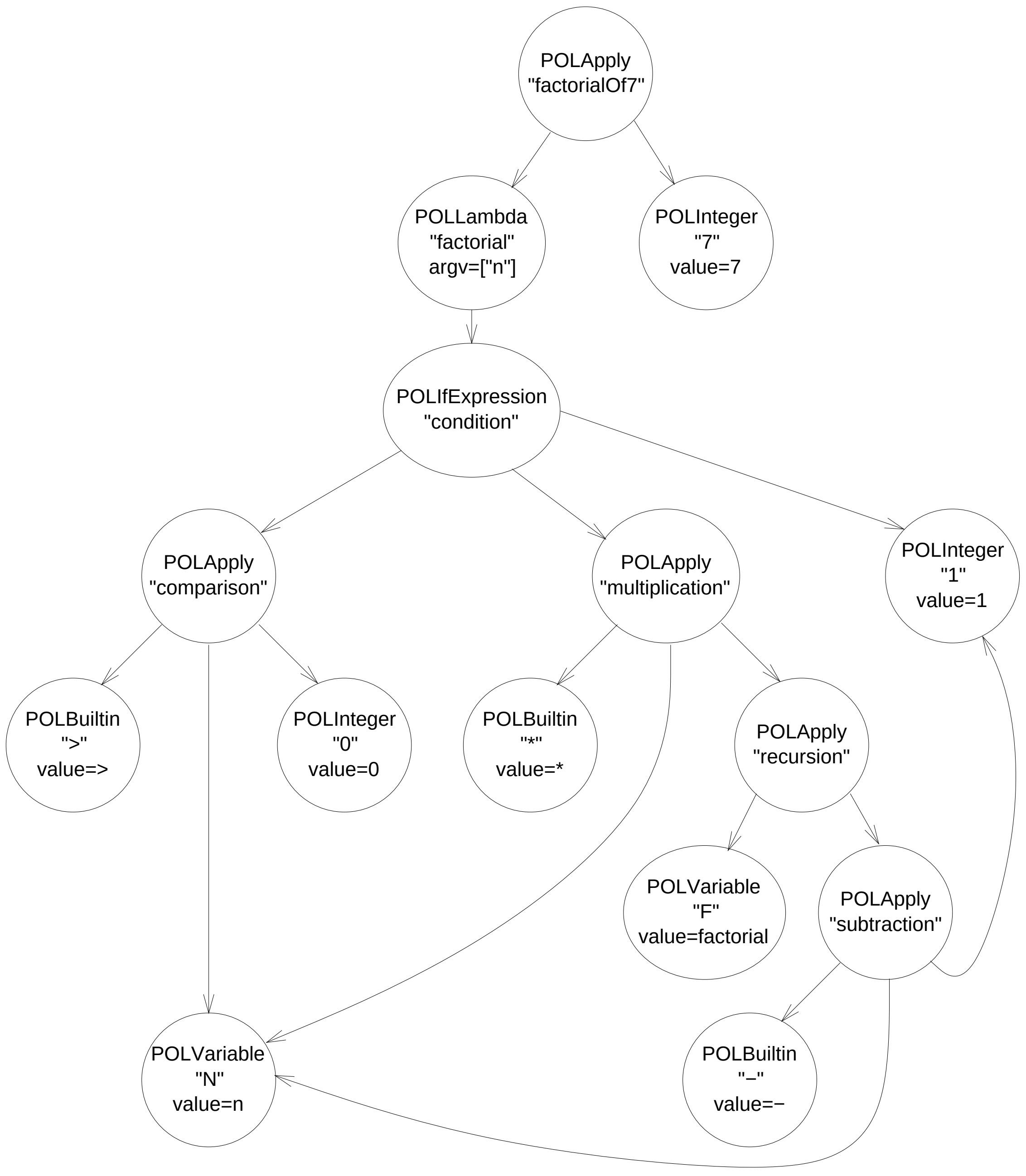

Figure C.3: Persistent objects in a parse tree for `factorial(7)`

C.5 Class Declarations

This section contains the simplified versions of the C++ class declarations for POLExpression and its subclasses.

```
enum POLErrorCode {
      POLSuccess = 0,
      ApplyNotInitialized,
      ApplyArgumentEvaluationFailed,
      IfNotInitialized,
      NewObjectFailed,
      ObjectOpenFailed,
      SubclassResponsibility,
      TooFewArguments,
      TooManyArguments,
      UnknownBuiltinBinaryFunction,
      UnknownBuiltinNaryFunction,
      UnknownBuiltinUnaryFunction
};

class POLExpression : public AutoloadPersistentObject {
public:
      POLExpression( PersistentMemoryObject * mo );
      int init( int argc, char *argv[], FileSystemInterface * );

      int execute();

      int asInt();
      int isTrue();

      void printOn( OutputStream * );
      POLExpression * eval( FileSystemInterface *, POLErrorCode & );

protected:
      POLExpression * convertArg( int argn, char * argv[], int & status,
                                  FileSystemInterface * fsi );
};
```

Figure C.4: Class Declaration: POLExpression

```
class POLInteger : public POLExpression {
public:
        POLInteger( PersistentMemoryObject * mo );
        int init( int argc, char *argv[], FileSystemInterface * );

        int isTrue();
        int asInt();
        void setValue( int value );
        void printOn( OutputStream * );

protected:
        int _value;
};
```

Figure C.5: Class Declaration: POLInteger

```
class POLString : public POLExpression {
public:
        POLString( PersistentMemoryObject * mo );
        int init( int argc, char *argv[], FileSystemInterface * );

        void printOn( OutputStream * );

protected:
        char _value[232];
};
```

Figure C.6: Class Declaration: POLString

```
class POLBuiltin : public POLExpression {
public:
        POLBuiltin( PersistentMemoryObject * mo );
        int init( int argc, char *argv[], FileSystemInterface * );

        void printOn( OutputStream * );
        POLExpression * eval( FileSystemInterface *, POLErrorCode & );

protected:
        char _name[40];
};
```

Figure C.7: Class Declaration: POLBuiltin

```
class POLExpressionList : public POLExpression {
public:
        POLExpressionList( PersistentMemoryObject * mo );
        int init( int argc, char *argv[], FileSystemInterface * );

        void printOn( OutputStream * );
        POLExpression * eval( FileSystemInterface *, POLErrorCode & );

        void flush();

protected:
        POLExpression * _car;
        POLExpression * _cdr;
};
```

Figure C.8: Class Declaration: POLExpressionList

```
class POLIfExpression : public POLExpression {
public:
       POLIfExpression( PersistentMemoryObject * mo );
       int init( int argc, char *argv[], FileSystemInterface * );

       void printOn( OutputStream * );
       POLExpression * eval( FileSystemInterface *, POLErrorCode & );

       void flush();

protected:
       POLExpression * _condition;
       POLExpression * _true;
       POLExpression * _false;
};
```

Figure C.9: Class Declaration: POLIfExpression

```
class POLLambda : public POLExpression {
public:
        POLLambda( PersistentMemoryObject * mo );
        int init( int argc, char *argv[], FileSystemInterface * );

        void printOn( OutputStream * );
        POLExpression * eval( FileSystemInterface *, POLErrorCode & );

        void flush();

protected:
        POLExpression * _code;
        int             _argc;
        char            _argv[7][32];
};
```

Figure C.10: Class Declaration: POLLambda

```
class POLApply : public POLExpression {
public:
        POLApply( PersistentMemoryObject * mo );
        int init( int argc, char *argv[], FileSystemInterface * );

        void printOn( OutputStream * );
        POLExpression * eval( FileSystemInterface *, POLErrorCode & );

        void flush();

protected:
        int             _argc;
        POLExpression * _argv[7];
};
```

Figure C.11: Class Declaration: POLApply

```
class POLVariable : public POLExpression {
public:
        POLVariable( PersistentMemoryObject * mo );
        int init( int argc, char *argv[], FileSystemInterface * );

        void printOn( OutputStream * );
        POLExpression * eval( FileSystemInterface *, POLErrorCode & );

protected:
        char _value[232];
};
```

Figure C.12: Class Declaration: POLVariable

Bibliography

[ABCM83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm W.P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, 1983.

[AG89a] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data*, pages 36–45, Portland, Oregon, May-June 1989.

[AG89b] R. Agrawal and N. H. Gehani. Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++. In *Proceedings of Second International Workshop on Database Programming Languages*, Oregon Coast, June 1989.

[AH87] Timothy Andrews and Craig Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In *Proceedings of OOPSLA '87*, pages 430–440, Orlando, Florida, October 1987.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[ATT85] ATT. *UNIX Time Sharing System Programmer's Manual, Eight Edition.* AT&T Bell Laboratories, Murray Hill, New Jersey, 1985.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[BBR+89] D.S. Batory, J.R. Barnett, J. Roy, B.C. Twichell, and J. Garza. Construction of File Management Systems from Software Components. In *COMPSAC 1989*, 1989.

[Bii88] BiiN. BiiN Operating System Technical Overview. Technical report, BiiN, 1988.

[BMO+89] 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.

[Bro89] Alfred Leonard Brown. Persistent object stores. Technical Report Persistent Programming Report 71, Universities of St Andrews and Glasgow, October 1989.

[BS88] Lubomir Bic and Alan C. Shaw. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[CDRS89] 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.

[CKH+89] Arunodaya Chatterjee, Arjun Khanna, Ying Hung, Russell McLaren, Sudha Narayanaswamy, and Nandini Ajmani. Es-kit: A Distributed, Object-Oriented Operating System. Technical Report ACT-ESP-374-89, Microelectronics and Computer Technology Corporation, October 1989.

[CRJ87] 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.

[CSG+88] Ellis S. Cohen, Dilip A. Soni, Raimund Gluecker, William M. Hasling, Robert W. Schwanke, and Michael E. Wagner. Version Management in Gypsy. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 201–215, Boston, Massachusetts, November 1988.

[Dei84] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley, Reading, Massachusetts, 1984.

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

[Deu89] L. Peter Deutsch. Design Reuse and Frameworks in the Smalltalk-80 Programming System. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume II, pages 55–71. ACM Press, 1989.

[Dig86] Digitalk. *Smalltalk/V Tutorial and Programming Handbook.* Digitalk Inc., 9841 Airport Boulevard, Los Angeles, California 90045, 1986.

[DS89] Peter Dibble and Michael Scott. Beyond Striping: The Bridge Multiprocessor File System. *Computer Architecture News*, 17(5):32–39, September 1989.

[FAC$^+$89] D.H. Fishman, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, and W.K. Wilkinson. Overview of the Iris DBMS. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 219–250. Addison-Wesley, Reading, Massachusetts, 1989.

[Flo88] G. H. Florijn. A Technical View of the Portable Common Tool Environment. Technical Report RP/dvm-88/8, Software Engineering Research Centrum, Utracht, The Netherlands, November 1988.

[Gir88] Gintaras R. Gircys. *Understanding and Using COFF.* O'Reilly and Associates, Inc., Newton, Massachusetts, 1988.

[GNS88] David K. Gifford, Roger M. Needham, and Michail D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, March 1988.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, Massachusetts, 1984.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[GWM89] Erich Gamma, André Weinand, and Rudolf Marty. Integration of a Programming Environment into ET++: A Case Study. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 283–298, Nottingham, UK, July 1989. Cambridge University Press.

[HKM$^+$88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[HS82] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures.* Computer Science Press, Rockville, Maryland 20850, 1982.

[Hug79] Joan K. Hughes. *PL/I Structured Programming.* John Wiley and Sons, New York, New York, 1979.

[IC91] 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.

[JC89] Gary M. Johnston and Roy H. Campbell. An Object-Oriented Implementation of Distributed Virtual Memory. In *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pages 39–57, Ft. Lauderdale, Florida, October 1989.

[JR91] 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.

[JW75] Kathleen Jensen and Niklaus Wirth. *Pascal: User Manual and Report.* Springer-Verlag, New York, New York, 1975.

[KBC$^+$89] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. Features of the ORION Object-Oriented Database. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 251–282. Addison-Wesley, Reading, Massachusetts, 1989.

[Kil86] T.J. Killian. Processes as Files. In *Proceedings of the Summer 1984 USENIX Conference*, pages 203–207, 1986.

[Kou91] Panos Kougiouris. Devices Drivers for an Object-Oriented Operating System. Master's thesis, University of Illinois at Urbana-Champaign, October 1991.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[Mac89] International Business Machines. Distributed Data Management, Level 2.0 Architecture, Reference. Technical Report SC21-9526-2, International Business Machines Corporation, January 1989.

[MCK91] Peter W. Madany, Roy H. Campbell, and Panos Kougiouris. Experiences Building an Object-Oriented System in C++. In *Technology of Object-Oriented Languages and Systems Conference*, Paris, France, March 1991.

[MCRL89] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 311–328, Nottingham, UK, July 1989. Cambridge University Press.

[MG89] 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.

[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*, pages 65–79, Denver, Colorado, October 1988.

[Nie89] Oscar Nierstrasz. A Survey of Object-Oriented Concepts. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 3–22. Addison-Wesley, Reading, Massachusetts, 1989.

[Nor85] Peter Norton. *The Peter Norton Programmer's Guide to the IBM PC.* Microsoft Press, 10700 Northrup Way, Box 97200, Bellevue, Washington 98009, 1985.

[NWO88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[OD89] John Ousterhout and Fred Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. *Operating Systems Review*, 23(1):11–28, January 1989.

[OJ90] William F. Opdyke and Ralph E. Johnson. Refactoring: an Aid in Designing Application Frameworks and Evolving Object-Oriented System. In James TenEyck, editor, *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 145–160, Nottingham, UK, September 1990.

[Org87] International Standards Organization. Information Processing Systems — Open Systems Interconnection — File Transfer, Access and Management. Technical Report ISO 8751-1, International Standards Organization, September 1987.

[PD88] D. V. Pitts and P. Dasgupta. Object Memory and Storage Management in the Clouds Kernel. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 10–17. IEEE, June 1988.

[PJC+90] Fred Pollack, Dave Johnson, Dave Carson, Ron Ebrsole, Vittal Kini, Konrad Lai, Bernie Silvernail, and Steve Stacey. A VLSI-Intensive Fault Tolerant Computer Architecture. In *Proceedings of IEE Spring 1990 Computer Conference*, February 1990.

[PKD+90] Fred Pollack, Kevin Kahn, T. Don Dennis, Gerald Holzhammer, Herman D'Hooge, and Steve Tolopka. An Object-Oriented Distributed Operating System. In *Proceedings of IEE Spring 1990 Computer Conference*, February 1990.

[PLNZ88] C. Brian Pinkerton, Edward D. Lazowska, David Notkin, and John Zahorjan. A Heterogeneous Remote File System. Technical Report 88-08-08, University of Washington, Seattle, Washington, August 1988.

[PS85] James L. Peterson and Abraham Silberschatz. *Operating System Concepts.* Addison-Wesley, Reading, Massachusetts, 1985.

[RC89] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89*, pages 267–278, New Orleans, Louisiana, September 1989.

[RCS89] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The Design of the E Programming Language. Technical Report 824, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, February 1989.

[RFH$^+$86] A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, and K. Yueh. RFS Architectural Overview. In *Proceedings of the Summer 1986 USENIX Conference*, pages 248–259, Atlanta, Georgia, 1986.

[RJC88] Vincent Russo, Gary Johnston, and Roy Campbell. Process management and exception handling in multiprocessor operating systems using object-oriented design techniques. In *Proceedings of OOPSLA '88*, 1988.

[Rus91] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, January 1991.

[Sam69] Jean E. Sammet. *PROGRAMMING LANGUAGES: History and Fundamentals.* Prentice Hall, Englewood Cliffs, New Jersey, 1969.

[SGK$^+$85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, Portland, Oregon, June 1985.

[SGM89] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and Migration for C++ Objects. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 191–204, Nottingham, UK, July 1989. Cambridge University Press.

[Sha90] Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, pages 15–24, November 1990.

[Sha91] Jay Shah. *VAX/VMS: Concepts and Facilities*. McGraw Hill, New York, New York, 1991.

[SS79] Nancy B. Stern and Robert A. Stern. *Structured COBOL Programming*. John Wiley and Sons, New York, New York, 1979.

[Ste87] Lynn Andrea Stein. Delegation is Inheritance. In *Proceedings of OOPSLA '87*, pages 138–146, Orlando, Florida, October 1987.

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

[Tan86] Andrew S. Tanenbaum. *OPERATING SYSTEMS: Design and Implementation*. Prentice Hall, Englewood Cliffs, New Jersey, 1986.

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

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA '87*, pages 227–242, Orlando, Florida, October 1987.

[Vli90] John M. Vlissides. *Generalized Graphical Object Editing*. PhD thesis, Stanford University, June 1990.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying Current Research in Object-Oriented Programming. *Communications of the ACM*, 33(9):104–124, September 1990.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of OOPSLA '87*, pages 168–183, Orlando, Florida, October 1987.

[WPE$^+$83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS Distributed Operating System. *ACM Operating Systems Review*, 17(5):49–69, October 1983.

[ZJ90] Jonathan Zweig and Ralph E. Johnson. The conduit: A communication abstraction in C++. In *Proceedings of the USENIX C++ Conference*, pages 191–204, San Francisco, California, April 1990.

Vita

Peter William Madany was born in 1961 and raised in South Holland, Illinois. He received his B.A. in Math/Computer Science and Chemistry from Trinity Christian College in 1982.

He worked for IIT Research Institute in Chicago, Illinois from 1980 until 1984. While working at IITRI, Peter earned his M.S. in Computer Science from Illinois Institute of Technology in 1984. During the first half of 1985, he completed two text retrieval projects for I.S. Grupe, Inc. in Lombard, Illinois. Then Peter joined AT&T Bell Laboratories in Naperville, Illinois, where he worked on an experimental data network controller until he enrolled in the Ph.D. program at the University of Illinois in 1988.

At the University of Illinois, Peter worked as a research assistant for Roy Campbell on the *Choices* operating system project. He also worked as an independent computer consultant.

Upon completion of his doctoral studies, he accepted a position as a Staff Engineer at Sun Microsystems Laboratories in Mountain View, California in 1991.