Department of Computer Science

Ivar Bergman
Per-Ola Gustafsson

User defined services in file systems: A case study and prototype implementation

---

D-level Dissertation

2004:03

User defined services in file systems: A case study and prototype implementation

Ivar Bergman
Per-Ola Gustafsson

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Ivar Bergman

Per-Ola Gustafsson

Approved, 2004-06-10

Opponent: Martin Blom

Advisor: Thijs Holleboom

Examiner: Donald Ross

Abstract

File systems are traditionally implemented as an isolated part of the operating system. File system services, such as on disk storage of data and journaling, are predefined by file system vendors. Because of this, it is difficult and sometimes impossible to introduce user defined services to a file system. This has caused application developers to implement their own filing services within their applications, at the expense of reusability and development time.

This thesis presents file system design techniques that allow file systems to be extended by user defined services. The design techniques are examplified by two file systems, WinFS and Reiser4. As a part of the thesis we have designed and implemented a file system prototype. This prototype, together with WinFS and Reiser4, has been used as a base for a survey that concentrates on the following topics of extendable file systems: file operations, dynamic loading of operations, file system operations and file system name spaces.

Contents

List of Figures

Chapter 1

Introduction

The file system is the most visible aspect of an operating system. It provides the mechanism for persistent storage of and access to information belonging to the operating system and all the users of the system.

Even though the file system is the most visible aspect of an operating system and most end users regularly utilize services provided by the file system, development of new services has not been as rapid as in application software development. One major reason for this is that it is difficult and sometimes impossible for a developer to add new services to the file system. This has caused application developers, such as database builders, to develop their own filing services inside their applications, at the expense of development time and reusability[5].

This thesis presents two different design techniques used to introduce new services in a file system. The main difference between the two design techniques is their view of file abstraction. One of the design techniques preserves the traditional abstract view of a file as a sequence of bytes with a static set of operations. The other design technique allows services to extend the set of file operations.

As case studies, the file systems Reiser4 and WinFS have been used to illustrate the two design techniques. Reiser4 is an example of a file system with a static set of operations,

while WinFS exemplify a file system that can introduce new file operations.

Both file systems are still in early development. The thesis also includes a chapter that describes the design of a file system prototype. This prototype is used to illustrate one solution of how to manage files of different types.

The prototype, together with WinFS and Reiser4, is used as a base for a general discussion of extendable file system. The discussion is concentrated on the following topics:

- File operations
- Dynamic loading of operations
- File system operations
- File system name spaces

1.1 Outline

Chapter 2 presents an overview of the development of file systems, from the early Multics system to todays file systems used in modern operating systems.

Chapter 3 distinguishes four different aspects of a file system: files, naming, storage and file system services, and discuss how file systems can be extended with new services.

Chapter 4 presents the two design techniques, exemplified by a presentation of the two file systems Reiser4 and WinFS. The goal is highlight features in each file system that relates to file systems extension, not to provide a complete description of the file systems.

Chapter 5 discusses the design of the file system prototype that is able to handle files with different sets of attributes and operations. The prototype, together with WinFS and Reiser4, is used as a base for a general discussion on extendable file systems.

Chapter 6 presents a conclusions of the thesis.

Chapter 2

A historical overview of file systems evolution

This chapter presents a historical overview of file systems evolution. Contemporary file systems have many fundamental concepts that can be traced back to file systems in early versions of UNIX and its ancestors, even though the abstract view of files and file systems has not changed much since 1970, file systems vendors have introduced new services to improving security, reliability and performance issues.

There is no single definition of a file system shared by all file system vendors, but still all modern file system share some fundamental concept of files.

2.1 Multics

Multics was initially described in six papers presented at the *1965 Fall Joint Computer Conference*[7]. The Multics operating system was targeting large main frames, and in 1969 it was opened for customers, and was in production until 2000, when the last Multics system was closed.

Multics was a multi-user system, where terminals were anonymous remote. Since a

Multics main frame allowed multiple users to log on, the system needed to handle privacy and security on each entry in the file system. Multics was the first system to provide a mandatory Access Control Lists (ACLs) on every entry to control file access. It was also the first system that structured files in a hierarchical tree structure.

The file system was also designed with the presumption that there would be file system errors (disk failure or incorrect usage), so an automatic file backup mechanism was provided by the system rather then by individual user. In addition, the Multics file system supported:

- More flexible file naming by allowing arbitrary long symbolic names on files.
- Symbolic links to files and directories.
- Storage quotas to control disk usage for individual system users.
- Volume management of removable devices, for example backup tapes.

2.2 UNIX

In 1969, a development team from Bell Labs left the Multics project since it had failed to produce the timesharing system Bell Labs wanted. K. Thompson, R.H Canaday and D. M. Ritche started to sketch on a file system that would later constitute the heart of the UNIX operating system[12]. In the time span between 1969 and 1974 the system grew into an established operating system research project at Bell Labs.

The team realized early, that the file system was the most important part of the UNIX operating system. Many ideas were inherited from Multics, but they clearly improved the concept of file abstraction by identifying that, from a user perspective, there are at least three different types of files:

- *Ordinary files* contain whatever information the user places on it, for example, symbolic or binary programs. No particular structuring is imposed by the system.

- *Directories* provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole.

- *Special files* are used to access devices (mostly hardware devices) on a machine. Special file support read and possible write operations, like ordinary files. But an execution of an operation results in activation of an associated device.

It is notable that the file systems internal structure in the earlier version of UNIX is almost identical to today's systems. It is based on an *inode* record containing: protection mode, size, type of the file and a list of the physical blocks holding the contents. A directory is defined by an inode where the file content is a sequence of names and its associated inode number. Special files do not contain device information explicitly in the inode; instead it is encoded in the inode as a number associated with the device.

The file system structure and concepts defined by the early Multics and UNIX operating system have clearly contributed to the way we define a file system. Even today, most file systems define a file as a sequence of bytes, and the system should not impose a structure on file data. Also, the files are organized in a tree structured hierarchy of directories.

2.3 Virtual file system

The early implementation of UNIX only had support for a single file system type for all mounted file systems. This solution worked well for main frame systems. But when microprocessors became available and used together with a centralized main frame, there was a need for a new file communication protocol that would allow users to seamlessly operate on both local and remote files.

Network File System (NFS) was designed to allow file operations to be executed transparently on both local and remote files. A remote file system (NFS server) is mounted into the local filesystems (NFS client) name space. Since two different file systems should coexist they must share the same interface layer to hide implementation details in the two

file systems. Figure 2.1 shows an overview of the NFS architecture. The details of NFS are outside the scope of this thesis, we will instead focus on the common file system interface layer called *virtual file system (VFS)*.

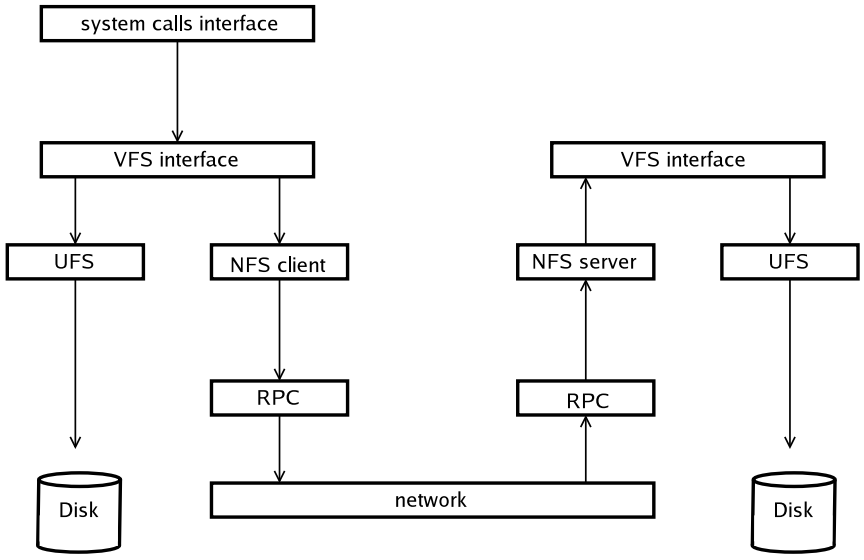

Figure 2.1: NFS and VFS architecture

VFS provides[8] a set of well-defined interfaces that are file system independent, each file system must implement these interfaces. There are two key objects that represent these interfaces: *vnode* interfaces implement file related operations and the *VFS* interfaces implement file system management operations. A call to one of these operations will be directed to the underlying file system.

When a file is opened, it will be represented by a vnode object in the UNIX kernel. The vnode object encapsulates the file's state and the operations that can be performed on the file, it contains three important items:

- File system independent data, such as file type (regular file, directory, device, etc), state, pointer to the underlying file system and a reference counter.
- A structure containing pointer to file system dependent implementation of file oper-

ations, such as: `open`, `read` and `write`.

- File system dependent data that is used internally by each file system implementation. In the case when the file is stored on a local UNIX file system this data will be the inode information.

The VFS layer provides administration interfaces to support commands like `mount` and `umount` in a file system independent manner. Like the vnode object, there is also a VFS object that encapsulates the state and the methods for the file system. Each file system must provide an implementation of the VFS object.

Figure 2.2 presents a schematic overview of how VFS is used in the UNIX kernel. As an example, the `open` system call, will first be processed by the vnode interface that determines which file system implementation that should process the open operation. The vnode interface uses the operation pointer for `open` to redirect the call to the correct file system implementation.

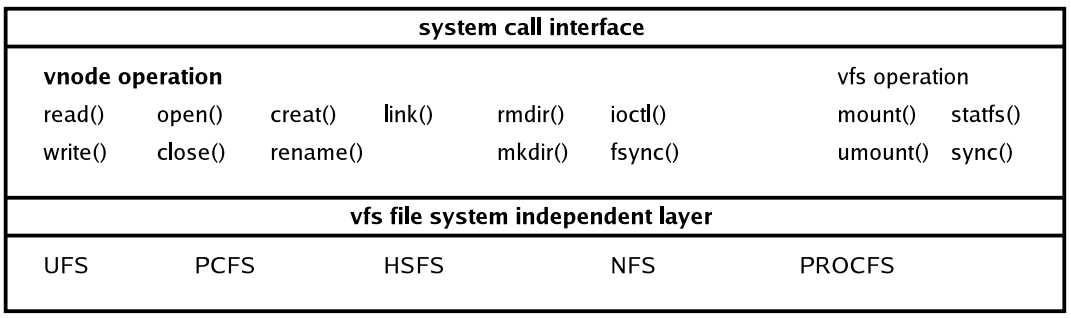

Figure 2.2: Virtual File System

As a consequence of introducing well-defined interfaces to file systems, the VFS interfaces have been used to implement other system resources as files and file systems. An example of the is the `/proc` file system in UNIX that exposes information about the operating system as files. A process can get information of the system memory on a specific machine via the file `/proc/meminfo`, see figure 2.3.

```
\$ cat /proc/meminfo
MemTotal:         125400 kB
MemFree:            5832 kB
Buffers:             872 kB
...
```

Figure 2.3: Linux proc file system

2.4 Modern file systems

The previous section discussed file systems related to UNIX-like operating systems. Other operating systems have their own file system architecture. For example, Windows NT has a file system named NTFS, which is not based on the VFS interfaces. However the Linux kernel development team has implemented a VFS wrapper that can be used to mount NTFS file systems on a Linux system. Therefore file system objects and operations in NTFS can be expressed as corresponding objects and operations in VFS. Even though file systems do not share a common design specification, they all adhere to a common understanding of the fundamental concepts of what file systems are, and VFS can be seen as a good representative for the concepts.

The major differences between modern file systems are not how they represent files or what operations defined on files, but rather what services the file system provides. A service can either be implemented as a part of a file system (hidden from the user) or be delivered as a normal application using the file system interface to implement file system services. For example, NTFS has a journaling service integrated that keeps the files system in a consistent state. But Windows NT is also shipped with a number of administration applications, like Disk Defragmentator and ScanDisk.

Common services provided by modern file systems are:

- Journaling, that addresses the problem of keeping the file system in a consistent

state. Journaling uses transaction logs to store file operations before the operations are applied to the underlying file system.

- Compression services at file system level, aim to increase the amount of data that a disk can store, by compressing every file in the system. This is possible since most files are not coded efficiently, i.e. usually too many bits are used to code the file data[3]. The file system will compress the data before it is written to disk, and uncompress data after it is read.

- Backup can be handled by the file system in two ways: either full backup or incremental backup. The idea of backup service[3] is to keep information on a separate location that can be used to restore lost data.

- Access control service allows users and processes to set protection rules on files. As an example, a file owner should have the possibility to set access control mode, so no other user or process in the system can operate on the file.

Figure 2.4 lists some of the most commonly used file systems on modern operating systems, together with the services that each file system provides. The list indicates that services in contemporary file system focus on reliability and security issues.

File system	**Description**	**Services**
Ext3	Linux File System	Same as Ext2 with journaling
Ext2	Linux File System	Access control, Symbolic links
NTFS 5	NT File System	Quota, Encryption, Journaling
NTFS 1.1	NT File System	Compression, Atomic transactions, Unicode file names
UFS	Solaris 7	Journaling, Snapshot[1], Access Control, Quota, Expandable

Figure 2.4: Common file systems and provided services

[1]File system can be locked to suspend all file system activities during backups

2.5 Additional file system services

The interface provided by VFS describes the file abstraction in the UNIX operating system. The file abstraction provided by UNIX does not include any high level file system services, like backup, encryption, decryption, compression etc. If a user needs any of the listed services or similar services she has to provide them herself, that is, the user is responsible to implement any additional services using the interface provided by VFS.

Additional services can be implemented on existing file systems using different techniques: the service can be implemented as a private part of an application, in which case the service is not accessible by any other application in the system, or as an operating system service. Operating system services are reusable by other applications and are implemented either as server applications or as libraries.

Operating system environments like UNIX provide a lot of tools that any application programmer can utilize to implement services and applications. For instance, services that have to be executed on a regular basis can use applications like *at* or *cron*, to handle the scheduling of services. Development languages like Perl, sed and AWK are commonly used to implement administration services.

Chapter 3

File systems

The main task of a file system is to provide means for users and processes to permanently store and retrieve information[13]. The information is stored in logical units called *files*. The file system should hide the complexity of the storage medium and provide a well defined interface for file manipulation.

A file system can be thought to consist of four distinct parts: file objects, naming, storage and file system services. Several operating system books[1, 3, 13] use the same division when presenting the concepts of file systems. However, such books seldom include a discussion of how file systems can be extended with new services. Instead, these books focus on describing a more static and traditional view of file system concepts. This chapter will describe the traditional view of the four parts in a file system, but also include the perspective of file system extensions.

3.1 Overview

The file system is regarded as one of the most important aspects of an operating system, and traditionally the two have been tightly integrated[5]. But the file system is not an entirely isolated unit in an operating system. For example, a user expects that a file

can be copied from the local hard disk to another computer, possible running a different operating system. This implies that there must exists some general agreement on the concept of files and file systems.

These general concepts shared by file systems are mostly related to how files and naming are presented to users. Figure 3.1 shows an overview of file objects in different abstraction layers, and the naming service provides a file object mapping between layers.

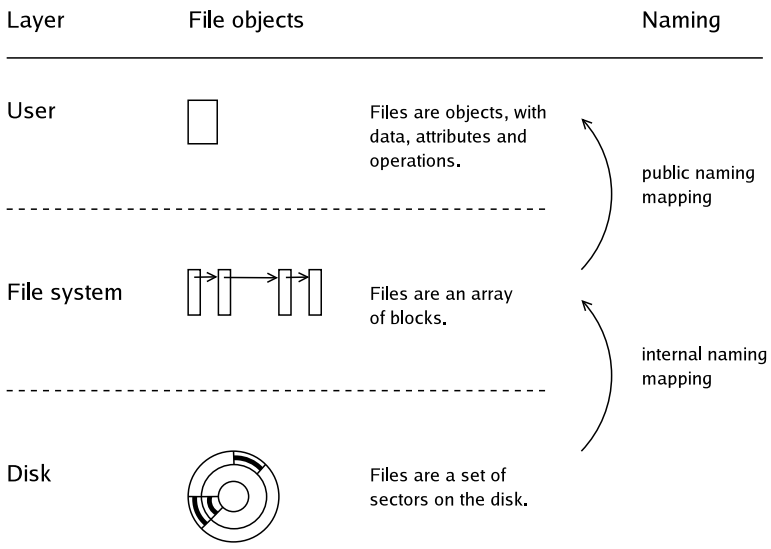

Figure 3.1: Overview of file system layers

Much can be said of the storage issues involved in file system, but since this thesis focus on file system as an abstraction concept, we will only mention issues of the file storage that directly effects the file system abstraction.

A file system also provides a set of services to operate and manage the file objects, naming structure and the storage issues. The last section in this chapter will discuss these services.

3.2 Files

A file is the basic abstraction provided by the file system. It is the object that a user or process will use to store data on a file system medium. Traditionally a file is defined as a named array of bytes, stored on disk. A more general definition would be; a file is a named instance of an abstract data type (ADT), stored on a file system. Figure 3.2 illustrates the interface between the user process and a file system. Under the file system layer there could be various system resources, for example a hard disk.

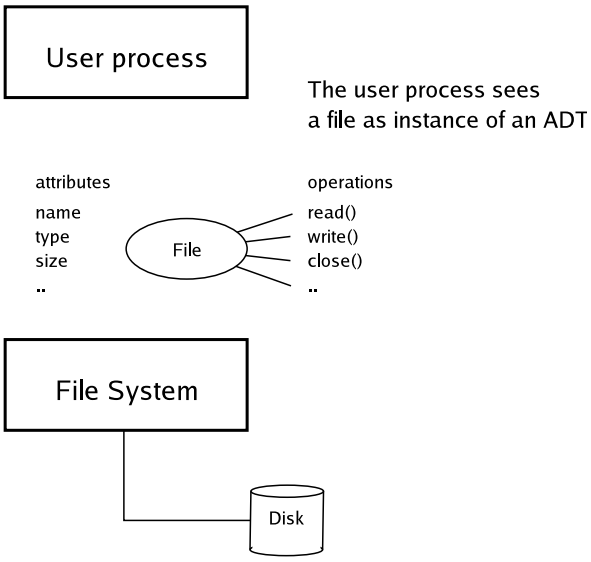

Figure 3.2: File system public interface layer

An ADT is a specification of a data type, or a pattern of a data type. A data type is defined to be a set of values, together with a set of operations on those values having certain properties [6].

3.2.1 File types

An important design decision is what file types should be known by the system. UNIX has six different types: *regular files*, *directories*, *character files*, *block files*, *pipes* and *sockets*.

As for regular files, it is often said that the operating system should not make assumptions on the file data structure to keep the system as flexible as possible. Executable files are an exception from this. The system must have means to determine if a file is executable. In UNIX this is done by examine the file, by looking for a special *magic number* that indicates how the file should be executed, as native binary code or by using a interpreter program[2].

It is interesting to note that a UNIX system can only make a best guess of the file type according to the pattern matching rules in the `magic` definition. As an example, consider the text file document in figure 3.3, that contains a short text describing how PDF documents can be converted into postscript files.

```
PDF documents can be converted to postscript file
by using the UNIX command pstopdf.
..
```

Figure 3.3: Consequences of UNIX magic number

This will be interpreted as a '`Macintosh PDF File (data) : F`' by the UNIX command `file`, only because the existence of the word `PDF` on the first line in the text file, matches the magic definition for Macintosh PDF Files.

Other operating systems provide other means to define the type of a file. Windows uses a file name suffix as type identifier, to separate the file type information from the file data. Hence, the system does not need to open a file to determine the file type, since it is a part of the file name.

Traditionally file systems do not expose the uniqueness of a specific file type as different file interfaces. The type information is mostly used by the operating system to associate a file with an application, or to determine execution environment (native code or interpreter). If a file type does not support an operation declared by the file interface its functionality is usually disabled. For example, in UNIX systems a write operation to the

device `/dev/random` (used to produce a series of random bytes) is simply ignored by the device driver.

File system service extensions could be achieved by allowing other file types apart from the ones provided by the file system. Such data model could be inspired by the data model provided by the object-oriented programming languages. A user could add a new file type into the system that extends an existing file type by introducing new operations and attributes, in the same manner as classes can be specialized by inheritance.

3.2.2 File attributes

So far we have seen that a file is defined by its name, data and type. But the system needs to associate more information about the files to handle them correctly. This additional information is called file attributes, or file meta data. An example of a file attribute is the access control information on each file that is used by the system to protect files from unauthorized access by other users.

Some file system provide a fixed set of file attributes while other provide the possibility to add new attributes. Figure 3.4 gives some examples of commonly used file attributes, grouping them into three different columns, based on what part of the system that uses the attribute.

System	**Application**	**User**
name	version number	free text comment
file type	content encoding	
size	file icon	
access control	window position	
time stamp		

Figure 3.4: Example of file attributes

Minimizing the number of file attributes is considered to be a good approach to keep the file system flexible, and especially avoiding attributes that are dependent on the file

content. The reason for this choice is that content dependent attributes complicate file operations like copy, read, write, etc. Consider two files `mysong` and `myimage`, the first is an mp3 music archive file and the second is a jpeg image file. How should the operation shown in figure 3.5 be interpreted?

```
> cp mysong myimage
```

Figure 3.5: The UNIX file copy operation

Should the copy operation be allowed at all? or should the operation duplicate the mysong file (attributes and content) and save it as the target file `myimage`? It gets even more complicated if we introduce another file system that does not support the same attribute space. All these questions must be defined as a semantic behavior of the file system.

3.2.3 File operations

Like all parts of an operating system, the file system implements objects and operations on those objects[3]. With respect to file systems, these operations can be divided into two groups. First, there must exist operations to create, open and delete files using the file name as argument. These operation are used to manage file objects and not used to operate on the file data. The second group of operations are operating on a specific open file to manipulate its data and attributes, for example: `read`, `write`, `seek`, etc.

Each file system has a different set of operations depending on their need. But again, operations in UNIX file system are representative for several existing file systems, figure 3.6 lists the operations on file names in a traditional UNIX file system, and figure 3.7 lists operations applicable on open files.

The open and create operations prepare access to an open file, in UNIX open files

Operation	Argument	Return value	Description
open	file name	open file	Creates an open file
create	file name	open file	Create a new file and an open file attached to it
status	file name	file info	Returns file meta-data
access	file name, access mode	true or false	Checks if the access mode violates protection rules
change mode	file name, access mode	true or false	Change protection mode
change owner	file name, owner	true or false	Change owner
pipe		two open files	Creates two open files that are attached to each other

Figure 3.6: Operations on file names in UNIX

are identified by a file descriptor number. The operations: status, access, change mode and change owner, operates on attributes for the named file. The pipe operation creates two special files used for interprocess communication, and consists of an object allowing unidirectional data flow between a pair of open files.

Operations on open files in UNIX reflect the fact that a UNIX file is interpreted as a sequence of bytes. A process can read and write bytes at some position of a file. The position can be changed by the seek operation. A process can also lock the file to ensure exclusive access to the data. Eventually an open file should be closed, this means that the open file object is deallocated from system memory, and the connection between the open file and the file itself (file descriptor) is removed.

The last operation in figure 3.7, the file control operation (`ioctl`), is used to perform various operations on special files to control the underlying device. For example, the `ioctl` operation can be used on CDROM devices to perform operations like, start, stop and eject. An `ioctl` call takes two or possible three arguments, a file descriptor, a command identifier and possible a pointer to data. However, the `ioctl` interface provides no standard operation to list supported commands for a device.

The file operations in UNIX are tightly connected to the file abstraction. It is left to

Operation	Argument	Return value	Description
read	open file	success	Reads file content
write	open file	success	Write file content
seek	open file	new position	Change to file pointer
close	open file	success	Destroy open file and updates the attached file
duplicate	open file	new open file	Makes a copy of an open file
lock	open file	success	applies or removes an advisory lock on the file associated with the open file.
file sync	open file	success	Write content to disk
file status	open file	file info	Returns open file's meta-data
file control	open file	success	Various control operations

Figure 3.7: Operations on open files in UNIX

each application that uses a file to interpret the data, the file itself provides no knowledge about the data.

An alternative approach would be to define a new file type (by introducing a new ADT into the file system) that provides specialized operations with knowledge of the data's structure. For example, a new file type could define new operations to handle a `md5` checksum mechanism to automatically validate the correctness of data in the file. Today this feature must be handled by external applications that calculate the checksum and a user must compare the result by a checksum value. In chapter 4 and 5 we will examine the consequences of introducing new file types into a file system.

3.3 Naming

Files are an abstraction of the underlying medium, hence the file system must provide a naming mechanism to allow processes and users to uniquely identify a file. Every file name will be added into a name space, this makes it accessible by other processes. A file system must define rules how files are named. All modern systems allow users (or processes) to

assign a lexical name from a limited number of characters to a file, but internally the file system uses an integer file identifier for each file.

It would be hard for users to provide a unique name for each file in the system if the file system did not have directories to group a set of file names together, separating them from file names in other directories. Since the birth of Multics and UNIX it has become standard to structure directories hierarchically, where directories can contain files and other directories in a tree structure. The top directory in such structure is usually named `root`. To ensure name uniqueness, every name must be unique within a directory. Also each directory can have its own name lookup operations, this makes it easy to combine name spaces that are implemented in different ways.

The naming system used on file systems has successfully been used as a unification concept of other system resources[3]. This means that the same naming mechanism can be used to describe other resources within the same name space as the files. Different concepts can be unified whenever:

- Two concepts or interfaces are basically the same.
- Functionality is not lost when combining them.

An example of concept unification can be seen in the way UNIX handles devices with the same mechanism as regular files[3]. Both file access and device access are similar to each other. This solution reduces the complexity of the overall system.

The lexical file naming and directory structure is mainly used to aid human users to organize the files in the system, but it is important to realize that alternative structures are possible. For example, a user can identify a specific file by execute a query on file attributes (name, author, create data, etc).

In modern file systems, file attributes are accessed and modified via a set of system calls, and not included in the directory name space. However, file attributes could also be

integrated into the file system name space. This would allow users to directly operate on file attributes and easily examine the set of accessible attributes.

In the article *Reiser4*[11], the author proposes a syntactical notion for a unified name space for files and attributes in UNIX systems. Attributes are represented as named objects attached to a file, similar to the way files are attached to a directory. A special syntax is introduced to separate filenames from attribute names, and the suggested syntax convention is to let all attributes be prefixed with the characters '..', as illustrated in figure 3.8, that identifies the `..uid` attribute associated with the file named `myfile`.

```
./mydir/myfile/..uid
```

Figure 3.8: Example of a prefixed attribute

David K. Gifford et al.[4] propose another solution to add file attributes into the name space, and the authors have implemented what they call a *semantic file system* that allows users to query the file system based on attributes. A query is defined to be a description of desired attribute or attributes, that permits entities to be selectivity located. In figure 3.9 the *semantic file system* is mounted under the normal UNIX name space, `/sfs`, and the query performed in the figure produces a listing of the titles of all documents owned by smith.

```
> cd /sfs
> ls owner:smith/title:
Economic Report Last Fall
Freds birthday speech
>
```

Figure 3.9: Example of a query in the semantic file system

The list of files generated as the result of a query is called a virtual directory. But unlike an ordinary directory, a virtual directory does not have to be explicitly created and saved on disk to be accessed.

3.4 Storage

We stated in the beginning of this chapter, that one of the most important features of a file system is to provide means to permanently store and retrieve data. The ideal situation would be if the file system abstraction could hide all details of the underlying medium, and allow users to manage files without any knowledge of how the disk works. But this is not the case. The simplest example of how the disk limits the usage of a file system is that a file can not be larger than the disk capacity.

Figure 3.10 illustrates the different layers involved in a disk based file system. Each layer provides another level of abstraction. A file system implementation does not have to deal with platters and cylinders, it only sees an array of logical sectors. It is the responsibility of the device driver to optimize the mapping from logical sectors to platters, cylinders and sectors.

The data is often presented to users as a sequence of bytes of arbitrary length. In reality the length of the file is from 0 to a very large upper limit, 4 GB is a common limit. The limit is imposed by the underlying file system as well as the underlying hardware[3].

The smallest unit of the disk is called data blocks, which have a predefined size. If a file is larger than the block size it must be stored on several blocks, possibly spread out on different sections of the disk medium. This is called file fragmentation, and a fragmented file can not be accessed as fast as a file stored in consecutive blocks. Different file systems provide different solution to this problem. The Ext2 file system on Linux systems uses block allocation algorithms to limit the fragmentation of files. Also, several operating systems provide tools to rearrange the disk blocks so that blocks containing a file are

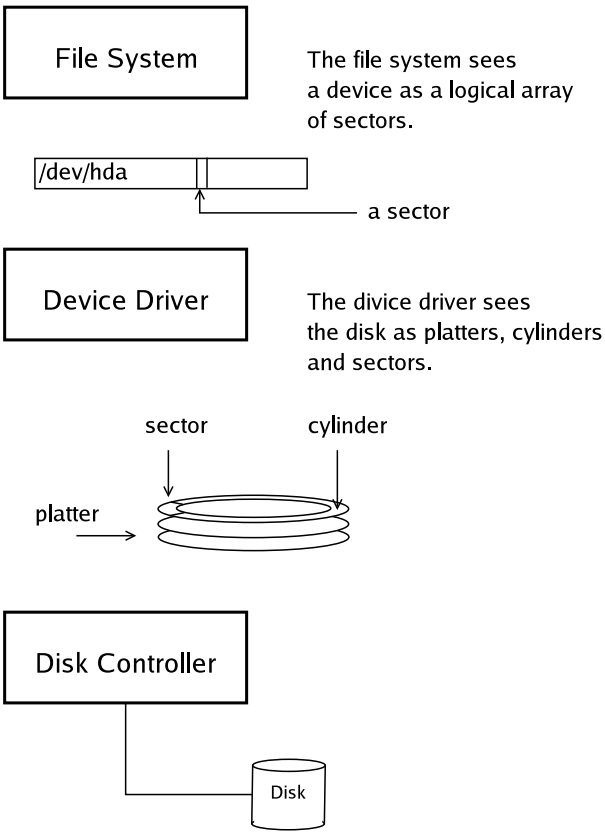

Figure 3.10: Disk based file system layers

stored consecutive on the disk.

3.5 File system services

So far we have discussed the central object in the file system, i.e. the file. But the file system itself is also an object, and just like a file it has operations, or file system services as the operations are called from now on. A file system service is mainly used to administrate the previously defined parts of the file system, i.e. files, naming and on disk storage. The following list illustrates common file system services:

Files Journaling services uses transaction logs to store file operations before the operations

are applied to the target file.

Naming In VFS, the `mount` operation defines how a file system naming system should be added into a name space. This is a general purpose command, and each file system must provide its own implementation of the mount operation.

Storage Every file system that manages disk storage, has specialized services and operations to utilize the disk space efficiently and provide a reliable storage medium to the users.

Other operations manage how files are being structured on the logical array of disk blocks. Chapter 2 listed some of the most common services that exists in modern file systems (journaling, backup, access control, etc). These services are implemented as operations on the file system object.

The next chapter will continue the discussion of extendable file systems in general, and includes a case study on two file systems that enables users to define new file system services.

Chapter 4

A case study on extendable file systems

In the previous chapters we have seen that file systems deal with more issues than just data storage, on modern file systems these issues usually concern services to increase reliability and security provided by the file system vendors.

This chapter will present two possible techniques used to allow third party service providers to enhance file systems with new services. As case study we will use two new file systems: WinFS and Reiser4.

4.1 Motivation

A great deal of benefit would accrue if it is possible to add new file system services to an existing system as easy as it is to add new user level applications[5]. From a user perspective, an operating system is not delivered as a monolithic set of programs but rather as a number of independently developed programs. This situation has resulted in a rich variety of different programs available for end users.

But file systems have traditionally been developed as stand alone systems, and it has

been hard for anyone but the file system owner to add new services. If file systems were designed with a well-defined mechanism to add new services, it would be possible for others to provide new and useful services.

Also, file systems could introduce some interesting features from object oriented databases to allow extendibility. If a file system provided a richer set of file types, this would have a great impact on the way end users utilize the concepts of files. For example, files could be organized based on different file types, instead of relying on directories as the only organizing structure.

4.2 Techniques

File systems that allow third party services, can use either of two techniques. The first technique allows external services to be added as new file types (i.e. extend the file abstraction), while the other technique can only introduce new services that preserve the file abstraction defined by the system. The two techniques are discussed in the next sections.

4.2.1 Extend file abstraction

The first technique of service extension focus on the concept of files. By adding new interfaces on the file objects, the file system can expose file type specific operations to other application in the system. This means that an application, that uses a specific file type can rely on the file to perform some operation instead on having to provide a corresponding implementation.

Since new file interfaces are applied to a specific file type in a file system they cannot be used to implement traditional file system services that need to operate on all files in the file system. As a consequence this technique will most likely be used by application developers to add some of its application logic to the file system. And by doing so, other applications in the system can utilize the same application logic on these file types.

4.2.2 Preserve file abstraction

This technique preserves the traditional file abstraction, in the sense that a file is a sequence of bytes and that the file system avoids having knowledge of the structure of the file data. This limits the services that are possible to introduce to a files system. John S. Heidemann has in a paper[5] identified the following possible services:

- Extended directory services, could help file system users to organize their files in a more convenient way.
- Compression and decompression, stores the file data in a compressed format on the disk.
- Encryption and decryption, would improve security by guaranteeing that all data on the disk is stored encrypted.
- Cache performance enhancement can usually improve the performance on disk operations by keeping the file data (or a part of the file data) in memory.
- Remote access services could be implemented as a file service extension to allow seamless access to remote files.
- Selective file replication could be handled internally by the file system.
- Undo and undelete is often implemented at application level, but this service could be provided by the file system. Hence this could be used by all applications.
- Transaction is a technique to guarantee that the file system stays in a consistent state. It means that a file operation is either completed or it is not performed at all. This is also called atomic operations.

The architecture should provide means to configure what services should be part of a file system. Each service should be implemented as self contained module to achieve the

desired flexibility. Also, a service should be able to define new file attributes to store state information about the file. For example an encryption service must probably associate files with an encryption algorithm.

In the next two sections the files systems WinFS and Reiser4 are presented as case studies on two techniques. WinFS is an example of a file system that lets users extend the file abstraction, whereas Reiser4 extends the file system without modifying the file abstraction.

4.3 WinFS

WinFS is a new file system developed by Microsoft, it is planned to ship as part of the next version of their operating system, that currently goes under the code name *Longhorn*. It is planned to be released some time in 2006. Microsoft describes WinFS as [9]:

> '..an active storage platform for organizing, searching for, and sharing all kinds of information. The platform defines a rich data model, built on top of a relational storage engine, supports a flexible programming model, and provides a set of data services for monitoring, managing, and manipulating data.'

This section will explain some of the new mechanisms that exist in WinFS to support addition of third party defined services. The *rich data model* mentioned in the quote above is a mechanism provided by WinFS to allow developers to add new file types and thus adding new file services into the system.

Before we describe the details of these mechanisms, we will present a brief overview of the WinFS architecture[10].

4.3.1 Architecture

WinFS introduces the term *item* to describe the smallest unit of consistency and operations, instead of using the traditional name *file*.

Microsoft has not implemented WinFS as a standalone file system, instead it is designed as a service on top of NTFS. An item in WinFS, stores its attributes (meta data) in the WinFS service. But if the item holds stream data (i.e. traditional file data), this is stored in NTFS as an ordinary file. A program the operates on the item stream data makes use of the streaming facilities provided by NTFS. Figure 4.1 shows the two layers in the WinFS architecture. A WinFS item has attributes in the WinFS service that describe properties of the stream data stored in a NTFS partition.

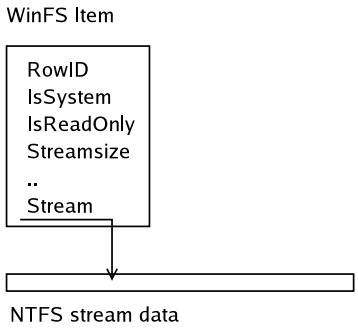

Figure 4.1: The WinFS NTFS stream data

Microsoft does not consider WinFS to be a traditional file system. They prefer to describe WinFS, as a storage platform with features inherited from file systems, relational databases and object databases.

4.3.2 WinFS data model

The top of the WinFS data model hierarchy (illustrated in figure 4.3) is a *WinFS service*, which simply is an instance of WinFS. Each WinFS service can contain one or more *volumes* that is the largest autonomous container of *items*. On a system there is one WinFS service that manages local or remote volumes. A user or a process can connect to (mount) a specific volume to access items on the volume. For example a programmer can connect to the default volume on the localhost by using the statement listed in figure 4.2. The

connection object is called `ItemContext`.

```
ItemContext context = ItemContext.Open("\\localhost\defaultstore");
```

Figure 4.2: Connect to the default WinFS volume

Items are organized into *folders* that can be either a containment folder (like a traditional directory) or a virtual folder that represents a dynamic collection of items based on a selection query. Figure 4.3 illustrates the data model hierarchy.

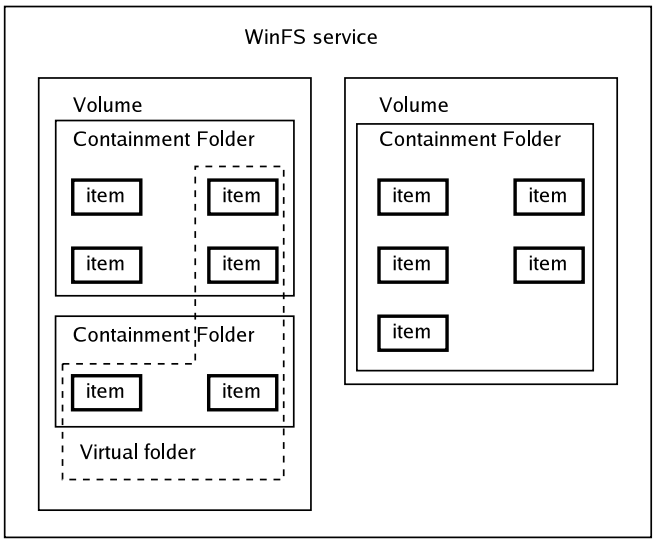

Figure 4.3: The WinFS data model hierarchy

WinFS introduces a XML based schema definition language used to describe WinFS types. Each item in the WinFS service is an instance of a specific data type. These scheme definitions give the system knowledge of the data structure for every item in the system. The WinFS service has 28 predefined schemes, for example, there are schemes defining `Person` items and `Document` items. WinFS allow programmers to add new schemes to the system, i.e. extending the data model[1].

[1]The current preview release of WinFS does not yet support this feature, but Microsoft claim that user

4.3.3 Relational storage

Microsoft introduce the concept of relationship between a source item and a target item. There are two kind of relations; *holding relationship* and *reference relationship*.

A holding relationship controls the lifetime of their target items, i.e. whenever a relationship is deleted the target item is also deleted. In WinFS folders are implemented as a holding relationship called FolderMember. This means that files must exist within a folder.

Reference relationships on the other hand, do not control the lifetime of the target item. As an example, the reference relationship could be the Author relationship between a Document and a Person. The Document and the Person will exist even if the Author relationship is deleted.

Possible relationships on an item are defined in the same XML file describing the item type. For example the XML file defining the item Folder contains a definition of the relation FolderMember that is used to relate any item with a folder.

4.3.4 Programming model

The primary API for accessing WinFS data, called WinFS API, is an object oriented API. It is designed to allow programmers to operate on items as instances of a specific class. The following list describes the three different data access APIs that are available for programmers.

- The WinFS API allows programmers to manipulate items via a strongly typed API. This means that items can be manipulated as instances of a specific class. For example an item can be opened as an instance of a Person class that exposes attributes like `fullname` and `firstname`; and operations to manipulate these attributes.

defined schemes will be included in the beta version.

- The ADO.NET API provides low-level access to WinFS data by using SQL queries, and operates on database tables and rows. ADO.NET also provides means to retrieve data as XML documents.

- The Win32 API provides backwards compatibility for old applications. This means that developers can operate on the items using the traditional view of files and folders.

Regardless of what API developers use to access WinFS data, each API ultimately manipulates data in WinFS using SQL queries. The WinFS API and ADO.NET API adds another layer of abstraction on the relational database holding the WinFS data.

Figure 4.4 illustrates the usage of the WinFS API. It contains a code fragment that fetches all items of the type Contact and makes a class-cast to Person. The full name attribute of the items is then written to the output console window. The first line creates an ItemContext, this is a connection to a specific WinFS volume. The Open operation without argument gets the default volume on the local host. After the task is completed the context must be closed to release allocated resources.

```
ItemContext context = ItemContext.Open();

foreach ( Person prs in Contact.findAll(context) ) {
  Console.WriteLine("Contact first name: {0}", prs.FullName);
}

context.Close();
```

Figure 4.4: Example of the object-oriented WinFS API

The WinFS API provides a mechanism for programmers to store and retrieve objects in a simple manner, much like most object-oriented languages provide some object serialization mechanism. But since the file system handles the storage of the items and have knowledge of their file types, any application can use these attributes. For example the

Windows Explorer application can present the `fullname` attribute for every item of the type Person.

When an item changes or is deleted, WinFS generates a notification. An application can make use of the WinFS notification mechanism to extend the services in the system by creating `Watchers` that can perform specific tasks on different events. For example if an item changes its state the watcher can notify an application.

4.3.5 Current status

WinFS is still under development. Microsoft has released an early preview version for members of MSDN (Microsoft Developer Network). But it is still unclear exactly what features of WinFS that will be included in the new Longhorn operating system, the beta release expected to ship in 2005 will indicate what direction Microsoft is planning for WinFS.

4.3.6 Summary of WinFS

The WinFS platform will provide the means to add new file types into the system. Microsoft believes that services added by third parties will most likely be using this to add some of the application logic as a part of the file type. For example a media archive file format can use file attributes to store information about the archive instead of encode it together with the media data. This would mean that a user could modify the attribute directly via the shell instead of using a specialized tool. Also a media archive file type could be extended with operations to validate data in order to identify bit errors.

4.4 Reiser4

Reiser4 is an open source file system developed by the Naming System Venture, usually called Namesys. The project is sponsored by the DARPA[2] organization. Today Reiser4 is under development and Namesys has not given a final date for the Reiser4 release. However, the team regularly releases development versions that make it possible to follow their progress.

4.4.1 Overview

Reiser4 is a completely new file system using the VFS interface. As of today the file system is only implemented on Linux operating system. The following list provides a short summary of interesting features in Reiser4[11].

- Efficient support for small files, by grouping small files into one disk block.
- Flexible plugin infrastructure allow developers to add new file system services.
- Atomic file system operations guarantees that file operations are completed and never left unfinished.
- Journaling to reduce the damage of system failure.
- File meta data is integrated into the file system name space.

This section will focus on the plugin infrastructure and file meta data, as means for extending Reiser4 with new file system services.

4.4.2 Plugins

Every file possesses a plugin identifier, this identifier is used to define a set of methods that embodies all of the different interactions on the file.

[2]Defence Advanced Research Projects Agency

A plugin is described by Reiser4 documentation as a set of internal *modules* used to increase extendibility and to allow external users to easily adapt Reiser4 to their needs. Plugins are classified into several disjoint types. Files belonging to a particular plugin are called *instances* of this type. The following list present some of the existing plugins in Reiser4.

- The *object plugin* determines how Reiser4 files should serve standard VFS requests (read, write, seek, etc).
- The *directory plugin* implements a traditional directory view of the Reiser4 file system.
- The *hash plugin* computes hash values for files, used to store and locate files within directories, instead of using the file name directly.
- The *perm plugins* control permissions granted for processes accessing files.

The future goal for Namesys is to allow dynamic loading of plugins, but in the current version of Reiser4 a programmer must modify some of the kernel source code to add new plugins.

4.4.3 File attribute name space

Since the Reiser4 file system has efficient support for small files, it can store file attributes as normal files without wasting disk space, this is done by allowing small files to share a single disk block. Recall that in a traditional Unix file system the minimum allocated disk space is a block with fixed byte size (1024, 2048 or larger). File attributes are usually just a few bytes, so it has been important for Reiser4 to optimize the disk block usage for attribute storage.

By treating file attributes as normal files Reiser4 has integrated the attribute naming into the file system name space, meaning the attributes can be read and written using

normal file operations. As shown in figure 4.5, a user can get the owner, `uid`, of the file `myfile.txt`, by using the Unix command `cat`. Note that the result is the integer value for the Unix user, not the lexical name.

```
> cat myfile.txt/..uid
500
```

Figure 4.5: Accessing Reiser4 file attribute

Each file in a Reiser4 file system has several attributes describing the file; owner, access control, pluginid, etc. A plugin developer has the possibility to add new file attributes. But arbitrary attributes can not be added by a normal user, new attributes have to be defined in a plugin.

The combined name space for files and file attributes, can result in somewhat confusing semantics, since normal files are both data containers and directories holding attributes. As an example, all files have a `..plugin` directory that contains attributes with plugin informations. Therefore it is possible to change the current working directory using the shell, see figure 4.6.

```
> cd myfile.txt/..plugin/
> ls
compression  crypto  digest  dir  dir_item
file  formatting  hash  perm  sd
```

Figure 4.6: Reiser4 plugin information for a file

But a user cannot read the directory information of the file itself, as shown in figure 4.7. On the other hand, this would be useful if a user would like to list all attributes associated with a file.

```
> cd myfile.txt/
> ls
ls: reading directory .: Not a directory
```

Figure 4.7: Reiser4 name space semantics

Reiser4 solves this by creating an attribute `..pseudo` that contains a list of all available attributes associated with the file. Figure 4.8 shows some of the attributes associated with the file myfile.txt.

```
> cat myfile.txt/..pseudo
..uid
..gid
..rwx
..key
..size
..plugins
...
```

Figure 4.8: Availible file attributes in Reiser4

The Reiser4 development team is currently discussing how the attributes name space syntax should be designed. So it is still unclear how the syntax for the attributes should be handled. The syntax describe above is how file attributes are used in the current version of Reiser4, but it might change in future versions of Reiser4.

4.4.4 Summary of Reiser4

The Reiser4 file system is under development and some of the features regarding attribute name space and plugin architecture are still being discussed at design level. It is still interesting to try to identify what kind of services the Reiser4 believes can be implemented

as plugins. The fact that Reiser4 preserves the VFS interfaces on files and attributes means that new services will most likely target low level functionality like security and reliability.

Chapter 5

The prototype file system PiVO

This chapter describes the architecture and design of a file system prototyped called PiVO. PiVO is able to handle files of different types, i.e. files with different sets of attributes and operations. The chapter also includes a description of the programming and user interfaces defined in PiVO. The use of the programming interface is illustrated by pseudo code for common operations applied on files in PiVO. The user interface is illustrated by examples from shell sessions using PiVO as the underlying file system. Also, the last section is a general discussion on extendable file systems; using PiVO, WinFS and Reiser4 as reference systems.

5.1 PiVO Architecture

5.1.1 Introduction

PiVO is a simple file system, or more accurately, it is a file system prototype, that consists of a shell and a file system. PiVO is able to handle files of different types, and provides a programming interface, and the shell provides an example of an end user interface. As described in chapter 3, a file system consists of four parts: files, file system services, naming and on disk storage. PiVO is designed to illustrated the first three parts of a file system,

hence the details of how a file is stored on a permanent disk storage are not discussed.

5.1.2 Files

All files in PiVO are defined as a set of attributes and operations. The set is assigned to a file when the file is created, and are not to be changed once assigned. The set of attributes and operations is divided in two parts: a predefined part and a template defined part. The reason behind the division is that even though files in PiVO are supposed to have individual sets of attributes and operations, the management of files is simplified if certain attributes and operations are guaranteed to be present on all files in the file system. The predefined part of the attribute and operation set is used to fulfill this requirement. The members of the predefined set is defined internally in PiVO, and cannot be changed unless the system is recompiled. The most important predefined attribute is '_data', and the operations 'read' and 'write'. The attribute '_data' is used as a handle for the data stored in the file and the operations 'read' and 'write' are used to access and modify the stored data.

The template defined set on the other hand is not, as the predefined set, required to be the same for all the files in the file system. The members of the set is defined in a unit called *template*. The designer of the template is free to define the attributes and operations for each template. All files in PiVO are derived from a template.

The set of attributes and operations constitutes the type of the file. Internally PiVO maintains a collection of the available file types in the system, and a user can dynamically add new file types to the collection. The collection of file types can be listed to make it easier for a user to choose the file type when a new file is created. If two files created from the same template are said to be of the same type, this means that the files have the exact same set of attributes and operations, and that the value of the attribute '_template' is the same. The attribute '_template' is used to indicate the name of the template used when creating the file.

Templates in PiVO are, in essence, files without data and are used amongst other things to define the operations on files. Operations are divided in two groups: file operations and file system services. The two groups of operations differ in the number of files they are allowed to operate on, and who is responsible for the execution of the operation. File operations are limited to one instance of a file, and the responsibility to execute the operation is left to programmers and end users. File system services on the other hand, are allowed to operate on a collection of files, and the execution of a file system service is handled entirely by PiVO. A file system service is executed as the result of an event. An event is generated when a predefined condition is fulfilled. PiVO is limited to handle only two events, the opening and closing of a file. If a file is to be part of the collection of files affected when a file system service is executed is based on whether the files template includes a definition of the file system events. This is a rather crude method and in a more advanced file system than PiVO one would expect the handling of events to be more flexible.

5.1.3 Internal and external name spaces

PiVO does not impose any predefined naming rules for the files in the system. Instead it defines two name spaces, one internal and one external name space. When a file is created it is assigned a unique integer value, called a file descriptor. The file descriptor is used internally in PiVO when referencing a file, and the assignment is managed by the internal name space. Even though the internal name space offers a solution to the problem of uniqueness, it is not user friendly to let users handle files having an integer value as name, this is where an external name space can be used. An external name space can define a new naming scheme, built above of the internal name space, thus encapsulating the internal names. The primary goal of an external name space is to simply the naming for users, this is done by using lexical names instead of integer values. An external name space in PiVO is defined, once again, using templates. Two common external name spaces

are directory structures and views. PiVO has one external name space, a simple directory structure.

5.1.4 Architecture

An overview of the architecture of PiVO is illustrated in figure 5.1. Here is a clean separation of duties, and the responsibility of the modules in figure 5.1 is described below.

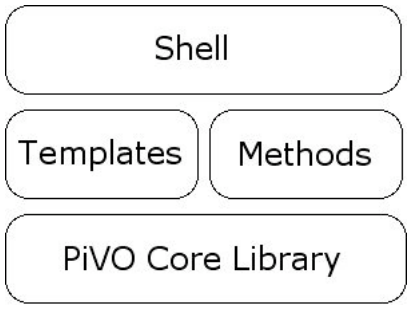

Figure 5.1: The elements of PiVO

- Shell

  The shell is a command language interpreter that executes commands read from the standard input or from a configuration file. The command language is used by users to interact with the underlying file system, and examples of operations are: list files in a directory, change current working directory, list all available operations on a file, and execute file operations.

  When the shell is started it creates an initial file system, using the directory structure template. The file system is not supposed to be persistent between two executions of the shell, and all created files during a shell session are stored in the file system created by the shell.

- Templates

  Templates are, in essence, files without data, and are used to define the template

specific set of attributes and operations in a file. This set, together with the predefined set of attributes and operations, defines the file type, and file types are a central concept in PiVO since it enables files with different characteristics to be created.

In addition to define the file operations, a template designer can use the template to decide if files of the defined file type are to be included in the set of files that are affected by file system operations. This is done by adding a definition for the file system operations in the template for the file type.

External name spaces are also defined using templates. PiVO defines one external name space, a simple directory structure that is used by the shell to let users navigate and uniquely identify files by lexical names.

PiVO maintains an internal collection of available templates. The collection can be dynamically updated by users to add new file types, file system services, or name spaces to the system.

- Methods

  All operations defined in templates must have a corresponding implementation. The implementations are defined in separate modules, called methods in figure 5.1. The separation of the operation definition and its implementation makes it possible for template designers to share operation implementations.

- PiVO Core Library

  The process of file creation and on disk storage is handled by the library module. The library module defines and assigns the predefined set of attributes and operations for the files, and is responsible for the execution of file operations and file system services. Since the library module is responsible for the file creation process, the library also handles the assignment of file descriptors for each created file.

  In addition to the file system specific functionality described above, the core library also defines a programmer interface that enables programmers to operate on files in

the file system. The shell in PiVO is an example application that uses the interface, and examples of its use is presented in section 5.2.

5.1.5 Attributes and operations

All attributes can be of one of two types: predefined or template defined. The two types of attributes are separated lexically using a simple syntax convention. The names of template defined attributes begin with a capital letter, while predefined attributes have names beginning with the character '_'. Figure 5.2 contains a listing of all predefined attributes in PiVO, and a short description of each attribute.

Attribute	Description
_creation time	Time of file creation.
_size	Size of the data.
_name	Name of the file.
_data	The data in the file.
_template	Name of the template used to create the file, i.e. the file type.

Figure 5.2: List of predefined attributes.

The operations are divided into three different types: predefined file operations, template defined file operations and file system services. Once again a simple syntax convention is used to separate the operations from each other. The predefined and template defined operations all begin with a lowercase letter, and file system operations all have names beginning with the character '#'. The PiVO implementation, presented in appendix A, defines two file system operations #a_open and #a_close, but a template designer may use any name for a file system service as long as the name is prefixed with the '#' character. The set of predefined file operations is listed in figure 5.3.

Operation	Description
create	Create a new file.
open	Open a stored file.
close	Close an open file.
delete	Delete a file from the on disk storage.
get_methods	Get a list of all operations defined on the file.
get_attributes	Get a list of all attributes defined on the file.
exec_m	Execute a file operation.
get_attr	Get the value of a specified attribute.
set_attr	Set the value of a specified attribute.
get_template_name	Get the value of the attribute _template.
set_template_name	Set the value of the attribute _template.
read	Read the data from a stored file.
write	Write data to a stored file.
get_name	Get the value of the attribute _name.
set_name	Set the value of the attribute _name.
get_size	Get the size of the stored dat.

Figure 5.3: List of predefined operations.

5.2 Programming interface

The programming interface defined in the core library encapsulates a file by a file object, and all operations on the files are managed through the file object. The file object is created with a call to the function `new_file`. A newly created file object is associated with a specific file when the file is opened, using the method `open` which is accessible through the file object. The method `open` takes two parameters, the file object and a file descriptor. The mapping between the lexical name and the file descriptor is handled by the external name space and the details of how this is done is implementation dependent. For this reason a generic function called `lookup_fd` is used to illustrate the name mapping in the examples.

Once a file is opened, it is possible to utilize the operations defined for the file to operate on the attributes, i.e. read or write the values of the attributes. The programming interface provides operations to read and write the values for all predefined attributes. The

operations are named after the attribute they operate on with a prefix of either "get_" or "set_" depending on whether the operation is used to read or write the attributes value. The only exception to the name convention are the names for the operations defined for the attribute '_data', these are named `read` and `write`.

The code segment shown in figure 5.4 illustrates the functions used to read the '_size' attribute of a file using the operation `get_size`.

```
int size;
File f = new_file();        // Create a new file object.
int fd = lookup_fd(name);   // Look up the file descriptor
f->open(f, fd);             // Open the file.
size = f->get_size(f);      // Get the size of the file.
printf("Size of file %s is %d\\n", name, size);

// Deallocate memory
f->close(f);
f->destroy(f);
```

Figure 5.4: Read a predefined attribute value.

Template defined attributes can be accessed in the same way as for predefined attributes, but the name of the operations are decided by the template designer. If the name of the attribute is known by a programmer, she can utilize generic attribute operations defined by PiVO. The operations are named `get_attr` and `set_attr`. Figure 5.5 illustrates how the operation `get_attr` is used to get the value of the attribute named 'Documentation'.

The `set_attr` operation is used in a similar way as `get_attr`. Figure 5.6 shows a code segment to set the value an attribute named 'Documentation'.

Template defined operations are executed using the generic operation called `exec_m`. The operation `exec_m` takes at least two arguments, a file object, the name of the operation to execute, and optional arguments to the operation. Figure 5.7 shows a code segment that

```
Data data = new_data();
String attribute = new_string();
String doc = new_string();
File f = new_file();
int fd = lookup_fd(name);

attribute->set(attribute, "Documentation");
f->open(f, fd);
data = f->get_attr(f, data);
f->close(f);
doc->set(doc, data->get(data));
printf("File %s is documented as: %s\n", name, doc->get(doc));

// Deallocate memory
doc->destroy(doc);
data->destroy(data);
attribute->destroy(attribute);
f->destroy(f);
```

Figure 5.5: Read an attribute using the generic access operation.

```
Data data = new_data();
String attribute = new_string();
String doc = new_string();
File f = new_file();
int fd = lookup_fd(name);

doc->set(doc, "This is a documentation string");
data->set(data, doc->get(doc);
attribute->set(attribute, "Documentation");

f->open(f, fd);
f->set_attr(f, attribute, data);
f->close(f);

// Deallocate memory
...
```

Figure 5.6: Write an attribute using the generic access operation.

executes the zip operation of a file.

```
File f = new_file();
int fd = lookup_fd(name);
String file_operation = new_string();

file_operation->set(file_operation, "zip");
f->open(f, fd);
f->exec_m(f, file_operation);
f->close(f);

// Deallocate memory
...
```

Figure 5.7: Execute a file operation.

5.3 User interface

As a part of the file system PiVO we have implemented a simple shell to illustrate how users could utilize the PiVO services. This section will use examples from shell sessions, to demonstrate how users operate on files and manage file templates.

When the PiVO file system has been initalized, a `root` directory has been created and two templates has been loaded into the system. As illustrated in figure 5.8, the user can list all loaded templates by using the shell command `loaded`

```
> loaded
plain
dir
```

Figure 5.8: List all loaded templates in a PiVO system.

Next user action, shown in figure 5.9, is to add a new template into the PiVO system

by executing the PiVO command `register`. After the template has been registered, it is possible for users to create files from the new template.

```
> register cprog
> loaded
cprog
plain
dir
```

Figure 5.9: Register a new PiVO template.

In PiVO all new files must be created by specifying which template to use, see figure 5.10. It is the template that determines the file type. A user can use the command `new` and specifying the name of the template and the name of the new file.

```
> new cprog file.c
```

Figure 5.10: Create a new file from a template

The final examples of the PiVO shell, illustrates how users access and execute file operations. The command `methods` is used to list all available file operations defined for a file, see figure 5.11.

```
> methods example.txt
zip
unzip
cat
```

Figure 5.11: List the methods on a file.

When a user invokes a file operation, the `@` sign is used to separate the file name and the operation name. As seen in figure 5.11 a user uses a pipe to write the string `Hello` to the file, and executes the operation `cat` on the file `example.txt`, the result of this operation is that the data in the file is echoed to standard output stream.

```
> /bin/echo Hello > example.txt
> example.txt@cat
Hello
```

Figure 5.12: Execute the cat operation on a file.

The PiVO user interface is very simple, but it illustrates some interesting issues that a user has to deal with when using a file system that can be extended new file types. First, a user must always specify the template of a new file, and secondly a user must learn what operations exist for specific file types just as a user learn how to work with normal applications.

5.4 General discussion of PiVO

To conclude this chapter, we will discuss the PiVO prototype file system from different viewpoints, and compare PiVO with WinFS and Reiser4. The goal of this discussion is not to present a complete list of differences or similarities between the file systems, but rather identify some interesting consequences when using a file system that can be extended by new services, and how the three file systems have addressed these aspects.

5.4.1 File abstraction

A file abstraction is how a file system presents the file interfaces to other parts of the system. This could mean that an end user experiences one file abstraction while other

parts of the system see a different file abstraction.

In PiVO end users and programmers can operate differently on files. An end user has only access to a file's operations while a programmer can manipulate file attributes as well. PiVO describes a file as an instance of some ADT. This is a wider definition than the more traditional view of a file as a sequence of bytes.

In WinFS there are three different file abstractions, one for each programming API, see chapter 4.3.4. The WinFS API provides an object-oriented programming model, and the file abstraction is based on the object-oriented concept of classes. The ADO.NET API enables programmers to access WinFS data through SQL queries. In this model, files are managed as tables consisting of tuples. Finally, the Win32 API provides backward compatibility with the NTFS file abstraction.

Reiser4 successfully keeps the traditional UNIX file abstraction by representing files as sequences of bytes. However they also succeed to include the file attributes in this file abstraction.

5.4.2 File templates

The file system extension mechanism in PiVO provided by templates, can be compared to WinFS type definitions and Reiser4 plugins, since they all allow third parties to introduce new services into the file system. There are of course immense differences between the three file system extension mechanisms at implementation level, but also at conceptual level.

In PiVO we have regarded the file operations as the important extension mechanism. Templates are used to introduce new file operations and new file system services. A PiVO template can define new attributes as well, but these cannot be accessed directly by a file system user. Attributes can only be manipulated by operations.

Reiser4 plugins are implemented as a part of the Linux kernel and the plugins are targeting services that extend the internal features of the file system.

WinFS mainly focus on describing the data structure of a item. New data types can be added to the WinFS service. Data types are defined by using a data definition language. Application developers can use the data type mechanism to structure application data, and the data definitions can be used by other parts of the system, to aid the interpretation of the items.

5.4.3 Dynamic loading of templates

As stated in chapter 4 it is desirable to be able to add new file system services as easily as installing new applications on an operating system.

An extendable file system should provide simple installation mechanism for new services. In PiVO a new template can be registered at run-time, and as soon as the template is loaded it is ready to use. How this is implemented in PiVO is described in appendix A.

Both PiVO and WinFS have separated the definition of the data types and the implementation of the operations. In PiVO, a template is used to define the attributes and operations, and the implementation of the operations are placed in one or more separate modules. WinFS uses a data type definition language to describe the data type, and the implementation is placed in a class library. This architecture allows developers to reuse the implementation of operations.

The current version of Reiser4 has no support for dynamic loading of plugins, this means that developers need to modify the Reiser4 source code and recompile the Linux kernel in order to add new services. The Reiser4 development team hopes to add this feature in future version of the Reiser file system.

5.4.4 File Operations

In a traditional operating system operations on files are managed by applications that are spread out on different locations of the system. For instance the file operation `chown` in UNIX is implemented as an application, and is normally found in the `/usr/sbin/` catalog.

The fundamental idea of type specific file operations, is that a file should have knowledge of how to operate on its own data structure. File operations are implemented as a set of executable operations on the file.

The set of file operations can be extended in both PiVO and WinFS, and both systems provide a programming API for accessing and executing file operations. However, as far as we have been able to deduce from the WinFS documentation, the Windows shell (Windows Explorer) does not expose all available file operations to its users. In PiVO we allow users to examine the set of operations for a specific file.

Reiser4 relies on the VFS interfaces and therefore cannot present new file operations. This preserves the separation of files and applications. The benefit of keeping a static file interface is that users are familiar with this view of files, and this has been a very successful architecture for decades.

5.4.5 File system operations

File system services are mainly of administrative characteristic, invoked as a result of an event (for example if a file is modified) or by some schedule mechanism (example regular backup). An event could also be triggered by a predefined rule. For example, a new mail in a user's mailbox could trigger an event that results in a user notification. Today these kinds of notification services are usually implemented as applications that regularly polls the user mailbox content.

PiVO provides the possibility to add new file system services on read and write operations. As an example, the plain template always compress the file data on a write operation, and conversely decompress on a read operation.

Reiser4 has focused on file system services that extend the way files are stored. The compression service implemented in PiVO could also be implemented as a Reiser4 plugin, and thus reduce the disk utilization. The Reiser4 documentation[11] clearly states that the focus for Reiser4 is to design a file system that has a mechanism to add new services

that increase file system security.

WinFS has an API for adding new notification rules 4.3.4, so called `Watchers`. The Watchers are managed by the WinFS service and thus provides a standard solutions for notifications based on file events.

5.4.6 Typed system

In PiVO, Reiser4 and WinFS, each file is associated with one specific file type. This could be used in several file system activities. For example, a user could query the file system to get the set of files belonging to a specific file type. Moreover, the type information could be used by the operating system to associate files with different types of applications (viewers, editors, etc).

In chapter 3 we described the semantic issues of a copy operation between two files of different types. By introducing a type checking mechanism, either at file system level or at application level, we can ensure that the operands are of compatible types, or else abort the copy operation if the operands are of inappropriate types.

A file type in PiVO is used to identify (by lexical reference) the template that is associated with the file. For example, when a user asks for a list of file attributes; the PiVO core library uses the file type (template name) attribute to find the correct template and displays a list of the template specific attributes.

In PiVO we have not addressed the issues of type equivalence of files or the consequences of a type checking mechanism. WinFS, on the other hand, relies on the type checking and type equivalence mechanism provided by the object-oriented .NET runtime environment.

5.4.7 File portability

File portability will be an issue when a user needs to reproduce a file on another file system, i.e. an user expect that a file can be copied onto other disk medium. The file system must therefore provide export and import schemes for every file type known by the system.

Even though file portability is a important issue for a production file system, PiVO has not addressed problems relating to file portability, since the focus for the PiVO prototype was to examine the consequences of introducing new file operations.

WinFS has a mechanism that is called *file backing*. Each WinFS item known by the system must provide operations that know how to convert the item data structure into a well known traditional file type. For example, in WinFS a music media file can store meta data (song name, artist, etc) as item attributes. The file backing operation could be implemented to convert the item's meta data into ID3 tags[1]. The file backing mechanism must also work the other way around to allow users to import files into a WinFS volume.

Reiser4 does not have this problem since they keep the traditional file abstraction and see files as a sequence of bytes, each file in a Reiser4 file system will be handled by the default plugin.

5.4.8 Naming syntax

An application that provides access to files, such as a shell, must define a naming syntax to let users identify attributes and operations. The naming syntax also have to provide means for a user to execute file operations.

The PiVO shell has introduced the `@` sign to identify operations on a file. However, the attributes cannot be accessed directly, the template designer must provide specific operations to allow access to attributes.

The shell in Windows, called Windows Explorer, is a graphical application. Windows Explorer can present a property page of all attributes associated with a file, through which a user can edit attribute values. File operations are executed via menu choices.

As discussed in chapter 4 Reiser4 introduces the `..` prefix for all file attributes. This is a rather elegant solution to introduce file attributes into the directory name space. Since attributes are presented as files, end users operate on attributes using the same syntax

[1] ID3 tags are a meta data definition for the music archive format MP3.

notation as they do on regular files. This means that applications, such as the UNIX shell, can be used together with Reiser4 without any changes.

5.4.9 Device and special files

In a traditional UNIX file system, operations to control devices have been handled by a generic control operation (`ioctl`) that accept the specific operation id as argument.

The `tcp` device in a Linux system can be used to connect to a web server, identified by a URL and a port number, as shown in figure 5.13. Hence, it is possible to create a bidirectional file descriptor (assigned number 5) associated with the server. As a result of the commands the default web page is returned to standard output.

```
> exec 5<> /dev/tcp/www.kau.se/80
> echo "GET HTTP / 1.0" >&5
> cat <&5
```

Figure 5.13: Use `/dev/tcp` to connect to a web server

In a file system where developers can add new file types these control operations can be implemented as normal file operations. For example the `tcp` device could implement an operation `get` that takes a url as argument, as shown in figure 5.14.

```
> /dev/tcp@get http://www.kau.se:80/
```

Figure 5.14: Define a `get` method on `/dev/tcp`.

Chapter 6

Conclusion

The focus of this thesis has been to identify techniques that allow file systems to be extended with new user defined file system services and present a survey of consequences of these techniques. By looking at file system development from a historical perspective, a general description of file systems has been identified. Several contemporary file systems conform to this traditional view of file systems. As an example, the virtual file system (VFS) in UNIX systems is representative for many file systems and treats files as a sequences of bytes.

A file system can be described as a system with four distinct parts: files and operations on files, storage of files, naming of files and finally file system services. In this thesis we have discussed traditional file systems on the basis of these four parts, together with comments on possible extensions.

Case studies on two new file systems, WinFS and Reiser4, that both provide techniques to extend the file system with new services have been presented. These two file systems propose different techniques to extend the file system. Reiser4 provides a plugin mechanism to add new services, without altering the traditional view files, whereas WinFS extend the concept of files by allowing users to add new file types into the system.

Further, a prototype file system has been design and implemented. The experiences

from the design and development of a file system prototype, together with WinFS and Reiser4, have been used in a discussion concerning aspects of file system development and usage. The goal of the discussion has been to highlight consequences of file operations, dynamic loading of operations, file system operations and file system name spaces when using extendable file systems.

It is still uncertain when the extension mechanisms in WinFS and Reiser4 will be publicly available. According to Microsoft the features of WinFS will be added successively in future releases of Windows. Therefore, it is hard to predict how application providers will utilize the extension mechanisms. But, based on our experiences from WinFS and Reiser4, we believe that extendable file systems will effect application development. This is because file system services can be used as an integrated part of an application, and the new user defined services can be reused by other applications. From an end user perspective, some services will expose their features as new file operations, and this will effect the way users operate on files.

References

[1] G. Gagne A. Silberchatz, P.B. Galvin. *Operating System Concepts, sixth edition*. John Wiley & sons, inc, 2002, ISBN 0-471-41743-2.

[2] M Back, H Bhme, M Dziadzka, U Kunitz, R Magnus, D Verworner. *Linux kernel internals, Second Edition*. Addison Wesley, 1998, ISBN 0-201-33143-8.

[3] Charles Crowley. *Operating Systems A Design-Oriented Approach*. 1997, ISBN 0-256-15151-2.

[4] David K. Gifford, Pierre Jouvolot, Mark A. Sheldon, Jr James W. OToole. Semantic File Systems. *Programming Systems Reseach Group, MIT Laboratory for Computer Science*, 1991.

[5] John S. Heidemann, Gerald J. Popek. File-system development with stackable layers, 1994.

[6] Kenneth C. Louden. *Programming Languages Principles And Practice*. PWS Publishing Company, 1993, ISBN 0-534-93277-0.

[7] Massachusetts Institute of Technology, Bell Telephone Laboratory, Inc. 1965 Fall Joint Computer Conference. http://www.multicians.org/papers.html, 2004-05-31.

[8] Jim Mauro, Richard McDougall. *Solaris Internals Core Kernenl Components*. Sun Microsystems, Inc., 2001, ISBN 0-13-022496-0.

[9] Microsoft. Introducing Longhorn for Developers. http://msdn.microsoft.com, 2004-05-31.

[10] Microsoft. Longhorn SDK. http://longhorn.msdn.microsof.com/, 2004-05-31.

[11] Hans Reiser. Reiser4. http://www.namesys.com/v4/v4.html, 2004-05-31.

[12] Dennis M. Ritchie. The evolution of the Unix Time-sharing System, 1984.

[13] A. S. Tanenbaum. *Operating Systems: Design and Implementation*. 1997, ISBN 0-13-638677-6.

Appendix A

Overview of PiVO source code

This section presents the source code for the prototype file system PiVO. The code is written in the C programming language and has been tested on Mac OS X 10.3.2. The lexical analyzer generator `flex` and the parser generator `bison` is used to generate C code for the shell, a simple command language interpreter. The internal representation of files in PiVO is managed using the embedded database library Berkeley DB 4.1.25 from Sleepy Cat Software.

The code should be fairly easy to port to another UNIX based operating system, since the only platform dependent part of the implementation is technique used to build dynamic libraries.

The basic design of PiVO is divided in to four distinct parts, or modules, as illustrated in figure A.1.

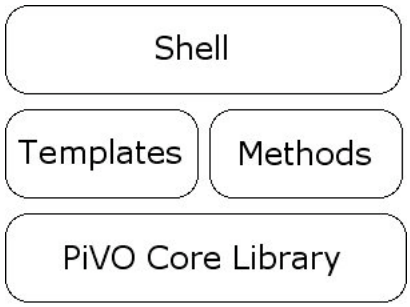

Figure A.1: PiVO overview

- `Shell`
  The `Shell` module represents a simple command language interpreter, or shell, and the source code for the module is listed in appendix C. When the shell is started it creates an initial file system that is the root of all user created files and directories.

- `Templates`
  The code for the templates is presented in appendix D. A template is a PiVO file including the definition of the set of attributes and operations that are unique for each file type. In addition to file type operations, the template can also include definitions of the file system operations. Templates are compiled as dynamic libraries and are stored in a predefined directory on the underlying host operating system. When a user issues the shell command `register` the set of attributes and operations in the dynamic library is copied to a PiVO file. Once the set is copied, the generated PiVO file can be used to create new files.

  There are three templates defined in PiVO, `plain` represents a normal text file, `cprog` is used C-program files, and `dir` is used to create directories. File system services are also defined in the templates.

- `Methods`
  A template define file operations and file system services. But the implementation of the operations is defined in one or more separate modules. This makes is possible for different templates to share operation implementations. The file and file systems operations are stored on the underlying operating system as dynamic libraries, and are loaded on demand when an operation is executed. Appendix E lists the source code for the various operations in PiVO.

- `PiVO Core Library`
  The PiVO Core Library is responsible for a diversity of services. It defines the predefined set of attributes and operations of all files and a programming interface to operate on the set of attributes and operations. It is responsible for the management of the internal set of registered templates. It also manages the on disk representation of files, and implements an internal name space in order to uniquely identify stored files. This diverse functionality is managed by a set of modules, and an illustration of the modules is shown in figure A.2.

  Note that the modules `templates` and `methods` also are included in figure A.2 to illustrate how these modules fit in to the design.

  The predefined set of attributes and operations is defined in the `file` module. The programmer interface is defined in appendix F.8 and the implemention is listed in appendix F.9.

  The creation of templates and the internal collection of templates is managed by the module `template`.

  A file in PiVO is stored on the underlying operating system in a directory called *pool*. The directory is represented by the disk like symbol in figure A.2. The module `object` is responsible for the on disk storage of all files, and it utilize the embedded database library Berkeley DB to store the files on disk. This means that a file in

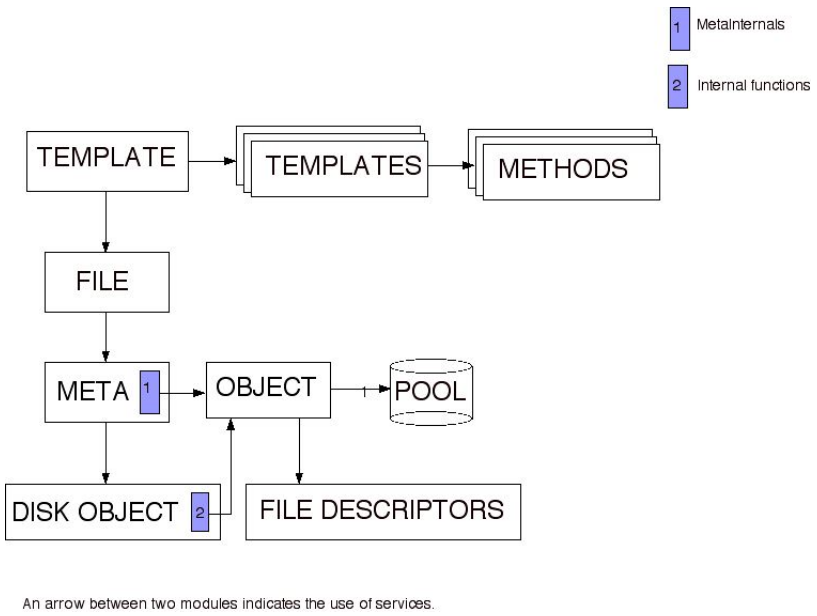

Figure A.2: Details of Core Library

PiVO is a Berkeley DB database. A Berkeley DB database is stored as a file on the underlying host operating system, and the file name of the database is the file descriptor of the PiVO file that is represented by the database. A file descriptor is a unique integer value assigned to the file when the file is created. The assignment of the file descriptor is managed by the module `file descriptors`.

The details of attribute access and modification are hidden from the module `file`, and are managed in the module `meta`. The module `meta` is further divided into a module called `metainternals`, but is illustrated in figure A.2 as one module.

The last module in figure A.2 `disk object` is a dummy module, but in a real world implementation of a file system, this module would be responsible for the on disk storage. The module is included only to illustrate the different modules that have to be present in a real file system.

A.1 Limitations

Since PiVO is a prototype used as reference in the discussion of design techniques in section 5.4 it has limitations. The implementation is limited in the following areas:

- Type check system
  PiVO does not implement a type check system for attribute values. An attribute is

assumed to store all its values as strings, except for the value of the attribute '_data' which is assumed to be of type 'Data' (see G.8 for a definition of the data type).

- On disk storage
  PiVO is implemented as a user space application on top of an underlying host operating system and it utilizes the file system on the underlying host to store its files. The details of how a PiVO file is stored on the file system is managed by an embedded database library.

- Sessions
  The file system created by the shell on start up is not persistent over shell sessions. This means that each shell session creates its own file system, and files created during a session cannot be accessed once the shell terminates.

- Error handling and stability
  PiVO is not designed to be as stable as a real file system, and because of this the amount of error handling in the source code is at a minimum.

- Performance
  Performance was never an issue when designing PiVO, and this has effected the implementation in that it includes a dummy module.

- Symbolic links
  The implementation does not support links between files.

- Shell commands
  The shell is used to create new files, load new templates and execute file operations. Therefore the set of supported user commands is at a minimum.

Appendix B

The execution of a file operation

One of the central ideas in PiVO is that template designers define operations on files, that are executable by application programmers and shell users. This section describes the internal functions in PiVO that are involved when an operation is executed. The description is based on the execution of the compression operation 'zip'. The internal functions presented in this section are simplified to make it easier to read and understand the code.

All file operations are defined in a template. The definition of the 'zip' operation is defined in the `plain` template and is shown in figure B.1. An operation is defined using two attributes. The first attribute defines the name of the operation, and is visible to users. All operations have a lexical name that corresponds to the name of the attribute. Internally the lexical name of the operation is replaced by an integer value. The integer value of an operation is defined in the value field of an attribute. The lexical name and internal integer representation of the 'zip' operation are defined on line 6 in figure B.1.

The second attribute is not visible to users and defines the path to a dynamic library that contains the implementation of the operation. The dynamic library represents one of the method modules shown in figure A.2. The second attribute has the same name as the operation, but is postfixed with the string '_m'. The definition of the library path to the 'zip' operation is shown on line 7 in figure B.1.

```
1 #define ZIP_STR "3"
2 #define ZIP_METHODS "/PiVO/methods/zip_methods.so"
3
4 /* Attributes and operations */
5 Private struct opattr ops_attrs[] = {
6   { "zip", ZIP_STR },
7   { "zip_m", ZIP_METHODS },
8 };
```

Listing B.1: Template definition of the zip operation in plain.c

The code segment shown in figure B.2 opens a file and executes the file operation 'zip' on the files data. The file interface is defined in `file.h` and it is accessible to a programmer through a file object. The file object is created in line 1, and associated to the file on line 6. Once the file object is associated with a file it is possible to apply file operations on the file. The predefined operation `exec_m()` on line 7 calls the template defined operation 'zip'. The name of the operation to execute is given as a string argument to `exec_m()`.

```
1  File f = new_file();
2  int fd = lookup_fd(name);
3  String file_operation = new_string();
4
5  file_operation->set(file_operation, "zip");
6  f->open(f, fd);
7  f->exec_m(f, file_operation);
8  f->close(f);
```

Listing B.2: Program code to execute the zip operation.

The predefined operation `exec_m()` is a pointer to the function `exec_m_file()`, and the code for `exec_m_file()` is shown in figure B.3. The function `exec_m_file()` is responsible to:

- Identify the name of the template associated with the file.
  The template name is used to identify which operation interpreter function to call. An operation interpreter function is defined in each template and it is responsible for loading and executing the operations defined in the template.

- Read the value of the attribute 'zip_m'
  The value of the attribute 'zip_m' is the absolute path to the dynamic library containing the operation function.

- Read the value of the attribute 'zip'
  The value of the 'zip' attribute is the internal integer representation of the 'zip' operation. The integer value is converted from a string representation to an integer value.

- Call the function `exec()` defined in the template module.
  As mentioned above, an operator is executed by the operation interpreter function and is defined in the template. The call to `exec()` is used to locate the files template in PiVO's internal collection of registered templates and execute the operation interpreter function.

```
1  Private boolean exec_m_file(File this, String method, ...)
2  {
3    int m_nr; /* Internal representation of file operation */
4
5    /* Get the name of the template */
6    String template = get_template_file(this);
7    char *name_of_template = template->get(template);
8
9    String method_m = new_string();
10   Data data;
11   Template t = new_template();
12   va_list args;
13
14   /* Append the string "_m" to the parameter method */
15   method_m->set(method_m, method->copy_str(method));
16   method_m = method_m->append_char(method_m, "_m");
17
18   /* Get the metod number */
19   data = get_attr_file(this, method);
20   m_nr = atoi(data->get(data));
21
22   /* Get the name of the method library to load */
23   data = get_attr_file(this, method_m);
24
25   /* Execute the method */
26   retval = t->exec(str, this, m_nr, data, &args);
27 }
```

Listing B.3: The function exec_m_file defined in file.c.

The function `exec()` on line 26 in figure B.3 is once again a function pointer, this time to the function `exec_template()`. The function `exec_template()` locates the template in PiVO's internal collection of templates, and calls the operation interpreter function defined in the template. The template is located in the list by the call to the function `get()` on line 8 in figure B.4. The call to the operation interpreter function is defined on line 11 in the same figure.

```
1  Private boolean exec_template(char * name_of_template, File f, int m_nr, \
2                                Data data, va_list *args)
3  {
4    Element el;
5    TemplateItem ti_el = new_templateitem();
6    TemplateItem ti;
7    ti_el->set_name(ti_el, name_of_template);
8    el = templateList->get(templateList, ti_el, _cmp_by_name);
9    ti = (TemplateItem)el->get_value(el);
10
11   return ti->exec(f, m_nr, data, args);
12 }
```

Listing B.4: The function exec defined in template.c.

The operation interpreter function for the plain template is shown in figure B.5. The function has two responsibilities, to load the dynamic library containing the function for the operation, and to execute the operation once loaded. The function consists of a switch statement that selects the correct behavior according to the `m_nr` parameter. The value of the `m_nr` parameter is the internal integer value of the operation to execute. The 'zip' operation is identified on line 8, and the call to the compression function is defined on line 12 in the same figure.

```
1  #define ZIP 3
2
3  Public boolean plain_exec(File f, int m_nr, Data data,  va_list *args)
4  {
5     void *handle = open_library(data->get(data));
6
7    switch(m_nr){
8     case ZIP:
9      {
10       boolean (*zip)(File f);
11       zip = load_library_symbol(handle, "zip");
12       if(zip != NULL){ return (*zip)(f); }
13       else { printf("\nInternal error while executing zip\n"); }
14     }
15     break;
16   }
17 }
```

Listing B.5: Interpretation of the file operations defined in plain.c.

Finally, the function for the 'zip' operation is called. The the code for the compression function is shown in figure B.6.

```
1  Public boolean zip(File f)
2  {
3    compress(dest, &dlen, source, slen);
4  }
```

Listing B.6: Compression of the data in a file defined in zip_methods.c.

Appendix C

Shell

In this section the source code for the command language interpreter, or shell, is presented. The shell is implemented using `Lex` and `Yacc`. When the shell is started it loads two templates named `plain` and `dir`. Templates are used when a user creates new files, or directories [1]. Next, the shell creates an initial file system, this is done by creating a directory using the `dir` template. The created directory is named *root* and is the parent of all subsequently created directories.

[1] A directory is a file with a listing of the files in the directory stored as its data.

C.1 grammar.l

```
/*--------------------------------------------------------------------------
 * Lexical definitions for the command shell.
 */

%{
#include "grammar.tab.h"
#include <stdio.h>
#include <string.h>

extern void print_prompt();
%}


HELP            "help"|"h"
REGISTER        "register"|"r"
LOADED          "loaded"|"l"
LS              "ls"
CD              "cd"
NEW             "new"|"n"
MKDIR           "mkdir"
METHODS         "methods"|"m"
ATTRIBUTES      "attributes"|"a"
QUIT            "quit"|"q"
DOT             "@"
PIPE            ">"
NL              [\n\r]
WS              [ \t]+
WORD            [a-zA-Z\._/]+
ST              [=]
UNDEF           .

%%

{WS}            { }
{NL}            { return EXEC;        }
{DOT}           { return DOT;         }
{PIPE}          { return PIPE;        }
{HELP}          { return HELP;        }
{REGISTER}      { return REGISTER;    }
{LOADED}        { return LOADED;      }
{LS}            { return LS;          }
{CD}            { return CD;          }
{NEW}           { return NEW;         }
{MKDIR}         { return MKDIR;       }
{METHODS}       { return METHODS;     }
{ATTRIBUTES}    { return ATTRIBUTES;  }
{QUIT}          { return QUIT;        }
{WORD}          { yylval.str = strdup(yytext); return WORD;      }
{UNDEF}         { printf("Illegal character '%c'\n", *yytext); return UNDEF; }

%%
```

C.2 grammar.y

```
/*-------------------------------------------------------------------------
 * Grammar definition for the command shell.
 */

%{
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "pivo.h"
#include "shell.h"
#include "command.h"

extern FILE *yyin;
Private boolean QUIET = FALSE;

Public int yyerror(char *str);
Public int yylex();
Public int yyparse();
Private void print_prompt();

%}

%union {
        char *str;
        struct alist *args;
        struct cmd *cmd;
}

%token WORD HELP REGISTER LOADED LS NEW METHODS ATTRIBUTES QUIT NL DOT
%token UNDEF EOF EXEC CD MKDIR PIPE

%token OP NATIVE

%type <str> WORD
%type <cmd> cmd
%type <args> args

%%

prompt:
  | prompt stmt { print_prompt();}

;

stmt:
    cmd EXEC                    { cmd_exec($1); }
  | cmd PIPE WORD EXEC          { $1->target = lookup_fd($3);
                                  cmd_pipe_exec($1); }
  | EXEC                        { }
;

cmd:
    CD args                     { $$ = cmd_new(CD, $2); }
  | HELP                        { $$ = cmd_new(HELP, NULL); }
  | REGISTER args               { $$ = cmd_new(REGISTER, $2); }
  | LOADED                      { $$ = cmd_new(LOADED, NULL); }
  | LS                          { $$ = cmd_new(LS, NULL); }
```

```
    | NEW  args                    { $$ = cmd_new(NEW, $2); }
    | MKDIR args                   { $$ = cmd_new(MKDIR, $2); }
    | METHODS args                 { $$ = cmd_new(METHODS, $2);}
    | ATTRIBUTES args              { $$ = cmd_new(ATTRIBUTES, $2); }
    | WORD DOT args                { $$ = cmd_new(OP, $3);
                                     $$->operand = lookup_fd($1); }
    | args                         { $$ = cmd_new(NATIVE, $1); }
    | QUIT                         { exit(0); }
;

args:
    args  WORD                     { alist_add($$, $2); }
  | WORD                           { $$ = alist_new(); alist_add($$, $1); }
;

%%

Private void print_prompt()
{
  if(QUIET == TRUE) { return ; }
  printf("6.100> ");
  fflush(NULL);

  return ;
}

Public int yyerror(char *str) { printf("\nOops!\n"); return 0; }

Public int yywrap()
{
  yyin = stdin;
  QUIET = FALSE;
  print_prompt();

  return 0;
}

Public int main(int argc, char *argv[])
{
  /* Load templates and set up root directory. */
  create_fs();

  /* Load optional shell init file*/
  if (argc == 2){
    FILE *initfile = fopen(argv[1], "r");
    if (initfile != NULL){
      yyin = initfile;
      QUIET = TRUE;
    }
  }

  /* Print the first prompt. */
  print_prompt();
  /* Start interactive session. */
  yyparse();

  return 0;
}
```

C.3 command.h

```
/*-----------------------------------------------------------------------------
 * Defines the command interface.
 */

#ifndef __COMMAND_H__
#define __COMMAND_H__
#include "pivo.h"

typedef struct item {
  char *str;
  struct item *next;
} item;

typedef struct alist {
  int size;
  item *first;
  item *last;
} alist;


typedef struct cmd {
  int op;
  int operand;
  int target;
  alist *args;
} cmd;

Public alist *alist_new();
Public void alist_add(alist *l, char *str);
Public void alist_prepend(alist *l, char *str);
Public char *alist_index(alist *l, int idx);
Public void alist_print(alist *l);
Public char **alist_array(alist *l);

cmd *cmd_new(int op, alist *args);

#endif
```

C.4 command.c

```
/*-------------------------------------------------------------------------
 * Implements the command interface.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "command.h"


Public alist *alist_new()
{
  alist *l = (alist*)calloc(1, sizeof(struct alist));
  if(l==NULL){
    fprintf(stderr, "Failed to allocate alist struct.\n");
  }

  l->size = 0;
  l->first = NULL;
  l->last = NULL;

  fflush(NULL);

  return l;
}

Public void alist_add(alist *l, char *str)
{
  item *i = (item*)calloc(1, sizeof(struct item));
  if(i==NULL){
    fprintf(stderr, "Failed to allocate alist item struct.\n");
  }
  i->str = strdup(str);
  i->next = NULL;

  if(l->last == NULL){
    l->last = i;
    l->first = i;
  }
  else{
    l->last->next = i;
    l->last = i;
  }

  l->size += 1;

  return ;
}

Public void alist_prepend(alist *l, char *str)
{
  item *i = (item*)calloc(1, sizeof(struct item));
  if(i==NULL){
    fprintf(stderr, "Failed to allocate alist item struct.\n");
  }
  i->str = strdup(str);
  i->next = NULL;
```

```
  if(l->first == NULL){
    l->last = i;
    l->first = i;
  }
  else{
    i->next = l->first;
    l->first = i;
  }
  printf("list size %d\n", l->size);
  l->size += 1;
}

Public char *alist_index(alist *l, int idx)
{
  item *i = l->first;
  for(;idx>0;idx--){
    if (i == NULL){ return NULL;}
    i = i->next;
  }

  if(i==NULL){ return NULL;}

  return i->str;
}

Public char **alist_array(alist *l)
{
  char **result;
  int i;

  result = (char**)calloc(l->size+1, sizeof(char**));

  for(i=0 ; i<l->size ; i++){
    result[i] = alist_index(l, i);
  }
  result[i] = NULL;

  return result;
}

Public void alist_print(alist *l)
{
  item *idx = l->first;
  while(idx != NULL){
    printf("%s\n", idx->str);
    idx = idx->next;
  }

  return ;
}

Public cmd *cmd_new(int op, alist *args)
{
  cmd *c = (cmd*)calloc(1, sizeof(struct cmd));
  c->op = op;
  c->args = args;

  return c;
}
```

C.5 shell.h

```
/*-----------------------------------------------------------------------------
 * Declares the root file system interface.
 */

#ifndef __SHELL__H
#define __SHELL__H

#include "pivo.h"
#include "command.h"

struct files {
  char *name;
  int fd;
};

Public File cwd;                /* Current working directory */

Public int create_fs();
Public int lookup_fd(char *name);
Public int cmd_pipe_exec(cmd *c);
Public int cmd_exec(cmd *c);


#endif
```

C.6 shell.c

```
/*------------------------------------------------------------------------
 * Implementation of the command shell.
 *
 * The command shell creates a root file system, and can be used to create
 * new files and directories in the root file system.
 * All created files and directories are created in the system catalog pool
 * (see libsrc/build_pool.h for details).
 * The command shell do NOT clean the pool catalog upon exit. This is usefull
 * when debugging the system.
 */

#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdarg.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include "grammar.tab.h"
#include "pivo.h"
#include "shell.h"
#include "templatepath.h"
#include "command.h"


struct dir_entry {
  int fd;
  char name[200];
};

struct help_text {
  char *cmd;
  char *doc;
};

Private struct help_text help_txt[] = {
  { "help",       "Print help" },
  { "register",   "Register template" },
  { "loaded",     "List registered templates" },
  { "ls",         "List files" },
  { "new",        "Create new file" },
  { "methods",    "List file methods" },
  { "attributes", "List file attributes" },
  { "quit",       "Quit the shell" },
  { NULL,          NULL }
};

Private List templates = NULL;  /* List of loaded templates */
Private File root;              /* File system root         */

/*------------------------------------------------------------------------
 * Private function declarations.
 */

Private int create_file(char *template, char *name);
Private int create_dir(char *name);
```

```
Private int lookup_template(char *template);
Private int add_file(File f);
Private int print_help(void);
Private int register_template(char *name);
Private int print_registered_templates(void);
Private int list_files(void);
Private int list_methods(char *name);
Private int list_attributes(char *name);
Private int exec(int fd, char *method, char *args);
Private int change_dir(char *name);
Private int exec_native(cmd *c);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public int create_fs()
{

  Template t = new_template();

  register_template("dir");
  register_template("plain");

  root = t->create_file("dir", "root");

  if(root->fd < 0){
    fprintf(stderr, "Could not create root directory, give up!");
    exit(1);
  }

  cwd = root;

  return 1;
}

Public int lookup_fd(char *name)
{

  struct dir_entry entries[200];
  String m = new_string();
  int index = 0;
  int i = 0;

  m = m->set(m, "get");

  /* Populate the entries struct with content of cwd. */
  cwd->exec_m(cwd, m, entries, &index);

  while(i <= index){
    /* If name is found, return the fd of that entry. */
    if(strcmp(entries[i].name, name) == 0){
      return entries[i].fd;
    }
    i++;
  }

  /* File name not found in cwd. */
  m->destroy(m);

  return -1;
}
```

```
Public int cmd_pipe_exec(cmd *c)
{
  int fd[2];
  int pid;

  if(pipe(fd)==-1){
    perror("failed to create pipe.");
    exit(1);
  }

  if((pid=fork())==-1){
    perror("failed to fork");
    exit(1);
  }
  else if(pid==0){
    /* child process */
    close(fd[0]);
    dup2(fd[1], 1);
    cmd_exec(c);
    close(1);
    exit(0);
  }
  else{
    /* parent process */
    char buf[1024];
    int len;
    Data dbuf = new_data();
    String sbuf = new_string();
    File out = new_file();
    out->open(out, c->target);
    bzero(buf, 1024);
    close(fd[1]);

    while((len=read(fd[0], buf, 1024))>0){
      buf[len] = '\0';
      sbuf->append_char(sbuf, buf);
      printf("Size :%d\n", sbuf->size(sbuf));
    }

    dbuf->set(dbuf, sbuf->get(sbuf), sbuf->size(sbuf));
    out->write(out, dbuf);
    out->close(out);
    wait(&pid);
    sbuf->destroy(sbuf);
    dbuf->destroy(dbuf);
    out->destroy(out);
  }

  return 1;
}

Public int cmd_exec(cmd *c)
{
  switch(c->op){
  case HELP:
    print_help();
    break;
  case CD:
    change_dir(alist_index(c->args, 0));
    break;
  case REGISTER:
```

```
      register_template(alist_index(c->args, 0));
      break;
    case LOADED:
      print_registered_templates();
      break;
    case LS:
      list_files();
      break;
    case NEW:
      create_file(alist_index(c->args, 0), alist_index(c->args, 1));
      break;
    case MKDIR:
      create_dir(alist_index(c->args, 0));
      break;
    case METHODS:
      list_methods(alist_index(c->args, 0));
      break;
    case ATTRIBUTES:
      list_attributes(alist_index(c->args, 0));
      break;
    case OP:
      exec(c->operand, alist_index(c->args, 0), alist_index(c->args, 1));
      break;
    case NATIVE:
      exec_native(c);
      break;
    }
    return 1;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private int create_file(char *template, char *name)
{
    int retval;
    File new_file;
    Template t = new_template();

    /* Abort operation if the template doesn't exists. */
    if(lookup_template(template) == 0){
      printf("Unknown template %s\n", template);
      return 0;
    }

    /* Abort operation if name already exists in cwd. */
    if(lookup_fd(name) >= 0){
      printf("%s already exists.\n", name);
      return 0;
    }

    /* Create file. */
    if((new_file = t->create_file(template, name)) < 0){
      printf("Error creating %s\n", name);
      return 0;
    }

    /* Add the new file to cwd. */
    if(add_file(new_file) == 0){
      printf("Error add %s to cwd\n", name);
      return 0;
```

```
    }

    retval =  new_file->fd;
    new_file->destroy(new_file);

    return retval;
  }

  Private int create_dir(char *name)
  {

    File f = new_file();
    String m = new_string();
    int fd = create_file("dir", name);

    if (fd < 0){
      fprintf(stderr, "Could not create directory.\n");
      return 0;
    }

    m = m->set(m, "move");

    /* Add the cwd fd as a dir entry in the new directory. */
    f->open(f, fd);
    f->exec_m(f, m, cwd);
    f->close(f);

    m->destroy(m);
    f->destroy(f);

    return fd;
  }

  Private int add_file(File f)
  {

    String m = new_string();

    /* Add the new file to the entry list in cwd. */
    m = m->set(m, "add");
    if(f->fd < 0) {
      fprintf(stderr, "Could not add: fd %d \n", f->fd);
      return 0;
    }
    cwd->exec_m(cwd, m, f);
    m->destroy(m);

    return 1;
  }

  Private int lookup_template(char *template)
  {
    Element el = new_element();

    if(templates == NULL){  printf("No registered templates\n"); return 0; }
    el = templates->get_first(templates);
    if(el == NULL){ printf("No registered templates\n"); return 0; }

    if(strcmp(template, (char *)el->get_value(el)) == 0){ return 1; }
    while((el = templates->get_next(templates)) != NULL){
      if(strcmp(template, (char *)el->get_value(el)) == 0){
        return 1;
```

```
    }
  }

  return 0;
}

Private int change_dir(char *name)
{
  int dir_fd = lookup_fd(name);

  if (dir_fd != -1){
    cwd->close(cwd);
    cwd->open(cwd, dir_fd);
  } else {
    printf("cd: no such directory %s\n", name);
    return 0;
  }

  return 1;
}

Private int print_help(void)
{
  int i = 0;

  while(help_txt[i].cmd != NULL){
    printf("%-15s %-15s\n", help_txt[i].cmd, help_txt[i].doc);
    i++;
  }

  return 0;
}

Private int register_template(char *name)
{
  String path = new_string();
  String suffix = new_string();

  path->set(path, TEMPLATE_PATH);
  suffix->set(suffix, "_template.so");

  path = path->append_char(path, name);
  path = path->append_char(path, suffix->get(suffix));

  if(load_template(path->get(path), name) == FALSE){
    printf("Unable to load template:%s\n", name);
    return 0;
  }
  if(templates != NULL){ templates->delete_all(templates); }
  templates = get_templates();

  path->destroy(path);
  suffix->destroy(suffix);

  return 1;
}

Private int  print_registered_templates(void)
{
  Element el = new_element();
  if(templates == NULL){  printf("No registered templates\n"); return 0; }
  el = templates->get_first(templates);
```

```
    if(el == NULL){ printf("No registered templates\n"); return 0; }
    printf("%s\n", (char *)el->get_value(el));
    while((el = templates->get_next(templates)) != NULL){
      printf("%s\n", (char *)el->get_value(el));
    }

    return 1;
  }

  Private int list_files(void)
  {

    struct dir_entry entries[200];

    File f = new_file();
    String m = new_string();
    int index = 0;
    int i = 0;

    m = m->set(m, "get");

    cwd->exec_m(cwd, m, entries, &index);

    if(entries == NULL){
      fprintf(stderr, "Error reading files in cwd.");
      return 0;
    }

    while(i < index){
      printf("%s\n", entries[i].name);
      i++;
    }

    m->destroy(m);
    return 1;
  }

  Private int list_methods(char *name)
  {
    File f = new_file();
    char *method;
    int fd;

    fd = lookup_fd(name);
    if(fd == -1){ printf("Unknown file %s\n", name); return 0; }
    f->open(f, fd);
    f->init_list(f);
    while((method = f->get_methods(f)) != NULL){
      printf("%s\n", method);
    }
    f->close_list(f);
    f->close(f);

    return 1;
  }

  Private int list_attributes(char *name)
  {
    File f = new_file();
    char *attr;
    int fd;
```

```
    fd = lookup_fd(name);
    if(fd == -1){ printf("Unknown file %s\n", name); return 0; }
    f->open(f, fd);
    f->init_list(f);
    while((attr = f->get_attributes(f)) != NULL){
      printf("%s\n", attr);
    }
    f->close_list(f);
    f->close(f);

    return 1;
  }


  Private int exec(int fd, char *method, char *args)
  {
    File f = new_file();
    String m = new_string();
    m->set(m, method);

    if(fd == -1){ printf("Unknown file\n"); return 0; }
    f->open(f, fd);
    f->exec_m(f, m, args);
    f->close(f);

    m->destroy(m);

    return 1;
  }

  Private int exec_native(cmd *c)
  {
    char **cmd = alist_array(c->args);
    int pid;
    if((pid=fork())==-1){
      perror("failed to fork");
      exit(1);
    }
    else if(pid==0){
      /* child process */

      execv(cmd[0], cmd);
      printf("Unknown command %s, use full path\n", cmd[0]);
      exit(0);
    }
    else{
      wait(&pid);
    }
    return 1;
  }
```

Appendix D

Templates

This section presents the source code for the templates `plain`, `cprog` and `dir`. The templates represents the different file types in PiVO, and are used when creating new files. The template `plain` represents a normal text file. The contents of a `plain` file is compressed by default by PiVO using the file system methods #a_open and #a_close. The contents of a `plain` file can be listed using the file operation `cat`.

The template `cprog` represents C-program source code files, and its contents can be listed using the file operation `cat`. The `cprog` template do not define any compression function, and no file system operation is defined for file derived from the `cprog` template.

It is not possible to compress the contents of a `cprog` file.

The last template, `dir`, represents an external name space, and defines files that can be used as containers for other files. That is, the `dir` template is used to define a simple directory structure.

D.1 plain.c

```
/*-------------------------------------------------------------------------
 * Defines the template for the file type plain.
 */

#include <string.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include "pivo.h"
#include "methods.h"

#define PLAIN_TEMPLATE "plain"

#define CAT 0
#define A_OPEN 1
#define A_CLOSE 2
#define ZIP 3
#define UNZIP 4

#define CAT_STR "0"
#define A_OPEN_STR "1"
#define A_CLOSE_STR "2"
#define ZIP_STR "3"
#define UNZIP_STR "4"

struct opattr {
  char *name;
  char *value;
};

/* Attributes and operations */
Private struct opattr ops_attrs[] = {
  { "Documentation", "Plain text file" },
  { "Usize", "0" }, /* Compression flag       */
  { "Csize", "0" }, /* Compressed file size   */
  { "cat", CAT_STR },
  { "cat_m", CAT_METHOD },
  { "#a_open", A_OPEN_STR },
  { "#a_open_m", A_OPEN_METHOD },
  { "#a_close", A_CLOSE_STR },
  { "#a_close_m", A_OPEN_METHOD },
  { "zip", ZIP_STR },
  { "zip_m", ZIP_METHODS },
  { "unzip", UNZIP_STR },
  { "unzip_m", ZIP_METHODS },
  { NULL, NULL}
};

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private void _add(Template t, char *operation, char *data);

/*-------------------------------------------------------------------------
 * Public functions.
 */
```

```
/* Execute a file operation or a file system operation */
Public boolean plain_exec(File f, int m_nr, Data data,  va_list *args)
{
  boolean retval = TRUE;
  char *tmp = data->get(data);
  void *handle = open_library(tmp);
  free(tmp);

  switch(m_nr){
  case CAT:
    {

      void (*cat)(File f);
      cat = load_library_symbol(handle, "cat");
      if(cat != NULL){ (*cat)(f); retval = TRUE; }
      else {
        printf("\nInternal error while executing cat\n");
      }
    }
    break;
  case A_OPEN:
    {
      void (*a_open)(File f);
      a_open = load_library_symbol(handle, "_a_open");
      if(a_open != NULL){ (*a_open)(f); retval = TRUE; }
      else {
        printf("\nInternal error while executing a_open\n");
      }
    }
    break;
  case A_CLOSE:
    {
      void (*a_close)(File f);
      a_close = load_library_symbol(handle, "_a_close");
      if(a_close != NULL){ (*a_close)(f); retval = TRUE; }
      else {
        printf("\nInternal error while executing a_close\n");
      }
    }
    break;
  case ZIP:
    {
      boolean (*zip)(File f);
      zip = load_library_symbol(handle, "zip");
      if(zip != NULL){ return (*zip)(f); }
      else {
        printf("\nInternal error while executing zip\n");
      }
    }
    break;
  case UNZIP:
    {
      boolean (*unzip)(File f);
      unzip = load_library_symbol(handle, "unzip");
      if(unzip != NULL){ return (*unzip)(f); }
      else {
        printf("\nInternal error while executing unzip\n");
      }
    }
    break;
  default:
```

```
      printf("Unknown method\n");
      retval = FALSE;
      break;
    }
    close_library(handle);

    return retval;
  }

  Public boolean plain_create(void)
  {
    int i = 0;

    Template t = new_template();
    t->create(PLAIN_TEMPLATE, plain_exec);

    /* Add operations & attributes */
    while(ops_attrs[i].name != NULL){
      _add(t, ops_attrs[i].name, ops_attrs[i].value);
      i++;
    }
    return TRUE;
  }


  /*-------------------------------------------------------------------------
   * Private functions.
   */

  Private void _add(Template t, char *operation, char *data)
  {
    int size = strlen(data);
    String attr = new_string();
    Data d = new_data();

    /* Add the operation or attribute */
    attr->set(attr, operation);
    d->set(d, data, size);
    t->add_attribute(PLAIN_TEMPLATE, attr, d);
    attr->destroy(attr);
    d->destroy(d);

    return ;
  }
```

D.2 cprog.c

```
/*-------------------------------------------------------------------------
 * Defines the template for the file type c-program.
 */

#include <string.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include "pivo.h"
#include "methods.h"

#define CPROG_TEMPLATE "cprog"

#define CAT 0

#define CAT_STR "0"

struct opattr {
  char *name;
  char *value;
};

/* Attributes and opertations */
Private struct opattr ops_attrs[] = {
  { "Documentation", "C program source file" },
  { "cat", CAT_STR },
  { "cat_m", CAT_METHOD },
  { NULL, NULL}
};


/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private void _add(Template t, char *operation, char *data);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public boolean cprog_exec(File f, int m_nr, Data data,  va_list *args)
{
  boolean retval = TRUE;
  char *tmp = data->get(data);
  void *handle = open_library(tmp);
  free(tmp);

  switch(m_nr){
  case CAT:
    {
      void (*cat)(File f);
      cat = load_library_symbol(handle, "cat");
      if(cat != NULL){ (*cat)(f); retval = TRUE; }
      else {
        printf("\nInternal error while executing cat\n");
      }
```

```
      }
      break;
    default:
      printf("Unknown method\n");
      retval = FALSE;
      break;
    }
    close_library(handle);

    return retval;
}

Public boolean cprog_create(void)
{
    int i = 0;

    Template t = new_template();
    t->create(CPROG_TEMPLATE, cprog_exec);

    /* Add operations & attributes */
    while(ops_attrs[i].name != NULL){
      _add(t, ops_attrs[i].name, ops_attrs[i].value);
      i++;
    }

    return TRUE;
}


/*---------------------------------------------------------------------------
 * Private functions.
 */

Private void _add(Template t, char *operation, char *data)
{
    int size = strlen(data);
    String attr = new_string();
    Data d = new_data();

    /* Add the operation or attribute */
    attr->set(attr, operation);
    d->set(d, (char *)strndup(data, size), size);
    t->add_attribute(CPROG_TEMPLATE, attr, d);
    attr->destroy(attr);
    d->destroy(d);

    return ;
}
```

D.3 dir.c

```
/*-------------------------------------------------------------------------
 * Defines the template for the file type directory.
 */

#include <string.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include "pivo.h"
#include "dirmethods.h"

#define DIR_TEMPLATE "dir"

/* Operation number */
#define ADD 0
#define DELETE 1
#define GET 2
#define MOVE 3

/* Operation numer as string */
#define ADD_STR "0"
#define DELETE_STR "1"
#define GET_STR "2"
#define MOVE_STR "3"

#define ZERO "0"
#define NROFF 200

struct opattr {
  char *name;
  char *value;
};

/* Attributes and operations */
Private struct opattr ops_attrs[] = {
  { "Documentation", "Directory" },
  { "add", ADD_STR },
  { "add_m", DIR_METHODS },
  { "del", DELETE_STR },
  { "del_m", DIR_METHODS },
  { "get", GET_STR },
  { "get_m", DIR_METHODS },
  { "move", MOVE_STR },
  { "move_m", DIR_METHODS },
  { "Nroff", ZERO },
  { NULL, NULL}
};

struct dir_entry {
  int fd;
  char name[NROFF];
};

Private struct dir_entry entries[200];

/*-------------------------------------------------------------------------
 * Private function declarations.
```

```
 */

Private void _add(Template t, char *operation, char *data);
Private void _add_direntry(Template t);

/*---------------------------------------------------------------------------
 * Public functions.
 */

/* Execute operations */
Public boolean dir_exec(File f, int m_nr, Data data,  va_list *args)
{
  boolean retval = TRUE;
  char *tmp = data->get(data);
  void *handle = open_library(tmp);
  free(tmp);

  switch(m_nr){
  case ADD:
    {
      File newfile = va_arg(*args, File);
      boolean (*add)(File f, File newfile);
      add = load_library_symbol(handle, "add");
      if(add != NULL){ return (*add)(f, newfile); }
      else { printf("\nInternal error while executing add\n"); }
    }
    break;
  case DELETE:
    {
      File target = va_arg(*args, File);
      boolean (*delete)(File f, File target);
      delete = load_library_symbol(handle, "delete");
      if(delete != NULL){ return (*delete)(f, target); }
      else { printf("\nInternal error while executing delete\n"); }
    }
    break;
  case GET:
    {
      struct dir_entry *cont = va_arg(*args, struct dir_entry *);
      int *index = va_arg(*args, int *);
      boolean (*get)(File f, struct dir_entry *, int *);
      get  = load_library_symbol(handle, "get");
      if(get != NULL){ return (*get)(f, cont, index); }
      else { printf("\nInternal error while executing get\n"); }

    }
    break;
  case MOVE:
    {
      File parent = va_arg(*args, File);
      boolean (*move)(File f, File parent);

      move = load_library_symbol(handle, "move");
      if(move != NULL){ return (*move)(f, parent); }
      else { printf("\nInternal error while executing move\n"); }

    }
    break;
  default:
    printf("Unknown method\n");
    retval = FALSE;
    break;
```

```
    }
    close_library(handle);

    return retval;

  }

  Public boolean dir_create(void)
  {
    int i = 0;

    Template t = new_template();
    t->create(DIR_TEMPLATE, dir_exec);

    /* Add operations & attibbutes */
    while(ops_attrs[i].name != NULL){
      _add(t, ops_attrs[i].name, ops_attrs[i].value);
      i++;
    }
    _add_direntry(t);
    return TRUE;
  }

  /*-------------------------------------------------------------------------
   * Private functions.
   */

  Private void _add(Template t, char *operation, char *data)
  {
    int size = strlen(data);
    String attr = new_string();
    Data d = new_data();

    /* Add the operation or attribute */
    attr->set(attr, operation);
    d->set(d, data, size);
    t->add_attribute(DIR_TEMPLATE, attr, d);
    attr->destroy(attr);
    d->destroy(d);

    return ;
  }

  Private void _add_direntry(Template t)
  {
    Data d = new_data();
    struct dir_entry *dir = (struct dir_entry *)calloc(NROFF, \
                                                       sizeof(struct dir_entry));

    dir = entries;
    d->set_void(d, dir, NROFF * sizeof(struct dir_entry));
    t->add_data(DIR_TEMPLATE, d);
    d->destroy(d);

    return ;
  }
```

Appendix E

Methods

The method modules are used to define the implementation of one or more file or file system operations. The method modules are supposed to be shared amongst the templates. Each compiled method module is saved as a dynamic library on the underlying operating system.

- `a_open_method.c`
  Defines the implementation for the two file system operations defined in PiVO, that is the #a_open and #a_close operations. When these operations are executed they call the zip and unzip operations on a file. The implementation of the zip and unzip operations are defined in `zip_methods.c`

- `cat_method.c`
  Defines the implementation for the cat operation defined in the templates `plain` and `cprog`.

- `zip_methods.c`
  Defines the implementation for the compression and decompression operations.

- `dir_methods.c`
  Defines the implementation of a simple directory structure. A directory stores the name and the file descriptors for the files in the directory in an array. The array is then saved in the data part of a directory, and stored on disk.

E.1 a_open_method.c

```
/*-------------------------------------------------------------------------
 * Defines the file system operations _a_open and _a_close.
 * The methods are executed each time a file is opened and closed.
 */

#include "pivo.h"

/*-------------------------------------------------------------------------
 * Public functions.
 */

/*
 * Called when a file is opened to uncompress its data.
 */
Public void _a_open(File f)
{
  String attr = new_string();

  attr->set(attr, "unzip");
  f->exec_m(f, attr);
  attr->destroy(attr);

  return ;
}

/*
 * Called when a file is closed to compress its data.
 */
Public void _a_close(File f)
{
  String attr = new_string();

  attr->set(attr, "zip");
  f->exec_m(f, attr);
  attr->destroy(attr);

  return ;
}
```

E.2 cat_method.c

```
/*-------------------------------------------------------------------------
 * Defines the file file operation cat. It writes the data of a file to screen.
 */

#include <stdio.h>
#include "pivo.h"

/*-------------------------------------------------------------------------
 * Public functions
 */

Public void cat(File f)
{
  Data fdata = f->read(f);
  printf("%s", fdata->get(fdata));

  return ;
}
```

E.3 dir_methods.c

```
/*-------------------------------------------------------------------------
 * Defines the operations on directories.
 */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "pivo.h"

#define NROFF 200
#define NAME_LEN 200

struct dir_entry {
  int fd;
  char name[NAME_LEN];
};

typedef struct dir_entry *Direntry;

/*-------------------------------------------------------------------------
 * Public function declarations.
 */

/* Add a file to the directory */
Public boolean add(File dir, File newfile);
/* Delete a file from the directory */
Public boolean delete(File dir, char *name);
/* Get the name of the file at index 'index' in the directory */
Public boolean get(File dir, struct dir_entry *entries, int *index);
/* Move the file fd_file to parent_fd */
Public boolean move(File dir, File parent);

/*-------------------------------------------------------------------------
 * Private function declrations.
 */

Private boolean _add(File dir, int fd_dir, int fd_file, char *name);
Private Direntry new_direntry(int nr);

/*-------------------------------------------------------------------------
 * Public functions.
 */

/* Add a file to the directory */
Public boolean add(File dir, File newfile)
{
  boolean retval;
  String snewname = newfile->get_name(newfile);
  char *newname = snewname->get(snewname);

  retval = _add(dir, dir->fd, newfile->fd, newname);
  snewname->destroy(snewname);

  return retval;
}

Public boolean delete(File dir, char *name)
```

```
{
  String attr = new_string();

  attr->set(attr, "fe");
  attr->destroy(attr);

  return TRUE;
}

Public boolean get(File f, struct dir_entry *entries, int *index)
{
  String attr = new_string();
  Data data;
  Direntry dirp;
  int i = 0;

  attr->set(attr, "Nroff");
  data = f->get_attr(f, attr);
  *index = atoi(data->get(data));

  if(*index== 0){ entries = NULL; return TRUE; }

  data = f->read(f);
  dirp = (Direntry)data->get_void(data);

  while(i < *index){
    entries->fd = dirp->fd;
    strcpy(entries->name, dirp->name);
    dirp++;
    entries++;
    i++;
  }
  attr->destroy(attr);
  data->destroy(data);

  return TRUE;
}

/* This method adds the . and .. entries to dir. TODO: if . and
   .. already exists it should remove these before adding the new
   entries. */
Public boolean move(File dir, File parent)
{
  _add(dir, dir->fd, dir->fd, ".");
  _add(dir, dir->fd, parent->fd, "..");

  return TRUE;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private Direntry new_direntry(int nr)
{
  return (Direntry)calloc(nr, sizeof(struct dir_entry));
}

Private boolean _add(File dir, int fd_dir, int fd_file, char *name)
{
  Data d  = new_data();
  Data d2 = new_data();
```

```
String nroff = new_string();
int nr_off = 0;
struct dir_entry entries[NROFF];
Direntry dirp;

if(strlen(name) > NAME_LEN || strlen(name) == 0){
  printf("Illegal name:%s\n", name);
  nroff->destroy(nroff);
  d->destroy(d);
  d2->destroy(d2);

  return FALSE;
}

nroff->set(nroff, "Nroff");
d = dir->get_attr(dir, nroff);
if(d == NULL){
  printf("Nroff is missing!\n");
  nroff->destroy(nroff);
  d->destroy(d);
  d2->destroy(d2);

  return FALSE;
}
nr_off = atoi((char *)d->get(d));

if(nr_off == NROFF){
  printf("Directory is full\n");
  nroff->destroy(nroff);
  d->destroy(d);
  d2->destroy(d2);

  return FALSE;
}

if(nr_off == 0){
  nr_off++;
  entries[0].fd = fd_file;
  strcpy(entries[0].name, name);
  d->set_void(d, entries, NROFF * sizeof(struct dir_entry));
  dir->write(dir, d);

  d->set(d, itoa(nr_off), strlen(itoa(nr_off)));
  dir->set_attr(dir, nroff, d);
  nroff->destroy(nroff);
  d->destroy(d);
  d2->destroy(d2);

  return TRUE;

}else {
  /* This code might be better with pointers etc. but it works */
  int i = 0;
  struct dir_entry c[NROFF];

  d = dir->read(dir);
  dirp = new_direntry(NROFF);
  dirp = (Direntry)d->get(d);
  while(i < nr_off){
    c[i].fd = dirp->fd;
    strcpy(c[i].name, dirp->name);
    dirp++;
```

```
        i++;
      }
      c[nr_off].fd = fd_file;
      strcpy(c[nr_off].name, name);

      d->set_void(d, c, NROFF * sizeof(struct dir_entry));
      dir->write(dir, d);

      nr_off++;
      d2->set(d2, itoa(nr_off), strlen(itoa(nr_off)));
      dir->set_attr(dir, nroff, d2);

      nroff->destroy(nroff);
      d->destroy(d);
      d2->destroy(d2);

      return TRUE;
    }
    nroff->destroy(nroff);
    d->destroy(d);
    d2->destroy(d2);

    return FALSE; /* Should never happen! */
  }
```

E.4 zip_methods.c

```
/*-------------------------------------------------------------------------
 * Defines the file operations zip and unzip, which compresses and decompresses
 * the data of a file.
 */

#include <stdlib.h>
#include <zlib.h>
#include <string.h>
#include <stdio.h>
#include "pivo.h"

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private int get_usize(File f);
Private void set_usize(File f, int usize);
Private void set_csize(File f);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public boolean zip(File f)
{
  Data fdata;
  Bytef *source, *dest;
  uLong slen, dlen;

  if(f->get_size(f)==0){
    /* No data to zip. */
    return TRUE;
  }

  if(get_usize(f) != 0){
    /* data already zipped. */
    return TRUE;
  }

  fdata = f->read(f);
  source = (Bytef*)fdata->get(fdata);
  slen = f->get_size(f);
  dlen  = slen * 1.1 + 12;

  dest = (Bytef*)calloc(dlen, sizeof(Bytef*));
  compress(dest, &dlen, source, slen);

  fdata->set(fdata, dest, dlen);
  f->write(f, fdata);
  set_usize(f, slen);
  set_csize(f);
  fdata->destroy(fdata);

  return TRUE;
}

Public boolean unzip(File f)
```

```
{
  Data fdata;
  Bytef *source, *dest;
  uLong slen, dlen;

  if(f->get_size(f)==0){
    /* No data to zip. */
    return TRUE;
  }

  if((dlen = get_usize(f)) == 0){
    /* data already unzipped. */
    return TRUE;
  }

  fdata = f->read(f);
  source = (Bytef*)fdata->get(fdata);
  slen = f->get_size(f);

  dest = (Bytef*)calloc(dlen, sizeof(Bytef*));
  uncompress(dest, &dlen, source, slen);

  fdata->set(fdata, dest, dlen);
  f->write(f, fdata);
  set_usize(f, 0);
  fdata->destroy(fdata);

  return TRUE;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private int get_usize(File f)
{
  int retval = 0;
  Data usize;
  String attr = new_string();
  attr->set(attr, "Usize");
  usize = f->get_attr(f, attr);
  if (usize != NULL){
    retval = atoi(usize->get(usize));
  }
  attr->destroy(attr);

  return retval;
}

Private void set_usize(File f, int usize)
{
  String attr = new_string();
  attr->set(attr, "Usize");
  Data value = new_data();

  value->set_int(value, usize);
  f->set_attr(f, attr, value);
  attr->destroy(attr);
  value->destroy(value);

  return ;
}
```

```
Private void set_csize(File f)
{
  char *size_str;
  String attr = new_string();
  Data data = new_data();

  attr->set(attr, "Csize");
  size_str = itoa(f->get_size(f));
  data->set(data, size_str, strlen(size_str));
  f->set_attr(f, attr, data);

  attr->destroy(attr);
  data->destroy(data);

  return ;
}
```

Appendix F

PiVO Core Library

The PiVO Core Library has a diversity of duties; It defines the programmer interface to files, manages an internal collection of registered templates, manages on disk storage for files and implements an internal name space.

The programmers interface is defined in `file.h`, and the implementation is defined in `file.c`. Through the interface a programmer can access file attributes and call operations.

The internal collection of templates and the registration of templates is managed in `template.c`.

The on disk storage of files is in reality managed by the embedded database library Berkeley DB 4.1.25. The implementation in `obj.c` communicates with the database, and uses the database to store files on disk, and access attributes of a file. The reason we decided to use Berkeley DB is because all elements in a Berkeley DB database is represented with a key/data pair, and this model fits with the concept of attribute name/attribute pair value in a PiVO file. Also, Berkeley DB defines access operations for both the keys and values in a database. The ability to get a listing of the keys in a database is used by the file operation `get_methods` and lets a user or programmer list all the available operations defined on a file.

The internal name space for files in PiVO is implemented in `filedescriptors.c` and assigns a unique integer value to each file when the file is created.

F.1 pivo.h

```
/*-------------------------------------------------------------------------
 * Defines the pivo library interface.
 */

#ifndef __PIVO__H_
#define __PIVO__H_

#include "globals.h"
#include "integer.h"
#include "data.h"
#include "itoa.h"
#include "registertemplate.h"
#include "diskobject.h"
#include "library.h"
#include "String.h"
#include "element.h"
#include "list.h"
#include "strndup.h"
#include "file.h"
#include "meta.h"
#include "template.h"
#include "filedescriptors.h"
#include "metainternals.h"
#include "templateitem.h"
#include "obj.h"

#endif
```

F.2 registertemplate.h

```
/*-------------------------------------------------------------------------
 * Defines the register template interface.
 */

#ifndef _REGISTERTEMPLATE_H
#define _REGISTERTEMPLATE_H

#include "globals.h"
#include "list.h"

boolean load_template(char *path, char *name);
Public List get_templates(void);

#endif
```

F.3 registertemplate.c

```
/*-------------------------------------------------------------------------
 * Defines the register template interface.
 */

#include <string.h>
#include "library.h"
#include "String.h"
#include "list.h"

#define MAX_TEMPLATES 200

Private char *templates[MAX_TEMPLATES];
Private int index = 0;

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private boolean (*template)(void);
Private boolean _add_registertemplate(char *name);

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private boolean _add_registertemplate(char *name)
{
  if(index >= MAX_TEMPLATES) { return FALSE; }
  templates[index] = strdup(name);
  index++;

  return TRUE;
}

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public List get_templates(void)
{
  List list = new_list();
  int i = 0;
  while(i < index){
    list->add(list, templates[i]);
    i++;
  }

  return list;
}

Public boolean load_template(char *path, char *name)
{
  void *handle;
  boolean retval;
  String _name = new_string();
  _name = _name->set(_name, name);
  _name = _name->append_char(_name, "_create");
```

```
    handle = open_library(path);
    if(handle == NULL){
      return FALSE;
    }
    template = load_library_symbol(handle, _name->get(_name));
    retval =  (*template)();
    if(retval == TRUE){
      _add_registertemplate(name);
    }
    _name->destroy(_name);

    return retval;
  }
```

F.4 template.h

```
/*-------------------------------------------------------------------------
 * Implements the template interface.
 */

#ifndef __TEMPLATE_H
#define __TEMPLATE_H

#include <stdarg.h>
#include "globals.h"
#include "String.h"
#include "data.h"
#include "file.h"


typedef struct template *Template;

struct template {

  /* Create a new template */
  boolean (*create)(char *name_of_template, \
                    boolean (*exec)(File f, int m_nr, Data data, va_list *args));
  /* Add attributes to a template */
  boolean (*add_attribute)(char *name_of_template, String attr, \
                           Data data);
  /* Add the data attribute to a template */
  boolean (*add_data)(char *name_of_template, Data data);

  /* Create a new file from a template */
  File (*create_file)(char *name_of_template, char *name_of_file);
  /* Execute a file method. The method is dependent on the template
     of the file. */
  boolean (*exec)(char *name_of_template, File f, int m_nr, \
                  Data data, va_list *args);
  /* Copy a template to another template */
  boolean (*copy)(char *name_src, char *name_dest, int fd);

  void (*destroy)(Template this);
};

Public Template new_template(void);

#endif
```

F.5 template.c

```
/*-------------------------------------------------------------------------
 * Implements the template interface.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include "template.h"
#include "templateitem.h"
#include "file.h"
#include "list.h"
#include "strndup.h"

Private Template t_template = NULL;
Private List templateList = NULL;

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private File create_file_template(char *name_of_template, char *name_of_file);
Private boolean create_template(char *name_of_template, \
                boolean (*exec)(File f, int m_nr, Data data, va_list *args) );
Private boolean add_attribute_template(char *name_of_template, \
                                       String attr, Data data);
Private boolean add_data_template(char *name_of_template, Data data);
Private boolean copy_template(char *name_src, char *name_dest, int fd);
Private boolean exec_template(char *name_of_template, File f, int m_nr, \
                              Data data, va_list *args);
Private void destroy_template(Template this);
Private boolean _cmp_by_name(void *d1, void *d2);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public Template new_template(void)
{
  /* Simulated Singleton (thanks Tomas) - Because we only need one
     instance of Template */
  if(t_template == NULL) {
    t_template = (Template)calloc(1, sizeof(struct template));
    t_template->create = create_template;
    t_template->add_attribute = add_attribute_template;
    t_template->add_data = add_data_template;
    t_template->copy = copy_template;
    t_template->create_file = create_file_template;
    t_template->exec = exec_template;
    t_template->destroy = destroy_template;
    templateList = new_list();
  }
  return t_template;
}

/*-------------------------------------------------------------------------
 * Private functions.
```

```
 */

Private boolean _cmp_by_name(void *d1, void *d2)
{
  TemplateItem t1 = (TemplateItem)d1;
  TemplateItem t2 = (TemplateItem)d2;

  return t1->cmp_name(t1, t2);
}

Private boolean create_template(char *name_of_template, \
                  boolean (*exec)(File f, int m_nr, Data data, va_list *args) )
{
  int fd;
  TemplateItem ti = new_templateitem();
  String name = new_string();
  name->set(name, name_of_template);

  /* Create a new template i.e. create a new file and save the
     file descriptor in the list templateList */

  File f = new_file();
  fd = f->create();
  f->open(f, fd);
  f->set_name(f, name);

  ti->set_name(ti, name_of_template);
  ti->set_fd(ti, fd);
  ti->set_exec(ti, exec);
  templateList = templateList->add(templateList, ti);
  f->close(f);
  name->destroy(name);
  f->destroy(f);

  return TRUE;
}

Private boolean add_attribute_template(char *name_of_template, \
                                       String attr, Data data)
{
  Element el;
  TemplateItem ti = new_templateitem();
  File f = new_file();

  ti->set_name(ti, name_of_template);
  el = templateList->get(templateList, ti, _cmp_by_name);
  if(el == NULL){ return FALSE; }
  ti = (TemplateItem)el->get_value(el);
  f->open(f, ti->get_fd(ti));
  f->set_attr(f, attr, data);
  f->close(f);
  f->destroy(f);

  return TRUE;
}

Private boolean add_data_template(char *name_of_template, Data data)
{

  Element el;
  TemplateItem ti = new_templateitem();
  File f = new_file();
```

```
    ti->set_name(ti, name_of_template);
    el = templateList->get(templateList, ti, _cmp_by_name);
    if(el == NULL){ return FALSE; }
    ti = (TemplateItem)el->get_value(el);
    f->open(f, ti->get_fd(ti));
    f->write(f, data);
    f->close(f);
    f->destroy(f);

    return TRUE;
}

Private boolean copy_template(char *name_src, char *name_dest, int fd)
{
    return TRUE; /* To be done if we get the time. */
}

Private File create_file_template(char *name_of_template, char *name_of_file)
{
    Element el;
    char *attr;
    String the_attr = new_string();
    Data src_data = new_data();
    Data dst_data = new_data();
    String templ = new_string();

    templ->set(templ, name_of_template);
    TemplateItem ti_el = new_templateitem();
    TemplateItem ti;
    File tem = new_file(); /* Template file */
    File nf = new_file();  /* New file      */
    int nf_fd;             /* New file file descriptor */
    char *tmp;

    /* Lookup the template in the list 'templateList' */
    ti_el->set_name(ti_el, name_of_template);
    el = templateList->get(templateList, ti_el, _cmp_by_name);
    ti_el->destroy(ti_el);

    if(el == NULL){ printf("Unknown template\n"); return FALSE; }
    ti = (TemplateItem)el->get_value(el);

    /* Open the template for reading */
    tem->open(tem, ti->get_fd(ti));

    /* Create and open the new file */
    nf_fd = nf->create();
    nf->open(nf, nf_fd);
    /*
       Copy attributes from the template to the new file.
       Note: All attributes except those with a leading '_' character
       are copied.
    */
    tem->init_list(tem);
    attr = tem->get_list(tem);
    if(attr[0] != '_'){
      the_attr->set(the_attr, attr);
      src_data = tem->get_attr(tem, the_attr);
      tmp = src_data->get(src_data);
      dst_data->set(dst_data, tmp, src_data->size(src_data) );
      free(tmp);
```

```
      nf->set_attr(nf, the_attr, dst_data);
    }
    while( (attr = tem->get_list(tem) ) != NULL){
      if(attr[0] != '_'){
        the_attr->set(the_attr, attr);
        src_data = tem->get_attr(tem, the_attr);
        tmp = src_data->get(src_data);
        dst_data->set(dst_data, tmp, src_data->size(src_data) );
        free(tmp);
        nf->set_attr(nf, the_attr, dst_data);
      }
    }

    /* Set the name of the new file */
    the_attr->set(the_attr, name_of_file);
    nf->set_name(nf, the_attr);

    /* Set the type of the new file to the template name */
    nf->set_template_name(nf, templ);

    /* Close the template and the new file */
    tem->close(tem);
    nf->close(nf);

    /* Clean up */
    src_data->destroy(src_data);
    dst_data->destroy(dst_data);
    the_attr->destroy(the_attr);

    templ->destroy(templ);
    tem->destroy(tem);

    return nf;
  }

  Private boolean exec_template(char * name_of_template, File f, int m_nr, \
                                Data data, va_list *args)
  {
    Element el;
    TemplateItem ti_el = new_templateitem();
    TemplateItem ti;

    ti_el->set_name(ti_el, name_of_template);
    el = templateList->get(templateList, ti_el, _cmp_by_name);
    ti_el->destroy(ti_el);
    if(el == NULL){ return FALSE; }
    ti = (TemplateItem)el->get_value(el);

    return ti->exec(f, m_nr, data, args);
  }

  Private void destroy_template(Template this)
  {
    if(templateList){ templateList->delete_all(templateList); }
    if(this) { free(this); }

    return ;
  }
```

F.6 templateitem.h

```
/*-----------------------------------------------------------------------------
 * Defines the template item interface.
 */

#ifndef _TEMPLATEITEM_H
#define _TEMPLATEITEM_H

#include <stdarg.h>
#include "globals.h"
#include "data.h"
#include "file.h"

typedef struct templateitem *TemplateItem;

struct templateitem {

  /* Private */
  int fd;
  char *name;
  boolean (*exec)(File f, int m_nr, Data data, va_list *args);

  /* Public */
  boolean (*set_exec)(TemplateItem this, \
            boolean (*exec)(File f, int m_nr, Data data, va_list *args));
  boolean (*execute)(TemplateItem this, File f, int m_nr, Data data, \
                     va_list *args);
  boolean (*set_name)(TemplateItem this, char *name);
  char *(*get_name)(TemplateItem this);
  boolean (*set_fd)(TemplateItem this, int fd);
  int (*get_fd)(TemplateItem this);
  boolean (*cmp_name)(TemplateItem this, TemplateItem other);
  boolean (*cmp_fd)(TemplateItem this, TemplateItem other);
  void (*destroy)(TemplateItem this);

};

Public TemplateItem new_templateitem(void);

#endif
```

F.7 tempalteitem.c

```
/*-------------------------------------------------------------------------
 * Implements the template item interface.
 */

#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include "templateitem.h"
#include "strndup.h"

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private void destroy_templateitem(TemplateItem this);
Private boolean set_name_templateitem(TemplateItem this, char *name);
Private char *get_name_templateitem(TemplateItem this);
Private boolean set_fd_templateitem(TemplateItem this, int fd);
Private int get_fd_templateitem(TemplateItem this);
Private boolean set_exec_templateitem(TemplateItem this, \
                boolean (*exec)(File f, int m_nr, Data data, va_list *args));
Private boolean execute_templateitem(TemplateItem this, File f, int m_nr, \
                                     Data data,  va_list *args);
Private boolean cmp_name_templateitem(TemplateItem this, TemplateItem other);
Private boolean cmp_fd_templateitem(TemplateItem this, TemplateItem other);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public TemplateItem new_templateitem(void)
{
  TemplateItem ti = (TemplateItem)calloc(1, sizeof(struct templateitem));
  ti->set_name = set_name_templateitem;
  ti->get_name = get_name_templateitem;
  ti->set_fd = set_fd_templateitem;
  ti->get_fd = get_fd_templateitem;
  ti->set_exec = set_exec_templateitem;
  ti->execute = execute_templateitem;
  ti->cmp_name = cmp_name_templateitem;
  ti->cmp_fd = cmp_fd_templateitem;
  ti->destroy = destroy_templateitem;

  return ti;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private void destroy_templateitem(TemplateItem this)
{
  /*if(this->name != NULL){ free(this->name); }*/
  if(this) { free(this); }

  return ;
```

```
}

Private boolean set_name_templateitem(TemplateItem this, char *name)
{
  this->name = (char *)strndup(name, strlen(name));

  return this->name == NULL ? FALSE : TRUE;
}

Private char *get_name_templateitem(TemplateItem this)
{
  return (char *)strndup(this->name, strlen(this->name));
}

Private boolean set_fd_templateitem(TemplateItem this, int fd)
{
  this->fd = fd;

  return TRUE;
}

Private int get_fd_templateitem(TemplateItem this){ return this->fd; }

Private boolean set_exec_templateitem(TemplateItem this, \
              boolean (*exec)(File f, int m_nr, Data data, va_list *args))
{
  this->exec = exec;

  return TRUE;
}

Private boolean execute_templateitem(TemplateItem this, File f, int m_nr, \
                                     Data data,  va_list *args)
{
  return this->exec(f, m_nr, data, args);
}

Private boolean cmp_name_templateitem(TemplateItem this, TemplateItem other)
{
  return strcmp(this->name, other->name) == 0 ? TRUE : FALSE;
}

Private boolean cmp_fd_templateitem(TemplateItem this, TemplateItem other)
{
  return this->fd == other->fd ? TRUE : FALSE;
}
```

F.8 file.h

```
/*-------------------------------------------------------------------------
 * Defines the file interface.
 */

#ifndef __FILE__H
#define __FILE__H

#include <stdarg.h>
#include "globals.h"
#include "data.h"
#include "String.h"
#include "list.h"

typedef struct file *File;

struct file {

  boolean first_attr_by_name;
  List attr_by_name;
  int fd;

  int (*create)(void);            /* Create a new file */
  int (*open)(File this, int fd); /* Open a file       */

  /* Generic set/get value of attr. */
  boolean (*set_attr)(File this, String attr, Data data);
  Data (*get_attr)(File this, String attr);

  Data (*read)(File this);                  /* Read the data in the file. */
  boolean (*write)(File this, Data data);   /* Write data to the file.    */
  String (*get_name)(File this);            /* Get the name of the file.  */
  void (*set_name)(File this, String name); /* Set the name of the file.  */
  int (*get_size)(File this);               /* Size of the file.          */

  /* Get/Set template name of the file */
  String (*get_template_name)(File this);
  void (*set_template_name)(File this, String owner);

  /* List all attributes & operations by name.*/
  boolean (*init_list)(File this);    /* Initialize list for get attr & ops */
  char* (*get_list)(File this);       /* Get all attributes and operations */
  char* (*get_methods)(File this);    /* List methods    */
  char* (*get_attributes)(File this);  /* List attributes */
  boolean (*close_list)(File this);   /* Delete list for get attr & ops */

  boolean (*exec_m)(File this, String method, ...); /* Execute operation */
  void (*close)(File this);                         /* Close file        */
  boolean (*delete)(File this);                     /* Delete file       */
  void (*destroy)(File this);                       /* File destructor   */
};

Public File new_file(void); /* File constructor */

#endif
```

F.9 file.c

```
/*-------------------------------------------------------------------------
 * Implements the file interface.
 */

#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>
#include <stdarg.h>
#include "meta.h"
#include "file.h"
#include "strndup.h"
#include "itoa.h"
#include "template.h"

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private int create_file(void);
Private int open_file(File this, int fd);
Private void close_file(File this);
Private Data get_attr_file(File this, String attr);
Private boolean set_attr_file(File this, String attr, Data data);
Private boolean write_file(File this, Data data);
Private Data read_file(File this);
Private String get_name_file(File this);
Private void set_name_file(File this, String name);
Private String get_template_name_file(File this);
Private void set_template_name_file(File this, String template);
Private int get_size_file(File this);
Private boolean delete_file(File this);
Private boolean init_list_file(File this);
Private char *get_list_file(File this);
Private char *get_methods_file(File this);
Private char *get_attributes_file(File this);
Private boolean close_list_file(File this);
Private boolean exec_m_file(File this, String method, ...);
Private void destroy_file(File this);
Private boolean _exec_a_open(File this);
Private boolean _exec_a_close(File this);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public File new_file(void)
{
  File f = (File)calloc(1, sizeof(struct file));
  f->first_attr_by_name = TRUE;
  f->attr_by_name = new_list();

  f->create = create_file;
  f->open = open_file;
  f->read = read_file;
  f->write = write_file;
  f->set_attr = set_attr_file;
```

```
  f->get_attr = get_attr_file;
  f->set_name = set_name_file;
  f->get_name = get_name_file;
  f->set_template_name = set_template_name_file;
  f->get_template_name = get_template_name_file;
  f->get_size = get_size_file;
  f->close = close_file;
  f->delete = delete_file;
  f->init_list = init_list_file;
  f->get_list = get_list_file;
  f->get_methods = get_methods_file;
  f->get_attributes = get_attributes_file;
  f->close_list  = close_list_file;
  f->exec_m = exec_m_file;
  f->destroy = destroy_file;

  return f;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private int create_file(void)
{
  Data data = new_data();
  String attr = new_string();
  Meta meta = new_meta();
  int fd = meta->create();
  char *date = itoa(time(NULL));

  data->set(data, strndup(date, strlen(date)), strlen(date));
  attr->set(attr, "_creation time");
  meta->write_attr(fd, attr, data);
  data->set_int(data, 0);
  attr->set(attr, "_size");
  meta->write_attr(fd, attr, data);
  attr->destroy(attr);
  data->destroy(data);
  meta->destroy(meta);

  return fd;
}

Private int open_file(File this, int fd)
{
  int retval;
  this->fd = fd;
  Meta meta = new_meta();

  retval = meta->open(fd);
  _exec_a_open(this);
  meta->destroy(meta);

  return retval;
}

Private boolean write_file(File this, Data data)
{
  Meta meta = new_meta();
  boolean retval = meta->write_data(this->fd, data);
  String attr = new_string();
```

```
    Data size = new_data();

    if(retval){
      size->set_int(size, data->size(data));
      attr->set(attr, "_size");
      meta->write_attr(this->fd, attr, size);
    }
    attr->destroy(attr);
    size->destroy(size);
    meta->destroy(meta);

    return retval;
  }

  Private Data read_file(File this)
  {
    Meta meta = new_meta();
    Data d;
    d = meta->read_data(this->fd);
    meta->destroy(meta);

    return d;
  }

  Private Data get_attr_file(File this, String attr)
  {
    Meta meta = new_meta();
    Data d;

    d = meta->read_attr(this->fd, attr);
    meta->destroy(meta);
    return d;
  }

  Private boolean set_attr_file(File this, String attr, Data data)
  {
    boolean retval;
    Meta meta = new_meta();
    /* No test to see if the user is trying to write a private attribute. */
    retval = meta->write_attr(this->fd, attr, data);
    meta->destroy(meta);

    return retval;
  }

  Private String get_name_file(File this)
  {
    Meta meta = new_meta();
    String attr = new_string();
    String retval = new_string();
    Data d = new_data();

    attr->set(attr, "_name");
    d = meta->read_attr(this->fd, attr);
    retval->set(retval, d->get(d));
    attr->destroy(attr);
    d->destroy(d);
    meta->destroy(meta);

    return retval;
  }
```

```
Private void set_name_file(File this, String name)
{
  Meta meta = new_meta();
  String attr = new_string();
  Data data = new_data();

  data->set(data, name->copy_str(name), name->size(name));
  attr->set(attr, "_name");
  meta->write_attr(this->fd, attr, data);
  attr->destroy(attr);
  data->destroy(data);
  meta->destroy(meta);

  return ;
}

Private String get_template_name_file(File this)
{
  Meta meta = new_meta();
  String attr = new_string();
  String retval = new_string();
  Data d;

  attr->set(attr, "_template");
  d = meta->read_attr(this->fd, attr);
  retval->set(retval, d->get(d));
  attr->destroy(attr);
  d->destroy(d);
  meta->destroy(meta);

  return retval;
}

Private int get_size_file(File this)
{
  int retval;
  Meta meta = new_meta();
  String attr = new_string();
  Data d = new_data();

  attr->set(attr, "_size");
  d = meta->read_attr(this->fd, attr);
  retval = atoi(d->get(d));

  attr->destroy(attr);
  d->destroy(d);
  meta->destroy(meta);

  return retval;
}

Private void set_template_name_file(File this, String template)
{
  Meta meta = new_meta();
  String attr = new_string();
  Data data = new_data();

  data->set(data, template->copy_str(template), template->size(template));
  attr->set(attr, "_template");
  meta->write_attr(this->fd, attr, data);

  attr->destroy(attr);
```

```
    data->destroy(data);
    meta->destroy(meta);

    return ;
  }

  Private void close_file(File this)
  {
    Meta meta = new_meta();
    _exec_a_close(this);
    meta->close(this->fd);
    meta->destroy(meta);

    return ;
  }

  Private boolean delete_file(File this)
  {
    boolean retval;
    Meta meta = new_meta();
    retval = meta->delete(this->fd);
    meta->destroy(meta);

    return retval;
  }

  Private boolean init_list_file(File this)
  {
    boolean retval;
    Meta meta = new_meta();
    retval = meta->get_attrs_by_name(this->fd, this->attr_by_name);
    meta->destroy(meta);

    return retval;
  }

  Private char *get_list_file(File this)
  {
    Element el;
    if(this->first_attr_by_name) {
      el = this->attr_by_name->get_first(this->attr_by_name);
      this->first_attr_by_name = FALSE;
    }
    else {
      el = this->attr_by_name->get_next(this->attr_by_name);
    }

    return el == NULL ? NULL : (char *)strdup((char *)el->get_value(el));
  }

  Private char *get_methods_file(File this)
  {
    Element el;
    char *attr;
    int len;
    boolean next = FALSE;

    do{
      if(this->first_attr_by_name) {
        el = this->attr_by_name->get_first(this->attr_by_name);
        this->first_attr_by_name = FALSE;
      }
```

```
      else {
        el = this->attr_by_name->get_next(this->attr_by_name);
      }
      if(el == NULL) { return NULL; }
      attr = (char *)strdup((char *)el->get_value(el));
      len = strlen(attr);
      if( (attr[len - 2] ==  '_') && (attr[len - 1] == 'm') ){
        next = TRUE;
      }
      else if(attr[0] == '#' || attr[0] == '_'){ next = TRUE; }
      else if(isupper(attr[0])){ next = TRUE; }
      else { next = FALSE; }

    }while(next);

    return attr;
  }

  Private char *get_attributes_file(File this)
  {
    Element el;
    char *attr;
    int len;
    boolean next = FALSE;
    Data data;
    String str_attr = new_string();

    do{
      if(this->first_attr_by_name) {
        el = this->attr_by_name->get_first(this->attr_by_name);
        this->first_attr_by_name = FALSE;
      }
      else {
        el = this->attr_by_name->get_next(this->attr_by_name);
      }
      if(el == NULL) { return NULL; }
      attr = (char *)strdup((char *)el->get_value(el));
      len = strlen(attr);
      if( (attr[len - 2] ==  '_') && (attr[len - 1] == 'm') ){
        next = TRUE;
      }
      /* We are not interested in the data in the file */
      else if(strcmp(attr, "_data") == 0){ next = TRUE; }
      /* We are not interested in the compression flag */
      else if(strcmp(attr, "Usize") == 0){ next = TRUE; }
      /* We are not interested in file system operations */
      else if(attr[0] == '#'){ next = TRUE; }
      /* We are not interested in file operations */
      else if(islower(attr[0])){ next = TRUE; }
      /* But we are interested in the attributes */
      else {
        next = FALSE;
      }

    }while(next);

    str_attr->set(str_attr, attr);
    data = this->get_attr(this, str_attr);
    str_attr->append_char(str_attr, "\t");
    /* Here we assume the value of the attribute is saved as a string */
    str_attr->append_char(str_attr, data->get(data));
    attr = str_attr->copy_str(str_attr);
```

```
    str_attr->destroy(str_attr);


    return attr;
  }

  Private boolean close_list_file(File this)
  {
    this->first_attr_by_name = TRUE;
    this->attr_by_name = this->attr_by_name->delete_all(this->attr_by_name);

    return TRUE;
  }

  Private String get_template_file(File this)
  {
    Meta meta = new_meta();
    String attr = new_string();
    String retval = new_string();
    Data d;
    char *tmp;

    attr->set(attr, "_template");
    d = meta->read_attr(this->fd, attr);
    tmp = d->get(d);
    retval->set(retval, tmp);
    free(tmp);
    attr->destroy(attr);
    d->destroy(d);
    meta->destroy(meta);

    return retval;
  }

  Private boolean _exec_a_open(File this)
  {
    boolean retval;
    String a_open = new_string();
    a_open->set(a_open, "#a_open");

    retval = exec_m_file(this, a_open, NULL);
    a_open->destroy(a_open);

    return retval;
  }

  Private boolean _exec_a_close(File this)
  {
    boolean retval;
    String a_close = new_string();
    a_close->set(a_close, "#a_close");

    retval = exec_m_file(this, a_close, NULL);
    a_close->destroy(a_close);

    return retval;
  }

  Private boolean exec_m_file(File this, String method, ...)
  {
    int m_nr; /* Method number */
    String stemplate = get_template_file(this);
```

```
    char *str = stemplate->get(stemplate);
    String method_m = new_string();
    Data data;
    Template t = new_template();
    va_list args;
    boolean retval = FALSE;
    char *tmp1;
    char *tmp2;

    if(*str == '\0'){ return retval; }

    /* Append the string "_m" to the parameter method, see plain.c for details. */
    method_m->set(method_m, method->copy_str(method));
    method_m = method_m->append_char(method_m, "_m");

    /* Get the metod number */
    data = get_attr_file(this, method);

    tmp1 = data->get(data);
    if(*tmp1 == '\0'){ return retval; }
    m_nr = atoi(tmp1);
    if(tmp1){ free(tmp1); }

    /* Get the name of the library to load */
    data = get_attr_file(this, method_m);
    tmp2 = data->get(data);
    if(*tmp2 == '\0') { return retval; }

    /* Execute the method & return the result */
    va_start(args, method);
    retval = t->exec(str, this, m_nr, data, &args);
    va_end(args);

    if(stemplate){ stemplate->destroy(stemplate); }
    if(method_m){ method_m->destroy(method_m); }
    if(data){ data->destroy(data); }
    if(tmp2) { free(tmp2); }

    return retval;
  }

  Private void destroy_file(File this)
  {
    if(this->attr_by_name != NULL){
      this->attr_by_name->delete_all(this->attr_by_name);
    }
    if(this){ free(this); }

    return ;
  }
```

F.10 meta.h

```
/*-------------------------------------------------------------------------
 * Defines the meta interface.
 */

#ifndef __META__H
#define __META__H

#include "globals.h"
#include "String.h"
#include "data.h"
#include "list.h"

typedef struct meta *Meta;

struct meta {

  int (*create)(void);
  int (*open)(int fd);
  Data (*read_attr)(int fd, String attr);
  Data (*read_data)(int fd);
  boolean (*write_attr)(int fd, String attr, Data data);
  boolean (*write_data)(int fd, Data data);
  void (*close)(int fd);
  boolean (*delete)(int fd);
  boolean (*get_attrs_by_name)(int fd, List attr_list);
  void (*destroy)(Meta this);

};

Public Meta new_meta(void);

#endif
```

F.11 meta.c

```
/*-------------------------------------------------------------------------
 * Implements the meta interface.
 */

#include <stdio.h>
#include <stdlib.h>
#include "metainternals.h"
#include "meta.h"
#include "diskobject.h"
#include "list.h"

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private int create_meta(void);
Private int open_meta(int fd);
Private Data read_attr_meta(int fd, String attr);
Private Data read_data_meta(int fd);
Private boolean write_attr_meta(int fd, String attr, Data data);
Private boolean write_data_meta(int fd, Data data);
Private void close_meta(int fd);
Private boolean delete_meta(int fd);
Private boolean get_attrs_by_name_meta(int fd, List attrs_list);
Private void destroy_meta(Meta this);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public Meta new_meta(void)
{
  Meta meta = (Meta)calloc(1, sizeof(struct meta));

  meta->create = create_meta;
  meta->open = open_meta;
  meta->read_attr = read_attr_meta;
  meta->read_data = read_data_meta;
  meta->write_attr = write_attr_meta;
  meta->write_data = write_data_meta;
  meta->close = close_meta;
  meta->delete = delete_meta;
  meta->get_attrs_by_name = get_attrs_by_name_meta;
  meta->destroy = destroy_meta;

  return meta;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private int create_meta(void)
{
  int retval;
  Diskobject disk = new_diskobject();
```

```
    retval =  disk->create();
    disk->destroy(disk);

    return retval;
  }

  Private int open_meta(int fd)
  {
    int retval;
    Diskobject disk = new_diskobject();

    retval = disk->open(fd);
    disk->destroy(disk);

    return retval;
  }

  Private Data read_attr_meta(int fd, String attr)
  {
    Diskobject dobj = new_diskobject();
    MetaInternals mi = new_metainternals();
    Data d;

    d = dobj->read(fd);
    mi->formatin(fd, attr, d);
    mi->destroy(mi);
    dobj->destroy(dobj);

    return d;
  }

  Private Data read_data_meta(int fd)
  {
    String attr = new_string();
    Data data = new_data();

    attr->set(attr, "_data");
    data = read_attr_meta(fd, attr);
    attr->destroy(attr);

    return data;
  }

  Private boolean write_attr_meta(int fd, String attr, Data data)
  {
    boolean retval;
    Diskobject dobj = new_diskobject();
    MetaInternals mi = new_metainternals();

    mi->formatout(fd, attr, data);
    mi->destroy(mi);

    retval = dobj->write(fd, data);
    dobj->destroy(dobj);

    return retval;
  }

  Private boolean write_data_meta(int fd, Data data)
  {
    boolean retval;
    String attr = new_string();
```

```
  attr->set(attr, "_data");
  retval = write_attr_meta(fd, attr, data);
  attr->destroy(attr);

  return retval;
}

Private void close_meta(int fd)
{
  Diskobject dobj = new_diskobject();
  dobj->close(fd);
  dobj->destroy(dobj);

  return ;
}

Private boolean delete_meta(int fd)
{
  boolean retval;
  Diskobject dobj = new_diskobject();

  retval = dobj->delete(fd);
  dobj->destroy(dobj);

  return retval;
}

Private boolean get_attrs_by_name_meta(int fd, List attrs_list)
{
  boolean retval;
  Diskobject dobj = new_diskobject();
  MetaInternals mi = new_metainternals();
  Data d;

  d = dobj->read(fd);
  retval = mi->get_attrs_by_name(fd, attrs_list, d);
  d->destroy(d);
  mi->destroy(mi);
  dobj->destroy(dobj);

  return retval;
}

Private void destroy_meta(Meta this)
{
  if(this){ free(this); }

  return ;
}
```

F.12 metainternals.h

```
/*-----------------------------------------------------------------------------
 * Defines the meta internal interface.
 */

#ifndef __META_INTERNALS__H
#define __META_INTERNALS__H

#include "globals.h"
#include "data.h"
#include "String.h"
#include "list.h"

typedef struct metainternals *MetaInternals;

struct metainternals {

  boolean (*formatout)(int fd, String attr, Data data);
  Data (*formatin)(int fd, String attr, Data data);
  boolean (*get_attrs_by_name)(int fd, List attrs_list, Data data);
  void (*destroy)(MetaInternals this);

};

Public MetaInternals new_metainternals(void);

#endif
```

F.13 metainternals.c

```
/*-------------------------------------------------------------------------
 * Implements the meta internals interface.
 */

#include <stdlib.h>
#include "metainternals.h"
#include "data.h"
#include "String.h"
#include "obj.h"
#include "list.h"

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private boolean formatout_metainternals(int fd, String attr, Data data);
Private Data formatin_metainternals(int fd, String attr, Data data);
Private boolean get_attrs_by_name_metainternals(int fd, List attrs_list, \
                                                Data data);
Private void destroy_metainternals(MetaInternals this);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public MetaInternals new_metainternals(void)
{
  MetaInternals mi = (MetaInternals)calloc(1, sizeof(struct metainternals));
  mi->formatout = formatout_metainternals;
  mi->formatin = formatin_metainternals;
  mi->get_attrs_by_name = get_attrs_by_name_metainternals;
  mi->destroy = destroy_metainternals;

  return mi;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private boolean formatout_metainternals(int fd, String attr, Data data)
{
  write_attr_obj(fd, attr, data);

  return TRUE;
}

Private Data formatin_metainternals(int fd, String attr, Data data)
{
  read_attr_obj(fd, attr, data);

  return data;
}

Private boolean get_attrs_by_name_metainternals(int fd, List attrs_list, \
                                                Data data)
{
```

```
  return get_attrs_by_name_obj(fd, attrs_list);
}

Private void destroy_metainternals(MetaInternals this)
{
  if(this){ free(this); }

  return ;
}
```

F.14 diskobject.h

```
/*-------------------------------------------------------------------------
 * Defines the diskobject interface.
 */

#ifndef __DISKOBJECT__H
#define __DISKOBJECT__H

#include "globals.h"
#include "data.h"

typedef struct diskobject *Diskobject;

struct diskobject {
  int (*create)(void);
  boolean (*open)(int fd);
  Data (*read)(int fd);
  boolean (*write)(int fd, Data data);
  void (*close)(int fd);
  boolean (*delete)(int fd);
  void (*destroy)(Diskobject this);
};

Public Diskobject new_diskobject(void);

#endif
```

F.15 diskobject.c

```
/*-----------------------------------------------------------------------*
 * Implements the diskobject interface.
 */

#include <stdlib.h>
#include "obj.h"
#include "diskobject.h"

/*-----------------------------------------------------------------------*
 * Private function declarations.
 */

Private int create_diskobject(void);
Private boolean open_diskobject(int fd);
Private boolean write_diskobject(int fd, Data data);
Private Data read_diskobject(int fd);
Private void close_diskobject(int fd);
Private boolean delete_diskobject(int fd);
Private void destroy_diskobject(Diskobject this);

/*-----------------------------------------------------------------------*
 * Public functions.
 */

Public Diskobject new_diskobject(void)
{
  Diskobject dobj = (Diskobject)calloc(1, sizeof(struct diskobject));
  dobj->create = create_diskobject;
  dobj->open = open_diskobject;
  dobj->read = read_diskobject;
  dobj->write = write_diskobject;
  dobj->close = close_diskobject;
  dobj->delete = delete_diskobject;
  dobj->destroy = destroy_diskobject;

  return dobj;
}

/*-----------------------------------------------------------------------*
 * Private functions.
 */

Private int create_diskobject(void) { return create_obj(); }

Private boolean open_diskobject(int fd)
{
  return open_obj(fd) >= 0 ? TRUE : FALSE ;
}

Private boolean write_diskobject(int fd, Data data) { return TRUE; }

Private Data read_diskobject(int fd) { return new_data(); }

Private void close_diskobject(int fd) { }

Private boolean delete_diskobject(int fd) { return delete_obj(fd); }
```

```
Private void destroy_diskobject(Diskobject this)
{
  if(this){ free(this); }

  return ;
}
```

F.16 obj.h

```
/*-----------------------------------------------------------------------
 * Defines the object interface.
 */

#ifndef __OBJ__H
#define __OBJ__H

#include "globals.h"
#include "data.h"
#include "String.h"
#include "list.h"

#include "pool.h"

Public int create_obj(void);
Public boolean delete_obj(int fd);
Public int open_obj(int fd);
Public void read_attr_obj(int fd, String attr, Data data);
Public void write_attr_obj(int fd, String attr, Data data);
Public void close_obj(int fd);
Public boolean get_attrs_by_name_obj(int fd, List attrs_list);

#endif
```

F.17 obj.c

```
/*-------------------------------------------------------------------------
 * Implements the object interface.
 */

#include <sys/types.h>
#include <limits.h>
#include <db.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "filedescriptors.h"
#include "obj.h"
#include "strndup.h"

Private Filedescriptors fds = NULL;

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private DB *_open(DB *obj, int fd);
Private boolean _close(DB *obj);

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private DB *_open(DB *obj, int fd)
{
  int retval;
  String path = new_string();

  path->set(path, POOL);
  path = path->append_int(path, fd);

  retval = db_create(&obj, NULL, 0);
  if(retval != 0){
    printf("Internal error while opening %s\n", path->get(path));
  }

  retval = obj->open(obj, NULL, path->get(path), NULL, DB_BTREE, DB_CREATE, 0);
  if(retval != 0) { printf("Unable to open %s\n", path->get(path)); }
  path->destroy(path);

  return obj;
}

Private boolean _close(DB *obj)
{
  int retval;
  if((retval = obj->close(obj, 0)) != 0 && retval == 0){ return FALSE; }

  return TRUE;
}

/*-------------------------------------------------------------------------
```

```
 * Public functions.
 */

Public int create_obj(void)
{
  if(fds == NULL){ fds = new_filedescriptors(); }
  return fds->get();
}

/* Dummy function, not used but included to make the interface consistent */
Public void close_obj(int fd) { }

Public int open_obj(int fd)
{
  DB *obj  = _open(obj, fd);
  if(_close(obj) == FALSE){ printf("Unable to close data base\n"); }

  return fd;
}

Public void write_attr_obj(int fd, String attr, Data data)
{
  DBT key, value;
  DB *obj = _open(obj, fd);

  memset(&key, 0, sizeof(key));
  memset(&value, 0, sizeof(value));

  key.data = attr->copy_str(attr);
  key.size = attr->size(attr);

  value.data = data->get(data);
  value.size = data->size(data);

  obj->put(obj, NULL, &key, &value, 0);
  _close(obj);

  if(key.data){ free(key.data); }
  if(value.data){ free(value.data); }

  return ;
}

Public void read_attr_obj(int fd, String attr, Data data)
{
  DBT key, value;
  DB *obj = _open(obj, fd);
  int retval;

  memset(&key, 0, sizeof(key));
  memset(&value, 0, sizeof(value));
  key.data = attr->copy_str(attr);
  key.size = attr->size(attr);

  retval = obj->get(obj, NULL, &key, &value, 0);
  if(retval != 0){
    data->set(data, NULL, 0);
    _close(obj);
    if(key.data){ free(key.data); }
    return ;
  }
```

```
    /* OBS the order is importan do not close the database before
       the data has been saved.  */
    data->set(data, value.data, value.size);
    _close(obj);
    if(key.data){ free(key.data); }

    return ;
  }

  Public boolean delete_obj(int fd)
  {
    boolean retval;
    String path = new_string();
    path->set(path, POOL);
    path = path->append_int(path, fd);

    /* The next line makes sence if or when we add a file descriptor population
       method in new_filedescritpors, which reads filenames from POOL and adds
       the names (the filedescritptors) in the file descriptor list. */
    if(fds == NULL){ fds = new_filedescriptors(); }

    fds->delete(fd);

    if(remove(path->get(path)) == 0){ retval = TRUE; }
    else { retval = FALSE; }
    path->destroy(path);

    return retval;
  }

  Public boolean get_attrs_by_name_obj(int fd, List attrs_list)
  {
    int retval;
    DBC *dbcp;
    DBT key, value;
    DB *obj = _open(obj, fd);

    /* Create a cursor */
    if ((retval = obj->cursor(obj, NULL, &dbcp, 0)) != 0) { return FALSE; }

    /* Initialize the key/data return pair. */
    memset(&key, 0, sizeof(key));
    memset(&value, 0, sizeof(value));

    while ((retval = dbcp->c_get(dbcp, &key, &value, DB_NEXT)) == 0){
      attrs_list = attrs_list->add(attrs_list, strndup((char *)key.data, \
                                                       (int)key.size));
    }
    if (retval != DB_NOTFOUND) { return FALSE; }

    /* Destroy the cursor */
    retval = dbcp->c_close(dbcp);
    if(retval != 0){ return FALSE; }

    /* Close the database */
    _close(obj);

    return TRUE;
  }
```

F.18 filedescriptors.h

```
/*-------------------------------------------------------------------------
 * Defines the file descriptor interface.
 */

#ifndef __FILEDESCRIPTORS__H
#define __FILEDESCRIPTORS__H

#include "globals.h"

typedef struct filedescriptors *Filedescriptors;

struct filedescriptors {

  int (*get)(void);
  void (*delete)(int fd);
  void (*print)(void);
  void (*destroy)(Filedescriptors this);
};

Public Filedescriptors new_filedescriptors(void);

#endif
```

F.19 filedescriptors.c

```
/*-------------------------------------------------------------------------
 * Implements the file descriptor interface.
 */

#include <stdio.h>
#include <stdlib.h>
#include "filedescriptors.h"
#include "list.h"
#include "element.h"
#include "integer.h"

Private List fd_list = NULL;
Private Filedescriptors fds = (Filedescriptors)NULL;
Private int next_file_descriptor;

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private int get_filedescriptors(void);
Private void delete_filedescriptors(int fd);
Private void print_filedescriptors(void);
Private void destroy_filedescriptors(Filedescriptors this);
Private boolean _cmp(void *e1, void *e2);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public Filedescriptors new_filedescriptors(void)
{
  /* Simulated Singleton (thanks Tomas) - Because we only need one
     instance of fd_list*/
  if(fds == NULL){
    fds = (Filedescriptors)calloc(1, sizeof(struct filedescriptors));
    fds->get = get_filedescriptors;
    fds->delete = delete_filedescriptors;
    fds->print = print_filedescriptors;
    fds->destroy  = destroy_filedescriptors;
    fd_list = new_list();
    next_file_descriptor = 0;
  }

  return fds;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private boolean _cmp(void *e1, void *e2)
{
  boolean retval = FALSE;
  int *i_e1;
  int *i_e2;

  i_e1  = (int *)calloc(1, sizeof(int));
```

```
    i_e2 = (int *)calloc(1, sizeof(int));

    i_e1 = (int *)e1;
    i_e2 = (int *)e2;

    if(*i_e1 == *i_e2) { retval = TRUE; }

    return retval;

  }

  Private void print_filedescriptors(void)
  {
    Element el;
    Integer tmp;

    el = fd_list->get_first(fd_list);
    tmp = el->get_value(el);
    printf("%d ", tmp->get(tmp));
    while((el = fd_list->get_next(fd_list)) != NULL){
      tmp = el->get_value(el);
      printf("%d ", tmp->get(tmp));
    }
    printf("\n");

    return ;

  }

  Private int get_filedescriptors(void)
  {
    int retval = next_file_descriptor;

    fd_list->add(fd_list, new_integer(next_file_descriptor));
    next_file_descriptor++;

    return retval;
  }

  Private void delete_filedescriptors(int fd)
  {
    Integer i = new_integer(fd);
    fd_list = fd_list->delete(fd_list, i, _cmp);

    return ;
  }

  Private void destroy_filedescriptors(Filedescriptors this)
  {
    if(fd_list != NULL){ fd_list->delete_all(fd_list); }
    if(this){ free(this); }

    return ;
  }
```

Appendix G

Support Code

This appendix lists support code used primarily by the Core Library. The support code is divided into the following sections:

- Appendix G.1 defines the global definitions in PiVO.
- Appendix G.2 to G.7 defines string management.
- Appendix G.8 to G.11 defines the data types `Data` and `Integer`. The data type `Data` is used to represent the data of a file, and the data type `Integer` is used internally by the module `file descriptors`.
- Appendix G.12 to G.13 defines the code for dynamic library management.
- Appendix G.14 to G.17 defines a generic list implementation.

G.1 globals.h

```
/*-------------------------------------------------------------------------
 * Defines globals.
 */

/*-------------------------------------------------------------------------
 * Declares globals
 */

#ifndef __GLOBALS__H
#define __GLOBALS__H

/*-------------------------------------------------------------------------
 * DEFINE GLOBALS
 */

#define Private static
#define Public

/*-------------------------------------------------------------------------
 * DEFINE TYPES
 */
#undef FALSE
#undef TRUE

enum boolean { FALSE, TRUE };
typedef enum boolean boolean;

typedef char * string;

#endif
```

G.2 String.h

```
/*------------------------------------------------------------------------
 * Defines the String interface.
 *
 * The name of this file has to be String.h and not string.h in order for
 * it to be included instead of <string.h>, which will happen otherwise.
 */

#ifndef __String__H
#define __String__H

#include "globals.h"

typedef struct String *String;

struct String {
  char *str;
  int len;
  /* Operations */
  void (*destroy)(String this);
  String (*set)(String this, char *str);
  char *(*get)(String this);
  int (*size)(String this);
  char *(*copy_str)(String this);
  String (*append)(String this, String src);
  String (*append_char)(String this, char *src);
  String (*append_int)(String this, int src);
  void (*print)(String this);
  void (*println)(String this);
};

Public String new_string();

#endif
```

G.3 string.c

```
/*-------------------------------------------------------------------------
 * Implements the string interface.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "globals.h"
#include "itoa.h"
#include "strndup.h"
#include "String.h"

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private String realloc_string(String str, int len);
Private void destroy_string(String src);
Private String set_string(String dst, char *str);
Private char *get_string(String src);
Private int size_string(String src);
Private char *copy_string(String src);
Private String append_string(String dst, String src);
Private String append_char_string(String dst, char *src);
Private String append_int_string(String dst, int src);
Private void print_string(String str);
Private void println_string(String str);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public String new_string()
{
  String str = (String)calloc(1, sizeof(struct String));
  str->destroy = destroy_string;
  str->set = set_string;
  str->get = get_string;
  str->size = size_string;
  str->copy_str = copy_string;
  str->append = append_string;
  str->append_char = append_char_string;
  str->append_int = append_int_string;
  str->print = print_string;
  str->println = println_string;

  return str;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private String realloc_string(String str, int len)
{
  str->len = len;
```

```
  str->str = (char *)realloc(str->str, len + 1);

  return str;
}

Private void destroy_string(String src)
{
  if(*src->str) { free(src->str); }
  if(src) { free(src); }

  return ;
}

Private String set_string(String dst, char *str)
{
  if(dst->str) { free(dst->str); }
  dst->len = strlen(str);
  dst->str = (char *)strndup(str, dst->len);

  return dst;
}

Private char *get_string(String src) { return src->str; }

Private int size_string(String src) { return src->len; }

Private char *copy_string(String src)
{
  return (char *)strndup(src->str, src->len);
}

Private String append_string(String dst, String src)
{
  char *s = src->str;
  char *d = dst->str;
  int new_len = dst->len + src->len;
  int old_len = dst->len;

  dst = realloc_string(dst, new_len);

  d = &(dst->str[old_len]);
  while(*s != '\0'){ *d++ = *s++; }
  *d = '\0';

  return dst;
}

Private String append_char_string(String dst, char *src)
{
  String s = new_string();

  s->set(s, src);
  dst = append_string(dst, s);
  s->destroy(s);

  return dst;
}

Private String append_int_string(String dst, int src)
{
  String s = new_string();
```

```
    s->set(s, itoa(src));
    dst = dst->append(dst, s);
    s->destroy(s);

    return dst;
  }

  Private void print_string(String str)
  {
    printf("%s", str->str);

    return ;
  }

  Private void println_string(String str)
  {
    printf("%s\n", str->str);

    return ;
  }
```

G.4 strndup.h

```
/*-------------------------------------------------------------------------
 * Defines the string duplicate interface.
 */

#ifndef __STRNDUP_H
#define __STRNDUP_H

#include "globals.h"

Public char *strndup(char *src, int len);

#endif
```

G.5 strndup.c

```
/*---------------------------------------------------------------------------
 * Implements the string duplicate interface.
 */

#include <string.h>
#include <stdlib.h>
#include "globals.h"

/*---------------------------------------------------------------------------
 * Public functions.
 */

Public char *strndup(char *src, int len)
{
  char *dest = (char *)calloc(len + 1, sizeof(char *));
  dest = (char *)memcpy(dest, src, len);
  dest[len] = '\0';
  return dest;
}
```

G.6 itoa.h

```
/*-------------------------------------------------------------------------
 * Defines the integer to ascii interface.
 */

#ifndef __ITOA_H__
#define __ITOA_H__
#include "globals.h"

Public char *itoa(int n);

#endif
```

G.7 itoa.c

```
/*-------------------------------------------------------------------------
 * Implements the integer to ascii interface.
 */

#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "globals.h"

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private char *reverse(char s[]);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public char *itoa(int n)
{
  int len = 0, i = 0;
  int sign = n;
  char *s;

  if(n == 0) { len = 1; }
  else {
    len = (int)log10(abs(n)) + 1;
    if(n < 0) { len++; }
  }
  s = (char *)calloc(len + 1, sizeof(char));
  if(sign < 0) { len--; }
  while(i < len){
    s[i++] = abs(n % 10) + '0';
    n /= 10;
  }
  if (sign < 0) { s[i++] = '-'; }
  s[i] = '\0';

  return reverse(s);
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private char *reverse(char s[])
{
  int c, i, j;

  for ( i = 0, j = strlen(s)-1; i < j; i++, j--) {
    c = s[i]; s[i] = s[j]; s[j] = c;
  }

  return s;
}
```

G.8 data.h

```
/*-------------------------------------------------------------------------
 * Defines the data interface.
 */

#ifndef __DATA__H
#define __DATA__H

#include "globals.h"


typedef struct data *Data;

struct data {

  char *i_data;      /* Pointer to data */
  int i_size;        /* Size of data    */
  boolean allocated; /* Did we use calloc to allocate the data in i_data? */

  /* Operations */

  /* Size of data */
  int (*size)(Data this);

  /* Get operation */
  char* (*get)(Data this);

  void* (*get_void)(Data this);
  boolean (*set_void)(Data this, void *data, int size);

  /* Set operations */
  boolean (*set)(Data this, char *data, int size);
  boolean (*set_int)(Data this, int data);
  boolean (*set_char)(Data this, char data);

  void (*print)(Data this);
  void (*println)(Data this);
  void (*destroy)(Data this);

};


Data new_data(void);


#endif
```

G.9 data.c

```
/*-------------------------------------------------------------------------
 * Implements the data interface.
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "itoa.h"
#include "strndup.h"
#include "data.h"

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private int size_data(Data this);
Private char *get_data(Data this);
Private void *get_void_data(Data this);
Private boolean set_void_data(Data this, void *data, int size);
Private boolean set_data(Data this, char *data, int size);
Private boolean set_int_data(Data this, int data);
Private boolean set_char_data(Data this, char data);
Private void _free_data(Data this);
Private void print_data(Data this);
Private void println_data(Data this);
Private void destroy_data(Data this);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public Data new_data(void)
{
  Data d = (Data)calloc(1, sizeof(struct data));
  d->allocated = FALSE;
  d->i_data = NULL;
  d->i_size = 0;
  d->size = size_data;
  d->get = get_data;
  d->set = set_data;

  d->set_void = set_void_data;
  d->get_void = get_void_data;

  d->set_int = set_int_data;
  d->set_char = set_char_data;
  d->print = print_data;
  d->println = println_data;
  d->destroy = destroy_data;

  return d;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */
```

```
Private int size_data(Data this) { return this->i_size; }

Private char* get_data(Data this)
{
  return (char *)strndup(this->i_data, this->i_size);
}

Private void _free_data(Data this)
{
  if(this->allocated){
    free(this->i_data);
    this->allocated = FALSE;
  }
  this->i_size = 0;
}

Private boolean set_data(Data this, char *data, int size)
{
  if(this->i_data){ _free_data(this); }
  this->i_data = (char *)calloc(size + 1, sizeof(char));
  this->i_data = memmove(this->i_data, data, size);
  this->i_size = size;
  this->i_data[size] = '\0';
  this->allocated = TRUE;

  return this->i_data == NULL ? FALSE : TRUE;
}

Private boolean set_int_data(Data this, int data)
{
  if(this->i_data){ _free_data(this); }
  this->i_data = itoa(data);
  this->i_size = strlen(this->i_data);

  return this->i_data == NULL ? FALSE : TRUE;
}

Private boolean set_char_data(Data this, char data)
{
  char *chr = &data;
  if(this->i_data){ _free_data(this); }
  this->i_data = chr;
  this->i_size = 1;

  return this->i_data == NULL ? FALSE : TRUE;
}

Private void print_data(Data this)
{
  int i = 0;
  while(i < this->i_size){
    printf("%c", this->i_data[i]);
    i++;
  }

  return ;
}

Private void println_data(Data this)
{
  print_data(this);
  printf("\n");
```

```
  return ;
}

Private boolean set_void_data(Data this, void *data, int size)
{
  if(this->i_data){ _free_data(this); }
  this->i_data = (void *)calloc(size, sizeof(void));
  this->i_data = memmove(this->i_data, data, size);
  this->i_size = size;
  this->allocated = TRUE;

  return this->i_data == NULL ? FALSE : TRUE;
}

Private void *get_void_data(Data this)
{
  return (void *)strndup(this->i_data, this->i_size);
}

Private void destroy_data(Data this)
{
  if(this->i_data){ _free_data(this); }

  if(this){ free(this); }

  return ;
}
```

G.10 integer.h

```
/*-------------------------------------------------------------------------
 * Defines the integer interface.
 */

#ifndef __INTEGER__H
#define __INTEGER__H

#include "globals.h"

typedef struct integer *Integer;

struct integer {

  int value;
  void (*set)(Integer this, int value);
  int (*get)(Integer this);
  void (*destroy)(Integer this);
  int (*size)(Integer this);

};

Public Integer new_integer(int value);

#endif
```

G.11 integer.c

```
/*-------------------------------------------------------------------------
 * Implements the integer interface.
 */

#include <stdlib.h>
#include "integer.h"

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private void destroy_integer(Integer i);
Private void set_integer(Integer this, int value);
Private int get_integer(Integer this);
Private int size_integer(Integer this);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public Integer new_integer(int value)
{
  Integer i = (Integer)calloc(1, sizeof(struct integer));
  i->value = value;
  i->set = set_integer;
  i->get = get_integer;
  i->destroy = destroy_integer;
  i->size = size_integer;

  return i;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */
Private void destroy_integer(Integer this) { if(this){ free(this); } return ; }

Private void set_integer(Integer this, int value) { this->value = value; }

Private int get_integer(Integer this){ return this->value; }

Private int size_integer(Integer this) { return sizeof(struct integer); }
```

G.12 library.h

```
/*-------------------------------------------------------------------------
 * Defines the interface for opertations on dynamic loaded libraries.
 */

#ifndef __LIBRARY__H
#define __LIBRARY__H

#include "globals.h"

Public void *open_library(char *pathToLibrary);
Public char *library_error(void);
Public void *load_library_symbol(void *handle, char *symbol);
Public boolean close_library( void *handle);

#endif
```

G.13 library.c

```
/*-------------------------------------------------------------------------
 * Implements the interface for opertations on dynamic loaded libraries.
 */

#include <dlfcn.h>
#include <stdio.h>
#include "globals.h"
#include "library.h"

/*-------------------------------------------------------------------------
 * Public funtions.
 */

/*
 * Name:
 *  Public void *open_library(char *pathToLibrary)
 *
 * Description:
 *  Loads a dynamic library from the filename given as argument pathToLibrary
 *  and returns a "handle" for the dynamic library.
 *
 *  If pathToLibrary is a NULL pointer, then the returned handle is for
 *  the main program.
 *
 *  If open_library fails, then NULL is returned and an error message
 *  can be extracted from the function library_error().
 *
 * Parameters:
 *  char *pathToLibrary - Absolute path to dynamic library
 *
 * Returns:
 *  Upon success a handle for the dynamic library,
 *  otherwise NULL is returned.
 *
 */
Public void *open_library(char *pathToLibrary)
{
  return dlopen(pathToLibrary, RTLD_LAZY);
}

/*
 * Name:
 *  Public char *library_error(void)
 *
 * Description:
 *  Returns the most resently reported error from an Library routine.
 *
 * Paramters:
 *
 * Returns:
 *  An error message.
 */
Public char *library_error(void)
{
  return (char *)dlerror();
}
```

```
/*
 * Name:
 *  Public void *load_library_symbol(void *handle, char *symbol)
 *
 * Description:
 *  Returns the address of a symbol in a dynamic library.
 *  The dynamic library and symbol are given as arguments.
 *  The symbol represents a function or variable.
 *
 * Paramters:
 *  void *handle - Handler of dynamic library
 *  char *symbol - Name of the symbol
 *
 */
Public void *load_library_symbol(void *handle, char *symbol)
{
  char *error;
  void *retval;

  retval = dlsym(handle, symbol);
  if((error = (char *)dlerror()) != NULL ){
    printf("%s", error);
  }
  return retval;
}


/*
 * Name:
 *  boolean close_library(void *handle)
 *
 * Description:
 *  Closes a previously opened dynamic library.
 *  The dynamic library to close is given as argument handle.
 *
 * Arguments:
 *  void *handle - The handler for the dynamic library to close.
 *
 * Returns:
 *  TRUE upon success
 *  FALSE otherwise
 */
boolean close_library(void *handle)
{
  dlclose(handle);
  return library_error() == NULL ? TRUE : FALSE;
}
```

G.14 list.h

```
/*-------------------------------------------------------------------------
 * Defines the list interface
 */

#ifndef __LIST__H
#define __LIST__H

#include "globals.h"
#include "element.h"

typedef struct list *List;

struct list {

  Element theList;
  Element cursor;

  List (*add)(List this, void *data);
  List (*delete)(List this, void *data, boolean cmp(void *e1, void *e2));
  List (*delete_all)(List this);
  Element (*get)(List this, void *data, boolean cmp(void *e1, void *e2));
  Element (*get_first)(List this);
  Element (*get_next)(List this);

};


Public List new_list(void);

#endif
```

G.15 list.c

```
/*-------------------------------------------------------------------------
 * Implements the list interface.
 */

#include <stdlib.h>
#include "list.h"

#define CURSOR(t) t->cursor
#define ELEMENT(t) t->theList
#define SET_ELEMENT(t, a) t->theList = a
#define SET_CURSOR(t, a) t->cursor = a

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private List add_list(List this, void *data);
Private List delete_list(List this, void *data, \
                         boolean cmp(void *e1, void *e2));
Private List delete_all_list(List this);
Private Element get_list(List this, void *data, \
                         boolean cmp(void *e1, void *e2));
Private Element get_next_list(List this);
Private Element get_first_list(List this);

/*-------------------------------------------------------------------------
 * Private functions.
 */

Public List new_list(void)
{
  List this = (List)calloc(1, sizeof(struct list));
  this->theList = new_element();
  this->add = add_list;
  this->delete = delete_list;
  this->delete_all = delete_all_list;
  this->get = get_list;
  this->get_next = get_next_list;
  this->get_first = get_first_list;

  return this;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private List add_list(List this, void *data)
{
  Element el = new_element();
  SET_ELEMENT(this,  el->set_value(el->set_next(el, ELEMENT(this)), data));

  return this;
}

Private List delete_list(List this, void *data,\
                         boolean cmp(void *e1, void *e2))
```

```
{
  Element tmp = ELEMENT(this);
  Element parent = NULL;

  /* Case 1: The element to delete is the first element */
  if(cmp(tmp->get_value(tmp), data)) {
    SET_ELEMENT(this, tmp->delete(tmp, parent));
    return this;
  }

  /* Case 2: The element to delete is in the list */
  parent = tmp;
  tmp = tmp->get_next(tmp);

  while(!tmp->is_null(tmp)){
    if(cmp(tmp->get_value(tmp), data)){
      tmp->delete(tmp, parent);
      return this;
    }
    parent = tmp;
    tmp = tmp->get_next(tmp);
  }

  /* Case 3: The element to delete is not in the list */
  return this;
}

Private List delete_all_list(List this)
{
  Element tmp1 = ELEMENT(this);
  Element tmp2 = ELEMENT(this);

  while(!tmp1->is_null(tmp1)){
    tmp1 = tmp2->get_next(tmp2);
    tmp2->destroy(tmp2);
    tmp2 = tmp1;
  }
  return NULL;
}

Private Element get_list(List this, void *data, \
                         boolean cmp(void *e1, void *e2))
{
  Element tmp = ELEMENT(this);

  while(!tmp->is_null(tmp)){
    if(cmp(tmp->get_value(tmp), data)){
      return tmp;
    }
    tmp = tmp->get_next(tmp);
  }
  return NULL;
}

Private Element get_first_list(List this)
{
  CURSOR(this) = ELEMENT(this);
  return ELEMENT(this);
}

Private Element get_next_list(List this)
{
```

```
    Element el = CURSOR(this);
    el = el->get_next(el);

    /* If cursor pointed to the last element when get_next_list was
       called, we have to reset the cursor and tell the user there
       are no next element.
    */
    if(el->is_null(el)){
      el = ELEMENT(this);
      return NULL;
    }
    SET_CURSOR(this, el);
    return el;
  }
```

G.16 element.h

```
/*-------------------------------------------------------------------------
 * Defines the interface for list elements.
 */

#ifndef __ELEMENT_H
#define __ELEMENT_H

#include "globals.h"

typedef struct element *Element;

struct element {

  Element next;
  void *data;

  boolean (*is_null)(Element this);
  boolean (*is_data_null)(Element this);
  void (*destroy)(Element this);
  Element (*delete)(Element this, Element prev);
  Element (*get_next)(Element this);
  void *(*get_value)(Element this);
  Element (*set_next)(Element this, Element next);
  Element (*set_value)(Element this, void *data);
};

Public Element new_element(void);

#endif
```

G.17 element.c

```
/*-------------------------------------------------------------------------
 * Implements the interface for list elements.
 */

#include <stdlib.h>
#include <stdio.h>
#include "element.h"


#define ELEMENT(t) t->next
#define DATA(t) t->data

/*-------------------------------------------------------------------------
 * Private function declarations.
 */

Private boolean is_null_element(Element this);
Private boolean is_data_null_element(Element this);
Private void destroy_element(Element this);
Private Element delete_element(Element this, Element parent);
Private Element get_next_element(Element this);
Private void *get_value_element(Element this);
Private Element set_next_element(Element this, Element next);
Private Element set_value_element(Element this, void *data);

/*-------------------------------------------------------------------------
 * Public functions.
 */

Public Element new_element(void)
{
  Element el = (Element)calloc(1, sizeof(struct element));
  el->data = NULL;
  el->is_null = is_null_element;
  el->is_data_null = is_data_null_element;
  el->destroy = destroy_element;
  el->delete = delete_element;
  el->get_next = get_next_element;
  el->get_value = get_value_element;
  el->set_next = set_next_element;
  el->set_value = set_value_element;

  return el;
}

/*-------------------------------------------------------------------------
 * Private functions.
 */

Private boolean is_null_element(Element this)
{
  return ELEMENT(this) == NULL;
}

Private boolean is_data_null_element(Element this)
{
  return DATA(this) == NULL;
```

```
}

Private void destroy_element(Element this)
{
  if(this != NULL){
    /* if(!is_data_null_element(this)){ free(DATA(this)); } */
    free(this);
  }
}

Private Element delete_element(Element this, Element parent)
{
  if(parent == NULL){
    Element el = this->get_next(this);
    this->destroy(this);
    this = el;
    return this;
  }
  parent = parent->set_next(parent, this->get_next(this));
  this->destroy(this);
  return this;
}

Private Element get_next_element(Element this)
{
  return this->is_null(this) ? NULL : this->next;
}

Private void *get_value_element(Element this)
{
  return this->is_data_null(this) ? NULL : this->data;
}

Private Element set_next_element(Element this, Element next)
{
  this->next = next;
  return this;
}

Private Element set_value_element(Element this, void *data)
{
  if(!this->is_data_null(this)){
    free(DATA(this));
  }
  /* OBS no copy is made, hence the caller must NOT change the data */
  DATA(this) = data;
  return this;
}
```

Appendix H

Shell scripts

PiVO uses three directories on the underlying host operating system. The directories are used to store the templates, file operations and files. The templates and the file operations are stored as dynamic libraries and are loaded in to PiVO when needed using the dynamic library management code listed in sections G.12 to G.13. A file in PiVO is stored on disk as a Berkley DB database, and the database is stored on disk under a predefined directory.

The absolute path names of the directories used by PiVO are defined in header files. The header files are automatically generated by shell scripts that are executed from make files when the source code is compiled. Automatically generating the header files makes it easier to port the code to a new machine. There is no need to edit any files, the issue of the make command will suffice.

Along with the header files, a shell script called `pivo` is created, which is used to start the PiVO shell in a controlled way.

H.1 build_header (Shell)

```
#!/bin/bash

#-------------------------------------------------------------------------------
# Autogenerates templatepath.h, which defines the location of the templates.
#


FILE=templatepath.h
BASE=`dirname $PWD`;

if [ -f $FILE ];
then
        rm $FILE;
fi

cat >> $FILE << END

/*-------------------------------------------------------------------------------
 * Defines the path to the templates
 * This file was autogenerated with build_header
 */

#ifndef __TEMPLATEPATH_H
#define __TEMPLATEPATH_H

#define TEMPLATE_PATH "$BASE/templates/"

#endif
END
```

H.2 build_pivo (Shell)

```
#!/bin/bash

#-------------------------------------------------------------------------------
# Autoegenerates pivo, a wrapper for the shell that clears the pool
# directory and calls shell with init.shell as argument.

BINARY=pivo
BASE=`dirname $PWD`;

cat >> $BINARY << END
#!/bin/bash

export DYLD_LIBRARY_PATH=$BASE/lib

# Clean pool
rm $BASE/pool/* 2>/dev/null

$BASE/bin/shell $BASE/bin/init.shell

END

chmod 700 $BINARY
```

H.3 build_dir_header (Templates)

```
#!/bin/bash

#-------------------------------------------------------------------------------
# Autogenerates the file dirmethods.h, which defines the location of
# the directory methods implementation.
#

FILE=dirmethods.h
BASE=`dirname $PWD`;

echo "Build dirmethods.h"

if [ -f $FILE ];
then
        rm $FILE;
fi

cat >> $FILE << END

/*-------------------------------------------------------------------------------
 * Defines the absolut path to the directory template modules
 * This file was autogenerated do not edit by hand.
 */

#ifndef __DIRMETHODS_H
#define __DIRMETHODS_H

#define DIR_METHODS "$BASE/methods/dir_methods.so"

#endif
END
```

H.4 build_header (Templates)

```
#!/bin/bash

#-------------------------------------------------------------------------------
# Autogenerats the file header.h, which defines the location of the
# _a_open, _a_close, zip, unzip, and cat method implementation.
#

FILE=methods.h
BASE=`dirname $PWD`;

echo "Build header.h"

if [ -f $FILE ];
then
        rm $FILE;
fi

cat >> $FILE << END

/*-------------------------------------------------------------------------------
 * Defines the absolut path to the methods a_open and cat
 * This file was autogenerated with build_header
 */

#ifndef __METHODS_H
#define __METHODS_H

#define A_OPEN_METHOD "$BASE/methods/a_open_method.so"
#define CAT_METHOD "$BASE/methods/cat_method.so"
#define ZIP_METHODS "$BASE/methods/zip_methods.so"

#endif
END
```

H.5 libsrc build_pool (PiVO Core Library)

```
#!/bin/bash

#-------------------------------------------------------------------------------
# Autogenerates pool.h, which defines the location of the pool directory.
#

FILE=pool.h
BASE=`dirname $PWD`;

if [ -f $FILE ];
then
        rm $FILE;
fi


cat >> $FILE << END

/*-------------------------------------------------------------------------------
 * Defines the POOL constant. This file was autogenerated with build_pool.h.
 */

#ifndef __POOL__H
#define __POOL__H

#define POOL "$BASE/pool/"


#endif
END
```

Appendix I

Debug

In this appendix the source code for a simple debug utility used to read the contents of a Berkeley DB database is presented. The debug utility can be used to list a files attributes and their values. The debug utility is not capable of displaying the data in a PiVO file unless it is represented as a string. This means that the data part of a directory or the compressed data of a `plain` file cannot be displayed using the debug program.

I.1 dbg_db.c

```
/*-----------------------------------------------------------------------
 * Implements a simple debug utility, which lists the contents of a db-file.
 */

#include <string.h>
#include <sys/types.h>
#include <stdio.h>
#include <db.h>

char *progname;

int display(char *database)
{
  DB *dbp;
  DBC *dbcp;
  DBT key, data;
  int close_db, close_dbc, ret;

  close_db = close_dbc = 0;

  /* Open the database. */
  if ((ret = db_create(&dbp, NULL, 0)) != 0) {
    fprintf(stderr, "%s: db_create: %s\n", progname, db_strerror(ret));
    return (1);
  }

  /* Turn on additional error output. */
  dbp->set_errfile(dbp, stderr);
  dbp->set_errpfx(dbp, progname);

  /* Open the database. */
  if ((ret = dbp->open(dbp, NULL, database, \
       NULL, DB_UNKNOWN, DB_RDONLY, 0)) != 0) {
    dbp->err(dbp, ret, "%s: DB->open", database);
    goto err;
  }

  /* Acquire a cursor for the database. */
  if ((ret = dbp->cursor(dbp, NULL, &dbcp, 0)) != 0) {
    dbp->err(dbp, ret, "DB->cursor");
    goto err;
  }

  /* Initialize the key/data return pair. */
  memset(&key, 0, sizeof(key));
  memset(&data, 0, sizeof(data));

  /* Walk through the database and print out the key/data pairs. */
  while ((ret = dbcp->c_get(dbcp, &key, &data, DB_NEXT)) == 0)
    printf("%.*s : %.*s\n", (int)key.size, (char *)key.data, (int)data.size, \
                            (char *)data.data);
  if (ret != DB_NOTFOUND) {
    dbp->err(dbp, ret, "DBcursor->get");
    goto err;
  }
```

```
    err:
     if (close_dbc && (ret = dbcp->c_close(dbcp)) != 0){
        dbp->err(dbp, ret, "DBcursor->close");
     }
     if (close_db && (ret = dbp->close(dbp, 0)) != 0){
        fprintf(stderr, "%s: DB->close: %s\n", progname, db_strerror(ret));
     }

     return (0);
   }

   int main(int argc, char *argv[])
   {
     progname = argv[0];

     if(argc == 2){ display(argv[1]); }
     else { printf("Usage: %s database\n", progname); }

     return 1;
   }
```