5-2010

Naked Object File System (NOFS): A Framework to Expose an Object-Oriented Domain Model as a File System

Joseph P. Kaylor

Konstantin Läufer
Loyola University Chicago, klaeufer@gmail.com

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Naked Object File System (NOFS): A Framework to Expose an Object-Oriented Domain Model as a File System

Joe Kaylor
Loyola University Chicago
joekaylor@gmail.com

George K. Thiruvathukal
Loyola University Chicago
gkt@cs.luc.edu

Konstantin Läufer
Loyola University Chicago
laufer@cs.luc.edu

Abstract

We present Naked Objects File System (NOFS), a novel framework that allows a developer to expose a domain model as a file system by leveraging the Naked Objects design principle. NOFS allows a developer to construct a file system without having to understand or implement all details related to normal file systems development. In this paper we explore file systems frameworks and object-oriented frameworks in a historical context and present an example domain model using the framework. This paper is based on a fully-functional implementation that is distributed as free/open source software, including virtual machine images to demonstrate and study the referenced example file systems.

1 Introduction

Kernel mode file systems require a complex understanding of systems programming, systems programming languages, and the underlying operating system. There are fewer people who have this skill set as object-oriented frameworks and languages are becoming more and more popular. For businesses with core skill sets in applications and enterprise development, it is becoming more difficult to find developers with systems skill sets to implement file systems, yet the need for systems expertise has not diminished at all, evidenced by simultaneous growing interest in embedded platforms. There is also much code that has already been developed using the patterns available and common to enterprise application frameworks that either cannot be used or are difficult to reuse in systems development. What is needed is a framework that allows enterprise development techniques and patterns to be applied to file systems development (not to mention systems programming in general). NOFS is our attempt to provide such a framework. Further, we believe that the file system itself provides a structured interface to the user and that it can be thought of as a natural object-oriented user interface that is common among Naked Object frameworks.

2 Background and Related Work

In very early systems, development of new file systems code was a challenge because of high coupling with storage device architecture and kernel code.

In the 1970s, with the introduction of MULTICS, UNIX, and other systems of the time, more structured systems with separated layers became more common. UNIX used a concept of *i-nodes*[**?**], which where a common data structure that described structures on the file system. Different file system implementations could share the same i-node structure. This included on-disk file systems and network file systems. Early UNIX operating systems shared a common disc and file system cache and other structures related to making calls to the I/O layer that managed the discs and network interfaces.

With newer UNIX-like systems such as 4.2 BSD and SunOS, an updated architecture was developed called vnodes [**?**]. The goal of vnodes was to split file system independent functionality in the kernel from the file system dependent functionality. Things like path parsing, buffer cache, i-node tables, and other structures became more shareable. Also, operations with vnodes became reentrant which made it possible for new behavior to be stacked on top of other file system code or to modify existing behavior. Vnodes also helped to simplify file systems design and make file systems implementations more portable to other UNIX-like systems. Many modern UNIX-like systems have a vnodes like layer in their file systems code.

With the advent of micro-kernel architectures, file systems being built as user mode applications became more common and popular even in operating systems with monolithic kernel architectures. Several systems with different design philosophies have been built. We describe three of these systems that are most closely related

to NOFS: FUSE, ELFS, and Frigate.

The Extensible File System (ELFS hereafter) is an object-oriented framework built on top of the file system that is used to simplify and enhance the performance of the interaction between applications and the file system [**?**]. ELFS uses class definitions to generate code that takes advantage of pre-fetching and caching techniques. ELFS also allows developers to automatically take advantage of parallel storage systems by using multiple worker threads to perform reads and writes. Also, since ELFS has the definition of the data structures, it can build efficient read and write plans. The novelty of ELFS is that the developer can use an object-oriented architecture and allow ELFS to take care of the details.

Frigate is a framework that allows developers to inject behavioral changes into the file system code of an operating system [**?**]. Modules built in frigate are run as user-mode servers that are called to by a module that exists in the operating system's kernel. Frigate takes advantage of the reentrant structure of vnodes in UNIX-like operating systems to allow the developer of Frigate modules to layer behavior on top of existing file systems code. Frigate also allows the developer to tag certain files with additional metadata so that different Frigate modules can automatically work with different types of files. The novelty of Frigate is that developers do not need to understand operating systems development to modify the capabilities of file systems code and they can test and debug their modules as user-mode applications. Frigate developers, however, still need to be aware of the UNIX file system structures and functions.

File Systems in Userspace (FUSE hereafter) is a user mode file systems framework. FUSE is supported by many UNIX-like operating systems such as Linux, FreeBSD, NetBSD, OpenSolaris, and Mac OSX. The interface supported by FUSE is very similar to the set of UNIX system calls that are available for file and folder operations. Aside from the ability to make calls into the host operating system, there is less shared with the operating system than with vnodes such as path parsing. FUSE has helped many file system implementations such as NTFS and ZFS to be portable to many operating systems. Since FUSE file systems are built as user-land programs, they can be easier to develop in languages other than C or C++, easier to unit test, and easier to debug.

Naked Objects [**?**] is the term used to describe the design philosophy of using plain object-oriented domain models to build entire applications. In the realm of desktop applications, Naked Object frameworks remove the concern of the developer in implementing user interfaces, model-view-controller patterns, and persistence layers. These components are generated for the domain model by the Naked Objects framework automatically either through the use of reflection or through additional metadata supplied with the domain model.

A characteristic feature of Naked Object frameworks is that they present an object-oriented user interface. Applications where the user is treated more as a problem solver than as a process follower benefit from an object-oriented user interface [**?**, p. 41]. For many applications, processes are very important and an object-oriented user interface is not the best fit. We believe that the interface presented to the programmer and to the user of a file system is also object-oriented. In a file system, the components are not exposed to the user to facilitate the moving, reading, writing, creation, or deletion of files and folders. These actions are accomplished with external programs and references to the actual objects as command line parameters. The user interaction with file systems is a noun-verb style of interaction and not a verb-noun interaction which is more common with typical desktop applications . Like the naked object user interfaces, file systems "provide the user with a set of tools which to operate and .. does not dictate the users sequence of actions"[**?**, p. 41].

3 Naked Objects File System (NOFS)

3.1 Novelty of NOFS

There are *three contributions* made by the NOFS framework. *The first* is that NOFS demonstrates the file system can be used as an object-oriented user interface in a Naked Objects framework and that the Naked Objects design principle can be applied successfully to file systems development, especially file systems of an experimental nature. *The second* contribution is that NOFS inverts and simplifies the normal file system contract. In FUSE and operating system kernels, there are a series of functions to implement and data structures to work with. With the NOFS framework, a domain model is inspected to produce a file system user interface. Domain models for NOFS do not implement file system contracts or work with file system structures. Instead, they are described with metadata that is used by NOFS to allow the domain model to interact with the FUSE file system framework. *The third* contribution made by the NOFS framework is that by providing an object-oriented framework to develop file systems, we allow developers who are unfamiliar with systems or UNIX programming to more easily and rapidly implement experimental or lightweight file systems. With this object-oriented framework, it becomes easier to unit test a file system implementation because details of the operating system do not need to be stubbed or mocked out; only the domain model needs to be verified.

3.2 Implementing a Domain Model with NOFS

Here we will explore developing a domain model with NOFS. We will look at parts of two domain models: an address book domain model which was developed for presentation purposes, and a Flickr domain model. The address book domain model models a series of contacts' names and phone numbers. The Flickr domain model models a series of user accounts and their photos from the Flickr.com online photo service. Flickr exposes a RESTful, service oriented API that we have used in our domain model.

3.3 Implementing Files

In NOFS, files are modeled as plain classes that are described with metadata. The methods on the class are not constrained to any specific interface but are used to model the structure of the data in a file. There are two ways for instances of class to expose their data: through translation of the return values of public methods to structured XML files or by defining the structure of these files by implementing an interface with read and write methods.

Listing 1: An example NOFS Domain Object

```
type ContactDOC =
  IDomainObjectContainer<Contact>;

@DomainObject
public class Contact {
  private String _name;
  private String _phoneNumber;
  private ContactDOC _container;

  @ProvidesName
  public String getName() {
    return _name;
  }

  @ProvidesName
  public void setName(String name)
    throws Exception {
      _name = name;
      _container.ObjectChanged(this);
  }

  public String getPhoneNumber()
    { return _phoneNumber; }

  public void setPhoneNumber(String value)
    throws Exception {
      _phoneNumber = value;
      _container.ObjectChanged(this);
  }

  @NeedsContainer
  public void setContainer(
    ContactDOC container) {
      _container = container;
  }
}
```

In listing 1, the class Contact (`http://tinyurl.com/nofs-Contact`) marks itself as a file object by using the @DomainObject Java annotation[1]. It tells NOFS that it manages its own file name with the @ProvidesName annotation on the getName accessor and the setName mutator methods. The persistence mechanism of NOFS is injected upon construction of the Contact class through the setContainer method which is marked by the @NeedsContainer method. An example representation of the Contact class as a file in an NOFS file system is as follows:

Listing 2: Contact Domain Object Representation

```
<Contact>
<PhoneNumber>555-1234</PhoneNumber>
</Contact>
```

In this example, the class FlickrPhoto (`http://tinyurl.com/nofs-FlickrPhoto`) marks itself as a file object by using the @DomainObject Java annotation. It tells NOFS that it is immutable by setting the CanWrite member of the DomainObject annotation to false. FlickrPhoto's reponsibility is to model a graphical image from the Flickr photo sharing website. Since it is convenient to expose to the file system these photos as an image file and not as an XML file, FlickrPhoto provides read and write methods as defined by the IProvidesUnstructuredData NOFS interface.

Listing 3: Flickr Photo Domain Object

```
@DomainObject(CanWrite=false)
public class FlickrPhoto
  implements IProvidesUnstructuredData {

  private byte[] _data;
  private String _name;

  public FlickrPhoto() {}

  public void setData(byte[] data)
    { _data = data; }
```

[1] Our listings have been formatted for conciseness. For lengthy examples, we refer the reader to a permanent (but also shortened) repository link. All examples and a test virtual machine are available via our NOFS home page at Google Code.

```
  @ProvidesName
  public String getName()
    { return _name; }

  @ProvidesName
  public void setName(String name)
    { _name = name; }

  public boolean Cacheable()
    { return false; }

  public long DataSize()
    { return _data.length; }

  public void Read(ByteBuffer buffer,
    long offset, long length) {

    for (long i = offset;
         i < offset + length
           && i < _data.length; i++)
      buffer.put(_data[(int)i]);
  }

  ...
}
```

In both the address book and Flickr examples, the domain models did not need to be aware of the UNIX file system calls required by the FUSE framework such as chmod, chown, mknod, and others. The developers of these domain models were able to be mostly concerned with implementing the details of the structure of the domain model.

3.4 Implementing Folders

As with files, folders in NOFS are modeled as plain classes that can be described with metadata. Folders can be modeled in one of two ways. The first way is as a class that is marked as a folder or as a method that returns a List type collection that is itself marked as a folder. Folders can be model as children individual files and other folders.

Listing 4: Implementing Folders using @FolderObject Annotation

```
@DomainObject
@FolderObject(CanAdd=true,
  CanRemove=true)
public class Category extends
  LinkedList<Contact> {

  private String _name;

  @ProvidesName
  public void setName(String name)
    { _name = name; }

  @ProvidesName
  public String getName()
    { return _name; }
}
```

In this example, the class Category (`http://tinyurl.com/nofs-Category`) marks itself as a folder by using the @FolderObject annotation. This is an example of a folder object that extends a collection of another object that is a file object type (in this case Contact). In this example, the CanAdd and CanRemove members of the @FolderObject annotation are used to tell NOFS that the folder modeled by the Category class is mutable by the user of the file system.

The class FlicrkRoot (`http://tinyurl.com/nofs-FlickrRoot`), which is a part of an example Flickr file system built for NOFS, marks itself as a folder by using the @FolderObject annotation. This class also marks itself with the @RootFolderObject annotation. The @RootFolderObject annotation is used for one class per file system to mark which folder class is the root of the NOFS file system. The FlickrRoot class exposes another folder object through the getUsers accessor method. This method is marked as a folder with the @FolderObject annotation and is allowed to be a folder because its return type is a collection of instances of a class that is a folder object (in this case FlickrUser). As with files in NOFS, folder classes don't need to implement the various FUSE methods such as getdir, readdir, chmod, and others. NOFS implements the FUSE contract and examines the folder objects to answer all of the calls.

3.5 Exposing Methods to the File System

In a user interface, reading and updating data is an important operation. NOFS accomplishes this by exposing folder and file objects that can be read from and written to using common file operations such as reading from files, writing to files, creating new files or folders, and deleting existing files and folders. Another important way of interacting with a user interface is through action oriented user interface components such as buttons or menus. NOFS allows file systems to expose these actions to the file system by generating Perl scripts that are exposed on the file system around methods that are marked as executable on a plain class.

In the FlickrRoot example, a method called AddUser is marked as executable using the @Executable Java annotation. When NOFS encounters a method marked with the @Executable annotation, it will generate a script file

on the file system that when invoked with the appropriate arguments will serialize those arguments into a standard XML format and send them to NOFS using an IPC channel which will in turn invoke the AddUser method with the arguments from the script. In the case of the AddUser script, it will accept a single primitive argument that is the user name. The AddUser script will appear as a file in every instance of the FlickrRoot folder.

Listing 5: RemoveAContact() @Executable example

```
type ContactDOC =
  IDomainObjectContainer<Contact>;

@Executable
public void RemoveAContact(Contact c)
  throws Exception {

  ContactDOC container
    = GetContainerManager()
      .GetContainer(Contact.class);
  container.Remove(c);
  _bookContainer.ObjectChanged(this);
}
```

In the listing 5, RemoveAContact (`http://tinyurl.com/nofs-Book`) is marked as an executable by using the @Executable Java annotation. This method is different than the previous example in that it takes a non primitive reference to an instance of the Contact class. When the user invokes the script, they will invoke it with a path reference to a Contact file. This path will then be resolved by NOFS to the specific instance of the Contact class that represents the file path. This instance of the Contact class will then be passed to the RemoveAContact method.

4 Conclusion

We believe that we have successfully demonstrated that the Naked Objects design principles can be applied to a file systems framework with NOFS. We have also shown how domain models can be exposed as an object-oriented user interface through the file system. Finally we have demonstrated how it is possible to remove the lower level details of the FUSE framework from the concern of the developer and allow them to be more concerned with the development of their domain model.

5 Experiences and Lessons Learned

Early on, we chose the Java platform to build NOFS, given the availability of a strong *ecosystem* for test-driven design and development and enterprise development. We believe at this time that we would have been better served to have used the .NET platform through Mono, even considering the present maturity of the Mono environment, which lags Microsoft's .Net significantly. A future Windows port would be simpler because the native call facility in .NET is more straight forward than JNI and because of the presence of user mode .NET file system drivers such as Dokan (which is used to provide FUSE support on Windows). Nevertheless, without Java, the robustness of the current prototype would likely not have been achieved. In future work focused on performance and benchmarking, however, we are likely to take a different tact when it comes to the overall development environment.

6 Future Work

As presented in this paper, NOFS provides a framework to translate a domain model into a file system. This reduces the need of the developer to understand the methods and structures that are necessary to implement a FUSE file system. NOFS however, does not provide a generic or object-oriented way to work with block devices or network/service access. The developer of an NOFS file system must still be concerned with the access to resources that are outside of the NOFS persistence layer if any. We plan to explore how to build such a layer, which we believe will be of great value in a cloud-enabled world.

References

[1] John F. Karpovich, Andrew S. Grimshaw, and James C. French. Extensible file system (elfs): an object-oriented approach to high performance file i/o. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 191–204, New York, NY, USA, 1994. ACM.

[2] Ted H. Kim and Gerald J. Popek. Frigate: an object-oriented file system for ordinary users. In *COOTS'97: Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 9–9, Berkeley, CA, USA, 1997. USENIX Association.

[3] S. R. Kleiman. Vnodes: An architecture for multiple file system types in sun unix. pages 238–247, 1986.

[4] R. Pawson. *Naked Objects*. PhD thesis, Trinity College, Dublin, Ireland, 2004.

[5] K Thompson. Unix implementation. pages 26–41, 1986.