5-2012

Simplifying Domain Modeling and Memory Management in User-Mode Filesystems with the NOFS Framework

Joseph P. Kaylor

Konstantin Läufer
Loyola University Chicago, klaeufer@gmail.com

George K. Thiruvathukal
Loyola University Chicago

Recommended Citation

J. P. Kaylor, K. Läufer, and G. K. Thiruvathukal, Simplifying domain modeling and memory management in user-mode filesystems with the NOFS framework. In Proc. 2010 IEEE Intl. Conf. on Electro/Information Technology (EIT), Indianapolis, Indiana, May 2012, doi:10.1109/EIT.2012.6220733.

Simplifying Domain Modeling and Memory Management in User-Mode Filesystems with the NOFS Framework

Joseph P. Kaylor, Konstantin Läufer and George K. Thiruvathukal
Emerging Technologies Laboratory
Department of Computer Science
Loyola University Chicago
Chicago, IL 60640
{jkaylor,laufer,gkt}@etl.luc.edu

***Abstract*—Transparent access to remote data sets and data arising from web services is a non-trivial challenge to application developers. This early stage work addresses this challenge with NOFS, an object-oriented framework for creating filesystems to support domain specific functionality. While an early stage work, we present a solution to solve the access problem. Our solution greatly simplifies the task of filesystems development by providing the glue code needed between a domain model and the filesystem contract. We demonstrate support for domain models that are larger than physical memory and demonstrate how the concerns of caching can be removed from user-mode filesystem implementations. Future work will addresses more robust solutions to caching and other performance strategies.**

I. Introduction

In this paper, we discuss the need for user-mode filesystems in scientific computing to solve the problem of transparent data access. We explain how to simplify data access with user-mode filesystems. We explain how to simplify the task of building user-mode filesystems with our extensions to the NOFS framework.

Specifically, in this paper we discuss how existing storage-based user-mode filesystems must be designed with domain object size and physical memory usage in mind. We then point out why caching is important for these user-mode filesystems and illustrate how this concern contributes to a significant portion of the code in a user-mode filesystem. We show how the weak reference pattern can be leveraged in a filesystem framework and, specifically, how to apply it to an example file-and-folder domain model. Finally, we demonstrate how the naked objects architecture effectively supports storage-based user-mode filesystems and how a naked objects framework can manage concerns such as caching and domain object lifetime.

II. Related Work

A. The Importance of Inter-Process Communication Through the Filesystem

In modern operating systems, most methods of inter-process communication (IPC) can be represented through the filesystem. Among these are pipes, domain sockets, memory mapped files, and regular files. These methods of IPC allow for separate programs to communicate and coordinate with each other. This communication and coordination allows software programs to be broken down into separate and reusable components. Without the ability to communicate across process boundaries, many components would have to be present in the address spaces of many programs, mostly through shared libraries. With IPC, software programs and components can be composed and reused in several different ways.

B. The Role of Application Filesystems in Software Composition

Application filesystems further enhance and expand the traditional filesystem based methods of IPC by representing complex file structures, offering advanced filesystem semantics, or through representing or composing one or more external services through the filesystem contract. IPC through application filesystems allow local abstractions and local compositions to work without opening a network socket or needing to write code to comply with a network protocol.

Some filesystems promote composition by presenting a network protocol through a filesystem contract. An excellent example of this is Plan 9's filesystem service: 9P [1]. Through 9P, Plan 9 is able to abstract many network protocols and external resources. Among these resources are: HTTP and FTP protocols, managing network sockets, and a filesystem based abstraction for Plan 9's window manager.

It is possible to achieve inter-machine IPC through the use of network filesystems. In network filesystems that support file locking mechanisms and have adequate solutions to the cache coherency problem, it is possible to perform inter-machine IPC through filesystem operations.

In the past several years, many FUSE [2] based application filesystems have been built to act as clients for popular web services such as Flickr, IMAP email services, Amazon S3, and several others. In our own research, we leveraged our existing NOFS framework to implement RestFS [3], a dynamically reconfigurable filesystem for exposing remote restful resources as a local filesystem. With RestFS, we were able to demonstrate an architecture that could map several different restful web services such as Yahoo! Placefinder, Flickr, and Twitter into local filesystem representations. We were able to further demonstrate how these web services and local software components could be composed locally and re-exposed as restful web services.

C. The Role of User-Mode Filesystems in Software Composition for Large Datasets

In our exploration of user-mode filesystems development, we have worked with three categories of user-mode filesystems. First is the storage filesystem. Storage filesystems are primarily concerned with the traditional role of filesystems used as a means to store regular files and folders. A good example of a storage oriented user-mode filesystem would be NTFS-3G. NTFS-3G is a FUSE filesystem that allows UNIX-like operating systems to mount NTFS volumes in read-write mode. The second category is the connector filesystem. Connector filesystems provide mappings between a resource and a local filesystem. RestFS is an example of a connector filesystem. RestFS provides a way to create files and folders that can be configured to map filesystem calls to a remote restful web service. The third category is the application filesystem. Application filesystems provide behavior in addition to the resources that are represented by the filesystem.

To encourage reuse and dissemination of information, scientists often publish datasets using a format standard to their field and provide one or more libraries for popular programming languages to read from and write to these datasets. To create a new library for a new language, an entirely new library must be constructed for the new language, or where possible, bindings from the new language to an existing library in another language can be constructed. Where datasets are published in formats such as XML or CSV and where good documentation exists, the challenge of writing new libraries for scientific datasets is greatly lessened.

User-mode file systems of all types can play an important role with large datasets. Datasets in formats such as XML or CSV have several performance disadvantages due to their human readability. Among the sources of these issues are: greater amounts of whitespace characters, representation of numeric values as text instead of binary, and challenges determining seek offsets for random file access. So, in part due to performance and data size concerns, many datasets are published in a binary format. User-mode filesystems can help to bridge this divide. By constructing a user mode-files system on top of a binary formatted data set, it is possible to represent a filesystem as human readable files such as XML or CSV. With these types of files, it is simpler to implement software to consume them in other programming languages.

D. The Challenges of Building User-Mode Filesystems With FUSE

In both user-mode filesystems built with FUSE and filesystems built as kernel-mode components there are common components that must be considered and constructed. Each filesystem implementation has some concept of an in-memory structural representation of a file, folder, symbolic link, and other basic filesystem components. Each filesystem implementation must dedicate some of its code base to interacting with its storage medium. This code can be a kernel block cache, another filesystem, or a network library. Also, each filesystem implementation must dedicate some of its code base to fulfilling the contract required by a filesystem. In FUSE filesystems, there are about 30 methods that can be implemented. Some are required and some have reasonable default behaviors.

In our first research filesystem, OLFS [4], we found a large portion of our code base was dedicated to the glue code between our in-memory structures and the filesystem contract, and between our in-memory structures and our storage medium. In our latest implementation of OLFS, our caching layer was 1363 lines of code, our FUSE glue code was 2535 lines of code, our domain model was 1469 lines of code. Overall, 72.6% of the OLFS implementation was dedicated to implementing an efficient cache and implementing the FUSE filesystem contract. 27.4% of the OLFS implementation was dedicated to the actual domain model. While reflecting on this work, we noticed the high percentage of effort and code needed to work with the details of FUSE compared to our domain model.

Any project to construct a user-mode filesystem to expose a binary dataset in a human readable format will have to write a large amount of code fulfilling the filesystem contract. The work and understanding required to write this type of code can be a disincentive to invest time in a user-mode filesystem project.

E. Naked Objects

Naked Objects [5] is the architectural approach of using plain object-oriented domain models to build entire applications. In the realm of desktop applications, Naked Objects frameworks remove the concerns of providing user-interface code or persistence layers. These are left to the framework. An important aspect of Naked Objects frameworks is the object-oriented user interface. The object oriented user interface favors applications where the user is treated as a problem solver rather than a process follower. Where process is important, object oriented user interfaces aren't a good fit.

We discovered that the problems of the user interface and persistence layers in desktop and web applications is similar to the problem of the filesystem contract and backend storage in user-mode filesystems.

We believe that the filesystem is an excellent example of an object oriented user interface. In a filesystem, processes for copying, moving, reading, writing, or deleting files isn't exposed by the filesystem. These processes are managed externally by the operating system's other programs. The interaction with filesystems is noun-verb style of interaction and not a verb-noun interaction, which is more common with non-object oriented user interfaces. Like Naked Object user interfaces, filesystems "provide the user with a set of tools which to operate and does not dictate .. the users sequence of actions" [5].

F. Naked Object Filesystem: NOFS

After our experiences with OLFS, we felt that user-mode filesystems could benefit from another abstraction. To that end, we implemented the NOFS framework. The NOFS framework allows a developer to implement only the domain model and not be concerned with the details of persistence or the filesystem glue code. The NOFS framework manages fulfilling the filesystem contract required by FUSE or Dokan and provides a library for managing the serialization and deserialization of domain objects.

Files and folders are implemented using regular .NET classes. Folders are recognized as lists of other objects returned from public methods or classes that implement list interfaces and are marked with attributes provided by the NOFS framework. Files are implemented as regular .NET classes and marked with attributes provided by the NOFS framework. It is possible for an application filesystem implemented with the NOFS framework to be concerned with no details of file structure or filesystem metadata or to implement all of the details. By implementing additional interfaces and providing additional metadata, domain models can take the level of responsibility for the filesystem details that the developer cares to implement. Where these details are not implemented, the NOFS framework provides reasonable default implementations.

In our past work, we have been able to demonstrate how complete filesystems can be implemented with the NOFS framework with as few as two classes and less than 200 lines of code.

III. Considerations for Domain Modeling Storage Filesystems

A. Domain Modeling in Linux Filesystems

The basic data structure in UNIX or Linux filesystems is the inode. For regular files, inodes contain information about the size of a file, user and group ownership, the file's mode bits, create, last access, and modification timestamps, and pointers to blocks on disk. An important aspect of the inode is that a single inode does not contain all of the pointers for all of the blocks in a file except for the smallest of files.

In the ext2 filesystem, the inode structure has 15 block pointers [6]. The first 12 pointers are to the first 12 blocks of the file. Pointer 13 points to an indirect block, 14 points to a double indirect block, and 15 points to a triple indirect block. Having the first 12 pointers available in the inode allows for sequential reads to begin performing immediately while locating later blocks through the double indirect and triple indirect blocks. The structure of double and triple indirect blocks gives reasonable random access performance by guaranteeing that finding the location of any one block in a file will require at most two reads.

Having only a few block pointers present in the inode structure is necessary to manage the memory utilization of the ext2 filesystem. For example, to map the blocks of a 10GB file with 512 byte blocks would require 20,971,520 pointers. Using 64-bit pointers, this would require 160MB of memory to hold the pointers for this file. This memory requirement can be reduced if larger block sizes are used or if a filesystem implementation can make use of contiguous block ranges instead of addressing each individual block.

B. Memory Management and Domain Modeling User-Mode Filesystems

An advantage of kernel mode filesystems in Linux and other modern operating systems is that the decision of what data is kept in memory is managed by the virtual memory manager. Managing what data is kept in memory is also an important concern for user-mode filesystems. User mode filesystems have a performance challenge related to the number of context switches each filesystem call requires [7]. For example, a call to read a file will context switch from the application to the kernel, context switch from the kernel to the user-mode filesystem, the response to the kernel will require an additional context switch back to the kernel, and the kernel will have to context switch back to the application that originated the read request. In this example there were four context switches. If the user-mode filesystem needs an operating system service such as communicating over the network, reading from a disc or the use of some other system resource, the number of context switches will increase. A user-mode filesystem can reduce the number of total context switches by keeping more of its domain model in physical memory. If there are several requests for a single resource, if that resource is cached in memory by the user-mode filesystem, then responses will be faster.

The simplest approach in a user-mode filesystem is to keep the entire domain model in memory for the duration of the filesystem service. For simple filesystems that require a few hundred kilobytes or a few megabytes of memory at most, this approach makes sense.

For user-mode filesystems that manage larger datasets, an important concern is balancing the amount of data that is kept resident in physical memory. This complexity is in addition to the responsibility of developing code to conform to the filesystem contract and managing the underlying data storage for the filesystem.

C. The NOFS approach to User-Mode Filesystems That Manage Large Data Sets

To simplify the development of application filesystems that manage large data sets, the NOFS framework has added the concept of domain object identity and the weak reference pattern into its library. The weak reference pattern allows for object references to be addressable without being directly referenced so that the garbage collector can collect these objects [8] [9]. The introduction of these components allow for an application filesystem developed with the NOFS framework to have the details of when objects are loaded, persisted, and which objects remain in physical memory managed by

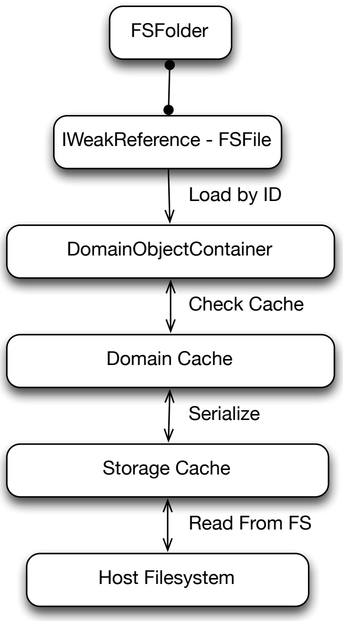

Fig. 1. Weak References with NOFS Cache

the NOFS framework and not by the implementation of the filesystem itself. By moving these responsibilities to the NOFS framework, the design complexity of user-mode filesystems can be reduced more closely to the complexity necessary in the simple case of loading the entire domain model into memory.

To enable this external memory management, the NOFS framework introduces three interfaces: IObjectWithID, IWeakReference, and IWeakReferenceList. In addition to these three interfaces, NOFS implements a caching component to reduce the number of times domain objects need to be loaded.

```
interface IObjectWithID {
  string Id { get; }
}

interface IWeakReference {
  IObjectWithID Get();
  string Id { get; }
  Type UnderlyingType { get; }
  void SetParent(IWeakReference parent);
}

interface IWeakReferenceList : IEnumerable
  int Count { get; }
  void Add(IWeakReference item);
  void Add(object item);
  void Remove(IWeakReference item);
  void Remove(object item);
}

interface IWeakReferenceList<T>
    : IEnumerable<IWeakReference>, WeakReferenceList
    where T : IObjectWithID
{
  void Add(T item);
  void Remove(T item);
  IEnumerable<T> GetAll();
}
```

For filesystems that want to make use of the weak reference pattern, NOFS requires that all domain objects that an IWeakReference can point to need to implement the IObjectWithID interface. This interface requires that the object return some unique identity for each unique domain instance. This identity helps establish which instances are the equivalent to other instances and acts as a pointer for the IWeakReference implementor to use to load the domain object when requested. NOFS doesn't make any guarantee that two subsequent calls to IWeakReference.Get() will return the same instance, so all comparisons must be based off of the identity value. The string type is used as the type for the identity rather than an integer or Guid to keep the requirements for the identity flexible. Aside from integers and Guids, it may be desirable to use URLs or other objects with string representations as the identity. The IWeakReferenceList and its sub-interface that adds methods with generic type constraints help the filesystem developer implement folders with the weak reference pattern. NOFS offers a default implementation of WeakReferenceList that can be used. NOFS also provides factories for creating IWeakReference instances given an identity value and provides the implementation for the Get() call.

In addition to the weak reference pattern which is important in determining which file and folder objects are kept in memory, there is an additional pattern that manages the data blocks of regular files. By default, NOFS regular file domain objects are translated to and from XML using the .NET serializer. If the domain object implements IProvidesUnstructuredData, then the file contents can be of a custom structure that is managed by the domain object. With this interface, the user can choose to either make use of an externally managed data source or one managed by NOFS. For the latter case, a new interface IDomainObjectRawDataStore provides methods for reading, writing, and truncating a binary file. With these two interfaces, it is possible for a NOFS filesystem to be unconcerned with the details of file reading or writing by implementing neither interface, or to be concerned with those details by implementing one or both interfaces.

```
interface IDomainObjectRawDataStore {
  long DataSize();
  int Read(byte[] buffer,long offset,long len);
  int Write(byte[] buffer,long offset,long len);
  void Truncate(long length);
}

interface IProvidesUnstructuredData
  : IDomainObjectRawDataStore {
  bool Cacheable();
}
```

With these two interfaces and the weak reference pattern, we were able to implement a simple storage based filesystem that uses the host's filesystem as the backing store with the NOFS framework in less than 300 lines of C# code.

The root of our reference implementation is expressed as the following class.

```
[RootFolder]
class FsRoot : FsFolder
{
```

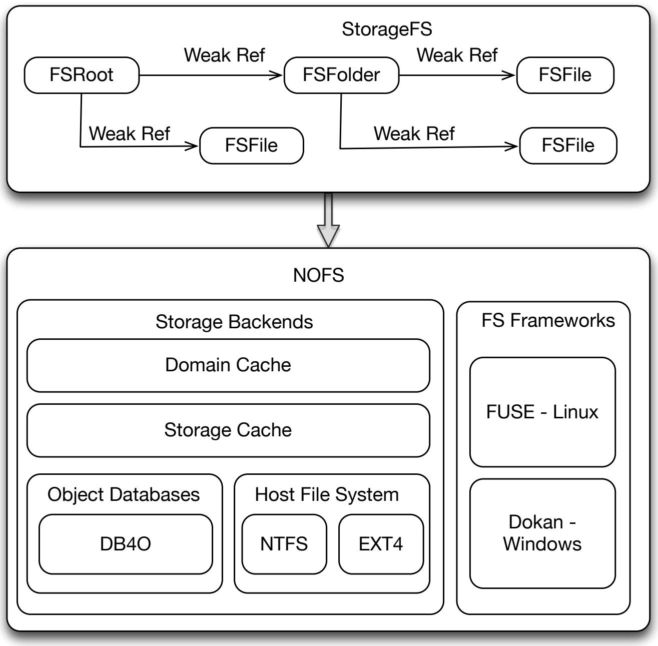

Fig. 2. Architecture of filesystem implementation as it relates to NOFS

```
  public FsRoot()
   : base("", Guid.NewGuid().ToString()){}
  public FsRoot(string name, string id)
   : base(name,id){}
}
```

This class is the instance from which the NOFS framework translates all paths. It represents the '/' part of any path passed to NOFS. The FsRoot class subclasses the FsFolder class.

```
[FolderObject]
class FsFolder : FsFolderOrFile,
   IWeakReferenceList<FsFolderOrFile>
{

}
```

The FsFolder class subclasses FsFolderOrFile and implements the IWeakReferenceList interface. When NOFS encounters instances of IEnumerable that have that attribute FolderObjectAttribute, it recognizes them as folders in the filesystem. Because FSFolder implements IWeakReference, all of the files and folders that are contained are weakly connected and do not need to be in physical memory for NOFS to load and examine the folder.

```
class FsFile : FsFolderOrFile,
   IProvidesUnstructuredData
{
  IDomainObjectRawDataStore _data;
  [NeedsRawDataStore]
  void SetDataStore(
    IDomainObjectRawDataStore data){
    _data = data;
  }

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

  bool Cacheable(){
    return false;
  }

  int Read(byte[] buff,long off,long len){
    return _data.Read(buff,off,len);
  }

  int Write(byte[] buff,long off,long len){
    return _data.Write(buff,off,len);
  }

  void Truncate(long length){
    _data.Truncate(length);
  }
}
```

FsFile is our class to represent regular files. This class implements IProvidesUnstructuredData. When NOFS sees a class implement the IProvidesUnstructuredData interface, it allows that class manage the Read, Write, and Truncate filesystem calls. FsFile also accepts an instance of IDomainObjectRawDataStore from NOFS after it is constructed. This helper interface provides access to the host filesystem file that contains the data for the file represented by this FsFile instance. Although in this example, the call is a simple pass through, it is possible to construct more complex implementations that translate the underlying data or provide other additional value.

```
[DomainObject]
class FsFolderOrFile : IObjectWithID
{
  private string _name;
  [NeedsContainer]
  IDomainObjectContainer Container {get;set;}
  string Id {get;set;}

  [ProvidesName]
  string Name {
    get { return _name; }
    set {
      _name = value;
      if (Container != null) {
        Container.ObjectChanged(this);
      }
    }
  }
}
```

FsFolderOrFile is the base class for folders and regular files in our reference implementation. This class is the type used in the IWeakReferenceList by the FsFolder class. This base class allows for both the regular files and folders to have a common type. This class is recognized as a regular file by NOFS because of the DomainObjectAttribute attribute. In the case of the FsFolder subclass, it is recognized as a folder because of the FolderObjectAttribute attribute. NOFS is able to determine that the Name property manages the file or folder name because of the ProvidesNameAttribute attribute. NOFS injects the IDomainObjectContainer in the Container property after the construction of a FsFile or FsFolder instance. The IDomainObjectContainer class manages serializing and deserializing of domain objects in the NOFS framework.

References

***Abstract*—Transparent access to remote data sets and data arising from web services is a non-trivial challenge to application**

developers. This early stage work addresses this challenge with NOFS, an object-oriented framework for creating filesystems to support domain specific functionality. While an early stage work, we present a solution to solve the access problem. Our solution greatly simplifies the task of filesystems development by providing the glue code needed between a domain model and the filesystem contract. We demonstrate support for domain models that are larger than physical memory and demonstrate how the concerns of caching can be removed from user-mode filesystem implementations. Future work will addresses more robust solutions to caching and other performance strategies.

IV. Introduction

In this paper, we discuss the need for user-mode filesystems in scientific computing to solve the problem of transparent data access. We explain how to simplify data access with user-mode filesystems. We explain how to simplify the task of building user-mode filesystems with our extensions to the NOFS framework.

Specifically, in this paper we discuss how existing storage-based user-mode filesystems must be designed with domain object size and physical memory usage in mind. We then point out why caching is important for these user-mode filesystems and illustrate how this concern contributes to a significant portion of the code in a user-mode filesystem. We show how the weak reference pattern can be leveraged in a filesystem framework and, specifically, how to apply it to an example file-and-folder domain model. Finally, we demonstrate how the naked objects architecture effectively supports storage-based user-mode filesystems and how a naked objects framework can manage concerns such as caching and domain object lifetime.

V. Related Work

A. The Importance of Inter-Process Communication Through the Filesystem

In modern operating systems, most methods of inter-process communication (IPC) can be represented through the filesystem. Among these are pipes, domain sockets, memory mapped files, and regular files. These methods of IPC allow for separate programs to communicate and coordinate with each other. This communication and coordination allows software programs to be broken down into separate and reusable components. Without the ability to communicate across process boundaries, many components would have to be present in the address spaces of many programs, mostly through shared libraries. With IPC, software programs and components can be composed and reused in several different ways.

B. The Role of Application Filesystems in Software Composition

Application filesystems further enhance and expand the traditional filesystem based methods of IPC by representing complex file structures, offering advanced filesystem semantics, or through representing or composing one or more external services through the filesystem contract. IPC through application filesystems allow local abstractions and local compositions to work without opening a network socket or needing to write code to comply with a network protocol.

Some filesystems promote composition by presenting a network protocol through a filesystem contract. An excellent example of this is Plan 9's filesystem service: 9P [1]. Through 9P, Plan 9 is able to abstract many network protocols and external resources. Among these resources are: HTTP and FTP protocols, managing network sockets, and a filesystem based abstraction for Plan 9's window manager.

It is possible to achieve inter-machine IPC through the use of network filesystems. In network filesystems that support file locking mechanisms and have adequate solutions to the cache coherency problem, it is possible to perform inter-machine IPC through filesystem operations.

In the past several years, many FUSE [2] based application filesystems have been built to act as clients for popular web services such as Flickr, IMAP email services, Amazon S3, and several others. In our own research, we leveraged our existing NOFS framework to implement RestFS [3], a dynamically reconfigurable filesystem for exposing remote restful resources as a local filesystem. With RestFS, we were able to demonstrate an architecture that could map several different restful web services such as Yahoo! Placefinder, Flickr, and Twitter into local filesystem representations. We were able to further demonstrate how these web services and local software components could be composed locally and re-exposed as restful web services.

C. The Role of User-Mode Filesystems in Software Composition for Large Datasets

In our exploration of user-mode filesystems development, we have worked with three categories of user-mode filesystems. First is the storage filesystem. Storage filesystems are primarily concerned with the traditional role of filesystems used as a means to store regular files and folders. A good example of a storage oriented user-mode filesystem would be NTFS-3G. NTFS-3G is a FUSE filesystem that allows UNIX-like operating systems to mount NTFS volumes in read-write mode. The second category is the connector filesystem. Connector filesystems provide mappings between a resource and a local filesystem. RestFS is an example of a connector filesystem. RestFS provides a way to create files and folders that can be configured to map filesystem calls to a remote restful web service. The third category is the application filesystem. Application filesystems provide behavior in addition to the resources that are represented by the filesystem.

To encourage reuse and dissemination of information, scientists often publish datasets using a format standard to their field and provide one or more libraries for popular programming languages to read from and write to these datasets. To create a new library for a new language, an entirely new library must be constructed for the new language, or where possible, bindings from the new language to an existing library in another language can be constructed. Where datasets are published in formats such as XML or CSV and where good documentation exists, the challenge of writing new libraries for scientific datasets is greatly lessened.

User-mode file systems of all types can play an important role with large datasets. Datasets in formats such as XML or CSV have several performance disadvantages due to their human readability. Among the sources of these issues are: greater amounts of whitespace characters, representation of numeric values as text instead of binary, and challenges determining seek offsets for random file access. So, in part due to performance and data size concerns, many datasets are published in a binary format. User-mode filesystems can help to bridge this divide. By constructing a user mode-files system on top of a binary formatted data set, it is possible to represent a filesystem as human readable files such as XML or CSV. With these types of files, it is simpler to implement software to consume them in other programming languages.

D. The Challenges of Building User-Mode Filesystems With FUSE

In both user-mode filesystems built with FUSE and filesystems built as kernel-mode components there are common components that must be considered and constructed. Each filesystem implementation has some concept of an in-memory structural representation of a file, folder, symbolic link, and other basic filesystem components. Each filesystem implementation must dedicate some of its code base to interacting with its storage medium. This code can be a kernel block cache, another filesystem, or a network library. Also, each filesystem implementation must dedicate some of its code base to fulfilling the contract required by a filesystem. In FUSE filesystems, there are about 30 methods that can be implemented. Some are required and some have reasonable default behaviors.

In our first research filesystem, OLFS [4], we found a large portion of our code base was dedicated to the glue code between our in-memory structures and the filesystem contract, and between our in-memory structures and our storage medium. In our latest implementation of OLFS, our caching layer was 1363 lines of code, our FUSE glue code was 2535 lines of code, our domain model was 1469 lines of code. Overall, 72.6% of the OLFS implementation was dedicated to implementing an efficient cache and implementing the FUSE filesystem contract. 27.4% of the OLFS implementation was dedicated to the actual domain model. While reflecting on this work, we noticed the high percentage of effort and code needed to work with the details of FUSE compared to our domain model.

Any project to construct a user-mode filesystem to expose a binary dataset in a human readable format will have to write a large amount of code fulfilling the filesystem contract. The work and understanding required to write this type of code can be a disincentive to invest time in a user-mode filesystem project.

E. Naked Objects

Naked Objects [5] is the architectural approach of using plain object-oriented domain models to build entire applications. In the realm of desktop applications, Naked Objects frameworks remove the concerns of providing user-interface code or persistence layers. These are left to the framework. An important aspect of Naked Objects frameworks is the object-oriented user interface. The object oriented user interface favors applications where the user is treated as a problem solver rather than a process follower. Where process is important, object oriented user interfaces aren't a good fit.

We discovered that the problems of the user interface and persistence layers in desktop and web applications is similar to the problem of the filesystem contract and backend storage in user-mode filesystems.

We believe that the filesystem is an excellent example of an object oriented user interface. In a filesystem, processes for copying, moving, reading, writing, or deleting files isn't exposed by the filesystem. These processes are managed externally by the operating system's other programs. The interaction with filesystems is noun-verb style of interaction and not a verb-noun interaction, which is more common with non-object oriented user interfaces. Like Naked Object user interfaces, filesystems "provide the user with a set of tools which to operate and does not dictate .. the users sequence of actions" [5].

F. Naked Object Filesystem: NOFS

After our experiences with OLFS, we felt that user-mode filesystems could benefit from another abstraction. To that end, we implemented the NOFS framework. The NOFS framework allows a developer to implement only the domain model and not be concerned with the details of persistence or the filesystem glue code. The NOFS framework manages fulfilling the filesystem contract required by FUSE or Dokan and provides a library for managing the serialization and deserialization of domain objects.

Files and folders are implemented using regular .NET classes. Folders are recognized as lists of other objects returned from public methods or classes that implement list interfaces and are marked with attributes provided by the NOFS framework. Files are implemented as regular .NET classes and marked with attributes provided by the NOFS framework. It is possible for an application filesystem implemented with the NOFS framework to be concerned with no details of file structure or filesystem metadata or to implement all of the details. By implementing additional interfaces and providing additional metadata, domain models can take the level of responsibility for the filesystem details that the developer cares to implement. Where these details are not implemented, the NOFS framework provides reasonable default implementations.

In our past work, we have been able to demonstrate how complete filesystems can be implemented with the NOFS framework with as few as two classes and less than 200 lines of code.

VI. Considerations for Domain Modeling Storage Filesystems

A. Domain Modeling in Linux Filesystems

The basic data structure in UNIX or Linux filesystems is the inode. For regular files, inodes contain information about the size of a file, user and group ownership, the file's mode bits, create, last access, and modification timestamps, and pointers to blocks on disk. An important aspect of the inode is that a single inode does not contain all of the pointers for all of the blocks in a file except for the smallest of files.

In the ext2 filesystem, the inode structure has 15 block pointers [6]. The first 12 pointers are to the first 12 blocks of the file. Pointer 13 points to an indirect block, 14 points to a double indirect block, and 15 points to a triple indirect block. Having the first 12 pointers available in the inode allows for sequential reads to begin performing immediately while locating later blocks through the double indirect and triple indirect blocks. The structure of double and triple indirect blocks gives reasonable random access performance by guaranteeing that finding the location of any one block in a file will require at most two reads.

Having only a few block pointers present in the inode structure is necessary to manage the memory utilization of the ext2 filesystem. For example, to map the blocks of a 10GB file with 512 byte blocks would require 20,971,520 pointers. Using 64-bit pointers, this would require 160MB of memory to hold the pointers for this file. This memory requirement can be reduced if larger block sizes are used or if a filesystem implementation can make use of contiguous block ranges instead of addressing each individual block.

B. Memory Management and Domain Modeling User-Mode Filesystems

An advantage of kernel mode filesystems in Linux and other modern operating systems is that the decision of what data is kept in memory is managed by the virtual memory manager. Managing what data is kept in memory is also an important concern for user-mode filesystems. User mode filesystems have a performance challenge related to the number of context switches each filesystem call requires [7]. For example, a call to read a file will context switch from the application to the kernel, context switch from the kernel to the user-mode filesystem, the response to the kernel will require an additional context switch back to the kernel, and the kernel will have to context switch back to the application that originated the read request. In this example there were four context switches. If the user-mode filesystem needs an operating system service such as communicating over the network, reading from a disc or the use of some other system resource, the number of context switches will increase. A user-mode filesystem can reduce the number of total context switches by keeping more of its domain model in physical memory. If there are several requests for a single resource, if that resource is cached in memory by the user-mode filesystem, then responses will be faster.

The simplest approach in a user-mode filesystem is to keep the entire domain model in memory for the duration of the filesystem service. For simple filesystems that require a few hundred kilobytes or a few megabytes of memory at most, this approach makes sense.

For user-mode filesystems that manage larger datasets, an important concern is balancing the amount of data that is kept resident in physical memory. This complexity is in addition to the responsibility of developing code to conform to the filesystem contract and managing the underlying data storage for the filesystem.

C. The NOFS approach to User-Mode Filesystems That Manage Large Data Sets

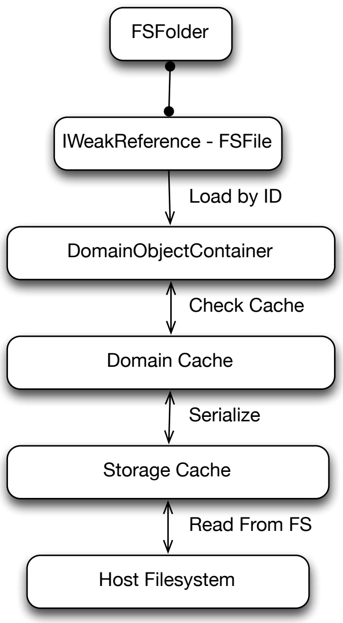

Fig. 3. Weak References with NOFS Cache

To simplify the development of application filesystems that manage large data sets, the NOFS framework has added the concept of domain object identity and the weak reference pattern into its library. The weak reference pattern allows for object references to be addressable without being directly referenced so that the garbage collector can collect these objects [8] [9]. The introduction of these components allow for an application filesystem developed with the NOFS framework to have the details of when objects are loaded, persisted, and which objects remain in physical memory managed by the NOFS framework and not by the implementation of the filesystem itself. By moving these responsibilities to the NOFS framework, the design complexity of user-mode filesystems can be reduced more closely to the complexity necessary in the simple case of loading the entire domain model into memory.

To enable this external memory management, the NOFS framework introduces three interfaces: IObjectWithID, IWeakReference, and IWeakReferenceList. In addition to these three interfaces, NOFS implements a caching component to reduce the number of times domain objects need to be loaded.

```
interface IObjectWithID {
  string Id { get; }
}

interface IWeakReference {
  IObjectWithID Get();
  string Id { get; }
  Type UnderlyingType { get; }
  void SetParent(IWeakReference parent);
}

interface IWeakReferenceList : IEnumerable
  int Count { get; }
  void Add(IWeakReference item);
  void Add(object item);
  void Remove(IWeakReference item);
  void Remove(object item);
}

interface IWeakReferenceList<T>
    : IEnumerable<IWeakReference>, WeakReferenceList
    where T : IObjectWithID
{
  void Add(T item);
  void Remove(T item);
  IEnumerable<T> GetAll();
}
```

For filesystems that want to make use of the weak reference pattern, NOFS requires that all domain objects that an IWeakReference can point to need to implement the IObjectWithID interface. This interface requires that the object return some unique identity for each unique domain instance. This identity helps establish which instances are the equivalent to other instances and acts as a pointer for the IWeakReference implementor to use to load the domain object when requested. NOFS doesn't make any guarantee that two subsequent calls to IWeakReference.Get() will return the same instance, so all comparisons must be based off of the identity value. The string type is used as the type for the identity rather than an integer or Guid to keep the requirements for the identity flexible. Aside from integers and Guids, it may be desirable to use URLs or other objects with string representations as the identity. The IWeakReferenceList and its sub-interface that adds methods with generic type constraints help the filesystem developer implement folders with the weak reference pattern. NOFS offers a default implementation of WeakReferenceList that can be used. NOFS also provides factories for creating IWeakReference instances given an identity value and provides the implementation for the Get() call.

In addition to the weak reference pattern which is important in determining which file and folder objects are kept in memory, there is an additional pattern that manages the data blocks of regular files. By default, NOFS regular file domain objects are translated to and from XML using the .NET serializer. If the domain object implements IProvidesUnstructuredData, then the file contents can be of a custom structure that is managed by the domain object. With this interface, the user can choose to either make use of an externally managed data source or one managed by NOFS. For the latter case, a new interface IDomainObjectRawDataStore provides methods for reading, writing, and truncating a binary file. With these two interfaces, it is possible for a NOFS filesystem to be unconcerned with the details of file reading or writing by implementing neither interface, or to be concerned with those details by implementing one or both interfaces.

```
interface IDomainObjectRawDataStore {
  long DataSize();
  int Read(byte[] buffer,long offset,long len);
  int Write(byte[] buffer,long offset,long len);
  void Truncate(long length);
}

interface IProvidesUnstructuredData
  : IDomainObjectRawDataStore {
  bool Cacheable();
}
```

With these two interfaces and the weak reference pattern, we were able to implement a simple storage based filesystem that uses the host's filesystem as the backing store with the NOFS framework in less than 300 lines of C# code.

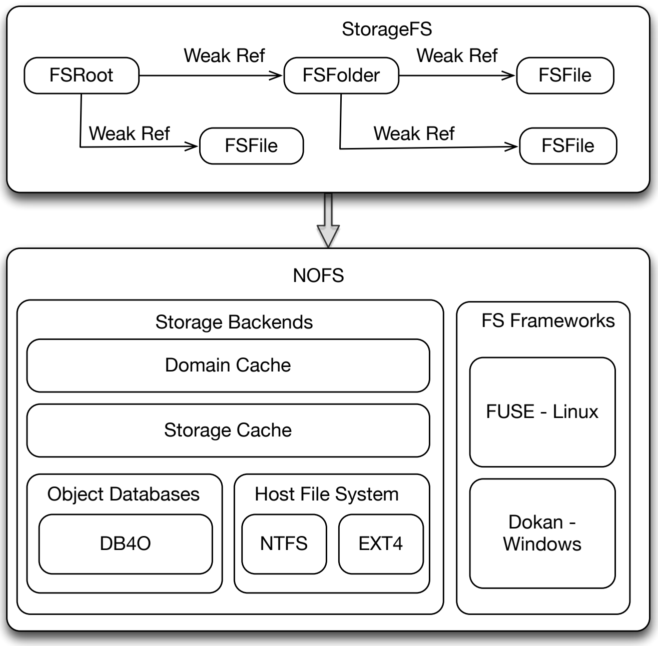

Fig. 4. Architecture of filesystem implementation as it relates to NOFS

The root of our reference implementation is expressed as the following class.

```
[RootFolder]
class FsRoot : FsFolder
{
  public FsRoot()
   : base("", Guid.NewGuid().ToString()){}
  public FsRoot(string name, string id)
   : base(name,id){}
}
```

This class is the instance from which the NOFS framework translates all paths. It represents the '/' part of any path passed to NOFS. The FsRoot class subclasses the FsFolder class.

```
[FolderObject]
class FsFolder : FsFolderOrFile,
   IWeakReferenceList<FsFolderOrFile>
{

}
```

The FsFolder class subclasses FsFolderOrFile and implements the IWeakReferenceList interface. When NOFS encounters instances of IEnumerable that have that attribute FolderObjectAttribute, it recognizes them as folders in the filesystem. Because FSFolder implements IWeakReference, all of the files and folders that are contained are weakly connected and do not need to be in physical memory for NOFS to load and examine the folder.

```
class FsFile : FsFolderOrFile,
   IProvidesUnstructuredData
{
  IDomainObjectRawDataStore _data;
  [NeedsRawDataStore]
  void SetDataStore(
    IDomainObjectRawDataStore data){
    _data = data;
  }

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

  bool Cacheable(){
    return false;
  }

  int Read(byte[] buff,long off,long len){
    return _data.Read(buff,off,len);
  }

  int Write(byte[] buff,long off,long len){
    return _data.Write(buff,off,len);
  }

  void Truncate(long length){
    _data.Truncate(length);
  }
}
```

FsFile is our class to represent regular files. This class implements IProvidesUnstructuredData. When NOFS sees a class implement the IProvidesUnstructuredData interface, it allows that class manage the Read, Write, and Truncate filesystem calls. FsFile also accepts an instance of IDomainObjectRawDataStore from NOFS after it is constructed. This helper interface provides access to the host filesystem file that contains the data for the file represented by this FsFile instance. Although in this example, the call is a simple pass through, it is possible to construct more complex implementations that translate the underlying data or provide other additional value.

```
[DomainObject]
class FsFolderOrFile : IObjectWithID
{
  private string _name;
  [NeedsContainer]
  IDomainObjectContainer Container {get;set;}
  string Id {get;set;}

  [ProvidesName]
  string Name {
    get { return _name; }
    set {
      _name = value;
      if (Container != null) {
        Container.ObjectChanged(this);
      }
    }
  }
}
```

FsFolderOrFile is the base class for folders and regular files in our reference implementation. This class is the type used in the IWeakReferenceList by the FsFolder class. This base class allows for both the regular files and folders to have a common type. This class is recognized as a regular file by NOFS because of the DomainObjectAttribute attribute. In the case of the FsFolder subclass, it is recognized as a folder because of the FolderObjectAttribute attribute. NOFS is able to determine that the Name property manages the file or folder name because of the ProvidesNameAttribute attribute. NOFS injects the IDomainObjectContainer in the Container property after the construction of a FsFile or FsFolder instance. The IDomainObjectContainer class manages serializing and deserializing of domain objects in the NOFS framework.

REFERENCES

***Abstract*—Transparent access to remote data sets and data arising from web services is a non-trivial challenge to application developers. This early stage work addresses this challenge with NOFS, an object-oriented framework for creating filesystems to support domain specific functionality. While an early stage work, we present a solution to solve the access problem. Our solution greatly simplifies the task of filesystems development by providing the glue code needed between a domain model and the filesystem contract. We demonstrate support for domain models that are larger than physical memory and demonstrate how the concerns of caching can be removed from user-mode filesystem implementations. Future work will addresses more robust solutions to caching and other performance strategies.**

VII. INTRODUCTION

In this paper, we discuss the need for user-mode filesystems in scientific computing to solve the problem of transparent data access. We explain how to simplify data access with user-mode filesystems. We explain how to simplify the task of building user-mode filesystems with our extensions to the NOFS framework.

Specifically, in this paper we discuss how existing storage-based user-mode filesystems must be designed with domain object size and physical memory usage in mind. We then point out why caching is important for these user-mode filesystems and illustrate how this concern contributes to a significant portion of the code in a user-mode filesystem. We show how the weak reference pattern can be leveraged in a filesystem framework and, specifically, how to apply it to an example file-and-folder domain model. Finally, we demonstrate how the naked objects architecture effectively supports storage-based user-mode filesystems and how a naked objects framework can manage concerns such as caching and domain object lifetime.

VIII. Related Work

A. The Importance of Inter-Process Communication Through the Filesystem

In modern operating systems, most methods of inter-process communication (IPC) can be represented through the filesystem. Among these are pipes, domain sockets, memory mapped files, and regular files. These methods of IPC allow for separate programs to communicate and coordinate with each other. This communication and coordination allows software programs to be broken down into separate and reusable components. Without the ability to communicate across process boundaries, many components would have to be present in the address spaces of many programs, mostly through shared libraries. With IPC, software programs and components can be composed and reused in several different ways.

B. The Role of Application Filesystems in Software Composition

Application filesystems further enhance and expand the traditional filesystem based methods of IPC by representing complex file structures, offering advanced filesystem semantics, or through representing or composing one or more external services through the filesystem contract. IPC through application filesystems allow local abstractions and local compositions to work without opening a network socket or needing to write code to comply with a network protocol.

Some filesystems promote composition by presenting a network protocol through a filesystem contract. An excellent example of this is Plan 9's filesystem service: 9P [1]. Through 9P, Plan 9 is able to abstract many network protocols and external resources. Among these resources are: HTTP and FTP protocols, managing network sockets, and a filesystem based abstraction for Plan 9's window manager.

It is possible to achieve inter-machine IPC through the use of network filesystems. In network filesystems that support file locking mechanisms and have adequate solutions to the cache coherency problem, it is possible to perform inter-machine IPC through filesystem operations.

In the past several years, many FUSE [2] based application filesystems have been built to act as clients for popular web services such as Flickr, IMAP email services, Amazon S3, and several others. In our own research, we leveraged our existing NOFS framework to implement RestFS [3], a dynamically reconfigurable filesystem for exposing remote restful resources as a local filesystem. With RestFS, we were able to demonstrate an architecture that could map several different restful web services such as Yahoo! Placefinder, Flickr, and Twitter into local filesystem representations. We were able to further demonstrate how these web services and local software components could be composed locally and re-exposed as restful web services.

C. The Role of User-Mode Filesystems in Software Composition for Large Datasets

In our exploration of user-mode filesystems development, we have worked with three categories of user-mode filesystems. First is the storage filesystem. Storage filesystems are primarily concerned with the traditional role of filesystems used as a means to store regular files and folders. A good example of a storage oriented user-mode filesystem would be NTFS-3G. NTFS-3G is a FUSE filesystem that allows UNIX-like operating systems to mount NTFS volumes in read-write mode. The second category is the connector filesystem. Connector filesystems provide mappings between a resource and a local filesystem. RestFS is an example of a connector filesystem. RestFS provides a way to create files and folders that can be configured to map filesystem calls to a remote restful web service. The third category is the application filesystem. Application filesystems provide behavior in addition to the resources that are represented by the filesystem.

To encourage reuse and dissemination of information, scientists often publish datasets using a format standard to their field and provide one or more libraries for popular programming languages to read from and write to these datasets. To create a new library for a new language, an entirely new library must be constructed for the new language, or where possible, bindings from the new language to an existing library in another language can be constructed. Where datasets are published in formats such as XML or CSV and where good documentation exists, the challenge of writing new libraries for scientific datasets is greatly lessened.

User-mode file systems of all types can play an important role with large datasets. Datasets in formats such as XML or CSV have several performance disadvantages due to their human readability. Among the sources of these issues are: greater amounts of whitespace characters, representation of numeric values as text instead of binary, and challenges determining seek offsets for random file access. So, in part due to performance and data size concerns, many datasets are published in a binary format. User-mode filesystems can help to bridge this divide. By constructing a user mode-files system on top of a binary formatted data set, it is possible to represent a filesystem as human readable files such as XML or CSV. With these types of files, it is simpler to implement software to consume them in other programming languages.

D. The Challenges of Building User-Mode Filesystems With FUSE

In both user-mode filesystems built with FUSE and filesystems built as kernel-mode components there are common components that must be considered and constructed. Each filesystem implementation has some concept of an in-memory structural representation of a file, folder, symbolic link, and other basic filesystem components. Each filesystem implementation must dedicate some of its code base to interacting with its storage medium. This code can be a kernel block cache, another filesystem, or a network library. Also, each filesystem implementation must dedicate some of its code base to fulfilling the contract required by a filesystem. In FUSE filesystems, there are about 30 methods that can be implemented. Some are required and some have reasonable default behaviors.

In our first research filesystem, OLFS [4], we found a large portion of our code base was dedicated to the glue code between our in-memory structures and the filesystem contract, and between our in-memory structures and our storage medium. In our latest implementation of OLFS, our caching layer was 1363 lines of code, our FUSE glue code was 2535 lines of code, our domain model was 1469 lines of code. Overall, 72.6% of the OLFS implementation was dedicated to implementing an efficient cache and implementing the FUSE filesystem contract. 27.4% of the OLFS implementation was dedicated to the actual domain model. While reflecting on this work, we noticed the high percentage of effort and code needed to work with the details of FUSE compared to our domain model.

Any project to construct a user-mode filesystem to expose a binary dataset in a human readable format will have to write a large amount of code fulfilling the filesystem contract. The work and understanding required to write this type of code can be a disincentive to invest time in a user-mode filesystem project.

E. Naked Objects

Naked Objects [5] is the architectural approach of using plain object-oriented domain models to build entire applications. In the realm of desktop applications, Naked Objects frameworks remove the concerns of providing user-interface code or persistence layers. These are left to the framework. An important aspect of Naked Objects frameworks is the object-oriented user interface. The object oriented user interface favors applications where the user is treated as a problem solver rather than a process follower. Where process is important, object oriented user interfaces aren't a good fit.

We discovered that the problems of the user interface and persistence layers in desktop and web applications is similar to the problem of the filesystem contract and backend storage in user-mode filesystems.

We believe that the filesystem is an excellent example of an object oriented user interface. In a filesystem, processes for copying, moving, reading, writing, or deleting files isn't exposed by the filesystem. These processes are managed externally by the operating system's other programs. The interaction with filesystems is noun-verb style of interaction and not a verb-noun interaction, which is more common with non-object oriented user interfaces. Like Naked Object user interfaces, filesystems "provide the user with a set of tools which to operate and does not dictate .. the users sequence of actions" [5].

F. Naked Object Filesystem: NOFS

After our experiences with OLFS, we felt that user-mode filesystems could benefit from another abstraction. To that end, we implemented the NOFS framework. The NOFS framework allows a developer to implement only the domain model and not be concerned with the details of persistence or the filesystem glue code. The NOFS framework manages fulfilling the filesystem contract required by FUSE or Dokan and provides a library for managing the serialization and deserialization of domain objects.

Files and folders are implemented using regular .NET classes. Folders are recognized as lists of other objects returned from public methods or classes that implement list interfaces and are marked with attributes provided by the NOFS framework. Files are implemented as regular .NET classes and marked with attributes provided by the NOFS framework. It is possible for an application filesystem implemented with the NOFS framework to be concerned with no details of file structure or filesystem metadata or to implement all of the details. By implementing additional interfaces and providing additional metadata, domain models can take the level of responsibility for the filesystem details that the developer cares to implement. Where these details are not implemented, the NOFS framework provides reasonable default implementations.

In our past work, we have been able to demonstrate how complete filesystems can be implemented with the NOFS framework with as few as two classes and less than 200 lines of code.

IX. Considerations for Domain Modeling Storage Filesystems

A. Domain Modeling in Linux Filesystems

The basic data structure in UNIX or Linux filesystems is the inode. For regular files, inodes contain information about the size of a file, user and group ownership, the file's mode bits, create, last access, and modification timestamps, and pointers to blocks on disk. An important aspect of the inode is that a single inode does not contain all of the pointers for all of the blocks in a file except for the smallest of files.

In the ext2 filesystem, the inode structure has 15 block pointers [6]. The first 12 pointers are to the first 12 blocks of the file. Pointer 13 points to an indirect block, 14 points to a double indirect block, and 15 points to a triple indirect block. Having the first 12 pointers available in the inode allows for sequential reads to begin performing immediately while locating later blocks through the double indirect and triple indirect blocks. The structure of double and triple indirect blocks gives reasonable random access performance by guaranteeing that finding the location of any one block in a file will require at most two reads.

Having only a few block pointers present in the inode structure is necessary to manage the memory utilization of the ext2 filesystem. For example, to map the blocks of a 10GB file with 512 byte blocks would require 20,971,520 pointers. Using 64-bit pointers, this would require 160MB of memory to hold the pointers for this file. This memory requirement can be reduced if larger block sizes are used or if a filesystem implementation can make use of contiguous block ranges instead of addressing each individual block.

B. Memory Management and Domain Modeling User-Mode Filesystems

An advantage of kernel mode filesystems in Linux and other modern operating systems is that the decision of what data is kept in memory is managed by the virtual memory manager. Managing what data is kept in memory is also an important concern for user-mode filesystems. User mode filesystems have a performance challenge related to the number of context switches each filesystem call requires [7]. For example, a call to read a file will context switch from the application to the kernel, context switch from the kernel to the user-mode filesystem, the response to the kernel will require an additional context switch back to the kernel, and the kernel will have to context switch back to the application that originated the read request. In this example there were four context switches. If the user-mode filesystem needs an operating system service such as communicating over the network, reading from a disc or the use of some other system resource, the number of context switches will increase. A user-mode filesystem can reduce the number of total context switches by keeping more of its domain model in physical memory. If there are several requests for a single resource, if that resource is cached in memory by the user-mode filesystem, then responses will be faster.

The simplest approach in a user-mode filesystem is to keep the entire domain model in memory for the duration of the filesystem service. For simple filesystems that require a few hundred kilobytes or a few megabytes of memory at most, this approach makes sense.

For user-mode filesystems that manage larger datasets, an important concern is balancing the amount of data that is kept resident in physical memory. This complexity is in addition to the responsibility of developing code to conform to the filesystem contract and managing the underlying data storage for the filesystem.

C. The NOFS approach to User-Mode Filesystems That Manage Large Data Sets

To simplify the development of application filesystems that manage large data sets, the NOFS framework has added the concept of domain object identity and the weak reference pattern into its library. The weak reference pattern allows for object references to be addressable without being directly referenced so that the garbage collector can collect these objects [8] [9]. The introduction of these components allow for an application filesystem developed with the NOFS framework to have the details of when objects are loaded, persisted, and which objects remain in physical memory managed by the NOFS framework and not by the implementation of the filesystem itself. By moving these responsibilities to the NOFS framework, the design complexity of user-mode filesystems can be reduced more closely to the complexity necessary in the simple case of loading the entire domain model into memory.

To enable this external memory management, the NOFS framework introduces three interfaces: IObjectWithID, IWeakReference, and IWeakReferenceList. In addition to these three interfaces, NOFS implements a caching component to reduce the number of times domain objects need to be loaded.

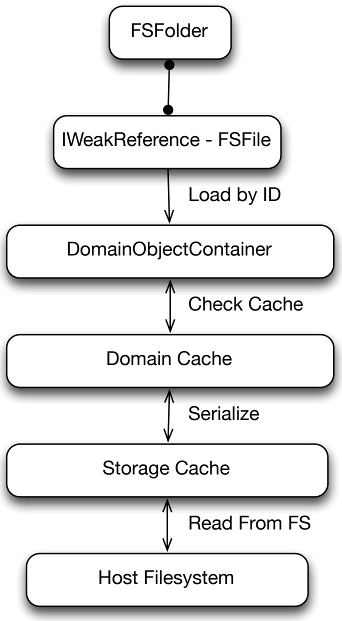

Fig. 5. Weak References with NOFS Cache

```
interface IObjectWithID {
  string Id { get; }
}

interface IWeakReference {
  IObjectWithID Get();
  string Id { get; }
  Type UnderlyingType { get; }
  void SetParent(IWeakReference parent);
}

interface IWeakReferenceList : IEnumerable
  int Count { get; }
  void Add(IWeakReference item);
  void Add(object item);
  void Remove(IWeakReference item);
  void Remove(object item);
}

interface IWeakReferenceList<T>
    : IEnumerable<IWeakReference>, WeakReferenceList
    where T : IObjectWithID
{
  void Add(T item);
  void Remove(T item);
  IEnumerable<T> GetAll();
}
```

For filesystems that want to make use of the weak reference pattern, NOFS requires that all domain objects that an IWeakReference can point to need to implement the IObjectWithID interface. This interface requires that the object return some unique identity for each unique domain instance. This identity helps establish which instances are the equivalent to other instances and acts as a pointer for the IWeakReference implementor to use to load the domain object when requested. NOFS doesn't make any guarantee that two subsequent calls to IWeakReference.Get() will return the same instance, so all comparisons must be based off of the identity value. The string type is used as the type for the identity rather than an integer or Guid to keep the requirements for the identity flexible. Aside from integers and Guids, it may be desirable to use URLs or other objects with string representations as the identity. The IWeakReferenceList and its sub-interface that adds methods with generic type constraints help the filesystem developer implement folders with the weak reference pattern. NOFS offers a default implementation of WeakReferenceList that can be used. NOFS also provides factories for creating IWeakReference instances given an identity value and provides the implementation for the Get() call.

In addition to the weak reference pattern which is important in determining which file and folder objects are kept in memory, there is an additional pattern that manages the data blocks of regular files. By default, NOFS regular file domain objects are translated to and from XML using the .NET serializer. If the domain object implements IProvidesUnstructuredData, then the file contents can be of a custom structure that is managed by the domain object. With this interface, the user can choose to either make use of an externally managed data source or one managed by NOFS. For the latter case, a new interface IDomainObjectRawDataStore provides methods for reading, writing, and truncating a binary file. With these two interfaces, it is possible for a NOFS filesystem to be unconcerned with the details of file reading or writing by implementing neither interface, or to be concerned with those details by implementing one or both interfaces.

```
interface IDomainObjectRawDataStore {
  long DataSize();
  int Read(byte[] buffer,long offset,long len);
  int Write(byte[] buffer,long offset,long len);
  void Truncate(long length);
}

interface IProvidesUnstructuredData
  : IDomainObjectRawDataStore {
  bool Cacheable();
}
```

With these two interfaces and the weak reference pattern, we were able to implement a simple storage based filesystem that uses the host's filesystem as the backing store with the NOFS framework in less than 300 lines of C# code.

The root of our reference implementation is expressed as the following class.

```
[RootFolder]
class FsRoot : FsFolder
{
  public FsRoot()
   : base("", Guid.NewGuid().ToString()){}
  public FsRoot(string name, string id)
   : base(name,id){}
}
```

This class is the instance from which the NOFS framework translates all paths. It represents the '/' part of any path passed to NOFS. The FsRoot class subclasses the FsFolder class.

```
[FolderObject]
```

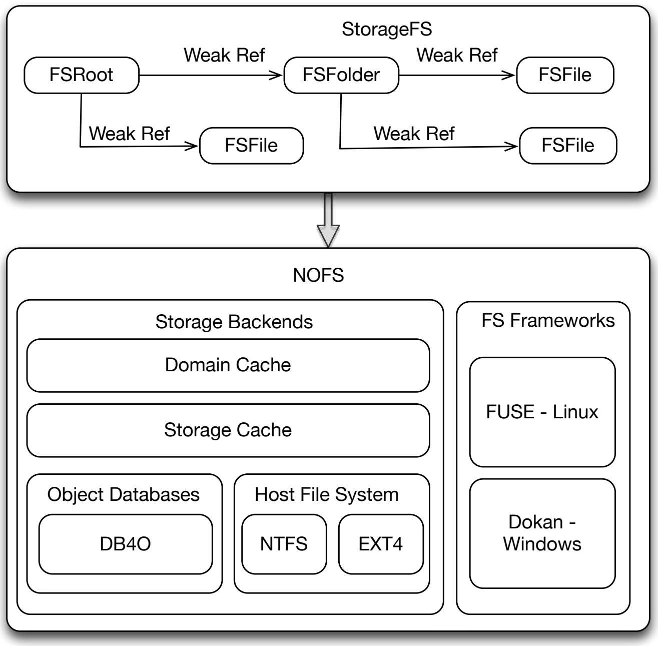

Fig. 6. Architecture of filesystem implementation as it relates to NOFS

```
class FsFolder : FsFolderOrFile,
   IWeakReferenceList<FsFolderOrFile>
{

}
```

The FsFolder class subclasses FsFolderOrFile and implements the IWeakReferenceList interface. When NOFS encounters instances of IEnumerable that have that attribute FolderObjectAttribute, it recognizes them as folders in the filesystem. Because FSFolder implements IWeakReference, all of the files and folders that are contained are weakly connected and do not need to be in physical memory for NOFS to load and examine the folder.

```
class FsFile : FsFolderOrFile,
   IProvidesUnstructuredData
{
  IDomainObjectRawDataStore _data;
  [NeedsRawDataStore]
  void SetDataStore(
    IDomainObjectRawDataStore data){
    _data = data;
  }

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

  bool Cacheable(){
    return false;
  }

  int Read(byte[] buff,long off,long len){
    return _data.Read(buff,off,len);
  }

  int Write(byte[] buff,long off,long len){
    return _data.Write(buff,off,len);
  }
  void Truncate(long length){
    _data.Truncate(length);
  }
}
```

FsFile is our class to represent regular files. This class implements IProvidesUnstructuredData. When NOFS sees a class implement the IProvidesUnstructuredData interface, it allows that class manage the Read, Write, and Truncate filesystem calls. FsFile also accepts an instance of IDomainObjectRawDataStore from NOFS after it is constructed. This helper interface provides access to the host filesystem file that contains the data for the file represented by this FsFile instance. Although in this example, the call is a simple pass through, it is possible to construct more complex implementations that translate the underlying data or provide other additional value.

```
[DomainObject]
class FsFolderOrFile : IObjectWithID
{
  private string _name;
  [NeedsContainer]
  IDomainObjectContainer Container {get;set;}
  string Id {get;set;}

  [ProvidesName]
  string Name {
    get { return _name; }
    set {
      _name = value;
      if (Container != null) {
        Container.ObjectChanged(this);
      }
    }
  }
}
```

FsFolderOrFile is the base class for folders and regular files in our reference implementation. This class is the type used in the IWeakReferenceList by the FsFolder class. This base class allows for both the regular files and folders to have a common type. This class is recognized as a regular file by NOFS because of the DomainObjectAttribute attribute. In the case of the FsFolder subclass, it is recognized as a folder because of the FolderObjectAttribute attribute. NOFS is able to determine that the Name property manages the file or folder name because of the ProvidesNameAttribute attribute. NOFS injects the IDomainObjectContainer in the Container property after the construction of a FsFile or FsFolder instance. The IDomainObjectContainer class manages serializing and deserializing of domain objects in the NOFS framework.

REFERENCES

***Abstract*—Transparent access to remote data sets and data arising from web services is a non-trivial challenge to application developers. This early stage work addresses this challenge with NOFS, an object-oriented framework for creating filesystems to support domain specific functionality. While an early stage work, we present a solution to solve the access problem. Our solution greatly simplifies the task of filesystems development by providing the glue code needed between a domain model and the filesystem contract. We demonstrate support for domain models that are larger than physical memory and demonstrate how the concerns of caching can be removed from user-mode filesystem implementations. Future work will addresses more robust solutions to caching and other performance strategies.**

X. INTRODUCTION

In this paper, we discuss the need for user-mode filesystems in scientific computing to solve the problem of transparent data access. We explain how to simplify data access with user-mode filesystems. We explain how to simplify the task of building user-mode filesystems with our extensions to the NOFS framework.

Specifically, in this paper we discuss how existing storage-based user-mode filesystems must be designed with domain object size and physical memory usage in mind. We then point out why caching is important for these user-mode filesystems and illustrate how this concern contributes to a significant portion of the code in a user-mode filesystem. We show how the weak reference pattern can be leveraged in a filesystem framework and, specifically, how to apply it to an example file-and-folder domain model. Finally, we demonstrate how the naked objects architecture effectively supports storage-based user-mode filesystems and how a naked objects framework can manage concerns such as caching and domain object lifetime.

XI. RELATED WORK

A. The Importance of Inter-Process Communication Through the Filesystem

In modern operating systems, most methods of inter-process communication (IPC) can be represented through the filesystem. Among these are pipes, domain sockets, memory mapped files, and regular files. These methods of IPC allow for separate programs to communicate and coordinate with each other. This communication and coordination allows software programs to be broken down into separate and reusable components. Without the ability to communicate across process boundaries, many components would have to be present in the address spaces of many programs, mostly through shared libraries. With IPC, software programs and components can be composed and reused in several different ways.

B. The Role of Application Filesystems in Software Composition

Application filesystems further enhance and expand the traditional filesystem based methods of IPC by representing complex file structures, offering advanced filesystem semantics, or through representing or composing one or more external services through the filesystem contract. IPC through application filesystems allow local abstractions and local compositions to work without opening a network socket or needing to write code to comply with a network protocol.

Some filesystems promote composition by presenting a network protocol through a filesystem contract. An excellent example of this is Plan 9's filesystem service: 9P [1]. Through 9P, Plan 9 is able to abstract many network protocols and external resources. Among these resources are: HTTP and FTP protocols, managing network sockets, and a filesystem based abstraction for Plan 9's window manager.

It is possible to achieve inter-machine IPC through the use of network filesystems. In network filesystems that support file locking mechanisms and have adequate solutions to the cache coherency problem, it is possible to perform inter-machine IPC through filesystem operations.

In the past several years, many FUSE [2] based application filesystems have been built to act as clients for popular web services such as Flickr, IMAP email services, Amazon S3, and several others. In our own research, we leveraged our existing NOFS framework to implement RestFS [3], a dynamically reconfigurable filesystem for exposing remote restful resources as a local filesystem. With RestFS, we were able to demonstrate an architecture that could map several different restful web services such as Yahoo! Placefinder, Flickr, and Twitter into local filesystem representations. We were able to further demonstrate how these web services and local software components could be composed locally and re-exposed as restful web services.

C. The Role of User-Mode Filesystems in Software Composition for Large Datasets

In our exploration of user-mode filesystems development, we have worked with three categories of user-mode filesystems. First is the storage filesystem. Storage filesystems are primarily concerned with the traditional role of filesystems used as a means to store regular files and folders. A good example of a storage oriented user-mode filesystem would be NTFS-3G. NTFS-3G is a FUSE filesystem that allows UNIX-like operating systems to mount NTFS volumes in read-write mode. The second category is the connector filesystem. Connector filesystems provide mappings between a resource and a local filesystem. RestFS is an example of a connector filesystem. RestFS provides a way to create files and folders that can be configured to map filesystem calls to a remote restful web service. The third category is the application filesystem. Application filesystems provide behavior in addition to the resources that are represented by the filesystem.

To encourage reuse and dissemination of information, scientists often publish datasets using a format standard to their field and provide one or more libraries for popular programming languages to read from and write to these datasets. To create a new library for a new language, an entirely new library must be constructed for the new language, or where possible, bindings from the new language to an existing library in another language can be constructed. Where datasets are published in formats such as XML or CSV and where good documentation exists, the challenge of writing new libraries for scientific datasets is greatly lessened.

User-mode file systems of all types can play an important role with large datasets. Datasets in formats such as XML or CSV have several performance disadvantages due to their human readability. Among the sources of these issues are: greater amounts of whitespace characters, representation of numeric values as text instead of binary, and challenges determining seek offsets for random file access. So, in part due to performance and data size concerns, many datasets are published in a binary format. User-mode filesystems can help to bridge this divide. By constructing a user mode-files system on top of a binary formatted data set, it is possible to represent a filesystem as human readable files such as XML or CSV. With these types of files, it is simpler to implement software to consume them in other programming languages.

D. The Challenges of Building User-Mode Filesystems With FUSE

In both user-mode filesystems built with FUSE and filesystems built as kernel-mode components there are common components that must be considered and constructed. Each filesystem implementation has some concept of an in-memory structural representation of a file, folder, symbolic link, and other basic filesystem components. Each filesystem implementation must dedicate some of its code base to interacting with its storage medium. This code can be a kernel block cache, another filesystem, or a network library. Also, each filesystem implementation must dedicate some of its code base to fulfilling the contract required by a filesystem. In FUSE filesystems, there are about 30 methods that can be implemented. Some are required and some have reasonable default behaviors.

In our first research filesystem, OLFS [4], we found a large portion of our code base was dedicated to the glue code between our in-memory structures and the filesystem contract, and between our in-memory structures and our storage medium. In our latest implementation of OLFS, our caching layer was 1363 lines of code, our FUSE glue code was 2535 lines of code, our domain model was 1469 lines of code. Overall, 72.6% of the OLFS implementation was dedicated to implementing an efficient cache and implementing the FUSE filesystem contract. 27.4% of the OLFS implementation was dedicated to the actual domain model. While reflecting on this work, we noticed the high percentage of effort and code needed to work with the details of FUSE compared to our domain model.

Any project to construct a user-mode filesystem to expose a binary dataset in a human readable format will have to write a large amount of code fulfilling the filesystem contract. The work and understanding required to write this type of code can be a disincentive to invest time in a user-mode filesystem project.

E. Naked Objects

Naked Objects [5] is the architectural approach of using plain object-oriented domain models to build entire applications. In the realm of desktop applications, Naked Objects frameworks remove the concerns of providing user-interface code or persistence layers. These are left to the framework. An important aspect of Naked Objects frameworks is the object-oriented user interface. The object oriented user interface favors applications where the user is treated as a problem solver rather than a process follower. Where process is important, object oriented user interfaces aren't a good fit.

We discovered that the problems of the user interface and persistence layers in desktop and web applications is similar to the problem of the filesystem contract and backend storage in user-mode filesystems.

We believe that the filesystem is an excellent example of an object oriented user interface. In a filesystem, processes for copying, moving, reading, writing, or deleting files isn't exposed by the filesystem. These processes are managed externally by the operating system's other programs. The interaction with filesystems is noun-verb style of interaction and not a verb-noun interaction, which is more common with non-object oriented user interfaces. Like Naked Object user interfaces, filesystems "provide the user with a set of tools which to operate and does not dictate .. the users sequence of actions" [5].

F. Naked Object Filesystem: NOFS

After our experiences with OLFS, we felt that user-mode filesystems could benefit from another abstraction. To that end, we implemented the NOFS framework. The NOFS framework allows a developer to implement only the domain model and not be concerned with the details of persistence or the filesystem glue code. The NOFS framework manages fulfilling the filesystem contract required by FUSE or Dokan and provides a library for managing the serialization and deserialization of domain objects.

Files and folders are implemented using regular .NET classes. Folders are recognized as lists of other objects returned from public methods or classes that implement list interfaces and are marked with attributes provided by the NOFS framework. Files are implemented as regular .NET classes and marked with attributes provided by the NOFS framework. It is possible for an application filesystem implemented with the NOFS framework to be concerned with no details of file structure or filesystem metadata or to implement all of the details. By implementing additional interfaces and providing additional metadata, domain models can take the level of responsibility for the filesystem details that the developer cares to implement. Where these details are not implemented, the NOFS framework provides reasonable default implementations.

In our past work, we have been able to demonstrate how complete filesystems can be implemented with the NOFS framework with as few as two classes and less than 200 lines of code.

XII. Considerations for Domain Modeling Storage Filesystems

A. Domain Modeling in Linux Filesystems

The basic data structure in UNIX or Linux filesystems is the inode. For regular files, inodes contain information about the size of a file, user and group ownership, the file's mode bits, create, last access, and modification timestamps, and pointers to blocks on disk. An important aspect of the inode is that a single inode does not contain all of the pointers for all of the blocks in a file except for the smallest of files.

In the ext2 filesystem, the inode structure has 15 block pointers [6]. The first 12 pointers are to the first 12 blocks of the file. Pointer 13 points to an indirect block, 14 points to a double indirect block, and 15 points to a triple indirect block. Having the first 12 pointers available in the inode allows for sequential reads to begin performing immediately while locating later blocks through the double indirect and triple indirect blocks. The structure of double and triple indirect blocks gives reasonable random access performance by guaranteeing that finding the location of any one block in a file will require at most two reads.

Having only a few block pointers present in the inode structure is necessary to manage the memory utilization of the ext2 filesystem. For example, to map the blocks of a 10GB file with 512 byte blocks would require 20,971,520 pointers. Using 64-bit pointers, this would require 160MB of memory to hold the pointers for this file. This memory requirement can be reduced if larger block sizes are used or if a filesystem implementation can make use of contiguous block ranges instead of addressing each individual block.

B. Memory Management and Domain Modeling User-Mode Filesystems

An advantage of kernel mode filesystems in Linux and other modern operating systems is that the decision of what data is kept in memory is managed by the virtual memory manager. Managing what data is kept in memory is also an important concern for user-mode filesystems. User mode filesystems have a performance challenge related to the number of context switches each filesystem call requires [7]. For example, a call to read a file will context switch from the application to the kernel, context switch from the kernel to the user-mode filesystem, the response to the kernel will require an additional context switch back to the kernel, and the kernel will have to context switch back to the application that originated the read request. In this example there were four context switches. If the user-mode filesystem needs an operating system service such as communicating over the network, reading from a disc or the use of some other system resource, the number of context switches will increase. A user-mode filesystem can reduce the number of total context switches by keeping more of its domain model in physical memory. If there are several requests for a single resource, if that resource is cached in memory by the user-mode filesystem, then responses will be faster.

The simplest approach in a user-mode filesystem is to keep the entire domain model in memory for the duration of the filesystem service. For simple filesystems that require a few hundred kilobytes or a few megabytes of memory at most, this approach makes sense.

For user-mode filesystems that manage larger datasets, an important concern is balancing the amount of data that is kept resident in physical memory. This complexity is in addition to the responsibility of developing code to conform to the filesystem contract and managing the underlying data storage for the filesystem.

C. The NOFS approach to User-Mode Filesystems That Manage Large Data Sets

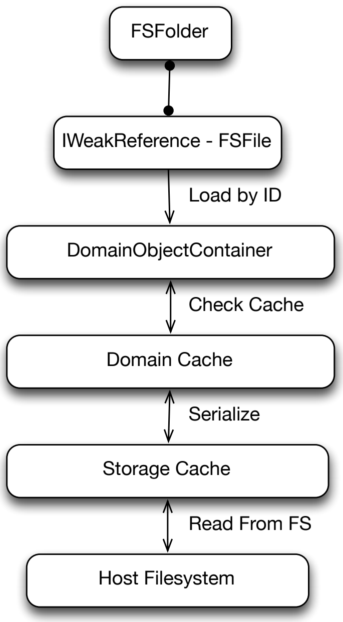

Fig. 7. Weak References with NOFS Cache

To simplify the development of application filesystems that manage large data sets, the NOFS framework has added the concept of domain object identity and the weak reference pattern into its library. The weak reference pattern allows for object references to be addressable without being directly referenced so that the garbage collector can collect these objects [8] [9]. The introduction of these components allow for an application filesystem developed with the NOFS framework to have the details of when objects are loaded, persisted, and which objects remain in physical memory managed by the NOFS framework and not by the implementation of the filesystem itself. By moving these responsibilities to the NOFS framework, the design complexity of user-mode filesystems can be reduced more closely to the complexity necessary in the simple case of loading the entire domain model into memory.

To enable this external memory management, the NOFS framework introduces three interfaces: IObjectWithID, IWeakReference, and IWeakReferenceList. In addition to these three interfaces, NOFS implements a caching component to reduce the number of times domain objects need to be loaded.

```
interface IObjectWithID {
  string Id { get; }
}

interface IWeakReference {
  IObjectWithID Get();
  string Id { get; }
  Type UnderlyingType { get; }
  void SetParent(IWeakReference parent);
}

interface IWeakReferenceList : IEnumerable
  int Count { get; }
  void Add(IWeakReference item);
  void Add(object item);
  void Remove(IWeakReference item);
  void Remove(object item);
}

interface IWeakReferenceList<T>
    : IEnumerable<IWeakReference>, WeakReferenceList
    where T : IObjectWithID
{
  void Add(T item);
  void Remove(T item);
  IEnumerable<T> GetAll();
}
```

For filesystems that want to make use of the weak reference pattern, NOFS requires that all domain objects that an IWeakReference can point to need to implement the IObjectWithID interface. This interface requires that the object return some unique identity for each unique domain instance. This identity helps establish which instances are the equivalent to other instances and acts as a pointer for the IWeakReference implementor to use to load the domain object when requested. NOFS doesn't make any guarantee that two subsequent calls to IWeakReference.Get() will return the same instance, so all comparisons must be based off of the identity value. The string type is used as the type for the identity rather than an integer or Guid to keep the requirements for the identity flexible. Aside from integers and Guids, it may be desirable to use URLs or other objects with string representations as the identity. The IWeakReferenceList and its sub-interface that adds methods with generic type constraints help the filesystem developer implement folders with the weak reference pattern. NOFS offers a default implementation of WeakReferenceList that can be used. NOFS also provides factories for creating IWeakReference instances given an identity value and provides the implementation for the Get() call.

In addition to the weak reference pattern which is important in determining which file and folder objects are kept in memory, there is an additional pattern that manages the data blocks of regular files. By default, NOFS regular file domain objects are translated to and from XML using the .NET serializer. If the domain object implements IProvidesUnstructuredData, then the file contents can be of a custom structure that is managed by the domain object. With this interface, the user can choose to either make use of an externally managed data source or one managed by NOFS. For the latter case, a new interface IDomainObjectRawDataStore provides methods for reading, writing, and truncating a binary file. With these two interfaces, it is possible for a NOFS filesystem to be unconcerned with the details of file reading or writing by implementing neither interface, or to be concerned with those details by implementing one or both interfaces.

```
interface IDomainObjectRawDataStore {
  long DataSize();
  int Read(byte[] buffer,long offset,long len);
  int Write(byte[] buffer,long offset,long len);
  void Truncate(long length);
}
```

```
interface IProvidesUnstructuredData
  : IDomainObjectRawDataStore {
  bool Cacheable();
}
```

With these two interfaces and the weak reference pattern, we were able to implement a simple storage based filesystem that uses the host's filesystem as the backing store with the NOFS framework in less than 300 lines of C# code.

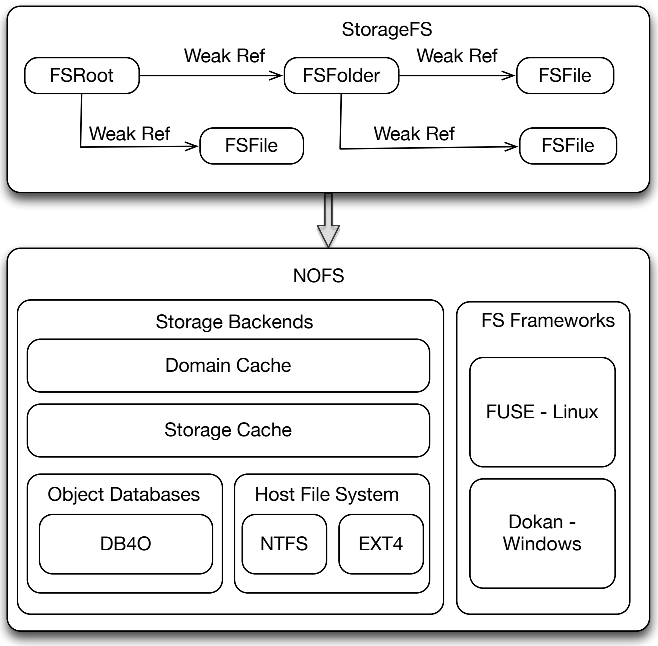

Fig. 8. Architecture of filesystem implementation as it relates to NOFS

The root of our reference implementation is expressed as the following class.

```
[RootFolder]
class FsRoot : FsFolder
{
  public FsRoot()
   : base("", Guid.NewGuid().ToString()){}
  public FsRoot(string name, string id)
   : base(name,id){}
}
```

This class is the instance from which the NOFS framework translates all paths. It represents the '/' part of any path passed to NOFS. The FsRoot class subclasses the FsFolder class.

```
[FolderObject]
class FsFolder : FsFolderOrFile,
   IWeakReferenceList<FsFolderOrFile>
{

}
```

The FsFolder class subclasses FsFolderOrFile and implements the IWeakReferenceList interface. When NOFS encounters instances of IEnumerable that have that attribute FolderObjectAttribute, it recognizes them as folders in the filesystem. Because FSFolder implements IWeakReference, all of the files and folders that are contained are weakly connected and do not need to be in physical memory for NOFS to load and examine the folder.

```
class FsFile : FsFolderOrFile,
   IProvidesUnstructuredData
{
  IDomainObjectRawDataStore _data;
  [NeedsRawDataStore]
  void SetDataStore(
    IDomainObjectRawDataStore data){
    _data = data;
  }

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

  bool Cacheable(){
    return false;
  }

  int Read(byte[] buff,long off,long len){
    return _data.Read(buff,off,len);
  }

  int Write(byte[] buff,long off,long len){
    return _data.Write(buff,off,len);
  }

  void Truncate(long length){
    _data.Truncate(length);
  }
}
```

FsFile is our class to represent regular files. This class implements IProvidesUnstructuredData. When NOFS sees a class implement the IProvidesUnstructuredData interface, it allows that class manage the Read, Write, and Truncate filesystem calls. FsFile also accepts an instance of IDomainObjectRawDataStore from NOFS after it is constructed. This helper interface provides access to the host filesystem file that contains the data for the file represented by this FsFile instance. Although in this example, the call is a simple pass through, it is possible to construct more complex implementations that translate the underlying data or provide other additional value.

```
[DomainObject]
class FsFolderOrFile : IObjectWithID
{
  private string _name;
  [NeedsContainer]
  IDomainObjectContainer Container {get;set;}
  string Id {get;set;}

  [ProvidesName]
  string Name {
    get { return _name; }
    set {
      _name = value;
      if (Container != null) {
        Container.ObjectChanged(this);
      }
    }
  }
}
```

FsFolderOrFile is the base class for folders and regular files in our reference implementation. This class is the type used in the IWeakReferenceList by the FsFolder class. This base class allows for both the regular files and folders to have a common type. This class is recognized as a regular file by NOFS because of the DomainObjectAttribute attribute. In the case of the FsFolder subclass, it is recognized as a folder because of the FolderObjectAttribute attribute. NOFS is able to determine that the Name property manages the file or folder name because of the ProvidesNameAttribute attribute. NOFS injects the IDomainObjectContainer in the Container property after the construction of a FsFile or FsFolder instance. The IDomainObjectContainer class manages serializing and deserializing of domain objects in the NOFS framework.

References

[1] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from Bell Labs," *Computing Systems*, vol. 8, pp. 221–254, Summer 1995.

[2] M. Szeredi, "Filesystem in Userspace." feb-2005.

[3] J. Kaylor, K. Laufer, and G. K. Thiruvathukal, "RestFS: resources and services are filesystems, too," in *Proceedings of the Second International Workshop on RESTful Design*, 2011, pp. 39–46.

[4] J. Kaylor, K. Laufer, and G. K. Thiruvathukal, "Online Layered File System (OLFS): A Layered and Versioned Filesystem and Performance Analysis," in *Proc. IEEE Intl. Conf. on Electro/Information Technology (EIT)*, 2010.

[5] R. Pawson, "Naked Objects," 2004.

[6] R. Card, "ISBN 90-367-0385-9. Design and Implementation of the Second Extended Filesystem," 1994.

[7] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010, pp. 206–213.

[8] K. Donnelly, J. J. Hallett, and A. Kfoury, "Formal semantics of weak references," in *Proceedings of the 5th international symposium on Memory management*, 2006, pp. 126–137.

[9] S. L. Peyton Jones, S. Marlow, and C. Elliott, "Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell," in *Selected Papers from the 11th International Workshop on Implementation of Functional Languages*, 2000, pp. 37–58.