ESPRIT Project No. 25 338

Work package C

Information Space

Requirements, Design, Interfaces and Report DC 1, 2, 3 and 6.1

ID:	InformationSpace DC 1, 2, 3 and 6.1	Date:	05.03.98
Author(s):	Douglas Donaldson, APM	Status:	Reviewed
Reviewer(s):	Laurent Amsaleg, Inria/Irisa	Distribution:	Project Confidential

Change History

Document Code	Change Description	Author	Date
InformationSpace	InformationSpace DC 1, 2, 3 and 6.1. No changes.	Donaldson, APM	31.3.98

1. Requirements

1.1 Original Requirements

The InformationSpace workpackage of the FollowMe project was conceived of as a set of transparency mechanisms to give a mobile user a consistent logical view of their information [1]. These mechanisms allow the physical location of the data to be hidden, and allow data to be distributed for efficient access. A further requirement was to create a user authentication procedure to control access.

While the project objectives as a whole have not changed, the project architecture has progressed leading to a change in emphasis between workpackages. The requirements on the InformationSpace are re-expressed here, making a distinction between the long term objectives of FollowMe as a research project, and the short term requirements of the pilot applications.

1.2 Objectives of the Research Project

The architectural goal of the InformationSpace workpackage is to extend the range of transparencies available through the FollowMe middleware. The Mobile Object Workbench provides location transparency and migration transparency for volatile objects and services. The InformationSpace aims to provide persistence transparency (robust objects) and a range of migration and replication transparencies for robust objects. This architectural base forms a general purpose middleware for application development independently of the more specific objectives of FollowMe; for this reason, the name of the workpackage was renamed from Personal Information Space to InformationSpace[1].

The architectural vision of the InformationSpace is to support application development by providing an extensible set of implementations of persistence mechanisms. Distributed application components request access to information objects, and the middleware components cooperate to supply information efficiently and consistently.

The InformationSpace is therefore not concerned with just giving mobile users a consistent logical view of their information. Instead, it is general purpose middleware which gives other application layers a consistent logical view of their information. The pilot applications form examples of such application layers. The Personal Assistant agent is an example of an application component directly concerned with the information requirements of mobile users. It is sensible to describe the InformationSpace used by a Personal Assistant as a Personal Information Space.

[1] It had a dreadful acronym too.

1.3 Software Requirements of FollowMe

From the pilot applications' perspectives, the primary requirement on the InformationSpace workpackage is to add persistence transparency to the location transparency of the MOW/FlexiNet ORB. Mobile objects are vulnerable to network and machine failure. They need to be able to put their information somewhere safe, where it can be recovered after a failure.

The pilot applications don't immediately require migration and replication of storable objects. Ideally, transparent migration and replication could be transparently added to versions of the InformationSpace used by the pilot applications, giving them performance benefits. More realistically, the ability to move storable objects would be initially be under the control of the application.

Requirement for User Authentication

User authentication issues for InformationSpace objects are the same as for other application and middleware objects. Therefore, problems and solutions are not discussed here.

MOW and FlexiNet provide a general binding architecture, with provision for bindings with desirable security properties. Similarly, the properties of the Information Space are achieved with specialised bindings. Secure InformationSpaces are therefore also achieved using a combination of binding components.

1.4 Consistent Logical View

The requirements for the InformationSpace state that the workpackage's mechanisms should give a mobile user a consistent logical view of their information. Figure 1 shows the intention of that phrase. A person's information (logical InformationSpace) is shown as a cloud, conceptually smeared across distributed computer systems. The user does not want to be concerned with the precise physical location of his/her information (represented by the T bar interface symbols). S/he would rather identify herself to the system, and be presented with her information, regardless of her geographic location or means of human-computer interaction. Additional difficulty is represented by the existance of other users and agents, whose InformationSpace may all overlap.

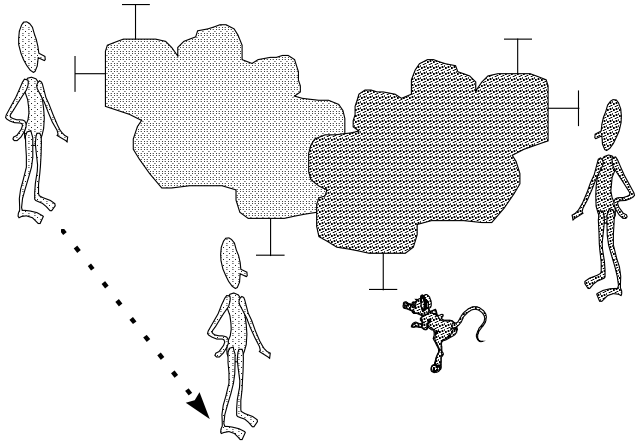

Figure 1 Users, Agents and InformationSpaces.

This is a difficult requirement for a computer system to meet. Information objects have a physical location on hosts which form the computer system. The physical objects which constitute an InformationSpace may be distributed across several hosts.

As a consequence of client mobility and distribution, mobile hosts may make varied connections to the system, and the information objects may be required to move or be copied between physical locations. Figure 2 shows the redeployment of information objects from host A to host B as the client moves to a different physical interface.

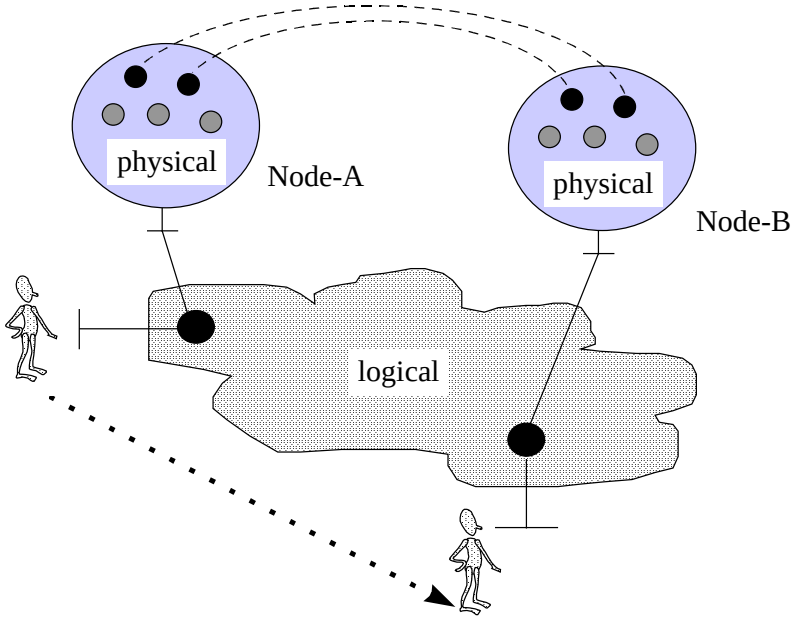

Figure 2 Logical and Physical InformationSpaces.

Algorithms for management of replicated information have to strike a compromise between service availability and consistency. They cannot both be achieved in the presence of communication and host failures.

1.5 Object Structuring and Ownership

How can algorithms for information management be chosen? As issues of information movement and replication are explored, it becomes clear that not all information is equal. Frequently, knowledge of the structure and meaning of information can be used to aid a choice of distribution strategy, for example whether to use optimistic or pessimistic replication control.

For simple data objects, including files, a choice of movement or replication would be made according to where and how the clients will access the data. Distributed clients can be given efficient read-only access to the information by replicating it. When the information changes, whether or not copies have to be immediately or gradually brought up to date depends on its meaning. A newspaper, for example, is timestamped, so readers can make their own assessment of its timeliness.

For structured information, such as a CAD document, knowledge of the meaning and the structure can allow efficient distribution of the contents. A CAD document of a building can be distributed according to how the team of technical authors are organised. The document can be split spatially if the authors work on different rooms, but split by utility if the authors are working on different aspects of infrastructure.

A useful concept to help with such discussions is *ownership*. A common pattern in object modelling is grouping, where a group object is identified as a container, aggregate or index over the group members. For simple data objects, the group can act as an owner or manager for the objects, controlling distributed access to the members. Complex data objects may have varied indexes over them, where no index has controlling rights over its contents. Such objects may own or manage themselves, coordinating distributed requests for access which arrive.

2. Design

2.1 Vision

The design vision is to produce an architecture for InformationSpaces which can be implemented incrementally, initially satisfying the short term requirements of the pilot applications, and later yielding the long term research objectives. Persistence transparency is the core requirement, with failure recovery, mobility and replication transparencies to be implemented later. Additional facilities which may be required include customised concurrency control, customised serialisation of objects and transactional capabilities. Additional desirable features of the design are integration with existing middleware (in particular the Mobile Object Workbench) and file system independence.

2.2 First Cut Design

A low level approach to storing objects is to abstract a file system as an object store service. Clients, including mobile objects, can be given a remote interface reference to this service and send objects by value to the store, giving the object a name. They can later be retrieved by name. A Java interface to such a service would be like this:

```
public interface CopyStore
{
  public String[] list();
  public void copyInto(String name, Object obj) throws ...;
  public Object lookUpCopy(String name)
    throws NameNotFoundException, ...;
  public void remove(String name)
    throws NameNotFoundException, ...;
  public String[] listCopyNames() throws ...;
  public Object[] listCopies() throws ...;
}
```

This has uses, but does not provide transparent persistence or help multiple clients to share the storable objects. A client has to copy the object out to examine or change it, then copy it back in. If multiple clients do this then some client's versions may be overwritten.

For a higher-level, more object-oriented approach, the storable objects should be first class objects. They should be able to offer services to clients through a method interface. A Java interface allowing such storables to be created looks like this:

```
public interface StorableStore
{
  public String getName();
  public Tagged newStorable(String name, Class clazz)
    throws InstantiationException, Exception;
  public Tagged lookup(StorableString name) throws ClientException, Exception;
  public Object lookupCopy(StorableString name)
    throws ClientException, Exception;
  public void remove(String name)
    throws ClientException, Exception;
  public String[] listStorableNames() throws Exception;
  public Tagged[] listStorables() throws Exception;
}
```

The newStorable method of this interface creates objects which may be transparently persistent. The return type, 'Tagged', is an abstraction of a Java interface of a named type [2]. The interface to the storable returned to the caller can be used as a normal reference. However, the Store is able, behind the scenes, to arrange that the effects of each invocation are stored before the result is returned to the client.

2.3 How this integrates with FlexiNet

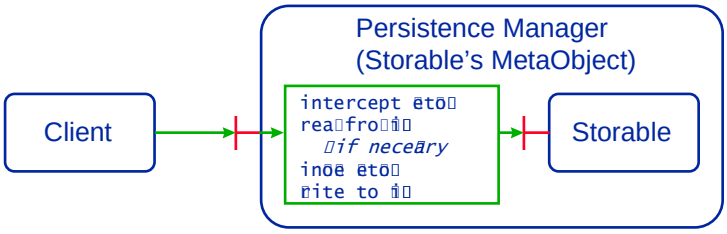

Figure 3. Encapsulated Storable

As clients must use the Store interface to manufacture storable objects, and clients are returned an interface reference to the new storable, a desirable isolation between clients and storables is achieved. Clients do not get object references to the storables, allowing the implementation of the storable's interface to be fully encapsulated. This is the same separation which allows the MOW's Places to manufacture Clusters with encapsulation.

Conceptually, the Store interposes a persistence manager between the client and the storable. The persistence manager reflects the storable's interface, interposing persistence behaviour before and after invocations. The persistence behaviour recovers the storable from disk if necessary, applies the invocation, saves the storable back to disk, then returns results to the client (Figure 3).

In practice, the Store's persistence manager is realised as a layer in the server side protocol stack (Figure 4). The persistence manager acts as a meta object for the destination object. This shows the InformationSpace as an example of FlexiNet binding.

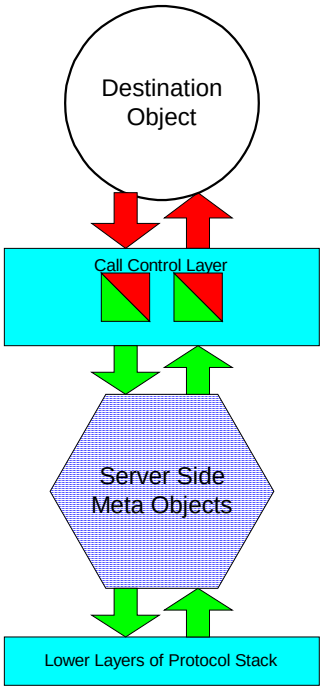

Figure 4. Meta Object in the Server Side Stack

2.4 Enhancing the Design

The design above, of a persistence manager acting as a meta object for a storable, generalises allowing advanced features to be hooked in. Some examples are given below.

From these examples, it can be seen that storables need a management policy, expressing the strategies in place for a given storable. Owner is a synonym for manager or meta-object, expressing the idea that an object's owner has a vested interest in the behaviour of the object. It is the owner's policy which dictates issues such as when requests for a storable to migrate are allowed or not, access control, lifespan of storables, resource usage, etc.

Advanced Persistence

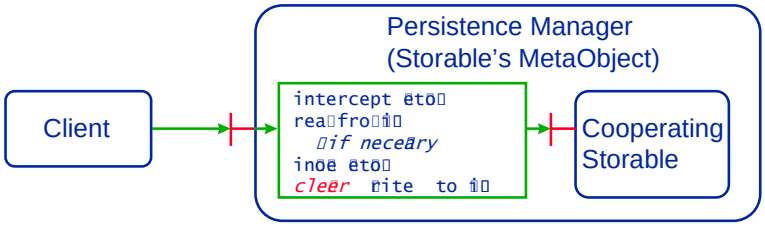

Figure 5. Advanced Persistence

If the meta object has knowledge of the semantics of the storable's methods, then it can take advantage of this for increased efficiency. After a read-only method, the storable doesn't need to be re-saved to store. It the meta object has knowledge of a cooperating storable's structure, it can efficiently save just the changes after a method invocation (Figure 5).

Advanced Concurrency Control

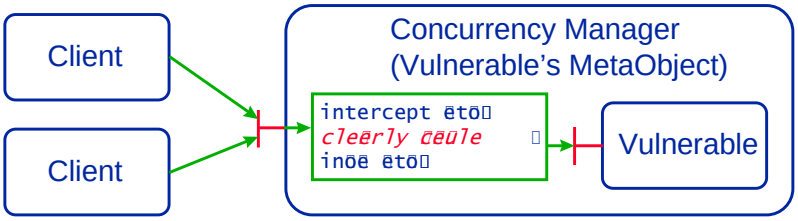

Figure 6. Advanced Concurrency Control

By default the meta object must serialise all invocations on the storable. If it allowed multiple concurrent method invocations, a client might be informed that its method invocation was successful, when storable hasn't reached a quiescent state allowing its meta object to store it. Storables must be inactive before they can be serialised to store.

If the meta object has knowledge of the semantics of the storable's methods, then it can take advantage of this for increased efficiency. It can schedule read-only methods concurrently. It can apply advanced scheduling algorithms, which take into account the storable's state, client's priority, method names and parameters, and the invocation history (Figure 6) [3].

Transactions

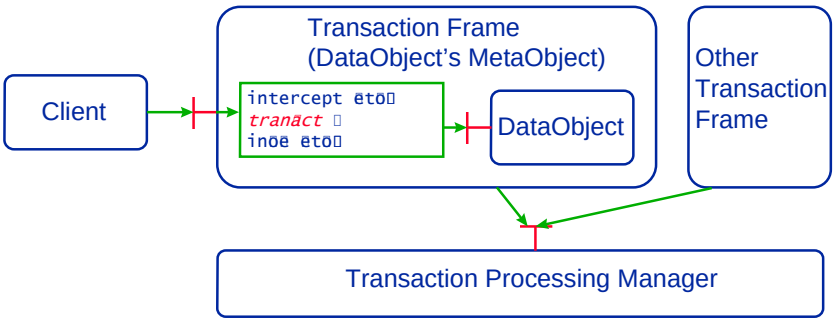

Figure 7. Transactions on Storables

The invocations on a storable may be one of a set of invocations on a set of storables forming a transaction. The storable's meta object can transparently coordinate the transaction, by using a Transaction service (Figure 7). This idea of a meta object as a Transaction Frame is being explored in the ANSA FlexiNet project, independently of FollowMe [4].

Automatic Failure Recovery

If a storable's host fails, the host can be restarted, and the client can rebind to the storable using `lookupStorableString(name)`. By default, this rebinding is not implicit, and the client would have to detect the failure before attempting reconnection. However, it could be arranged that when an interface reference to a storable is bound, the client stub is 'clever', in that it can automatically catch failures and attempt to rebind on behalf of the client. Such a mechanism would have similarities with that of a client of a mobile object, which must rebind after object movement.

Migration Transparency

A Store may have a method allowing a storable to be moved to another Store. This might be desirable for efficient access to the storable by client's near that Store's host. Movement failure would have to leave the storable untouched (losing a storable is unacceptable). The mechanisms of the MOW could allow the move to be transparent to clients.

A 'clever' client stub could transparently request that the storable moves near to the client. This would allow the storable to *follow* the client. The clients, Stores and storables would have to be configured with a pragmatic policy for deciding when to move, to prevent thrashing.

Replication Transparency

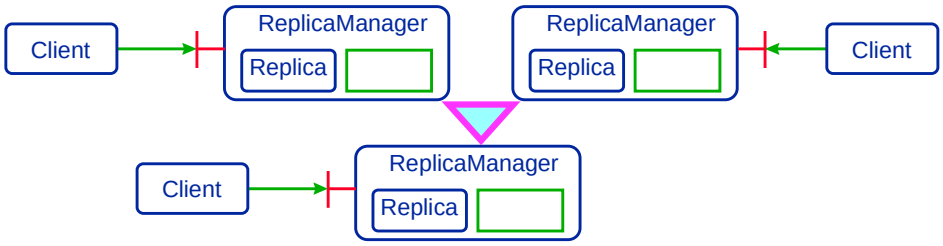

Figure 8. Replication of Storables

Similarly to migration, a Store could be requested to replicate its storables (Figure 8). 'Clever' client stubs would be required which could take advantage of the efficiency opportunities of the replicated information. As discussed in section 1.5, choice of replication algorithm would depend on the meaning of the information in question.

2.5 Why Name Storables?

Storable objects could be created in a Store without a name. The client gets an interface reference back which is sufficient to allow the storable to be used. However, the client needs to be able to name the storable so that it can regain an interface reference to it if the Store crashes (then recovers). Additionally, the administrator of the physical storage may wish to browse the storables and recover disk space. This is impossible without associating a name with the storable. Although there is no guarantee that the name supplied when creating a storable is meaningful, it offers the possibility for cooperating clients to help the adminstrator.

In practice, a contract exists between the client and the administrator, where the administrator lends or rents physical storage to the client. A client would not keep valuable information in a Store which is not trusted. An administrator would want to configure the physical storage so that user authentication is required for storable objects.

2.6 Naming Structure for Storables

The CopyStore and StorableStore interfaces in section 2.2, First Cut Design, show the Stores naming their contents. A Store may be shared, however, between many clients, and more structured names are desirable to prevent conflicts. To achieve this, a Directory interface is introduced. Directories form a tree structure. Each store is associated with a Directory tree. The tree is navigated by Directory name from a root Directory. The Directory, not the Store, offers the copyIn

and newStorable methods for associating names with objects, and a newDirectory method for creating sub Directories. All the objects and storables are in the same Store, which manages them (Figure 9).

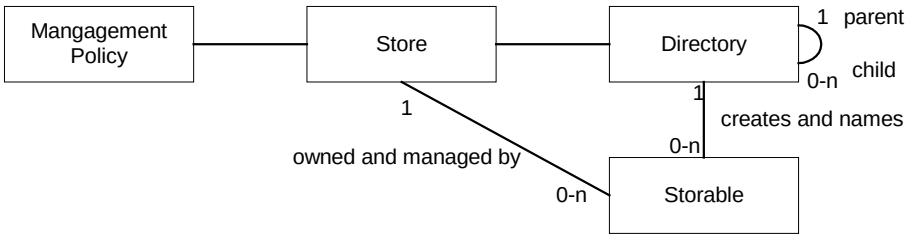

Figure 9. Class Diagram of Stores, Directories and Storable Objects.

2.7 Managing Stores

A Store as described above is the owner and manager for all the storables in its Directory tree. The Store forms a natural group object for its storables, and can either be a meta object for all the storables, or can insert meta objects into the bindings for storables (one per Store or one per storable).

This means that in a given system, different Stores will be configured with different management policies for their storables. The Store, then is not at the granularity of a physical storage area.

A physical storage area, such as a physical file system or database may be shared by many applications, agent systems or agents. To allow applications to share storage areas, a management layer needs to be introduced:

```
public interface StoreManager
{
  public Store newStore(String name) throws StoreException;
  public Store newStore(String name, Class cls, Object[] args)
    throws InstantiationException, StoreException;
  public void removeStore(String name, Client client) throws StoreException;
  public String[] listStores() throws StoreException;
  public void emptyStore(String name) throws StoreException;
  public Store lookupStore(String name, Client client) throws StoreException;
}
```

A StoreManager provides a name space of Stores, and is also responsible for a region of physical storage. The physical storage may be a disk or a database (Figure 10). Method newStore(name) returns a default Store. Such a Store may be configurable with a management policy. Method newStore(String name, Class cls, Object[] args) allows custom Stores to be created with different Store-like behaviours.

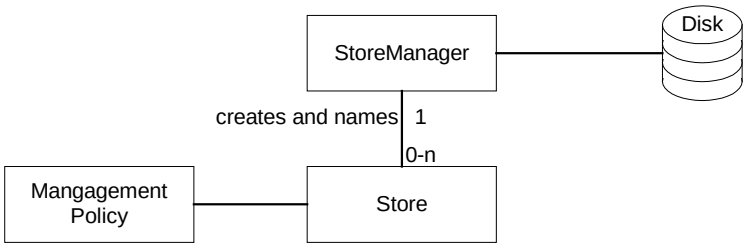

Figure 10. Class Diagram of StoreManagers and Stores

Clients may require several StoreManagers, at different physical locations or with different policies. Rather than introduce StoreManagerManagers, or hierarchic StoreManagers, this is regarded as an issue of middleware configuration. Administrators of physical storage are expected to register them in a NameServer's Directory, where clients can look them up.

2.8 Strong versus Weak Grouping

In a complex distributed application, information will be indexed in diverse ways. In some application architectures, the indexes will effectively be strong containers or group objects for the information, meaning that the index owns the information. In other architectures, the indexes will have a weaker meaning, with possibly sharing the contents with other orthogonal indexes. The information objects in this case are not owned by the indexes, and may be separately configured with their own persistence policies.

The architecture of Stores and StoreManagers described above aims to support both such viewpoints. An index which represents a weak group is just an object (possibly storable, but not a Store) which holds a set of interface references to storables. These storables can be shared between many indexes. Client invocations can be interpreted as requests to move the information near to the client, but the decision to move or not is up to the storable's owner's policy.

Indexes which represent strongly encapsulated groups can be implemented as custom Stores, with custom methods (not necessarily newStorable and remove) for manipulating its contents. The custom Store, being the container and owner of its contents, can mediate access to them.

2.9 Locking

No locking operations are described on the Store, allowing a client to perform a sequence of operations on a Storable in mutual exclusion before unlocking. This approach has technical difficulties in a distributed environment, for example the client may fail before unlocking the storable.

The long term strategy to allow sequences of operations is to integrate a transaction processing manager with the InformationSpace.

In the short term, applications must be structured to take account of the granularity of atomic actions provided by storable objects, which is at the level of method operations. For example, if a client needs the two items from the head of a stored list, the list data type should provide an operation to return the two items at its head. Alternatively, the client needs to be assured that it is the only one, for example from knowledge of its system configuration.

2.10 Stores, StoreManagers, and Places

Stores and StoreManagers don't have to run at a Place (in the FollowMe sense). There is no requirement for MobileObjects to move to them.

However, their implementation may use the encapsulation mechanisms of Clusters, which are manufactured by Places. So the implementations may create a Place, whose interface reference may be public or anonymous.

A future implementation of movement of storables may use the movement mechanisms of MobileObjects. This can again be implemented without publishing interface references to Places used in the implementation.

A typical system configuration would have one StoreManager per physical storage area. This would be used to create Stores with different policies. Clients could get an interface reference to a StoreManager from a NameServer or from a Place. A Place would be configured with one or more StoreManagers which it may tell clients about when they request local storage. StoreManagers may be shared between Places (Figure 11).

Figure 11. Class Diagram of Places and StoreManagers

2.11 Implementation of Directory and StoreManager

For flexible implementation of Directory and Store, a layer of abstraction needs to be introduced between them and the storage area. DataDirectory forms such a layer (Figure 12):

```
public interface DataDirectory
{
  public String getName();
  public DataDirectory getParent();
  public void copyToString name, byte[] data, boolean append throws IOException;
  public byte[] loadData(String name, int len) throws FileNotFoundException, IOException;
  public String[] listData();
  public DataDirectory newDataDirectory(String name) throws IOException;
  public DataDirectory loadDataDirectory(String name)
    throws FileNotFoundException, IOException;
  public String[] listDataDirectory() throws IOException;
  public void rm(String name, boolean recurse)
    throws FileNotFoundException, IOException;
}
```

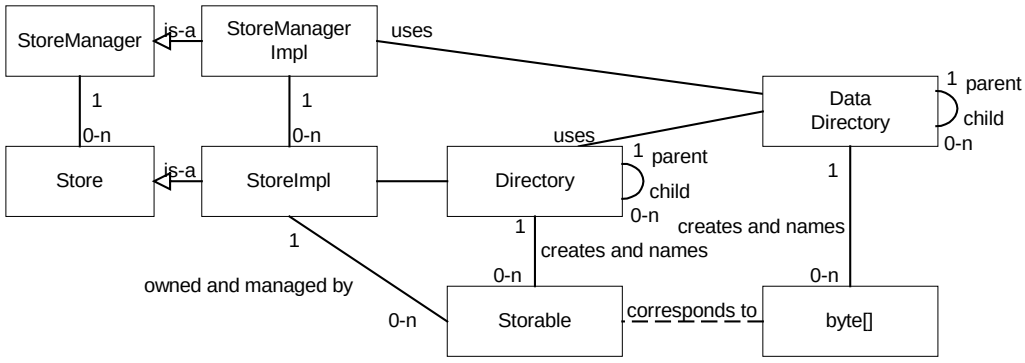

Figure 12. Class Diagram for DataDirectory

2.12 Implementation of DataDirectory

The proposed implementation of DataDirectory is FSDirectory (aka FileSystemDirectory), which uses a file system for its implementation (Figure 13). An alternative might be a DatabaseDirectory, which used a database for its implementation.

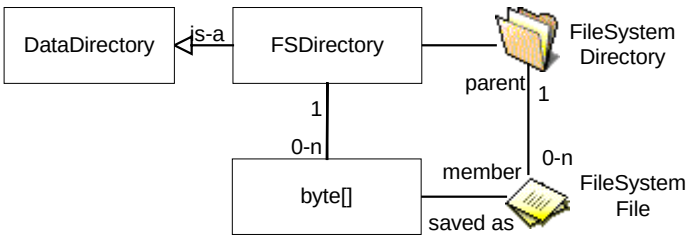

Figure 13. Class Diagram for FSDirectory

2.13 Implementation of FSDirectory

FSDirectory needs to make named sub directories, and save byte arrays as named files. Two issues are atomic write of data, and file naming.

For writing, appending and overwriting, it is unacceptable for a failure half way to leave the state inconsistent. The copyIn operation of DataDirectory must be atomic. This is achieved by writing the data to a new file name, then renaming the new to the actual file name. The rename must be done in two steps (first renaming the old to an old file), because not all file systems (NT) support atomic rename over an existing file. The state diagram for copyIn is shown in Figure 14.

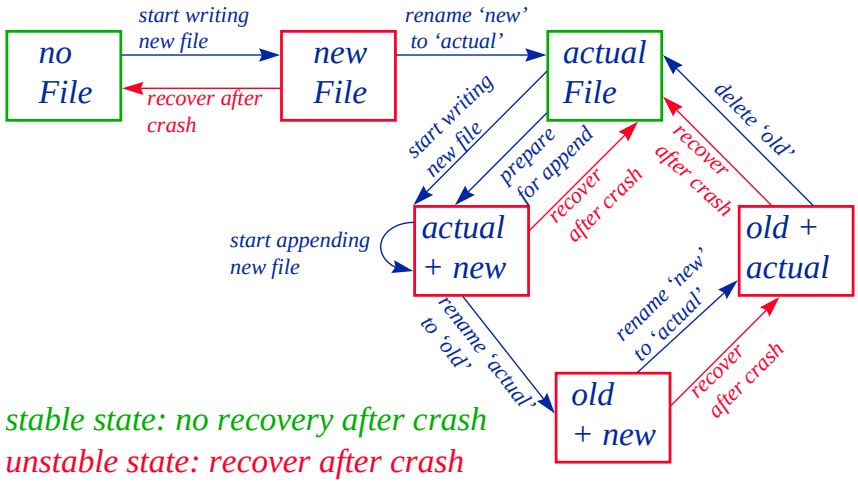

Figure 14. State Diagram for Atomic Write

The names supplied FSDirectory to it are arbitrary Java strings. They may not be valid file names. A StringMapper interface is used to map Java strings to file names:

```
public interface StringMapper
{
  public String mapStringToName(String name);
  public String mapNameToString(String filename) throws IOException;
}
```

3. Interfaces

3.1 Interface UK.co.ansa.ispace.StoreManager

public interface **StoreManager**

A StoreManager creates Stores. Stores are named at creation, and this name can be used to bind to the Store. If a Store-Manager is reinstantiated after a crash, it can reinstantiates all the Stores it knows about, allowing clients to rebind to them.

See Also:
Store

Methods

newStore

```
public abstract  Store newStore(String name) throws  ISException
```

Create a named Store.

Parameters:
name - A name for the Store which is meaningful to the client.
Returns:
The new Store.
Throws: ISException
The Store could not be created.

newStore

```
public abstract Tagged newStore(String name,
                                Class cls) throws InstantiationException,  ISException
```

Create a named store of user defined type. Once created, init() will be called on the new object.

Parameters:
name - A name for the store which is meaningful to the client.
cls - The class of the store to be created.

Returns:
An interface to the new store.
Throws: InstantiationException
The storable could not be created, or init() raised an exception.
Throws: ISException
The Store could not be created.

newStore

```
public abstract Tagged newStore(String name,
                                Class cls,
                                Object args[]) throws InstantiationException,  ISException
```

Create a named store of user defined type. Once created, init(arg0,arg1,...) will be called on the new object.

Parameters:
name - A name for the store which is meaningful to the client.
cls - The class of the store to be created.
args - The arguments to pass to `init(...)`
Returns:
An interface to the new store.
Throws: InstantiationException
The storable could not be created, or init() raised an exception.
Throws: ISException
The Store could not be created.

remove

```
public abstract void remove(String name) throws IllegalArgumentException,  ISException
```

Destroy a given Store. Its contents are all removed.

Parameters:
name - The name of the Store to destroy.
Throws: IllegalArgumentException
The named Store could not be found.
Throws: ISException
The Store could not be destroyed.

listNames

```
public abstract String[] listNames() throws  ISException
```

List all managed Stores's names.

Returns:
The Stores under management. Not guaranteed to match up with Stores returned from listStores.
Throws: ISException
The list could not be created.

listStores

```
public abstract  Store[] listStores() throws  ISException
```

List all managed Stores.

Returns:
The Stores under management. Not guaranteed to match up with names returned from listNames.
Throws: ISException
The list could not be created.

lookup

```
public abstract  Store lookup(String name) throws IllegalArgumentException,  ISException
```

Get a reference to a named Store.

Parameters:
name - Name of the Store to lookup.
Returns:
The Store named name.
Throws: IllegalArgumentException
name not known.
Throws: ISException
The name is known but does not correspond to a Store created using newStore, or the Store could not be looked up.

3.2 Interface UK.co.ansa.ispace.Store

public interface **Store**

A Store is used to store Objects. Stores are created by StoreManagers and other Stores, which delegates authority to the Store to use some storage area. Stores manage their storables with a single management policy. Management policies cover issues such as persistence transparency choices, concurrency control choices, mobility and replication transparencies, etc. Stores use a directory hierarchy for naming storables.

StoreManagers can reinstantiate Stores, which can recreate their state from store.

See Also:
StoreManager, Directory

getName

```
public abstract String getName()
```

Find out the Store's name. A Store is given a name when created by a StoreManager.

Returns:
The Store's name relative to its StoreManager.

getStoreManager

```
public abstract  StoreManager getStoreManager()
```

Find out the Store's Manager. assert(getStoreManager().lookup(getName()) == this); // all things being equal

Returns:
The Store's Manager.

getRootDirectory

```
public abstract  Directory getRootDirectory()
```

Get the Store's Directory hierarchy.

Returns:
The Store's root Directory.

3.3 Interface UK.co.ansa.ispace.Directory

public interface **Directory**

A Directory is name structure for objects in a Store. Directories are created by Stores and other Directories. Stores manage their storables with a single management policy. Management policies cover issues such as persistence transparency choices, concurrency control choices, mobility and replication transparencies, etc. Directories form naming hierarchies for storables.

A storable object is given a name when it is put into a Directory. This name is used to retrieve a copy of or reference to the object.

Stores can reinstantiate Directory structures after a crash, which can recreate their state from store.

Two styles of storable object are described here.

1. 'black box' style, where objects are copiedIn, copiedOut or removed from the Directory. The client has to manage the copies and versioning. A 'black box' object in the Directory is dead, in that methods cannot be invoked on it; either the object would have to be copied out, modified, then copied back in, or else the client keeps its own copy, and backs it up by recopying it into the Information Space.
2. The second style is a 'white box' style, where the object 'lives' in the Directory, and clients can invoke methods on it. Such an information object is created at the Directory using newStorable(), which returns an interface for the client to use. This is a 'magic' interface, because the Store intercepts all invocations, reconstructs the object from store, does the invocation, then saves the modified object back to disc. For such a strategy to work, some assumptions are necessary about the stored object. It should not engage in unterminating computations, or communications. The effects of the invocation may have to be bounded.
This white box style is intended as a hook for more specialised behaviours. The Store's default implemenation of persistence could be substituted for more intelligent behaviour, which allow more concurrency than the default, distributed transactional behaviour, or more efficient streaming of changes to storage.
The magic references allowing the persistence transparency may also extended allowing replication transparency or mobility transparency of stored objects.

See Also:
StoreManager, Store

Methods

getName

```
public abstract String getName()
```

Find out the Directory's name. A Directory is given a name at creation, except for the root Directory.

Returns:
The Directory's name relative to its creator. Null if this is the root Directory.

getAbsoluteName

```
public abstract String[] getAbsoluteName()
```

Find out the Directory's name relative to the root. Directorys are created in a tree hierarchy.

Returns:
The Directory's name relative to the root. Empty if this is root.

getPath

```
public abstract String[] getPath()
```

Find out the Directory's path name. This is the absolute name of its parent. if (!this is root) getPath + getName = getAbsoluteName.

Returns:
The Directory's parent's absolute name. Null if this is the root.

getParent

```
public abstract  Directory getParent()
```

Find out the Directory's Parent. if (!this is root) getParent().lookupDirectory(getName()) == this

Returns:
The Directory's Parent. Null if this is the root.

getRootDirectory

```
public abstract  Directory getRootDirectory()
```

Find out the Directory's Root. All Directories in a tree have the same root. getRoot().lookupDirectories(getAbsoluteName()) == this

Returns:
The Directory's root Directory.

getStore

```
public abstract  Store getStore()
```

Find out the Directory's Store. All Directories in the tree have the same Store. getStore().getRootDirectory().lookupDirectories(getAbsoluteName()) == this

Returns:
The Directory's Store.

listNames

```
public abstract String[] listNames() throws  ISException
```

List all stored objects' and sub Directory's names.

Returns:
The names of the stored objects.
Throws: ISException
The list could not be created.

listObjectNames

```
public abstract String[] listObjectNames() throws  ISException
```

List all stored objects' names.

Returns:
The names of the stored objects.
Throws: ISException
The list could not be created.

copyInto

```
public abstract void copyInto(String name,
                              Object obj) throws  ISException
```

Put an Object by value into the Directory. If there is already an object with that name, it gets overwritten.

Parameters:
name - A name for the Object which is meaningful to the client.
obj - The object to be copied into the Directory.
Throws: ISException
The Object could not be stored.

listCopyNames

```
public abstract String[] listCopyNames() throws  ISException
```

List all stored copies's names.

Returns:
The names of the stored objects copiedIn.
Throws: ISException
The list could not be created.

lookupCopy

```
public abstract Object lookupCopy(String name) throws IllegalArgumentException,  ISException
```

Copy a named object by value out of the Directory. The original is untouched.

Parameters:
name - The name of the object to lookup.
Returns:
The reconstituted object.
Throws: IllegalArgumentException
The name is not known.
Throws: ISException
The name is known but does not correspond to an object copied in, or the Object could not be looked up.

listCopies

```
public abstract Object[] listCopies() throws  ISException
```

List all objects copiedIn.

Returns:
The stored objects.
Throws: ISException
The list could not be created.

newStorable

```
public abstract Tagged newStorable(String name,
                                   Class cls) throws InstantiationException,  ISException
```

Create a new storable object inside this Directory. Once created, init() will be called on the new object.

Parameters:
name - A name for the Object which is meaningful to the client.
cls - The class of the storable to be created.
Returns:
An interface to the new storable.
Throws: InstantiationException
The storable could not be created, or init() raised an exception.
Throws: ISException
The Object could not be stored.
See Also:
Tagged

newStorable

```
public abstract Tagged newStorable(String name,
                                   Class cls,
                                   Object args[]) throws InstantiationException,  ISException
```

Create a new storable object inside this Directory. Once created, init(arg0,arg1,...) will be called on the new object. The Store will attempt to find a matching method. If more than one method matches, the resulting behaviour is undefined.

Parameters:
name - A name for the Object which is meaningful to the client.

cls - The class of the storable to be created.
args - The arguments to pass to `init(...)`
Returns:
The interface to the new storable.
Throws: InstantiationException
The storable could not be created, or init() raised an exception.
Throws: ISException
The Object could not be stored.
See Also:
Tagged

listStorableNames

```
public abstract String[] listStorableNames() throws  ISException
```

List all storable's names.

Returns:
The names of the objects created with newStorable.
Throws: ISException
The list could not be created.

lookupStorable

```
public abstract Tagged lookupStorable(String name) throws IllegalArgumentException,  ISExcep-
tion
```

Get a reference to an (interface of a) named Storable Object.

Parameters:
name - Name of Storable to lookup.
Returns:
The interface to the storable named name.
Throws: IllegalArgumentException
The name is not known.
Throws: ISException
The name is known but does not correspond to a storable created using newStorable, or the storable could not be looked up.

listStorables

```
public abstract Tagged[] listStorables() throws  ISException
```

List all storables.

Returns:
The storable objects.
Throws: ISException
The list could not be created.

lookupCopyOfStorable

```
public abstract Object lookupCopyOfStorable(String name) throws IllegalArgumentException,
ISException
```

Copy a named storable by value out of the Directory. The original is untouched.

Parameters:
name - The name of the object to lookup.
Returns:
The reconstituted object.
Throws: IllegalArgumentException
The name is not known.
Throws: ISException
The name is known but does not correspond to a storable or object, or the Object could not be looked up.

newDirectory

```
public abstract  Directory newDirectory(String name) throws  ISException
```

Create a new sub Directory inside this Store.

Parameters:
name - A name for sub Directory which is meaningful to the client.
Returns:
The new sub Directory.
Throws: ISException
The sub Directory could not be created.

newDirectories

```
public abstract  Directory newDirectories(String names[]) throws  ISException
```

Create new sub Directories inside this Directory.

Parameters:
names - Names for the sub Directories which are meaningful to the client.
Returns:
The new sub Directory at the tip of the branches.
Throws: ISException
The sub Directories could not be created.

listDirectoryNames

```
public abstract String[] listDirectoryNames() throws  ISException
```

List all sub Directory's names.

Returns:
The names of the sub Directories.
Throws: ISException
The list could not be created.

lookupDirectory

```
public abstract  Directory lookupDirectory(String name) throws IllegalArgumentException,  ISEx-
ception
```

Get a reference to a sub Directory.

Parameters:
name - Name of the sub Directory.
Returns:
The sub Directory named name.
Throws: IllegalArgumentException
name not known.
Throws: ISException
The name is known but does not correspond to a Directory created using newDirectory, or the Directory could not be looked up.

lookupDirectories

```
public abstract  Directory lookupDirectories(String names[]) throws IllegalArgumentException,
ISException
```

Get a reference to a sub sub Directory. getStore().getRootDirectory().lookupDirectories(getAbsoluteName()) == this

Parameters:
names - The relative path name of the sub sub Directory.
Returns:
The sub sub Directory at the end of the path names.
Throws: IllegalArgumentException
some name not known.
Throws: ISException
Some name is known but does not correspond to a Directory created using newDirectory, or the Directory could not be looked up.

listDirectories

```
public abstract  Directory[] listDirectories() throws  ISException
```

List all sub Directories.

Returns:
The sub Directories.
Throws: ISException
The list could not be created.

remove

```
public abstract void remove(String name) throws IllegalArgumentException,  ISException
```

Delete an object from the Directory. If the name corresponds to a sub Directory which is not empty, deletion fails.

Parameters:
name - The name of the object to destroy.
Throws: IllegalArgumentException
The named object could not be found.
Throws: ISException
The Object could not be deleted.

remove

```
public abstract void remove(String name,
                            boolean recurse) throws IllegalArgumentException,  ISException
```

Delete an object from the Directory. If the name corresponds to a sub Directory and recurse is true, the sub DataDirectory and all its contents are removed. If recurse is false and the DataDirectory is not empty, deletion fails.

Parameters:
name - The name of the object to destroy.
recurse - If true, attempt to recursively delete subdirectories' contents.
Throws: IllegalArgumentException
The named object could not be found.
Throws: ISException
The Object could not be deleted.

3.4 Interface UK.co.ansa.ispace.DataDirectory

public interface **DataDirectory**

A DataDirectory is name structure and store for bytes. It abstracts a file system directory tree structure. A byte arrays are given a name when it is put into a DataDirectory. This name is used to retrieve a copy of the bytes. DataDirectories save bytes on some underlying storage system. When reinstantiated after a crash, they can recreate their state from storage.

getName

```
public abstract String getName()
```

Find out the DataDirectory's name. A DataDirectory may be given a name at creation.

Returns:
The DataDirectory's name relative to its creator. Null if unnamed, e.g. if it is a root.

getAbsoluteName

```
public abstract String[] getAbsoluteName()
```

Find out the DataDirectory's name relative to the root. DataDirectorys are created in a tree hierarchy.

Returns:
The DataDirectory's name relative to the root. Empty if this is root.

getPath

```
public abstract String[] getPath()
```

Find out the DataDirectory's path name. This is the absolute name of its parent. if (!this is root) getPath + getName = getAbsoluteName.

Returns:
The DataDirectory's parent's absolute name. Null if this is the root.

getParent

```
public abstract  DataDirectory getParent()
```

Find out the DataDirectory's Parent. if (!this is root) getParent().lookupDataDirectory(getName()) == this

Returns:
The DataDirectory's Parent. Null if this is the root.

getRootDataDirectory

```
public abstract  DataDirectory getRootDataDirectory()
```

Find out the DataDirectory's Root. All DataDirectories in a tree have the same root. get-Root().lookupDataDirectories(getAbsoluteName()) == this

Returns:
The DataDirectory's root Directory.

listNames

```
public abstract String[] listNames() throws  ISException
```

List all stored data and sub DataDirectory's names.

Returns:
The names of the stored objects.
Throws: ISException
The list could not be created.

copyInto

```
public abstract void copyInto(String name,
                              byte data[]) throws  ISException
```

Write bytes into the DataDirectory. If there is already an object with that name, it gets overwritten.

Parameters:
name - A name for the data which is meaningful to the client.
data - The data to be copied into the DataDirectory.
Throws: ISException
The data could not be stored.

copyInto

```
public abstract void copyInto(String name,
```

```
                                byte data[],
                                boolean append) throws  ISException
```

Write bytes into the DataDirectory. Append data to existing data with that name if append is true.

Parameters:
name - A name for the data which is meaningful to the client.
data - The data to be copied into the DataDirectory.
append - Append bytes if append is true.
Throws: ISException
The data could not be stored.

listDataNames

```
public abstract String[] listDataNames() throws  ISException
```

List all stored data's names.

Returns:
The names of the stored data.
Throws: ISException
The list could not be created.

lookupData

```
public abstract byte[] lookupData(String name) throws IllegalArgumentException,  ISException
```

Read data out of the DataDirectory. The original is untouched.

Parameters:
name - The name of the data to lookup.
Returns:
The bytes.
Throws: IllegalArgumentException
The name is not known.
Throws: ISException
The name is known but does not correspond to data, or the data could not be looked up.

lookupData

```
public abstract byte[] lookupData(String name,
                                  int len) throws IllegalArgumentException,  ISException
```

Read len bytes of data out of the DataDirectory. The original is untouched.

Parameters:
name - The name of the data to lookup.
len - The number of bytes of data to lookup.
Returns:
Len bytes.
Throws: IllegalArgumentException
The name is not known.
Throws: ISException
The name is known but does not correspond to data, or len bytes of data could not be looked up.

listData

```
public abstract byte[][] listData() throws  ISException
```

List all data.

Returns:
The data.
Throws: ISException
The list could not be created.

newDataDirectory

```
public abstract  DataDirectory newDataDirectory(String name) throws  ISException
```

Create a new sub DataDirectory inside this DataDirectory.

Parameters:
name - A name for sub DataDirectory which is meaningful to the client.
Returns:
The new sub DataDirectory.
Throws: ISException
The sub DataDirectory could not be created.

newDataDirectories

```
public abstract  DataDirectory newDataDirectories(String names[]) throws  ISException
```

Create new sub DataDirectories inside this DataDirectory.

Parameters:
names - Names for the sub DataDirectories which are meaningful to the client.
Returns:
The new sub DataDirectory at the tip of the branches.
Throws: ISException
The sub DataDirectories could not be created.

listDataDirectoryNames

```
public abstract String[] listDataDirectoryNames() throws  ISException
```

List all sub DataDirectory's names.

Returns:
The names of the sub DataDirectories.
Throws: ISException
The list could not be created.

lookupDataDirectory

```
public abstract  DataDirectory lookupDataDirectory(String name) throws IllegalArgumentExcep-
tion,  ISException
```

Get a reference to a sub DataDirectory.

Parameters:
name - Name of the sub DataDirectory.
Returns:
The sub DataDirectory named name.
Throws: IllegalArgumentException
name not known.
Throws: ISException
The name is known but does not correspond to a DataDirectory created using newDataDirectory, or the DataDirectory could not be looked up.

lookupDataDirectories

```
public abstract  DataDirectory lookupDataDirectories(String names[]) throws IllegalArgumentEx-
ception,  ISException
```

Get a reference to a sub sub DataDirectory. getRootDataDirectory().lookupDataDirectories(getAbsoluteName()) == this

Parameters:
names - The relative path name of the sub sub DataDirectory.
Returns:
The sub sub DataDirectory at the end of the path names.
Throws: IllegalArgumentException
some name not known.
Throws: ISException
Some name is known but does not correspond to a DataDirectory created using newDataDirectory, or the DataDirectory could not be looked up.

listDataDirectories

```
public abstract  DataDirectory[] listDataDirectories() throws  ISException
```

List all sub DataDirectories.

Returns:
The sub DataDirectories.
Throws: ISException
The list could not be created.

remove

```
public abstract void remove(String name) throws IllegalArgumentException,  ISException
```

Delete an object from the DataDirectory. If the name corresponds to a sub DataDirectory which is not empty, deletion fails.

Parameters:
name - The name of the object to destroy.
Throws: IllegalArgumentException
The named object could not be found.
Throws: ISException
The Object could not be deleted.

remove

```
public abstract void remove(String name,
                            boolean recurse) throws IllegalArgumentException,  ISException
```

Delete an object from the DataDirectory. If the name corresponds to a sub DataDirectory the sub DataDirectory and recurse is true, all its contents are removed. If recurse is false and the DataDirectory is not empty, deletion fails.

Parameters:
name - The name of the object to destroy.
recurse - If true, attempt to recursively delete subdirectories' contents.
Throws: IllegalArgumentException
The named object could not be found.
Throws: ISException
The Object could not be deleted.

3.5 Interface UK.co.ansa.ispace.StringMapper

public interface **StringMapper**

StringMapper is used by FSDirectories. Arbitrary Java Strings cannot generally be used as file names; but applications want to use them. Implementations of StringMapper map Java Strings to file names and back. mapStringToFSName must be a bijection (defined for every java string, and one to one). Its inverse, mapFSNameToString, cannot be a function. The suggested range for mapStringToFSName is strings over the set: { '!','#','$','%','&','\'','(',')', '+',',','-',
';','=','@','[',']','^','_','`','{','}', '0','1','2','3','4','5', '6','7','8','9','a','b','c','d','e','f','g','h','i','j','k','l',
'm','n','o','p','q','r','s','t','u','v','w','x','y','z'} possibly without the punctuation.

mapStringToFSName

```
public abstract String mapStringToFSName(String name)
```

Choose a file system name given an arbitary Java String.

Parameters:
name - A Java string which will be mapped to a file name.
Returns:
A file name under the mapping.

mapFSNameToString

```
public abstract String mapFSNameToString(String fsname) throws  ISException
```

Find the Java string which mapped to the given file name.

Parameters:
fsname - A file name.
Returns:
The Java string which maps to file name.
Throws: ISException

when file name cannot be revere mapped to a String.

newFileSuffix

```
public abstract String newFileSuffix()
```

Suggest a suffix for temporary file names which cannot conflict with file names generated by the mapping. E.g. ".new", "#" assuming no Java string maps to file names ".new" or "~".

Returns:
A suffix string which cannot confict with a mapped string.

oldFileSuffix

```
public abstract String oldFileSuffix()
```

Suggest a suffix for backup file names which cannot conflict with file names generated by the mapping. E.g. ".old", "~" assuming no Java string maps to file names ".old" or "~".

Returns:
A suffix string which cannot confict with a mapped string.

3.6 Class UK.co.ansa.ispace.ISException

public class **ISException**
extends FlexiNetException

A tag to allow matching of all IS exceptions in one go. Note FlexiNet exceptions may also need to be caught.

See Also:
FlexiNetException, FlexiNetRuntimeException

Constructors

ISException

```
public ISException(String s)
```

ISException

```
public ISException()
```

[1] FollowMe Annex A.1 Project Program Part 2, "Description of the RTD Project".
[2] FollowMe Work package B 'Mobile Object Workbench' documentation.
[3] Bloom, T., 1979. “Evaluating Synchronisation Mechanisms”, in *Proceedings of the Seventh Symposium on Operating System Principles (Pacific Grove, California)*, from *ACM SIGOPS*, pp. 24-32, ACM Press (New York, New York).
[4] ANSA's FlexiNet.