EXHIBIT B

(Part 3 of 3)

maid," and so on. Items with textual properties, such as a file comment or the contents of a text file that include any of these will be gathered into the collection.

Synchronizing with Changes

From time to time, the user may want to ensure that the data he/she is viewing from within MFS is consistent with the data from the outside sources, such as the file system. The user may then choose the Update command to tell MFS that it should synchronize its mirrored data structures with those elsewhere (such as the file system, the email servers, and so on). MFS will then update all information stored in the catalog and object store as required; the user will simply see the changes in the items (e.g. new items in a folder window; changed names; etc.) as they are discovered.

Overview of Specific Functional Modules Enabling the Inventive System

An exemplary MFS-enabled computer system to implement the inventive information management features comprises the following elements from which one skilled in the art will be enabled to make and use MFS:

- a computer system as described above including a CPU, one or more input peripheral a display device and an operating system;
- an object store data structure, in which data is stored persistently on a device such as a disk drive;
- a set of foundation objects that define items, containers, and collections, and which may be refined for particular uses;
- a catalog data structure in which foundation objects and their properties are maintained, using the object store for reading and writing low-level data;
- a set of consistency maintenance threads that manage information flow through the system, comprising at least one of each of: an updater, which is responsible for maintaining correct metadata for objects; a notifier, which manages dependency relationships between objects; a classifier, which assigns objects to containers and collections based on their property values; and a synchronizer, which is responsible for writing changed metadata back to the object store.
- a display and layout system, consisting of window management routines; scenes for displaying groups of objects; figures for each object's display; forms for defining figure layout of properties; and a set of views for displaying various types of content data; and
- a set of domains, which define objects and behaviors for different information-management tasks such as personal information management (contacts, appointments, and so on); file management (files, folders, documents, and so on); and also define scanners and matchers, which are responsible for scanning external data sources, creating and updating reference objects for each of the external objects that will be managed within MFS.

FIG. **32** describes in overview the communications between these modules. The object store (**3201**) sets and gets values, communicating with the synchronizer (**3202**). The catalog (**3203**) reads and writes values through the synchronizer to the catalog's property B-Trees when values change. The updater (**3204**) determines what the values of properties should be for various objects, and is notified by the notifier (**3205**) when the synchronizer (**3202**) writes changes to the object store (**3201**). The classifier (**3206**) determines what collections objects should be in, and runs when the notifier (**3205**) tells it of changes. It then writes new values for object's container list causing the catalog (**3203**) to write back the container list through the synchronizer (**3202**) and finally to the object store (**3201**).

The Object Store

The fundamental storage mechanism for MFS architecture is an object-oriented database, or the object store, that provides permanent and temporary storage facilities for low-level objects. This object store is capable of saving and restoring the complete state of any object, thus providing a persistent repository of the user's information.

Implementation. Object classes are registered with the object store at program initialization time, in order to inform the object store of the classes of objects that may be created by reading from the store. When an object is requested, the object store looks up the class of the object, which is noted in the data header of the object in the store, and requests that the object be created by the class.

A class whose instances can be stored in the object store provide six basic operations: Initialize, StreamIn, StreamOut, StreamLength, Reference, and Finalize.

Initialize is called by the object store after the object is read into memory to allow it to perform any one-time setup that is required.

StreamIn, StreamOut, StreamLength are functions that are called to ask the object to create a flattened representation of the object's information. This may include references to other objects, values, or raw data. These operations are called by the object store when creating an object to initialize the object's state from a stream of data read from the store, or to transform the object into a flat stream of data for writing out to the store. In this way each object class specifies the particular information that must be written in order that the object may be completely recreated at a future time.

Reference is called by the object store to traverse an object's reference tree. If a given object has references to any other objects, Reference must be defined to provide access to these references. This is used to attach recursively all objects that are referenced by a given object when the main object is attached to the object store. Similarly, when an object is detached the object store detaches all objects referenced by it that are referenced by that object. This operation allows the object store to determine where references to other objects occur within a given object, to allow objects within the store to contain other objects as parts.

Finalize is called just before the object is written out to the object store. It allows the object to perform any final cleanup before the object ceases to exist in memory.

Objects are attached to the object store to allow them to be stored persistently within the object store, and are detached from the object store when they are no longer persistent. Objects have a UID that is unique in the object store that never changes during the life of the program. Objects may be referenced and loaded from the object store by using this UID. The object store automatically calls the correct constructor for creating an object given an object's UID by looking up the object's class, which also resides in the object store.

All storable objects maintain a reference count for memory management. Objects that are attached to the object store may be written to the object store and removed from memory when their reference counts reach one (e.g. are only referenced by the object store). Objects that are not attached to the object store are reclaimed when their reference counts reach zero. Circular references are not detected nor managed in any way. A special smart pointer structure keeps track of when objects are being used in the program, and increments and decrements the reference count as needed. This structure maintains a pointer to the object and a reference count when

the object is in memory, and a UID and pointer to the current object store when the object has not yet been loaded.

Foundation Objects

An MFS Object is something that can be organized, sorted, searched for, and otherwise manipulated by the user in MFS. MFS Objects represent entities that encapsulate a given kind of information: email messages, mailboxes, image files, text documents, and so on. Objects have intrinsic data and type (e.g. an object may be an email message) and also have attached property values.

Reference Objects. Reference objects are mapped from the external world by the creation of an identifier, the UUID that uniquely specifies a given external object. Each object type is responsible for the creation of the UID. By way of example, a file may create a UUID by using the file system's file ID or inode, combined with the volume's creation date. This allows a fast and reliable mapping between an external entity and one stored within MFS; this is needed, for example, when a file has changed on disk and MFS needs to find the internal representation to update the object's properties.

Containers. Containers group objects together. There are many different kinds of containers: disk volume, folder, and collection, by way of example. Each kind of container has different properties: a folder groups objects together physically, and a collection groups objects together logically, based on the user's specification.

Each object maintains a set of the containers in which it appears as a property of the object, called pContainers; similarly, all containers maintain a set of objects that appear in the container, called pObjects. Other properties are computed from these, such as pObjectCount, which is computed from the pObject property. These properties define some of the basic link-metadata stored for all objects in the MFS.

Collections. Collections group objects together logically, rather than physically, as folders in file systems do. Rather than specifying where an object is (e.g. in the folder named "Leslie's Finances"), collections allow the user to specify which objects should be grouped together in a variety of ways. Collections are containers, like folders or directories, in that they can be open-ed to display their contents, but they differ in that they:

- contain objects from a variety of locations;
- contain objects of a variety of types (e.g. not just files or folders);
- can have objects manually categorized by being added to the collection, or removed from the collection, without moving the original objects themselves;
- can display dynamic, changing contents, updated in real time, based on a working set;
- can automatically collect objects based on a specification, similar to a database query, by which objects are selected by Boolean combinations of property terms and operators;
- and can also automatically collect objects based on a key phrase search of the objects' textual properties, including its contents.

By way of example, a collection may have files from both the local file system and email messages fetched from a server; images from a digital camera may be manually classified by the user into named collections such as "Summer Vacation" and "Kids"; different sizes of digital images can be automatically classified as they are added to the working set into "4×6" and "8×10" collections by specifying different width and height queries for the given collections; and a collection for objects that have something to do with Scandinavia might have a set of key phrases defined that include the names of Scandinavian countries, cities, and geographical areas (See FIG. **15**).

Like containers, collections have the link metadata property pObjects that is a set of reference objects that belong to the collection. Collections also have additional link-metadata properties called pInsiders and pOutsiders, and use additional link metadata properties in reference objects called pInclusions and pExclusions.

PInsiders is a property that contains the set of reference objects that always belong to the collection, regardless of any other collection specification. Similarly, pInclusions is the set of collections to which a given reference object belongs, again, regardless of any other action by a collection to select the object. This permits manual inclusion (categorization) of objects by MFS into collections that persists despite any automatic collection processes, such as collecting objects by metadata or key phrase query. Manually categorizing an object by adding it specifically to a given collection results in the following changes to the link-metadata:

- The object's pInclusions and pContainers properties are modified, to insert the given collection to each set;
- The collection's pInsiders and pObjects properties are similarly modified, to insert the given object into each set.

Automatic classification of objects into collections is done when the collection's pQuery property, which defines the Boolean metadata specification for objects belonging to the collection, is set to a non-empty value by the user. Setting the pKeyPhrases property also triggers automatic classification by key phrases. This begins the classification process:

1) First, the collection's pObjects property is invalidated. This adds the collection to the Updater thread, which then requests the collection to update its metadata.
2) The collection determines that its pObjects property is invalid. The collection asks the catalog to perform the query on the contents of the catalog, returning a set of reference objects that match the query.
3) If the pKeyPhrases property is not empty, then the catalog also returns a set of reference objects whose textual properties (e.g. name, textual contents, etc.) contain one or more of the key phrases.
4) The union of these two sets of objects is compared to the current set of objects that is the value of the pObjects property. This comparison returns a set of added objects, removed objects, and objects that persisted in the collection.
5) For each added object, the object's pContainers property is updated by inserting the collection, and the collection's pObjects property is updated by inserting the object from the set;
6) For each removed object, the object's pContainers property is updated by removing the collection, and the collection's pObjects property is updated by removing the object from the set;
7) By setting the pObjects property for the collection, and the pContainers properties for the added and removed objects, the catalog creates notify events for each affected object; these events are handled by the notifier, causing affected windows to redraw as required.

In this way, the link metadata is updated so that both object metadata specifies container membership, and containers (in this case, collections) have metadata specifying the objects that belong to them.

Collections respond to changes in the environment by adding and removing objects as needed to satisfy their specification. This occurs in real time as objects change their proper-

ties (e.g. their names or textual content), as new objects are created or added to the working set, or as objects are deleted or removed from the working set. For example, if the user adds a comment to a file object and there exists a collection that specifies a key phrase that occurs in that object's comment, then the object will be immediately added to the collection. This is done by the classifier thread, described below.

For each object to be reclassified, all collection specifications are evaluated, resulting in a new set of collections for the changed object. For key phrases, the classifier can return all matching collections in one pass: key phrases are compiled into a graph, with terminal nodes listing all matching collections. Novel use of the Aho-Corasick algorithm allows text to be scanned efficiently, returning all matching collections into which the object will be classified. Then, the following MFS process occurs:

1) The set of collections is compared to the object's pContainers property, resulting in three subsets: added, removed, and persistent objects. The pInclusions set is also inserted to the added set, and the pExclusions set inserted to the removed set, to ensure that manual classifications are taken into account;
2) For each added container, the object adds itself to the collection in the manner described above: updating its pContainers property by inserting the collection, and updating the collection's pObjects property by inserting itself;
3) For each removed container, the object removes itself from the collection in the manner described above; updating its pContainers property by removing the collection, and updating the collection's pObjects property by removing itself.
4) Setting these properties causes the catalog to enqueue events on the notifier, which then causes windows to be appropriately updated (e.g. the contents of collection windows).

These processes will be described in more detail below.

The Catalog

An object may have an arbitrary number of property values attached to it in the form of MFS metadata. Property values can be textual, date, numeric, Boolean, type, or image values, among others. The catalog manages the definition of metadata (e.g. the property names and types), and the linking of objects to their property values.

Implementation. The catalog database structure stores an object's properties by providing a property object that contains a B-Tree to store the property values. The property object is stored in the object store and maintains certain information such as the property name, whether the property is a sortable or searchable property, the order of sorting, the property's data type, whether changing the property should notify other objects, the B-Tree itself, and so on. The object store provides a function to retrieve an object by name; thus, if the property pModificationDate is required, the object store is called to retrieve the object by providing the name "Modification Date," which is the property object itself. The property B-Tree data structure is also stored in the object store, and maps the object's UID to the property value for that object. In this way a new property can be added at any time, simply by creating a new property object and corresponding B-Tree in the object store. New property values for an existing property are also easily added, by first finding the correct property object and B-Tree, and then by inserting a value for an object's unique ID into that B-Tree.

Values are flattened into streams of data to be stored in the B-Tree. The value can have any length; the B-Tree node is variable length depending on the lengths of the values stored within the node.

As object MFS metadata is written through the catalog, the catalog maintains a value cache, mapping objects and properties to property values, as well as a change set that maps objects to a set of properties that have been changed. The values in the value cache are eventually written back to the property B-Trees via the synchronizer, by using the change set to determine which values need to be written.

The catalog is also responsible for notifying other parts of the MFS system of changes in objects via the consistency maintenance processes. When an object's property is written, the value is compared with the value as stored in the catalog. If the value is different, an event is created and posted to a notification queue described later in this document.

Consistency Maintenance

MFS provides a sophisticated architecture for maintaining object property values and information displays correct by supporting a threaded dataflow mechanism that processes events. In particular, the catalog provides change notification, such that when objects change their property values by setting them in the catalog, a process is started to tell all potential users of that object of the change. For example, when an object is added to or removed from a collection, all objects that are affected by that change are notified, in particular the collection's window.

Objects may depend on the property values of other objects. In the process of updating one object, others may be notified of the change; and when a property value changes, dependents are notified and specific dependent properties are then made invalid, to be updated by the updater thread.

For example, the physical size of a folder depends on the physical size of all the objects contained in that folder. Those objects may be files or other folders. The folder is thus a dependent of all the items in the folder, and thus is notified if any of them change so that it may recompute as required.

In this way, any changes to objects can be tracked and values propagated to dependent objects. For example, in a personal finance application, the values of checks written must be taken into account when balancing the checkbook; reconciling the checkbook involves propagating values from reconciled checks to the current balance.

Implementation details. The consistency maintenance process is a composite process by which objects are created; their properties computed and set; their collections determined through classification; their dependents notified of the changes; and finally, deleted when no longer used.

There are four separate processes that communicate between one another and provide distinct services: the updater, the synchronizer, the notifier, and the classifier. Each process communicates with the others by means of an event queue: a queue of events describing tasks to handle in order. Events on the event queue specify the information needed by each process to perform the required function.

Objects maintain a set of properties in an update set that need to be refetched from the original source (via the domain) or recomputed. When this set is changed due to the object invalidating an individual property through the Invalidate function, the object asks the catalog to add the object to the updater's queue.

Values are stored temporarily in a value cache, keyed by property and UID. Periodically the cache is synchronized with the value trees stored in the object store. At this time a notification event is queued on the notifier thread.

The notifier thread's job is to tell interested listeners which objects have been changed, and which properties of those objects. Listeners include dependent objects (e.g. containers may want to know that they have to invalidate their physical sizes if any of their contained objects had changed size) and user interface elements (windows displaying object information).

The Updater. The updater is the process by which invalid object properties are computed and new values set for future retrieval. The updater walks through the update queue and tells each object to update its properties. This is a two-part process, involving two functions: Fetch and Compute, as follows:

Fetch, causes the object to find the needed property values from their original sources. These are considered concrete properties in that there is a direct one-to-one correspondence between the property value for the object and a value stored elsewhere in the operating system. For example, if a file object's name is invalid, Fetch will ask the file's domain to get the filename from the file system directly; and

Compute, updates properties that are derived from the concrete properties. For example, a derived property called pFullName might be the concatenation of pFirstName and pLastName in a contact record; or, the physical size of a folder might be the sum of the sizes of the objects within the folder.

During the Fetch and Compute methods, the object will call SetValue(property, value) on the properties for which it has determined values. Set Value tells the catalog that the property for this object has the given value, and the catalog will store it away.

In the process of storing the value for the object, the catalog determines whether the object actually changed the value; if the value being set was the same as the previous value, then nothing occurs. If the value did in fact change, the property is added to an update set maintained for that object. More specifically, the updater performs the following procedure, illustrated in FIG. **33**:

1) First, the updater retrieves an update event from the updater event queue (**3301**). The update event record consists of an object specifier and a set of properties that require updating for that object: the invalid set.
2) Next, the updater forwards the invalid set to the object (**3302**), and requests that the object Fetch the given properties (**3303**). It is the object's responsibility to know how to do this, since for each type of object this procedure may be different. Then the updater requests the object Compute derived properties (**3304**) that may be based on the properties fetched (e.g. the physical size of a folder is derived from the sum of the sizes of each object in the folder).
3) During the Fetch and Compute procedures, the object being updated will set the property values that were requested (**3305**). In setting the value for the given object and property, the catalog stores the object and its new property and value in the synchronizer's data structure (typically a hash table) (**3306**), and then updates a change set (**3307**): a set of objects and associated properties that have been changed. This change set will be referenced by the synchronizer later in the process.

The Synchronizer. The synchronizer is the process by which objects' updated property values are written back to the object store. On a periodic interval, the synchronizer will perform the following procedure, illustrated in FIG. **34**:

For each object in the catalog's change set (**3401**), the synchronizer will walk through the value cache (**3402**), where it will fetch the current (old) value (**3403**) and new (cached) value (**3404**) compare the object's property value with the current value in the property B-Tree (**3405**). If the value is different, the new value will replace the old (**3406**) and the property will remain in the value cache. If the value is the same, the property will be removed from the change set (**3407**).

When the object's values have been synchronized with the values stored in the object store, a notifier object changed event is created and added to the notifier's event queue (**3408**, **3409**). This event includes the object specifier and the set of properties for which new values were written. Note that properties whose values were the same were removed from the set, so only properties with new values remained in the set and are in the notification.

The Notifier. The notifier is the process by which other processes, and objects, are notified of additions, changes, and removal of objects in the system. Concurrently, the notifier performs the following procedure, illustrated in FIG. **35**:

1) The next notifier event is removed from the notifier event queue (**3501**). There are three different types of notifier events: an object added event, which is queued by another thread when an object is first created in the system; an object changed event, which is enqueued by the synchronizer when an object's property values are changed; and an object removed event, which is enqueued when an object is removed from the system.
2) The notifier then broadcasts the event to all listeners by going through the subscriber set (**3502**), copying the event (**3503**), and enqueuing the event on the subscriber's queue (**3504**). One of the typical subscribers is the classifier (**3505**), which receives events and determines whether the object needs to be reclassified.
3) If the event is an object changed event (**3506**), then the object itself is subsequently notified of the change (**3507**). This is done by computing dependent properties (**3508**), invalidating them (**3509**), and queuing an update event (**3510**) to the updater (**3511**) so that they are refetched and recomputed. The effect of this is that the object can then notify its own dependents, such as figures depicting the object on the screen, or other objects whose properties are dependent on properties of the original object.

Dependent properties allow the object to invalidate certain properties that are computed from other properties that had changed; for example, a container may invalidate its physical size property if its object set property had changed. Invalidating one or more of an object's properties in this way will result in the object in turn being placed on the updater's event queue as described, for further processing to compute the desired properties.

Other parts of the MFS system can subscribe to the notifier thread at any time. For example, when an object is displaying properties in a window, the window management system in MFS temporary subscribes to the notifier thread so that it may update the window contents when the object changes, such as the object's containers.

The Classifier. The classifier is the process by which objects are added to and removed from collections based on their property values. Concurrently, the classifier performs the following procedure, described in FIG. **36**:

The next classifier event is removed from the classifier's event queue (**3601**). Since the classifier is subscribed to the notifier, it receives notifier events when objects are added to the system; when objects change their property values; and when objects are removed. In each of these cases the classifier

is responsible for determining the set of containers (folders, collections, or other specific containers) to which the object belongs.

1) If the event is an object added event (**3602**), then the classifier determines the set of containers to which the object belongs and creates an added set and an empty removed set (**3603**).
2) If the event is an object changed event (**3604**), then the classifier performs the following procedure (**3605**). First, the existing container set is retrieved. Next, the object is classified, resulting in a set of containers to which the object should now belong. Next, these two sets are compared, resulting in the added set, which includes containers to which the object should be added; and the removed set, containers from which the object should be removed.
3) If the event is an object removed event (**3606**), then the classifier creates an empty added set, and sets the removed set to the object's container set (**3607**).
4) Finally, the object is added to the containers in the added set (**3608**), and removed from containers in the removed set (**3609**).

In this way, each object referenced in the classifier event queue ends up in the correct set of containers that select for its current property values.

Classification of a Single Object

The classifier determines container membership for an object through the process described in FIG. **37**:

Initially, the result set, which contains the set of containers to which the object should belong, is set to empty (**3701**). Then the classifier asks the object's source (e.g. the File or EMail domain) to perform an initial classification of the object (**3702**), resulting in a new result set. The Files domain, by way of example, would add a file object to its enclosing folder.

Objects can be classified into collections by specifying in each collection a list of key phrases whose occurrence in an object means that the object should be referenced in the collection. A collection may have many key phrases, and the same key phrases may be specified in many different collections. The MFS-configured computer system's key phrase classifier performs a single-pass, multiplex sorting of a given object into an unlimited number of collections based on the pKeyPhrases properties defined in those collections and the textual content of the object.

The classifier runs through each text property in the object (**3703**) and for each property goes through each key phrase in the classifier (**3704**) determining whether the key phrase exists in the property's value text (**3705**). If so, it adds the entire set of collections associated with the key phrase, since a single key phrase may be listed by multiple collections (**3706**).

The key phrase classifier is based on a novel use and implementation of the Aho-Corasick string search algorithm. The classifier begins by scanning each collection when MFS is launched, and adds each key phrase to the Aho-Corasick finite state machine. At the terminal nodes for each key phrase is a list of collections that specify that phrase; as the collections are scanned and each key phrase is added, the list of collections at each key phrase is kept up to date with all collections that specify it.

All objects maintain a list of collections in which they occur. Classification is accomplished by scanning the text body of an object using the Aho-Corasick algorithm. When a key phrase is found within the text, the list of collections for that phrase is fetched from the finite-state machine and united to a final (list or set) of collections in which this object should appear. When the entire text body has been scanned, the final set is compared with the initial set. For collections that appear in both sets, nothing is done. For collections that appear in the initial set but not the final set, the object is removed from those collections. For collections that appear in the final set but not in the initial set, the object is added to those collections. Finally, the object's collection set becomes the final set, reflecting that object's membership in those sets.

Next, the classifier goes through each collection (**3707**) and determines if the object satisfies the query specified by the collection (**3708**). If so, the collection is inserted into the result set (**3709**).

Finally, the result set is returned (**3710**) and the object is placed into the collections listed in same.

View by Reference

A container C (folder, collection, or any other container) is viewed by reference using the following process.

1) An empty result set R is created.
2) For each object in the container C, the set of collections to which that object belongs is added to the result set.
3) A new container V representing the reference view, is created.
4) For each collection A in the result set R, a new proxy collection P is created, whereby the contents of the proxy collection P is simply the objects in C that are also in the collection A; this is done through a set intersection of the collection A and the container C. Generally, this proxy collection is simply defined by an MFS metadata query on P which states that the contents of the proxy collection are the intersection of the contents of collection A and container C.
5) The final container V that is the reference view now contains a set of proxy collections, each of which holds a subset of the original objects in C.

The reference view may then be further refined by choosing a proxy collection P's contents (a subset of C's) to view by reference. This is done as follows:

1) The reference view V adds P to its prefix set.
2) V replaces its proxy collections with new proxy collections, using the same process as above, but with one difference: each proxy collection's MFS metadata query now states that the contents of the proxy collection are the intersection of the contents of collection A, container C, and all the collections in V's prefix set.

In this way, a view of the "Today" collection, which shows the objects modified today, can more easily be viewed by reference, which shows that (for example) the Received email collection was changed today, as well as the Documentation project. Clicking on the Received proxy collection in the view reveals email objects received today; further refining by Received will show the collections in which email was received today: typically a list of the contacts from whom email was received.

Display and Layout

MFS provides an architecture for display and layout of objects in a variety of ways. Individual objects are viewed in content viewers defined by each domain, which is responsible for the individual object types. Viewing containers of objects (e.g. collections or folders in icon or list views) is based on three classes of objects: forms, figures, and scenes. A unique type of list view is implemented by MFS's sticky paths mechanism.

Content Viewers. For each specific type of object, a content viewer is available for viewing the actual object data. By way of example, contact objects are displayed in a window showing first and last names, addresses, and so on. Email messages have their own specific viewer, with the standard to, from, and

body panes within the window. Text files or notes are displayed in a standard text-editing window.

In the case of content viewers that provide text fields, key phrases can be highlighted automatically when examining the object's contents and provide a hyperlink to the defining collection automatically. If multiple collections specify a given key phrase, the popup menu will list all collections that do so, allowing the user to choose which collection should be opened.

An object that has been classified into several different containers will indicate this in the Information window, where all of the containers are listed and may be opened.

Forms. A form is a 2-dimensional layout of property values of a single object. For example, a standard icon view includes two fields: the icon property situated and centered above the name property. A list view form will include a left to right arrangement of the object's icon, name, and additional properties as required by the display. Forms are used by figures to determine the appearance of the object in the window.

Figures. A figure is a drawable entity representing an object. Figures are linked to forms, which define how the figure should be drawn. Figures also provide the ability to be highlighted when clicked; to have their properties edited directly, such as the name in an icon view; and to be dragged from one place to another within the MFS interface. Figures are arranged within scenes, which determine where each figure should be located.

Scenes. A scene is an arrangement of figures in 2 or 2½ dimensions (2½ dimensions include a representation of depth). The scene is generally responsible for determining the form the figures within the scene should take; thus, MFS defines a small icon scene whereby the form defines a small rectangle for the icon property and a rectangle to its right for the name property; a large icon scene with the icon rectangle above the name rectangle; variants on the previous; and a preview scene where the object's preview property is drawn within a slide frame, along with the object's name, size and modification date; and various list views, among others.

The scene is also responsible for locating each figure within the scene based on certain conditions. For example, in the small icon scene the objects are sorted by a given property (chosen by the user) and then laid out top-down, left-to-right in the window; scrolling to the right shows additional figures. The large icon scene lays out the figures left-to-right, then top-down in the window; scrolling down shows additional figures.

The user typically chooses which scene to display objects in, by selecting an item in the View menu. Unique and specialized scenes may be defined by domains as well, if needed.

Sticky Paths

Sticky paths are a unique way of displaying hierarchies of objects within MFS. Often hierarchies of objects are displayed in a sort of outline view, whereby objects are listed in some order (typically alphabetical), and sub-objects that are contained in other objects may be displayed or hidden at the user's control. An object that contains other objects in this way may be either expanded (displaying its sub-objects) or collapsed (hiding them). Each object has a depth, a numeric value that describes how far down the hierarchy it exists; in particular, how many nodes down the hierarchy tree from the root. Objects at the same depth are known as siblings. The depth determines how far the object is indented to the right in the outline display.

When an object is collapsed, any object that contains others is indicated in some way with a clickable region, typically a symbol such as a + sign or a triangle, that may be clicked. Clicking on the region expands the object by displaying those objects which are contained within below the object and indented to the right by a specific amount, due to their depth being one greater than the depth of the parent object. Other objects that were at the same level as the object being expanded are moved down the display by the amount needed by the expanded object.

Objects within an expanded object may in turn be expanded, resulting in several levels of expanded objects and multiple indentations.

The path to an object is defined as the name of the object itself, prefixed by the names of the nested containers in which the object exists in outermost order. For example, if an object E is contained in an object D, and in turn D is contained in C, and C is contained in B, the path to the object E is generally described as B:C:D:E.

In a highly-hierarchical display with many objects that do not fit on a single screen, the user must scroll the hierarchy display in order to see objects lower down on the list. In particular, if some objects have many sub-objects which are in turn expanded to show their respective sub-objects, it is quite easy to forget what part of the hierarchy one is looking at, i.e., where the user is on the path, since the enclosing objects have scrolled off the top of the display.

Sticky paths are a mechanism by which a scrollable outline of this form is displayed in two dynamic parts: a path area and a scrollable area. Sticky paths provide the user with a constant awareness of his location in the hierarchy by:

1) constantly displaying the current path to the topmost item in the scrollable area above the scrollable area, and dynamically updating the path as the objects are scrolled up and down;
2) dynamically resizing the scrollable area to accommodate the path display.

Implementation Details. The sticky path scrolling mechanism is implemented, by way of example, in the following pseudocode:

1) Get the old path frame from the current display.
2) Get a list of container objects that comprise the path to the topmost figure in the outline. Do this by determining the object at the top of the scrolling region, and then walking up the outline item's parent tree until there are no more parents.
3) Set the path display by starting at the top of the list and drawing each parent in turn, indented appropriate to the parent's depth.
4) Get the size of the new path frame.
5) Determine the difference between the heights of the old and new frames.
6) If the difference is zero, then the size of the path hasn't changed, and the bits can be scrolled within the scrolling region.
7) If there is a difference in height, then we first adjust the size and location of the scrolling region based on the amount of the change.
8) If the difference is greater than zero (e.g. the path is smaller than it had been previously), then we don't scroll, but we do have to refresh the topmost figures of the area that was vacated when the path region was made smaller.
9) If the difference is less than zero (e.g. the path is larger than it had been previously) then the resizing of the scrolling region is sufficient, and no scrolling is necessary since the topmost figure in the scrolling region will have been moved up into the vacated path area.

In this way, the current path to the topmost item is always visible.

Domains

Domains define an "area of expertise" for MFS. Typical domains include personal information management (appointments and contacts); file management (folders, files, disks); image file management (also known as digital asset management); and email, among others. Domains provide a way to extend MFS's capabilities and functions by leveraging MFS's architecture in new ways.

A domain is responsible for implementing the following procedures:

- Registration of new object classes and properties for same;
- Creation of new objects of specific classes when needed;
- Creating and managing UUID mappings between reference objects and external data;
- Adding metadata properties to objects;
- Basic classification of objects by class and property values;
- Updating of object metadata in response to changes in the operating environment; and
- Performing basic operations on behalf of objects that the domain manages.

The following describes these procedures for domains defining file management, email handling, music organization, personal information management, image management, and organization by time.

The File Domain. This domain registers new object classes for disks, folders, and files. The properties that are registered include file and folder names; creation date; modification date; physical size; and permissions, among others.

The domain is also responsible for scanning folder and file objects, and resolving changes with the objects on the disk as the disk contents change. For example, when a folder's modification date differs from its corresponding object in MFS, the domain compares the folder's contents with the contents of the folder object, and creates or deletes file and folder objects in the folder object as required to match exactly the contents of the disk folder. Similarly, if a file's modification date changes, its corresponding file object is updated with the current filename, modification date, size, and so on in order to mirror exactly the file's property values.

Certain file types are handled specially by this domain. In particular, application and document files must have the appropriate icons associated with them, and behaviors such as opening a document must be defined to launch the correct application.

This domain provides window layouts for information about files and folders, and utilizes built-in MFS windows for displaying folder contents. Window layouts for certain types of files also are supported, including text files and clippings.

All sources provide the ability to scan external data to add information to the Working Set, and match the external data to update MFS's internal reference objects as the external data changes. By way of example we describe the File domain's implementation details of these two processes.

Implementation Details. The File domain is notified of files to add to the Working Set as follows. The user drags a folder to MFS's workspace window; this causes a reference object is created for the folder by the specific source handling the folder; in this case, File source, which is responsible for all file system objects. Next, a scanner thread is created with the reference object as a parameter. This thread performs the following functions, in order:

1) Traverse: A procedure recursively descends the folder hierarchy, creating data entries that are stored in an array. Each entry contains a file system specifier that represents the file; a depth in the folder hierarchy; and a flag that determines whether the file system specifier is for a folder or a file. The array is then sorted, deepest objects in the tree first (so that files within a folder are created before the folder is).
2) Annotate: Once this array has been populated, the entries are annotated with metadata that can be efficiently fetched "en masse", such as file and folder comments.
3) Create: The entries are fetched one by one from the array. For each entry, a reference object is created with the entry's information (e.g. the file specifier and any metadata that was previously fetched and added to the catalog). A new array of reference objects is created.
4) Classify: Each object in the array is then classified by examining its metadata and determining in which collections the object belongs, based on the collections' specifications. Every collection that is modified (e.g. that has received a new object through the classification process) is added to yet a third array for notification.
5) Notify: Finally, each collection that participated in the classify step is notified that it has been changed. This typically results in the collection updating dependent property values (e.g. count of contained objects), which are then updated in a separate thread.

Then, the folder reference object is then added to the HFS source's working set, which is the set of all folders that MFS should manage. The Workspace window is then updated, since the working set property determines which folders are shown in the window.

Because the file system data that the File source tracks changes over time, the File source has the ability to match these changes and propagate them throughout the catalog and object store. This is done as follows:

1) During the match process, MFS compares its stored information against the source's versions of the same information. If there is an indication of difference between a reference object in MFS and the actual external object (by noting a changed modification date, for example), MFS invalidates the reference object's metadata.
2) Once the metadata has been invalidated for all objects suspected of being changed externally, MFS puts each object on the updater queue.
3) While items exist on the updater queue, the updater does the following:
4) Takes the next item off the queue
5) Tells the object to update itself. This, in turn, causes the object to go to the source to determine what the true values should be for each of the invalid property values. Once the values have been retrieved from the source (by asking the file system for file metadata such as name, modification date, etc.), the new values are set in the catalog and the property is validated.
6) The catalog then creates notification events based on the objects and values that were set, and enqueues them on the notifier queue
7) The notifier goes through each event, telling all of the object's dependents (any containers to which the object belongs as a member, as well as any other dependent objects) that it has changed the given properties.
8) Those objects, in turn, determine whether any of their properties need invalidation. For example, if a file's size has changed, the containing folder's size property needs to be updated, since it is dependent on the sizes of all the files within the folder.

Using these two processes, scan and match, the File domain creates a Mirrored Object System within MFS that exactly represents the file and folder hierarchy that the Domain tracks, regardless of external changes.

The EMail Domain. This domain registers new object classes for mailboxes, which describe servers and passwords for retrieving mail; signatures, for signing messages; and email messages themselves. Properties for these objects include server names, addresses, and passwords, for mailboxes; a name and text string, for signatures; and the full suite of email properties for messages, including From, Date, Subject, and message body.

The domain is responsible for communicating with mail servers for both sending and receiving email; creating outgoing message objects; and for creating new received message objects as they are downloaded from the server. Attachments are handled by communication with the File domain for creating and linking to file objects as they are downloaded to disk. Finally, window layouts are provided for outgoing and incoming email messages; mailboxes; and signatures. Behaviors such as sending messages, forwarding, replying, and so on are also supported by the domain.

The Music Domain. This domain, a client of the File domain, registers a new object for a music file, generally in the MP3 format. Properties registered include the track's title, artist, album, genre, and comments.

The domain is responsible for extracting the property values from the file using the ID3 tags that are embedded in the file, and for setting the properties in the catalog for the object. The domain also creates predefined collections for titles by album, and albums by artist, based on the music files that are handled by the system.

The domain may also provide a music player for all files in a given container; in this way the user can play all the tracks on a given album, or tracks grouped together in an arbitrary collection.

Finally, a window layout is provided for information about the music track, showing album, artist, title, and so on in addition to the generic file information such as filename, file size, and so on.

The Personal Information Domain. This domain registers new objects for contacts, notes, appointments, projects, events, and tasks. Specific properties are registered for each object: for contacts, the standard list of contact information such as first and last name, email address, phone, and so on; for notes, the note text; for appointments, the date and time, repeat interval, description, contacts, and so on; for projects, the project name and description; for events, the event name, date, and so on; and for tasks, the task description, priority, and the like.

The domain is responsible for scanning and matching with system-level address book databases, creating, deleting, and modifying contacts as required. Depending on the domain implementation, it may also match with other PIM databases such as Outlook and Palm in the same way, by creating mirror objects in MFS for each object in the target database.

The domain creates predefined collections for all notes, all contacts, and so forth, as well as predefined collections for each contact that collect all objects that reference the contact's name.

Finally, window layouts are provided for each type of object to allow display and editing of the object's data.

The Image Management Domain. This domain, a client of the File domain, registers new objects for file types that store images. Properties registered include image resolution, width, height, depth, and the like.

The domain is responsible for extracting the attribute from the file and attributing the MFS object appropriately, as well as for reading the file data and displaying it as an image within an MFS window.

The domain creates predefined collections for all images of various types (e.g. JPEG files, GIF files, Photoshop files, etc.), as well as a single Images collection for all images.

Finally, a window layout is provided for the display of image files.

The Time Domain. This domain provides no new object classes, but creates and maintains a set of dynamic collections that are based on relative time. For example, the domain creates and keeps up to date a Today collection that changes each day; similarly, Last Week, Last Month, Last Year, and collections created on demand by the user are handled by this domain.

The domain provides a root collection called Time; in this collection the various other collections are created and stored. A collection for the current year contains collections for each month of the year to date; in turn, each month has collections for each week of the month.

Finally, the domain provides window layouts for unique views of objects by time, including a Timeline view where documents are arranged by a date property within a given range, among others. This domain is particularly adaptable to use in the legal field where extensive docketing systems are required.

Additional Domains. It should be understood, as will be evident to one skilled in the art that a wide variety of other Domains may be added, e.g., Location, Space, Event, Symptom, Cause of Action, etc., as the Domains described above are merely exemplary and not limiting of the scope, nature and character of domains that the inventive system can employ. The Domains can be special in nature, as noted by the Symptom for those in the medical profession, and Cause of Action for those in the legal services profession. Another Domain could be "MO" for modus operandi, for use by investigators and police, which can be set to automatically group in collections sets of facts (objects representing text narratives of criminal activities, images and the like) based on similar MOs. This automatic building of collections could be a powerful tool in the criminal justice field. Likewise, engineering professions can build collections with similarities in data trends or values, e.g., temperatures, materials values, velocity, concentrations of chemical components, etc. for analytic purposes.

INDUSTRIAL APPLICABILITY

The inventive data storage organization, archiving, retrieval and presentation system architecture and technology can be used in a wide variety of applications; the primary being desktop file organization and server data management. The inventive system is remarkably robust, yet is a relatively small application program that can function with any type of Operating System: Microsoft Windows, Windows NT, Windows 2000, and Windows XP; Apple Macintosh OS 9 and OSX; BSD Unix, HP-UX, Sun Solaris, Linux, and the like. Currently the inventive technology is preferably implemented in its current best mode in a form that is executable on the Apple Macintosh OS9 and OSX operating systems.

As to Desktop Organization, the invention is useful as an improved desktop organization application for all types of data, limited only by the domains that can be conceived-of. Domains may be easily created to extend the MFS capabilities to new areas of expertise.

As to File Organization, similar to the Apple Macintosh Finder or Microsoft Windows Explorer, the inventive MFS system provides basic disk navigation and display features. In addition, the File domain allows additional properties to be specified for files, including: a due date; a file species (e.g. an

application, a bookmark, a text-readable file, an image file, a font file, etc.); and a file path. Folders have additional properties that are maintained automatically: the size of the contents of the folder; and the depth of the folder from the root directory of the disk, among others.

As to Image Cataloging, image asset management is easily implemented as an extension of the MFS File domain. A domain that can extract relevant information from images found on disk (e.g. size, type of image, colors used, resolution, and so on) is created as a representative object within MFS that has the given properties. Users can then view and select the images that satisfy certain collection criteria. Comments on the objects can also be used to describe and/or define the content of the images, and collections can be created to organize all images based on their described/defined content.

As to Personal Information Management the inventive MFS system is useful for scheduling, organization and tracking of appointments, contacts, events, notes, projects, and tasks as typical kinds of objects that are defined by the PIM (Personal Information Management) domain. What makes the inventive system of particular interest to industry is that the PIM Domain functionality provides a new feature: the ability to organize information objects by person and by project.

As to EMail, a basic embodiment of the inventive MFS system provides basic EMail client services. The objects that the basic EMail domain defines include: mailboxes, email messages, and signatures. This can be expanded to include all types of e-business trust services, including: signature legalization; payment transfers; electronic record retention and verification; electronic filing of documents, including formal/legal documents, applications, forms and the like; privacy and confidentiality guards; identity verification and authentication; access guards; time verification and authentication, including times of sending, acceptance, receipt and performance; and the like.

As to EMail Notification, the inventive MFS system permits the user to watch all postings and create emails describing when a message will be classified into a given collection (automatic forum). This results from the functionality of the classifier; as it is data-driven, it classifies all collections simultaneously. When a key phrase is found it lists all collections from all users that specify that phrase. The phrases can be defined in Boolean search terms for the broadest inclusive categorization and inclusion.

As to Custom Desktop Client, the inventive MFS system includes, by way of example, the useful functionality of Portfolio management by which the user, as a customer of a financial services site, such as Marketocracy.com, can communicate with the site server to synchronize data, e.g., to provide automatic portfolio updates, stock quote display, market, sector and stock performance graphing, and the like.

As to Personal Finance, a personal finance domain may be implemented in MFS to provide objects for checks, checkbooks, receipts, invoices, stocks, funds, and so on. The normal accounting principles apply; the value propagation feature mechanism is used to ensure that the checkbook properties (e.g. balance) are computed from the check properties (e.g. amount of check). Stocks and funds can be updated over the network and their values displayed in MFS.

It should be understood that various modifications within the scope of this invention can be made by one of ordinary skill in the art without departing from the spirit thereof and without undue experimentation. It is apparent to those skilled in the art that many features and functionalities of the inventive MFS can be enabled and realized separately. For example, sticky paths or drag and drop link creation can each be coded in a separate application or applet, or can be provided as incremental upgrades to a program, or as plug-ins or add-ons to existing other productivity, organizational or creativity application programs or applets. That is, MFS can include less than all the dozen or more features described above, and, conversely, MFS is extensible to include additional features and is adaptable to co-operatingly interact and enhance other applications. This invention is therefore to be defined by the scope of the appended claims as broadly as the prior art will permit, and in view of the specification if need be, including a full range of current and future equivalents thereof.

The invention claimed is:

1. A computer data processing system comprising:
- a) a computer-readable memory configured to store informational object organized in a hierarchy;
- b) a display configured to display information of at least a portion of said hierarchy;
- c) an applications program with code configured to:
  - i) render visible information of at least one of said informational objects, with dynamic updating, in a sticky path portion of said display;
  - ii) expand said information of at least one of said informational objects; and
  - iii) collapse said information of at least one of said informational objects.

2. The computer data processing system of claim **1** wherein the computer-readable memory is located in more than one physical location in a distributed environment.

3. The computer data processing system of claim **1** wherein said applications program with code is further configured to permit a user to create said hierarchy.

4. The computer data processing system of claim **1** wherein said applications program with code is further configured to permit a user to add an informational object to said hierarchy.

5. The computer data processing system of claim **1** wherein said applications program with code is further configured to permit a user to remove an informational object from said hierarchy.

6. A computer data processing system comprising:
- a) a computer-readable memory configured to store informational objects in at least one group;
- b) a display configured to display information of at least two informational objects of said group;
- c) an applications program with code configured to:
  - ii) render visible information of at least one of said informational objects, with dynamic updating, in a sticky path portion of said display;
  - iii) expand said information of at least one of said informational objects;
  - iv) collapse said information of at least one of said informational objects.

7. The computer data processing system of claim **6** wherein the computer-readable memory is located in more than one physical location in a distributed environment.

8. A computer data processing system having a central processing unit configured with an integrated computer control software system for the management of information relating to multiple levels of objects organized in a multi-branch expandable hierarchy structure, wherein each branch of said hierarchy includes an identifier indicating that branch and its contents, said data processing system comprising:
- a) a computer readable memory including a storage structure for storing information relating to said objects selected from at least one of said objects, object metadata, object names, object identifiers, or object location paths;

b) a computer display connected to said memory for displaying in a view, said objects, said object metadata, and said objects identifiers from said storage structure;

c) a computer-user interface for selectively displaying at least a portion of said expandable hierarchy in said view; and

d) an applications program having code processed by said central processing unit so as, upon user scrolling of said hierarchy structure, to continuously render visible, with dynamic updating, at least one sticky path display region of said view, said code being processed so as to provide the function of at least one of:

i) a portion of said hierarchy structure is displayed in said view;

ii) individual levels of the hierarchy structures may be expanded or collapsed through user action; and

iii) upon user scrolling of said hierarchy structure displayed within said view, said sticky-path display region of the view dynamically updates to visibly display a current portion of a hierarchical path to said expandable hierarchy structure.

9. A computer data processing system as in claim **8** wherein said applications program processes so as to display said dynamically-updated current region of said path or nesting as currently-visible contents in said view immediately following the beginning of said view.

10. A computer data processing system as in claim **8** wherein, upon user scrolling of said view, said applications program processes so as to provide the function of at least one of:

a) when an identifier of an expanded branch of the hierarchy begins to scroll off said view, said branch identifier is displayed in said sticky-path region along with an identifier of its parent, if one exists, otherwise said branch identifier is displayed by itself in said sticky path region; and

b) when the last object contained in said expanded branch begins to scroll off said view, said branch identifier is removed from said sticky-path region of the view, while identifiers of each parent branch remain in said sticky path region, until the last object of each branch is in turn scrolled off of said view.

11. A computer data processing system as in claim **10** wherein said applications program processes so as to cause said sticky-path region to be displayed during scrolling, and thereafter is removed.

12. A computer data processing system as in claim **10** wherein said applications program processes so as to cause said sticky-path region to selectively appear when a user selects a specific object or target in the view or elsewhere in the display.

13. A computer data processing system as in claim **10** wherein said applications program processes so as to cause said sticky-path region to be transparent, translucent, or opaque.

14. A computer data processing system as in claim **10** wherein said applications program processes so as to cause said sticky-path region to appear and disappear depending on scrolling action.

15. A computer data processing system as in claim **10** wherein said applications program processes so as to cause said sticky-path region to appear and disappear depending on the location in said view of a pointer.

16. A computer data processing system as in claim **10** wherein said applications program processes so as to cause said sticky-path region to appear at a margin of the view or, if not at a margin, within the view's content area.

17. A computer data processing system as in claim **10** wherein said applications program processes so as to cause user selection of a specific region in a branch identifier to collapse said branch, adjusting the display as needed.

18. A computer data processing system as in claim **10** wherein said applications program processes so as to provide for user selection of a specific region in a branch identifier to cause scrolling to occur such that the first object in said branch appears adjacent to said identifier.

19. A computer data processing system as in claim **8** wherein said hierarchy objects represent at least one of textual or graphical indicia in at least one of block-structured or meta-language coding.

20. A computer data processing system as in claim **19** wherein hierarchy objects are represented in at least one of:

a) a textual outline wherein different hierarchy depths are indicated by whitespace;

b) a graphical outline wherein different hierarchy depths are indicated by location in a view; and

c) a graphical outline wherein different hierarchy depths are indicated by spatial enclosure.

* * * * *