An Object Oriented Database Technology for UNIX System Management

Steve Landers (scl@fs.com.au)
Michael Selig (mike@fs.com.au)

Functional Software
P.O. Box 266
Willetton WA 6155
Australia

Abstract

Since the first release of its products, Functional Software has been using relational database technology to implement UNIX system management. This provides the ability to implement maintainable software modules more quickly, and to provide flexibility so that they can be easily customised by the end user. It also facilitates abstraction of most system management tasks, so that the user need not be aware of the "nitty gritty", such as the differences in system file formats between different flavours of UNIX.

With the gradual acceptance of Open Systems within commercial data centres comes a new set of problems. Implementing concepts like "centralised control with delegated responsibility for certain tasks" in a manageable and flexible way requires many of the concepts of Object Oriented paradigm. This paper describes the current directions that Functional Software is taking in this area.

The Data Centre

The concept of the data centre is well known within the field of commercial computing. A data centre is characterised by:

- a relatively small number of large central hosts
- centralised staff dedicated to managing and operating the computing equipment
- segregation of responsibility between staff
- an emphasis on standards and procedures
- a formal audit review cycle

With the gradual acceptance of Open Systems by commercial Data Centres comes the perception that UNIX is ill-equipped to deal with the demands of such an environment. Unfortunately, this perception often reflects reality.

The UNIX System Administrator

Many UNIX sites are administered by a small group of highly skilled, highly motivated individuals who:

- share all tasks from operations to management
- adhere to unwritten standards
- make extensive use of undocumented procedures
- almost always use a super-user account to perform system administration
- have never seen a systems auditor

There is little automation and delegation - consequently they spend a large percentage of their work day performing routine and repetitive tasks.

Whilst such installations are usually efficiently run, the organisation is not resilient to change - one person leaving the organisation can have a major impact.

In addition, Open Systems bring their own problems to the commercial data centre:

- multiple vendors, each with their own UNIX variants
- multiple architectures (even from the same vendor)
- a wide range of scale, from small(ish) hosts dedicated to specific applications to large general purpose hosts
- interfaces to proprietary systems.

Yet despite these problems, the combination of Open Systems with more traditional Data Centre management practice has much to offer a commercial computing organisation.

The Challenge of UNIX Data Centre Management

The challenge is to provide a system management framework that facilitates the control and standardisation without losing the flexibility and productivity more typical of a UNIX environment.

Such a framework must support existing practice whilst encouraging better practice. Often, system managers are so busy "cutting down trees" that they don't have time to "sharpen their axe". A framework or product that comes with a significant, albeit temporary, drop in productivity will often not be used. What is needed is the ability to leverage existing investment in system management procedures whilst standardising and documenting them.

Change must be accommodated, but with consistency and control. To remain competitive in the 90's, a business must react quickly to changing circumstances. The data centre staff in a large organisation should not be worried about implementation issues on different UNIX variants, but rather work with abstractions that hide the underlying system management complexities.

Because UNIX administration staff are often highly skilled and motivated, a system management framework should free them from routine and repetitive tasks and allow them to concentrate on higher value tasks. If an organisation is already sensitive to change,

every effort should be made to provide staff with job satisfaction and an appropriate challenge. Innovative and creative solutions to technically challenging problems should be encouraged, but in a consistent and controlled manner.

The system management framework should also accommodate different management styles, which may vary according to the demands of the site or even the whim of the system administrator. Some may prefer to use particular UNIX utilities to perform backups, others may prefer command line rather than a menu interface. All should be able to choose that which is most appropriate in a particular circumstance.

An Objective Look at Things

We could summarise these requirements by saying that a system management environment should be:

- flexible
- extensible
- consistent
- portable

These are the benefits that are often attributed to the object oriented paradigm. It is therefore not surprising that a number of organisations have been looking at system management from an object oriented perspective. Such groups include The Open Software Foundation, UNIX International System Management Working Group and the IEEE 1003.7 Committee on System Administration.

The recommendations (and in some instances, the products) coming from such organisations are remarkably similar in that they:

- focus more on networks of workstations than central Data Centres
- emphasise GUI interfaces over screen and command line interfaces
- are more concerned with simplifying the use of facilities rather than automation
- emphasise compiled languages (usually C++) or proprietary interpreted languages rather than making use of the UNIX shell
- are limited in their support of existing equipment and practice

As has been discussed in a previous paper, the system management philosophy adopted by Functional Software was to make maximum use of existing UNIX facilities and standards, in line with the Software Tools philosophy embodied in the UNIX programming paradigm [1].

Functional Software has been delivering a commercial quality system management product (THE ADMIN MANAGER) since 1989. This product is built using a set of technology comprising:

- The Functional Toolset - a suite of user interface tools that allow full screen applications to be constructed using the Bourne shell as a programming language

- The Functional Database - a suite of database tools that allow full relational databases to be built using flat UNIX files for tables, and the shell as a programming language

Using this database strategy allows various system management data to be normalised and stored in tables. For example, the device file for a particular tape drive is stored once rather than being spread throughout a number of backup shell scripts.

Commands to perform various system management tasks may be stored as data in tables, and various parameters interpolated at run time. The effect is to allow existing procedures to be used as part of an auditable, structured system management environment, hopefully freeing time to allow better practice to be implemented.

Full integration is provided between the shell, the user interface tools, the database tools and any commands specified by users. This is achieved by having a one-to-one correspondence between database columns, screen fields and shell environment variables.

Whilst THE ADMIN MANAGER meets many of the demands of commercial data centres, and has been successfully implemented in over one hundred large commercial installations in Australia, nevertheless there is potential for further improvement in the areas of:

- centralised control with distributed management
- consistent command line and menu based interfaces
- "pluggable" re-implementation of system management functions

With this in mind, Functional Software chose to look again at the system management issue from an object oriented perspective.

What is Object Orientation?

An object comprises data (attributes) and the operations (methods) that may be performed on the data.

For example, consider the UNIX file system table. If we regard this as an object then the operations would be mount, unmount, check, etc. The data would have different structure, location, and even operations depending on the UNIX variant. Other objects might include printers, tape drives and users.

Implementing such objects under the Functional Software environment turned out to be relatively straight forward. The key decision was to layer the object framework upon the existing relational database and user interface toolset. This allowed the existing benefits to be realised plus facilitated the additional ones resulting from an object oriented view of system management.

Representing Data

Within this framework, data are represented by tables. Each object has a table to hold its primary data (the primary table), and there may be additional tables to store local or implementation data. For example, a file system table will have (at least) the Device,

Filesystem and Options columns.

The Functional Database uses two UNIX files to represent a table - one to hold the data and the other the data dictionary information. The data dictionary stores a description of the columns within the data file, their format, the key columns and the physical location of the data file.

UNIX System files with single character separated fields (such as the password and group files) can be directly supported, as can white space separated files such as the hosts file and file system table.

Representing Methods

One of these additional tables is a per-object method table. For example, a file system object might have the following method table:

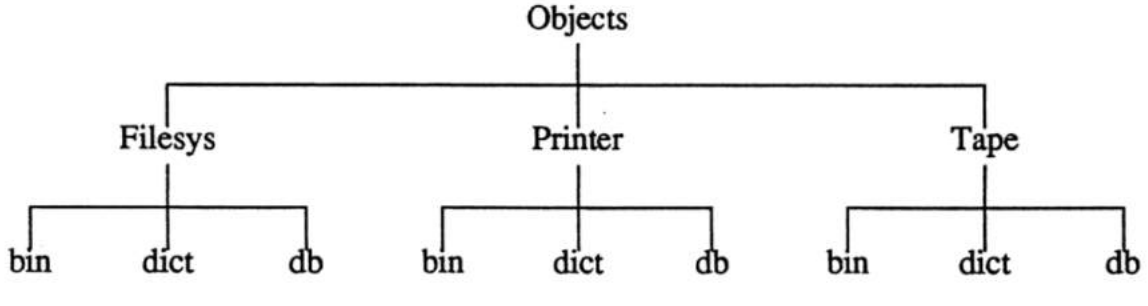
Method	Commands	Required
details	df $Filesystem	
mount	mount $Device $Filesystem	Filesystem
unmount	umount $Filesystem	Filesystem
check	fsck $Device	Device
display	db_scroll filesys	
add	db_maint -a filesys	Filesystem
change	db_maint -c filesys	Filesystem
remove	db_maint -r filesys	Filesystem

Note that some of the methods perform system management operations, whilst others manipulate the object data (eg: add, change). Note also how the column values for a particular row in the filesys table may be referred to by the corresponding shell variable.

The Required column specifies the list of columns required for a particular method. Typically this is just the key column, but it may be any column or the keyword ALL. This allows error checking to be performed when the method is run.

Supporting Multiple Objects

The UNIX filesystem hierarchy is used to support multiple objects, each with its own local executables, data and dictionary information.



There is a global object table, that specifies the name of each object, plus the root directory of each object's hierarchy. Each object has the following directories:

- a **bin** directory to store any executables local to the object
- a **db** directory to store any data local to the object
- a **dict** directory to store any data dictionary definitions local to the object

Command Line Access

Command line access to the operations of an object was implemented via a command called, naturally enough, Object. Object is invoked as follows:

```
Object objname [key ...] [col=val ...]
```

where *objname* is the name of the object, *key* is a value used to select a row from the object table, and *col=val* specifies a list of column variable assignments. These override the values selected using the key. For example:

```
Object filesys mount /usr1
```

When run, Object looks in the global object table to find the root directory of the filesystem object. It then prepends the bin directory to the PATH variable, and also prepends the db and dict directories to the appropriate database path variables. It then performs the following tasks:

- if a key has been specified, the corresponding row from the objects data table is selected and the corresponding column variables set so that any references within the method commands may be satisfied.
- any column variables specified on command line are set to override the values from the table. If the method was maintaining data then it would be necessary to specify any changes or additions on the command line, but it is also useful for one-off changes such as mounting an alternative device:

  ```
  Object mount /usr1 Device=/dev/rfd0
  ```

- the commands corresponding to the method are executed.

One very quickly tires of typing Object, and this was overcome by providing an (optional) link from the Object command to a file of the same name as each object. To implement, an extra column was added to the global object table to specify whether the link should be made. The Object command was modified to use its invocation name as the object name. Then we could say:

```
filesys mount /usr1
```

or

```
printer offline pr1
```

Screen Based Interfaces

The Functional Toolset contains a set of user interface tools that implement most of the

components of a full-screen interface [1]. The developer describes the contents of a screen (eg: a menu or form) and at run-time the tool decides on a layout suitable for the user's screen or display.

This approach has the advantage that consistent screen and GUI versions of the same application can be supported by providing curses and X-windows versions of a small number of re-usable tools. If both Motif and OpenLook version of the tools are available, then the user can choose the preferred one using a single environment variable. A major advantage of this approach is that the screen and GUI interfaces interpret the same source (ie: the description files).

Supporting screen based interfaces to objects involved modifying the method table so that there is a one-to-one correspondence between the methods and the options in a menu. The method table was expanded to include the additional information needed to generate a menu description file. Columns were added to indicate:

- the security profile needed for the method to be visible on the menu
- whether to prompt the user to choose a row before executing the method commands
- if a "please wait" message should be issued before the method is executed
- if there should be a pause (ie: prompt the user to press a character) after the method and before returning to the menu
- if stderr should be captured and displayed in a pop-up window
- any help messages
- any commands that the method is conditional on - if the commands are not found then the menu option is not displayed.

For example, the following is the mount row in the file system method table:

Column	**Value**
Operation	mount
Description	Mount a file system
Command	mount $Device $Filesystem
Security	["$SEC_filesys" = true]
Choose	yes
Wait	no
Pause	yes
Window	error
Help	Mount (bring online) a file system
Ext-help	man mount
Conditional	mount

Standard table maintenance procedures can be used to maintain the method table, and then the corresponding menu description regenerated. Once this is done, invoking the Object

command with the -m (menu) flag causes the object menu to be displayed. For example, to display the operations permissible on a file system, you would invoke:

```
filesys -m
```

If you had a security profile that allowed the data to be maintained you would also see the add/change and remove options.

Mixing Command Line and Screen Input

A user has the flexibility to use either

- all screen based input
- mixed command line and screen based
- all command line input

For example, invoking:

```
printer -m
```

would display the operation permissible on a printer (ie: all screen based input). However, if you wanted to disable a printer and were not sure of the printer name, then you could invoke:

```
printer -m disable
```

This would run the disable method of the printer object by invoking the corresponding option in the disable menu. If the choose column was set, then you would be able to choose a printer from the printer table using a screen based choose tool.

If you wanted to use all command line to disable a printer (called printer_1) you would invoke:

```
printer disable printer_1
```

Note that specifying the key (printer_1) was equivalent to choosing the corresponding row from the printer table. If the key parameter was omitted, then you would get an error message because the printer name would be a required column.

The advantage of being able to mix command line and screen based input is that the user can be prompted if there are any required data that has not been supplied.

Configurability

Often administrators want to support new methods, or change the implementation of existing methods. This can be done by modifying the contents of the method table for a particular object using standard table maintenance procedures. The interfaces to the object remain the same: the Object command and the corresponding object menu.

Virtual Tables

Although the object framework described thus far realises many objectives we set, it needs to be expanded to totally accommodate the differences between UNIX variants.

The Functional Database supports virtual tables, where the data comes from a command rather than a file. This was used to support the various version of the UNIX ps (process status) command in a consistent manner. Instead of specifying a file, the DBTABFILE variable specifies a command string that runs the appropriate ps command and extracts column values. For example, the ps table definition on SunOS contains:

```
DBTABFILE="!ps -axgu | dbcols -s 1 10 14 1 8 ...
```

whilst on SVR4 it contains:

```
DBTABFILE="!ps -efl | dbcols -s 1 14 19 6 13 ...
```

db_cols is a command that skips any heading lines, extracts character fields specified by position ranges, strips leading and trailing white space, and returns tab separated fields.

This approach has the advantage of hiding the differences between UNIX versions within the data dictionary. Shell scripts can be written that select from the ps table using named columns without regard for the underlying UNIX system differences.

UNIX (Per)versions

However, to fully support UNIX variants we must support the differences in

- structure (eg: the format of the file system table)
- methods (eg: the operations permissible on a particular tape drive or printer type)

For example, the file system table structure on a particular UNIX variant is generally one of the following:

Level 7	**SVR3**	**BSD**	**SVR4**
Device	Device	Device	Device
Filesystem	Filesystem	Filesystem	FSCK_device
Options	Options	FStype	Filesystem
	FStype	Options	FStype
		Frequency	Pass
		Pass	Automount
			Options

Similarly, some tape drives support an unload operation (eg: many 9 track tape drives) whereas others support re-tension operations (eg: QIC cartridge tape drives).

Classes, Class Types and Instances

To accommodate the differences in structure and method, the object framework supports the notion of classes and class types.

A class is a generalised software template, roughly equivalent to what we have thus far been calling an object. For example, the UNIX file system table could be considered a class.

A class type is an alternative template or implementation of a class. For example, the file system class might have alternative class types of Level 7, SVR3, BSD and SVR4, each with its own set table definitions and data.

The definition of an object now becomes a specific instance of a class. When the object is created (ie: instantiated) the class type must be specified and the methods and structure for that class type are used.

An instance would usually represent the state of an object on a host or grouping of hosts.

Inheritance

One problem with this scheme of class types is the amount of replicated data and table definitions. For example, each class type would need a definition for the method table, plus data for the standard maintenance methods (display/add/change/remove).

One way to overcome this is to introduce the concept of inheritance, where data and structure are inherited from other classes.There are two types of inheritance needed:

- **class inheritance**, where structure and commands are inherited from another class type (eg: the method table definition)
- **object inheritance**, where data is inherited from a particular object instance (eg: the mount method is common to all file system classes)

Inheritance can be between types in the same class (eg: the mount method) or from unrelated classes (eg: the method table definition and standard maintenance methods).

The inheritance for a particular instance is specified via an inheritance table, where both class and object inheritance are explicitly specified. In addition, the location of the object to inherit must be specified. This may be done by naming the class to inherit, or using "instance" to indicate that current object. This is useful when specifying the concatenation order of inherited data.

For class inheritance, the location may be the name of a class, or of a class type (specified by class/type). For object inheritance, it may be the class or instance.

For example, consider the inheritance table for a file system object:

Table	Inheritance Type	Location
method	class	standard
method	object	instance
method	object	filesys/SVR4
method	object	filesys

This illustrates the inheritance in an instance of a SVR4 file system object. The first row says that the method table structure is inherited from a standard object that is used to hold any global definitions.

The last three rows specify the data for the method table of this file system instance. Firstly, any methods belonging to the instance, followed by any from the class type, then any from the class. This allows the administrator of this instance to add or replace methods without affecting other instances.

Also, note that the class and class type have been stated explicitly. This means that the instance will need to have its own copy of the inheritance table. We can generalise this further by changing the definition to:

Table	Inheritance Type	Location
method	class	standard
method	object	instance
method	object	$Class/$Classtype
method	object	$Class

The Object command maintains the Class, Classtype and Instance variables for the current object. This allows us to have a global generic inheritance table that will be used if the object/class/type does not have one defined. An instance inheritance table will take precedence over a class type and class inheritance tables. If none is found, then the global one is used.

It is also possible to inherit data from multiple locations. For example, we might want to manage a tape drive with a stacker unit. The class data (ie: table definitions) would still come from the standard class, but we would also want to inherit drive operations and the stacker operations relevant to the particular drive.

Table	Inheritance Type	Location
method	class	standard
method	object	instance
method	object	driveops/$Drivetype
method	object	stacker/$Stacker
method	object	standard

Drivetype and Stacker would be columns in the tape object primary data, used to indicate the tape drive operations type (eg: mt or ctape), and the stacker type.

One further comment about inheritance - the structure of any inherited table must be compatible since the data files are concatenated.

Scope and Locality

When defining tables we need to specify where the data is stored and who can see it.

The visibility of a table is called the scope, and may be one of the following:

Scope	Data Visibility
instance	private to the instance
type	visible from objects of the same class type
class	visible from objects of the same class
global	visible from all objects

The storage location of a table's data is called the locality, which may be one of the following:

Locality	Storage Location
instance	separate copy per instance
type	separate copy per class type
class	separate copy per class
global	one copy for all objects

All inheritance is based on visibility. When an object is created (eg: a particular instance of the file system class) the inheritance and scope tables are used to update the data dictionary so that any use of the tables retrieves the appropriate data.

Copy on Write Tables

When reading from a table, the data is the concatenation of all the individual tables specified by the inheritance. When writing the data, any changed or new rows are written to the location specified by the locality of the table. The main disadvantage to this scheme is that there may be several versions of each row (eg: an SVR4 specific mount, and a filesys class generic mount). This is not a problem with methods, since only the first method found will be used and the inheritance table will specify the order in which the data is concatenated.

However, there are other situations where you do not want this to occur so a modified scheme is supported. Rather than the inheritance specifying the concatenation of several files, it can be used to specify a path to look for a single data file.

The first file of the same name as the table found in the path will be used as the data file.

If the table is modified, then the new version will be written into the location specified by the tables locality.

This effectively allows the support of "copy-on-write" tables. This is particularly useful for supporting software distribution, since locally modified data will now be held separately from the release data.

Distribution

Another issue is how to handle the centralised management of objects distributed over multiple hosts. Usually, an instance of an object will represent a host or grouping of hosts. This is specified in a per class distribution table which records the instance used by each host.

For example, if we were centrally maintaining users across several machines, the user class might have a distribution table like the following:

Host	Instance
funcy	Research and Development
whiskers	Research and Development
clogg	Technical Services
quay	Technical Services
playboy	Technical Services
sporran	Marketing
haggis	Administration

Data would be maintained from a central "Management Host". When managing users (eg by maintaining user information, or disabling an account) the administrator could either use the menu or command line interface. If using menus, the administrator would first choose the instance and then proceed to select one of the options from the user menu. If using command line, the instance would have to be specified on the command line.

The method corresponding to the option would then be distributed to each host in the current instance. The distribution is implemented by generating a command line invocation and sending it to each host that uses the instance. For example, if a new user "fred" was added to the "Technical Services" instance, then commands similar to the following would be sent to clogg, quay and playboy:

```
Object user add Logname=fred Fullname="Fred Bloggs" \
          UID=100 GID=100 Home=/users/fred \
          Shell=/bin/ksh
```

This distribution strategy is built on the premise that data is accessed far more often than it is changed, therefore data replication is not a problem as long as there is a central point of update. This is more typical of a centralised data centre than a large network.

However it can be used to manage hosts that are, in turn, the management hosts of other

system management domains (such as NIS or NCS). This allows the centralised control strategy to tie together other existing technologies for distributed access to system files.

Communications Methods

Communication with a host is via any arbitrary communications method. All that must be supported is the ability to run an arbitrary command on the remote machine.

The communications method is defined in a global table that specifies:

- the commands to initiate communication with the remote host
- whether the method supports queuing (eg: uucp) or if queuing should be provided for it (eg: Berkeley rsh)
- whether the method supports interactive use (eg: Berkeley rsh) or if it can only be used in a batch mode (eg: uucp)

Against each host in the global hosts table is a column that specifies the communications method used to distribute operations to the host.

For example, the two most commonly used communications methods are the Berkeley rsh (remote shell) command and uucp. On System V hosts, the remote shell is often called remsh to avoid clashing with the restricted shell. These could be specified as follows:

Comms	Queuing	Interactive	Command
rsh	no	yes	rsh $Host "$Command"
remsh	no	yes	remsh $Host "$Command"
uucp	yes	no	uux "$Host!$Command"

Specifying communications in this way allows most communications protocols supported under UNIX to be used to distribute updates. More importantly, it enables existing technologies to be used whilst allowing for the use of new technologies such as DCE.

Delegation

It is not always reasonable to force the central management of all resources. This was overcome by allowing the responsibility for maintaining an instance to be delegated to another host. To implement, an additional column was added to the per-class instance table to specify the host that has the responsibility for maintaining the instance data.

For example, in the previous example of a user class, we might have an instance table like the following:

Instance	Class type	Management host
Research and Development	user	funcy
Technical Services	user	quay
Marketing	user	haggis
Administration	user	haggis

Depending on security requirements, changes are either propagated back to the central management host for distribution or sent directly to the other hosts who use the particular instance of the object.

Some Brief Notes on Implementation

As has been discussed, the implementation of the object framework was layered onto the existing user interface and database management tools.

Extensive use was made of the UNIX directory hierarchy to separate the structure and data for classes, class types and instances.

Access to the various directories is via a number of path variables. In addition to the usual UNIX PATH variable, these include:

DBDIR	the list of dictionary directories
DBPATH	the list of data directories
MENUDIR	the list of menu directories

A configuration file for each instance contains a series of shell variable assignments setting the various path variables. The Object command sources this file once the root directory of the object has been determined.

The framework is semi-static, in that an instance's configuration file must be re-generated if its structure changes.

Well, how big is it?

One benefit of interpreted code (ie: the use of the shell and a small number of interface and database tools) is that the size of the software is minimised. The whole of the Functional Toolset, Functional Database, and layered object management framework is less than 5 Mbytes, depending on the architecture.

Equally important, because there are a small number of tools each optimised for a particular task, the processing load on the host system is relatively small.

What has been learned

The main lesson is that object oriented is a very loose term!

It has been possible to take a small database that uses flat files, some interface and database tools, plus the Bourne shell, and come up with a development environment that provides many of the features and benefits attributed to "Object Oriented" technology. Does this make it an object oriented environment?

One point that is clear - object oriented does not necessarily mean "Object Oriented Programming Languages" or "Object Oriented Databases". It is a way of looking at problems that allows the abstraction of a problem to be isolated from the implementation.

Too often "object oriented" is equated with terms like classes, instances, instantiation, inheritance, distribution, object brokers, polymorphism, encapsulation, schema evolution, constraints, etc.

What is really of issue is that the software environment is flexible, extensible, consistent and portable - the very benefits that UNIX brought to the world of operating systems over 20 years ago.

For too long commercial users regarded UNIX as a "toy" operating system - at a time when the size of a system was considered a measure of its worth. It is ironic that as UNIX is becoming main stream many now have concerns about "Kernel Bloat", and the size and complexity of database and windowing systems.

In the world of software, big is not necessarily beautiful. Let's not make the same mistake with system management frameworks.

References

[1] Steve Landers and Michael Selig, *A Software Tools Based Application Environment*, Proceedings of the Australian UNIX Users Group AUUG '91 Conference, Sydney Australia, September 1991