Mapping Objects to Files: A UNIX File System Interface to an Object Management System

Gert Florijn, Leo Soepenberg, and Atze Dijkstra

RUU-CS-93-08
February 1993

Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Mapping Objects to Files: A UNIX File System Interface to an Object Management System

Gert Florijn, Leo Soepenberg, and Atze Dijkstra

Technical Report RUU-CS-93-08
February 1993

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924-3275

Mapping Objects to Files: A UNIX File System Interface to an Object Management System

Gert Florijn, Leo Soepenberg, Atze Dijkstra

University of Utrecht
Utrecht, the Netherlands
email: florijn@cs.ruu.nl

Abstract

This report describes the design of a UNIX file system interface for the Camera Object Management System (OMS), an extensible, fully object-oriented environment comparable to Smalltalk. The interface was developed to let the UNIX system act as a user-interface to the OMS, and to be able to run existing UNIX tools from within the OMS. Via the interface, users are presented with a file tree which in actual fact is generated from objects and attributes stored in the OMS. Manipulating the tree implies sending messages to OMS objects.

The design uses the NFS protocol for client-server communication and is based on a user-level NFS server that connects to multiple OMS processes. Essential in the architecture is that we do not impose one particular mapping from OMS concepts (objects, etc.) to a file tree. Instead, multiple of such mappings (or views) can be defined within each OMS. Furthermore, it is possible to switch dynamically between views.

In this report we discuss our design goals and describe the major components of the system. In addition, we give some examples of how object-oriented concepts in the OMS can be accomodated in a UNIX file tree, by describing some of the sample views that we have defined.

1. Introduction

From a user's perspective, the UNIX file system can be viewed as a naming service and user interface to user-level and system-level objects. Originally [Thom78], devices were the only system objects represented in the file tree. More recent work in this area, e.g. in the context of the "research" version of UNIX [Hume89] and Plan 9 from Bell Labs [Pike90, Pres91], has demonstrated how entities such as processes, windows and network connections can be successfully represented by file trees.

Controlling resources via a file system does not imply that the resources themselves are stored persistently on disk. Instead, they can be accessed using a file system interface: the state of the resource is represented in files and directories. The user accesses and controls the resource via the standard file system operations (open, read, etc). These are propagated by the user's (operating) system to a server that implements the resource.

The main advantage of using the file system as an interface to resources is uniformity. Users and programmers do not need to learn different programming interfaces or commands to access or control diverse operating system concepts. Once they know how to deal with the file system, they can control the execution of processes, manipulate windows, etc.

Project goals

This report describes our experience in developing a UNIX file system interface for a non-traditional, user-level resource: an extensible Object Management System (OMS). The OMS is a truly object-oriented system, bearing much similarity to Smalltalk [Gold85]. It is part of the Camera system [Flor92], a distributed software development environment.

Within the OMS data is represented as objects, which are instances of classes. Objects have attributes which are values of basic data types, while references among objects are modelled using relations. Functionality is modelled through methods associated with classes; methods are invoked by sending messages to objects. The OMS is extensible in that users can (dynamically) add new objects, classes and methods. Also, the OMS provides built-in derivation management for functional methods. It maintains a cache for invocations of such methods and minimizes the amount of recomputation necessary.

Our main objective in this project was to achieve a high degree of integration between an existing, unmodified UNIX environment and the Object Management System. This is reflected in the three major design goals.

Firstly, we wanted to use the the UNIX file system, and the existing suite of tools such as editors and browsers, as an end-user interface to the OMS. From this perspective it is necessary that all the concepts of the OMS (objects, classes, message passing, instantiation) can be represented as a file-tree or as operations on that tree.

Secondly, we wanted to be able to invoke (non-interactive) tools such as compilers or text-formatters from within the methods defined on OMS-classes and to let these tools fetch their data from objects stored in the OMS. Thus, the OMS should be able to present a file-tree image that is compatible with the regular UNIX file-tree, since these tools have (semi-) hard-wired assumptions about path names.

Finally, as a specialization of the second goal, we wanted to be able to let some of these tools be treated as side-effect-free functions from the viewpoint of the OMS. Only the net end-results of tool executions should be visible; creation and removal of temporary files, should not be propagated to the OMS. This would allow the built-in derivation management functionality of the OMS to be used on existing tools.

Outline of the approach

Our solution is, technically speaking, simple and straightforward. For practical reasons, we use the NFS protocol [Sand86, Inc89] as the interface between the UNIX-file system and the OMS, effectively turning the OMS in a special, user-level NFS server. Thus, each distinct OMS is mounted on a special place in the client's UNIX file-tree and the NFS operations are turned into OMS messages sent to OMS objects.

Though basically simple, the overall design contains some architectural aspects that are of interest. Instead of having a single, static mapping from OMS concepts to a UNIX file-tree, we have introduced a *view* mechanism which, in essence, provides user-level control over this mapping. Thus, by programming a view into the OMS, a user can specify how the mapping should take place. Each OMS can contain several view definitions, and multiple instances of each view can be active in the same OMS. Furthermore, the model allows client processes to dynamically switch between views.

Obviously, we have experimented with the view mechanism to see how information in the OMS can be best be presented as file trees. For example, we have introduced a view that maps the super-class relation among classes into a file tree where each class maps to a directory, while a different view provides access to the definition of a class, i.e. the definition of its attributes and methods. Another view maps any OMS object onto a directory, while the object's attributes and the methods that can be invoked on it are represented as files in that directory. Method invocation is modelled by reading the file associated with the method, while arguments can be passed by writing them in a argument file that is also associated with the method.

In order to obtain functional behaviour from tools we reuse another piece of file system technology. The Translucent File Service (TFS) [Hend88] provides the ability to isolate changes to an existing file-tree in a separate file tree. By placing a TFS instance between the UNIX client and the OMS for the tool that is invoked, we can collect the changes made by the tool in a TFS change-layer, and return the changes after the tool execution has completed. This aspect is further discussed in [Soep93].

About this report

The remainder of this report discusses the design of the file system interface in some more detail. Section 2 presents the Camera OMS and section 3 discusses our design goals. Section 4 outlines the technology involved when providing a file system interface for a resource. Section 5 discusses the basic integration of NFS and OMS, while section 6 discusses the notion of views. Section 7 describes some example views and section 8 illustrates the concepts by an annotated example. Finally, section 9 provides conclusions.

2. The Camera Object Management System

The Camera system [Flor92, Lipp92a] is a software development environment aimed at supporting cooperative work in loosely coupled infrastructures, i.e. situations where network connections are not permanently available or are too slow to allow for transparent integration.

A Camera system can host multiple development activities, each of which has its own, private workspace. These workspaces are self-contained repositories: they contain all the data and tools necessary to do the work. There are no references to anything outside the workspace from within it. Workspaces are fully isolated from one another. When working within a particular workspace a user is not affected by activities in other workspaces. Workspaces are, as a whole, subject to versioning. Distinct states of the workspace contents, called snapshots, are saved by the system for later reference [Lipp91].

The Camera system has a two-level architecture, i.e. there are two levels on which information is stored and functionality is represented. The top-level - called the Album - contains and controls the workspaces, the snapshots and other information pertaining to the development process, while the workspace (or more accurately, its current snapshot) contains the data under development. Both the Album and the workspaces have the same, object-oriented data model as defined by the Object Management System.

The Object Management System

The Camera OMS data model is truly object-oriented. Data is stored as objects which have a system generated, world-wide unique identifier. Objects are instances of classes which define the structure of objects by listing their attributes. Attributes can store values of various primitive data types (e.g. integer, longfields, symbol). Since the OMS is used as part of a development environment, objects are relatively coarse-grained entities, e.g. pieces of soure code or documentation.

Associations among objects are not modelled through object-pointers but through (n-ary) relationships. Relationships are tuples in a relation and can also contain primitive values. The functionality that manipulates and transforms the data in an OMS is represented as methods associated with classes. The basic interaction paradigm is message passing: users send messages to objects to invoke behaviour.

The OMS is extensible. Users can define new classes, relations and methods on the fly. Classes can inherit from (multiple) existing classes. Methods can be defined using an external language; in the current implementation they can be defined in Elk, a small and extensible implementation of Scheme. More dynamics can be added by using triggers – which invoke an action when certain conditions become true – and through (partially) computed relations, where a script determines which tuples make up a relation.

Finally, the OMS is modelled in itself. This means that meta-concepts such as classes, relations and methods are objects, while other meta-concepts are represented in specific relations (e.g. instance-of, method-of). This provides a uniform interface to the system: adding new definitions is done by sending messages to these meta-objects. These (meta) classes and relations are depicted in figure 1.

The basic data model for the Album and the Workspace OMS is the same. This is also reflected in the implementation. Each active Album and Workspace is a separate process using the same basic engine. However, the differences between the two levels become apparent in a number of built-in classes and relations that only exist on the Album level. They represent system concepts such as workspaces and snapshots that do not occur in the

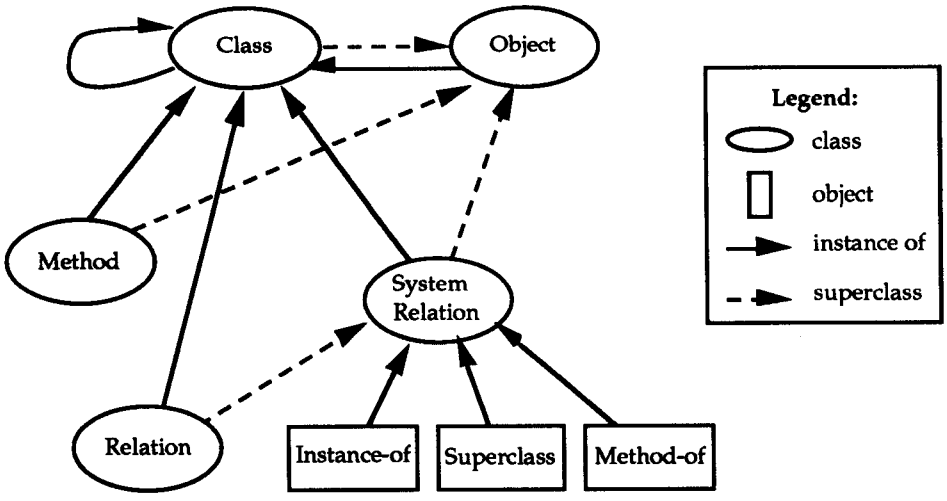

Figure 1: The basic OMS class structure

Workspace OMS. Another difference between the two levels is that the Workspace OMS provides built-in facilities for derivation management [Lipp92b].

Derivation management in the Workspace OMS

The Workspace OMS provides built-in support for derivation management. A derivation manager keeps derived values (such as the compiled form of a program source) up-to-date with the values on which these values depend (e.g. the source code, the compiler, include files, search paths, options, etc.). The main role of a derivation manager is to optimise the amount of computation involved (mostly by caching previously computed values).

In Camera, derived values are modelled as the result of invocations of functional (i.e. side effect-free) methods. The derivation manager caches - in a hidden cache - the results of such invocations and also tracks all dependencies for a given derived value. This is done by logging all attributes that are accessed and other functional methods that are invoked during the computation. The dependency information is made visible in a system-maintained (read-only) relation. Whenever some part of the Workspace OMS is changed, some cached derived values may have become inconsistent. Re-computation of such values and further propagation of changes (either on demand or immediately) is then necessary. Propagation stops whenever the end-result of a computation is the same.

There are several advantages of combining the derivation management with the (implementation of) the Workspace OMS. First, users do not have to specify derivations and dependencies explicitly in some separate file as is the case with tools like Make [Feld79]. In the OMS one only specifies the methods that compute derived values and associates these with the relevant class. Since dependency tracking is implicit and complete[1] the system guarantees that derived values are always consistent with all their inputs. There is no chance of forgetting implicit dependencies on things like options and tools. Furthermore, propagation of changes does not depend on criteria such as the last-modification time. If a re-computation results in the same outcome (as when a comment is changed in a source file), propagation stops. Finally, making the dependencies visible in a system-maintained relation allows for new applications such as risk analysis for changes.

---

[1]Dependency tracking is complete because the OMS is self-contained .

3. Why a File System interface?

Although the OMS is conceptually very simple - basically it is just a virtual machine with one instruction: send a message to an object - it provides a lot of diverse functionality. Thus, providing a proper user-interface for the system is not an easy task.

Of course it is possible to build a customized interface. In fact we have developed several of these. The simplest one is comparable with the UNIX "shell". It is a Scheme interpreter extended with functions to connect to OMS processes and to send messages to objects in these OMS's. The second one is an Emacs mode, which connects to an OMS process and allows the user to navigate through the OMS and browse and modify its contents. Finally, we have designed a graphical browser for the OMS [Scha92].

It would be an interesting alternative however, if users could use an existing and familiar model (i.e. the UNIX file system) to access the OMS. This would allow them, in principle at least, to use their existing user-interface repertoire, including command interpreters, editors and graphical browsers, to control and interact with the OMS. Needless to say, this would make the Camera system much easier to adopt, less complex, and applicable in different environments with different local conventions.

Invoking tools from methods

Since the OMS (especially the workspace OMS) is used as a repository in a software development environment, it stores the usual software artifacts such as source-code and documentation. Obviously, much of the functionality needed to manipulate and transform such artifacts (compilers, formatters, linkers) is already present in the host-environment.

Of course, it would be possible to rewrite these tools to into methods on the appropriate classes within the OMS. We decided however that it would be more practical if we could invoke such existing tools from within methods. In principle this could be done by executing the tools in their normal UNIX environment and using wrappers to get the results from UNIX back to the calling method. However, this would require detailed insight into all tools used in this way. Furthermore, it would not make it possible to use the built-in derivation management functions in the OMS for these tools, since access on dependents (e.g. reading an include file) would not go through the OMS. By having a file-system interface, however, we do not have these disadvantages. Tools access a UNIX tree that is actually stored within the OMS.

The goal to apply the OMS derivation management to existing tools like compilers, complicates things slightly further. These tools must not only fetch the data they use from the OMS, but, viewed from the OMS, they must also have functional behaviour. Many tools, such as a compiler, are conceptually functional: they take a file as input and produce a result. However, viewed at the file system level they are not functional, because they create and remove temporary files. In order to make these tools functional from the OMS perspective, these temporary files should remain hidden. Only the net-results of a computation, together with the references to non-temporary data, should be seen.

4. File System Interfaces

Implementing a file system interface for a resource requires several pieces of functionality (see also figure 2). First we need a mechanism to redirect or map resource specific file operations to functions implemented by a server for that resource. This implies that the client system must be able to determine which elements in the file tree belong to a particular resource and to decide which operations should be propagated. In principle, these are orthogonal design decisions that could be treated distinctly. Within the UNIX environment, these issues are (commonly) solved by extending the basic model of multiple file systems and mounting with a virtual file system interface and a file system switch [Wein84].

The basic unit of interest is that of a file system, a self-contained, acyclic and rooted graph. By mounting the root of one file system on top of a particular point in another file system we

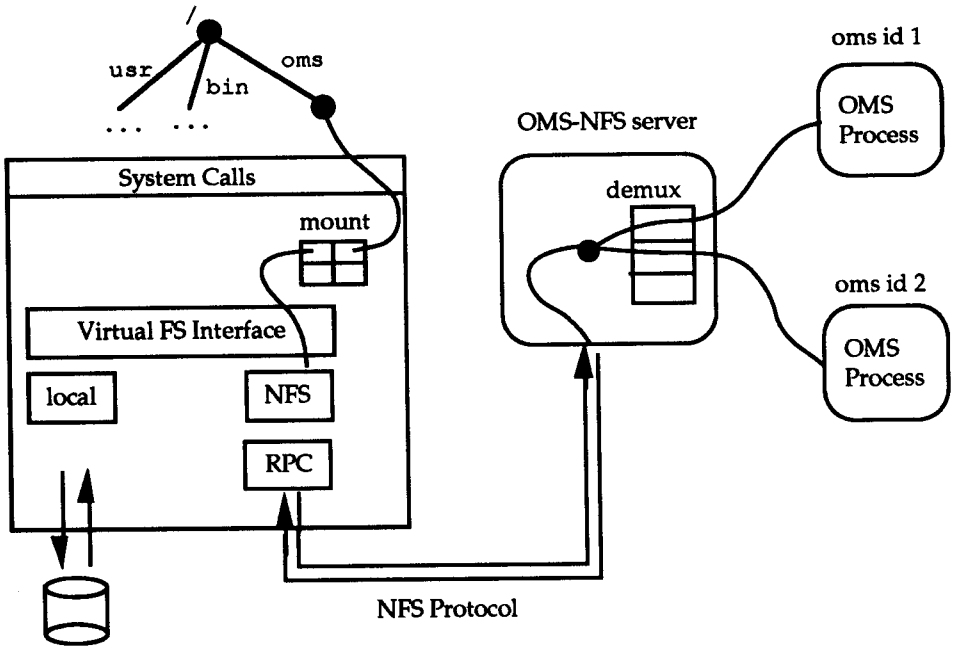

Figure 2: Architecture of the OMS-NFS Server Model

construct one name space consisting of multiple file systems. The virtual file system interface is a sort of abstract data type describing the expected functionality of a file system (implementation). All the relevant system calls are mapped to this interface. The file system switch allows us to provide distinct implementations of this interface for different kinds of file systems. One obvious implementation is to map the operations of the file-system interface to disc. Another interesting implementation is to propagate the operations to a (possibly remote) server using a particular protocol. By connecting an instance of this file system type to a particular server and by mounting the file name space provided by that server into the client's existing name space, the client's users can access the server via a sub-graph of their file name space.

While most current UNIX systems follow this general model, there are substantial differences on aspects such as the design of the virtual file system interface and the interaction model and protocol used between client and server. A discussion of these issues falls outside the scope of this report, although some points are mentioned in the conclusions.

NFS

For practical reasons - such as availability and the possibility (in principle) to create a user-level server - we decided to adopt Sun's NFS [Sand86] model to implement the interface to the OMS. This means that the client and server interact via remote procedure calls using Sun's RPC architecture [Inc88]. The operations that the server needs to implement are described in the NFS protocol specification [Inc89].

A key aspect of the NFS model is that it is stateless. The server does not have to maintain any information about client connections. Each NFS call contains all the information needed to perform the operation. Another important point is that file-names are parsed and translated on the client system. This implies that servers never carry out lookup operations on complete path names, but only resolve one level at a time.

The primary concept in the working of the NFS protocol is that of a file handle. It is given by the server to the client to identify a particular node in the server's file system. The client does not interpret file handles. Instead it passes the file handle to the server when it wants certain operations to be performed on that node. When mounting the server's file system, the handle of the root node is returned.

5. Connecting NFS to OMS

The result of the choice for NFS is that an OMS process must act as a (user-level) NFS server. It must handle the incoming remote procedure calls, and unpack and map these into message sends to objects in an OMS.

Obviously, a machine can run many different OMS processes at any given moment in time. The objective is that each of these OMS's can be present in the client's file name space, and that the operations on a tree corresponding to a particular OMS process are in fact propagated to that server. Unfortunately, NFS does not provide this indirection; all NFS operations for mount points originating from a particular machine are propagated to the same communication port on that machine.

This problem can be solved by separating the NFS server functionality from the actual OMS processes, and by letting the server do the de-multiplexing, using an identifier for OMS processes encoded in file handles. This is illustrated in figure 2. The OMS-NFS server is mounted on a particular point in the file tree (say, the directory `/oms`) and it propagates operations on its sub-trees to the OMS processes.

In order to do the de-multiplexing, the NFS-server must have a mapping from names for OMS processes (used by clients as names occurring in the directory `/oms`) to their communication ports. This can be done by letting the OMS processes register with the OMS-NFS server, who stores their names and communication ports in a table[2]. The available OMS processes can now appear as sub-directories of `/oms` using the names maintained by the OMS-NFS server. Furthermore, an identifier for the OMS process must be encoded in the file handles of the entries in `/oms`, and in the file handles for their sub-nodes, so that the OMS-NFS server can propagate the operations to the correct process.

In order to cross the boundary between NFS-OMS server and an OMS process, we need an object-identifier of an OMS object to which NFS operations (e.g. a lookup) can be sent. In principle, each OMS process could have one special object that performs this function. However, we have followed a different approach, leaving the choice for the object to the user. The object-identifier must be encoded in the file-name. So, the user process uses a file name like:

```
/oms/oms-id/object-id
```

to identify an OMS object (identified by `object-id`) of the chosen OMS (identified by `oms-id`). This two level scheme is also reflected in the file handles. For now, a file handle for an object in an OMS is represented by a tuple <*x*, *y*> where *x* represents the OMS process and *y* represents the object-identifier of an object within the OMS.

When the client system resolves such a file name, the following steps takes place. The OMS-NFS server is confronted with a "lookup" call whose arguments are the file handle for `/oms` and the name `oms-id`. It returns a file handle <*x*, *nil*> in which the oms-id and a special object-identifier (nil) are encoded. Then, the OMS-NFS server receives a lookup call on this handle together with name `object-id`. It return a new handle <*x*, *y*> where *y* is the encoded object-identifier. After that, any calls with a valid <*x*,*y*> pair are forwarded to OMS *x*, and sent to the object identified by *y*.

---

[2]A problem with this approach is that it makes it slightly more difficult to have multiple incarnations of the OMS-NFS server, since the OMS processes then have to be known by all these servers. A possible alternative is to use the OMS process's communication port numbers as their names.

6. File-system views in the OMS

Having decided on a general model for connecting the file system and the OMS, there is but one major issue left: how should we map the information in the OMS into a file tree?

The expected use of the system imposes some constraints on this issue. From the end-user perspective *all* the information in the OMS (objects, attributes, relations, etc.) should be accessible through file and directories. In particular, OMS concepts that have no equivalents in the UNIX file system (e.g. sending messages to objects) should somehow be accommodated. On the other hand, use of tools like compilers requires that we can generate the image of a regular UNIX tree (assuming that the relevant information is stored in the OMS).

Given this, which entities in the OMS should be represented as directories and files in the tree? One obvious choice would be to map OMS objects to directories and use the OMS directory-relation for generating a name space. The various attributes of objects could then be mapped to files. However, we could also use the sub-class/super-class and instance-of relationships to build a graph of classes mapped to directories and objects mapped to files.

Even from these simple examples it is clear that no single mapping of OMS to a file name space would match all our needs. So we need a mechanism to allow several of these mappings (or views) to co-exist within one OMS.

Managing multiple views

Distinguishing between different views is very simple and can be implemented within the OMS. It simply implies adding an extra level of indirection to the file handle. We assume that a view is represented by an object in the OMS which implements the various NFS-calls according to the semantics of the view type (defined in an OMS class). Reaching the view-object is simple, we can use a path-name like:

```
/oms/oms-id/view-object-id
```

as indicated earlier. We now however need to distinguish objects within views. Thus, we have to extend the file handle with an additional field, to encode the identifier of an object within the particular view. So, we now have a tree-tuple <*x*,*y*,*z*> as a file handle, where *x* identifies the OMS process, and *y* and *z* identify the view and an object within the view. The OMS-NFS server forwards the operations to OMS object *y*, passing *z* as an additional argument. As before, the handle of the "root directory" of view *y* is defined by a "nil" value for *z*.

Switching between views can be implemented easily also. The only thing that has to happen is that a file handle is returned (e.g. after a lookup) in which the view identifier is replaced by some other. By keeping the object identifier the same, we can even dynamically switch views on the same object. Presenting view-switching to the user must be done by the views. In our current model we use special names (see below).

7. Sample Views

This section some of the views that we have defined. The next section shows how the various views can be used in practice.

The Object View

This view works on a single, arbitrary object. Within this view, the object is a directory, while its attributes are presented as files. Reading and writing these files accesses or changes the value of the corresponding attribute of the OMS object. The directory is read-only, since the user cannot add or remove attributes to OMS objects. For write operations, the view must check for the correct syntax of values, i.e. they must represent syntactically correct Scheme values.

Methods that can be invoked on the object are also represented as files. These files are read-only, and reading them gives the result of invoking the method. Methods that require arguments are also accommodated. For each such method, a writable file with suffix "`-args`" is defined and maintained (by the view object). The file is used to store the arguments for a subsequent reference to the method.

To implement this view, we need an extra field in the file handle that allows us to distinguish between the various attributes and methods. This field is added and maintained by the NFS methods on the view.

The Class View

The Class View is comparable to the Object View in that it focuses on a single OMS object. However, as the name suggests, this object should be a class (an instance of class Class). The Class View provides access to the class definition.

The class is represented as a directory. Method definitions and attribute definitions that are associated with the class are presented as files which can be read and changed. The directory has read-write mode, which means that the user can create new files (but no sub-directories). Such new files are mapped to attribute definitions or method definitions using the OMS convention that method names start with colons. Again, this view uses an extra field in the file handle to distinguish between method and attribute definitions.

The Class Graph View

This view provides access to all objects and classes in the OMS. It maps classes to directories and instances to files (with exceptions made for class Class). The tree is created using two relations maintained in the OMS: the "super-class" relation among classes and the "instance-of" relation among classes and objects (see figure 1). Thus, the tree displays the inheritance hierarchy among classes while also providing access to the instances. The root of the graph is the class Object.

Each directory represents a particular class. Sub-directories are sub-classes of the class, while files are instances of that class. The parent directory refers to the super-class. Since a class can have multiple super-classes a directory can have multiple parents. To reach these parents we use a naming scheme similar to the Portable Common Tool Environment [Boud88], i.e. we use names like ..1 and ..2 to refer to additional parents.

Every directory in the tree is readable and writable. Creating a new sub-directory implies creating a new sub-class of the current class. Creating a new file in a directory means that a new instance of the class is created. This requires that we maintain a (perhaps separate) mapping of names to instances for each class.

8. Using the views

The following annotated script from the UNIX Shell shows the use of the views discussed above from an end-user perspective. It also illustrates some points not discussed earlier. We assume that the OMS process and the OMS-NFS server are activated, and that the OMS-NFS server is mounted on the directory `/oms`. We start by using the Class-Graph-View to navigate to class C-src which is a sub-class of Object[3]. Class C-src is used to describe objects that contain a piece of C source code.

```
# Navigate to class C-src
> cd /oms/ghf-album1/class-graph-view/Object/C-src
> ls
>
```

As follows from the empty directory contents, class C-src has no instances or sub-classes. In order to see the definition of class C-src, we switch to the Class-View on the same object. Note that switching views is implemented in the view handlers. A name beginning with '@' is inspected to see whether it represents a valid view name or identifier. If so, the view implementation returns a file handle with the view-part set to that view. Note also that the view implementation has decided to hide these names from the directory listing.

```
# Inspect the Class View on the current object
> ls @class-view
:compiled       :linked         :prettyprinted      contents
> cat @class-view/:compiled
....
>
```

We can see that class C-src defines one attribute (`contents`) and three methods (`:compiled`, `:linked` and `:prettyprinted`). Reading the contents of the `:compiled` file returns the definition (interface and implementation) of the corresponding method.

There are several ways to create a new instance of class C-src. One is to switch to the Object-View on the class, and invoke the method `:new`. Since we are still in the Class-Graph View, we use a different way; we simply create a file in this pseudo-directory.

```
# Create an instance of class C-src
> touch x
> ls
x
>
```

Note that in order to create named files (intead of generating names based on OMS object-identifiers) we need to maintain a mapping of names to OMS objects within the view.

Now we switch to the object-view on the new instance, and put some C-code into the `contents` attribute using any editor.

---

[3]We use symbolic names for object-identifiers and OMS-identifiers to reduce the visual clutter.

```
# Switch to object-view and do some work
> cd @object-view@x
> ls
:compiled       :linked         :prettyprinted
:compiled-args  :linked-args    contents
> ed contents
...
>
```

In order to switch to the object-view we use the @view-id clause mentioned before. In addition, we use the mechanism that if another name component beginning with @ follows the view-id, it is interpreted within the old view, but the file handle returned is within the new view. This is necessary here, since otherwise we would be in the object-view on class C-src.

We can see now the attributes of the object and the invokable methods. As we can see, two methods take arguments. Invoking methods is done by refering to the corresponding files. In this example, the contents of the file `:linked` is the linked executable version of the contents of C-src.

```
# Invoke some methods
> cat :prettyprinted
...
> ed :linked-args
...
> cat :linked
...
> :linked
:linked: Permission denied
> ls -l :linked
-r--r--r--  1 florijn  staff 12345 Feb 17 11:23 :linked
> chmod +x :linked
chmod: Permission denied
>
```

Note that through the use of built-in derivation management, the contents of `:linked` will always be up-to-date with anything it depends on; not only the attribute contents, but also indirect dependencies.

As can be seen from this example, methods are read-only. For the time being, modes of entries cannot be changed yet, since they are not stored but generated on the fly. So, the only way to execute the result of the link operation is by copying the value into a regular UNIX file[4]. Furthermore, the modes are the same for owner, group and world. This is caused by the fact that the OMS does not provide any permission model. The owner and group field of all OMS file and directories is generated from the user-id and group-id of the OMS process.

The method `:linked` invokes the regular UNIX C-compiler on the C code stored in the attribute `contents`. The outline of this method is as follows:

---

[4]Note that an alternative approach could be to give every object execute permission.

```
method :linked(args: string)
{
     x = send(Unix-View, new);
     ...add self to x's Unix tree under some name /tmp/y.c
     pid = fork();
     if (pid == 0) {
          chroot("/oms/omsid/x")
          cd("/tmp");
          execute("cc y.c");
     }
     else {
          waitfor(pid);
          ...retrieve results of compilation out of x's UNIX
              tree and return them
     }
}
```

The method starts by creating a new instance of view-class Unix-View. We have not discussed this view before, but basically, it mimics a standard UNIX File-Tree in the OMS. In fact, we reuse the underlying UNIX tree, and simply create a mirror image of it with references to the underlying files inside the OMS. We put the current object into that name-space, using a generated name. The Unix-View will retrieve the `contents` attribute whenever the file is referenced.

After creating the child process we do a `chroot` call to let the compiler see the UNIX tree in the OMS. The parent process waits for the compiler to terminate and then retrieves the results. These are in fact not stored inside the OMS, but are found in a TFS change layer [Hend88]. The OMS supports mullti-threading, so while the parent process is waiting the OMS process is ready to handle other calls (e.g. to retrieve include-files).

9. Conclusions

This report has explored some of the aspects of integrating the UNIX file system with an object-oriented system. We have considered two different angles: using the file system as an end-user interface and using the OMS as a substitute of the file-system for existing tools such as compilers.

The approach to integration described in this report meets the design goals discussed in section 3. Via the view mechanism described in section 6 the user can define arbitrary mappings of OMS concepts to UNIX trees. As a special case, existing tools like compilers can be run on top of the OMS. In this report we have only indicated some of the possible mappings from objects to files. Further study in this field could deliver more insights. A fundamental mechanism like view-switching is likely to be necessary so that different perspectives can be used in combination with each other.

From a slightly different angle, this project can be seen as an experiment in software architecture. By exploiting concepts like file system types and file system protocols, we have achieved our goals with minimum effort. Consequently, the implementation of the system is extremely simple. It is based on radically stripped down version of a user-level NFS daemon, and the addition of some classes to the OMS.

Although justifiable from a practical viewpoint, the choice for NFS has made life harder for us, and introduced additional complexity to the system. First of all, a better description of the NFS protocol, including the semantics expected by the client, is crucial to do the kinds of experiments documented here.

There are also some architectural complications. Current NFS implementations do not properly distinguish the concepts discussed in section 4. For example, they use NFS not only as a file-system protocol, but also tie it directly to the NFS file system type. As a consequence, our approach only works on systems on which no traditional NFS server is active. In principle one can add a new file system type that uses the NFS protocol; this is done by the loopback file system and the translucent file system. However this requires kernel modifications. Even a simple model allowing users to add file-system types (similar to adding device drivers) is not available.

An additional issue is that the existing NFS file-system type implementations we know of do not distinguish between server instances. All NFS operations for a particular site are forwarded to the same server address. The use of special servers for particular cases is not supported, which forced us to do de-multiplexing in the OMS-NFS server.

While removing these limitations appears relatively trivial it would make the use of file system technology as a means of integrating system a lot easier and give rise to many interesting new developments.

References

[Boud88] G. Boudier, F. Gallo, R. Minot, and I. Thomas, "An Overview of PCTE and PCTE+," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, P. Henderson (Ed.), November 1988.

[Feld79] S.I. Feldman, "Make – A Program for Maintaining Computer Programs," *Software – Practice & Experience*, no. 9, pp. 255–265, 1979.

[Flor92] G. Florijn, E. Lippe, A. Dijkstra, N. van Oosterom, and D. Swierstra, "Camera: Cooperation in Open Distributed Environments," in *Proceedings of the EurOpen and USENIX Spring 1992 Workshop/Conference*, EurOpen and USENIX, Jersey, Channel Islands, April 1992, pp. 123–136.

[Gold85] A. Goldberg and D. Robson, *SMALLTALK-80 The Language and its Implementation*. Addison Wesley, July 1985.

[Hend88] D. Hendricks, "The Translucent File Service," in *Proceedings of the 1988 EUUG Autumn conference*, Cascais, Portugal, October 1988, pp. 87–93.

[Hume89] A.G. Hume, "The Use of a Time Machine to Control Software," in *Proceedings of the USENIX Software Management Workshop*, Usenix, New Orleans, Louisiana, April 1989, pp. 119–124.

[Lipp91] E. Lippe and G. Florijn, "Implementation Techniques for Integral Version Management," in *Proceedings of the 1991 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, P. America (Ed.), Springer Verlag, Geneva, Switzerland, July 1991.

[Lipp92a] E. Lippe, "Camera: Support for Distributed Cooperative Work," Ph.D. thesis, Utrecht University, October 1992.

[Lipp92b] E. Lippe and G. Florijn, "Derivation Management in the CAMERA Object Management System," in *Proceedings of Computing Science in the Netherlands 1992*, J.L.G. Dietz (Ed.), Utrecht, the Netherlands, November 1992, pp. 203–214.

[Pike90] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," in *UNIX – The Legend Evolves. Proceedings of the Summer 1990 UKUUG Conference*, London, UK, July 1990, pp. 1–9.

[Pres91] D. Presotto, R. Pike, K. Thompson, and H. Trickey, "Plan 9, A Distributed System," in *Proceedings of the Spring 1991 EurOpen Conference*, Tromso, Norway, May 1991, pp. 43–50.

[Sand86] R. Sandberg, "The Sun Network Filesystem: Design, Implementation and Experience," in *Proceedings of the Spring 1986 EUUG Conference*, Florence, Italy, 1986.

[Scha92] C.G. van Schaik, "A Graphical User Interface for the CAMERA prototype," Master's thesis, Utrecht University, Department of Computer Science, March 1992.

[Soep93] Leo Soepenberg, "Using Unix-tools in CAMERA: Mapping OMS values to files", Master's thesis, Utrecht University, Department of Computer Science, February 1993.

[Sun88] Sun Microsystems Inc., "RPC: Remote Procedure Call Protocol Specification Version 2 – RFC 1057," Technical Report SRI - Network Information Center, June 1988.

[Sun89] Sun Microsystems Inc., "NFS: Network File System Protocol Specification – RFC 1094," Technical Report SRI - Network Information Center, March 1989.

[Thom78] K. Thompson, "UNIX Implementation," *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1931–1946, July 1978.

[Wein84] P. Weinberger, "The Version 8 Network File System (abstract)," in *Proceedings of the 1984 USENIX Summer Conference*, Salt Lake City, 1984, pp. 86.