An Object Layer for Conventional File-Systems

Thesis · October 1998

CITATIONS
0

READS
42

1 author:

Martin Wheatman

28 PUBLICATIONS **40** CITATIONS

SEE PROFILE

An Object Layer for Conventional File-Systems

A Thesis Submitted For The Degree of M.Phil.

By

Martin Wheatman

Acknowledgements

Firstly, I would like to thank my project supervisor, Dr. Pete Sawyer, for his guidance, advice and encouragement, throughout the production of this work. Credit is also due to Linus Torvalds, creator of Linux, without which this project could not have been implemented.

I would also like to thanks friends and family for putting up with my periods of incommunicado. I would especially like to thank Dr. R. E. Haworth for her love, support and inspiration, and for producing James David Haworth Wheatman, 28th November, 1996.

Contents

Introduction.

Object-orientation (OO) can be viewed as a list of properties, in which each implementation subscribes to different combinations. The property of persistence, whilst being an important property of an object [Booch 1991], is missing from many implementations, namely the OO programming languages (e.g. C++, Java), since these are deeply rooted in the conventional operating systems, upon which they are implemented.

Conventional operating systems support two memory models: one for volatile memory, the process address space; and one for persistent memory, the file system[1]. OO programming languages, like their conventional counterparts, provide the ability to describe the flow of control and data processing all within the process address space. This is deeply in-grained in that memory model, e.g. the calling of a function causes its actual parameters and local variables to be placed on a stack (a process address space data structure), to provide memory to be manipulated by the flow of control. Since on conventional operating systems there is no mapping between volatile and persistent memory, the provision of persistence gets pushed ‘upwards’ into the application code, usually by the flow of control performing explicit calls to manipulate a database. This betrays any attempt at transparency, i.e. the provision of persistent facilities with the least alteration for existing source code.

[1] This is a simplification since the persistence of the process address space and the file-system both overlap, in that a file-system will normally provide virtual memory, for example as swap space, and main memory will provide buffering to cache file input and output. However, the two memory models in conventional operating systems are (broadly) valid abstractions since virtual memory is not organised in files and directories (although it may be a file in itself), and is meaningless without the process address space it is supporting. However, the buffer cache, whilst being in volatile memory represents data which ought to be persistent. On a system crash, unwritten buffered file system is lost, which may lead to a corrupt file-system.

One established approach in providing persistence is to use a relational database to record the state of a running program. This allows the system designers to add data integrity constraints, such as transactions, in a consistent manner. It also requires a thorough investigation into the nature of the data before coding begins. However, this is far from ideal, since the relational database stores data in a completely different format to the programming language address space, which results in more code to convert the types, and a runtime overhead. This is known as 'impedance mismatch' [Zdonik 1990, Srinivasan 1997]. Because the relational database systems used to implement persistence are proprietary, there are a variety of ways to provide this conversion. There are also, within each proprietary 'solution', several methods of implementing persistence e.g. either embedding SQL and explicit database calls in a native language interface, or using proprietary programming languages supplied with the database system, to implement stored procedures. This all adds to the confusion over what is, and what isn't, implemented where. In addition, since all this extra code has to be written, despite the use of pre-processors for embedded SQL, there are extra opportunities for errors to be added to the system.

This is quite aptly described by [Booch 1991] as a 'clash of cultures' between the database and programming worlds. It can, however, be minimised by the use of OO databases which use a common object model to OO programming languages, to the point where the syntax of the OO programming language, in accessing persistent memory, need not be circumvented. [Sirinivasan 1997] highlights two broad approaches to providing access to persistent memory in programming languages by:

- providing a persistent class from which to inherit methods to access persistent memory; basically a read and write method. If this is to be transparent, the persistent class needs to be independently inherited from, so that persistence can be applied to any class.
- indicating the persistence of a class at run-time by specifying its implementation on allocation (e.g. by a cast in C++). Although the syntax is not broken, the code needs to be pre-processed to insert specific code to save/read the object into/from persistence memory. This allows the same type definition to be persistent or volatile as required.

Despite the second option being ‘less clean’, in requiring pre-processing, it may be preferred, especially if multiple inheritance is not an option. Also, persistence specification at allocation time allows a more arbitrary implementation of persistence between different implementations of the same record. This is effectively orthogonal persistence. This does however, lead to the restriction that the allocation of memory is dictated at allocation time and remains constant throughout the lifetime of that memory. The use of explicit read and write calls in the former method allows a more arbitrary use of persistent memory, within the lifetime of that memory.

This use of an object model, which is common to both the programming language and the database, is one example of using a ‘Unified Memory Model’ to circumvent the clash of cultures. One research goal, in making persistence transparent, is the provision of ‘orthogonal persistence’ [Dearle 1992], where the type of memory used to implement an object is orthogonal to its shape, i.e. its class. This more radical approach involves redesigning the operating system to combine the process address space and file-system. Memory is allocated as required by a running program (a thread), and is mapped onto persistent or volatile memory as required. Since persistent data is potentially ‘very large’, persistent operating systems only become practical with the introduction of 64 bit architectures. Examples of these operating systems include Grasshopper [Dearle 1994] and Opal [Chase 1992]. However, they represent a radical change to operating system design. Despite using conventional hardware, if not currently ‘commodity’ hardware, they require a full migration to a new system. To allow existing software to be supported, existing applications have to be ported to the new system. The notion of ‘file’ still needs to be implemented.

Conventional operating systems already have facilities for providing persistence, namely a file system. It would be advantageous if we could simply save persistent data to (ordinary) files:

- files provide efficient access to persistent memory: there should be little difference in performance between a file-system and a database. [Stonebraker 1991] shows that

there are similar features implemented in both file-systems and databases to support caching of data within fast/volatile memory to improve performance;

- being based upon a conventional operating system, this will allow existing software to be used alongside object space;
- rather than attempting to unify the memory model, the abstraction of a file is used as the bridge between the two models. Files are familiar structures for programmers, and with a suitable approach to pre-processing, arbitrary persistence should be achievable.

This does mean, however, that the quality of persistence would be dependent on the quality of persistence provided by the operating system: object space may only be 'stable' rather than 'resilient'[2].

However, using files and directories to provide an object space, as noted in [Nestor 1986] has five obstacles. In summary these are:

- Organisation; whereas directories use a tree structure, the structure in object based systems is naturally a network, and the conventional methods used to overcome these differences involve the repetition of directory structure;
- Abstraction; using the pathname to represent the logical structure of object space couples the logical structure of object space with that of physical location on a network of disks. Further there is no way of clustering objects, since the position of a file in a disk partition cannot be specified;
- History; whilst the history versions of a file is maintained by third party products, such as [Tichy 1982], this only works for files which are text based. Further, the ability to recreate particular versions of whole systems is not supported in conventional file-systems;
- Attributes; there is a fixed set of attributes for files and directories;

[2] In quantifying persistence [Dearle92] notes the qualities of 'stability' and 'resilience': 'Stability is the ability of a system to be consistently check-pointed on a secure medium, so that computation may resume from that point at some future time', and resilience as the ability to 'safely resume computation after an unexpected system crash, such as a power failure.'

- Synchronisation; whilst conventional operating systems support multiple users, the support in the file-system, for example the prevention of one user overwriting another's changes, is limited.

In addition, using files to provide persistence has the following problems:

- writing to files does work and is commonly used in simple examples, such as configuration files which have a simple format. This is error prone with large examples, since it is not trivial to write to a file, especially if there are arbitrary values to be stored. Therefore, saving persistent data in files requires a structured approach to achieve transparent file access (probably using source code pre-processing), and to the facilities that files provide;
- more importantly, both the schema (the 'shape' of the data) and the semantics are lost when a value is written into a file. The structured approach mentioned above needs to encompass the schema, as is achieved by Java serialisation. The loss of semantics may also be reduced when saving data to files, if the data is converted from the binary format of the process address space, e.g. into a printable character format. The overhead of this conversion may be outweighed by the increased independence and clarity of this data, especially where the data outlives the program which creates it, especially if the new version is implemented on a different architecture;
- we need access to stored procedures, including for example 'database' integrity methods, which are independent of the current application.

Objectives

The objective of this project is to provide a 'low-cost' and 'open' representation of an object model upon a conventional operating system. It is low-cost in that is uses a conventional file-system to provide persistence, and hence works alongside existing software. It is open in structure in that it allows direct access to values in files. It should also demonstrate OO facilities. The object model should provide entities (classes and objects) and members (attributes and methods) as classes, and also be extensible by

supporting class derivation and member inheritance from the resultant class inheritance net. In this project object space is the aggregation of fragments is explored, which allows the coexistence of conflicting states within objects space (so as to provide facilities for, for example, version control). A messaging system is prototyped as a basis for an OO shell. In addition, the project should identify features useful in modelling objects in this way, which will be encapsulated in the core model. A method of providing persistence to source code is proposed, allowing the arbitrary provision of persistence of source code defined data structures.

This thesis continues with a review of other ‘pragmatic’ approaches to the use of plain files in storing structured data in section 2. A high level overview of the features of this project in section 3, with a detailed description in section 4. Section 5 goes on to describe a worked example, and section 6 concludes with the achievements of this project and describes further work appropriate to this project.

Review.

Using an OO Database (OODb), either Object Relational Databases or true OO database management systems (OODBMSs), is one way to model persistent objects. However, there are many systems that model persistent objects without resorting to OODbs, either because of the lack of OODbs when designed, e.g. the Portable Common Tool Interface (PCTE) [Thomas 1988] and Continuus/CaseWare [Continuus], or through the need to present an open/non-proprietary system, such as the World Wide Web [Berners-Lee 1994b].

The goal of this project is to implement an OO object model upon a conventional file system. Since there is no one object model, it is important to look firstly at some of the object models already in use to identify features to be supported. A study of object models is non-trivial since there are many object models in existence, from 'non-executable' models, to OODbs and OO programming languages (OOPLs), each ascribing different importance to the properties of objects. This makes a complete review prohibitively long. Therefore, the two object models have been chosen, from the range of 'executable' object implementations.

This chapter starts by describing features common to OOPL and OODb object models, using Java [Flanagan 1997] and ORION [Kim 1990] as examples. For the purposes of illustrating these features, a simple graphical representation of the object model is built up alongside this description. It is intentional not to use an existing graphical object notation, as found, for example, in [Booch 1991], so that the reader need not be familiar with any one notation, and so that the notation is not encumbered with superfluous semantics. This chapter then briefly describes some of the approaches to providing facilities for modelling objects by modifying operating systems. It goes on to describe different approaches to modelling objects upon conventional operating systems, by limiting the representation of objects to the file-system. This review is rounded off with an overview of the World Wide Web (WWW) [Berners-Lee 1994b]. Although the WWW does not claim to be an OO system, it does support many features found in OO

systems and, moreover, it is implemented on conventional operating systems and file-systems.

Object Models.

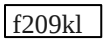
The basic concepts that are common to both OOPLs and OODbs are as follows.

Objects.

The fundamental building block or ‘entity’ in OO systems is the object. An object is a logically contiguous portion of memory, either in persistent or volatile memory. Objects are used to represent, or model, real world (or at least problem domain) objects.

Objects have two properties: identity and content. Identity is that which distinguishes an object. In the Relational Model, [Codd 1970], this is achieved by tuple key value. Typically, tuples from different relations may record information about the same real world entity, but need not use the same values or domains. In object models, identity is typically a singular- unique value. Different implementations of the object model implement identity in different ways. In a (compiled) program, such as Java, and C++ [Stroustrup 1987], the object identifier is the memory address of the object (known as an ‘object reference’ in Java, or an ‘lvalue’ in C++). In OODb systems, an identifier needs to be persistent. The translation between the persistent and volatile forms of identity leads to another example of type impedance mismatch, for which mechanisms such as address translation (or so called ‘Pointer Swizzling’) is required, as described in [Dearle 1992]. [Khoshafian 1990] describes identity further.

The graphical representation chosen for an object in this project is a square. Identity is represented as a unique identifier within the object.



g

Fig. 2-1

The content of an object, where data about that object is stored, is multi-valued: analogous to a programming record structure or relational database tuple. The individual

items are often called members, and typically can have many different types of values, including the usual atomic values of integer, boolean, date and string. Also, most OO implementations support operation members (known methods) and logical relationships. The latter includes aggregations (component objects) and associations (object references). Figure 2-2 shows a typical object with two members, one being an object reference.

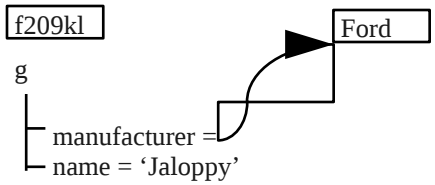


Fig. 2-2

The equals sign ‘=’ in the diagram simply separates the member name and its value. Since methods are shared between similar objects, they are stored elsewhere.

Components model objects within objects, such as an engine object being a component of car object. The car object is therefore a composite or ‘complex’ object. [Kim 1990] notes four types of component, quantified by whether the component is dependent or independent of the object of which it is a component, and whether or not the component may be shared between other composite objects. Figure 2-3 shows a typical object containing a component. Note that, in this example, the identity of the engine component will be dependent on the identity of the f209klg object: other car objects may also have their own ‘engine’ component which is different to this one. Independent and shared components, for the purposes of this graphical representation, are dealt with as an association, as in figure 2-2.

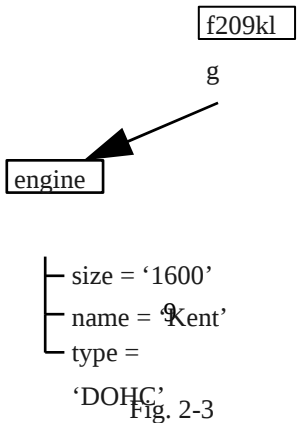


Fig. 2-3

So far described, apart from the multi-typing of members (including component/complex objects), the object model is little different from the Relational Model [Codd 1970], or the record structured model in traditional programming languages, such as C [Kernighan 1988]. The next step in the object model is that of classification.

Classification.

The classification of objects is another fundamental feature of all object models. In the real world, there exists many different (and similar) objects. Systems that model the world as objects need to reflect these differences. Classification is achieved through the introduction of the entity ‘class’.

A class is often regarded as an entity in a similar fashion to objects: having identity and content. Each object is an instance of (at least) one class, and hence has a type. Objects therefore have links to the classes which define them, in a similar fashion to reference links between objects. The content of a class is the definition of content of objects of this class. This definition, or ‘meta-data’, called class composition in [Kim 1990], includes: the methods applicable to objects of this class; attribute member type definitions; and, any default values of these members. A class is often called a ‘placeholder’ for meta-data.

A similar graphical notation is therefore chosen for classes, using circles instead of squares, as shown in figure 2-4. As for figure 2-2, the colon is merely a separator between the member names and their values, which in the case of attribute definition are links to the classes of those member definitions. The semi-colon denotes a method, the equals sign denotes a default value.

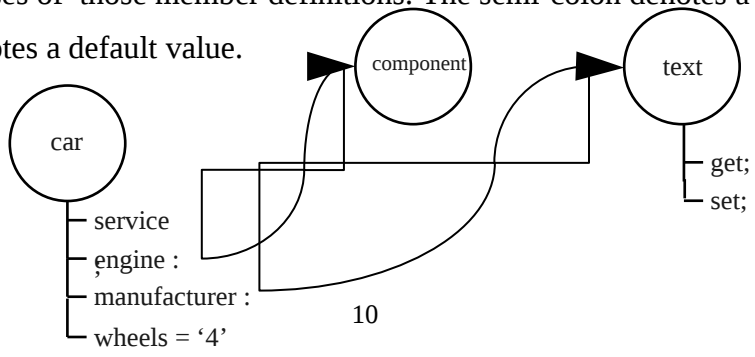


Fig. 2-4

A class will define the shape the objects of which they are an instance. For example, in figure 2-5, the object f209klg may have an engine component because one is defined for objects of the class car. Also, in the graphical representation, a solid arrowhead is used to distinguish between the object-object and class-instance links.

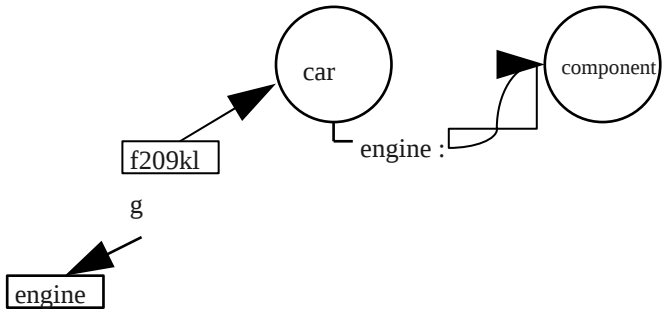


Fig. 2-5

Furthermore, this classification extends to classes themselves, whereupon, a class hierarchy is formed: each class is derived from a base class, and inherits the base class' definition. This is normally rooted in a class 'class'. Inheritance, another fundamental feature of object models, is the transitive classification of entities. If C' is a class derived from C, any class derived from (or object created from) C' will implicitly be derived from/an instance of C. Inheritance comes in two flavours, single inheritance, creating a class hierarchy, and multiple inheritance: extending this hierarchy to form a class lattice. This is illustrated in fig. 2-6. The class 'car' is derived from both motorised and vehicle, inheriting from both, and including its own member, 'configuration'. A query on the value of wheels for the object f209klg will return 4, even though f209klg does not hold the value 4: it is implied. Further, f209klg can have an engine because the class car has an engine attribute, inherited from motorised. A car is defined effectively, as a four wheeled motorised vehicle.

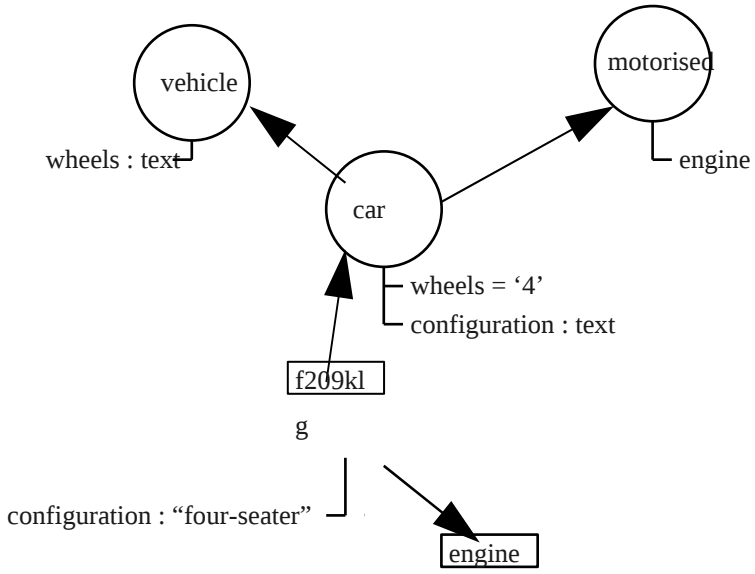


Fig. 2-6

In figure 2-6, car inherits from more than one class. This multiple inheritance raises problems, and is dealt with, where provided, in different ways in different object model implementations. With single inheritance, where the classification of classes forms a simple hierarchy, a member, bound to a derived class, will simply ‘override’ members of the same name in the base classes. With, multiple inheritance, where the classification of classes forms a directed acyclic graph, there is no simple override mechanism, because of the possibility of ambiguous members. Some forms of ambiguous members may be ignored or resolved. For example, where there are different paths through the inheritance net to the same member, the is no ambiguity. The case where an overriding member is bye-passed by another path through object space to the overridden member may be resolved by inferential distance [Touretzky 1986]. However, two truly conflicting members need to be identified by different means, and specifically resolved. The mechanism by which this is overcome in C++ is virtual functions (functions whose implementation is resolved at runtime by the :: operator). In Java, due to the performance overhead, on a system already relying on interpreted pseudo-machine code, multiple inheritance is not provided, suggesting that the overheads of multiple inheritance out-weigh its benefits. Multiple inheritance is further described in [Cardelli 1990].

In addition, object models often support polymorphism. This is the assigning of different types of values to a given member, where the assigned values are derived from the given member type. For example, in figure 2-6, another class may have a member of type ‘vehicle’, which will accept values of type ‘car’. This is also known as dynamic binding.

The combination of the class composition and the inheritance net forms a schema - a definition of the shape of object space.

Goals of Using an Object Model.

The main goals of OO, as defined in [Booch 1991], are; abstraction, encapsulation, modularity and hierarchy. These are attainable by the use of the object model. However, achieving these goals depends on the correct application of the above features. The class is the unit of data abstraction, within which information hiding, or encapsulation, occurs. Although this need not be successfully implemented - e.g. a Java class may be implemented with all members defined as public, and so not provide encapsulation. Indeed, there is a conflict, as identified in [Synder 1990], between the need to hide implementation details and the need for access, for example, to allow derived classes to modify that implementation.

Another important word used is ‘reuse’. This, again, is usually provided by the notion of class and inheritance: a derived class ‘reuses’ the methods and attribute definitions of the base class (those which are not overridden). Reuse is also provided by other specialised forms of ‘classification’, such as the grouping of ‘versionable’ objects in [Kim 1990], see below, and possibly during the process of schema versioning. Due to the lack of definition of object behaviour, despite attempts to formalise it [Duke 1991] and [Cusack 1992], reuse is difficult to define [Tracz 1988]. The word reuse does, however, encompass the ‘ethos’ of much of OO: the division of systems into components, and their subsequent reuse.

There are other features of OO. However, these are not common to all object models. For example, templates are provided by some object models but not others. Another important example is the provision of persistence. Whilst it is an important property of real-world objects, persistence is not universal in all object models.

Persistence.

Persistence is the ability of objects (or in the broader sense, data) to outlive the programs that create them: both the execution of the program and even the lifetime of the program, [Atkinson 1983] notes 6 levels of persistence, from the length of life-time of data in registers, to that of data on backing store. Orthogonal persistence [Dearle 1992] is where the quality of persistence of an object is independent of the type of that object: one instance of a type may be persistent, another instance may be implemented only in volatile memory.

There is no fundamental difference in the abstraction of persistent and volatile memory: objects in both are simply a logically contiguous block of bytes. However, on conventional operating systems the two memory models have developed different facilities for the structuring of memory, and the semantics of that structuring. Persistent memory is based on the notion of files and directories (or databases), and volatile memory is based around ‘raw’ blocks of bytes. The former is addressed by name, the latter by memory address.

An example of this difference in the semantics of structuring is that of object identity: in volatile memory this is simply an address in the process address space. This is, however, unsuitable for a persistent object, since it is dependent on the state of a program, and a more meaningful, machine independent identity is required. Conversion between the two forms can make up a significant proportion of a program’s processing requirements.

Another example of the difference between the two types of memory is that persistent memory often records data in a machine independent form, often in printable characters, volatile memory often works in raw binary data. A typical example is that of time: a system call to retrieve time is usually given as the number of seconds from a given point, e.g. midnight on 1st January, 1970, as an integer. The representation in persistent memory, lets say a database, may be a string ‘199611291709’, which allows the value to be recognised by differing system. Whilst there is a conversion between the two, there is an overhead in saving a given volatile time to persistent memory, and back. The greater

the proportion of a program is involved in this conversion, the greater the overhead. This is another example of type impedance mismatch[3].

However, since it is the structures that are different between memory models, and not the implementation of memory, to a larger extent impedance mismatch is minimised by the use of similar (object) models in both domains. Typically, persistence is provided by a binding between an OOPL and an OODb. [Srinivasan 1997] identifies two methods of providing persistence:

- by inheritance, where the object inherits persistent behaviour from a persistent class. This requires that the language supports multiple inheritance, and is often used in C++ implementations; and,
- by use of a modified 'new' operator an object may be created in persistent store, instead of the program address space. Persistence is achieved by reachability: if an object can be traversed to from a persistent object, then this object itself becomes persistent on updating the database. This is sometimes known as transitive persistence, and is used in systems such as Java and Eiffel [Meyer 1997], through their ability to serialise objects.

The mechanisms which provide persistence for a system also varies, greatly, between object models:

- the Java literature does not mention persistence, as such, however it provides mechanisms to achieve persistence, such as file I/O and serialisation. Indeed, the Java Applet class cannot interact with the file-system;
- ORION does not mention persistence at all, since all objects are naturally assumed to be persistent.

Some mechanisms provide persistence, upon conventional operating systems, other than explicit OODb manipulation, are described below.

[3] [Zdonik 1990] notes two main forms of impedance mismatch. Firstly, those due to the various differences in the type systems as described above. Secondly, the those due to the difference in styles of environment: the declarative data manipulation language, and the imperative programming language.

Serialisation.

Serialisation is often regarded as a ‘less-clean’ method, since it requires some pre-processing of code, but may be preferred in that persistence, to a larger extent, is transparent. However, it does not require multiple inheritance.

Serialisation does seem to imply, as noted by [Howard 1997], a great overhead is incurred if a single (persistent) atomic value needs to be updated in a highly connected object space, since all components of an object need to be serialised together, recursively. This may be overcome, as in [Howard 1997], by partitioning a system, however this encroaches on the transparency of persistence. Partitioning also implies separate databases or files, plus a mechanism to name objects other than as ‘object reference’ (which lead to recursive persistence), leading to the need for some schema in file space. Further, schema versioning (of serialised objects) is also a user space problem: a program needs to contain the classes of all serialised objects which may, in the future need to be de-serialised.

POMS: A Persistent Object Management System.

Alternatively, the language can be modified to include persistent facilities. For example PS-ALGOL [Atkinson 1983] provides some orthogonality. However, these facilities do not result in `transparent’ persistence. The language includes extra read and write functions to interact with persistent object space. This is known as ‘transactional persistence’.

Sun Microsystems mmap System Call.

The mmap() system call allows the user to map a region of memory over a file, by setting the MAP_FILE flag in the mmap() call, and providing a suitable file descriptor. This is supported in many UNIX-style operating systems.

Allocations in both volatile and persistent memory can be extended, (certainly in C) by the realloc() system call (for non-automatic memory allocation), and by the fact that files are of arbitrary length. The mmap() call, however, does not allow the mapped file to be extended. The mmap() system call is therefore more suitable for use along the lines of a

swapfile: a permanent, large file within which values, along with the structures to access these values, can be stored. All values are stored together in this file. In this project, the policy is to store each value separately as a file in its own right.

User Provided Memory Manipulation.

[Dearle 1992] also outlines another possible mechanism for providing persistence, which the operating systems Chorus [Rozier 1988] and Mach [Acceta 1986] support. These operating systems provide user programmable page fault functions which are called at page fault time. Further, they both provide the abstraction of ‘memory objects’ over real memory. On a swapping a memory object, this user function is called to provide any user defined actions, for example updating a persistent version. However, they are discounted in that neither do they provide the user with the ability to control when an object is made persistent, nor do programmers want to be involved in such low-level programming.

The following subsections describe ‘vehicles’ of the object model: systems which can be used to implement OO systems.

Object Model Example: Java.

Java is a programming environment which uses a syntax similar to that of C and C++. Its object model includes classes, class instances (objects), created with the ‘new’ operator; and members: variables and methods for both objects and classes. A so called ‘class’ member contains a value shared between all instances of this class. Objects are created within the current address space of the virtual machine (VM). This means that, as with conventional native binaries further work is required to support persistence.

Java only supports single inheritance. Overriding and overloading are therefore fairly straight-forward. Java also supports the notion of ‘interface’: an API specification which *must* be implemented in the classes which ‘implement’ it. Indeed, the lack of an interface method implementation is a compiler error. Many interfaces can be implemented by a class, but since only one method of a given name (or strictly of a given ‘signature’) can be implemented in this class, method ambiguity is avoided. One example is the run method in the runnable interface. It contains the functionality called when an object, which implements the runnable interface, is started as a thread.

Java also supports a variety of ‘visibility modifiers’, to give the programmer greater flexibility in the encapsulation that a class provides. This helps to overcome the encapsulation/inheritance conflicts, as described in [Snyder 1990]. The four modifiers, of public, protected, private and private protected, change the visibility of the default state of variables and methods in a class.

Granularity.

There is a relationship between the Java code and the file that contains it: only one public class can be provided by each source file, meaning that there is a 1:1 relationship between file and program (i.e. a C or C++ program can be split over many files). Each public class has a ‘main’ function declared as public static which acts as either a test program for that class, or a system main program.

In this review we look at examples of systems where files are used to model objects which contain links to other objects. In the case of the WWW this includes hyper-links to other HTML files. Java also models objects (logical units of code) as files, and contains links to other files, and file contents through the use of the ‘import’ statement. However, Java goes two steps further.

Firstly, and especially when modelling objects as files, there is (naturally) a relationship between a file’s name and its contents. This is the case with all files, not just languages, but with Java this is enforced by the compiler. For example, a C file called fred.c will typically be compiled:

```
cc -o fred fred.c
```

to force the output to be called the file name (without the file extension). A Java source file containing a public class ‘fred’ (and it can only contain one public class), has to be called fred.java or a compiler error is produced. Further, the compiled byte code file will be called fred.class.

Secondly, there is a relationship between an object and the objects around it: there is a natural structure to the file-system which contains objects as files. Java recognises this in the relationship between packages (the grouping of classes in the Java system) and their position within the file system. Java compiled .class files are placed in a directory path that has the same name as its package name, e.g. with a package name ‘my.package.fred’, the resultant .class file to be placed in the directory /my/package/fred .

Serialisation.

Java provides a simple form of persistence, from the 1.1 release, through the writeObject() and readObject() methods of the ObjectOutputStream and ObjectInputStream classes, which serialise and deserialise objects. Serialised objects can be sent through a stream, for example to another VM, or to a file, which constitutes a persistent object. Class version problems are prevented with a version identifier, which is a secure hash value of the class definition or signature.

Summary.

Since Java creates objects only in volatile process space, it does not, natively, support persistent objects. It provides mechanisms for writing to file-systems. However, these do not provide an object model on which to write an object. It provides mechanisms for creating serialised objects, for example maintaining identity across address spaces, however these are cumbersome mechanisms for implementing persistent systems, incurring potentially great overheads in translating from the persistent to the volatile. Further work on implementing persistence in Java is being developed by the ODMG, see below. Java, also, does not support multiple inheritance. However, it recognises that there are implicit links between the code and the files in which it sits.

Object Model Example: ORION An OO Database.

OODbs are primarily concerned with persistent objects in the context of a database system, but one which model data with an object model, rather than the conventional relational model. A relational database system uses tables, or relations, to model data. The rows, or tuples, represent entries in the table, the columns represent a value about that entry. Each table uses a combination of the columns to uniquely identify each entry.

OO Database Data Models.

OODbs perform the functionality of a database ([Date 1990] defines this as ‘essentially nothing more than a computerized record-keeping system’), using an object model to model the ‘record’. [Date 1990] describes OODbs as, specifically, supporting objects, classes, methods, messages and class hierarchy. Further, objects are composed of other objects or instance variables, and differentiates between primitive types and user-defined classes. Class hierarchy is subdivided into structural inheritance, and behavioural inheritance, and allows method overloading. Because of the success of the preceding generation of relational databases, OODbs are usually supported by the equivalent Data Description Languages (DDL), Data Manipulation Language (DML) and Structured Query Language (SQL). Above all, objects are persistent in an OODb.

There are many OODbs, such as POSTGRES [Stonebraker 1990, Stonebraker 1991], GemStone [Purdy 1987, Butterworth 1991], Jasmine [Ishikawa 1993] and O2 [Deux 1990, Deux 1991]. [Srinivasan 1997] classifies three forms of OODb:

- the gateway approach, which uses a client server model to access some legacy database, giving an OO client interface;
- the Object Relational, which includes extensions to [Codd 1970] for object concepts such as object identity; and,
- the pure OODBMS.

This section looks at the object model of the OODBMS, using: the ORION OO database system as described in [Kim 1990]. ORION has been chosen because of the extensive documentation available on its data model.

The ORION Object Model.

The core object model, described in [Kim 1990], is based around the following notions:

- Object and object id - the fundamental notion is around the ‘object’ which is represented by a unique identifier;
- Attributes and methods - the information about each object is stored in ‘attributes’ of an object. These are atomic values. The behaviour of each object is stored in

methods. Whilst these are used for setting and getting values and maintaining the integrity of each object, it is the authors opinion that they are not limited to these functions, and can be quite arbitrary in nature. This allows the implementation of active objects if this entity is required;

- Encapsulation and message passing - the internals of object are not visible to other objects. Objects interact by a well-defined interface a messaging system;
- Class - defining the 'shape' of an object of this class. This is known as a class-composition hierarchy (given the existence of composite objects, see below) representing the part-of relationships mentioned by this and other object models. This also allows the specification of shared attributes (visible to all objects of this class) and default attributes (giving an initial value for attributes of this class);
- Class hierarchy and inheritance - this is where the database system can truly be called OO, not just object based. [Kim 1990] specifies full or partial inheritance along with 'the usual' singular and multiple inheritance. Partial, which is unique to this model, allows certain attributes to not be inherited by particular classes. This appears to be similar to the use of visibility modifiers in Java: it is not only as useful in itself, but is also suggested as a solution to some of the problems of multiple inheritance. Method overloading is also defined in this model.

Semantic Extensions.

This is a fairly straight forward object model, covering most concepts in other object models. The core model alone is inadequate to model some application areas mentioned as users of OO database technology, such as Computer Aided Software Engineering (CASE) and Computer Aided Design (CAD). The model is 'extended' by the notions of Composite and Versionable objects. Composite objects are an implementation of entity valued attributes, which are supported as a standard feature of other object models. Versionable objects are a 'special type' of object, since a Versionable object requires overheads which it is advantageous not to have in all objects. Versioning is a special type of reuse.

The model is further extended with the notion of clustering, a constraint on the implementation of objects on disk to force improvements in performance. [Kim 1990] identifies two forms of clustering in relational databases:

- storing tuples of the same relation together,
- storing tuples of different relations together, to aid joins.

This can be extended in the OO model

- cluster all classes on a class-composition hierarchy,
- cluster all classes in a class hierarchy,
- cluster instances of a class or class hierarchy.

However, these extensions do not appear to be user extensible. A Version is a ‘special type’ of object: it is not a class which can be modified in any way, for example by overriding methods. [Kim 1990] does not describe any bindings to programming languages, all manipulation appears to be through the Data Manipulation Language.

Schema Modification / Object Properties.

Much work has been done in defining the changes that can be applied to schema over time, e.g. [Banerjee 1987, Skarra 1986, Zdonik 1986]. The integrity of object space is maintained, on modification of an ORION schema by invariants and rules, described in [Banerjee 1987, Kim 1990], which must be satisfied on modification. In summary, the taxonomy of schema change is divided into three types of changes:

- Changes to a class member;
- Changes to a class;
- Changes to the class hierarchy.

These changes must conform to the given invariants, such that:

- the class hierarchy forms a directed acyclic graph;
- class names are unique;
- all inherited members are unique, implying partial inheritance;
- full inheritance can be forced by inherited member renaming.

There are then ‘rules’ on how these invariants are maintained when conflicts arise during the schema changes, see chapter 5 of [Kim 1990] for further description.

Summary.

The core ORION object model, whilst supporting a similar object model, has some fundamental differences to that of many OOPLs:

- it is assumed that all objects are persistent;
- inheritance includes multiple inheritance and ‘selective’ inheritance, help overcome some of its associated problems;
- the core model is ‘extended’ by the notions of composite object and versions;
- object representation can be ‘tuned’ by clustering;
- no binding to an OOPL is described.

However, OODBMS, by their very nature are proprietary systems, and although they perform the same tasks, their implementations are very different. For example, the OODBMS O2 [Deux 1990, Deux 1991], takes a different, bi-modal approach to schema modification. The user can manipulate and test the schema in a development mode, and once ‘proven’, this schema can be compiled into a runtime version. Further, O2 has a Schema Manager which manages changes to the schema, such as only allowing deletes of leaf nodes, so reducing the problems in removing classes from a lattice. O2 also describes programming environments, as bindings to variants of the C and C++ languages. Persistence, in these OOPL environments, is achieved by reachability. Further, and more relevant to the core of this project, O2 is based on a third-party file-system: the Wisconcin Storage System (WiSS).

A Common Model.

It is clear that although each model supports similar notions, each implementation supports a different set of properties of objects, or meets these objectives in different ways. This does not mean that any one is better or worse than any other, since they provide the properties of interest to the domain to which they apply, and, definitively,

they support a core set of properties in order to be classified as OO, beyond being simply ‘modular’. For example, [Booch 1991] describes persistence as a property of objects, whereas in the Java model, an implementation based solely in the (volatile) process address space, the property of persistence is not mentioned. Further, persistence is not mentioned in [Kim 1990], since all objects are inherently persistent.

They each talk of the concepts of Data Abstraction: splitting the internal implementation from the external view of an object. Further, objects, containing identity and data, are instances of classes. The class contains the commonality between objects, including behaviour, and the composition of objects. Common mechanisms (object space behaviour) include: Inheritance, Polymorphism and Messaging Passing (or discrete communication).

This intersection of features is not a good starting point for an object model, since it is more notable for its exclusions than for its commonality. One common exclusion is the provision of persistence. OODbs are proprietary and each have their own bindings to OOPLs, if any, though [Srinivasan 1997] classifies two methods. The Object Database Management Group (ODMG), is attempting to define standards to which these proprietary mechanisms should adhere.

The Object Database Standard.

One attempt to pull together disparate persistence solutions is specified in the ODMG Standard, V2.0 [Cattell 1997]. The ODMG is a standards organisation involving database and language vendors, committed to evolving this ODMG standard, which includes a specification of language bindings for C++, Java and Smalltalk.

The standard talks about the binding of languages to OODBMS, specifying the properties of these bindings, such as the requirement that the binding should not break the syntax of the programming language. For example, the Java binding must contain only constructs already found in the language, so as to produce a ‘clean’ implementation. This reduces the cost of software production and maintenance: the programmer has only one programming model to work with.

Since Java and Smalltalk only support single inheritance, the persistence of their objects is by reachability. Because of its multiple inheritance capabilities, the C++ binding implements persistence by inheritance. It also allows object query language (OQL) and object manipulation language (OML) statements to be embedded within the code.

However, ODMG V2.0 is a standard not an implementation, and so it ‘describes’ but does not ‘provide’ implementations. For example [Cattell 1997] discusses that the implementation of persistent Java may be provided by:

- a bytecode post-processor;
- a source pre-processor, or;
- a modified Java Virtual Machine.

Summary.

There is a common object model, of objects, members and classes, and is expressed in graphical notation in figures 2-1 to 6. It is supported on conventional operating systems separately in the volatile and persistent memory models, by OOPLs and OODbs. These different models lead to complexities in programs, as there is a need to transfer objects between volatile and persistent memory. This is still non-trivial, despite using common object models. Since OODbs are dependent on persistent data, the issues of schema changes are more apparent. Schema changes in programming languages only become apparent when dealing with persistent data.

Unconventional Operating Systems.

Operating systems have been developed which provide facilities to support objects, especially their persistent nature. This section briefly outlines such operating systems and outlines their advantages and disadvantages.

Emeraude.

The first example is that of Emeraude, an implementation of the object based PCTE [Thomas 1988]. PCTE is a ‘public tools interface’ providing an Object Management System (OMS) and executive environments on Bull, and latterly Sun (and other) hardware. Tools requiring entity modelling on these platforms can be integrated through

the PCTE common facilities, and so allowing software to be ported easily between these platforms. It not only provided new facilities, such as entities in an OMS (replacing files and directories), but also existing operating system facilities, which were satisfactory, were re-written, such as the process management sub-system. The portability goals of PCTE have also been addressed.

Persistent Operating Systems.

[Tanenbaum 1987] lists four main components of an operating systems as: a file system, memory management, input-output and process management. Persistent operating systems combine the file-system and memory management functions to create a unified memory model, using the full (usually 64-bit) address space. This allows an executing program to allocate memory as required, and have it map onto persistent or volatile space as necessary. The quality of persistence is therefore orthogonal to the type of objects. Memory mapping mechanisms also allow the reuse of memory, for example ‘container mapping’ as described in [Dearle 1994].

There are several examples of persistent operating systems, such as Grasshopper [Dearle 1994], Opal [Chase 1992], Cedar [Swinehart 1986], Pilot [Redell 1980] and Psyche [Scott 1990].

Summary.

When working in an object environment, there is always the temptation to modify the objects that form the building blocks at a particular level either to introduce more functionality, or to reduce the complexity of this level. This may:

- unnecessarily complicate the building blocks, and;
- make it difficult to reconcile any changes with existing users of these new building blocks.

This has been the case where operating systems have been developed to support facilities for, or directly provide, objects. Whilst PCTE is an example of the former problem, persistent operating systems are examples of the latter: they present a radical change to operating system design. Despite using conventional hardware (although 64-bit machines are not yet commodity hardware), there are serious problems with moving these

unconventional operating systems into the wider world, even within the academic community. Unless these unconventional operating systems can co-exist and co-operate with conventional operating system, mass migration is required.

- This means existing APIs will have to be re-implemented for the new system to allow existing tools to be supported. This includes the notion of ‘file’ which needs to be re-implemented.
- An immense investment has already occurred in computing systems, and unless this migration can be (at least relatively) easy, there will be an extra burden on society, in terms of cost of upgrades, and costs during the transitional phase of possibly incompatible systems.

OO is arguably about the development of components and the reuse of these components. If OO broadly supports an “evolutionary” approach to software development, the migration to unconventional operating systems represents a “revolutionary” approach. Persistent operating systems are not widespread and will, in all practicality, remain ‘vehicles of research’, unless they produce radical benefits to users. The arguments against unconventional operating systems are not on their technical merits, indeed it is very much worthwhile that the details of memory implementation is removed from the application developer. However, the goal, and this solely a pragmatic argument, that the most effective way forward is to achieve this with the tools we already have. How many other software engineering problems result in a redesign of the operating system?

Conventional Operating Systems.

Other studies [Dearle 1992, Nestor 1986] have argued that conventional operating systems are not suitable to support an OO object model, in particular persistent objects. [Dearle 1992] states that conventional operating systems have separate memory models for persistent and volatile memory. [Nestor 1986] states that:

- the tree-structured organisation of file-systems is unsuitable (ambiguous) for object systems;
- there is no abstraction of logical objects and their physical location;
- source file version control, and binary file recreation is unsophisticated;
- the attributes of file-systems are not extensible;

- there are insufficient system provided facilities for sharing objects between multiple users.

The argument of this thesis is that conventional operating systems are a good starting point for implementing OO systems, because they do provide the necessary basic services for persistent objects, and because there has been a high amount of development already put into these systems. If the arguments for an unconventional operating system become compelling, it will be easier to migrate OO applications from conventional operating systems, than to migrate non-OO applications. Further, it is arguable conventional operating systems already support some OO facilities.

This section examines the facilities for supporting objects on conventional operating systems. It goes on to look at the mechanisms by which persistence can be supported, outlines other facilities missing in the support of objects, and the facilities provided by modified file-systems.

Object Facilities.

In the UNIX operating system [Bourne 1982], everything has a persistent representation in file space: device drivers, the file-system as a logical entity, and processes as instances of executable programs. Indeed, the commonly supported '/proc' file-system represents running process. Everything is a file, and there is a sub-typing of files: directories, for example, are a sub-type of file. Even plain files may be sub-typed by the program which created them. Commonly, file type is available to the users and the system for example by file name, system provided file type or by interpretation of the file contents. Although directories are not sub-typed.

Most, if not all, file systems support directories: entities which can contain files and other directories. This is a simple form of complex object: it can contain other objects (which are dependent and not shared components, as classified by [Kim 1990]). Most file-systems support symbolic links which can be used, as described by [Ponder 1997], to support other logical relationships (such as shared or independent components, as classified by [Kim, 1990]).

Further, the quality of persistence can be chosen by the use of different types of file-system, such as a synchronous or ‘write through’ file-system, where there is no file system cache, and on completion of a write, the file and its status is physically updated on the disk. On most UNIX file-systems, the O_SYNC flag can be used, when opening a file, to specify such cache-less action on that file. Many of the actions on the file-system are atomic, such as the creation of directories and file locking, allowing transitional functionality.

If hierarchical structures are used to model objects and, more importantly, class hierarchies, the user is limited to, amongst other things, single inheritance. This can be overcome, however, by using symbolic links to link one directory with many others, allowing the transformation of this hierarchic approach into a network. However useful, such use requires a rigorous approach. One example of such a rigorous approach is described in [Ponder 1997]. Although an OO approach is not specifically taken in [Ponder 1997], it gives a user-level approach to providing directories outside the normal hierarchical structure.

The shell, or command interpreter, whether used explicitly, or implicitly through a windowing system, simply takes the specification of a program and runs it with the given parameters. This constitutes a simple messaging system. Shellscript can therefore act as a data manipulation language. This is aided by the shell defining environment variables, such as the UNIX ‘PATH’ variable to determine the context with which to search for a command. By rearranging the ordering of the directory specifications in the PATH environment variable, a simple form of overriding can be achieved. Additionally, methods can be invoked which are dependent on the type of object but this is generally provided via a front-end, as in a graphical user interface.

Missing Facilities.

The object based approach, described above, falls short of useful functionality in terms of an OO environment. This section outlines some of the features, missing from conventional operating systems which prevent conventional operating systems from claiming to be OO.

Schema.

One problem with file-systems is that they are limited to a hierarchic structure: a directory (or folder) which contains files as well as other directories. In addition, there is no way of constraining this structure, by, for example, preventing the addition of (particular) files to directories. Further, complex objects are not typed like files.

The key to this project is that an abstract object model, a schema, should map onto files and directories. Such a schema must support:

- class hierarchy: an inheritance net, including non-hierarchical structures, if multiple inheritance is to be supported.;
- class composition: object typing including atomic, object valued and component members, along with members.

The file typing system, described in , covers plain files, and should be extended to cover composite objects. However, it need not define the structure within files (for example C source code), if this 'fine structure', (c.f. [Thomas 1988]), is well defined elsewhere. In conventional operating systems, fine structure is regarded as an application detail, and is generally defined externally as standards. In practice, fine structure is defined by the programs that generate and manipulate it. This fine structure, then, needs to be understood (separately) by each program, therefore object space on conventional operating systems will be 'less' integrated. Further, recent developments in conventional operating systems, specifically the Java model, reflect the relationship between the fine and coarse structure, in that, for example, a file may contain only one public class. However, tight integration is not a key goal of OO upon a conventional file-system, as long as the required operations on this fine structure are supported.

To demonstrate that it is OO, a schema should support, to a greater or lesser extent, OO features such as inheritance and polymorphism.

A Polymorphic Messaging System.

Operating systems need to provide a messaging system, to invoke methods in object space, and to hide the detail of object space from the user. The ability to invoke

programs, and therefore manipulate state, is provided in a simple manner, both by the shell, and as a programming interface (for example in the UNIX system calls). However, shell commands are operation based. For example the command:

edit my_file.txt

will invoke a text editor to edit the given file, but the command:

edit my_file.jpg

will probably produce either an error, or at least no ‘useful’ action. What is needed is for the edit command to work with the abstract object model: it should interrogate the type of file and call the appropriate program, either a text editor or image editor. This is an example of the shell being polymorphic.

It is arguable that this is circumvented by the use of graphical user interfaces (GUIs). For example, double clicking on an icon invokes a default action on the object that that icon represents; a menu of operations can be produced by operation of the mouse in some way; or a drag and drop notation may be implemented. However, GUIs are not extensible, they cannot be programmed by the end user (except, perhaps, by recording and replaying).

In addition overriding capabilities of the PATH environment variable are not, generally, exploited, since the path, being a list, would represent a singularly inherited hierarchy.

Approaches to Reuse.

Conventional operating systems do not directly support reuse. Reuse is a wide subject and covers such topics as inheritance, versionable objects, and schema versioning. Conventional operating systems rely on third-party file based configuration management/revision control systems. Although it is arguable that file version control systems are ‘system provided’ in UNIX since systems such as RCS [Tichy 1982] are bundled with the operating system, however these are limited to the management of individual text files: binary files cannot be stored, and configuration control, the recording of the state of many files, is not directly supported.

A general approach to object space reuse should be supported, encompassing the drawbacks identified by [Nestor 1986], namely history (source file version and binary file recreation) and multi-use synchronisation. Further, reuse should be used to cope with the problems of:

- multiple changes, where the same user is working on several ‘versions’ of the same object at once, and also;
- schema versioning, reuse of the inheritance net.

Modelling Objects Upon Existing Operating Systems: A Two-Tiered Approach.

To overcome the shortcomings of modelling objects upon a conventional operating system, a ‘two tiered’ approach, of ‘fine’ and ‘coarse’ grained data, as outlined in [Thomas 1988], is commonly taken. The distinction between fine and coarse grain data on a conventional file-system is at the file level: a file models an entity, presenting an atomic structure upon which operations can be performed, as a whole, and which contains (fine) structured data to store the state of the object. Since the file-system does not hold information on the shape of object space, either files must take on more content (no longer modelling objects, but object space), or tools operating on these objects must maintain the integrity of the coarse grain tier.

The object model described in [Flanagan 1997], as illustrated in section 2.1, not only model objects solely in volatile memory, but also only at the fine tier. This section looks at some other examples of the modelling of objects as files.

Ipsys Software plc., and Fine Structure Objects.

Ipsys products [Lincoln] use a proprietary entity based database, known as the Two Tier Database, because it conceptually has both a fine- and coarse-grained entity model. However, coarse grained objects are not fully utilised as there is no mapping from a file system to a non-hierarchical object space. The fine structure objects (FSOs) supported a network structure of classed entities, a model similar to that used by the Portable Common Tool Environment (PCTE). PCTE is particularly suited to ‘upper’ CASE toolsets (i.e. analysis and design techniques), as these require the representation of network (i.e. directed acyclic graph) structures. FSOs are not conducive to development environments (lower case toolsets), where object granularity is at the file and their

directory level (and sometimes the grand-parent directory), as there is difficulty in maintaining the integrity of links between fine/coarse and coarse/fine grained objects.

CaseWare/Continuus Mapping Database References To Real Files.

One example of a product that has taken a pragmatic approach to presenting users with a model of objects in the file system, is the configuration management tool CaseWare, and its successor Continuus/CM [Continuus]. These are based upon a relational database, which models files as objects and with attributes. The code, which may be anything from source code to test-scripts or HTML documentation, is stored in a 'source' attribute. On editing the source, the programmer's 'favourite' editor is invoked with the contents of this attribute. This source attribute is represented as a real file. This model is extensible: there is the ability to add new attributes to entities, for example, to add a 'release' attribute to allow a build operation to select the most suitable source file for a particular release. With both Continuus and CaseWare, there is the operation of 'reconciling' the database with changes by the developer, and 'synchronising' the developers' work area with the state of the database (i.e. to include newly promoted items from other developers), so as to allow development to proceed in the developer's file space.

One improvement of Continuus over CaseWare is that it generates 'standard' makefiles within file space: CaseWare generates its own makefiles, in a proprietary form. This is advantageous in that it allows developers to use their synchronised work areas fully. Continuus therefore becomes a configuration management tool, rather than a development environment. Third party tools may easily be used on personal work area, and hence all these tools become 'integrated' through the file-system: without a repository to define the interface between tools.

Modified File Systems.

This project assumes that a key feature of an object model is a persistent object. If a persistent object can be satisfactorily be implementation on the file-system, it would be sensible to limit the modification of operating systems, in providing an object model, to changes to the file-system. The viability of implementing different file-systems, such as [Rosenblum 1992] and [Hartman 1995], upon the same operating system, show that the

introduction of persistent objects need not necessarily lever the introduction of (fundamentally different) operating systems, such as Grasshopper [Dearle 1994].

The author does not know of a file-system that claims to be OO. This is possibly a problem of definition: a disk partition which supports objects is effectively an OODBMS, minus the query functionality. Indeed, [Stonebraker 1991b] notes that file systems provide many of the features of (relational) database systems, in providing access to persistent storage. [Stonebraker 1991b] is also relevant to OODbs.

However, if the modularity of the file-system can be extended as outlined in [Heidemann 1994], to construct file-systems of 'stackable layers', then an OO system could be provided with relative ease 'on top' of a conventional file-system. This should also be extensible in that other facilities could be provided, above this object layer, without affecting it. [Gelinas 1995] contains an example of providing one type of file-system upon another. [Deux 1990] provides an example of an OODb on top of a 'file-system', the Wisconsin Storage System (WiSS). Both are conceptually stacked layers.

One issue which is raised by using file-systems is what is stored in a single file, and what is stored in multiple files: what is the granularity of an object? This is often defined by the problem domain, for example, a file may contain the source code to a single program, but that source code is itself a structure composed of other objects. This issue is compounded on the introduction of clustering of objects.

Summary.

The provision of persistence in non-OO applications is an issue, although largely an invisible one. [Atkinson 1983] claims 30% of application code is dedicated to transferring data from a persistent to volatile state. Object based implementations upon conventional operating systems also have their drawbacks:

- file-systems (providing the persistent functionality of conventional operating systems) have not been developed to support an object model, and applications rely on third-party products such as databases and configuration management systems;

- the conventional approach to persistence is a (proprietary) database storing data, from the process address space. Issues, such as mismatch impedance, are minor problems, which can be ‘fixed’ by reprogramming or upgrading hardware;
- other *ad hoc* approaches to persistence exist, however, without standards, such as laid out in [Cattell 1997], leads to the fragmentation of resources, especially human.

Regardless of its importance to a particular system, in a multi-user environment persistence is a fundamental feature of the object model which will need providing.

World Wide Web (WWW): Real Files With Fine Structure.

The success of the WWW [Berners-Lee 1994b] is partly due to its straight forward approach to files and file-systems. The conventional view of the WWW is as a wide area (globally distributed) file-system. This section briefly describes differing views of the WWW, and outlines its short comings in being implemented on conventional operating systems.

WWW as an Object Model.

The WWW can be viewed as an object system: a network of passive (HTML [Berners-Lee 1995]) and active objects (Java/CGI), accessed by a simple messaging system (HTTP), its implementation is based upon real files and file systems and the Internet. This has been exploited by much research in attempts to combine the WWW with various Object technologies, such as CORBA through CorbaScript [Merle 1996], OO Databases [Yang 1996] and in an attempt to solve the problems of broken links [Ingham 1996] (an implementation of SSP Chains [Shapiro 1993]).

The case for the WWW being an object model is straightforward. Information is stored in plain files, usually containing Hypertext Mark-up Language (HTML), which is plain ASCII text with embedded tags (the fine-structure) to express interpretation of the text. The nodes of this network occur where the HTML include hypertext links (or ‘HREF’ tags) to other files or Uniform Resource Locators (URLs) [Berners-Lee 1994a] which are fine-to-coarse grained links (or fine-to-fine grained when using named tags within files).

Access to these files is via what are, effectively, simple messages passed to objects within the system. Clicking on highlighted text navigates the user to the given HTML page, however the mechanism behind this is the Hypertext Transfer Protocol (HTTP) [Fielding 1997], where the browser takes the machine id from the URL, connects to that machine and issues the GET command, which is part of HTTP. HTTP supports several commands, however the majority of commands issued (and indeed implemented in Web servers) is GET. This can be demonstrated by typing, for example:

```
telnet www.wheatman.demon.co.uk 80
GET /index.html
```

Stored procedures are provided by the WWW through server side procedures (CGI scripts) and client side procedures, also known as 'active content', through Java. This model is extensible in that application specific data types can be supported by a client side display function.

So the WWW is an object system, consisting of objects (of an extensible number of types), a messaging system, and active content.

WWW as an OO Operating System.

[Black 1995] suggests that the HTTP daemon is in fact an OO operating system, noting as the essential properties of an OO operating system:

- Uniform naming, provided by the URL;
- Persistent objects as files on a (distributed) file-system;
- Delivery of invocation messages, via the HTTP protocol;
- Implementations of some common and useful objects;
- Extensibility: the implementation of user definable operations 'on the fly';
- 'Critical Mass' meaning that the HTTP daemon is now a 'standard' part of most operating systems, and the HTTP service is globally available. The WWW certainly had critical mass when this piece was written, and this is more true today.

[Black 1995] also references earlier work [Bhasker 1983], in testing for OO in operating systems. However, neither stress class nor inheritance, and so perhaps these essential

properties outline modular or object based operating systems. [Black 1995] also mentions that operations can be introduced by the introduction of CGI scripts. However, these are more arbitrary than possibly desirable. It is noted that this is ‘essentially a hack’. After the introduction of Java as active content Tannenbaum’s model of an operating system is satisfied: input and output is provided by the client-side browser, the file-system is the provided by Web itself, and process and memory management are provided by the Java VM running within the browser.

Referential Integrity.

In the simplicity of the WWW implementation lies not only its strength, but also its weaknesses. The WWW lacks facilities for:

- directory facilities,
- referential integrity.

The former is, to a larger extent, solved by search engines, e.g. http://www.altavista.com/ which create virtual indexes, based on a search string. These are relatively quick and provide a large number of hypertext references. The power of search engines will need to keep up with the ever increasing use/size of the Web. Judicious use of search strings (along with other search specifications, such as the language to search) will become more important in providing useful search facilities.

Referential Integrity is still a problem, and [Ingham 1996] predicts that this will grow as the Web matures. Whilst HTML can be checked mechanically, by accessing the references on a given Web page, and by tools such as [Bowers 1996], ‘dangling’ references, ones which do not return a valid Web page, can be produced by either:

a) a client side error

- the user types in an invalid URL;

or, b) server-side operations, including pages being:

- removed, where the page becomes obsolete;
- renamed, where the page is superseded by one or more other pages;

- moved, where the filename stays the same, but the path portion of the URL changes.

Server-side operations are not ‘errors’ in that they may be created by the natural evolution of the WWW, and by the fact that links are one-way only (a server doesn’t know of the links to the soon-to-be deleted page). These can be avoided by a rigorous approach to server-side operations, such as those provided by a strongly typed object environment maintaining the integrity of object (Web) space, such as outlined in [Ingham 1996].

Conclusion.

In summary, the object model is based on the notion of:

- entities, which have identity and content (or members); and,
- links between entities.

There are two basic entities:

- Objects, whose members’ values are either atomic or links to other objects.
- Classes, whose members are the definition of the objects (or classes) for which the class is a template.

Both of these basic entities are reused in various ways, such as inheritance and versioning; Also, the object model contains other useful, but non-essential, facilities such as polymorphism.

The use of conventional file-systems is well established in the modelling of objects, however these approaches are not OO. Indeed, [Date 1990] suggests that a file in a file-system is equivalent to a table in a database, and it is a row in a table which is the manifestation of (or at least a view on) an entity. [Stonebraker 1991b] suggests that the mechanisms behind database and file systems are similar. However, database implementations may vary: some databases are more akin to swap space in conventional operating systems, i.e. a database is either a file, or a disk partition, or a combination of both, with its own internal structure. Using conventional file-systems to model objects directly should deliver the benefits of these (ready-developed) systems, such as

portability, performance, distribution, (global) scalabililty, and more generally, ‘openness’ as opposed to the proprietary approach of OODbs.

In reviewing several systems which model objects, two issues have emerged:

- persistence of objects being stored;
- granularity of objects being stored.

Persistence is generally ignored in OO programming languages. As noted in [Srinivasan 1997] there are generally two approaches to providing persistence, each with different merits. Indeed the ODMG V2.0 specification uses both persistence by reachability (for Java and Smalltalk) and persistence by inheritance, for C++. The lack of multiple inheritance is circumvented, by reachability, although implementation is not defined. Other mechanisms, such as serialisation, are generally seen as not appropriate for providing fine-grained persistence.

With the support of composite objects, there is the need to support many levels of granularity. In a conventional file-system this results in the introduction of a two-tiered approach to separate structures within files, and the structuring of files within directories. The latter tier is largely uncontrolled. Existing problem domains dictate the level of granularity, i.e. source code is stored within a file. Otherwise the level of split between coarse and fine grained objects is an arbitrary distinction: are components objects, i.e. files, in their own right? The answer will be problem domain dependent.

A side issue granularity is that of clustering attributes. This is also an issue in OODbs. [Kim 1990] specifies that attributes may be ‘clustered’, i.e. stored close by on disk to reduce the time taken to access values. This can be mirrored on a conventional file-system. For example, given that normally one text attribute is stored in one file. If it is useful for an attribute to be clustered, then one file could contain the values of this attribute for all objects, e.g. as in the comma separated notation of storing spreadsheets. Here the clustered values are stored together in a similar manner to a default values, i.e. a member which is shared between objects.

However, conventional file-systems do lack some major facilities of object models, as outlined: the lack of a schema (other than the hierarchical model), which results in problems such as maintaining referential integrity. Further, there is a lack of reuse of coarse grained objects.

The objective of this project is to model objects on a conventional operating system. The key to this is the methodical use of file space structures, and to move away from strictly hierarchic file-system structures. The provision of an object model upon a conventional file-system can be refined to:

- introduce a schema to define object space;
- employ systems, to overcome the hierarchical file-system, to support non-hierarchical structures such as multiple inheritance and shared components;
- provide a messaging system to support polymorphic messages, method overriding and reuse;
- provide a generic reuse mechanism to support schema versioning.

These facilities are provided, initially at least, in user space, as a stackable layer on top of a conventional file-system, as described in [Heidemann 1994] and implemented in [Deux 1990, Gelinas 1995]. Reusing the file-system as a component, in this way:

- allows the OO use of computers, in multi-user environment: especially whilst supporting existing users and applications;
- benefits from other properties of a component system: for example, the underlying file-system can be improved independently of the object layer;
- it should be portable across differing systems through a common set of features in the file-systems - such as that provided by Java File class.

Further, this layering should allow object space to be portable between operating systems, across a network, using suitable ‘middleware’, such as CORBA [OMG 1995], DCE [Shirley 1992] or general remote procedure call [Bloomer 1992].

High Level Description.

Overview.

Using conventional operating systems, there are two methods of implementing persistent objects: by using a (possibly OO) database, or by using a file system. The strengths and weaknesses of the use of file-systems and databases (and also persistent operating systems) are discussed in chapter 2. The work described in this thesis is based upon a conventional file-system.

When using a conventional file-system, a ‘two tier’ approach is often followed. In this way objects can be modelled either as files or, along with other objects, as part of a file’s content: as ‘fine structure’. This pragmatic approach allows the application to use the most appropriate representation in modelling existing data, such as files or records. This approach is outlined in [Thomas 1988], although the PCTE OMS is not a conventional file system. The coarse grain object is normally a file: it has content and identity. An example of fine structure on a conventional file system is a record structured file. It conforms to the structure laid down in the program that created it. Indeed, a database system, contained within a file, can be viewed as a single object containing fine structure. The structuring of the coarse grain is weak.

Ideally, both tiers should support the same object model, though most certainly by using different mechanisms, allowing the level of the boundary between the coarse and fine grain to be set at a level relevant to the application. Given that objects may be composite, and given that the boundary between the fine and coarse grains, at least conceptually, is invisible, coarse grained objects may contain fine grained objects, and fine structure should be able to contain (at least references to) coarse grained objects. Since there is no schema for the coarse grain, other than the hierarchical structure of the file-system, and its access privileges, the integrity of coarse grained data, beyond the basic file-system integrity, has to be maintained by the application.

[Atkinson 1983] reports that 30% of an application's code is dedicated to swapping objects between volatile and persistent states. This could be significantly reduced by using a uniform memory model.

Despite this lack of schema, reuse and transparency of persistence, file-systems do provide the essential features required in an object space, including: 'basic' persistence, multi-user facilities, effective distribution, and atomic behaviour. Further, the hierarchical structures commonly available in file-systems can be overcome by using symbolic links to model a non-hierarchical network (a directed acyclic graph) of links between entities. As found in, for example, PCTE OMS reference links.

The objective of this project is to implement an object space upon a conventional file system. It should provide OO facilities, such as inheritance and reuse, concentrating on implementing this object space in the coarse grain. This project also identifies features useful to the implementation of object spaces and reuse on conventional file-systems.

This chapter goes on to give some background motivation for this project, an overview of the object model supported, including its graphical representation, as outlined in chapter 2, a statement of the current state of the project, and a summary the novel features of this project.

Background.

This project was born from the work the author did with PCTE at Software Sciences Ltd. and latterly Ipsys Software plc. The PCTE OMS models non-OO objects with a schema, allowing two types of link between entities, reference links and composition links. An entity exists because there is a composition link to it. Composition links can only form a directed acyclic graph. Reference links are placed between entities by the applications in a structure defined by the schema.

Ipsys use the notion of a two tiered database, whereby at the first tier, objects are represented as a directory containing a .body file, containing the contents (fine structure) of the object, and a .index file allowing applications fast access to that content. Other than this structuring, the first tier is ignored because of the lack of schema. The second

tier, containing fine structure, contains a schema, which sets out the possible structure of object space.

This project goes at least some way to solving the shortcomings of the above mentioned approaches. A coarse grained schema is developed, but it exists upon a conventional operating system. This is an example of the reuse of a file-system/operating system, and fits in with the notion of stackable layers, outlined in [Heidemann 1994].

Object Model.

As identified in chapter 2, there is no one object model. However, they each, by and large, talk of essentially the same features. Indeed, the ODMG specification allows for many variants in the implementation of OODb bindings to OOPLs. Chapter 2 looked at several object model implementations. The object space provided by this project provides similar facilities to these other object models, and the following sub-sections outline the features of this object model.

Entities

Entities, both objects and classes, are represented as directories, since they, like files, have identity and content. This is unlike other coarse grain object models, which model entities as files. This means that the links between entities, normally represented as fine structure, may be moved into the coarse grain/file-system. This reduces the differentiation between coarse and fine grained data, allowing the boundary between the coarse and fine grain to be sensitive to the needs of the application area.

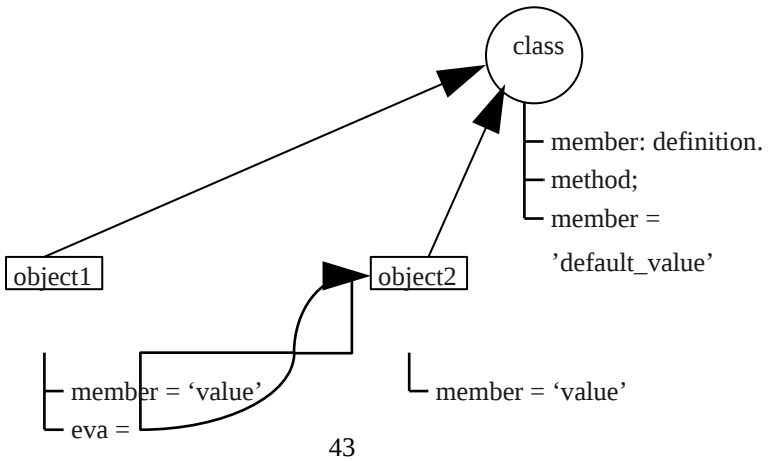


Fig. 3-1

Objects contain member values including attributes, ‘entity valued attributes’ (links to other objects) and (‘is a’) links to the classes to which they belong. Figure 3-1 shows two objects both containing a member of value ‘value’, with object1 containing an entity valued member of value object2. These are represented as two directories, both containing files ‘member’, containing the text value, and a symbolic link from the directory object1 to object2.

Classes are a placeholder for attribute type definitions (‘has a’ links) and default attribute values, both of which define the ‘shape’ of objects of this class. Classes also hold methods defining the behaviour of objects of this class. A class is outlined in figure 3-1, with three files: a member definition, (a symbolic link to the class of the member); a method or executable file; and, a file called member containing ‘default_value’.

Links Between Entities

Links between entities are important to the notion of entities, and are implemented by symbolic links. Links are implemented as part of the content of entities. Links include ‘isa’ and ‘has’ links, defining class hierarchy and class composition, along with attribute type definition links and entity valued attributes. This representation is extensible, allowing user defined links to be defined easily to capture application-specific information, such as ownership or container-ship. Figure 3-1 shows two types of link: two isa links and an entity valued member. Both are represented slightly differently (graphically), but both are implemented as symbolic links, and are named differently. The naming of members is detailed in chapter 4.

The notion of coarse grained links does not preclude fine grained links, such as HTML HREF tags, or C source #include directives, which will coexist. Further, it is possible to implement, for example, an HTML object in the coarse grain, as an ordered series of files containing plain text (including HTML tags) and coarse grained links to other fine and coarse grained objects. This is not be particularly useful, since it introduces impedance

mismatch, but illustrates that the same object may be implemented in both the coarse and fine grain.

Members.

Members are the contents of an object, defined by its class, and are represented as files and symbolic links within the object directory. A member may be whatever state is attributed to an entity by the application, including: executables, containing methods or the ‘behaviour’ of object space; normal files containing values, or state; or, links between entities, as outlined in 3.2.2. See figure 3-1 for member graphical representation.

Further, the content members may be defined (elsewhere), and if so, it can be described as ‘fine structure’. In the broadest sense, if the structure is defined as being an arbitrary collection of bytes, all files contain fine structure. More common examples of this sort of fine structure include:

- A C program source file is fine structure: it conforms to a pattern which is defined by the language definition. Notably it includes references to external entities through the #include compiler directive (fine to coarse grain links).
- A file containing Hypertext Mark-up Language (HTML) is fine structure: it conforms to a pattern laid down in the HTML standard. Notably, it contains references (by the very nature of hypertext) to other objects (fine to coarse grain links), and to structures within those objects (fine to fine links).

Reuse.

Object space supports reuse by:

- inheritance, including the overriding of methods in derived classes. On deriving a new class, the derived class is initially empty (save for the isa link to the base class). Methods in the base class maybe overridden by methods in the derived class. On overriding a method the functionality of the overridden method is available to the new method by a generated mechanism, and so methods may serve as wrappers to the base class method.

- polymorphism, by allowing the different methods, of the same name in different classes. This may also be used in conjunction with the inheritance mechanisms described above; and,
- schema versioning by the use of object space fragments (see 3.4.3).

Message Interpreter.

This project also implements a messaging system, allowing the dynamic binding of methods to classes, and hides object space implementation details from the user. In conjunction with several primitive operations, the message interpreter is the main mechanism by which object space is queried and manipulated.

Current State.

The current prototype is mainly written in shell script. This does not lead to an efficient system, since often there are several levels of call between different scripts. This also is not generally portable, despite one initial goal being the production of a portable project. However, it is convenient for the manipulation of the file-system: shellscript is the 'natural DML' of a file-system. The prototype provides the facilities outlined in chapter 3, and further described in chapter 4. Further work is described in chapter 6.

Novel Features.

This project, being developed from 'scratch', rather than from other research work, contains much novel work[4]. The following sub-sections outline some of the main features.

Open/Pragmatic Structure.

Entities are, unlike other implementations of objects upon conventional file-systems, represented by directories. Attributes are represented as files, and links between entities as symbolic links. This allows the structure of object space to move 'up' into the file-system. It also allows access to existing data files as members, as well as providing an

[4] [Bueche 1989] describes an OO shell, upon the UNIX operating system, however it builds its object model in the shell's process address space. It therefore requires a 'synchronise' operation to bring its saved state (stored in a file) up to date with the state of the file-system on start-up.

atomic structure for objects. Data structures can be modelled in both the coarse and fine grain.

Uniform Addressing of Components.

There is an historical distinction between class hierarchy and class composition. Typically, in programming languages, the syntax for addressing members of objects is different from that of the addressing (sub-)components of objects. For example:

Assuming a class hierarchy of ‘app’ being derived from ‘derived’, which is in turn derived from ‘base’, an object, ‘a’, of class app, will address method ‘b’ of class ‘base’, thus,

```
a.b();
```

However, assuming that ‘a’ is composed of component ‘c’, which has a method ‘d’, then ‘a’ addresses ‘d’, thus:

```
a.c.d();
```

This means that in the case of class composition, the source code contains explicit details of the composition of classes: information pertaining to the shape of object space. On the modification of object space, this information (not being recorded in ‘one place’) may need to be changed throughout the code.

However, in the case of class hierarchy, the path to the method is implied, and therefore may not need to be changed on schema update. This implied path to the inherited member allows only single inheritance.

This project provides a uniform messaging system, which treats components and inherited members in the same way, in a combination of the above mechanisms. This allows the reduction of information given in component specification, whilst allowing the specification of multiple components and multiple inheritance. This messaging system is described further in chapter 4.

Fragments.

On developing a core object model, and having approached reuse merely from the inheritance/method overriding approach, the author wanted to ‘ring-fence’ this work to proceed with further development, and a general approach to reuse was sought. Conventional operating systems do not provide native support for reuse, and so need to resort to text file difference control systems, and other third party configuration control systems. However, with the production of an object space, and the need to support versionable objects, the implementation of reusable objects should be native.

Object space in this project is viewed as being an aggregation of a number of object space fragments. This aggregation is viewed to work in the same way as the aggregation functionality of class inheritance. In the same way as the derived members can be accessed once the isa link is established between base and derived classes, so an empty fragment can reuse a reused fragment once the reuse link (another link between entities) is established. The concept of both inheritance and reuse involve reading from a [remote?] reused entity and writing to the [local?] reusing/derived entity.

Since an object space fragment contains the schema as coarse-grained data, a simple form of schema evolution, as outlined by [Kim 1990], is implemented. It is ‘simple’, in that it, currently, only supports fragment aggregation: the adding of members, methods, classes and classifications to object space, not a full taxonomy of schema changes [Banerjee 1987].

Transitive Links.

Given that inheritance is transitive: if a is derived from b, and b is derived from c, therefore a is derived from c. Since the inheritance net is represented by symbolic (user level) links, the notion of a ‘transitive link’ emerges, whereby, given that a symbolic link from directory b to directory c exists, and a link from directory a to directory b exists, implies that a link from a to c exists.

This transitive nature is conceptual: it is not available to the underlying file-system. For example, a ‘cd’ from a to b should (and does), result in the current directory being changed to b. However, using an object space equivalent of ‘ls’ on ‘a’ will list all the

files in (i.e. members of) b and c as well. The (simplistic) notion of the current directory as a ‘context’ is lost.

Transitive links are complemented by the notion of coincident directories.

Coincident Directories.

The common property of inheritance and reuse is the linking of directories, which conceptually have their contents superimposed. In the example given above, where a is derived from b and b is derived from c, a coincides with b and c, and b coincides with c. This is the same as the superimposition of reused object space fragments. The mechanisms by which these are achieved are provided by this project.

Summary.

This project provides an object space upon a conventional file-system. It demonstrates OO facilities, such as inheritance and polymorphism. The next chapter describes these facilities (structures and mechanisms) in more detail.

Detailed Description.

This chapter describes the system prototype in detail, which has been implemented using the Linux [Linux] operating system, a free UNIX variant. The facilities provided depend more on those available in the file-system, than in the operating system, and since modern file-systems provide similar facilities, there should be little difficulty in converting this project to run on other platforms. On file-systems that do not support symbolic links some work will be required. However, there should be no fundamental problems.

This chapter starts by looking at the representation of objects, followed by an outline of the environment used. Following this, operations that work on object space as a whole are described. Then a closer look at objects space is taken, starting with the core entity model, fragments, and then the (OO) object model. Finally, an illustration of the use of fragments in the user workspace is illustrated. Throughout this chapter, where examples look at the underlying representation of entities, they are given in terms of the listing produced by the ‘ls -F’ command. Specifically, the file name is followed by a forward slash, ‘/’, to signify the file is really a directory, a commercial at sign, ‘@’, to signify that the file is really a symbolic link, or an asterisk, ‘*’, to signify that the file is executable. The prototype source code for the utilities can be found in appendix A, the source for object space methods can be found in appendix B.

Object Representation.

As outlined in chapter 3, objects are represented in the file-system as directories, containing files which are members, and symbolic links which are links between entities. A directory, whilst being persistent has both content, consisting of files, and a unique identification, and can therefore undergo atomic operations. There are three tiers to the object space model: fragments, entities and members, represented as three levels in a directory structure. This may be increased to implement components (see chapter 6). Fragments, the top level, are orthogonal to the object model: an object may exist in many fragments. Fragments are linked by reuse links forming a “reuse net” as a directed acyclic graph. Objects are modelled in the latter two tiers. Looking at the file-system representation of objects, typically there are files and directories, thus:

```
frag_one/f209klg/
frag_one/f209klg/index
frag_one/ford/

frag_two/f209klg/
frag_two/f209klg/manufacturer -> ../ford[5]
frag_two/f209klg/index
```

Since file-systems have an inherent hierarchical structure, an abstract view of objects is used, to allow non-hierarchical structures in object space. However, an attempt has been made to map this abstract object model onto files and directories to allow native facilities to be used where possible, and to ease the migration of existing software structures. This is to integrate with the common software production practice of directories containing various source, object and executable files. For example, given the files in the following directories:

```
dev/bin/fred
dev/src/fred.c
dev/include/fred.h
```

which maps onto the object space in figure 4.1[6]:

[5] Note that the symbolic link ‘manufacturer’ links to a directory that does not exist. This is due to it an abstract entity, which exists elsewhere in object space.

[6] On close inspection figure 4.1 is indeed simplistic, however it should convey the idea of mapping conventional development environments onto object space.

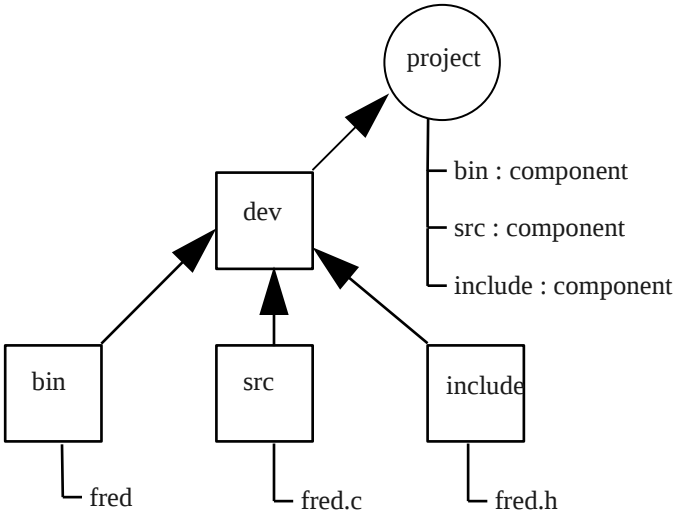


Fig. 4-1

Environment.

An environment variable, $PROJECT, is used to point to the project installation directory. This is assumed to be a ‘well-known’ value which is set for each user and (effectively) does not change. A user specific symbolic link ($HOME/.os) is a link to the user’s current local object space. The local object space (fragment) is where the user writes to object space. This allows each user to have their own object space. This link is not an environment variable since although it needs to change, those changes need to be persistent.

Layered Utilities.

Sections -3 describe three ‘layers’ of utilities: operations on object space that do not pertain solely to specific entities, i.e. object space operations which are not methods.

Five Primitives.

Primitives hide the complexities of the representation of objects within fragments within file-space, and should only change when the underlying representation of entities changes. This subsection describes the primitive operations on object space. There are currently five primitives: dir_name, filename, invoke, isa and has. Mainly prototyped in

shellscript, primitives are operations on object space, which are not methods. They do not pertain to a particular entity, but view objects as their native directory/file implementations. Reducing the numbers of primitives also reduces the amount of work to be performed on porting the project to other platforms or modifying the underlying object representation.

Primitives are fragment-aware in that they are designed to work with the reuse net. Due to the use of fragments in the current implementation of object space, there is a ‘core’ function within each primitive to perform the function of the primitive, which is then called for each reused fragment. It is assumed that most fragments will be stable whilst being reused, and therefore reconciliation (i.e. the consolidation of overridden members - especially the consolidation of changes between users) may not be trivial.

The first two primitives are filename and dir_name, which provide a mapping between the entity and member names and their filename and directory name representations. The third is the ‘invoke’ primitive to call methods of entities. The final two, isa and has, return information about the OO object model. These primitives are outlined in greater detail below.

Filename.

The filename primitive provides a mapping between entity and member names onto real files. An example call:

```
filename f209klg index
```

returns:

```
/home/martin/.os/f209klg/name
```

Since all writes are performed in the local object space fragment, there is the need to differentiate between member read and write access, so a -w option is provided to specify a write. If the -w switch is specified, then the user’s current fragment is echoed, whether or not the file exists. There may also a need to find the ‘next’ member in the object space reuse net ‘behind’ this member, for example, if filename is being used to ‘reconcile’ the hidden value, i.e. combine different versions of values. This is similar to passing method calls up the inheritance hierarchy, but pertains to the reuse hierarchy.

The core function of this primitive checks the existence of the required entity/member combinations in the given fragment. Unfortunately, the implementation in appendix A shows its prototype nature in that it will return as soon as one member has been found: it cannot report on, for example, multiply inherited members.

Dir_name.

The dir_name primitive is similar to the filename primitive. The core function of this primitive checks the existence of the required entity in the given fragment. Again, this implementation will return the first directory name found in a breadth-first search, which is not ideal if multiple inheritance is to be supported. An example call may be:

```
dir_name f209klg
```

returns:

```
/home/martin/.os/f209klg
```

There are three modes of operation: with no command line switch it returns the first found in the inheritance net; with -w specified returns the directory name of the entity to create (which is created); and finally -r finds the entity ‘behind’ the current entity.

Invoke.

Invoke hides the implementation details of object space when calling methods. It allows the user, and other scripts to invoke the methods of entities. For examples of the invocation of invoke, see the object space descriptions in and .

Invoke has two mandatory parameters: the entity name and the method name. The appropriate method is found and then invoked with the remaining parameters, as arguments to this method. If a -f switch is used, the location of the method to be invoked is merely echoed.

Invoke performs a breadth first search of object space for the method in the reuse hierarchy, in a similar fashion to the filename and dir_name primitives. Again, this function returns on finding the first value, and so can only return the first method it comes across in object space.

Invoke was initially written to work with the notion of roles. Roles are not implemented in the current prototype, the introduction of which is discussed in .

Isa.

The isa primitive outputs the class hierarchy. It relates solely to the object_model fragment, and its reusing fragments. Typical output from the isa primitive looks like:

```
f209klg is a car
car is a vehicle
```

The isa primitive calls a utility, called all, especially written for this project, which expands the output transitively. For example the output above is expanded into:

```
f209klg is a car
f209klg is a vehicle
car is a vehicle
```

‘all’ is seen as a text manipulation utility, rather than a primitive in itself, since it need not know the representation of entities upon the file-system. ‘all’ is written in C, in the file all.c, and is included in appendix A.

The isa primitive allows arguments to query object space. Firstly, the query:

```
isa vehicle
```

returns the classes which vehicle is derived from. Secondly, the query:

```
isa f209klg vehicle
```

returns ‘yes’ if the entity f209klg is derived from/ and instance of the vehicle class. The isa primitive allows the use of the percent (‘%’) character as a wildcard:

```
isa % vehicle
```

returns all the entities that are derived from/an instance of vehicle.

Has.

The ‘has’ primitive outputs the class composition. It pertains solely to the object model fragment, and derived classes. Typical ‘has’ output from the core model as seen is:

```
entity has a context method
entity has a create method
entity has a link method
entity has a remove method
member has a name method
```

The ‘has’ primitive has the ability to accept the character ‘%’ as a wildcard in the same way as the isa primitive, such that:

```
has % set
```

returns:

```
fragment has a set method
text has a set method
```

Inherit.

The second layer primitive ‘inherit’ echoes the originator class of an inherited member, deduced from the output from the isa and has primitives. It therefore also works on the object model fragment and fragments that reuse it.

The inherit function first creates the inheritance net represented by the file-system. The inherit prototype then calls a C program to perform text processing on the output from the isa and has primitives, to calculate what has what, given the isa inheritance net. This text processing is as generic as possible for it to handle other transitive relationships.

Given that, as in figure 2-6, the class vehicle has a wheels attribute, and car is derived from wheels, the command:

```
inherit -one[7] car wheels
```

returns:

```
vehicle
```

Whilst the file-system provides the ability to support multiple inheritance, the utilities here are not written to exploit the situations that may arise from having a member inherited from more than one class. For example, the inherit utility may output a list of

[7] Inherit requires a mandatory flag indicating how to deal with multiple inheritance. Only the ‘-one’ flag is currently supported, which signifies that the first relevant match is returned.

classes from which an entity inherits a member, but the utility that uses this, namely the ‘send’ utility (see below), does not manage this list (it expects a single item). Further, the representation of a member is a file. This does not explicitly state the class. For example, an instance *i* may inherit a member *m* from base class *a*. Member *m* is created and managed using the definition in class *a*. If multiple inheritance is allowed (or indeed the inheritance net changes), it may result in instance *i* also including a definition of another member *m* in a different base class *b*. The inherit utility, as with all utilities in this project, takes a simplistic approach, returning the ‘closest’ member definition in the inheritance net. The resolution of such problems with multiple inheritance are beyond the scope of this project.

The source for inherit.c, and the inherit script, can be found in appendix A.

Send.

The inherit utility the provides the basic functionality of an OO environment. A third level primitive called ‘send’ is prototyped. This polymorphic message interpreter supports simple messages, such as the following:

```
class derive person
person bind text name
person instance martin
martin name set ‘Martin John Wheatman’
martin name get
```

In this example the last message returns:

```
Martin John Wheatman
```

Send also calls all the other primitives so this can be used as the message interpreter of an OO shell.

An experimental message interpreter has been prototyped which traverses object space, interpreting messages such as:

```
f209klg manufacturer name get
```

This applies the get method on the name member of the entity which has the value of the member manufacturer of the entity f209klg. The source code for this is not included in the appendix.

Overriding Methods.

Since method overriding is supported by the send utility, it is necessary to allow the invocation of the overridden method by a simple mechanism. This is provided, for example, in Java, by the 'super' mechanism. In this project, the mechanism is provided as a protocol in calling the message interpreted, by using a -generated switch. On receiving a -generated switch in the message, 'send' makes calls to the 'inherit' utility forcing it to search the class's inheritance net and not the class itself. This allows the method to contain the line (exactly as is printed below) in any method:

```
send $0 -generated $*
```

to '*re-send*' the given message into this method's class's inheritance net, for an overridden method. This overriding method is effectively a '*wrapper*' to the overridden method.

Inefficiencies of Send.

As mentioned, this message interpreter is a prototype, and as such is not as efficient as might be expected. Firstly, it is written in shell-script, simply to fit into the rest of the project. This allows relatively easy prototyping and inspection of the implementation of the OO environment.

Further, it not only calls other scripts, such as the primitives, but also may call these primitives several times during the invocation of a message. No attempt is made at caching the output from these primitives, resulting in repeated work. Caching may also be possible between different messages, depending on the integration of the message interpreter with object space. It is envisaged that a message interpreter would also include an object buffer, however, this is outside the scope of this project.

Core Model.

The core model contains two entities, 'entity' and 'member'. Entities are place-holders for behaviour and state, such as methods and member values, and explained in more detail below. If we examine object space we will see two directories:

```
entity/    member/
```

Entity.

Entity contains four methods: create, context, link and remove which are listed in appendix B.

The method ‘create’ simple creates an entity. An example invocation may be:

```
invoke entity create f209klg
```

Since entities are represented as directories, the main function of this method can be performed by the UNIX mkdir command. This requires the name of the directory (in this case the entity name) to be supplied as a parameter. For the moment, object space is, effectively the directory pointed to by the symbolic link in the user’ home directory, $HOME/.os . No attempt is made to connect the newly created entity to any other.

The ‘link’ method, which is used to create a link between two entities, is called with three parameters: the entity the link comes from, the link type (name), and the entity to which it is linked. The main part of this method is performed by creating a symbolic link. For example:

```
invoke entity link f209klg manufacturer ford
```

The ‘context’ method echoes the full path name to the current directory.

The ‘remove’ method checks to see if the specified entity exists, and if so, deletes it.

Member.

The entity ‘member’ contains one method, name, to echo the name of a member, giving a consistent approach to naming members.

There is a -n option to specify that the member is one of a group, i.e. it may hold several values, identified by a number. If the -n option is specified, the output will be the next one in the group, e.g. if the member name is to be ‘name’ and the -n option is specified, and no members of this type currently exist, the output will be ‘0.name’. If this output is used to create a member, the subsequent output will be ‘1.name’ in successive calls, and so on.

There is the option to specify a name as well as type, to give, for example, ‘left.arm’, ‘right.arm’. This can be used in conjunction with the -n option. The author wanted to be able to model, for example, a book object containing chapter attributes thus: ‘chapter.0.txt’, ‘chapter.1.txt’, etc. Further combinations have been avoided, it is assumed that another schema may be used to overcome these cases.

This is the only method for the ‘member’ entity. There will be different types of members, including entity/link, attribute/value and class/methods (and composite/sub-components), and member names are seen as their only common feature.

Summary.

This core model supports the basic entity model: how to create, link together and remove entities. For the purposes of this project, referential integrity is checked at a much higher level, so links may exist to entities that have been removed. This ‘lazy’ removal policy, for the moment, is deemed acceptable, however some mechanism will need to be introduced to deal with the case on the re-introduction of entities.

During the development of the prototype scripts, a mechanism was sought to modify the model without changing this core model - to replace where necessary but not to overwrite, and is used a similar mechanism to the object model’s inheritance mechanism. Since meta-data is stored within object space, this mechanism can be useful for, amongst other things, schema versioning. Thus, the concept of fragments is introduced, within its own fragment.

Fragments.

Reuse is a goal common to all OO implementations. It is conventionally achieved by several mechanisms, including inheritance and versionable objects. The introduction of object space ‘fragments’, as discussed, shifts the emphasis from reusing individual objects and members, to reusing regions of object space. Since this project stores meta-data as part of object space, this project supports (at least a simple form of) schema versioning, as outlined in [Kim 1990]. The analogy of a fragment is the overhead projector slide (OHP), which can be covered by another to ‘reuse’ the one beneath, leaving it unchanged, as outlined in figure 4-2.

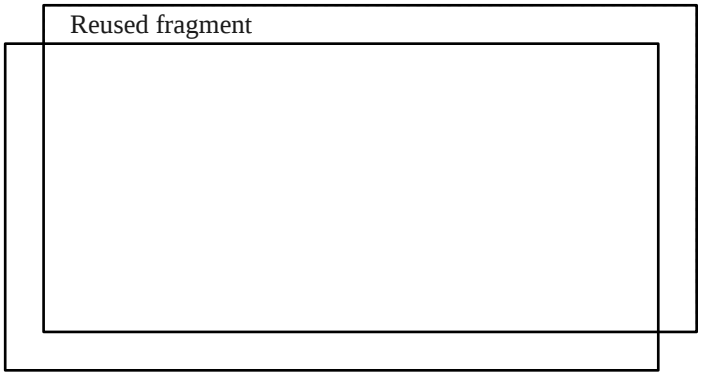


Fig. 4-2

Fragments increases the ease by which object space resides on many partitions. Object space is an overlapping collection of fragments, linked by reuse links. This uses a similar mechanism to isa links, in which derived classes are linked to their base classes, and is analogous to the reuse between classes, and hence the phrase *reuse net*. Object space has three tiers: of members, classes and fragments. A fragment is similar to the notion of composite objects, as proposed in , composed of non-shared/dependent sub-components. As mentioned in 4.1, $HOME/.os is a symbolic link into the users (current) workspace fragment, from which other fragments are reused. The notion of objects are orthogonal to the notion of fragments: each fragment contains a ‘slice’ of object space, and therefore, probably, only a ‘slice’ of an object.

Since object space spans many directories, the basic implementation of the core model is re-implemented to span reuse links. Object space now contains three entities: entity, member, and fragment. On examining object space we see another addition to object space: a reuse link:

```
0.reuse@  entity/   fragment/  member/
```

This reuse link points to the core model fragment, allowing it to be reused. Most methods, mentioned so far, need not be re-implemented as they work in the $HOME/.os, and are protected by fragment-aware primitives, described in . However, entity now contains a ‘destroy’ method, which removes an entity from all fragments in a ‘reuse net’.

Fragment.

The ‘fragment’ entity contains several experiments in what can be achieved using fragments.

The main method within the fragment entity is the ‘reuse’ method, which is used to create reuse links between fragments, in a similar fashion to the entity/link method. It is called with one or two parameters: the first (optional) is the fragment which will reuse, the second is the fragment to be reused. If only one parameter is supplied it is assumed to be the reused fragment and the reusing fragment is the user’s $HOME/.os fragment. For example:

```
invoke fragment reuse core_model
```

Another useful method is the ‘set’ method, used to set/echo the user’s current ($HOME/.os) fragment.

```
invoke fragment set workspace2
```

The fragment entity also contains some experiments in fragments including transactions, using the begin and quit methods for creating a fragment on top of object space, which can be rolled back with the quit method. These are experimental and are not described further.

As an aside, the introduction of fragments, being orthogonal to objects, means that the implementation has the overheads of the replication of information within object space. This is chiefly the replication of object identity, but also in the content of objects, between fragments. This is assumed not to be an issue, in that:

- object identity does not, generally, change;
- older fragments can be either ignored where overridden, or cached, in some way, if referenced often. Reused fragments can be regarded as ‘stable’, all writes are performed in the user’s object space ($HOME/.os).
- other methods of caching objects within the process address space can be employed to reduce overheads.

The entity model is now a functionally-rich environment upon which to implement the object model, which is described in the next sub-section.

The Object Model.

The object model fragment reuses the fragments fragment, and so reuses the whole entity model. It contains six entities: attribute, class, member, method, object and text, and represents a ‘typical’ object model as outlined in chapter 2. The entities contained within the object model are all classes linked together by isa links (which are managed by the class ‘class’). The class hierarchy is arranged as shown in figure 4-3.

The ‘class’ and ‘object’ classes represent the class and object entities. The ‘attribute’ class is the member class, and is a base class to the text class. A member value of text is the only atomic value represented as yet. Partly this because this is only a prototype, and also because most types can be stored in a textual form, e.g. in Java all classes override the toString method.

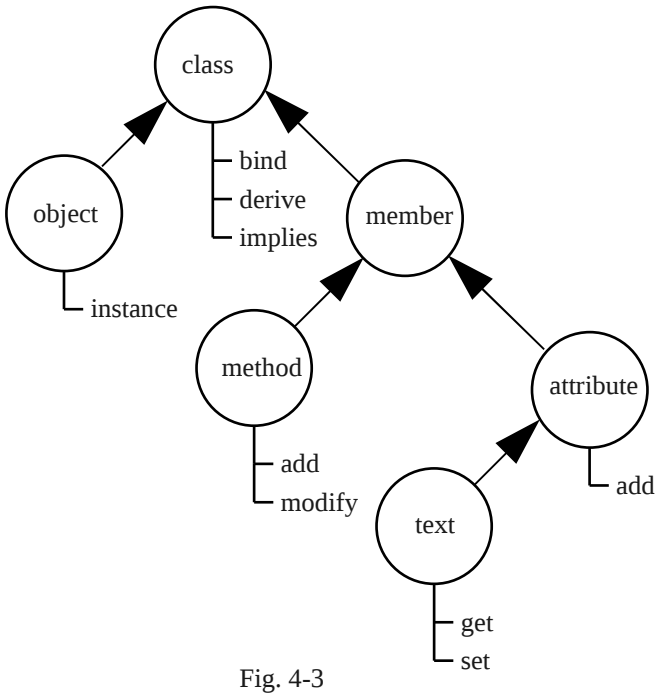


Fig. 4-3

Class.

The entity ‘class’ contains the methods: bind, derive and implies. The entity class manages the creation of the inheritance net, rooted in the entity ‘class’, and the creation of class composition.

The derive method creates a new class, derived from a given class. All classes are rooted in the entity ‘class’. The method ‘derive’ uses the ‘invoke’ primitive (see) to perform the majority of the functionality of this method, since it calls functionality from object space. Firstly, it attempts to create the entity given as the second parameter. The second invocation of the invoke primitive links the two classes with the implies method, for example:

```
send vehicle derive car
```

The ‘implies’ method invokes the entity link method between the first and second parameters with the link (member) name given by the appropriate invocation of the member name method, for example:

```
send car implies motorised
```

The bind method binds an attribute definition to a class entity. The bind method simply tells the given attribute class, via the invoke primitive, to add itself to the given class, for example:

```
send car bind text index
```

will add an attribute definition to the car class, of type text, called index. This is polymorphic in that if text is replaced with method, the add method of the class ‘method’ will be invoked, which results in a method (in this case called index) being added to the class.

Object.

The object class has the ‘instance’ method for creating instances of classes. A typical message might be:

```
send car instance f209klg
```

The instance method creates an entity, and links it to the given class as a link called ‘instance’.

Member.

The class member is an incarnation of the member entity in the ‘core_model’ and ‘fragments’ fragments. It contains no members, however is a logically ‘correct’ in that both the method and attribute classes, which are derived from member, and to be derived from, member needs to be a class.

Method.

This class contains the method add and modify, to create ‘normal’ and wrapping methods, respectively. An edit method should be implemented.

Attribute.

The entity attribute, derived from member, has the method ‘add’ to create an attribute definition, which is a link between a class and the attribute’s type (class), with a given name.

```
Send car add text index
```

Text.

The class ‘text’, derived from attribute, contains two methods: set and get, for writing and reading text values.

User Workspace.

The user’s workspace is in the directory $HOME/.os . This is where all the user’s work is written (only the top OHP slide gets written). The user’s workspace reuses the object model fragment, probably indirectly, via other fragments, and hence reuses the fragments and core_model fragments. This section briefly describes the typical use of fragments.

Given that users have the development environment, as illustrated in figure 4-3, users can issue the following messages to create a new fragment, place it into the current reuse net, and set this as the new current workspace:

```
invoke fragment new work_space_2
invoke fragment reuse work_space_2 users_work_space_1
invoke fragment set work_space_2
```

This will result in the object space, for this user, as shown in figure 4-4.

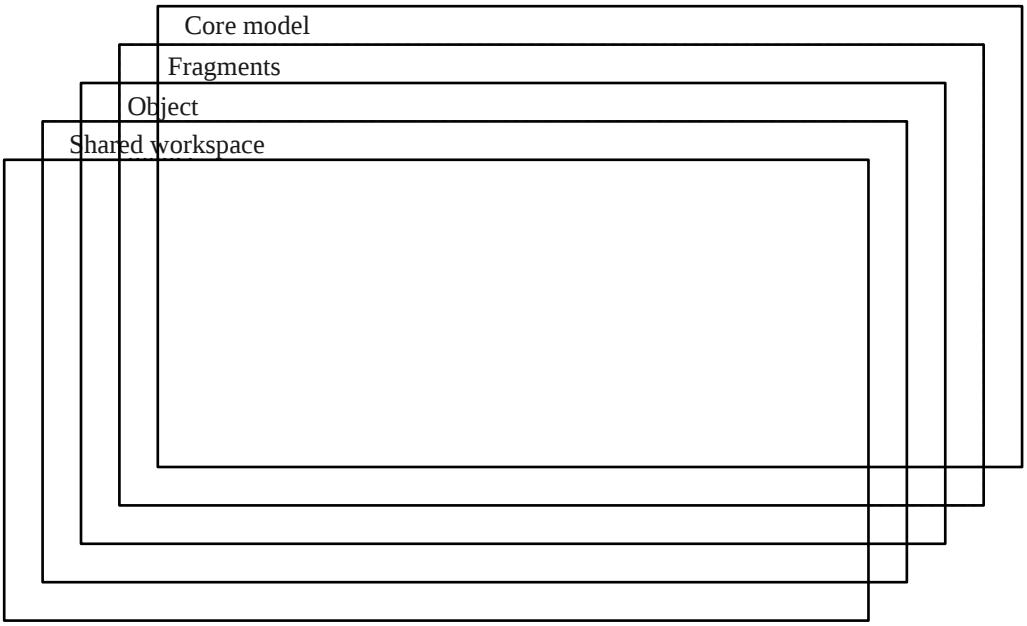


Fig. 4-3

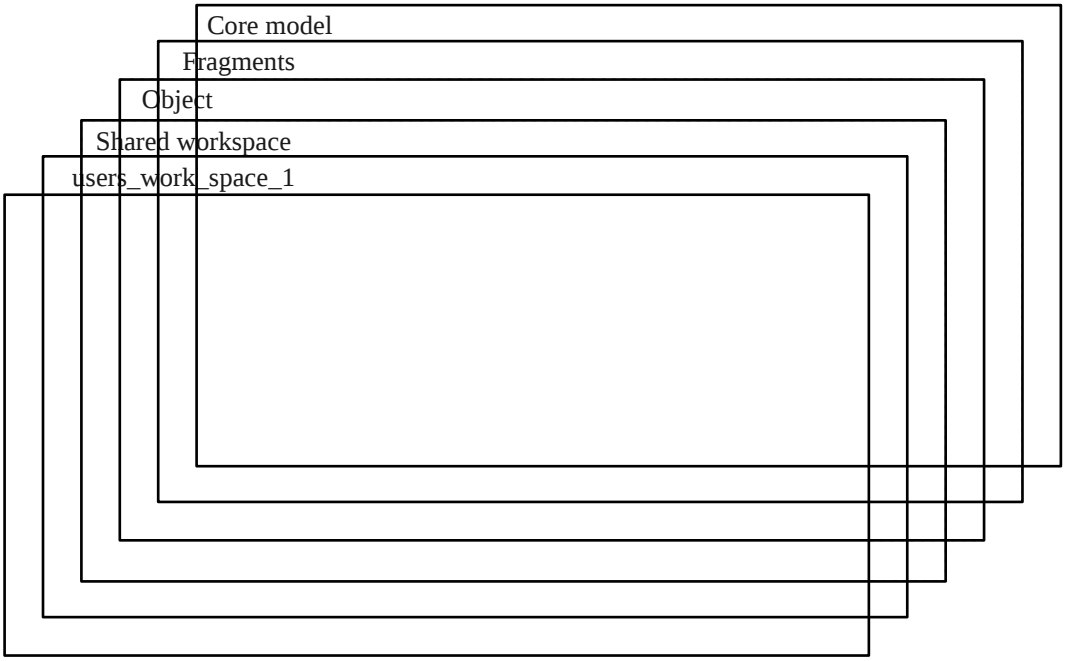


Fig. 4-4

These three method invocations can be condensed into a fragment ‘save’ method.

The user can then modify this new object space by invoke methods found earlier in this chapter. The writes of all methods should be performed in the user's current workspace, for example:

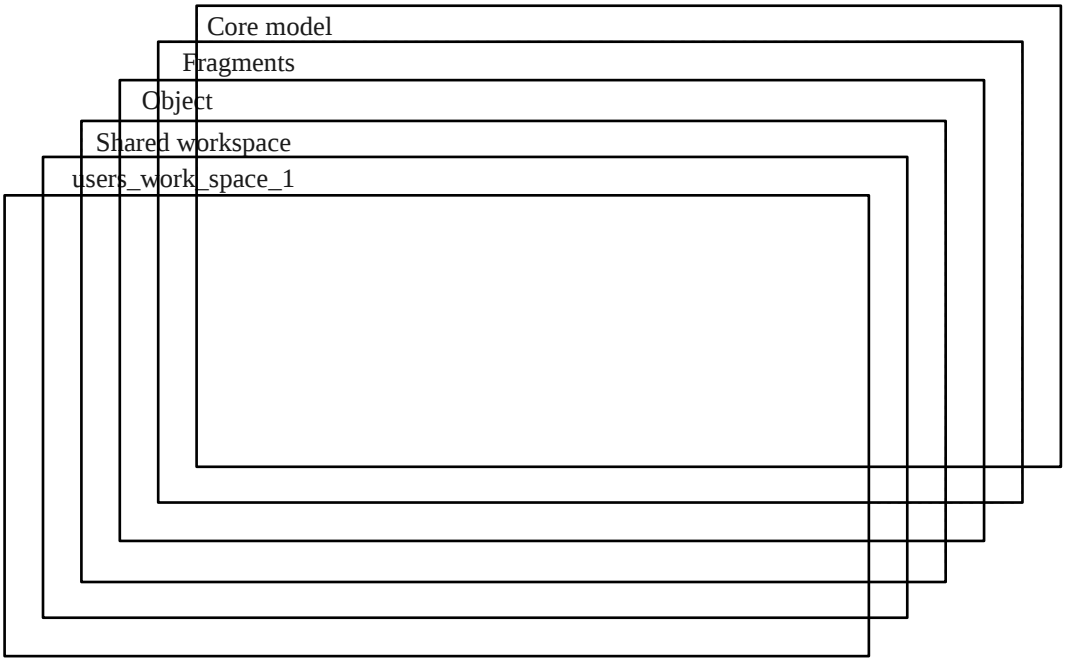


Fig. 4-5

Summary.

This chapter looked in detail at a representation of entities upon a conventional file-system: an object model. This includes the entities and methods that object space contains and the operations upon then (both primitives, and operations layered above these primitives). The object model uses native features where possible. The main features include:

- object space is encapsulated within five 'fragment-aware' primitives. There are operations to provide a mapping between the 'logical view' of object space and the files and directories of object space;
- the hierarchical structures of the underlying file-system are overcome by separating the logical view of object space from its implementation, however native features are used where possible. An abstract object model is defined by a schema, which consists of a class lattice and class composition, and which demonstrates inheritance and polymorphism. The abstract object model is implemented by the introduction of fragments: an object may have an implementation in many

fragments. It is the members of objects that have an implementation as real files, allowing the integration of existing software structures. Fragmentation also allows the introduction of reuse;

- Reuse is provided through the reuse of fragments of object space, allowing schema modification, as well as through the conventional mechanisms of inheritance and overriding/overloading;
- Schema information is part of object space, and so can be ‘versioned’ using the same mechanisms as for versioning object space state and behaviour. Schema additions are stored in separate fragments, so the original schema is still available, unchanged, thus allowing a simple form of schema versioning. However, this means that schema deletions and changes (deletions and additions) are required to be limited to this fragment. A proposal for storing deletions in the same way as additions, using an is-not-a link is proposed in chapter 6;
- A polymorphic message interpreter is provided.

There is yet more work to be done to the basic model described in chapter 4. Dependent, non-shared sub-components are not supported. Back-links are not represented, garbage collection and reconciliation (of fragments and classes) are not addressed. Reuse implies stability, but does not absolutely require it (allowing changes in body/implementation but not in object header/definition). These issues, and others, such as improving schema modification, and the provision of multiple inheritance, are addressed in chapter 6. Chapter 5 goes onto use the facilities described in this section to illustrate the creation and use of object space entities.

Worked Examples.

This section attempts to give a flavour of this project's many application areas. Firstly, a top-down view of the messaging system is presented to illustrate the OO facilities provided by the object model. Following this, a simple web server example shows how the fragments model integrates with the file-system. Lastly an example of how member-value clustering may be represented in fine structure, and how it is displayed in a web client.

Example Messages.

A shell has been prototyped which uses send as it's 'command' interpreter. With '->' as prompt, the following are some example messages outlining this project's OO capabilities. Given the object model illustrated in figure 4-3, the following messages create a person class and instance, and bind and use a name attribute:

```
-> object derive person
-> person bind text name
-> person instance janet
-> janet name set 'Janet Wheatman'
-> janet name get
Janet Wheatman
```

Using the 'send' message interpreter, described in chapter 4, these messages can be invoked from a conventional shell command line. One feature illustrated by this example, is that the send mechanism mostly uses problem-domain specific tokens. Expanding this example, the send program demonstrates the OO facilities supported in this project. For example, if we derive a class, it inherits the composition of its base classes, thus:

```
-> person derive programmer
-> programmer instance martin
-> martin name set 'Martin Wheatman'
-> martin name get
Martin Wheatman
```

Person inherits the derive method, programmer class inherits the name member. Further, we can override methods:

```
-> programmer bind method derive
-> programmer derive john
```

The first message illustrated here shows a degree of polymorphism, in that the (class) method ‘bind’ simply calls the appropriate ‘add’ method - telling the class (in this case method) to add itself to the class programmer (and call the newly bound method ‘derive’). The second message calls this newly bound derive method. Extra functionality could be added to this new method. Alternatively, if the method is to recall the existing method derive, to reuse this method, a ‘modify’ method of the ‘method’ class can be invoked directly:

```
invoke method modify programmer derive
```

In both cases, the generated shell script contains an executable stub. The resultant method from the ‘modify’ method will contain one call to the utility send to re-send the given message into its inheritance net, as described in .

Summary

This is a useful example in that it demonstrates the send program, its OO facilities, and the abstract object model. However, this project is not a genealogy program: the problem domain is contrived. Neither is this project a study of a particular object model, so although the above methods are implemented in this project, a different object model may be implemented, and would behave in a different manner. Further effort is needed to construct an example which demonstrates the type checking made available in this project. We need an example to demonstrate the representation of the abstract object model upon a conventional file-system.

Web Server.

A web server is a daemon process which serves files over a network. Hyper-text mark-up language (HTML), the glue with which web objects are connected, does not impose any structure on web space: it may entirely be contained in files in one directory. However, some structuring is normally attempted, by grouping files into directories. This structuring, because of it hierarchic nature is prone to the conventional problems in using file-systems as object space, as outlined in [Nestor 1986]. The facilities of this project

can be applied a web server for it to become a server of members of (non-hierarchic) objects.

The following example illustrates the mapping of an abstract object space onto a conventional file-system, and the versioning facilities provided by the introduction of fragments. This example concentrates on the filename primitive, described in section . Whilst it does not demonstrate OO facilities, the mechanism behind the reuse net is similar to that of the inheritance net. Since this example works on plain HTML, it avoids invoking either methods or common gateway interface (CGI) scripts, and therefore avoids using the invoke primitive, described in . Further, a web server does not serve composite objects (i.e. just plain files), so the dir_name primitive (section) is also avoided, making for a simple example.

Given a Uniform Resource Locator (URL) passed to a web server, e.g.

http://www.wheatman.demon.co.uk/martin/name

Assuming the initial protocol (i.e. http:) and server (i.e. www.wheatman.demon.co.uk) sections of the URL have been used in locating this server, the path and filename, e.g. /martin/name will be used, by a conventional web server, to retrieve the appropriate file from the file-system.

In a web server implemented using the facilities of this project, the path components will be interpreted as a traversal of object space, with the final component (the filename) being the member name. During the construction of object space, the validity of entities can be checked through the use of a schema. This provides a ‘rigorous approach’ to serving files. In this way, we have the following advantages over the conventional web server,:

- ensure the existence of a member on modification: the web page editor does not edit the identity of a web page, just its content;
- the validation of links within web pages are open to the same checking as are available to conventional web servers, such as WebLint [Bowers 1996], however it can be applied automatically after the modification of the contents of a page;

- ensure that web pages which are superseded, either by being split or deleted, are replaced by relevant superseding pages. This requires the implementation of split and delete methods for the web page object;
- provide ‘the latest’ version of a web page (or optionally the previous n-th version, if implemented in the schema), since edits are written to the latest fragment of object space. The precise nature of this depends on the version control implemented within a fragment, however it should be possible to provide (at least) coarse grained version control, through the use of object space fragments.

The traversal of object space will include, as in the send message interpreter, the traversal of links between objects, as specified in the schema. The example code given in the appendix shows traversal of object space limited to the identification of members. In practice there is the need to support the traversal of object space: allowing object references such as ‘/martin/mother/mother’ .

A drawback of this web server example is that it merely demonstrates the member specification to filename mapping - no type checking against any schema is performed. This is a simple web server.

The Example Programs.

The source code in appendix C is the complete source for two programs. The first program is a simple web server, server1.c . The second comprises of two source files (server2.c and fname.c), and a header file (fname.h) defining the exports from the latter source file. The file server2.c is similar to server1.c except in that it contains a call to the function fname. This is a ‘first-cut’ translation of the filename primitive, as described in chapter 4, and included in appendix A. This allows the resultant web server to search the fragments in object space for the appropriate object member.

These programs were developed by the author from the example program ‘listen’, attributed to Norman Wilson and listed in [Graham 1995], and the HTTP protocol described in [Graham 1995].

In running the program, thus,

```
server1 8080 &
```

the first program serves files on the port 8080 as found in the given directory below the running web server. For example, given the URL:

```
http://localhost:8080/martin/name
```

serves the file name in the directory martin, below the current directory of the web server.

In running the second version, similarly:

```
server2 8080 &
```

this will find members in object space from the current working object space. One limitation of object space is highlighted, in that each user is limited to one object space per user. In practice it should be possible to specify object space as an environment variable: see for further discussion. The operation of these web servers is described below.

A Conventional Web Space Example.

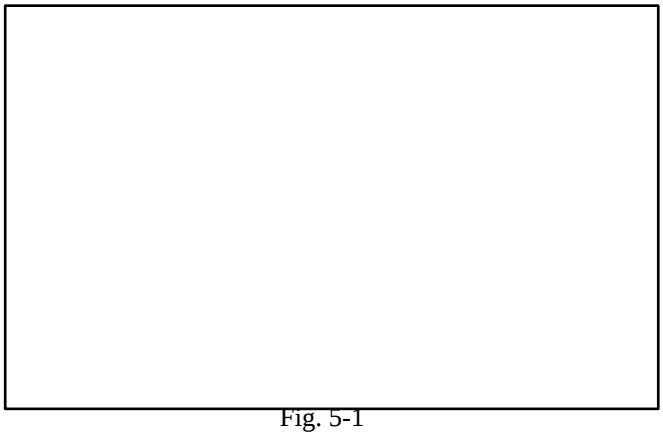


Fig. 5-1

Given the object space outlined in figure 5-1. The entity ‘janet’ has a component with an object valued link to the ‘martin’ entity’s name component. Note the link between the two objects is a fine-to-coarse grained HTML link, i.e. an href anchor attribute, not a coarse grained link as described in chapter 4[8].

[8] Note also that this is a link between names. Typically, in such an object example, the son and name members would be siblings. However, HTML does not impose any schema, so this unstructured example

The URL http://localhost:8080/janet/name will return the web page as shown in figure 5-2, for both the web servers. Following the link ‘Son’, on figure 5-2, gives the page illustrated in 5-3, again for both the conventional and modified web servers.

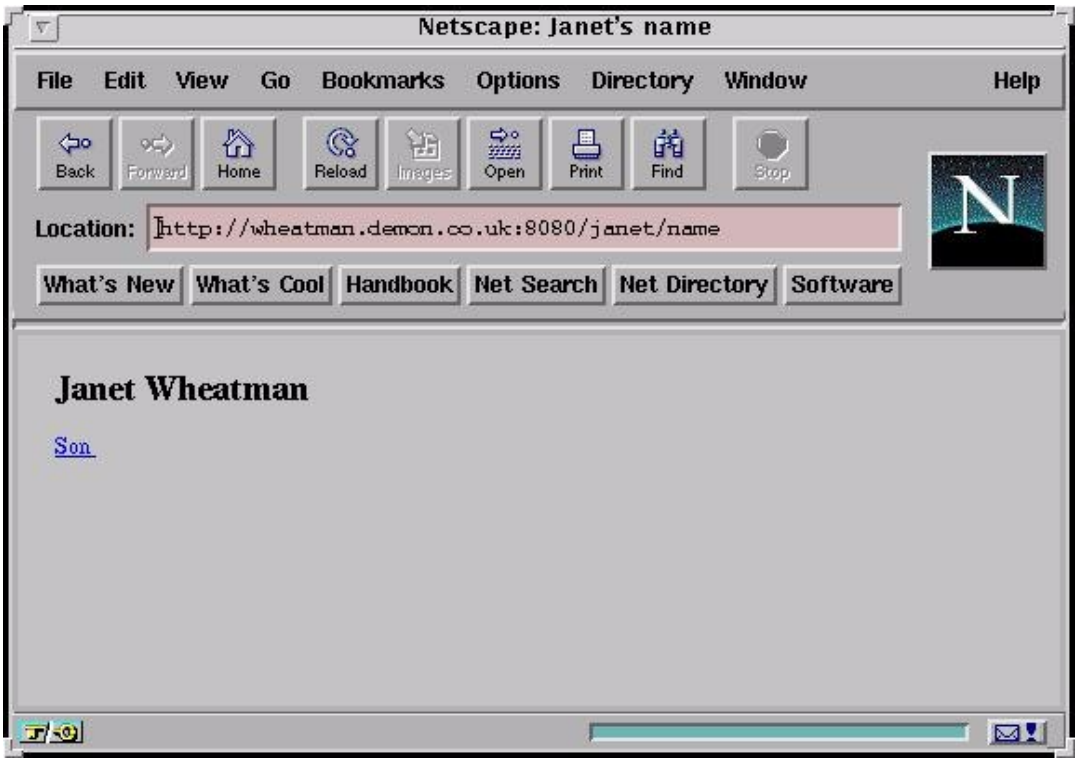


Fig. 5-2

is realistic.

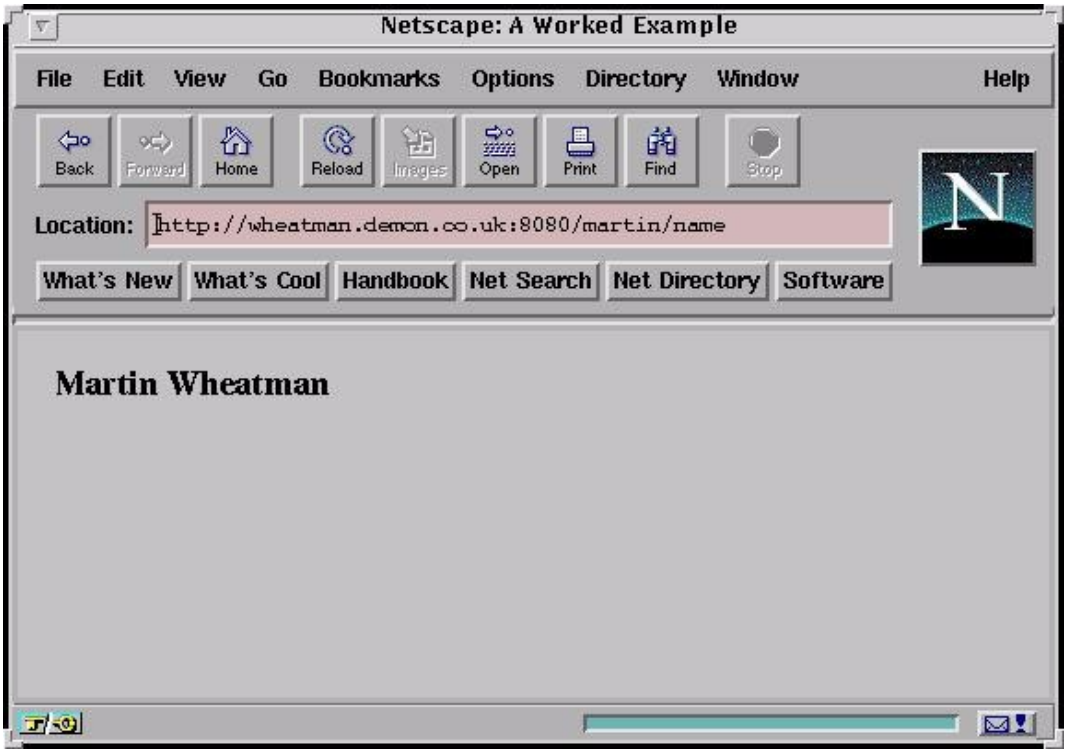


Fig. 5-3

A Fragmented Example.

In the first example, the modified web server operates in the same way as the standard web server. However, on the introduction of object space fragments, it will operate differently.

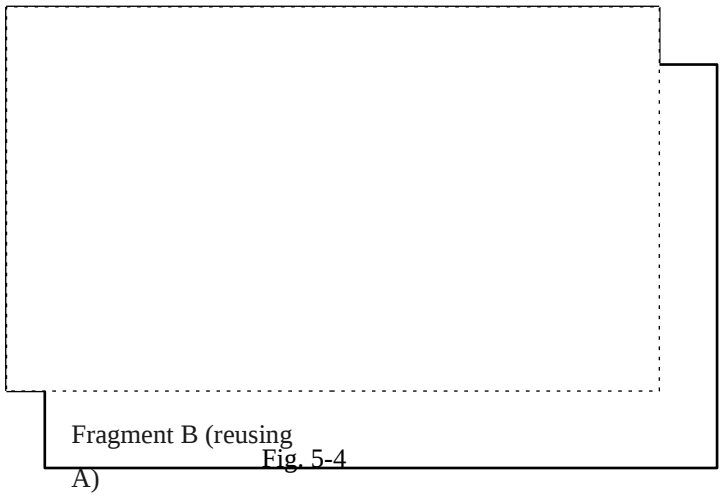


Fig. 5-4

Given the object space outlined in figure 5-4, where a fragment is overlaid on the object space fragment described in 5-1. The new fragment contains the object space delta, including an updated name component of the martin entity.

Using the standard web server, the URL http://localhost:8080/janet/name will not be resolved in this object space: there is no notion of object space reuse, and /janet/name does not exist in the new fragment (Fragment B). Using the modified web server, the above URL will resolve to *exactly the same* HTML file as in figure 5-2: since the web page is not defined in the new fragment, the original one will be served via the reused fragment. However, following the same link will result in the new web page being served (as in figure 5-5). Since this is a fresh query of the object space (Fragment B), and this /martin/name exists in the new fragment. This re-querying is the normal method of retrieving inline images (i.e. the graphical images displayed within a web page).

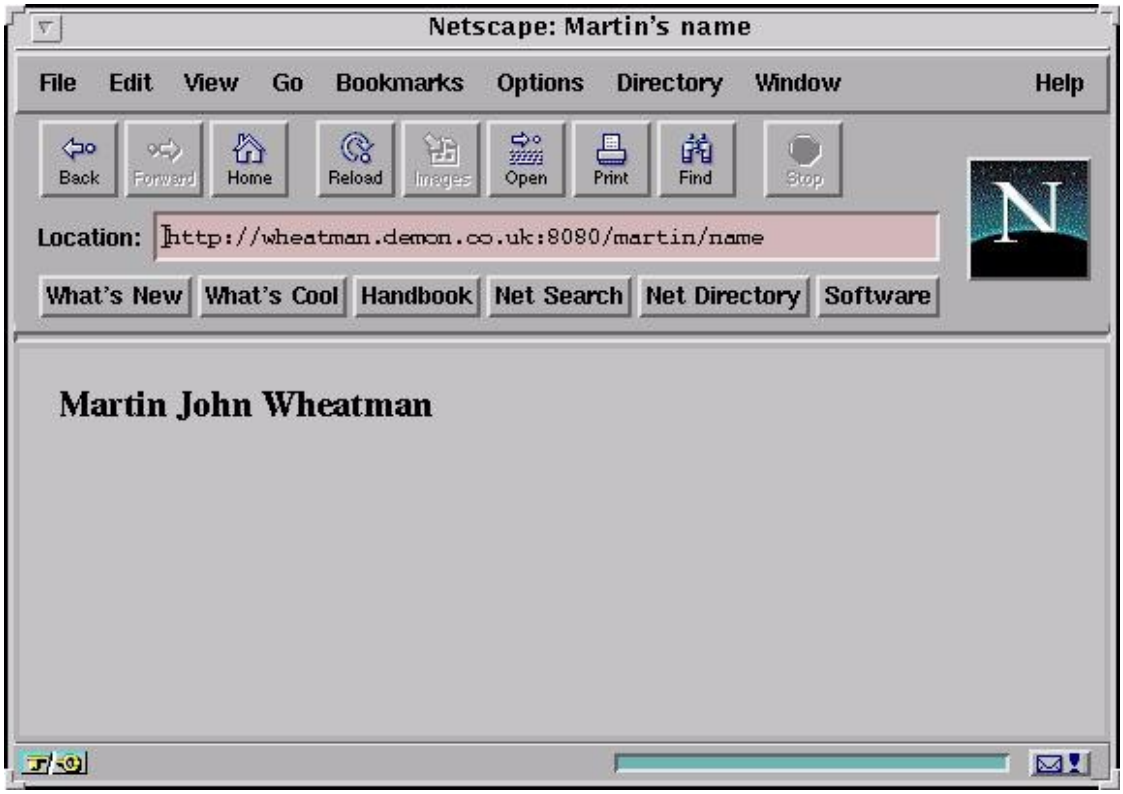


Fig. 5-5

A Client Example.

Given that the WWW notion of an object is a file containing fine structure, it may be argued that this is significantly different to that of an object as represented in this project (i.e. a directory). If member values are represented as files, then an HTML representation of an entity will be a web page of hyper-links to the class member values (file values). Each instance would look the same, and each value would have to be navigated to identify the instance. It is suggested that this is not a good use of [the] hypertext [mechanism]. However, this use of HTML need not prevail.

Since there is no ‘inline text’, like inline images, an entity-to-HTML function is required. Using the ‘overriding’ facilities of this project, it could be implemented as a wrapper to the setting of an attribute (or any member value), at edit time. This proposed wrapper method (see section) may be implemented in the same way as a clustered member, which also has not yet been implemented in this project. Equally, it can be achieved either as a CGI/server-side or as a Java/client-side function, adding, at runtime, the member values to the entity page. This entity-to-HTML function can be located within the class, as any other method. Many different representations may therefore be stored together, for example a family tree network, a personnel record, or an index to a library of pictures all could use the information stored in the same person class and its instance.

The HyperSheet or WebSheet

One suggested application of these wrapper functions, is the inception of a hypertext form of a spreadsheet, where each cell of a table contains either a conventional spreadsheet cell value or a hyper-link to another table. The text representing the hyper-link will be a ‘hot-spot’ value, defined by the co-ordinates of the cell in the linked-to table. A static example of such a ‘Web-sheet’ or ‘Hyper-sheet’ is given below.

As an example of such a WebSheet, the following comma-separated tables may be translated into HTML. The given hyper-link in the outer table (outert), is identified and has its cell ‘value’ (i.e. 3%4), replaced by the appropriate ‘hot-spot’ value in the given inner table (table), defined by the co-ordinates (3, 4). The tables below may be attribute

values in themselves, or an implementation of clustered members as proposed in chapter 6.

outert:

1,2,3,4
1,2,3,4
1,3%4,3,4
1,2,3,4

table:

1,2,3,4
2,5,4,7
3,4,5,8
4,5,66,9

These above comma-separated variable files can be converted to HTML using the csv2html.c code found in appendix D, and called as wrapper functionality to the editing of the member values. The resultant web pages are shown below, in figures 5-6 and 5-7. An editor similar to conventional spreadsheets can be written to add constraints on other cells, such a the summation of other cells.

Summary.

It is not suggested in these examples that the hyper-text links are replaced by the type of links proposed in this project, as stated in . This project is about the structuring of coarse-grained data. This example illustrates how fine structure can be used in a clustered manner (each value in the comma-separated table being a member value), and so circumvent the situation where an object is represented in HTML, individually, as a class-template with links to the particular values. Secondly, emergence of hypertext aware spreadsheet is an interesting notion. It is suggested that with such a HyperSheet format, spreadsheets would reduce in size, becoming hierarchical (or more likely form a directed acyclic graph), and may introduce the typing of cell columns, for example, a particular column may only contain links, another may only contain dates, etc., more akin to columns (or domains) in a relational database.

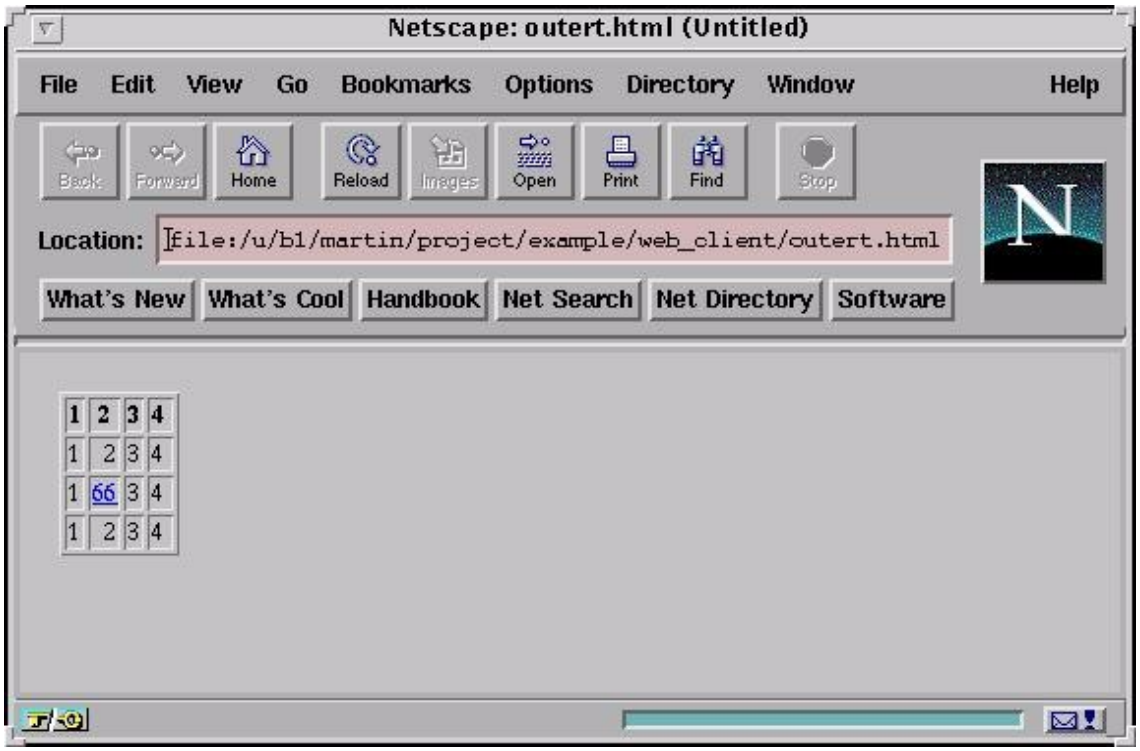


Fig. 5-6

When the ‘66’ hot-spot link is followed, the web page in figure 5-7 is displayed.

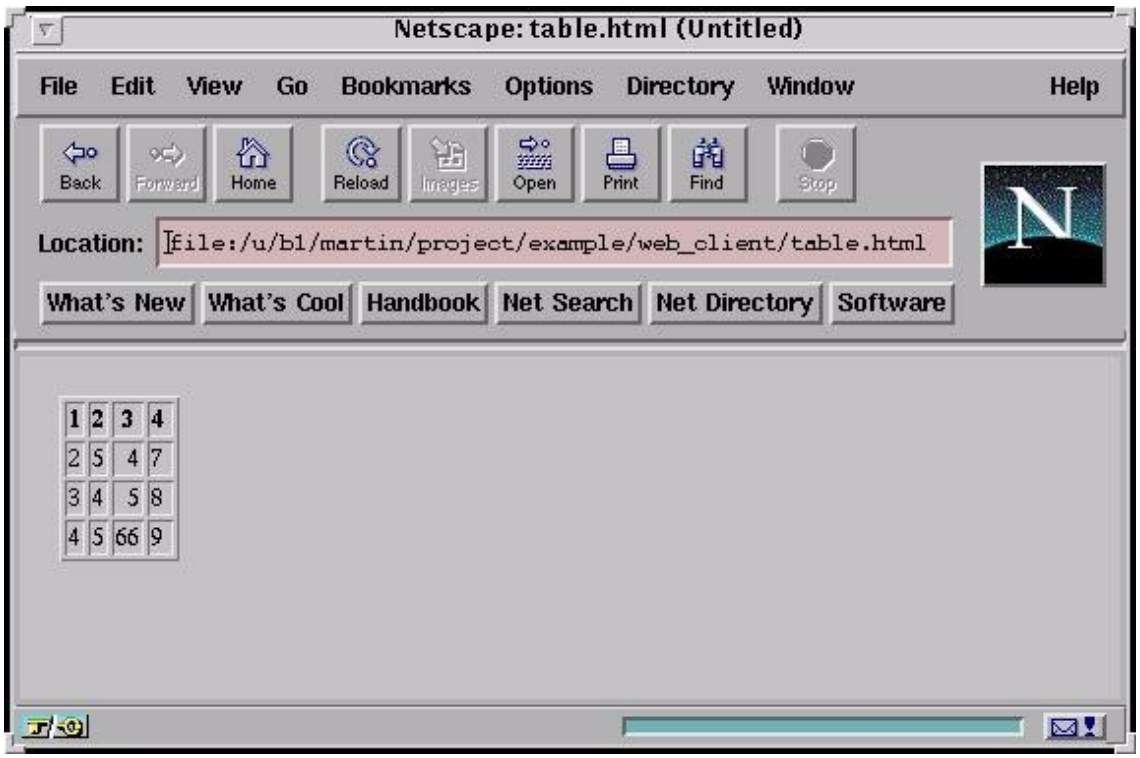


Fig. 5-7

Conclusion.

Aims.

This project aimed to represent an object model on a conventional file-system, including a schema and messaging system.

Achievements.

The prototype methods and utilities outlined in chapters 3 and 4, and listed in appendices A and B, produce a realistic model of persistent objects, upon a conventional file-system. Useful features include a generic approach to reuse by the introduction of object space fragments. This uses a generic mechanism by which entities are linked to each other. Fragments are represented using a mechanism similar in concept to composite objects. Fragments also illustrate a move away from a one-to-one mapping between objects and directories, and members and files. [Nestor 1986] highlights areas where hierarchical file-systems are insufficient in modelling a one-to-one relationship, directly. With the use of fragments, it can be seen that an object may be represented in more that one place: an abstract object model (including the notion of fragments) is implemented in a layer *above* files and directories.

Limitations of the Current Prototype.

Whilst this project has been successful in achieving a realistic object model, this section describes the areas of this project which have not been attempted or are missing from the 'final' prototype. These include:

- a lack of shared and non-shared, independent components,
- a lack of clustered members,
- reconciliation of fragments is not implemented,
- reuse implies stability, but is not enforced,
- roles not implemented, and,
- lack of garbage collection,

and are outlined in more detail below.

Clustered Members and Components

Firstly, there should not be much work in implementing both clustered members and sub-components. It did not initially seem that either would be of great importance to the central object model, since the object model, as it stands, contains the notion of class composition through a has-a link. However, the absence, at least of sub-components, has limited the example in section 5. A typical model of components is that described in [Kim 1990], where there are:

- dependent and independent components, for example an engine, typically, is a dependent component of a car, whereas an owner of a car is independent, and;
- shared and non-shared components, for example an engine is a non-shared component of a car, whereas the owner is shared (i.e. a person may own more than one car).

Currently the object model contains independent components through the has-a linking of members to classes. Dependent components would be implemented by the creation of sub-directories below the dependent object.

The sharing of components needs to be addressed. Currently independent components need to be prevented from being shared. This could be achieved possibly by tagging the attribute definition in some way, making the share-ability of components orthogonal to the type of the component. Dependent components have not been implemented. The sharing of dependent components could be implemented in a fashion similar to the sharing of sub-directories as described in [Ponder 1997]. However, this highlights a potential problem with [Ponder 1997], in that if garbage collection is to work, it must be able to destroy the composite object, without destroying the shared (dependent) component. A better solution would be to force all shared objects to be independent, and rely on some mechanism on removal of objects to note when all the dependent containers of a component have been removed. This becomes a problem, since there may be other (un-reused) fragments recording sharing information.

It is assumed that the implementation of clustered attributes is also relatively straight forward. The most likely mechanism is for the class to hold a file containing the clustered

values of instances, in the same way that a class contains a file containing default values for attributes, which is shared amongst the instances. This will also require some special treatment on the removal of objects. The conversion of clustered (text) attributes into hyper-text is demonstrated in section .

Fragment Reconciliation Is Not Implemented.

The process of reconciliation corresponds to the movement or removal of members (if fully overridden), and the subsequent removal of empty nodes in the ‘reuse lattice’. This can be view as the compressing of the class lattice. Reconciliation must not affect the apparent shape or behaviour of the reuse lattice.

The achievement of reconciliation depends on the existence of back-links in the reuse lattice - ‘reused-by’ links, to ensure that items within the fragment cannot be reached other routes. One of the strengths of this implementation of object space is that it supports reuse and overriding, and reconciliation is a difficult process. Inevitably, there will become situations where it would be advantageous to reconcile fragments: where the benefits of reconciling many fragments, in terms of reduced time in message interpretation, out-weigh the cost of reconciliation. Since overridden methods are irreconcilable, without automatic code editing, the reconciliation of fragments may need to be on a per member basis, and should largely be a manual process. This will include checking if overriding methods re-call the overridden method?

The are probably better ways of optimising object space, such as propagating transitive links to reduce the runtime of binding methods (ensuring that this process can be undone).

Reuse Stability Is Not Enforced.

Reuse implies that the reused item is stable, however this is not enforced in the current prototype. A minor change is required, by introducing a fragment-level ‘attribute’ to signify the stability of the fragment ‘object’. However, the ability to change the content of a fragment may be a useful feature, allowing ‘minor’ changes to be performed without changing the external view of the fragment. Perhaps this stability attribute can be

combined with the mechanism which ensures the class composition and class lattice of this fragment remains immutable: the stability attribute is a file containing the (cached) class composition and lattice of this fragment. However, this requires more work. The immutability of fragments can be further controlled by the introduction of roles.

Roles Implementation.

The notion of roles has been experimented with, but not implemented in the current prototype. The idea is to enforce the correct usage of the native file access mechanisms of the underlying file-system, however, this probably means a non-portable design, and so has been left for future work. The implementation initially prototyped the ownership of methods by a roles manager/object space owner, and the application of methods through the group access permissions. Roles therefore equate to groups in UNIX. The invoking of specific methods in specific classes, by specific users (or groups of users) is preventable, and the sending of messages is specifically allowed by two simple tables, in two separate files. This implementation is inspired by the hosts.allow and hosts.deny files in UNIX, which if present, allow and deny specified users access to a machine's networking functionality.

Garbage Collection

There is currently no garbage collection mechanism in the prototype. This is because objects, as defined in this project, are persistent and that garbage collection pertains solely to objects implemented within the process address space. If a binding to a programming language is developed, then the garbage collection of that language will be used for volatile objects (i.e. objects that are not marked as persistent). However, if created in error, entities can be destroyed. There is no constraint, as yet, to ensure that this has no effect on object space. This constraint should be supported along with other schema modifications, described below. Further, with the implementation of an object buffer, objects would progress successively through slower and slower memory, until eventually stored off-line (for example in a CD jukebox), with a possible, final removal. This progression would be marked by some problem domain specific state. The removal

of entities from object space causes many problems, and possible mechanisms for overcoming or avoiding these is outlined below as further work.

Further Work.

Despite overcoming some of the reservations for using a conventional file-system for a basis of an object management system, [Nestor 1986], the prototype is a sound basis for further work. Apart from the ‘missing features’ are outlined above in , the project is suitable for research into the problems outlined below.

Schema Modification.

The natural evolution of object space is achieved by the addition of classes and members, normally through the deriving of new classes. Through the introduction of fragments, which also contain meta-data, a simple mechanism to allow two schema to coexist is already implemented. The schema is currently the aggregation of all fragments in a reuse lattice: schema modifications are encapsulated within the reusing fragments, so the original schema is still available, unchanged in the reused fragment. However, the mechanisms to remove members, such as isa links, need development.

However, a complete taxonomy of schema changes [Banerjee 1987] should be addressed. This includes the ability to move members around the class lattice, coupled with the ability to re-classify classes, allowing the changing of a class into a leaf node in the class lattice, and allowing its removal without affecting the behaviour of object space, since it will not be reused. The full taxonomy of schema changes should also be subject to schema versioning.

Schema Versioning.

By extending the mechanism of aggregation of object space, schema deletions should also be limited to the reusing fragment in the same way as additions. The ‘is-not-a’ and ‘has-not-a’ links are proposed, changing the representation of a schema from simply an aggregation of fragments, to the schema definition of the current fragment being applied to the schema of the reused fragments. This will allow, for example, the removal of an isa link in a previous fragment to be reversed (i.e. re-added) in the current fragment. Although, the addition of is-not-a and has-not-a links are trivial, the implementation of

this requires extra processing, especially if a generic mechanism (to allow the removal of any user defined transitive link) is implemented. However this may be offset with the introduction of cached schema in reused (and therefore stable) fragments.

In addition, access to the data in reused fragments needs to be addressed, especially where the schema change defines the removal of a member, which is replaced in a later fragment. For example, are the existing values for this member still valid? This is probably a problem domain specific issue, and so a mechanism should be sought to allow either case to be specified. One solution, outlined in [Monk 1993] would be to use update/backdate functions allowing either schema version to be applied. Certainly an orthogonal solution would be necessary, where the validity of ‘old’ values is decided on a per member basis, independent of member type.

Further, since schema reuse is analogous to class inheritance, reuse will encounter similar problems as those found with multiple inheritance, if multiple reuse is not prevented.

Multiple Inheritance.

The object model can support multiple inheritance. However the mechanisms which operate on object space simply work with the first match. An improved send mechanism which works on *arbitrary length messages* has been prototyped, but is not included in this thesis, because of its experimental nature. In this experiment, the context of the message (the entity which the message is sent to) is changed, as required during the interpretation of the message.

Most OO systems allow multiple components, but not multiple inheritance. Multiple components are dealt with by a full path to the component being given, whereas inheritance allows the runtime system to pick the most appropriate route across object space. Hence, the colour of a car object may have to be specified, thus:

```
car.bodywork.paint.colour = red;
```

whereas, an inherited member maybe accessed, thus:

```
car.id = 12345;
```

even though the member id is inherited from vehicle through the path `car -> motorised -> vehicle`.

With a message interpreter working on arbitrary length messages (switching the context), the two mechanisms can be combined to allow the path through object space to be appropriately ‘clarified’, without the absolute path being specified in the message. These mechanisms may also be used to solve schema clashes arising from the multiple reuse policy also allowed at the moment.

Further, default values could be implemented to short-cut this interpretation stage, to specify that a car’s colour is the colour of the bodywork’s paint, rather than the colour of the wheels.

Multiple Views Of Object Space.

The current provision of access to a user’s object space is via a symbolic link in the user’s home directory. This link has a presence in object space because changes to it need to be persistent. However, it may be necessary for a user to log in again with a different view on object space. This would be implemented by a single environment variable, with the initial value of, say ‘$HOME/.os’, which could be changed to reference other persistent pointers into object space, e.g. ‘$HOME/.os1’. Typically, a web server is run as a user which has minimal access rights to the file-system, such as ‘nobody’. By the setting of an environment variable, therefore, each instance of a web server could be started to point to a particular view of object space, by the same user, as highlighted in chapter 5.

Performance Improvements.

The main drawback of the current prototype is the poor performance. Whilst the emphasis in this project has always been on functionality, simple example queries take seconds to perform. Performance improvements can be achieved by the use of:

- binary methods and operations, improving the performance times of each component of object space; and,
- an object buffer, managing a fragment (schema) cache, reducing the latency of accessing object space.

The introduction of an object buffer may required the increasing of the limits on the number of files being open at once, which is often imposed by the operating system. This should be less of a problem given the increasing amounts of memory available, and the reduction of its cost.

Object Model Extension.

It is also hoped that the implementation will be flexible enough to experiment further with object space, to extend object space with useful features. Extensibility should include user-defined transitive relationships, beyond the system defined ‘is a’ and ‘has a’ relationships, such as container-ship and ownership, and include the behaviour of these mechanisms.

Two other types of link need to be implemented: back links and generic transitive links. Back-links implementation two-way links, or relationships. These will need to be represented in the file-system in some form, since they are persistent and also if they are to be orthogonal to (i.e. not dependent on) the forward link.

Transitive links are links that don’t exist, i.e. where a link is implied by other links, such as links inherited in the object model fragment. For example, where > represents some link:

if A > B and B > C, therefore A > C.

A > C is a transitive link: it need not exist, but logically it does. Transitive links may occur through different types of link. Two transitive links are currently provided, ‘isa’ and ‘has’. For example:

```
person has a name
user is a person
martin is a user
```

martin therefore may have a name value, and we can say ‘has a’ is transitive through ‘is a’. To be extensible, generic transitive links will require further work.

Application Areas.

Since this project is a practical one, one area of further work is to produce examples of this project in action. Two areas are outlined here, which have been considered as examples for chapter 5, and which may be addressed in future.

Fragmentation as Configuration Management.

Since the fragmentation model, introduced in this project, allows the versioning of object space, one useful application area of this project is configuration management. Producing a configuration management system using the facilities in this project along with a version control package, such as [Tichy 1982], should be viable. Further, this should tie in with modelling sub-directories as sub-components, allowing normal file-space projects to be imported within a configuration management programme.

Binding to Programming Languages.

One goal of research into OO, is the unification of the volatile memory of the process address space and the persistent memory of the database. Since this project aims to produce a similar object model to that of OOPLs, on a conventional file-system, an obvious goal for further work would be to produce a binding between the objects in OOPLs and the objects in this project.

A simple prototype (in C or C++) would be to introduce a persistent storage class to map volatile, manually allocated memory on to persistent memory. It could be implemented by pre-processing the source code:

- source code for the allocation of memory would be translated into the source code for the creation of a file;
- the source code to access this memory - to write to it, read from it, and to select particular portions will be translated into reads, writes and seeks on a file;
- the naming of persistent objects depends on the position of the persistent variable within the source file, so the scope of variables within the source can be dealt with;
- the issue of the sharing of persistent data, e.g. shared or protected between users, should be possible, but requires further work.

This mapping would allow orthogonal persistence in that the type of memory used to implement memory allocated in the program source is not tied to the object model, and can even be applied to arbitrary data items within a program. The initial prototype may not need to work on a specifically OO programming language. A solution for OOPLs would follow.

Conclusion.

This project has demonstrated OO features on a conventional operating system. It uses files to represent members rather than entities, as is common in other systems. It works well in demonstrating the structure of object space: allowing straightforward access to values in object space. The tools to access object space were initially implemented in shell scripts, for them to be simple and ‘open’.

Further, the notion of fragments has been introduced to support the object model, which is believed to be novel, in an attempt to model reuse and schema versioning. The OO world has progressed since the start of this project, most notably in the introduction of Java, which incorporates many of the ideas outlined in this project. This project is still relevant, however, in that it is a persistent system, which Java is not, and it provides a fresh approach to persistence in utilising operating system facilities.

References.

[Acceta 1986] Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M. Mach : A New Kernel Foundation for Unix Development Proceedings, Summer Usenix Conference, USENIX pp.93-112 1986

[Atkinson 1983] Atkinson, M.P., Bailiey, P.J., Cockshott, W.P., Chisholm, K.J., Morrison, R. An Approach to Persistent Programming The Computer Journal Vol.26 No.4 pp.360-365 1983

[Banerjee 1987] Banerjee, J., Chou, H-T., Garza, J.F., Kim, W., Woelk, D., Ballou, N., Kim, H-J. Data Model Issues For Object-Oriented Applications ACM Transactions on Office Information Systems Vol.5 No.1 pp.3-26 January, 1987

[Berners-Lee 1994a] Berners-Lee, T., Manister, L., McCahill, M. Uniform Resource Locators (URL) RFC1738 <ftp //ds.internic.net/rfc/rfc1738.txt>

[Berners-Lee 1994b] Berners-Lee, T., Cailliau, R, Luotonen, A., Frystyk Nielsen, H., Secret, A. The World-Wide Web Communications of the ACM Vol.37 No.8 pp.76-82 August, 1994

[Berners-Lee 1995] Berners-Lee, T., Connolly, D.W. Hypertext Mark-up Language - 2.0 RFC1866 <ftp //ds.internic.net/rfc/rfc1866.txt>

[Bhasker 1983] Bhaskar, K.S. How object-oriented is your system? SIGPLAN Notices Vol.18 No.10 pp.8-11 October, 1983

[Black 1995] Black, A.P., Walpole, J. Objects to the rescue! or httpd the next generation operating system Operating System Review Vol.29 No.1 pp.91-95 Jan. 1995

[Bloomer 1992] Bloomer, J. Power Programming with RPC O'Reilly and Associates, Inc. ISBN 0-937175-77-3 Sept, 1992

[Booch 1991] Booch, G. Object Oriented Design With Applications The Benjamin/Cummings Publishing Company, Inc. ISBN 0-8053-0091-0 1991

[Bourne 1982] Bourne, S.R. The UNIX System Addison Wesley ISBN 0-201-137917 1982

[Bowers 1996] Bowers, N. Weblint: Quality Assurance for the World-Wide Web Computer Networks and ISDN systems Vol. 28 pp1283-1290 1996

[Bueche 1989] Bueche, E.C., Franklin, M.T., Holley, E.R., Korth, H.F. Oonix: An Object-Oriented Unix Shell Proc. of the 22nd Hawaii International Conference on System Science(HICSS) pp.928-935, 1989.

[Butterworth 1991] Butterworth, P., Otis, A., Stein, J. The GemStone Object Database Management System Communications of the ACM Vol.34 No.10 pp.64-77 October, 1991

[Cardelli 1990] Cardelli, L. Semantics Of Multiple Inheritance Readings in Object-Oriented Database Systems Ed. Zdonik, S.B., Maier, D. Pub. Morgan Kaufmann Publishers, Inc. ISBN 1-55860-000-0 pp.59-83 1990

[Cattell 1997] Cattell, R.G.G., Barry, D., Bartels, D., Berler, M., Eastman, J., Gamerman, S., Jordan, D., Springer, A., Strickland, H., Wade, D. The Object Database Standard - ODMG 2.0 Morgan Kaufmann, Series in Data Management Systems 1-55-860-463-4 1997

[Chase 1992] Chase, J.S., Levy, H.M., Baker-Harvey, M., and Lazowska, E.D. Opal: A Single Address Space System For 64 Bit-Architectures Proc. Of The Third Workshop on Workstation Operating Systems(WWOS III) Key Biscayne pp80-85 Apr 1992.

[Codd 1970] Codd, E.F. A Relational Model of Data for Large Shared Data Banks CACM Vol.13 No.6 June 1970

[Continuus] <http //www.continuus.com/> Continuus Software Corporation, Irvine, Ca. USA.

[Cusack 1992] Cusack, E. Object Oriented Modelling In Z For Open Distributed Systems Open Distributed Processing (Ed. Demeer, J., Heymer, V., Roth, R.) Ch. 39 No. pp.167-178 1992

[Date 1990] Date, C.J. An Introduction To Database Systems, Volume 1, Fifth Ed. Addison-Wesley ISBN 0-201-52878-9

[Dearle 1992] Dearle, A., Rosenberg, J., Henskens, F., Vaughan, F., Maciunas, K. An Examination of Operating System Support for Persistent Object Systems 25th Hawaii International Conference on System Sciences IEEE Computer Society Press Vol.1 pp.799-789 1992

[Dearle 1994] Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindstrom, A., Rosenburg, J., Vaughan, F. Grasshopper An Orthogonally Persistent Operating System Computing Systems Vol.7 No.3 pp.289-312 1994

[Deux 1990] Deux et al., O. The Story of O2 IEEE Transactions on Knowledge and Data Engineering Vol.2 No.1 pp.91-107 March, 1990

[Deux 1991] Deux et al., O. The O2 System Communications of the ACM Vol.34 No.10 pp.34-48 October, 1991.

[Duke 1991] Duke, R., King, P., Rose, G., Smith, G. The Object-Z Specification Language Technology Of Object Oriented Languages and Systems - Tools (5th Int. Conf. On) Vol.5 No.53 pp.465-483 July-Aug, 1991

[Fielding 1997] Fielding, R., Gettys, J., Mogul, J.C., Frystyk, H., Berners-Lee, T. Hypertext Transfer Protocol -- HTTP/1.1 draft-ietf-http-v11-spec-rev-00.txt INTERNET-DRAFT HTTP Working Group July 30, 1997

[Flanagan 1997] Flanagan, D. Java In A Nutshell 2nd Edition O'Reilly and Associates 1-56592-262-X May, 1997

[Gelinas 1995] Gelinas, J. UMSDOS HOW-TO Ref. <http://sunsite.unc.edu/LDP/HOWTO/HOWTO-INDEX.html Version. 1.1 Nov, 1995

[Graham 1995] Graham, I.S. HTML Sourcebook John Wiley & Sons, Inc. ISBN 0-471-11849-4 1995

[Hartman 1995] Hartman, J.H., Ousterhout, J.K. The Zebra Striped Network File System ACM Transactions on Computer Systems Vol.13 No.3 pp.274-310 August, 1995

[Heidemann 1994] Heidemann, J.S., Popek, G.J. File-System Development with Stackable Layers ACM Transactions on Computer Systems Vol.12 No.1 pp.58-89 February, 1994

[Howard 1997] Howard, R. Simple Object Persistence with STORE_TABLE Journal Of Object Oriented Programming Vol. No. pp.49-54 June, 1997

[Ingham 1996] Ingham, D., Caughey, S., Little, M. Fixing the "Broken-Link" Problem The W3Object Approach Computer Networks and ISDN Systems Vol.28 No.711 pp.1255 1996

[Ipsys] <http://www.lincolnsoftware.com/>

[Ishikawa 1993] Ishkawa, H., Suzuki, F., Kozakura, F., Makinouchi, A., Miyagishima, M., Izumida, Y., Aoshima, M., Yamane, Y The Model, Language and Implementation of an Object-Oriented Multimedia Knowledge Base Management System ACM transactions On Database Systems Vol.18 No.1 pp.3-50 March, 1993

[Kernighan 1988] Kernighan, B.W., Ritchie, D.M. The C Programming Language, 2nd Ed. Prentice Hall ISBN 0-13-110362-8 1988

[Khoshafian 1990] Khoshafian, S.N., Copeland, G.P. Object Identity Readings in Object-Oriented Database Systems Ed. Zdonik, S.B., Maier, D. Pub. Morgan Kaufmann Publishers, Inc ISBN 1-55860-000-0 pp.37-46 1990

[Kim 1990] Kim, W. An Introduction to Object-Oriented Databases Pub. The MIT Press ISBN 0-262-11124-1 1990

[Linux] <http://www.li.org>

[Merle 1996] Merle, P., Gransart, C., Geib, J-M. CorbaWeb- A Generic Object Navigator Computer Networks and ISDN Systems Vol.28 No.711 pp.1269 1996

[Meyer 1997] Meyer, B. Object-oriented Software Construction, 2nd Ed. Prentice Hall ISBN 0-13-629155-4

[Monk 1993] Monk, S.R. A Model For Schema Evolution in OO Database Systems PhD. Thesis Lancaster University 1993

[Nestor 1986] Nestor, J.R. Toward a Persistent Object Base Technical Report, SE1-86-TM-8. Software Engineering Institute July, 1986

[OMG 1995] OMG Common Object Request Broker Architecture and Specification 2.0 July, 1995

[Ponder 1997] Ponder, C. Organizing UNIX Directories as Lattices Operating Systems Review Vol.31 No.4 pp.72-77 Oct. 1997

[Purdy 1987] Purdy, A., Schuchardt, B., Maier, D. Integrating an Object Server with Other Worlds ACM Transactions on Office Information Systems Vol.5 No.1 pp.27-47 January, 1987

[Redell 1980] Redell, D., Dalal, Y., Horsley, T., Lauer, H., Lynch, W., McJones, P., Murray, H., Purcell, S. Pilot: An Operating System For A Personal Computer Communications of the ACM Vol.23 No.2 pp.81-92 February, 1980

[Rosenblum 1992] Rosenblum, M., Ousterhout, J.K. The Design and Implementation of a Log-Structure File System ACM Transactions on Computer Systems Vol.10 No.1 pp.26-52 February, 1992

[Rozier 1988] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P, Neuhauser, W. CHORUS Distributed Operating Systems Computing Systems Vol.1 No.4 pp.305-367 1988

[Satyanarayanan 1997] Satyanarayanan, M., Spasojevic, M. AFS and the Web Competitors or Colaborators? Operating System Review Vol.31 No.1 pp.18-23 January, 1997

[Scott 1990] Scott, M.L., LeBlanc, T.J., Marsh, B.D. Multi-model parallel programming in Psyche Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming pp.70-78 March, 1990

[Shapiro 1993] Shapiro, M., Dickman, P., Plainfosse, D. SSP Chains Robust, Distributed References Supporting Acyclic Garbage Collection Decentralised and Distributed Systems Vol.39 No.30 pp.45- 1993

[Shirley 1992] Shirley, J., Hu, W., Magid, D. Guide to Writing DCE Applications, 2nd Ed. O'Reilly and Associates, Inc 1-56592-045-7 1992

[Skarra 1986] Skarra, A.H., Zdonik, S.B. The Management of Changing Types in an Object-Oriented Database OOPSLA Proceedings pp.483-495 September, 1986

[Snyder 1990] Snyder, A. Encapsulation and Inheritance in Object-Oriented Programming Languages Readings in Object-Oriented Database Systems Ed. Zdonik, S.B., Maier, D. Pub. Morgan Kaufmann Publishers, Inc ISBN 0-55860-000-0 pp.84-94 1990

[Srinivasan 1997] Srinivasan, V., Chang, D.T. Object Persistence in object-oriented applications IBM Systems Journal Vol. 36 No.1 pp.66-87 1997

[Stonebraker 1990] Stonebraker, M., Rowe, L.A., Hirohama, M. The Implementation Of POSTGRES IEEE Transactions on Knowledge and Data Engineering Vol.2 No.1 pp.125-142 March, 1990

[Stonebraker 1991a] Stonebraker, M., Kemnitz, G. The POSTGRES Next-Generation Database Management System Communications of the ACM Vol.34 No.10 pp.78-92 October, 1991

[Stonebraker 1991b] Stonebraker, M.J. Operating System Support for Database Management Readings In Database Systems Ed. Stonebraker, M.J. pp.167-173, 1991.

[Stroustrup 1987] Stroustrup, B. The C++ Programming Language Addison-Wesley Publishing Company, Inc. ISBN 0-201-12078-X July, 1987

[Swinehart 1986] Swinehart, D., Zellweger, P. Beach, R., Hagmann, R. A structural view of the Cedar programming environment ACM Transactions on Programming Languages and Systems Vol.4 No.8 October, 1986

[Tanenbaum 1987] Tanenbaum, A.S. Operating Systems: Design and Implementation Prentice Hall ISBN 0-13-637331-3 1987

[Thomas 1988] Thomas, I. The PCTE Initiative and the PACT project ACM SIGSOFT Software Engineering Notes Vol.13 No.4 pp.52-56 Oct, 1988

[Tichy 1982] Tichy, W.F. Design, Implementation and Evaluation of a Revision Control System Proc. of the 6th International Conference on Software Engineering, IEEE, Tokyo Sept. 1982

[Touretzky 1986] Touretzky, D. The Mathematics of Inheritance Systems Morgan Kaufmann Palo Alto CA. 1986

[Tracz 1988] Tracz, W. Software Reuse Maxims ACM SIGSOFT Software Engineering Notes Vol.13 No.4 pp.28-31 Oct, 1988

[Yang 1996] Yang, J.J., Kaiser, G.E. An Architecture for Integrating OODbs with WWW Computer Networks and ISDN Systems Vol.28 No.711 pp.1243 1996

[Zdonik 1986] Zdonik, S.B. Maintaining Consistency in a Database with Changing Types SIGPLAN Notices Vol.21 No.10 pp.120-127 October, 1986.

[Zdonik 1990] Zdonik, S.B., Maier, D. Readings in Object-Oriented Database Systems Morgan Kaufmann Publishers, Inc. ISBN 1-55860-000-0 1990.

Appendix A.

Layered Utility Source Code.

This appendix gives a ‘warts-and-all’ snapshot of the layered utilities described in chapters 3 and 4. There is an extra script, ‘misa’, which is only used by the ‘has’ script, and so should be implemented as part of this script.

filename

```
#!/bin/sh
# filename: returns the filename of the given entity+member
# provides a mapping from entity+member -> filename

filename_core() {
   for dirs in $*; do
        cd $dirs 2> /dev/null
        if [ -f "$entity/$name" ]; then
           member=`/bin/pwd`/$entity/$name
           return
        fi
        cd -
   done

   # if we've not found the member yet, search the reused fragments...
   for dirs in $*; do
        cd $dirs 2> /dev/null
        for i in `ls -d *.reuse 2> /dev/null`; do
           filename_core $i
        done
        cd -
   done
}

# Start here.....

write=0; recon=0
if [ "$1" = "-w" -o "$1" = "-r" ]; then
```

```
        if [ "$1" = "-w" ]; then
                write=1
        else
                recon=1
        fi
        shift
fi

if [ "$1" =  "-help" ]; then
        echo "Usage: `basename $0` -help|[-w] <entity> <member>"
        exit
fi

if [ $# -lt 2 -o $# -gt 3 ]; then
        echo "Usage: `basename $0` -help|[-w] <entity> <member>"
        exit 1
fi

entity=$1
name=$2

if [ $write -eq 1 ]; then # write
        cd $HOME/.os
        if [ ! -d $entity ] ; then
                mkdir $entity # should've been done by dir_name ???
        fi
        member=`/bin/pwd`/$entity/$name
elif [ $recon -eq 1 ]; then # reconcile
        filename_core `ls -d $HOME/.os/*.reuse`
else # read
        filename_core $HOME/.os
fi
echo $member
```

dir_name

```
#!/bin/sh
# dir_name: returns the directory name of the given entity
# provides the mapping: entity name -> directory name

dir_name_core() {
   for dirs in $* ; do
        cd $dirs 2> /dev/null
        if [ -d "$name" ]; then
                entity=`/bin/pwd`/$name
                return
        fi
        cd -
   done

   for dirs in $* ; do
        cd $dirs 2> /dev/null
        for i in `ls -d *.reuse 2> /dev/null`; do
           if [ "$entity" = "" ]; then # not found yet...
                dir_name_core $i
           fi
        done
        cd -
   done
}



# Start here.....................................
# first deal with any switches...
write=0; recon=0
if [ "$1" = "-w" -o "$1" = "-r" ]; then
        if [ "$1" = "-r" ]; then
                recon=1
        else
                write=1
        fi
        shift
fi
```

```
# Check params
if [ $# -ne 1 -o "$1" = "-help" ]; then
        echo "Usage: `basename $0` -help | -w|-r <entity>"
        if [ "$1" = "-help" ]; then
        echo "    -r    : for reconciling: return the next one up the "
        echo "    -w    : find the one we'd write to"
        fi
        exit 1
fi
name=$1
export name

if [ $write -eq 1 ]; then
        dir_name_core $HOME/.os
        if [ -n "$entity" ]; then
                entity=`basename $entity`
        else
                entity=$name
        fi
        entity="$HOME/.os/$entity"

        if [ ! -d $entity ] ; then
                mkdir $entity
        fi
else
        if [ $recon -eq 1 ]; then
                dir_name_core `ls -d $HOME/.os/*.reuse 2> /dev/null`
        else
                dir_name_core $HOME/.os
        fi
fi

if [ -n "$entity" ]; then
        echo $entity
fi
```

invoke

```
#! /bin/sh
#invoke: runs a given method of a given entity

if [ $# -lt 2 ]; then
   echo "Usage: invoke [-f] <entity> <method> [<params>...]"
   exit
fi

cd $HOME/.os
cd `/bin/pwd`

find=0
if [ $1 = "-f" ]; then
   shift
   find=1
fi

context=$1; shift
method=$1; shift

bfs() {
   for file in `ls -d *.reuse 2>/dev/null`; do
         if [ -x $file/$context/$method ]; then
            echo $dl/$file/$context/$method
            return
         fi
   done
   for file in `ls -d *.reuse 2>/dev/null`; do
         olddl=$dl
         export dl=$dl/$file
         cd $file
         bfs $context $method
         export dl=$olddl
         cd -
```

```
    done
}

if [ -f $context/$method ]; then
    meth=`/bin/pwd`/$context/$method
else
    export dl=`/bin/pwd`
    meth=`bfs $context $method`
fi


if [ "" = "$meth" ]; then
    echo "invoke: $context $method not found."
    exit 99
else
    if [ $find -eq 1 ]; then
            methd=`dirname $meth`
            cd $methd
            echo `/bin/pwd`/$method $*
    else
            $meth $*
            exit $?
    fi
fi
```

isa

```
#!/bin/sh
#isa - using the entity links method

isa_core() {
    cd $1
    invoke entity links class| awk '{print $1" is a "$3}'
    invoke entity links instance| awk '{print $1" is a "$3" "$2}'
    cd -
}
```

```
basic_isa () {
   isa_core  $HOME/.os | sort -u
}

# Start Here.....................................................

if [ $# -gt 2 -o "$1" = "-help" ]; then
        echo "Usage: `basename $0` [-all] | [ <derived>|%  [<base>] ]"
        exit
fi

if [ $# -eq 0 ]; then
        basic_isa
elif [ $# -eq 1 -a $1 = "-all" ]; then
        basic_isa | all
elif [ $# -eq 1 ]; then
        basic_isa | all | grep "^$1 "
elif [ $# -eq 2 ]; then
        TMP_FILE=/tmp/isa.$$

        basic_isa | all > $TMP_FILE
        if [ "$1" = "%" ]; then
                grep " $2 $" $TMP_FILE
        else
                if [ `grep "^$1 " $TMP_FILE| grep " $2 $"| wc -l` -gt 0 ];
                then
                        # May not be so: -d means were in the right dir
                        if [ -d `dir_name $1` ]; then
                                echo Instance
                        else
                                echo Yes
                        fi
                fi
        fi
        rm -f $TMP_FILE
fi
```

all.c (compiling to all)

```
/* all.c - Martin Wheatman 16th July 1995
   Description - Takes output from isa and returns expanded net.
*/

#include <stdio.h>
#include <strings.h>

#define ONE 0
#define NEAREST 1
#define ALL 2

/* remember:- !TRUE != FALSE */
#define FALSE 0
#define TRUE 1
#define TAB_SZ 200

void
Usage (char *prog) {
  printf ("Usage: %s -[one|nearest|all] <derived> <member>\n", prog);
}

char *B[TAB_SZ], *D[TAB_SZ], *I[TAB_SZ];
char *b[TAB_SZ], *d[TAB_SZ], *inst[TAB_SZ];
int BDsize = 0;
int dbindex=0;

void
print (int sz, char * a[], char* b[], char* c[]) {
  int i;

  for (i=0; i<sz; i++)
    printf ("%s is a %s %s\n", a[i], b[i], c[i]);
}
```

```
int
add (char *d, char *b, char* ins) {
  int i, found = FALSE;

  for (i=0; i < BDsize; i++)
    if (!strcmp(D[i], d) && !strcmp(B[i], b)) {
      found = TRUE;
      break;
    }

  if (FALSE == found) {
    D[BDsize] = (char *) malloc (strlen(d)+1);
    B[BDsize] = (char *) malloc (strlen(b)+1);
    I[BDsize] = (char *) malloc (strlen(ins)+1);
    strcpy (D[BDsize], d);
    strcpy (B[BDsize], b);
    strcpy (I[BDsize++], ins);
  }
}

void
deref (char *name, char *indirect, char *ins) {
  int j;

  add (name, indirect, ins);
  for (j=0; j<dbindex; j++)
    if (!strcmp(d[j], indirect))
      deref (name, b[j], ins);
}

main (int argc, char *argv[]) {
  char line [80], first[32], second[8], third[8], fourth[32], fifth[10];
  int i;

  gets (line);
  while (!feof(stdin)) {
```

```
    strcpy (fifth, "");
    sscanf (line, "%s %s %s %s %s\n", first, second, third, fourth, fifth);
    if (!strcmp (second, "is")) {
      d[dbindex] = (char*) malloc (strlen(first)+1);
      b[dbindex] = (char*) malloc (strlen(fourth)+1);
      if (!strcmp(fifth, "instance")) {
           inst[dbindex] = (char*) malloc (strlen(fifth)+1);
           strcpy (inst[dbindex], fifth);
      } else {
           inst[dbindex] = (char*) malloc (1);
           strcpy (inst[dbindex], "");
      }

      strcpy (d[dbindex], first);
      strcpy (b[dbindex++], fourth);
    }
    gets (line);
  }

  /*print (dbindex, d, b, inst);
  printf ("expanded is\n");*/
  for (i=0; i<dbindex; i++)
    deref (d[i], b[i], inst[i]);

  print (BDsize, D, B, I);
}
```

has

```
#!/bin/sh
# SCRIPT: has - prints out the entity/member mappings

as_core() {
        cd $1
        ls -ld ./[A-Za-z]*/[A-Za-z]*.* 2>/dev/null | sed 's/[\.\/]/ /g' \
                | awk '
                ($11=="method") {print $9" has a "$10" "$11}
```

```
                ($12=="method") {print $9" has a "$11" "$13" "$12}
                ($11=="member") {print $8" has a "$10" "$11" "$13}
                ($12=="member") {print $8" has a "$11" "$10" "$12" "$14}'
        for i in `ls -d *.reuse 2> /dev/null`; do
                has_core $i
        done
        cd -
}

basic_has() {
#    has_core $HOME/.os | sort -u
   misa attribute | awk -F. '{print $1" "$2}' | \
   awk '{ print $1 " has a " $2" "$3" "$4}'
   misa -m | awk -F. '{print $1" "$2}' | \
   awk '{ print $1 " has a " $3" "$2}'
}

if [ $# -eq 0 ]; then
        basic_has
elif [ "$1" = "-all" -a $# -eq 2 ]; then
        for i in `isa $2 | awk '{print $4}'`; do
                basic_has | grep "^$i"
        done
elif [ $# -eq 1 -a "$1" != "-help" ]; then
        basic_has | grep "^$1" #type is grep'd value type not the has...
elif [ $# -eq 2 ]; then
        if [ $1 = "%" ]; then
                basic_has | awk '(name==$4){print}' name=$2
        else
                basic_has| grep "^$1 has a $2"
        fi
elif [ $# -eq 3 ]; then
        basic_has | grep "^$1 has a $2 $3"
else
        echo "Usage: `basename $0` [ -help | entity [ member [ type ] ] ]"
        echo "     : `basename $0` $*"
fi
```

misa

```
#!/bin/sh
# general member query - q'n'd

lname=class
method=0
if [ $# -gt 0 ]; then
        export lname=$1
        if [ $1 = "-m" ]; then
           method=1
        fi
fi

misa() {
   cd $1
   if [ $method -eq 1 ]; then
        for i in `ls */* 2>/dev/null`; do
           if [ -x $i -a ! -d $i ]; then
                echo $i | awk -F/ '{print $1 " method " $2}'
           fi
        done
   else
        ls -l */*.$lname 2>/dev/null| \
        awk -F/ '{print $1 " " $2 " " $3}' |\
        awk 'BEGIN {ln=ARGV[1];ARGC=1;OFS=":"}
           {print $9" "$10" "$13}' $lname
   fi
   for reused in `ls -d *.reuse 2>/dev/null`; do
        misa $reused
   done
   cd -
}

misa $HOME/.os | sort -u
```

inherit

```
#!/bin/sh
# SCRIPT: inherit       [-nocheckself]- don't check self message generated
#                       -one|-nearest|-all - what to return -one (for the mo)
#                       <derived>     - the derived entity
#                       <inherited>   - what class(es) is this a member of?
#       returns: 0=not found, 2/4=attribute, 1/3=method, 1/2=class
search_first()
{
        exit_code=0
        type="`has $2 $3`"
        mtype="`echo $type | awk '{print $5}'`"
        mstype="`echo $type | awk '{print $6}'`"
#echo "*TYPE:$type*mtype:$mtype*mstype:$mstype*"
# This check should be parameterised??? What are we looking for?
#if [ "$mtype" = "attribute" -o "$mtype" = "method" ]; then
#if [ -d $mtype.class ]; then
        if [ "$mstype" != "class" -a "$type" != "" ]; then
                echo $2
                if [ $1 = "-one" ]; then
                        # workout exit code...
                        if [ "$mtype" = "method" ]; then
                                exit_code=3
                        elif [ "$mtype" = "member" ]; then
                                exit_code=4
                        fi
                fi
        elif [ "$mstype" = "class" ]; then
                echo $2
                if [ $1 = "-one" ]; then
                        # workout exit code...
                        if [ "$mtype" = "method" ]; then
                                exit_code=1
                        elif [ "$mtype" = "member" ]; then
                                exit_code=2
                        fi
                fi
```

```
        fi
#fi
        export exit_code
}

#cd $OBJECT_SPACE

exit_code=0
#if [ $1 = "-class" ]; then
#       class_opt="-class"
#       class_id=".class"
#       shift
#fi

if [ $1 = "-nocheckself" ]; then
        shift
else # Check self first
        search_first $*
        if [ $exit_code -gt 0 ]; then
                exit $exit_code
        fi
fi

(isa ; has) | a.out $1 $2 $3
```

inherit.c (compiling to a.out)

```
#include <stdio.h>
#include <strings.h>
#define ONE 0
#define NEAREST 1
#define ALL 2

/* remember:- !TRUE != FALSE */
#define FALSE 0
#define TRUE 1
```

```
#define TAB_SZ 200

void Usage (char *prog)
{
  printf ("Usage: %s -[one|nearest|all] <derived> <member>\n", prog);
}

main (int argc, char *argv[])
{
  char line [80], first[32], second[8], third [8],
       fourth [32], fifth[32], sixth [32];
  int opt, acindex = 0, dbindex = 0, found, foundfirst, h, j, k;
  char *derived, *member; /* to reference parameters */
  char *a[TAB_SZ], *b[TAB_SZ], *c[TAB_SZ], *d[TAB_SZ];
  int p[TAB_SZ], m[TAB_SZ];

  FILE* log;
  log=fopen ("/home/martin/inherit.log", "w");

  /* Interpret parameters */
  if (0 == strcmp (argv [1], "-one")) {
    opt = ONE;
  } else if (0 == strcmp (argv [1], "-nearest")) {
    opt = NEAREST;
  } else if (0 == strcmp (argv [1], "-all")) {
    opt = ALL;
  } else {
/*
  No Usage message- this is required to run silently.
    Usage (argv[0]);
*/
    exit (-1);
  }
  derived = argv[2]; member = argv [3];
  fprintf (log, "Ok\n");

  gets (line);
```

```
  while (!feof(stdin)) {
    sscanf (line, "%s %s %s %s %s %s\n", first, second, third, fourth, fifth, sixth);
    if (0 == strcmp (second, "has")) {
      a[acindex] = (char *) malloc (strlen(fourth)+1);
      c[acindex] = (char *) malloc (strlen(first)+1);
      strcpy (c[acindex], first);
      strcpy (a[acindex], fourth);
      m[acindex] = (0 == strcmp (fifth, "method"))?1:2;
      p[acindex] = (0 == strcmp (sixth, "class"))?0:2;
/*printf ("Found ca %s/%s, is :%s:%s:%d:%d:\n", first, fourth, fifth, sixth, m[acindex],
p[acindex] );*/
      acindex++;
      sixth [0] = '\0';
    } else if (0 == strcmp (second, "is")) {
      d[dbindex] = (char *) malloc (strlen(first)+1);
      b[dbindex] = (char *) malloc (strlen(fourth)+1);
      strcpy (d[dbindex], first);
      strcpy (b[dbindex++], fourth);
    }
    gets (line);
  }

/*printf ("Ok to process\n");*/
  /* Ok, lets do some processing */

  foundfirst = FALSE;
  for (found = TRUE; found == TRUE;) {
    fprintf (log, "Looking thru' db for derived (%s)\n", derived);
    found = FALSE;
    for (h=0; h<dbindex; h++) {
      fprintf (log, " Comparing d[h] (%s) with derived (%s)\n", d[h], derived);
      if (0 == strcmp(d[h], derived)) {
          found = TRUE;

          fprintf (log, "  Found, looking thru' ca for b[h] (%s)\n", b[h]);

          for (j=0; j<acindex; j++) {
```

```
            fprintf (log, "    Cmp'ing c[j] (%s) with b[h] (%s)\n", c[j], b[h]);
            if (0 == strcmp (c[j], b[h])) {
              fprintf (log, "    Checking a[j] (%s) for member (%s)\n", a[j], member);
              if (0 == strcmp (a[j], member)) {
                fprintf (log, "FOUND: %s", b[h]);
                printf ("%s\n", b[h]);
/*              printf ("%s %d %d\n", b[h], m[j], p[j]);*/
                if (opt == ONE)
                    exit (m[j]+p[j]);
                if (opt == NEAREST) {
                    foundfirst = TRUE;
                    found = FALSE; /* Terminate at the next level change */
                }
              }
            }
          }
          /* Shorten inheritence net:                                    */
          /*  derived entries with what were looking for - must go first ! */
          /*  don't shorten if -n and now found first at this level       */
          if ((opt != NEAREST) || (foundfirst != TRUE))
            for (k=0; k<dbindex; k++) {
           fprintf (log, "comparing d[k] (%s), with b[h] (%s)\n", d[k], b[h]);
              if (0 == strcmp(d[k], b[h]) && d[k]) {
                fprintf (log, "COPYing over d[k] (%s)", d[k]);
                free(d[k]);
                d[k] = (char *) malloc (strlen(derived)+1);
                strcpy (d[k], derived);
                fprintf (log, " with derived (%s)\n", d[k]);
              }
          }
          /* remove the current entries - must go last! */
          fprintf (log, "NULLing b[h] (%s) and d[h] (%s)\n", b[h], d[h]);
/* These two lines are a problem to SunOS/Solaris */
/*        b[h] = NULL;  Don't bother freeing these! */
/*        d[h] = NULL; */
          strcpy (b[h], "");
          strcpy (d[h], "");
```

```
      } /* if strcmp d derived */
    } /* for h=0 ... */
  } /* for found */
  fclose (log);
}
```

send

```
#!/bin/sh
# SCRIPT: send- simple message interpreter- determines and invokes
# methods

member_type()
{
        has $1 $2 | awk '($5=="method"){ print "method"}\
                        ($5=="value") { print "value" }\
                        {print $6}'
}

multi_sw="-one" # ["-one"|"-nearest"|"-all" ]: only "-one" supported

one=$1; two=$2
if [ "$1" = "-generated" ]; then
        shift
        generated="-nocheckself"
        export generated
        # ...this is where the 'magic' comes in $1 is /a/b/c/entity/meth
        dir=`dirname $1`
        one=`basename $dir` # the entity
        two=`basename $1` # the method name
        context=$saved_context
fi
shift; shift

if [ `isa|grep ^$one|wc -l` -gt 0 ]; then
        entity=$one
fi
```

```
if [ -n "$entity" ]; then # found an entity...
        if [ "$context" = "" ]; then # maintain given entity as context
                context=$entity
                saved_context=$entity
                export saved_context
        fi
        member=`inherit $generated $multi_sw $entity $two`
        rc=`echo $?`
        if [ $rc -eq 1 -o $rc -eq 2 ]; then
                class=""
        fi
        if [ $rc -eq 1 -o $rc -eq 3 ]; then # person derive martin
                invoke $class $member $two $context $*
        elif [ $rc -eq 4 -o $rc -eq 2 -o $rc -eq 3 ]; then
        # martin name set...
                mtype=`member_type $member $two`
                method=`inherit $generated $multi_sw $mtype $1`
                rc=`echo $?`
                if [ $rc -eq 3 -o $rc -eq 1 ]; then
                # $1 was the method, e.g.
                        tfirst=$1; shift # del $1 so we can pass on $*
                        invoke $class $method $tfirst $context $two $*
                else
        echo "send: No $1, with $entity:$context $member:$mtype rc=$rc"
                fi
        elif [ "$two" = "isa" -o "$two" = "has" ]; then
           # martin isa person
                $two $one $*
        else
                echo "send: Can't find member '$two', given '$entity'."
        fi
elif [ -x `which $one 2> /dev/null | awk '{print $1}'` ]; then
        # isa ...
        $one $two $*
elif [ -x `which $two 2> /dev/null | awk '{print $1}'` ]; then
        # % isa ...
```

```
        $two $one $*
else
        echo "Cannot find $one in database `pwd`"
fi
```

Appendix B.

Object Space Member Code.

This appendix gives a snapshot of the current object space's prototype methods. For clarity's sake, only the outermost method in the reuse net is given.

entity/create

```
#!/bin/sh
# entity/create
#echo CALL `basename $0` $*

if [ $# -eq 0 -o $# -gt 2 ]; then
        echo "Usage: {entity} create <name>"
        exit
fi

. $PROJECT/scripts/assert
assert exists $HOME/.os

cd $HOME/.os
mkdir $1 2>/dev/null
```

entity/link

```
#!/bin/sh
# entity/link
#echo CALL `basename $0` $*

if [ $# -ne 3 ]; then
        echo "Usage: {entity} link <from> <link-type> <to>"
        echo "Given: {entity} `basename $0` $*"
        exit
fi

. $PROJECT/scripts/assert
assert exists $HOME/.os
assert exists $1
assert exists $3
```

```
invoke entity create $1
cd $HOME/.os/$1
ln -s ../$3 $2
```

entity/destroy

```
#!/bin/sh
# entity destroy

if [ $# -lt 1 ]; then
    echo "Usage: entity destroy <entity>"
    echo "Given: $0 $*"
    exit
fi

exists=`dir_name $1`
while [ "$exists" != "" ]; do
    rm -rf $exists
    exists=`dir_name $1`
done
```

member/name

```
#! /bin/sh
# fragments/member/name - echos the name of the member (next number if -n)

numeric=0
if [ "$1" = "-n" ]; then
    numeric=1
    shift
fi

if [ $# -ne 3 -a $# -ne 2 ]; then
    echo "Usage: name [-n] <entity> <type> [<name>]"
    echo "Given: `basename $0` $*"
    exit
fi
```

```
context=`invoke entity context $1`
shift

if [ $# -eq 1 ]; then
   if [ $numeric = 1 ]; then
        n=`ls -1d $context/*.$1 2>/dev/null|wc -l`
        echo $context/`echo $n`.$1
   else
        echo $context/$1
   fi
else
   if [ $numeric = 1 ]; then
        n=`ls -1d $context/$1.*.$2 2>/dev/null|wc -l`
        echo $context/$1.`echo $n`.$2
   else
        echo $context/$1.$2
   fi
fi
```

fragment/new

```
#!/bin/sh
# fragment new

if [ $# -eq 0 -o $# -gt 2 ]; then
        echo "Usage: fragment new <name> [<reuses>]"
        exit
fi

#if name is relative e.g. martin rather than /home/martin
if [ -d .. -a -w .. ]; then
        cd ..
fi

mkdir $1
cd $1
```

```
here=`/bin/pwd`
cd -

if [ $# -eq 2 ]; then
        cd $2
        there=`/bin/pwd`
        cd -
        invoke fragment reuse $here $there
fi
```

fragment/set

```
#!/bin/sh

if [ $# -eq 0 ]; then
        if [ -d $HOME/.os ]; then
                cd $HOME/.os
                /bin/pwd
        fi
elif [ $# -eq 1 -a -d $1 ]; then
        cd $1
        here=`/bin/pwd`
        cd
        rm .os
        ln -s $here .os
elif [ $# -eq 1 -a ! -d $1 ]; then
        echo "fragment set: no $1 - must be abs. or rel to $HOME/.os."
else
        echo "Usage: $0 [<dir>]"
fi
```

fragment/reuse

```
#!/bin/sh
# fragment/reuse

if [ $# -eq 1 ]; then
   reusing=$HOME/.os
```

```
elif [ -d "$HOME/.os/../$1" ]; then
    reusing=$HOME/.os/../$1
    shift
elif [ -d "$1" ]; then
    reusing=$1
    shift
else
    echo Sorry, $1 is not a sibling fragment.
    exit
fi

if [ $# -ne 1 ]; then
    echo "Usage: fragment reuse [<name>] <reused>"
    exit
fi

if [ -d $HOME/.os/../$1 ]; then
    reused=../$1
else
    echo Reused $1 is not a sibling fragment, sorry
    exit
fi

cd $reusing
ln -s $reused `ls -1d *.reuse 2>/dev/null| \
    wc -l|awk '{print $1}'`.reuse
```

class/derive

```
#!/bin/sh
#object_model/class/derive
#echo CALL: `basename $0` $*

if [ $# -ne 2 ]; then
        echo "Usage: class derive <from> <to>"
        exit
```

```
fi

. $PROJECT/scripts/assert
assert exists $1
assert NOT exists $2

invoke entity create $2
if [ $? -ne 0 ]; then
   echo class/derive: entity create failed. continuing...
fi
invoke class implies $2 $1
```

class/bind

```
#!/bin/sh
# object_model/class/bind
echo CALL `basename $0` $*

if [ $# -ne 3 ]; then
        echo "Usage: class bind <class> <type> <name>"
        echo "Given: class bind $*"
        exit
fi

. $PROJECT/scripts/assert
assert exists $1
assert exists $2

invoke entity create $1
member=`invoke member name $1 $3`
echo bind member is $member

echo invoke $2 add $1 $3
send $2 add $1 $3
```

class/implies

```
#! /bin/sh
```

```
# object_model/class/implies - link a base/derived class pair
echo CALL `basename $0` $*

if [ $# -ne 2 ]; then
    echo "Usage: class implies fr to"
    echo "Given: class implies $*"
    exit
fi

. $PROJECT/scripts/assert
assert exists $1
assert exists $2

invoke entity create $1
invoke entity link $1 `invoke member name -n $1 class` $2
```

object/instance

```
#!/bin/sh
# object_model/object/instance - create an instance entity
#echo CALL `basename $0` $*

if [ $# -ne 2 ]; then
        echo "Usage: object instance <class> <name>"
        echo "Given: object instance $*"
        exit
fi

. $PROJECT/scripts/assert
assert exists $1
assert NOT exists $2

invoke entity create $2
invoke entity link $2 instance $1
```

attribute/add

```
#! /bin/sh
```

```
# attribute add
#echo CALL {attribute} `basename $0` $*

if [ $# -ne 3 -a $# -ne 4 ]; then
   echo "Usage: {attribute} add [-n] <type> <entity> <name>"
   echo "Usage: {attribute} add      text    user    name"
   echo "Given: {attribute} `basename $0` $*"
   exit
fi

if [ "$1" = "-n" ]; then
   n="-n"; shift
fi

invoke entity create $2
member=`invoke member name $n $2 $3 attribute`

invoke entity link $2 `basename $member` $1
```

method/add

```
#! /bin/sh
# object model/method/add
#echo CALL {method} `basename $0` $*

if [ $# -ne 3 -a $# -ne 4 ]; then
   echo "Usage: {method} add [-n] <class> <entity> <name>"
   echo "e.g.:  {method} add method car drive"
   echo "Given: {method} `basename $0` $*"
   exit
fi

if [ "$1" = "-n" ]; then
   n="-n"; shift
fi

member=`invoke member name $n $2 $3`
```

```
if [ $? -ne 0 ]; then
   echo call to member failed >&2
   exit 99
fi

echo "#!/bin/sh"> $member
echo "# METHOD: $2 $3">> $member
echo ""> $member
echo "echo CALL \$0 \$*">> $member
echo ""> $member
chmod u+x $member
```

method/modify

```
#!/bin/sh
# Modify - inherit a method and modify it...
#echo CALL {method} `basename $0` $*

if [ $# -ne 2 -a $# -ne 3 ]; then
        echo "Usage: {class} `basename $0` <class> <method> [<name>]"
        exit
fi

if [ "$1" = "-class" ]; then
        class=$1
        shift
fi

# 1. Validate parameters...
. $PROJECT/scripts/assert
assert exists $1

from=`inherit -one $1 $2`
rc=`echo $?`
if [ $rc -eq 1 ]; then
        priv_ext='.class'
elif [ ! $rc -eq 3 ]; then
```

```
        echo "$0: $2 is not a method"
        exit
elif [ "$from" = "" ]; then
        echo "$0: internal error: no method $2 inherited by $1, rc=$rc"
        exit
fi

# 2.check if it already exists?????
file=`filename -w $1 $2`
if [ -f $file ]; then
        echo $file already exists
        exit
fi

# 3. Check prospective method
if [ $# -eq 3 ]; then # a name change
        method_name=$3
        command="$2"
else
        method_name=$2
        command="\$0"
fi

# 4. Build reused method
echo "#!/bin/sh"> $file
echo "# METHOD: $1 $method_name">> $file
echo ""> $file
echo "echo CALL {$1} \$0 \$*">> $file
echo ""> $file
echo "send -generated $command \$*">> $file
chmod u+x $file
```

text/set

```
#!/bin/sh
# object_model/text/set - sets a text attribute value.
```

```
if [ $# -lt 3 ]; then
        echo "Usage: text set <object> <attrib> <value>..."
        echo "Given: text set $*"
        exit
fi

entity=$1;
attrib=$2;
shift; shift

echo $* > `filename -w $entity $attrib value`
```

text/get

```
#!/bin/sh
# object_model/text/get - get the entity/member value

if [ $# -ne 2 ]; then
        echo "Usage: text get <entity> <member>"
        echo "Given: text get $*"
        exit
fi

location=`filename $1 $2 value`
if [ -f "$location" ]; then
        cat $location
else
        echo "$1 does not have a value for '$2'"
fi
```

Appendix C.

Web Server Example Code.

The following source code is the complete source for the two web server programs, as described in section . The first program is a simple web server, server1.c . The second comprises of two source files (server2.c and fname.c), and a header file (fname.c) defining the exports from the latter source file. The file server2.c is similar to server1.c except in that it contains a call to the function fname, which is a "first-cut" translation of the filename script, as included in chapter 4. This allows the resultant web server to search the fragments in object space for the appropriate object member. Each file is described in further detail below.

These programs were developed by the author from the example program "listen", attributed to Norman Wilson and listed in [Graham 1995], and the HTTP protocol described in [Graham 1995]. The first program can be compiled by the following command:

```
cc -o server1 server1.c
```

The second program is compiled by the command:

```
cc -o server2 server2.c fname.c
```

In running this code, thus:

```
server1 8080 &
```

the first program serves file on the port 8008 as found in the given directory below the running web server. For example, given the URL:

```
http://localhost:8080/text.txt
```

serves the file text.txt in the current directory. In running the second version, thus:

```
server2 8080 &
```

this will find members in objects space from the current working object space. Here we see another limitation, in that each user is limited to one object space per user, we also

need to specify which object space in the environment: see chapter 6 for further discussion. Given that the current fragment reuses another containing the member (file) text.txt, and the URL

http://localhost:8080/text.txt

is requested, this version of the web server will serve this member, although this is not a direct path from the current directory to the file text.txt in the sibling directory/fragment. On modifying this member, when the new member is written back into the current fragment, it is this member that is served, and when a new fragment is created, this member is still served, until superseded in the new fragment. This may be attained by the implementation of a PUT method.

Program 1.

server1.c:

```
 1 #include <stdio.h>
 2 #include <fcntl.h>
 3 #include <sys/types.h>
 4 #include <sys/socket.h>
 5 #include <netinet/in.h>
 6 #include <netdb.h>
 7
 8 extern int errno;
 9
10 #define ERROR(n,str) { perror (str); exit (n); }
11 #define HEADER_OK "HTTP/1.0 200 OK\nContent-type:text/plain\n\n"
12 #define NO_METH "HTTP/1.0 501 OK\n\
13 Content-type:text/html\n\n<h1>Error 501: Invalid method.</h1>\n"
14 #define NO_FILE "HTTP/1.0 404 OK\n\
15 Content-type:text/html\n\n<h1>Error 404: No such file.</h>\n"
16 #define TX(sd,str) write(sd, str, strlen(str));
17 #define RX(sd,buf) read(sd, buf, sizeof(buf));
18
19 int
```

```
20 main(int argc, char *argv[]) {
21   int ld, addrlen, sd, rc, port;
22   struct sockaddr_in addr, name;
23   char buf [256], *end, str [80];
24
25   if (2 != argc || (argc == 2 && 0 >= (port=atoi(argv[1]))) ) {
26     fprintf (stderr, "Usage: %s <port(+ve)>\n", argv[0]);
27     exit (1);
28   }
29
30   if ( (ld = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0 )
31     ERROR (1, "socket")
32
33   name.sin_family = SOCK_STREAM;
34   name.sin_port = htons(port);
35   name.sin_addr.s_addr = INADDR_ANY;
36   if ( bind (ld, (struct sockaddr*)&name, sizeof (name)) < 0)
37     ERROR (2, "bind")
38
39   fprintf (stderr, "Listening on %d.\n", ntohs(name.sin_port));
40   if ( listen (ld, 5) < 0 )
41     ERROR (3, "listen");
42
43   while (1) {
44     FILE* f;
45
46     addrlen = sizeof (addr);
47     if ((sd = accept (ld, (struct sockaddr*)&addr, &addrlen)) < 0)
48       ERROR (4, "accept")
49
50     if ( fork () )
51       close (sd);
52     else {
53       close (ld);
54       RX (sd, buf);
55       printf ("%s: req: %20.20s\n", argv[0], buf);
56       if (!strncmp (buf, "GET ", 4)) {
```

```
57
58        if (end = (char*)strchr (buf+5, (int)' '))
59          *end = '\0';
60
61        if (f = fopen (buf+5, "r")) {
62          TX (sd, HEADER_OK);
63          while (!feof(f)) {
64            strcpy (str, "");
65            fgets (str, 80, f);
66            TX (sd, str);
67          }
68          printf ("%s: served file: %s\n", argv[0], buf);
69          fclose (f);
70        } else
71        TX (sd, NO_FILE);
72      } else
73        TX (sd, NO_METH);
74      exit (0);
75    }
76  }
77 }
```

Lines 10-17 define some useful macros, which are used throughout the code to make it more readable. The RX and TX macros make direct reads to and from the socket, which requires the length of the message to be specified. The macros NO_METH, NO_FILE and HEADER_OK are the only replies made by the web server, other than the content of served files. The HTTP protocol specifies that a request is made using a "method". The only method supported by this web server is the GET method - the method which serves files. If another method is requested (such as the POST method) the HTML message NO_METH is returned, displaying an appropriate message in the user's web client (browser). If a GET method is specified, but the file requested does not exist, the HTML message NO_FILE is returned, which again, is displayed in the user's browser. If a GET method is specified and the file exists, the HEADER_OK is returned, followed by the contents of the file.

Lines 30-41 set up a socket to listen to a port. The remaining lines of the program is a loop which accepts requests for files. The fork allows the child to service this request, whilst the parent returns to listen for subsequent requests. The HEADER_OK header is returned in line 62, followed by a loop in lines 63-67 to copy the file to the client socket.

Program 2.

fname.h:

```
 1 #ifndef FNAME_H
 2 #define FNAME_H
 3
 4 void fname (char* flag, char* entity, char *member, char *name) ;
 5
 6 #endif
```

fname.c:

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <sys/stat.h>
 5 #include <sys/types.h>
 6 #include <dirent.h>
 7 #include <unistd.h>
 8 #include <malloc.h>
 9
10 #include "fname.h"
11
12 #define DEF_LIST(l,sz) int sz=0; char** l=NULL;
13 #define NEW_LIST(l,sz) sz=0; if(l)free(l); l=(char**)malloc(++sz);
14 #define ADD_LIST(l,sz,str) { \
15   l=(char**) realloc (l, ++sz); /* why +2 ?? */ \
16   l[sz-2] = (char*)malloc (strlen (str) + 1); \
17   strcpy (l[sz-2], str); \
18   l[sz-1] = NULL; }
```

```
19 #define SMALL 80
20 #define OSPACE ".os"
21
22 void
23 filename_core(char* dirs[], char* entity, char* member,char* name){
24   int i = -1;
25   char wd [1024];
26   char pwd [80];
27   char temp [80];
28   struct stat buf;
29   DIR* d;
30   struct dirent* de;
31   DEF_LIST(list,sz);
32
33   for (i=0; dirs[i]; i++) {
34     getcwd (pwd, 80);
35     chdir (dirs[i]);
36     getcwd(temp, 80);
37     sprintf (name, "%s/%s/%s", temp, entity, member);
38     if (0==stat (name, &buf) ) {
39       return;
40     }
41     strcpy (name, "");
42     chdir (pwd);
43     strcpy (pwd, "");
44   }
45
46   /* # if member not found, search the reused fragments... */
47   for (i=0; dirs[i]; i++) {
48     getcwd (pwd, 80);
49     chdir (dirs[i]);
50     d = opendir (dirs[i]);
51     while ((de = readdir (d)) && !strcmp("", name)) {
52       if (!strcmp ((de->d_name+strlen(de->d_name)-6), ".reuse")) {
53      NEW_LIST(list,sz);
54      ADD_LIST(list,sz,de->d_name);
55      filename_core(list, entity, member, name);
```

```
56       }
57     }
58     chdir (pwd);
59     strcpy (pwd, "");
60   }
61 }
62
63 void
64 fname (char* flag, char* entity, char *member, char *name) {
65  int write=0, recon=0;
66  DIR* d;
67  struct dirent* de;
68  char home [SMALL], pwd [SMALL],
69    ospace [SMALL], temp [SMALL];
70  struct stat buf;
71  DEF_LIST(list,sz);
72
73  if (!strcmp (flag, "-w") || !strcmp (flag, "-r") ) {
74    if ( !strcmp (flag, "-r") )
75      recon=1;
76    else
77      write=1;
78  }
79
80  if (!strcmp (flag, "-help") ) {
81    fprintf (stderr, "Usage: fname -help|-w <entity> <name>\n");
82    fprintf (stderr,
83         "     -r     : for reconciling: one beyond the one read\n");
84    fprintf (stderr,
85         "     -w     : find the one to write to\n");
86    exit (1);
87  }
88
89  strcpy (home, (char*)getenv("HOME"));
90  sprintf (ospace, "%s/%s", home, OSPACE);
91  if ( write ) {
92    chdir (ospace);
```

```
 93    if ((-1 != stat (entity, &buf)) && ! S_ISDIR(buf.st_mode) )
 94      mkdir (entity, 0x777);
 95    sprintf (name, "%s/%s/%s", ospace, entity, member);
 96  } else if ( recon ) {
 97    d = opendir (ospace);
 98    NEW_LIST(list,sz)
 99    while (de = readdir (d))
100      if (!strcmp ((de->d_name+strlen(de->d_name)-6), ".reuse")) {
101        sprintf (temp, "%s/%s", ospace, de->d_name);
102        ADD_LIST(list,sz,temp);
103      }
104    filename_core (list, entity, member, name) ;
105  } else {
106    NEW_LIST(list,sz)
107    ADD_LIST(list,sz,ospace);
108    filename_core (list, entity, member, name) ;
109  }
110 }
```

Fname is a first attempt translation of the "filename" primitive described in chapter 4, implemented as a C function. "first attempt" means that it is a straight translation, including no improvements and defunct code, such as lines 80-87.

Line 10 includes fname.h, listing the prototypes of the exported functions. The macros in lines 12-18 implement a simple dynamically allocated list. Lines 22-61 implement the main functionality of the filename function, firstly looking for a file representing the given entity/member reference, in the current context, and, if not found, in lines 46-60, searching any reused fragments.

The fname function is defined in lines 63-110. Lines 73-78 deal with any flags supplied to this function. Lines 80-87 as described, are defunct. Lines 89-90 set our view of object space. The lines 91-109 deal with the three ways of calling this function, as provided in the script, including the specifying the retrieving of the member in the current fragment, the member which would be read if no member in the current fragment, and the one previous to this. This example, however, does not allow calling this function to retrieve

the filename of a specific version of a member name, in this manner. Another indicator of the simplicity of this example.

server2.c

```
 1 #include <stdio.h>
 2 #include <fcntl.h>
 3 #include <sys/types.h>
 4 #include <sys/socket.h>
 5 #include <netinet/in.h>
 6 #include <netdb.h>
 7
 8 #include "fname.h"
 9
10 extern int errno;
11
12 #define ERROR(n,str) { perror (str); exit (n); }
13 #define HEADER_OK "HTTP/1.0 200 OK\nContent-type:text/plain\n\n"
14 #define NO_METH "HTTP/1.0 501 OK\n\
15 Content-type:text/html\n\n<h1>Error 501: Invalid method.</h1>\n"
16 #define NO_FILE "HTTP/1.0 404 OK\n\
17 Content-type:text/html\n\n<h1>Error 404: No such file.</h>\n"
18 #define TX(sd,str) write(sd, str, strlen(str));
19 #define RX(sd,buf) read(sd, buf, sizeof(buf));
20
21 int
22 main(int argc, char *argv[]) {
23   int ld, addrlen, sd, rc, port;
24   struct sockaddr_in addr, name;
25   char buf [256], *end, str [80], *member, filename [80];
26
27   if (2 != argc || (argc == 2 && 0 >= (port=atoi(argv[1]))) ) {
28     fprintf (stderr, "Usage: %s <port(+ve)>\n", argv[0]);
29     exit (1);
30   }
31
32   if ( (ld = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0 )
```

```
33    ERROR (1, "socket")
34
35  name.sin_family = SOCK_STREAM;
36  name.sin_port = htons(port);
37  name.sin_addr.s_addr = INADDR_ANY;
38  if ( bind (ld, (struct sockaddr*)&name, sizeof (name)) < 0)
39    ERROR (2, "bind")
40
41  fprintf (stderr, "Listening on %d.\n", ntohs(name.sin_port));
42  if ( listen (ld, 5) < 0 )
43    ERROR (3, "listen");
44
45  while (1) {
46    FILE* f;
47
48    addrlen = sizeof (addr);
49    if ((sd = accept (ld, (struct sockaddr*)&addr, &addrlen)) < 0)
50      ERROR (4, "accept")
51
52    if ( fork () )
53      close (sd);
54    else {
55      close (ld);
56      RX (sd, buf);
57      printf ("%s: req: %20.20s\n", argv[0], buf);
58      if (!strncmp (buf, "GET ", 4)) {
59
60        if (end = (char*)strchr (buf+5, (int)' '))
61          *end = '\0';
62
63        if (member = (char*)strchr (buf+5, (int)'/')) {
64          *member = '\0';
65          member++;
66        }
67        fname ("", buf+5, member, filename);
68
69        if (f = fopen (filename, "r")) {
```

```
70          TX (sd, HEADER_OK);
71          while (!feof(f)) {
72            strcpy (str, "");
73            fgets (str, 80, f);
74            TX (sd, str);
75          }
76          printf ("%s: served file: %s\n", argv[0], buf);
77          fclose (f);
78        } else
79          TX (sd, NO_FILE);
80      } else
81        TX (sd, NO_METH);
82      exit (0);
83    }
84  }
85 }
```

The second example web server differs from the first in that it makes a member name from the requested URL (as calculated in lines 63-66), and uses this and the name of entity (at buf+5), in a call to fname, in line 67, to get the required file to be served. A file-system path name to object space translator, minus the mapping of sub-components, can be
found in the send program, chapter 4.

Appendix D.

The listing below is a very basic program to convert a comma-separated variable file into the equivalent HTML table. On finding a variable, the value is checked to see if it is a hyper-link and if it contains a ‘%’, by a call to the parse_href() function (lines 36-75). If so, the appropriate value for this cell is taken from the specified co-ordinated cell in the referenced table, by a call to the get() function, lines 12-34. Improvements to this program should include regenerating the linked-to table on searching for the text value to represent this table. When the linked to table is edited, the ‘hot-spot’ value may have changed. This means that either the linked-to table should be guaranteed to be stable, or the generation of the text value from the hot-spot, should be performed at runtime.

csv2html.c:

```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 #define COPY_FROM_TO(chp,ch,str,op_str) \
 5     cp = chp; \
 6     cpend = strchr (chp, (int)ch); \
 7     *cpend = '\0'; \
 8     cpend++; \
 9     strcat (op_str, cp); \
10      strcat (op_str, str);
11
12 char*
13 get (char *fname, int row, int col) {
14   FILE* fp;
15   int i;
16   static char *cp; char *cp2;
17   static char line [80];
18
19   if (fp = fopen (fname, "r")) {
20     fgets (line, 80, fp);
21     for (i=1; i<row; i++)
22       fgets (line, 80, fp);
```

```
23    i=1; cp = line;
24    do {
25      i++; cp++;
26    } while (i<=col && (cp = (char*)strchr (cp, ',')));
27
28    /* Remove rest of line */
29    cp2 = strchr (cp, ',');
30    *cp2 = '\0';
31  } else
32    cp = fname;
33  return cp;
34 }
35
36 char*
37 parse_href (char* s1) {
38   static char output [80] = {""}; char s [80] = {""};
39   char link_name [80] = {""}, *cp = NULL, *cpend = NULL;
40   int i, j, val1, val2; char coords [80] = {""};
41   char *v1ch, *v2ch;
42
43   strcpy(s, s1);
44   if ((cp = strchr (s,  (int)'<')) &&
45       (cp = strchr (cp, (int)')) &&
46       (cp = strchr (cp, (int)'>')) &&
47       (cp = strchr (cp, (int)'%')) &&
48       (cp = strchr (cp, (int)'<')) &&
49       (cp = strchr (cp, (int)'>'))   ) {
50
51    /* Copy the sections of s across to output */
52    COPY_FROM_TO(s,'=',"=",output) /* this macro sets cpend */
53    COPY_FROM_TO(cpend,'>',">",link_name)
54    strcat (output, link_name);
55    /* Knock off the ".html >" */
56    cp = strrchr (link_name, (int)'.');
57    *cp = '\0';
58
59    strcpy (coords, cpend);
```

```
60    v1ch = coords;
61    v2ch = strchr (coords, (int)'%');
62    *v2ch = '\0';
63    v2ch++;
64    val1 = atoi (v1ch);
65    val2 = atoi (v2ch);
66    strcat (output, get (link_name, val2, val1));
67
68    cpend = strchr (cpend, (int)'<');
69    COPY_FROM_TO(cpend,'<',"<",output)
70    COPY_FROM_TO(cpend,'>',">",output)
71
72    return output;
73  }
74  return s1;
75 }
76
77 main (int argc, char * argv []) {
78   char *el, IPfn[80], OPfn[80], ipchs[80];
79   FILE* OP, *IP;
70
81   if (2 != argc) {
82     fprintf (stderr, "Usage: %s <table>\n", argv[0]);
83     exit (1);
84   }
85
86   sprintf (OPfn, "%s.html", argv[1]);
87   OP = fopen (OPfn, "w");
88   fprintf (OP, "<table border>\n");
89
90   /* Do the first line of the table as a header */
91   sprintf (IPfn, "%s", argv[1]);
92   IP = fopen (IPfn, "r");
93   fgets (ipchs, 80, IP);
94   el = strtok (ipchs, ",");
95   while (el) {
96     fprintf (OP, "<th>%s</th>\n", el);
```

```
 97     el = strtok (NULL, ",");
 98   }
 99
100   /* Read following lines as table cell values */
101   while (!feof(IP)) {
102     strcpy (ipchs, "");
103     fgets (ipchs, 80, IP);
104     if (strcmp(ipchs, "") && strcmp (ipchs, "\n")) {
105       fprintf (OP, "<tr>\n");
106       el = strtok (ipchs, ",");
107       while (el) {
108     fprintf (OP, "<td align=right>%s</td>\n", parse_href(el));
109     el = strtok (NULL, ",");
110       }
111     }
112   }
113   fprintf (OP, "</table>\n");
114 }
```

The comma-separated variable files in chapter 5 can be converted to HTML using the following code.

table.html:

```
<table border>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4          </th>
<tr>
<td align=right>2</td>
<td align=right>5</td>
<td align=right>4</td>
<td align=right>7     </td>
<tr>
<td align=right>3</td>
```

```
<td align=right>4</td>
<td align=right>5</td>
<td align=right>8    </td>
<tr>
<td align=right>4</td>
<td align=right>5</td>
<td align=right>66</td>
<td align=right>9    </td>
</table>
```

outer_table.html:

```
<table border>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4          </th>
<tr>
<td align=right>1</td>
<td align=right>2</td>
<td align=right>3</td>
<td align=right>4    </td>
<tr>
<td align=right>1</td>
<td align=right><a href=./table.html>66</a></td>
<td align=right>3</td>
<td align=right>4    </td>
<tr>
<td align=right>1</td>
<td align=right>2</td>
<td align=right>3</td>
<td align=right>4    </td>
</table>
```