Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

A Comparison of Object Models

Andrew Watson

Abstract

Everyone in the computing world is constantly talking about "objects" at the moment. An increasing number programming environments and infrastructures are available to support "objects". There are many tangible advantages claimed for software written using object technology, such code reuse, better productivity, easier maintenance. However, "objects" are not all things to all men.

This report analyses the implicit object models behind several products gaining wide exposure at the moment, not to judge which is "best" (since there is no universal "best"), but to analyse how well each is suited to its own chosen application domain, and (in some cases) what features make it unsuitable for other particular domains.

APM.1530.00.01 **Draft** 8th September 1995
Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Copyright © 1995 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

A Comparison of Object Models

A Comparison of Object Models

Andrew Watson

APM.1530.00.01

8th September 1995

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Copyright © 1995 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Architecture Projects Management Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Contents

1	**1**	**Introduction**
1	1.1	Overview
3	**2**	**The Object Models**
3	2.1	The models being considered
3	2.2	What is an “Object Model”
3	2.3	Common features of object models
4	2.3.1	The Basics
4	2.3.2	Encapsulation
4	2.3.3	Run-time instantiation & first-class object references
4	2.3.4	Substitutability (*alias* inclusion polymorphism)
7	**3**	**General descriptions of the models**
7	3.1	CORBA
7	3.1.1	Background: The Object Management Architecture
8	3.1.2	Specifics
13	3.2	System Object Model
13	3.2.1	Intended use
14	3.2.2	Specifics
16	3.3	Microsoft COM
16	3.3.1	Background (and some politics)
17	3.3.2	Technical objectives
18	3.3.3	Specifics
21	3.4	CLOS & Dylan
21	3.4.1	Intended use
22	3.4.2	Specifics

1 Introduction

1.1 Overview

Everyone in the computing world is constantly talking about “objects” at the moment. An increasing number programming environments and infrastructures are available to support “objects”. There are many tangible advantages claimed for software written using object technology, such code reuse, better productivity, easier maintenance. However, “objects” are not all things to all men. Different classes of computing problems require different technology solutions. “Objects” are no more the universal panacea than anything else could be. This leads to a diversity of technologies, all claiming to be “object based”, but actually emphasising different aspects of the object paradigm, depending on the problem they are designed to solve. In extreme cases two application domains will demand mutually contradictory technical solutions. The result is a great deal of protracted technical argument over what particular features belong in a “true” object system, and a surprising amount of diversity amongst object-based software development environments. This diversity is inevitable and the arguments have no resolution, because there is no one set of features that address all problem domains.

This report attempts to cut the Gordian knot by analysing the implicit object models behind several products gaining wide exposure at the moment, not to judge which is “best” (since there is no universal “best”), but to analyse how well each is suited to its own chosen application domain, and (in some cases) what features make it unsuitable for other particular domains.

2 The Object Models

2.1 The models being considered

These are the systems whose implicit models are analysed in this report.

- CORBA
- CLOS/Dylan
- COM
- SOM/DSOM

One of these is a programming language; the others are platforms for building distributed applications with varying degrees of language- and operating-system independence.

2.2 What is an "Object Model"

Every computing system has an implicit computational model - the basic set of concepts and entities used by the programmer to construct and manipulate an analogue of the application domain's entities. For instance, the intended application domain for the programming language Fortran was scientific applications requiring processing large quantities of numerical data. The language's basic modelling concepts are arrays of floating point numbers (representing the data), and named procedures that manipulate these arrays (representing the algorithms used to generate the desired results from the data).

An object model is simply another name for the computational model of an object-based system. Application domain entities are represented using collections of data (representing their state) associated with procedures that represent their bahaviour.

2.3 Common features of object models

Although object technology is becoming very widespread, there is little general agreement as to what are the desirable (or even the essential) features of an object model. Certainly the various object models discussed in this document differ greatly in the facilities that they offer. However, before attempting to discuss their individual strong and weak points, it seems worth while at least trying to identify a common set of features that distinguish object models from other kinds of computational model.However, even this short list is possibly contentious, and not all the object systems analysed in this document share even these features!

2.3.1 The Basics

An object might be defined as a collection of data associated with one or more procedures (often called methods) that process those data. This definition covers every object model, but it also covers many systems that would not be widely regarded as object-based, such as the Ada package system. There are three further criteria which apply to almost all object-based systems, although not all meet even these few criteria.

2.3.2 Encapsulation

A key attributes of an object is that it prevents undisciplined and possibly inconsistent manipulation of it state by restricting direct manipulation of its state to a particular set of procedures (sometimes called methods, a term that originated with Smalltalk). All other access by an object's clients is achieved indirectly, by invoking these methods.

This encapsulation provides a separation between the internal implementation of an object (including the precise data representation and method code used to implement the abstraction that the object presents) and the external interface presented by the object. It is this separation (or encapsulation, as it's called) that provides many of the software engineering advantages claimed for object technology, such as the re-use and increased ease of maintenance of code.

Not all object system designers regard encapsulation as a primary requirement. Those working with databases, for instance, regard an object as a collection of data that represents some entity on the real world. The ability to instantiate new objects at run time and the ability to manipulate collctions of diverse objects are very useful to the object database designer, but encapsulation would merely prevent access to the data that the objects represent.

2.3.3 Run-time instantiation & first-class object references

Another key attribute of an object system is that new objects can be instantiated from some kind of template (often called a class or factory) at run time. This is a key differentiator from programming language module systems (such as those in Ada and Modula-3) which provide enapsulation but usually only allow one instance of a module in any one system, and certainly don't permit new instances of the module to be created at run-time under program control.

The use of first-class object refernces goes hand-in-hand with run-time instantiation, and could also be regarded as a key distinguishing feature of an object model.

2.3.4 Substitutability (*alias* inclusion polymorphism)

One facility that all programmers using object-oriented languages take for granted is the ability to use an object as though it has a only a subset of its actual facilities. This is used to good effect where-ever it is desirable to use one piece of code to manipulate a diverse set of objects with different interfaces; provided there's a set of operations that are common to all the objects, the client code can invoke operations from that set on any of the objects without having to be aware of the type or implementation of the object in question. For instance, say a diverse collection of different objects representing strings,

integers, matrices and dates all implement a print operation that displays their value. A single client can invoke this print operation on any of them, without having to be aware of how they are implemented, but guaranteeing to achieve the same effect in each case.

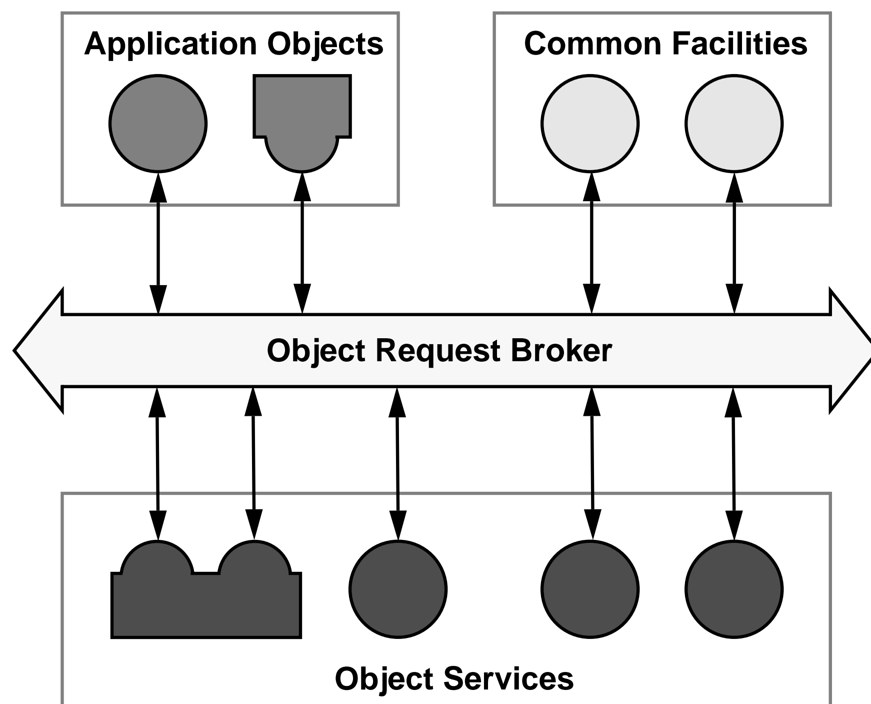
3 General descriptions of the models

3.1 CORBA

3.1.1 Background: The Object Management Architecture

The Object Management Group (OMG) aims to establish an architecture and set of specifications to enable distributed integrated applications. The general architectural framework is set out in the *Object Management Architecture Guide*. It includes the Reference Model shown in Figure 3.1. The solid shapes represent software with application programming interfaces, and the dotted boxes represent categories of objects with object interfaces.

Figure 3.1: The Object Management Architecture



The four major components of the OMA Reference Model are:

- The Object Request Broker (ORB) provides a common set of interaction semantics that allows separately-written applications to work together, potentially by invoking each other's operations across networks in a distribution-transparent fashion.

- Object Services, which support basic functions for using and implementing objects.
- Common Facilities, which provide general purpose services useful in many applications.
- Application Objects are specific to particular end-user applications.

3.1.1.1 *CORBA*

When specifications for the ORB component of the OMA were solicited by the OMG in 1990, a single composite response eventually emerged, written by a committee of designers from several companies, and named the "Common Object Request Broker Architecture", or CORBA [CORBA 93]. The name of this particular specification has since become synonymous in the public mind with the ORB component of the architecture.

CORBA is a framework for distributed object application construction and integration. For this reason it has an implicit object model which is entirely concerned with the interaction between objects (an "interaction model" in ANSA terms), and says nothing about how objects are implemented internally (in ANSA terms it doesn't have a "construction model"), since to prescribe how the objects are implemented would needlessly restrict the range of applications that could interoperate.

Since the ORB provides all the inter-object communication in the Object Management Architecture, it implicitly defines the OMG's object model. Although the OMG has a separate group working on Object Models in general (the Object Model Subcommittee), and they in turn maintain a document describing "The OMG Object Model", this activity should be viewed as more of a discussion forum than an effort to produce a definitive model. In practical terms the OMG's object model is defined by the CORBA specification.

CORBA uses a fairly straightforward, classical object model. Most implementors have interpreted the specification as dictating that the object is the unit of distribution, although this is one of the (many) areas that are not explicitly stated by the specification, and there have also been interpretations that allow (for instance) the code implementing different operations on the same object to be located on different hosts. It is to be hoped that, as the CORBA specification is revised and tightened up over the coming years, this problem of multiple interpretations will decline.

3.1.2 Specifics

3.1.2.1 *Encapsulation*

CORBA doesn't so much enforce encapsulation as lack any mechanisms that violate it, an approach that it has in common with many object systems. While this achieves encapsulation if the object system is all-encompassing, most CORBA-compliant platforms run under a host operating system, which may provide facilities that an application writer can use to violate the encapsulation of CORBA objects.

In ANSA/ODP terminology, CORBA provides a single interface per object [1]. It's noticable that some OMG Object Services specifications (notably the Event Service) use a notion of a "conspiracy of objects", each providing a separate but related service its clients. This is isomorphic to the equivalent ANSA/ODP structure of one object with multiple interfaces, providing related services. However, to implement the conspiracy one would probably have to use some technique that allows objects to communicate other than via their (single) interface (thus breaking encapsulation). With luck this realisation will encourage the OMG membership to re-examine the ANSA/ODP model.

3.1.2.2 Run-time instantiation

Since CORBA does not have a construction model, it has no facilities for creating objects, but instead allows the client applications that are using it to register new objects. This registration creates an *Object Reference*, which can then be passed to other objects to allow them to make invocations on this object.

In theory, object lifecycle issues (creation, deletion, etc) are handled via a separate object service (the Lifecycle service). However, these functions are so intimately involved with resource allocation issues that there's little that's useful that can be said in the description of such a service, and its implementation will be highly operating system- and environment-dependent.

3.1.2.3 Run-time deletion

Once again, CORBA has no specific support for deleting objects and freeing the resources they consume. The lifecycle service recommends that objects provide a "delete" operation, so that they can be requested to delete themselves and free their resources. There is no garbage collection or reference counting implemented to try and determine if there are any clients holding references to the object.

3.1.2.4 Inclusion polymorphism

The interface to a CORBA object can be described using an Interface Definition Language (IDL). The main purpose of IDL is to allow client stubs and server skeletons for object invocation be to be generated automatically. A so-called "inheritance" mechanism in IDL allows one interface to be specified to include all the operations from one or more other interfaces in the same IDL file, and the inheritance rules are designed to ensure that the derived interface is automatically substitutable for (i.e. a subtype of) all the interfaces from which it inherits. Thus interface inheritance is a sufficient condition for inclusion polymorphism on CORBA objects. However, the specification does not say whether it is **necessary** to use interface inheritance in order to construct a new interface which can be used as if it were another interface with fewer operations. As usual, this omission has led to different groups of implementors taking both implementation choices, although some agreement on the "sufficient" rather than "necessary" interpretation now seems to be emerging.

1. Regrettably there is a terminology clash between ANSA/ODP and the OMA here; what the OMA terms an "interface", ANSA/ODP calls a "type" - since there is no OMA notion of multiple ANSA/ODP interfaces per object, there is no term to describe the concept

3.1.2.5 Object Handle Semantics

CORBA's Object References are opaque tokens used to refer to objects when making invocations on them. They are created when a new object instance is registered with the CORBA implementation as being ready to receive invocations, and (as one would expect) can be duplicated, stored, and passed as parameters or results to invocations on CORBA objects.

In the first version of the CORBA specification a deliberate decision was taken to make object references almost completely opaque. No particular concrete representation or indeed size of representation is dictated. Apart from being duplicated, passed as parameters and (of course) used to invoke operations on the object to which they referred, there was a very limited set of operations defined on the handle itself:

- A boolean operation to determine if this handle is null (i.e. does not refer to an object, but has been set to nil).
- A pair of operations to turn a handle into a string form suitable for storage, and back again. Although not explicitly stated in the specification, the intention was that this transformation to and from a string form would only ever take place at a single "location". Although the scope of these locations isn't defined either, it can probably be assumed to be a single object or (perhaps) a single instance of an ORB implementation. Passing the string representations as an invocation parameter to another object and performing the string-to-reference operation on it should not be expected to yield a working object handle.
- An operation to get hold of an interface description for the object to which the reference refers (`get_interface`).

Revisions 1.1 and 1.2 of the CORBA specification did not contain operations to compare object references or attempt to determine whether a pair of references refer to "the same" object. This was a conscious decision, documented in [POWELL 93].

3.1.2.6 Method resolution

In the syntax for CORBA operation invocation in all language mappings, the operation on the object to be called is identified by name. However, in version 1 of the CORBA specification it wasn't stated how method resolution is mechanised at run-time. There are two possible approaches (resolution by name, and resolution by table offset), and there were some implementations that took each.

Method resolution by name is the more general technique, although arguably less efficient to implement. It is a direct realisation of the programmer's intuitive notion of how method resolution works; the invocation data passed at run-time from the client to the service-providing object includes the name of the operation to be invoked, as written in the source program. The object infrastructure at the service-providing end compares this name against those of the methods that the referenced object implements, and dispatches the invocation to the correct one. Although the process may be speeded up by hashing techniques, it will always involve more work at run time than (say) a 'C' function call.

Under some circumstances the resolution process can be speeded up by doing the table lookup once, at compile time. If it is known at compile time exactly what operations are implemented by the service being invoked, and how they

are laid out in the table of operations stored at the server end then the invocation mechanism can simply transmit the table offset rather than the whole name, and the method lookup at the server becomes a single indirection through the method table.

A number of early implementations of the ORB 1 specification (which left implementor free to choose their own on-the-wire invocation formats) used offset resolution, while others used name resolution. The new CORBA 2 specification does lay down the wire format for invocations, and specifies that the operation to be invoked is specified by a string, but unfortunately doesn't say what that string contains. Most implementers seem to be taking it to be a simple, unqualified operation name, but there are other opinions. This also has a direct bearing on the inclusion polymorphism issue discussed above; an offset-based scheme only provides inclusion polymorphism if it is known that all the interfaces for which the current interface may be substituted have operations of the same name at the same offset, otherwise unexpected things may happen.

3.1.2.7 *Interaction model*

As a result of being designed by a committee, CORBA has three sets of interaction semantics: synchronous, deferred synchronous, and oneway. In each case the client identifies the server object by an Object Reference, and the operation on that object by quoting its name.

CORBA's main interaction model is synchronous, akin to a Remote Procedure Call. When a thread running in a client program makes an invocation on another object, that thread blocks until the operation on the server object has been invoked and the results returned, whereupon the thread at the client end resumes with access to the results. All of the existing OMG-standardised mappings of programming languages onto CORBA represent this as an ordinary (synchronous) procedure call or method invocation in the language, with one result passed back as the result of the invocation. A CORBA operation invocation can return multiple results, and in the original 'C' language mapping this was catered for in the traditional C language way, by passing in as a parameter a reference to a data structure to be filled with result data. Unfortunately this mechanism was enshrined in the Interface Definition Language as OUT and IN/OUT parameters (the latter used to pass data in both directions), thus embedding in CORBA a distinction between data and references to data. This has posed something of a puzzle to the subsequent designers of mappings for languages where this distinction does not exist, such as Smalltalk.

Failure is a possible outcome from an invocation in any distributed system infrastructure, and this is signalled in CORBA's synchronous invocation mechanism using an exception. Depending on whether a particular programming language has a suitable exception mechanism, its mapping will represent this either as a native language exception or using a data structure passed back from the call. The semantics of the synchronous call depend on whether it is successful or not. If an exception is signalled then the semantics are "at most once"; the (potentially-remote) operation may have been executed, or not, or partially, and the client is unable to determine which without further information. If the operation returns successfully then the semantics are "exactly once"; the client program knows for certain that the operation was executed precisely once.

Because CORBA's synchronous invocation mechanism blocks the calling thread until the operation has completed, the ability to multi-thread client code is almost essential to allow a client to issue multiple invocations on different servers, or to do local computation in one thread while another is blocked on an invocation. Threads are even more vital within server code, since otherwise a service that directly or indirectly invokes one of its own operations would deadlock. CORBA does not discuss or specify multithreading at all; for the client side this is justifiable, since threading is an orthogonal issue to the invocation mechanism, which is all CORBA seeks to specify. Unfortunately threading interacts rather more intimately with the way the implementation of the server is registered with and invoked by the ORB implementation, and future revisions of the CORBA specification will probably have to address this.

CORBA also includes two other, distinct invocation models, which can be selected by the programmer at will. The first, and most widely misunderstood, is the oneway invocation. This is intended to be precisely analogous to the ANSA/ODP announcement, but the description in the CORBA specification is somewhat vague. In all current programming language mappings oneway invocations look syntactically just like ordinary synchronous invocations. The only apparent difference is that the calling thread is not blocked, and the invocation does not return any results. However, within the CORBA implementation the differences are deeper.

A oneway call is intended to provide an interface to an unreliable datagram style of interaction, such as UDP (Unreliable Datagram Protocol) in IP networks. Once the datagram has been composed and dispatched over the network the sender has no further control over its delivery, nor direct knowledge of whether it was delivered. If the receiver is being bombarded with oneway invocations from many sources it will likely run out of resources to process them, and will have to throw some away. It is for this reason that the semantics of oneway are termed "best effort" (i.e. "best effort, but no promises"). In contrast, the implementation of synchronous invocation does not discard the invocation parameters until it knows that the invocation got through (usually because the results have been returned), so that the invocation data can be retransmitted if necessary. This is usually the desired model; however, it does mean that the invocation data are indiscriminately buffered for possible retransmission, whereas they may be perfectly adequately stored somewhere else (if the application is reading the data verbatim from a file, say), or be so ephemeral that re-transmission would not make sense. In these cases the application programmer will want to take control of the buffering and retransmission strategy himself by using oneway invocations; however, in the vast majority of cases it's more convenient to delegate this job to the infrastructure by using synchronous invocations.

The third set of invocation semantics offered by CORBA is the so-called deferred synchronous invocation, which is only available via the Dynamic Invocation Interface (DII). Whereas a synchronous invocation blocks the calling thread until the remote operation has run, the deferred synchronous call returns a token to the caller that represents the invocation. The calling thread can then continue executing, and later on passes the token as a parameter to a DII call to redeem the result of the invocation. This call will block if the remote operation hasn't yet completed, or return immediately if it has, but in either event will yield the result when is does return.

Although deferred synchronous invocation seems neatly to avoid the need for multithreading to avoid the possibility of deadlock and provide parallelism at the client, this is to some extent an illusion. One of the aims of the CORBA specification is distribution transparency, so that the client and service provider could just as easily be linked together in the same process as running on separate machines on a network. The local optimisation of the synchronous invocation is a simple subroutine call, as available in the instruction sets of all processors, and therefore fast, simple and lightweight. On the other hand, perversely, the local implementation of the deferred synchronous mechanism is complex and may have to employ a multithreading mechanism to provide parallelism between caller and callee if deadlock is to be avoided in all cases.

There is also the vexed question of the scope of the deferred synchronous invocation tokens. Where can they be redeemed? Can they, for instance, be passed as a parameter to another object and redeemed there? Designing a mechanism to allow this in the general case would be highly complex. Can they be duplicated or stored on long-term storage? If so it's not clear how long the infrastructure has to retain the results of an invocation against possible redemption. Unfortunately, none of these questions are addressed in the CORBA specification, and it's hard to see how they could be in any rigorous way. This detracts a great deal from the ability to write portable applications that use deferred synchronous invocations.

3.1.2.8 Metadata

All metadata in CORBA-based systems is related to the Interface Definition Language (IDL), used to express the interfaces presented by objects. These interfaces are composed of named operations, each with a fixed set of typed parameters. At run-time equivalent information about an interface is available from the Interface Repository. The interface description for any object can be reached via the get_interface pseudo-operation on the object handle. The information in the IR is purely descriptive; changing the contents of the IR does not of itself change the interfaces of any objects.

The Interface Repository was (deliberately) incompletely specified in the CORBA 1 specification; only the interfaces for reading information out of it were described. A full interface, including update operations, has been provided as part of the CORBA 2 specification, but unfortunately must only be considered a stopgap, since it raises as many questions as it answers (for instance - how does one prevent simultaneous, incompatible updates?). Further work in this area is proceeding within OMG under the more general title of change management.

3.2 System Object Model

3.2.1 Intended use

Several companies are designing general object models that will provide the foundations for distributed, object-based system software in the future; the System Object Model (SOM) is IBM's offering. It's also the object model that underlies the OpenDoc compound document architecture.

SOM's initial objective was to provide a general-purpose object model to be used for installing and re-using modular system and application software. In order to achieve this it has attempted to overcome the main shortcoming of

most object systems in this regard, namely that one must have source code available in order to re-use via implementation inheritance. SOM allows objects to inherit the implementations of method bodies from each other, even when the objects are implemented in different programming languages.

Providing a distributed infrastructure does not seem to have been one of SOM's initial objectives (in contrast to CORBA, which was always aimed at distribution). However, the most recent version (SOM 2.0) is moving towards CORBA-compliance; it uses an extended CORBA IDL, and there is an optional DSOM (Distributed SOM) code library which provides distribution for SOM objects.

However, objectively speaking, SOM is still not particularly CORBA-compliant. For instance, the IDL allows many features to be specified in the so-called “private” section of the IDL file, an extension to the CORBA specification. Although the SOM IDL compiler can be persuaded to generate a version of the IDL file without this private section, this is merely to enable IDL to be exported to other CORBA implementations; to attempt to use SOM as a “pure” ORB, without private sections in the IDL files probably wouldn't achieve much.

3.2.2 Specifics

3.2.2.1 Encapsulation

SOM supports a “classical” object model, with one interface per encapsulated object.

It appears that collections of communicating SOM objects must being in the same address space (although this restriction is removed in DSOM). In SOM (but not DSOM) data pointers can be passed as invocation parameters; this would allow references to the implementation data inside one object to be passed to another object; this is a violation of encapsulation, since the concrete format of the data within an object effectively becomes part of its interface if a pointer to that data is passed out of the object.

3.2.2.2 Run-time instantiation

COM objects are instantiated at run-time by way of a request to the class object.

3.2.2.3 Method resolution

SOM offers a choice of three method resolution techniques - name, offset or function.

The default, at least for non-distributed SOM, seems to be offset resolution, in which the client stub code for invoking a particular named operation includes the offset of that operation in the IDL file from which it was built, and the invocation mechanism actually transmits the offset number rather than the name to server stub. Unlike other resolution-by-offset implementations, SOM allows control over the order of the operations in the dispatch table by specifying them by name, in the desired order, as parameters to the *releaseorder* directive in the private section of the IDL file. Placeholder operation names that are not actually specified in the IDL file proper can also be used as parameters to releaseorder, for instance to ensure that remaining operation names maintain the same offset if some operations are removed from an IDL specification.

The decision to make offset resolution the default probably stems from a desire to make invocation as fast as possible in the non-distributed case, where other overheads are small. The adverse effect on inclusion polymorphism is somewhat ameliorated by the provision of the releaseorder directive. SOM also provides resolution-by-name, although the documentation doesn't discuss this much. Finally, there is a third mechanism, function dispatch, which apparently enables the programmer to customise method selection on a per-object basis by providing a dispatch function.

3.2.2.4 Interaction model

SOM provides exactly the same set of interaction models as CORBA, although it seems that the Dynamic Invocation Interface and its associated deferred-synchronous invocation model are provided only for CORBA compatibility, since IBM's own SOM documentation mentions their existence only in passing, in an appendix.

3.2.2.5 Metadata

SOM has a run-time class/meta-class system for representing metadata about objects. This is very unusual for an object system intended to be used for large-scale system infrastructure, although there are several single-user interactive object programming environments and languages that have this property (e.g. Smalltalk, Common Lisp Object System). Other infrastructures certainly provide readable metadata on objects (such as the interface descriptions in the CORBA interface repository), allowing potential clients to discover information about the object at run-time; however in each case their information is purely descriptive, so that changing the run-time description no more alters the object described than altering the engine capacity figure in the owner's handbook increases the power of my car's engine. In SOM, on the other hand, it seems that information about object classes is stored in a run-time object that represents the class, and altering that information automatically alters the class, and therefore every instance.

However, although run-time classes are said to be a “distinguishing attribute of SOM technology” and “very convenient and powerful”[1], there is no actual mention in the SOM descriptions of how it might be used. As in Smalltalk and CLOS, the class of a class is known as a metaclass. Changing the information stored in metaclasses potentially has the power to alter the behaviour of the whole object system in far-reaching ways; the SOM notes this, but restricts itself to saying that “The potential power of metaclasses as a way of dynamically adding, deleting or altering class objects and thereby affecting the object instances that will be instantiated has yet to be fully appreciated”.

In my view the whole concept of run-time classes and metaclasses is a dangerous red-herring for an object infrastructure designed for building system software. While dynamically altering class information is a powerful tool in an interactive programming environment, it would be highly dangerous to allow any object to meddle with the class of another object in a multi-user, multi-process environment. Practical experience with metaclasses shows that they offer few practical advantages in terms of altering the behaviour of an object system in useful ways. Against this must be set the potential performance penalty of making key system control structures alterable at run-time. Finally, it's also worth noting that at least one ambitious attempt to

1. SOMobjects: A practical introduction to SOM and DSOM, p16

define a *standard* metaclass protocol (for CLOS) has already foundered on the sheer complexity of the task. From the hints in the SOM descriptions I have read, it seems that IBM's engineers are also finding writable metaclasses a troublesome concept.

3.3 Microsoft COM

3.3.1 Background (and some politics)

The Component Object Model (COM) was originally the name given to the implicit object model underlying Microsoft's revised compound document architecture offering, OLE 2 (Object Linking and Embedding). At some point COM was repositioned as a general-purpose object model for building system software, and the name changed to Common Object Model. In the latter guise it is being promoted as the basis for Microsoft's future object-based operating systems (i.e. Cairo and its successors). Since Microsoft rarely expand the COM acronym, there's much room for confusion between the two names.

There are many general similarities between COM and IBM's rival SOM product. Both are advertised as general-purpose object frameworks for system software, but currently the main application for each is supporting a compound document architecture (OpenDoc for SOM, OLE for COM). Each began life lacking support for distribution, but was later extended to accommodate it. In each case this has been done by a partial reimplementation, and some changes to the specification; however, IBM are much more forthright about the changes engendered by support for distribution, and have separate names for the distributed and non-distributed versions. Microsoft, by contrast, are extremely coy about the fact that COM as currently shipped (for instance, with OLE 2 implementations and in developers' kits) has no distribution support; this will apparently be shipped in late 1995 or early 1996. The distributed version of COM will not have a separate name, and Microsoft are apparently hoping that existing applications will seamlessly become distributed by virtue of being run on the new infrastructure. Meanwhile, the similar objectives of the two systems and the rivalry between the companies that created them has led to a fierce war of words, with each side issuing "Technical papers" that compare their own product favourably with the other.

COM is essentially a binary specification, clearly reflecting Microsoft's interest in, and dominance of, a single CPU architecture. Where CORBA is defined in terms of the IDL for specifying interfaces and a standard mapping onto each programming language, COM is defined in terms of things like the layout of operation dispatch tables in memory. Because of its pan-industry origins, CORBA was initially intended to provide source-level portability of applications across CORBA implementations on a number of platforms, so that the same source program could be moved between CORBA implementations on a variety of platforms, recompiled, and would run. COM's objective, on the other hand, is binary compatibility, so that diverse libraries of objects can interact with each other, but only within a Wintel environment. With the arrival of CORBA 2, the CORBA specification has now been extended to cover wire protocols that ORBs use to communicate with each other, so that guaranteed interoperability between ORB implementations from different vendors on different platforms will now be possible. However, paradoxically, because of OMG's success in fostering consensus across over 500 members

organisation, each deploying and using a variety of processor architectures and operating systems, it is highly unlikely that there will ever be a CORBA binary portability standard.

Despite these differences, it looks as though the commercial battle to establish one, pre-eminent object model for system software will be between Microsoft's COM and OMG's CORBA; the former will have an instant large presence on the desktop by virtue of being supplied as part of OLE 2, while the latter is supported by almost everyone else, with many CORBA products shipping for all platforms. Although a member of OMG, Microsoft had, at least until August 1994, completely ignored the OMG process, and were known to be rubbishing OMG specifications in private conversations with potential customers. At that point Microsoft made overtures to OMG, suggesting cooperation on an effort to standardise bridging between the two worlds. At the time of writing (September 1995) this is still in progress. There has been much speculation about why Microsoft's position should have apparently changed in late 1994. This author's analysis is as follows:

- By late 1994 there was widespread awareness of OMG and its specifications in the industry. Microsoft were being asked by many small customers and a few large ones (I've heard General Motors mentioned in this context) when they were going to join the OMG fold, or at least build bridges.
- At that point shipping of distributed COM was still many months, perhaps years, away, so Microsoft did not have a viable distributed infrastructure to offer customers who were persistent enough to ask for one.
- The US government started dropping broad hints that compliance with OMG specifications would be an essential requirements in some future government contracts.

Despite apparently cooperating with OMG on the RFP for bridging technology, Microsoft's heart doesn't seem to be in it, so it looks as though this is just a respite in the developing battle for supremacy in the distributed object market.

3.3.2 Technical objectives

It's clear that the consideration uppermost in the minds of the designers of COM was version compatibility; that's to say, the problem of ensuring that a new, revised service provider supporting added facilities can nevertheless work with a client written against the original service specification, and vice-versa.

Where an object model supports inclusion polymorphism, version compatibility can to some extent be provided by contriving that the new service type is a a subtype of the old. As long as the semantics of the operations common to both the old and new services have not changed, old clients can use the new service just as though it were the old one, and hence need not be changed. New clients wishing to use the facilities of the new service type, or the old one if the new is not available, must contain code to use both the old and new services, choosing which to employ with the help of run-time type information about the server.

The problem with this approach is that it may not always be practical to make the new service type a strict subtype of the old. In order to address this issue

Microsoft's designers, although still wedded to the classical object model in which every object provides just one service, allow the object to provide many separate versions of that service using what they term 'multiple interfaces'. Unfortunately, their use of the word 'interface' is different to both the ANSA/ODP and OMG/CORBA usage. In COM-speak, an interface is essentially a type, and a an object may support a single instance of each of many types. Every interface (type) has a globally-unique name provided by a DCE UUID, and every object must provide an instance of a standard interface called **IUnknown** (by convention, all COM interface names begin with the letter I). All interfaces, including **IUnknown**, provide an operation called **QueryInterface**, which when invoked with an interface UUID returns a reference to the instance supported by this particular object, if there is one.

This interesting and unique design has many ramifications. It's certainly less flexible than the ANSA/ODP model of objects with multiple instances of multiple interfaces, since the design of **QueryInterface** makes the implicit assumption that no object will support more than one instance of one type. On the other hand, the ANSA/ODP design allows one object to provide multiple version of a service through multiple interfaces, with each being separately registered with a trader for service location, but in addition allows an object to provide multiple instances of the *same* service, each maintaining individual per-client state. However, the COM approach to versioning also has advantages, not the least of which is efficiency; once a client has acquired a reference to an interface on an object it can determine whether that object supports a particular version of a service very quickly, with the use of 128-bit UUIDs to label interfaces permitting the implementation of **QueryInterface** in a few dozen machine instructions. Apparently one of the objectives of COM was to permit a client to determine which versions of a service are offered by a particular object in less than one keystroke time on a PC.

3.3.3 Specifics

3.3.3.1 Encapsulation

Like CORBA and SOM, COM is essentially a set of mechanisms for allowing objects to interact, rather than a complete object environment. These interaction mechanisms are specified in terms of the concrete layout of the data structures that a COM client uses to interact with a service-providing object. At present COM can only be used for communicating between objects inside one process, although the distributed version (based on a cut-down version of DCE RPC) will be available at some point. COM is implemented as a collection of library routines, provided under Windows as a Dynamically Linked Library (DLL), and hence applications using COM also have complete access to all the facilities provided by Windows and the implementation language in which they are written (C++ is the default implementation language assumed by Microsoft, but there's nothing to stop application writers using assembler). As a result, although COM doesn't provide any features that break encapsulation, neither does it prevent the application using host operating system features to do so.

3.3.3.2 Run-time instantiation

COM supports run-time instantiation of objects via factory objects, but because of the implicit assumption that interfaces are used to provide multiple versions of the single conceptual service offered by an object, no provision has been made for interfaces on an object to be dynamically created. This is not to

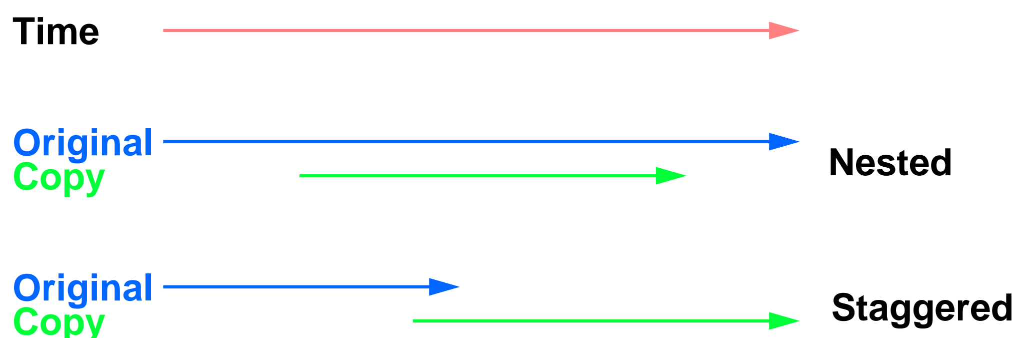
say that it couldn't be done (a sufficiently determined programmer with an assembler-level understanding of COM's structures could certainly manage), but rather that Microsoft have not foreseen the possible need, and have provided no support for this.

3.3.3.3 *Run-time deletion*

Microsoft's designers have clearly given some thought to run-time resource management, and the COM specification goes into the subject in much more depth than either the SOM or CORBA documentation. COM implements a simple but probably-effective reference counting scheme. Every interface must implement two operations: `AddRef` and `Release`. When a client holding a reference to an interface duplicates it (for instance, by passing it as a parameter in a call on another interface) then it should first of all call `AddRef` on the interface to inform the object that the number of extant references has increased, and conversely, when discarding a reference `Release` should be called to decrement the count. In this way, when an object learns that no other object holds references to any of its interface, it can presumably make arrangements to delete itself and free its resources.

This simple reference counting scheme has many disadvantages. To start with, calling `AddRef` and `Release` will add at least a 100% overhead to the number of invocations in an application, maybe more. More sophisticated reference counting schemes, such as weighted reference counts, don't require a call back to the referenced interface every time one of its references is copied. The COM documentation mentions possible optimisations to the calling pattern to alleviate the problem, such as not calling the reference count operations at all for reference copies whose lifetimes are nested inside the original's, or not calling `Release` and then `AddRef` if it's known that the copy will outlive the original (see Figure 3.2). However, these highlight the second major problem with this strategy, namely the possibility for programming errors to create insidious bugs when reference management is mishandled. However, in spite of all these faults, it has to be said that the COM designers have at least tried to tackle this resource management problem, which is not addressed at all in the SOM or CORBA specifications.

Figure 3.2: COM reference count optimisations



3.3.3.4 Inclusion polymorphism

COM's designers seem to have thought inclusion polymorphism unnecessary, and partly because of other design decisions to promote run-time efficiency, COM provides no support for it. However, a sufficiently determined application programmer could provide limited inclusion polymorphism under some circumstances.

The basis of inclusion polymorphism is that the client may treat an interface as though it has a type that is a supertype of the one that it actually implements. Since the types of COM interfaces are represented at run time exclusively by UUIDs, the only relationship that can easily be determined is interface identity, based on comparing the 128-bit UUIDs of a pair of interfaces. This would seem to rule out inclusion polymorphism.

However, since it's only the implementation of the `QueryInterface` operation that holds information on the mapping from UUIDs to interface instances for a particular object, it would be possible to implement a form of inclusion polymorphism in which the `QueryInterface` implementation maintains several UUIDs for each interface, including those for the supertypes of the type of that interface. When a client interrogates `QueryInterface` with one of these supertype UUIDs, a reference to the interface would be returned, and the client could use it as if it had the supertype without error. Unfortunately, there is a further obstacle, in the shape of COM's offset dispatch mechanism, under which operations are specified at invocation time by their offset numbers in the `vtbl` structure, rather than by name. To get around this the implementor would have to take care to assign the same offset to operations of the same name and signature amongst this group of interfaces. The most convenient way to ensure this would be by some form of inclusion or "inheritance" mechanism in the IDL used to specify the interfaces. However, details of COM's IDL are sketchy (it seems to have been something of an afterthought), and it's not clear whether this is supported.

As can be seen, COM's basic structures can be adapted to support inclusion polymorphism only with some difficulty. It will be interesting to see whether those using COM in anger agree with the designers' sentiment that it's unnecessary.

3.3.3.5 Method resolution

COM's method dispatch is exclusively by table offset. The basic interface data structure contains a pointer to a vtbl, a dispatch table containing pointers to the actual method code, and per-object state data (see Figure 3.3). The vtbl structure is essentially that produced for method dispatch by a C++ compiler.

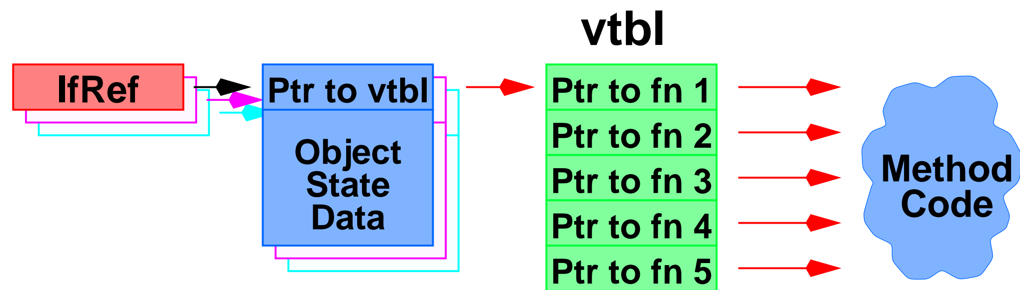
3.3.3.6 Interaction model

The COM documentation is almost completely silent on the subject of the interaction model. Judging from the fact that COM seems to have been based on a C++ philosophy and C++ as an implementation language, the interaction model is probably synchronous invocation.

3.3.3.7 Metadata

COM maintains almost no run-time metadata. All information about interface types is discarded at compile time, being represented at run time exclusively by 128-bit UUIDs (see section 3.3.3.4).

Figure 3.3: Structure of a COM interface



3.4 CLOS & Dylan

3.4.1 Intended use

3.4.1.1 Background: Lisp and object systems

Dylan (DYnamic LAnguage) is a programming language with a C-like syntax but Lisp-like semantics developed at the Apple Computer eastern development lab in Cambridge MA, formerly the independent Lisp vendor Coral. Dylan's underlying model is a slightly simplified version of that found in CLOS (the Common Lisp Object System), an optional extension to the Common Lisp language [KEENE 89] [STEELE 90]. CLOS is turn owes much to previous Lisp-based extension languages such as Flavors and LOOPS. Like Smalltalk, CLOS and Dylan are programming languages best suited to an interactive, single-programmer development style, and when coupled with a comprehensive programming environment, can be used to good effect by experienced programmers to turn out complex applications in very short order. Neither language was conceived for use in distributed environments.

Since the Dylan object model differs only from that of CLOS in having had some of the more baroque and intimidating features removed, this section refers to them interchangeably. Where Dylan differs, this is explicitly noted.

Despite being described by their designers as object-based, the model underlying Dylan, CLOS and their ancestors is completely unlike that of any other object-based system in this report. I would prefer to term them "generic function based" rather than "object based". Oddly enough, the difference stems originally from the syntax of the Lisp languages.

Because of its shared internal representations of programs and data as lists, Lisp has always been a very useful framework for experimenting with programming language semantics. A particularly good example of this is the excellent programming tutorial "Structure and Interpretation of Computer programs", which forms the basis for the 6001 introductory programming course at MIT. Using the Scheme language (another Lisp variant chiefly notable for using lexically-scoped variables and having full support for closures and continuations) the authors demonstrate many programming techniques (including object-oriented programming and logic programming)

by building implementations of demonstration languages to do these things under Scheme.

At the same time that Goldberg et al were developing Smalltalk at Xerox PARC, several groups were taking full advantage of the extensibility of Lisp to experiment with object-oriented programming. One group, working alongside the Smalltalk developers at PARC, and indeed using the same D-machine hardware (such as the Dorado, Dandelion and Daybreak) developed an extension to Interlisp-D called LOOPS, whose underlying object model was essentially that of Smalltalk [BOBROW 83]. The LOOPS-style method invocations looked like this:

```
(← x print OutputStream)
```

"send message to object *x* telling it to invoke operation *print* on argument *OutputStream*"

Meanwhile, another group at MIT developed Flavors under Lisp-machine Lisp, and it was from this work that CLOS and its Generic Function model developed. It seems likely that this group wanted their object system to blend more seamlessly into the underlying Lisp language; their invocation syntax looked like this:

```
(print x OutputStream)
```

"do *print* operation on object *x* with argument *OutputStream*"

Getting rid of the ← operator in every operation invocation does indeed make this look more Lisp-like. However, for symmetric binary operations like addition, this syntax highlights one of the odd features of OOP; why should one object be chosen as the recipient of a message, and the other simply be an argument? In other words, why should I choose to write:

```
(+1 2)
```

"do *plus* operation on object *1* with argument *2*"

... instead of:

```
(+ 2 1)
```

"do *plus* operation on object *2* with argument *1*"

Faced with this conundrum, the Flavors designers chose to "generalise" their invocation scheme so that the selection of which method code to run was **not** done by the receiving object, but could be determined by the classes of any or all of the invocation parameters. This means that (conceptually at least) the operation dispatch is no longer done by the object receiving the invocation, but by some external (nameless) mechanism, making decisions based on the classes of all the arguments to the operation.

Thus was the generic function model born. When the Lisp community began work on Common Lisp, slated to be the combined specification to supersede all previous dialects and unite the diverging Lisp community, it was the Flavors model, not the LOOPS model, that was chosen as the basis of the Common Lisp Object System, or CLOS.

3.4.2 Specifics

3.4.2.1 Encapsulation

CLOS and Dylan do not provide encapsulation on a per-object basis.

Classical object models bundle together all the object state and operation bodies inside an encapsulation boundary, so that only the names and

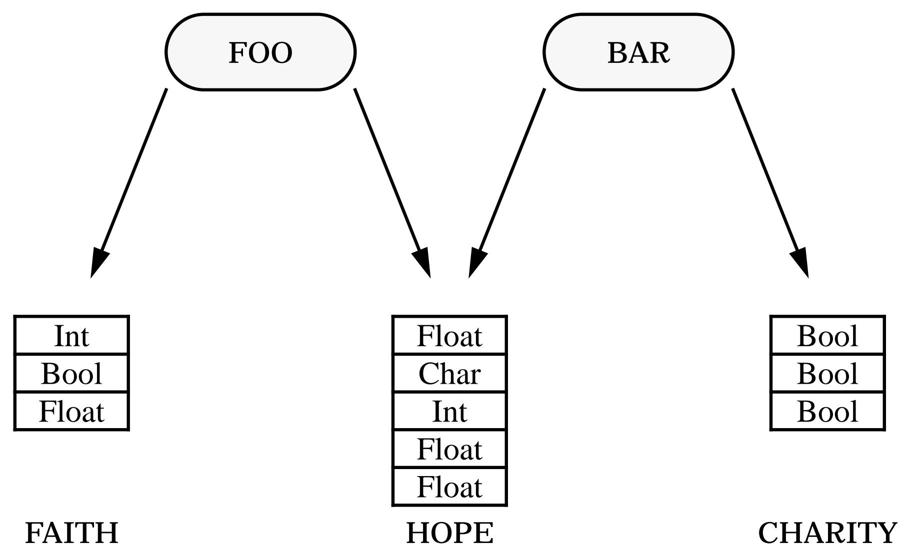
signatures of the operations are “visible” from outside the object. This means that the implementation details of an object can be changed without having to change any of the object’s clients, provided the overall functions performed by the methods and their signatures remain the same. Hence all the implementation details of the object fall inside the object encapsulation boundary and are hidden from the object’s clients.

In the generic function model of CLOS and Dylan, the “objects” are in fact just record structures, each containing a number of slots to hold the instance data, and labelled with a type. Named “methods” operate on one of more object at a time, and have access to the internal slot structure of the objects in order to be able to do so. Since every method can operate simultaneously on objects of different types (and therefore must have knowledge of all their structures), while every object can potentially be operated on by more than one method, there is in general nowhere where an encapsulation boundary can be drawn, and within which changes can be made to method code or object structure without affecting some other object or method (see Figure 3.4).

Figure 3.4: Why generic functions prevent encapsulation

Generic function Foo contains a method that specialises on arguments of type Faith and Hope

Generic function Bar contains a method that specialises on arguments of type Hope and Charity.



3.4.2.2 *Run-time instantiation*

Both CLOS and Dylan allow run-time instantiation of objects. A CLOS object is just a vector of slots, each of which is capable of holding a single CLOS value.

CLOS derives its basic data structures from Common Lisp, the language in which it is embedded. Like all Lisps, Common Lisp allows any value to be stored in any binding, and be passed as any argument to a function. Many

polymorphic functions simply move their arguments around without having to interpret them, and hence work correctly for all data types; an example is the `cons` function, which builds a two-element “cons-cell” whose contents are the two arguments it is passed. Because of this polymorphism, all Lisp data types must be represented as machine values of the same size. This is achieved by storing smaller values (such as integers and characters) in one machine word, and larger values (such as arrays or CLOS objects) in memory, and manipulating them via memory pointers. Hence, a CLOS object is implemented as a vector of equal-sized slots, capable of holding any Lisp or CLOS value.

A template for object instantiation is called a class, just as in other OO languages, although unlike the classes of all other OOLs, it does not define any code associated with the object. Instead it defines:

- The name of the type of the object
- The number of slots in the object
- The names of the slots

The `make-instance` generic function, given as parameters the name of a class and the initialisation values for the slots, makes an object that is an instance of that class and fills in the slots with the given values. The result is a reference to the object which can in turn be assigned, stored and passed as a parameter just like any other Lisp/CLOS value.

3.4.2.3 *Run-time deletion*

Both CLOS and Dylan are intended for non-distributed, interactive programming, and the language specifications stipulate that objects that are no longer accessible (that is, whose references are no longer stored in any other data structures accessible to the program) should be automatically deleted and their storage recycled. Most Common Lisp (and therefore CLOS) implementations achieve this via garbage collection mechanisms of varying degrees of sophistication. This is ultimately the safest and most convenient way of managing object deletion, although with some garbage collectors it may lead to unpredictable pauses in program execution while the GC runs. The perfect, non-intrusive garbage collection scheme is the Holy Grail of Lisp programming.

3.4.2.4 *Inclusion polymorphism*

CLOS classes can be defined in terms of other classes via a form of implementation inheritance. In essence, a class definition that specifies inheritance from another class defines a new class that includes all the slot definitions of the earlier class, and adds some new slots. The inherited-from class is called the superclass, and a class definition may specify multiple superclasses, in which case the new class defines all the slots in all the superclasses. Any method that operates on instances of any of the superclasses will also operate on instances of the new class, since all the named slots from the superclass will be present. This provides CLOS’s inclusion polymorphism for objects.

Classes definitions can specify multiple superclasses, providing a form of multiple inheritance. However, if a class definition inherits two or more slots with the same name from different superclasses, there will only be one slot of that name defined in the new class. This can have unexpected consequences, and in practice this multiple inheritance is not used much in real programs.

3.4.2.5 Object Handle Semantics

Object references are the same size as every other Lisp value, and so can be passed as parameters and stored in just the same way as any other Lisp data. Given an object reference, any client can use it to manipulate the contents of the object's slots using the `slot-value` function, which allows manipulation of the slots by the names they are given in the class definition - this ability is not restricted to methods associated with the object. Hence both the structure and contents of an object are visible to the entire rest of the program, and there is no encapsulation[1].

Common Lisp possesses four (4) equality predicates that can be applied to any pair of values in the language (`eq`, `eql`, `equal`, `equalp`). Without going into too much detail, `eq` and `eql` can be used to compare the pointers to objects, and will return false if they refer to different instances of the same class that happen to have the same values in the slots. `Equal` and `equalp` recurse over the contents of the objects, and return false only if they differ in structure or content. Allowing any holder of a pair of object references to compare them in this way is a weak violation of encapsulation, but insignificant compared to the major violation of allowing any holder of an object reference to manipulate the object's slots.

Finally, object handles can be passed as the parameters to invocations of generic functions, which in turn leads to the subject of method resolution.

3.4.2.6 Method resolution

A generic function in CLOS is a collection of different functions (called methods) with the same name. These methods are free-standing; they do not form part of any object. However, the parameter lists of different methods in the same generic function differ in the types specified for their parameters. When a named generic function is applied to a collection of parameters it is these parameter specialisers that determine which of the methods will actually be executed on the parameters.

As an example, here are a couple of trivially simple method definitions for a generic function called `what-type`:

```
(defmethod what-type ((x integer))
    (print "That was an integer with value:")
    (prin1 x)
)

(defmethod what-type ((x float))
    (print "That was a float with value:")
    (print1 x)
)

(defmethod what-type ((x t))
    (print "Neither float nor integer")
)
```

1. Although there is a module system in Common Lisp and CLOS, it doesn't help much. It doesn't encapsulate values, but instead controls the visibility of names, and is notoriously difficult to control. If an object is passed outside a module then it's still possible that the recipient will be able to use it to manipulate the object.

When the generic function `what-type` is called with a parameter of type integer, the first method will be selected and applied to the argument, yielding an appropriate message, while if it is a float the second one will be called. If it is neither an integer or a float the third one will be run.

Although this example is a very trivial application of the CLOS method selection rules, it can immediately be seen that the process is completely different to that employed by any other object-oriented language in this report. Rather then the invocation being (conceptually) directed to a specific object which uses the *name* of the method (or something derived from it) to select the method to run from amongst those associated with the object, CLOS invocations are (conceptually) directed at a generic function, which uses the *types* of the arguments to determine what code to run. Methods can be specialised on defined classes as well as built-in types like integer, and so it's implicit in the CLOS philosophy that there is only one, global class hierarchy per CLOS program.

CLOS method selection is significantly more flexible (and therefore complex) than can easily be explained in less than several dozen pages. For instance:

- Methods can be specialised on the types of multiple arguments simultaneously (known as multimethods)
- If no method is available whose type specialisers exactly match the types of the arguments then the method having the specialiser(s) that are the next least specific (such as the nearest superclass) is chosen. In the example given, the last method is chosen for arguments that don't match the first two because `t` is defined to be the least specific type specialiser[1].
- One can also separately define methods which execute before, after, or around the standard method, or redefine the method selection algorithm so that *all* the methods of the generic function get called in order of specificity, or all get called and have their results added together, or a host of other options.

To be fair, very few programmers use even multi-methods, let alone some of the other bizarre options for controlling the method dispatch, and in Dylan most of these options have been removed, leaving just the basic mechanism for selecting which method to run based on the types of multiple arguments. However, even the cut-down Dylan model still makes some important assumptions:

- That there is one, globally-agreed type name space
- That no-one changes the type name space whilst methods are executing

These assumptions are invalid outside single-user interactive programming environments.

3.4.2.7 Interaction model

The CLOS and Dylan interaction models are simple, synchronous invocation. There are no other options. Neither language includes specific support for concurrency provision via threads, but neither is it excluded. However, multithreading Lisp-like systems can be tricky, since concurrency and garbage collection don't mix well. Program execution cannot usually be arbitrarily pre-

1. If you don't have a background in Common Lisp, you may be puzzled by the significance attached to this one letter. I'm afraid this is something you're just going to have to learn to live with

empted for garbage collection, since the run-time system often temporarily puts the storage heap into inconsistent states, and conversely, most garbage-collectors do the same when they run. In practice, most Lisp-like programming languages are single-threaded.

3.4.2.8 *Metadata*

In common with most Lisp and Lisp-like programming systems, CLOS and Dylan implementations keep all the information about the program being executed in memory at run-time, so there is complete meta-data; indeed, this is the reason why programming environments in the Lisp world are so powerful and informative. Information about such details as the superclasses of a particular class, or the list of all the methods that make up a generic function, can be obtained from within the program itself by executing a few function calls. CLOS (but not Dylan) also contains a complex and powerful meta-object protocol, which can be used to alter almost any aspect of the behaviour of the language, effectively turning it into a completely different programming language altogether, which has nothing in common with the language described in this section.

Fortunately few programmers venture into this territory.

References

[BOBROW 83]

Bobrow Daniel G. et al., *The LOOPP Manual*; Xerox Corporation, 1983.

[CORBA 93]

Digital Equipment Corporation et al., *The Common Object Request Broker: Architecture and Specification, revision 1.2*; **OMG 93-12-43**, Object Management Group, December 1993.

[IBM 94]

IBM, *SOMobjects: A practical Introduction to SOM and DSOM*; **GG24-4357**, IBM, July 1994.

[KEENE 89]

Keene, Sonya E., *Object-Oriented Programming in Common Lisp*; Addison Wesley, 1989.

[LINDEN 93]

van der Linden R. J., *An Overview of ANSA*; **AR.000.00**, APM Ltd., Cambridge U.K., May 1993.

[MICROSOFT 94]

Microsoft Corporation & Digital Equipment Corporation, *Common Object Model Specification*; Microsoft document, September 1994.

[POWELL 93]

Powell Mike, *Objects, References, Identifiers and Equality White Paper*; **OMG 93-7-5**, Object Management Group, July 1993.

[STEELE 90]

Steele, Guy L., *Coomon Lisp, the language (second ed.)*; Digital Press, 1990.