Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Distributing Objects

Andrew Herbert

Abstract

The ANSA Computational Model introduces the concept 'object' as a unit of encapsulation and distribution. Similar concepts, also called 'objects', have appeared in several other areas of computing, from object-oriented databases, object-oriented programming languages, and Smalltalk, application environments, ET++, HP New Wave Environment and graphical user interfaces.

These other uses of the term 'object' have been reviewed by Alan Snyder of HP in a technical report called “The Essence of Objects”. This report reproduces Snyder's analysis of object-oriented concepts and show how the additional concerns that arise when thinking about software for distributed, heterogeneous, multi-vendor distributed systems lead to a refinement of the essence of objects into set of concepts identical to the ANSA computational model.

APM.1009.01 **Approved** 24 May 1994
Technical Report

Distribution:
Supersedes:
Superseded by:

Distributing Objects

Distributing Objects

Andrew Herbert

APM.1009.01

24 May 1994

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	(0223) 323010
INTERNATIONAL	+44 223 323010
FAX	+44 223 359779
E-MAIL	apm@ansa.co.uk

1 Introduction

The ANSA and ODP Computational Models introduce the concept ‘object’ as a unit of encapsulation and distribution. Similar concepts, also called ‘objects’, have appeared in other areas of computing, from object-oriented databases (such as Iris [FISHMAN 87]), object-oriented programming languages (such as C++ [STROUSTRUP 86], and Smalltalk [GOLDBERG 83]), application environments (such as MacApp [SCHMUCKER 86], ET++ [WEINAND 88], HP New Wave Environment [HP 89]) and graphical user interfaces (such as the HP New Wave Desktop).

These other uses of the term ‘object’ have been reviewed by Alan Snyder of HP in a technical report called “The Essence of Objects” [SNYDER 90].

This ANSA Technical Report reproduces Snyder's analysis of object-oriented concepts and discusses the additional concerns that arise when thinking about software for distributed, heterogeneous, multi-vendor distributed systems and the benefits that come from using object concepts to build distributed systems. This report is therefore both and analysis and critique of object-orientation, from the point of view of distributed computing and an introduction to the ANSA computational model for an object-oriented perspective.

The audience is assumed to have some familiarity with common programming and computer systems: detailed understanding of either objects or ANSA is not a pre-requisite.

(Text quoted from Snyder's report are shown in italics. Without the benefit of the framework offered by his work, this ANSA report would have been much harder to write.)

2 The essentials of objects

Snyder characterizes the essential principles of objects as follows:

1. *An object is not just bits*
   - 1a. *An object embodies an **abstraction***
   - 1b. *An object provides services*
2. *Clients **request** services from objects*
   - 2a. *Objects are **encapsulated***
   - 2b. *Clients issue requests*
   - 2c. *Requests are named*
   - 2d. *Requests identify objects*
   - 2e. *Requests may take arguments and produce results*
   - 2f. *Services can be described*
3. *Requests can be generic*
4. *Objects may be organized **hierarchically** in terms of the services they provide*
5. *Objects may be organized **hierarchically** in terms of the degree to which they share a common implementation*
   - 5a. *Objects may share a common implementation (multiple **instances**)*
   - 5b. *Objects may partially share a common implementation (**implementation inheritance**).*

The following sections explore the impact of distribution upon these principles and shows how ANSA concepts can be reconciled with other kinds of object-orientation.

3 An object is not just bits

An object is not just a data structure. It embodies an abstraction that is meaningful to its clients (users or programs). The purpose of the object is to represent abstract information. An object is more than just information: it provides a ***service*** *to its clients. This service corresponds to the abstraction. Objects do more than just read and write data. The service iscarried out by executing code that access or manipulates the actual data. The set of services provided by an object is called the* ***behaviour*** *of the object.*

This principle is the primary benefit of object-orientation: it separates the **service guaranteed** by an object from the **technology implementing** that service.

Benefit: Objects provide applications with systems independence - different services can be implemented on different environments selected individually to meet the performance, integrity or connectivity requirements of each service in turn. Object orientation both requires and provides a uniform way of invoking both local and remote services.

Benefit: Objects enable controlled, incremental evolution of a system since the location and even the implementation of a service - in terms of programming language, operating system, hardware - can be changed without modification or disruption to any of the clients of that service.

4 Clients request services from objects

Clients respect the intent to use objects to represent abstractions. Instead of directly accessing the data representing the information embodied in objects, clients issue requests for service that are carried out by objects.

4.1 Objects are encapsulated

This principle reveals the merit of selecting the object-oriented approach for distributed applications: since objects are encapsulated they can be separated from their clients provided that some means is provided for clients to remotely access an object's services. Applying the same argument in reverse, distribution enhances object-orientation - it enforces the encapsulation of objects and prevents direct access to data by physically separating them.

In non-distributed systems, the benefit of encapsulation is to guarantee that an object satisfies application-defined integrity constraints since there is no direct client contact with the data. The encapsulation property of objects equates two notions - objects as:

1. a unit of service, or unit of representation. This is an object in the sense of a design, or a representation of a part of a system. Within the object the design or representation can change without affecting the rest of the system.

   Benefit: objects provide strong modularity in design and permit incremental development and evolution of designs

2. a unit of programming: an object in an object oriented programming language, for instance. A unit of service would be made up of one, or more, units of programming.

   Benefit: objects in programs can be subject to scope checking to ensure a program maintains the modularity of the design it implements and separately compiled and linked into multiple programs.

In distributed systems there are further integrity constraints which have to be met by encapsulation: the different types of encapsulation needed are:

3. A unit of distribution - encapsulation for objects that may be located and even migrate around a distributed system, independently from other objects, but which must remain integral within themselves

   Benefits: objects can be relocated without modification to their clients to make best use of available resources and deliver optimal performance to users; objects can migrate to offload functionality from a processor being taken out of service, to balance load between processors, to bring data to the processor where it is being used to reduce latency, to move data out to storage services when it is not in current use without involving the object's clients

4. A unit of failure - encapsulation for objects such that all of an object fails, or none of it does. This is usually achieved by keeping the whole of an object on the same processor

Benefits: limits on the propagation and extent of possible failures simplifies the isolation of faults and error handling; distributed applications can be written to provide 'graceful degradation' in the presence of individual object failures.

5. A unit of replication - multiple copies of objects can be provided in different locations working together through a consistency protocol to provide the illusion of a single service supported by all the copies

Benefits: replication techniques can be used to make highly available and fault tolerant implementations of critical objects

6. A unit of security - encapsulation such that all of the components of an object are subject to the same access control rules. This usually means the components belong to the same principal and that interaction within the components of the object is not subject to access control

Benefits: hardware protection mechanisms can be used to physically enforce object encapsulation; encipherment can be used to physically protect request and responses between objects.

In a distributed system, the boundaries of the various kinds of units listed above are not identical; unfortunately the term 'object' is used to mean any of them, sowing considerable confusion when object-oriented programmers first meet distributed systems. In fact, all of these units exist in an object oriented programming system, but they are implicit and equated to the program unit. In other words, the object oriented programming concept of an object is a simplification of the more general concept of object which arises in object oriented distributed systems.

The challenge in a distributed system is to manage the different units (objects) at run-time with a mixture of programming language and configuration tools; but without the simplification of a single programming environment and recompilation assumed by current object oriented programming systems.

4.2 Clients issue requests

Clients issue requests for services that are carried out by objects. A request causes [program] code to be executed to perform the requested service. The details of where and how this code runs is not of concern to the client.

In a distributed system there may be many client objects s on separate processors, all simultaneously requesting the same service and therefore a server object must be able to exercise concurrency control over requests. Concurrency control takes two forms:

1. ordering and synchronization: the server may require that requests be processed one at a time, or that only certain sequences of overlapped execution are possible (for example, producers and consumers can both access a finite buffer while there are free slots in it), or that only a certain number of requests be executed at once to limit the consumption of resources

2. separation: the server may require that sequences of requests from separate clients be scheduled in an order that avoids conflicting updates to the data contained in the object (for example conflicts between reads and write to the records of a data base object).

A client may wish to abandon a sequence of requests if an intermediate request produces a particular result (e.g. a debit on an empty account), or if an

object invoked as part of the sequence fails. This requires that the execution model support the notion of committing and aborting sequences of requests and the correct interplay of commit and abort with the scheduling mechanisms for separation (i.e. transaction processing capabilities).

In a distributed system objects may be remote from their clients, introducing a latency due to the overheads of communication. A client may be able to reduce latency by making concurrent requests to different objects (or even the same object if its concurrency controls permit). Furthermore, it must be possible to select that a concurrent request is to become either part of a transaction or if it is to start a new independent transaction.

4.3 Requests are named

Requests are typically named. To make a request, the client identifies the request by name.

4.4 Requests identify objects

To make a request a client must identify one or more objects to perform the requested service. An object can identified directly or reliably. Object reference is direct in the sense that one is naming the object not describing it. Object reference is reliable in that, within certain limits of time and space, repeating reference to an object will reliably access the same object.

Requests are directed towards objects which are units of service; it may be that an object which is a unit of distribution encapsulates several units of service. The latter can be conveniently termed the **interfaces** of the distributed object (and thus objects in object oriented programming languages can be equated with distributed objects containing just one interface). For example, there may be some data which is encapsulated in a single object for reasons of security, but for which there is the notion of both 'user' services and 'manager' services. The two forms of service can be readily distinguished by putting each in a separate interface. In a distributed system requests identify **interfaces**.

Very few distributed systems have the notion of 'system restart'; they are required to remain in continuous operation and so an object (i.e. interface) reference has to retain its meaning for all time. Nor can it be assumed that separate distributed systems will never become joined (for example when the organizations merge or do business with one another) and so an interface reference has to retain its meaning throughout space.

It may not be possible in a distributed system to distinguish between an object which cannot be accessed because of disruption of communications and an object which has become lost from the system because it did not take steps to ensure its reliability. Clients must be prepared to cope with the failure of communication, and objects which use replication to increase their stability must take steps to ensure the replicas present a consistent service to their clients even if replicas are unable to communicate with one another.

Since objects may migrate in a distributed system, interface references must be **location independent** names. A distributed system must include a **relocation service** for discovering the current location - i.e. address - of objects which have migrated so that requests can be delivered to the correct

place. (An interface reference can include an address hint for the interface to save on the time overheads of name to address resolution and so that the locator only has to know about interfaces which have actually moved, thereby saving the potentially vast overhead of storing name to address translations for all interfaces).

Client to server configuration can be **early** or **late**. In early configuration client and server are made together and an interface reference for the servers embedded in the client. Late configuration is a dynamic process, called **trading**. An object providing a service registers (or has registered on its behalf) a description of the service provided and its interface identifier with a **trading service**. A client object wanting to use a service queries the trading service, and if a matching service offer is found its interface reference is returned. The client can then use the interface reference to make requests.

Both location and trading services may be built upon name services which provide name-to-name translations.

Service descriptions given by servers and clients must necessarily describe the range of services available at the interface so that the client gets access to an object providing at least the service required. There may be many objects providing a suitable service and therefore service descriptions may involve names and attributes to permit disambiguation between service offers. For example a service may be further qualified by its location, who owns it, how secure it is, how fast it is, how robust it is against failures, and so on.

Benefit: Providing a trading service makes a system configuration open-ended - new objects can be added to the system and made accessible to existing clients without requiring that the clients be rebuilt in any way. Interface references (i.e addresses) need not be built into programs.

4.5 Requests may take arguments and produce results

Particularly in a computational context (as opposed to a user interface), it is commonly the case that a request may have associated argument values (which may be object references) and the service may return one or more results (which may also be object references) when it completes.

In a distributed system all arguments and results have to be either interface references or immutable data types (i.e. integers, booleans, characters etc.). It is not meaningful to pass pointers since client and server may not be on the same computer. (Some systems give the illusion of passing pointers by wrapping them up as an interface reference to a service for accessing memory locations, or by copying the data referenced by the pointer - reinforcing the need for the interface reference concept).

Passing an interface reference gives the recipient the right to share in the use of a service (hence the need for the concurrency controls mentioned in Section 4). Passing an immutable data type requires that a copy of the data type be made at the recipient. In a computational model in which all data types are objects, both these schemes can be viewed as providing sharing semantics, as can other schemes such as migrating the object to the recipient, or replicating the object so that both sender and recipient have local copies kept in step by some sort of consistency protocol.

Benefit: treating all arguments and results as interface references (i.e. a pure object model of data) provides a clean computational abstraction of a

wide range of argument and result passing schemes. Alternative schemes can be substituted without requiring changes to the programs involved.

Many programming languages only permit a request to return a single data type as result. Often what is returned is a memory address for a data structure made by the called service. Since memory addresses cannot be permitted as results in distributed systems, an interface reference to a result object constructed by the service would have to be used instead. But this then imposes a significant latency overhead when the recipient of the reference tries to access the object. Therefore in a distributed system it should always be possible to pass multiple arguments and obtain multiple results.

In a distributed system a request may fail because of some communications problem or resource limitation. This fact has to be conveyed back to the caller as an abnormal outcome of the request. It may also be that the service has several possible outcomes. These could be encoded as a datatype - a discriminated union for example; alternatively a general mechanism permitting a request to generate different outcomes could be provided, with facilities for the requester to take different actions depending upon the particular outcome obtained for any given request.

The synchronous request-response style of interaction is well suited to distributed computing. It matches well with the concept of remote **procedure call** found in many distributed systems architectures. It also fits well with the concept of **nested transactions** as a way of providing the kinds of atomicity guarantees given in Section 4.2. Note that a request which returns no results is strictly a request that returns an 'empty' response - the response contains no results, but there is an explicit indication of termination. Request-response interactions create chains of dependent nested calls. Request-response calls are subject to latency since the client making the call has to wait for the call to complete before doing anything else. In some situations this can be overcome by making the call in parallel with other work that can be done and synchronizing with the results of the call only when they are required. Alternatively if there is no need to synchronize with the processing of the request by the server an alternative 'fire-and-forget' style of interaction can be used. When using this style of interaction, the programmer must take care to ensure that subsequent requests do not assume that the previous ones have been completed.

It is useful to compare conventional object-orientation and remote procedure call in terms of the object with the "multiple interfaces, each interface supporting several services" model outlined above. Conventional object-oriented systems merge the concepts of object and interface into the single concept 'object' and thereby lose the ability to determine which services are available to which clients and the ability to distinguish clients by giving each one a separate interface and associating client state with the interface used. Remote procedure call systems merge the concept of interface and request together into the concept 'procedure'' and thereby lose the benefits of encapsulation and abstraction that come from grouping services together as outlined in Section 2. Some remote procedure call systems do provide the notion of interface so that services can be grouped to form an abstraction, but they do not provide means to pass such interfaces as arguments and results, and therefore lack the flexibility of object-oriented systems. The general object / interfaces / services model supports both conventional object-orientation and remote procedure call as a special case.

4.6 Services can be described

The set of services provided by an object to its clients is often made explicit to clients in the form of an interface description that identifies a set of requests that can be made to an object. This ***interface description*** *is sometimes called a protocol. Often this specification will include information about the expected arguments and results associated with each request. Such a specification is sometimes called a signature.*

(In distributed systems the term protocol usually refers to the means provided by networks to copy data from one computer to another.)

Services must be described by signatures in distributed systems, since clients and servers are often written by different programmers in different locations and at different times. The signature provides a contract between the two programmers, telling them what service is to be provided at an interface to an object. It may be that the two programmers use different languages to write their programs and the programs run on computers with different data representations. The signature provides the information needed to automatically generate the data type conversions needed to permit interworking between client and server.

Benefit: signatures permit decoupling of client and server programs and the use of multiple implementation languages.

As discussed in Section 2d, there may be many objects with the same signature. Therefore in a trading system additional attributes beyond signatures must be used to distinguish between different offers that have identical signature.

4.7 Requests can be generic

A client can issue the same request to different kinds of object that provide 'similar' (at least snynomous) operations. Depending upon which objects are identified by the client, different code may be run to perform the requested service. The selection of code to execute is based on the object identified by the clients in the request. In the general case, the identification of objects is not determined until the request is actually issued, so the selection of code would happen at that time. (In some cases information exists at compile time or link time to statically bind a request to the code that will implement it). Also in the general case there is no limit on the number of different kinds of object that may support a given request.

The benefit of generic request in distributed systems is that services can be more general, which implies more reusable, and that users profit by being able to apply a standard model in many cases. For example all objects can be made to support a common management model by requiring they offer a common management interface.

An open system is one in which new objects can be introduced dynamically, such that the new objects can be operated upon by existing clients, without changing the existing clients. The existing clients are able to use the new objects because the new objects support the requests for generic services made by the existing clients.

The use of trading to discover available services makes a system open-ended. The trader matches the type required by the client to the type offered by the

server. This test can be made one of **conformance** rather than equality. Conformance means that the server does at least what the client requires, may be more. It enables servers to be upgraded without disrupting clients.

Benefit: openness is mandatory requirement for practical distributed systems.

5 Services may be organized hierarchically

Objects can be classified in terms of the services they provide to clients or equivalently, in terms of the requests that can be made of them. Objects that provide the same set of services would be classified together. This classification may be based on explicit descriptions of services (or requests) called interfaces.

[Note: since the term 'interface' has a particular meaning in distributed systems - see 4.4 above - classifications of services will be called **types** in the following discussion.]

An object could provide a subset of the services provided by another object, leading to a hierarchical classification. This interface hierarchy can be used as a type hierarchy in describing permissible values for arguments to procedures in a program etc.

Benefit: a classification of objects based on their services is a way of organizing objects to make their behaviours easier to understand. A classification of object services can also be used to describe the services expected of an object by a client.

Benefit: checking the type of an object before it is used increases the safety of a system - it is a guarantee that server will understand all of a client's requests and produce results that the client can understand.

The need for type descriptions - signatures - in distributed systems was discussed in Section 4.6. Organizing types hierarchically eases the burden of writing such descriptions since a complex service can be defined as being an extension of a set of simpler services. This is particularly useful when there are large numbers of generic requests that can be made of an object.

The existence of a type hierarchy means that the model of type checking in a distributed system should be one of **type conformance** rather than type equality: a client request is acceptable to an object if the object is capable of responding to the request, if the client offers arguments which conform to those expected by the server and if the server returns results which conform to those expected by the client. The use of type conformance increases the genericity of objects and hence the openness to service evolution in a system since it permits an object to incrementally 'widen' the type of an interface without disrupting the client.

6 Implementations may be organized hierarchically

Objects may be organized hierarchically in terms of the degree by which they share a common implementationMechanisms are generally provided [in object systems] to allow different objects to share the same implementation. Mechanisms are often provided by which the implementation of one object can not only share the implementation of another object, but also extend or refine it.

6.1 Objects may share a common implementation

T*he implementation of an object generally specifies both the format of the data used to represent the information associated with an object and the code used to implement the services it provides. Mechanisms are generally provided to allow different objects to share the same implementation. Objects that share a common implementation have identical data formats and share executable code; however each object has its own copy of the actual data. Objects that share the same implementation would be classified together. Each object can be thought of as an instance of the common implementation.*

It is the sharing of implementations that is of primary interest, not the classification resulting from it. In general clients should be concerned with the services provided by an object, not how the services are implemented, and thus should not be interested in an implementation-oriented classification.

Benefit: sharing one implementation among many objects has the obvious benefit of reduced source code duplication (which eases maintenance by avoiding the need for manual propagation of changes) and reduced executable code size (where sharing of executable code is possible).

In a distributed system there may be instances of a common implementation on many different computers with different data representations and instruction formats, so sharing of a single execution image and data format is not possible. (Where several instances reside on the same computer, they may share an execution image and data format)).

Some object systems are **reactive** in that a change to the source code for a set of instances is immediately reflected in a change in behaviour of the instances. This is a difficult effect to achieve in a distributed system; it raises many questions about consistency and atomicity since there is not a single copy of the executable code to be updated.

Programmers may wish to exercise control over where an instance is created. This is readily accomplished by providing **factory** services which create new instances upon demand at specific locations. A factory service embodies a template for the class of which the object to be made is an instance. All the objects made by the same factory may share the same data format and executable code.

If an object is to be able to migrate from one computer to another means must be provided for the object to externalize itself into a representation which can be moved between machines and re-instantiated at the destination. In the general case, the external representation must include information about all the activities that were taking place inside the object at the moment it began migrating. If the source and destination computers are identical, the external representation can be close to the internal representation. If the potential destinations for a migrating object are known in advance the size of the external representation can be reduced by prearranging for the code and data formats to exist at the destinations.

6.2 Objects may partially share a common implementation

Mechanisms are often provided by which the implementation of one object can not only share the implementation of another object, but also extend or refine it. (Such mechanisms are generally called inheritance mechanisms.) In this case of partial sharing implementations, the classification of object implementations becomes hierarchical.

Benefits: in addition to the maintenance and size benefits listed above, partial sharing of implementations extends the benefits of software re-use to cases where implementations are similar but not identical. Partial sharing is a useful technique for encouraging consistent behaviour among related objects.

Inheritance is often singled out as the primary feature of object-orientation and is justified as a vehicle for achieving software re-use. Unfortunately, inheritance has a number of problems that make it unsuitable for general use in distributed systems [RAJ 89]:

1. Encapsulation is violated - Inheritance may violate encapsulation in at least three ways: a subclass may (a) refer to data defined in the superclass, (b) request an internal service of the superclass, and (c), refer to the superclasses of its superclasses. The consequence of this in a distributed system is that the locality of objects is lost - inheritance introduces object dependence on unknown, potentially remote, inherited information defeating the major benefits of objects as independent units of migration, failure propagation and security. The crux of the issue is that much of what is inherited is implicit - inheritance lacks the succinct and explicit of interface found in the service side of the story

3. Classes are not automatically reusable - For successful reuse, inheritance requires the use of a set of coding rules and a set of design rules to ensure consistent interpretations. In a distributed context it is not viable to expect all code to be written to the same conventions except in as much as there must be a commitment to the same means of interaction between objects. The internal structure of objects is a local concern guided by the implementor's local rules. (Indeed it cannot be assumed in a distributed system that all implementors are using object-oriented languages, let alone the same language and the same inheritance structure!)

4. Class organization is not scalable - Inheritance is successful where software is written by a few people working together with an agreed hierarchy and where the number of classes is hundreds at most; inheritance falls down when there are large numbers of classes involved or where there are large numbers of people involved; the problem is mostly one of keeping track of the implicit aspects of inheritance pointed out in (1) above

5. Reactive inheritance is difficult to achieve - Reactive inheritance requires a consistent, atomic update be made to all members of a class and its subclasses wherever they are located

6. There should be no linkage between typing and implementation - the desirability of types as a means to permit multiple implementations of the same service has already been discussed. Many object-oriented systems use inheritance as a substitute for type checking - two objects are deemed to be of the same type if they are made from the same components. This is too restrictive a view for a distributed system. Implementations have no part to play in the classification of services. Thus simple-minded implementation inheritance has little part to play in distributed systems. In fact, the objectives of maintenance and re-use must be met by more structured techniques for the identification, sharing and composition of source code components alone. (Only where a same component is referenced several times by objects which are going to be co-located on the same computer is there scope for sharing object code as an optimization). If maintenance and re-use are conducted at this level, many other structures beyond inheritance become available for classifying and linking together software components. Distributed systems are facilitated by objects whose definition and implementation are fully self-contained.

7 References

[BLACK 86]
Black A, Hutchinson N, Jul E, Levy H; **Object Structure in the Emerald System,** Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 86). ACM Sigplan Notices **21** ,11 pp78-86 (November 1986)

[BLACK 87]
Black A, Hutchinson N, Jul E, Levy H, Carter L; **Distribution and Abstract Types in Emerald,** IEEE Transactions on Software Engineering **SE-13**, 1 pp65-76 (January 1987)

[FISHMAN 87]
Fishman D H, et al.; **Iris: An Object-Oriented Data Base System,** ACM Transaction on Office Automation Systems **5**, 1 pp48-69 (1987).

[GOLDBERG 83]
Goldberg A, Robson D; **SmallTalk 80: The Language and its Implementation,** Addison-Wesley, Reading, Massachusetts (1983)

[HP 89]
Hewlett-Packard, **HP New Wave User Guide,** Part number 5958-9678, (August 1989)

[LISKOV 88]
Liskov B; **Distributed Programming in Argus**, Communications of the ACM **31**, 3 pp300-312 (March 1988)

[RAJ 1989]
Raj R K, Levy H M; **A Compositional Model for Software Reuse,** Technical Report TR 89-01-04, Department of Computer Science, University of Washington, Washington (1989).

[SALTZER 78]
Saltzer J H; **Naming and Binding of Objects,** Operating Systems An Advanced Course (LNCS 60), Springer-Verlag (1978)

[SCHMUCKER 86]
Schmucker K J; **MacApp: An Application Framework,** Byte **11**, 8 pp189-193 (August 1986).

[SNYDER 90]
Snyder A; **The Essence of Objects,** Report STL-89-25, Hewlett-Packard Laboratories, Palo Alto, California (1989)

[STROUSTRUP 86]
Stroustrup B J; **The C++ Programming Language,** Addison-Wesley, Reading, Massachusetts (1986)

[WEINAND 88]
Weinand A, Gamma E, Marty R; **ET++ - An Object Oriented Application Framework in C++,** Proceedings of the 1988 ACM Conference

on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 88). ACM Sigplan Notices **23**, 11 pp46-57 (November 1988).