Distributed System Design using CORBA Components

Douglas I. Donaldson & Jeff N. Magee
Department of Computing, Imperial College of Science, Technology and Medicine,
University of London, 180 Queen’s Gate, London SW7 2BZ, UK
mailto:{did,jnm}@doc.ic.ac.uk

Abstract

Systems designers have distributed object technology available to them, such as implementations of the OMG’s CORBA. However, object-oriented design methods mostly consider development of sequential systems. Few methods consider the problems of designing for distribution or give guidance to allow the benefits of distribution to be realised. This paper proposes the environmental object model and design method based on a containment hierarchy, with constraints on links between instances. The model and method promote the development of complex distributed abstractions, implemented as a system of CORBA servers. Throughout the paper, a case study of a distributed grouped diary service is used.

1. Introduction

Object models and object-oriented design methods are rapidly becoming accepted technologies for systems design. They are popular because they are based on principles of abstraction, encapsulation and information hiding, and support incremental development and software reuse [1].

Distributed systems offer advantages of increased system availability, performance, reliability and scalability. Object technology for distributed systems is also maturing, with enabling infrastructure such implementations of the Object Management Group’s (OMG) Common Object Request Broker Architecture (CORBA) available for a variety of networked systems.

CORBA is the centrepiece of the OMG’s Object Management Architecture, and specifies a standard mechanism for a client (object) to request the services of a server (object) [2]. Service types are declared in an Interface Definition Language (IDL). A client may send a request via a local stub which (statically) supports the server’s IDL interface. Alternatively, it may dynamically compose a request and send it via a Dynamic Invocation Interface (DII). An Object Request Broker (ORB) provides services to communicate the request to the server. A family of Interoperability Protocols (IOPs) allows ORBs with different implementations, on different processing and networking hardware to communicate, for example over the internet.

Motivation There are difficulties in using existing object-oriented design methods for developing distributed systems. Design methods are usually implicitly aimed at developing centralised, sequential systems. They do not exploit the advantages of distributed systems, or suggest how to integrate distributed algorithms to solve the problems of distribution such as synchronisation, replica management or hardware failure.

The problems appear to lie with the object models which form the foundation of design methods. These are usually derived from the object models of sequential languages (following the pattern set by SA/SD which was founded on the ideas of structured programming). This makes them suitable for developing sequential implementations. The object models are weak at expressing the internal concurrency and synchronisation requirements of complex, distributed abstractions, and so the design methods do not promote development of distributed services.

This paper proposes an object model and design method created for developing complex concurrent and distributed abstractions. The environmental object model (section 2) uses a component hierarchy for system structure. The environmental design method (section 3) takes advantage of its underlying model in order to construct distributed services using CORBA servers.

Diary System Concepts A case study of a distributed grouped diary system is used as an example of environmental design. This was conceived as a service for the members of the authors’ research section. All members (identified by login name) have a public diary. In addition, group diaries can be created for useful groupings of members, or for groups of groups. The timetable for a person or group consists of the entries from that person or group’s diary plus all the entries from all the groups the diary owner is a member of, directly or indirectly. This allows events and meetings of groups of members to be easily scheduled. Examples of the groups that might be usefully created are students, researchers, lecturers, administrators, seminar-group, whole-section, etc. The diary service is distributed, to improve reliability and service availability.

2. Environmental Object Model

The following two complementary observations were important during the development of the environmental object model:

• Bottom up If a system of objects are cooperating to provide a service, then there are software engineering benefits to be gained from identifying a new object type which is an abstraction of the cooperative service. The implementation of the new class can be closely based on the mechanisms of the original collection.

• Top Down When a complex system needs to be developed, it is again beneficial to treat the run-time system as an object with interfaces which provide the system services. The system object’s operations control
private cooperating components to implement the system’s responsibilities.

The environmental object model builds on these observations by restricting system structure to a containment hierarchy of objects, with object links constrained by the hierarchy. The model combines the best features of component models [3] and object models. It is component-oriented because it promotes logical structuring of a system as a hierarchy of components. It is object-oriented because the system structuring principle does not interfere with the use of classes to implement the objects at all levels. Class relationships such as specialisation/generalisation and genericity are still needed to express commonality and allow incremental development.

Definitions

An environmental object system is an object system where the instance structure graph is a containment tree (connected directed acyclic graph), and the links from any instance are constrained to lead to only that instance’s parent, siblings (including itself) and children in the containment tree. The root instance of the tree contains every other object (directly or indirectly) and is known as the system object.

The Unified Modeling Language (UML) is used for class and object diagrams [4]. Object diagrams for environments use an object composite style to emphasise the containment structure (figure 1). Multiple non-distinguished objects are shown with a stack icon. In an object diagram of an environmental system, no link may cross an object’s boundary.

The two most important object associations are links and containment. A link is a one-directional association indicating a physical connection between two objects. A client object may call methods of the server objects it holds links to. Containment (also known as aggregation) is a structural association indicating that a container object or environment conceptually holds or surrounds the contained object or component. The term containment usually means the direct containment association just described, and is not transitive. However, the indirect containment association is transitive, and can be useful. A container indirectly contains its components’ components, their components, and so on.

An environment is an object which locally conforms to the constraints of an environmental system. (The idea of a collection of components ‘living in’ their container gave rise to the term ‘environment’ here.) Such an object can only have links to itself, its container (its own environment), other objects which its container contains (its peers), and its components (or members). If every instance in an object system is an environment, then the object system is environmental.

A class is a static description of the behaviours of set of objects. An environmental class is a class whose instances are potentially environments. Whether or not the class’s instances are environments depends on whether the clients of the class respect the constraints of an environmental system. A (non-environmental) client could give an environmental class’s instance a link to an object remote in the containment tree, when the instance was expecting a link to a peer. The instance would no longer be an environment.

The rich variety of class relationships available under programming languages complement these constraints on system structuring. Object links and containment associations correspond to using and containment class relationships. A class will also have using relationships with the classes of parameters in its interface, of its operations’ temporary objects and of exceptions it throws or catches. It may have a specialisation relationship with a class it inherits from. It may also be an instance of a generic class. Although a component instance may only have one container (at run-time), the component’s class will not be (statically) nested within the container’s class, but may form containment and other relationships with many classes.

The objects within an environmental system both provide services to and require services of entities outside the system under consideration. Links to and from external objects may exist from and to components at any level of the system structure. This is needed to avoid every communication being routed to the system object. Objects within the system may require services of a range of granularities. The lowest level services such as memory management or mathematical services may be considered uniformly available. For higher level services, such as file or device management, the internal component will need to be given the name of or a link to the resource so that the resource can be used. In a complementary way, the system object, although ultimately responsible for providing all the system services, can delegate these to internal components. System clients can gain links to these and then directly request system services. Module Relationships are needed for physical complexity management during system implementation. The namespace concept for class (module) nesting forms a suitable mechanism allowing collections of cohesive class modules to be grouped. This does not interfere with the logical system structuring into an instance hierarchy.

A collection of servers at multiple sites in a networked system can communicate and cooperate to provide a consistent service interface. Such a collection is an implementation of a distributed object. The view of the object’s state obtained by a client is independent of which server the client gains a link to. The mechanism by which the servers maintain that consistent view of the state is the distributed object’s implementation secret.

The environmental object model is a constrained object model. Most object-oriented programming languages can be used in an environment-oriented way by writing environmental classes to build environmental object systems. However, an environment-oriented programming language would not just enable development of environmental object systems, but would support development.

The environment-oriented language of [5] shows that static checks of a program can be made to ensure that it will construct an object system conforming to the
constraints of the environmental model. It uses type annotations (environment, peer, member or self) to express the structural relation between each object and the objects it uses. Typing rules are used to provide the necessary checks. This type system shows that the environmental model’s constraints are amenable to tool support.

**Concurrency and Synchronisation**

The environmental model treats activities (or threads) as a concept unrelated to objects. Activities may be started as needed, and (in the absence of explicit synchronisation) pass from object to object without regard for object boundaries. Active and passive objects are not strongly distinguished: an object is active if it has any thread activity within it.

Activities are started using asynchronous (or one-way) invocation. This may be simulated using explicit thread creation if asynchronous invocation is not available in the development platform.

Before environmental synchronisation is described, the idea of a *synchronisation contract* is defined. A synchronisation contract forms part of a class’s specification and defines how the class’s objects should behave when clients make multiple concurrent service requests. It specifies the constraints on concurrent service requests by clients and the scheduling policy of the server. The server’s clients must not exceed the limit on requests, and in turn the server must respond to legitimate requests in the manner described in the scheduling policy.

Four examples of placing constraints on clients are given. The clients of a server offering a *sequential* interface may only request one service at a time between them. With a *readers/writer* interface, clients can request concurrent read services as long as there are no clients using write services, but a request for a write service must not be made concurrently with any other request. The clients of a server with a *concurrent* interface are not limited in sending concurrent service requests. Clients must follow some protocol if using an *ordered* interface, for example `lock;op;unlock` or `begin_transaction;use;end_transaction`.

Examples of scheduling policy (all for servers with concurrent interfaces) include *first come first served* (service requests are processed serially in order of arrival time) and *readers/writer* (concurrent reader requests start executing concurrently as long as there are no writers executing, and writer requests start executing in mutual exclusion).

There are many other varieties and combinations of scheduling policy, including buffer policies, timestamp ordering, shortest job next and other client prioritising schemes. Policies such as database schedulers might require significant computation before a request is executed, and have to deal with deadlock, preemption and operation abortion. If a scheduling policy cannot be formally defined, a liveness condition should be given (that all requests terminate eventually).

An environmental class implementation will have to include synchronisation mechanisms which implement the class’s synchronisation contract, and may also have to synchronise the service requests that the environment’s member components make on each other.

The principle of *environmental synchronisation* is that it is the responsibility of the (designer of the) environment object to impose such synchronisation mechanisms as needed on the object itself and its components to ensure that the specified contract is safely fulfilled, and that the system makes progress.

Three patterns of environmental synchronisation are described here. A server which needs to control concurrency at the granularity of operations can be implemented with a serialising *wrapper* acting as a protecting environment for the inner implementation. A server with concurrent internal activities can *wrap a component* (one with a sequential interface) within an environment which serialises concurrent requests. An environment which contains a collection of data components is similar to a database object. The environment’s methods are like transactions over the data. For an efficient implementation, the environment may need to allow the methods to execute concurrently, with the operations synchronising only when conflicts occur. This *transaction scheduling* can be implemented with an internal scheduler object to serialise the operations. The optimum choice of scheduling algorithm will depend on the patterns of data access.

**Benefits of the Environmental Model**

The environmental model offers software engineering benefits of abstraction, encapsulation, information hiding, complexity management, and improved modelling for concurrent and distributed abstractions.

The model allows *complex abstractions* to be defined in environmental classes, such as diary service objects (environments for diaries) or bank objects (environment for bank accounts), which are more than just structural groupings. They have service interfaces and encapsulated implementations. They perform roles as manager objects, controller objects and factory objects for their components, and can provide support services, maintain invariants and enforce constraints. The operations of an environment do not just delegate requests to its components, but can validate the request and coordinate the cooperating components which are used to implement the requests.

The constraints on links within an environmental system enforce the encapsulation and hiding of the implementation of its classes. An environment’s external clients (its peers and its own container) are not allowed to gain links to its components, and so cannot use those links to violate constraints the environment is maintaining.

The structural constraints of the environmental model allow the implementation of an environmental class to be *understood, maintained and re-used* because its behaviours depend on its methods, its components and the external services it uses. The implementation does not depend on objects remote in the system structure.

The physical complexity management mechanisms of systems’ programming languages (such as C++’s namespaceing or SMALLTALK’s class categories) have inspired the logical complexity management mechanisms of popular object-oriented design methods (such as class categories, subsystems or domains). These class clustering mechanisms partition a system model by grouping collections of cohesive classes.

Collections of classes do not form high level abstractions because they do not contribute state or
operations directly, but only through their contained classes. This means that they do not have the advantages of object classes of abstraction and information hiding, and so can be hard to understand and re-use.

By contrast, the containment hierarchy of an environmental object system allows progressively more complex abstractions to be built from components. System structuring this way allows the composition of a system object from components, or the decomposition of a system into objects. Either way, the environmental system’s design can be understood from the behaviours and interactions of the system object and its components. This means that the principles of abstraction, encapsulation and information hiding can be applied scalably.

None of the classes of the Diary System developed in section 3 have fat interfaces or is over-complex when considered in isolation. This is because an environment object is not an omnipotent object, but a manager and controller.

A component structure is particularly beneficial when building distributed systems [3]. It allows a distributed environment to encapsulate the knowledge of where its components are located, and so provide a uniform service interface. It allows modular subsystems to be improved or replaced with a predictable impact on the rest of the system.

An ‘unrelated’ concurrency model was chosen as an aid to the development of complex concurrent and distributed abstractions. It permits the greatest modelling flexibility when designing systems from analysis models. During analysis, it is often assumed that all objects are concurrent, which means that they have the potential to take part in many simultaneous activities. An important cited benefit of object-oriented analysis and design is the potential for analysis models to be carried through as the basis for system design. An object model for design should therefore allow the concurrency of the analysis models to be naturally expressed.

By contrast, in the ‘related’ approach every activity is permanently bound within an object. This model has been popular with concurrent object-oriented programming languages because it offers the possibility of unifying the concepts of object and process. However, the transformation from “all objects are concurrent” during analysis to “all objects are active” during design gives modelling problems: activity is not an inherent attribute of an object. Instead, design objects are better modelled as services, which respond to concurrent requests as needed. This is especially true for distributed objects. The related approach is also criticised in [6].

The ‘unrelated’ thread model is often criticised as being prone to the traditional problems of synchronising concurrent programs. However, the ‘related’ model has not freed programmers from these problems. The synchronisation contracts of the environmental model follow best object-oriented practice by keeping a clear separation between the synchronisation specification of a class and the synchronisation implementation. This extends Meyer’s use of preconditions and postconditions on operations to specify the contract offered by a server [7], and isolates the client from the implementation of synchronisation within a server. It allows high level synchronisation schemes to alleviate traditional synchronisation problems. An implementation of a CORBA server is likely to have a mixture of concurrent and sequential code, and synchronisation contracts make the boundary between concurrent and sequential objects explicit.

Environmental synchronisation takes advantage of the simplified communication structure of environmental systems by only imposing the necessary synchronisation within any object to implement its synchronisation contract and ensure safety and liveness. Knowledge of the patterns of data access is particularly important when choosing transaction scheduling algorithms.

3. Environmental Design Method

This section describes a method for the design of distributed systems based on the environmental object model. The method guides the development of complex abstractions implemented as CORBA servers organised into a logical hierarchy for ease of understanding and reuse.

The presentation is necessarily abstract, with the waterfall method (analyse, design, implement) framing the description. The method can be refined into a design process (a method scaled to industrial size) but the controlling framework for that process would depend on that project’s size, cost, lifetime, quality of specification, funding, personnel, tools, software resources available, wider objectives including design for reuse and so on.

Analysis

Systems analysis is not given detailed consideration here, because the environmental design method is not tightly coupled to an analysis method. This section discusses the relationship between the environmental object model and object-oriented analysis (OOA).

The analysis phase of system development specifies what a system should do and how it should behave. The products of OOA can be considered to be a class specification for the entire system.

A class specification describes the responsibilities of the objects of that class or system. These include providing service interfaces, maintaining state information and behaving in specified ways to service requests. A class can be responsible for providing services without having a direct interface for those services. This can be achieved by using internal components which do directly provide interfaces for those services.

Notations used for class specifications include IDL, class invariants, assertions on operations, class diagrams, object diagrams, state transition diagrams, user interface definitions, and descriptions of allowed resources, allowed time and space complexity and synchronisation contracts (see above).

The environmental method keeps an explicit separation between analysis and design phases. The environmental object model is not used for analysis. This is because it is a computational model, and the class specifications of analysis are declarative.

Analysis of the Diary System  The principle analysis classes of the Diary System are Diary Owner (uniquely associated with one Diary); User (a Diary Owner which is
not a Group); Group (a Diary Owner which may have other Diary Owners as members; the grouping relationship must be acyclic); Entry (an event held in one Diary. Attributes of an Entry describe what, where and when it occurs); Diary (a collection of Entries); and Timetable (a collection of Entries derived from a search of the Diary System).

The principle system services are shown in table 1. The people who use the Diary System should be able to start a user interface from any host in the local network and request system services using it.

### Table 1. Diary System events identified during analysis

<table>
<thead>
<tr>
<th>Event Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a New User’s Diary</td>
<td>A User without a Diary is named, and the system creates an empty Diary for that new Diary Owner. The new User is not initially in any Groups.</td>
</tr>
<tr>
<td>Create a New Group’s Diary</td>
<td>A new Group is named, and the system creates an empty Diary for that new Diary Owner. The new Group initially is in no Groups, and has no members.</td>
</tr>
<tr>
<td>Delete Diary and Owner</td>
<td>A Diary Owner is named, and the system deletes the owner’s Diary and all its Entries. The Diary Owner is deleted from the grouping relationship.</td>
</tr>
<tr>
<td>Add Entry to Diary</td>
<td>A Diary Owner is named, and a new Entry is supplied. The system adds the Entry to that Owner’s Diary.</td>
</tr>
<tr>
<td>Remove Entry from Diary</td>
<td>An Entry within a Diary is identified, and the system deletes that Entry.</td>
</tr>
<tr>
<td>Get Entry from Diary</td>
<td>An Entry within a Diary is identified, and that Entry is returned to the client.</td>
</tr>
<tr>
<td>Query Timetable</td>
<td>A client supplies some search parameters and the system returns all the Entries which match the search criteria. Query parameters may include a match against the Diary Owners, the Entry attributes and whether or not the Groups of the matching Owners (directly or indirectly) are to be taken into account.</td>
</tr>
<tr>
<td>Add Diary Owner to Group</td>
<td>A Diary Owner and a Group are named and the system makes the Owner a member of that Group. Placing an Owner in a Group must not create a cycle in the grouping relationship.</td>
</tr>
<tr>
<td>Remove Diary Owner from Group</td>
<td>A Group and a Diary Owner which is a member of that Group are named and the system removes that member from that Group. Removing an Owner from a Group does not change the Diaries of the Owner or the Group.</td>
</tr>
<tr>
<td>Query a Group’s Members</td>
<td>A Group is named and the system returns the Diary Owners which are members of that Group.</td>
</tr>
<tr>
<td>Query the Groups a Diary Owner is a Member of</td>
<td>A Diary Owner is named and the system replies with the Groups that Owner is a member of. The client may request that the Groups that the Owner’s Groups are in (and so on) are also returned.</td>
</tr>
</tbody>
</table>

### Logical Design

The design phase of system development identifies a collection of classes and defines them to a level of detail such that programming the classes is relatively straightforward. A class definition consists of a class interface and a class implementation.

A class interface has similarities with a class specification, but exists to define what a (computational) class does, rather than to declare what a class should do. It should document design choices made from the choice of designs which would match the specification. It should contain this additional information in order to allow the class to be understood and (re)used without including information about its implementation. An interface may contain or refer to a specification. Each class interface needs to contain an IDL description even if the class will not end up as a CORBA server (available for remote invocation).

A class implementation defines how a class does what it claims to do in its interface. This information includes the state information held by the class’s objects, and the algorithms used for the operations. Additional class, object and other diagrams or code may be needed to show the class’s implementation relationships, and to illustrate the algorithms.

The analysis model (system class specification) is not used as an initial design, as is done in many object-oriented methods. While there is beneficial continuity in using object models for analysis and design, this continuity cannot be seamless, because of the different meanings of object models for analysis and for design. Therefore, design has to be a problem solving phase, and cannot simply transform a system specification into an implementation. The beneficial continuity that can be gained exists because often the classes of the analysis models may give inspiration for a design architecture to ‘solve the problem.’

Design is split into logical and physical phases in order to isolate the difficulties introduced by distributed platforms from the difficulties of solving the problem posed by a system specification. The logical design phase identifies and defines an architecture of environmental classes which provide a solution to the functional requirements of the specification. The physical design phase transforms the logical containment hierarchy to complete an architecture which also satisfies the behavioural and physical requirements of the specification. Logical design includes designing for concurrency, and physical design includes designing for distribution.

This separation results in the final design being easier to understand (because of the separation of concerns between the logic of the solution and the support for distribution), the logical design becoming a unit of reuse (stable in the presence of change of the physical requirements), the physical design being maintainable and extensible, and increased continuity from specification through to coding.

**Top Down Logical Design** Logical design will usually proceed in both top-down and bottom-up manners. These are described separately, but in practice both sorts of design would be interleaved, and the class definitions yielded from each would be examined for consistency.

The first class which can be identified during logical design is the system object’s class. This might also be known as a main object or a root object. The system
object’s class has as its specification the system specification from the analysis phase.

The steps needed to define an environmental class from its specification are:

1. Define the Environment’s Class Interface from its Class Specification. An environment object (or object system) provides services to external clients but can do so without the operations being directly included in the class’s service interface. The specification (especially a system specification) may or may not define whether the services the object provides must be supported directly by an instance of that class, or can be supported by an instance which it contains as part of its implementation. This distinction must be clarified. Those operations that the object directly supports should be defined as an IDL interface for the class; a synchronisation contract for those operations also needs to be defined.

2. Identify Components. The component objects will cooperate to fulfil the responsibilities of the environment object. They are chosen to maintain the state information needed by the implementation, to participate in the environment’s behaviours, and to provide the services that the environment object is responsible for but is not directly supporting in its interface.

3. Specify Components’ Classes. Having identified components, their classes must be specified. If the operations of an environment invoke operations of its components, then these operations need to be in the component’s class interface. The specifications need to include synchronisation contracts.

4. Define the Environment’s Operations. Defining the environment’s behaviours in response to direct service requests completes its implementation. The algorithms will include invocations of the operations of the components. The specification of the components’ operations should be defined in step 3. If the components require support operations to implement their environment, these need to be included in the environment’s IDL interface (or as a separate support interface).

Consideration of concurrency is included in all of the steps. As components and operations of the environment are defined, an environmental synchronisation scheme can be chosen which will implement the environment’s synchronisation contract. If the synchronisation chosen is a ‘wrapper’ scheme, with granularity at the level of operation invocation, the conditions under which operations can proceed can be included in the environment’s interface (in step 1). If the environmental scheme chosen requires a transaction-style scheduler, with the operations requesting locks from the scheduler, then steps 2 and 3 include specifying a scheduler. Step 3 also defines the synchronisation contracts of components. If in step 4 the environment’s operations on its components, or the components’ operations on each other, are seen to violate the synchronisation contract of a component, or might cause deadlock, then components and mechanisms to ensure safety need to be introduced.

The steps of logical design form a micro cycle, which will be re-iterated until components have been identified and specified, and operations have been defined to implement all of the environment’s responsibilities. The environment’s implementation must satisfy the constraints of the environmental model: that no component will send a reference to one of its components, and that the environment does not pass a reference to one of its components out to a client, and that the environment does not pass a reference it holds to a peer or its environment to a component.

Classes will be identified for each of the object types which are named as parameters and return values in the interface definition. These object types might not be components of the class being designed, but might be passed around from environment to component, used as local variables for operations, or thrown/caught as exceptions. These classes should be fed into the bottom up design process.

Top down design progresses from the system object’s class down through the classes of the components of the system object. It must go on at least until object’s have been identified to directly provide all of the system’s services, and mechanisms to implement those services have been defined.

Top Down Logical Design of the Diary System The first stage of logical design of the diary system identified that it will contain User Interface objects, which are clients of a Diary Service object and a Grouping Service object (figure 2). IDL interfaces were written for the Diary and Grouping Services (figures 3 and 4). The Grouping Service is not coupled to the Diary Service and could be reused outside the diary system, or promoted to an operating system service. Both of these service objects have concurrent interfaces, but the scheduling is not specified at the granularity of operation invocations. User Interfaces are dynamically created by (human) users. Each one can be a sequential object. The Diary System object does not need to synchronise requests from User Interface components to Diary or Group Service components.

The Diary Service is shown in figure 5 as an environment for Diaries. The Grouping Service is shown in figure 6 as an environment for a graph of Nodes. The behaviours of the Diary and Grouping Services are not shown but are mostly straightforward. The addMemberToGroup operation of the Grouping Service needs to check that the invariant property that there are no cycles in the graph.

Synchronisation The Diary Service will receive concurrent operation invocations and the design should allow those which don’t conflict to be processed concurrently, whilst synchronising those that conflict (for
example, two users should be able to edit different Diaries concurrently). To choose a synchronisation strategy it is helpful to examine the data access requirements of the operations of the Diary Service. Create Diary and Destroy Diary change the Map. Add Entry and Remove Entry read from the Map and change the identified Diary. Get Entry reads from the Map and the identified Diary. Query reads from the Map and from one or more Diaries, possibly all of them.

Considering each component of the Diary Service to be a resource for which a lock must be obtained by a Diary Service operation before access is allowed, it can be observed that if the operations follow a transaction-style two-phase locking strategy (which is serialisable), then there will be no deadlocks. This is because every operation acquires its lock on the Map before subsequent locks; and every write operation accesses at most one Diary; so there will never be a cycle of operations waiting for resources the others are holding. Further consideration of this issue and synchronisation of the Grouping Service is given during physical design.

**Bottom Up Logical Design** Bottom up design also aims to define the interface and implementation of classes. The initial candidate classes for bottom up design are entities from the problem domain, which will become the data objects of the system. Such objects will be manipulated by the controlling environment classes in the system structure.

Many OOAD texts give advice about choosing low level abstractions. In the Diary System, DiaryOwner, Diary, Entry, EntryId, Date and Time are some from the vocabulary of the problem domain. OwnerPattern, EntryPattern and Node were identified as parameters to a Query during top down design. Some IDL interfaces are given in figure 7.

**Physical Design**

Physical design transforms the classes defined by logical design to take into account physical requirements of the system. The most important physical consideration is the (possible) distributed nature of the hardware platform.

The essence of the transformation of an instance structure is to map the service interfaces provided by each logical environment in a logical system structure to a collection of servers running on a distributed platform which will cooperate to provide a physical, distributed service.
It is important to note that the logical containment associations of the environment model do not constrain the components of an environment to be on the same node as the environment. The environment and its components might have distributed implementations, or the environment might run on one node with its components on different nodes. The communication links between instances are still constrained by the (logical) containment of an environmental object system.

Top Down Physical Design  Transformation of a logical design starts from the top of the system structure and works down. The process is described for a general environment object.
1. **Choose Service Access Points.** The logical environment has an interface of operations it directly supports. It must be decided whether this service interface can be accessed at a single server object at one node of the system, or can be accessed at any one of many server objects at different sites in the system. For the system object, this choice may be specified in the system specification. For internal components, the choice will depend on the requirements of their internal and external clients. Note that this choice does not necessarily affect clients of the service. External clients have a choice of binding mechanism of binding to a specific service access point on a particular node, or using a naming and location service to bind to service provider identified by its service name. These mechanisms are both useful alternatives whether the service is provided at one access point or many.
2. **Choose a Component Configuration.** The logical environment was designed to contain component objects which would cooperate to implement it. It must be decided where these components will exist and provide their services within a distributed system. The service access points of a component are not in general constrained to be co-located with the service access points of their environment. If the component is directly providing a system service then the location of access points may be specified in the system specification. A collection of components may be sited on different nodes to gain concurrency.

There are three principle possibilities for a component’s service access points. The component may be located as a single service object on one node (perhaps dynamically allocated), it may provide a distributed service to its clients, with many access points, or it may be replicated at many nodes. The implication of the choice of replicating a component is that a mechanism must be introduced within the component’s environment to manage the consistency of the replicas. The problem of managing replicas is coupled with that of distributed concurrency control and is considered further below.
3. **Transform the Behaviours of the Object.** The logical response of the environment object to service requests needs to be transformed to take account of the chosen service access points of the object and its components, and of the (distributed) environmental synchronisation of the object. Object creation may be transformed to remote object creation. Object links can be implemented as remote references if the server is remote from the client, or as pointers if the server is in the same address space as the client.

In principle, the synchronisation contract of a class will not change as a class is carried over from logical design to physical design. For example, a combination of a sequential interface with distributed service access points requires the distributed clients to cooperate so that no more than one service is requested at any access point at a time. If this cooperation would be inefficient, then either the logical choice of a sequential interface, or the physical choice of distributed service access points should be re-examined.

The implementation of synchronisation within an environment is transformed in step 3, according to the choices of service access points made in steps 1 and 2. The principle of environmental synchronisation remains valid, that the responsibility for ensuring liveness, and data consistency and safety of a distributed implementation remains with the implementation of the environment.

If an environment’s service is accessed at a single access point then concurrent programming techniques can be used to schedule its operations. If the environment has distributed service access points, but its state components are centralised, then responsibility for scheduling may be delegated to a component located with the state component, to avoid the need for a distributed scheduler. A distributed environment with replicated state will need to combine a distributed scheduling algorithm with replica management protocols. It may be necessary to prototype some alternatives before a suitable trade-off between reliability, service availability, concurrency and complexity of implementation is reached.

Physical Design of the Diary System  These steps were applied to the diary system object. The system object (figure 2) offers no services and so has no methods. However, it does have one remaining direct responsibility, which is to construct its components and start up their services. Its implementation is achieved as an operating system shell script, which starts the Object Request Broker services (if necessary), registers the diary and group services with an implementation repository, and starts the services.

The Diary Service object can be considered the root of a new system, offering services to dynamic User Interface (and other) clients. As a distributed service, it should provide a number of service access points. The Diary Service object is therefore transformed into a number of CORBA Diary Server objects. These have the same IDL interface as the Diary Service object (figure 3), and will run on different hosts in a networked system, cooperating to provide consistent views of the system state. Clients will be able to bind to a named Diary Server, or bind via a naming or location service.

The components of the Diary Server objects are Diaries, and a map to be able to identify a Diary given its Owner’s name. It would be possible for each Diary to be able to offer a distributed service to its Diary Server environment, so that the server could simply forward requests to a local Diary access point. However, considerations of concurrency control (see below) and granularity (a Diary is not a complex enough an object to be worth a distributed implementation) suggest that each Diary Server should keep local copies of all the Diaries (which will not
be CORBA servers) and explicitly manage this state replication.

The operations of the Diary server which just read the state of the Map and Diaries (getEntry, query) can read local copies, but the operations which change the Map (createDiary, destroyDiary) or a Diary (addEntry, removeEntry) need to propagate those changes to all copies.

The strategy for replica management depends on the system support available, ranging from an existing replica manager service through to reliable communications protocols. The Diary System development did not have these available, so replication management was implemented by adding Transmitter-Receiver CORBA servers to each Diary Server (to allow explicit broadcast of state changes), and Lock Servers were added to implement a distributed Lock Service. The two-phase locking strategy identified during logical design can be used in a distributed implementation, as long as a distributed locking algorithm is used. Each Lock object in each Lock Server implements a distributed reader/writers protocol in cooperation with the other Lock objects at other sites. A suitable protocol was devised.

Figure 8 shows a Diary Server with its Transmitter-Receiver and Lock Server components. A minimal implementation class of a Diary Server is shown in figure 9.

The Grouping Service object was also transformed to a collection of distributed Grouping Servers, with replicated Node and Map components.

Because of an assumption that changes to the grouping relationship would be relatively infrequent, the increased concurrency of a transaction-style scheduler was not chosen. Instead, a distributed form of ‘wrapper’ synchronisation was used which allowed read requests to each Grouping Server to query the local state, but executed all write requests arriving at all Grouping Servers in the same global order, thus ensuring consistency. Each write service request to a Grouping Server is timestamped and broadcast to every other Grouping Server. A distributed queue algorithm is used to apply the operations in the same order at every site.

**Bottom Up Physical Design** Bottom up physical design refines the data abstractions from logical design, and develops others identified during top down physical design. These low level abstractions typically do not provide distributed services, so their development is more conventional than that of the high level system and subsystem objects providing distributed services.

**Implementation**

Physical design should have refined the design classes until they are directly implementable on the development platform chosen. During the implementation phase the classes are coded and tested against their specification.

**Implementation of the Diary System** The Diary Servers and Grouping Servers were implemented with Sun ‘CC’ C++ compiler and Iona’s OrbixMT CORBA implementation, on a network of hosts running the Solaris operating system.

The User Interfaces were implemented as Tcl/Tk scripts, where Tcl was extended with procedures which were clients of the CORBA servers.

```c++
class DiaryServer {
public:
  DiaryServer( ... );
  virtual void createDiary( ... ) { ... };
  virtual void destroyDiary( ... ) { ... };
  virtual void addEntry( const String& owner,
                        const Entry* e ) {
    lockServer->readlock( lockIdOfMap );
    lockServer->setWriteLock( lockIdOfDiaryOfOwner);
    Diary* d = map->lookup( owner );
    EntryId eId = d->add( e );
    txrx->broadcastAddEntry( owner, e);
    lockServer->unlock( lockIdOfDiaryOfOwner);
    lockServer->unlock( lockIdOfMap );
  };
  virtual void removeEntry( ... ) { ... };
  virtual Entry* getEntry( ... ) { ... };
  virtual EntryIdSequence query( const OwnerPattern& op,
                                const EntryPattern& ep,
                                CORBA::boolean closure ) {
    lockServer->readlock( lockIdOfMap );
    lockServer->readlock( ... all diaries of matching owners, and their groups’ diaries... );
    ... query all diaries ...
    unlock( ... all locks held ... );
    return( ... all matching EntryIds ... );
  };
private:
  LockServer* lockServer;
  Map< Owner, Diary* > *map;
  Transmitter-Receiver* txrx;
  GroupServer* gap; // link to a peer
};
```

**Figure 9. Diary Server implementation class**
4. Discussion and Conclusions

Related Work The Regis/Darwin system [3] combines a process-parallel, component-oriented language (Regis) with a separate and explicit configuration language (Darwin). Only the leaves of the structure tree are computational components. This is by contrast with the environmental model, where every component in the tree is a full object, with run-time state, behaviour and identity. This gives the advantage that environments are full abstractions and can be understood by examining their behaviours and components, without descending the structure tree to the leaves. They are better able to act as coordinators and managers for multiple components. They can be transformed to implement distributed services with more than one access point.

Ensembles provide a framework for decomposition of an object-oriented analysis [8]. They represent a cluster of less abstract entities. Unlike class categories, the ensemble hides its constituents, and acts as a gateway for interaction. The ensemble can act as a manager, imposing constraints. Every object, including the ensemble, is considered to have a single thread of control. This is claimed to mean that the ensemble is encapsulating parallelism. A similar approach is not advocated for design or for structuring of an implementation.

This model is close enough to the environmental model to suggest that there would be continuity benefits to using it during an analysis phase preceding design.

HOOD is a design method specialised for real-time space applications in Ada which creates a hierarchic component structure [9] from the top down. The HOOD object model is essentially that of Ada tasks, where objects have a distinguished ‘body’ thread controlling scheduling of operations. HOOD’s component model is object based, but with a class concept which allows Ada genericity. The object model is unusual as parent objects are not allowed to have their own operations, but instead behaviours are delegated to ‘operation’ components.

The main differences between HOOD and environmental design are that HOOD doesn’t assume an object-oriented specification (and so includes domain modelling activities). HOOD doesn’t combine top down structural design with bottom up class design, and doesn’t separate logical from physical design. When designing for distribution, HOOD names its top level components virtual nodes, makes an allocation of virtual nodes to processors. Links between virtual nodes may then be inter-processor connections. This does not promote development of distributed services.

Conclusions This paper has introduced the environmental object model for system structuring based on containment. This allows complex abstractions to be built which act as managers, coordinators and controllers. The constraints on links between objects keep the components within an environment private, and prevent objects which are distant in the system structure from violating invariant properties of the environment’s component set.

The model’s constraints on structuring and links can be statically checked using language or tool support. A language with type annotations to enforce a structural hierarchy is described in [5].

The concurrency model keeps activities unrelated to objects, which provides better modelling fidelity. Synchronisation contracts keep the specification of objects’ behaviour in the presence of concurrency separate from the implementation of concurrency control. Environmental synchronisation allows use of concurrency control solutions which match the problem. This allows scheduling algorithms (which are essential for highly concurrent implementations) to be integrated into object solutions. The choice of a scheduling algorithm often depends on the patterns of access of operations on data. The constraints on links within the environmental model allows this to be analysed, as was shown for the Diary Service’s operations on its Diaries.

The environmental design method takes advantage of these system structuring and concurrency models. For the Diary System, the logical top down process identified concurrent Diary and Grouping Services which are complex concurrent abstractions. The Diary Service could safely control access to its state components, which are sequential Diary objects, which are in turn environments for Entries. The physical design process considers the requirements for service availability and location of system services to place CORBA servers at nodes within a network which will cooperate to provide each logical object’s services.