CHAPTER -8

DISTRIBUTED OBJECTS & COMPONENTS

- This chapter examines two of the most important programming abstractions, namely distributed objects and components, and also examines associated middleware platforms including CORBA, Enterprise JavaBeans and Fractal.

- **Distributed object middleware •** The key characteristic of distributed objects is that they allow you to adopt an object-oriented programming model for the development of distributed systems and, through this, hide the underlying complexity of distributed programming.
  - In this approach, communicating entities are represented by objects. Objects communicate mainly using remote method invocation, but also possibly using an alternative communication paradigm (such as distributed events).

Benefits of Distributed object middleware

- The *encapsulation* inherent in object-based solutions is well suited to distributed programming.
- The related property of *data abstraction* provides a clean separation between the specification of an object and its implementation, allowing programmers to deal solely in terms of interfaces and not be concerned with implementation details such as the programming language and operating system used.
- This approach also lends itself to more *dynamic* and *extensible* solutions, for example by enabling the introduction of new objects or the replacement of one object with another (compatible) object.

Component-based middleware

Component-based solutions have been developed to overcome a number of limitations that have been observed for application developers working with distribute :

- ***Implicit dependencies****:* Object interfaces do not describe what the implementation of an object depends on, making object-based systems difficult to develop (especially for third-party developers) and subsequently manage.

- ***Programming complexity:*** Programming distributed object middleware leads to a need to master many low-level details associated with middleware implementations.

- ***Lack of separation of distribution concerns****:* Application developers are obliged to consider details of concerns such as security, failure handling and concurrency, which are largely similar from one application to another.

- ***No support for deployment:*** Object-based middleware provides little or no support for the deployment of (potentially complex) configurations of objects.

8.2 Distributed objects

Middleware based on distributed objects is designed to provide a programming model based on object-oriented principles and therefore to bring the benefits of the objectoriented approach to distributed programming. Distributed objects are a natural evolution from three strands of activity:

- In ***distributed systems***, earlier middleware was based on the client-server model and there was a desire for more sophisticated programming abstractions.
- In ***programming languages***, earlier work in object-oriented languages such as Simula-67 and Smalltalk led to the emergence of more mainstream and heavily used programming languages such as Java and C++ (languages used extensively in distributed systems).
- In ***software engineering***, significant progress was made in the development of object-oriented design methods, leading to the emergence of the Unified Modelling Language (UML) as an industrial-standard notation for specifying (potentially distributed) object-oriented software systems

Figure 8.1 Distributed objects

Objects	*Distributed objects*	*Description of distributed object*
Object references	Remote object references	Globally unique reference for a distributed object; may be passed as a parameter.
Interfaces	Remote interfaces	Provides an abstract specification of the methods that can be invoked on the remote object; specified using an interface definition language (IDL).
Actions	Distributed actions	Initiated by a method invocation, potentially resulting in invocation chains; remote invocations use RMI.
Exceptions	Distributed exceptions	Additional exceptions generated from the distributed nature of the system, including message loss or process failure.
Garbage collection	Distributed garbage collection	Extended scheme to ensure that an object will continue to exist if at least one object reference or remote object reference exists for that object, otherwise, it should be removed. Requires a distributed garbage collection algorithm.

Differences between objects and distributed objects

- ***Class*** is a fundamental concept in object-oriented languages but does not feature so prominently in distributed object middleware.
- As noted in the CORBA casestudy, it is difficult to agree upon a common interpretation of class in a heterogeneous environment where multiple languages coexist.
- In the objectoriented world more generally, class has several interpretations, including the description of the behaviour associated with a group of objects (the template used to create an object from the class), the place to go to instantiate an object with a given behaviour (the associated factory) or even the group of objects that adhere to that behaviour.

- The style of ***inheritance*** is significantly different from that offered in most objectoriented languages.

- In particular, distributed object middleware offers *interface inheritance*, which is a relationship between interfaces whereby the new interface inherits the method signatures of the original interface and can add extra ones.

- In contrast, object-oriented languages such as Smalltalk offer *implementation inheritance* as a relationship between implementations, whereby the new class (in this case) inherits the implementation (and hence behaviour) of the original class and can add extra behaviour.

- Implementation inheritance is much more difficult to implement, particularly in distributed systems, due to the need to resolve the correct executable behaviour at runtime.

- Consider, for example, the level of heterogeneity that may exist in a distributed system, together with the need to implement highly scalable solutions.

Additional Functionality of Distributed object middleware

- ***Inter-object communication*****:** A distributed object middleware framework must offer one or more mechanisms for objects to communicate in the distributed environment. This is normally provided by remote method invocation, although distributed object middleware often supplements this with other communications paradigms (for example, indirect approaches such as distributed events).

- ***Lifecycle management*****:** Lifecycle management is concerned with the creation, migration and deletion of objects, with each step having to deal with the distributed nature of the underlying environment.

- ***Activation and deactivation*****:** In non-distributed implementations, it can often be assumed that objects are active all the time while the process that contains them runs.

- In distributed systems, however, this cannot be assumed as the numbers of objects may be very large, and hence it would be wasteful of resources to have all objects available at any time.

- In addition, nodes hosting objects may be unavailable for periods of time. Activation is the process of making an object active in the distributed environment by providing the necessary resources for it to process incoming invocations – effectively, locating the object in virtual memory and giving it the necessary threads to execute. Deactivation is then the opposite process, rendering an object temporarily unable to process invocations.

- ***Persistence***: Objects typically have state, and it is important to maintain this state across possible cycles of activation and deactivation and indeed system failures. Distributed object middleware must therefore offer persistency management for stateful objects.

- ***Additional services***: A comprehensive distributed object middleware framework must also provide support for the range of distributed system services including naming, security and transaction services

8.3 Case study: CORBA

- The Object Management Group (OMG) was formed in 1989 with a view to encouraging the adoption of distributed object systems in order to gain the benefits of object-oriented programming for software development and to make use of distributed systems, which were becoming widespread.

- OMG introduced a metaphor, the *object request broker* (ORB), whose role is to help a client to invoke a method on an object.

- This role involves locating the object, activating the object if necessary and then communicating the client’s request to the object, which carries it out and replies.

The main components of CORBA's language-independent RMI framework are the following:

- an interface definition language known as IDL
- an architecture
- an external data representation, called CDR, it also defines specific formats for the messages in a request-reply protocol and messages for enquiring about the location of an object, for cancelling requests and for reporting errors;
- a standard form for remote object references

The CORBA architecture also allows for CORBA services – a set of generic services that are useful for distributed applications.

8.3.1 CORBA RMI

- Programming in a multi-language RMI system such as CORBA RMI requires more of the programmer than programming in a single-language RMI system such as Java RMI.

- In CORBA, proxies are generated in the client language and skeletons in the server language.

- **CORBA's object model:** Here, clients are not necessarily objects. a client can be any program that sends request messages to remote objects and receives replies. The term *CORBA object* is used to refer to remote objects. Thus, a CORBA object implements an IDL interface, has a remote object reference and is able to respond to invocations of methods in its IDLinterface. The class concept does not exist in CORBA. Therefore classes cannot be defined in CORBA IDL, which means that instances of classes cannot be passed as arguments. However, data structures of various types and of arbitrary complexity can be passed as arguments.

IDL interfaces *Shape* and *ShapeList*

CORBA IDL - A CORBA IDL interface specifies a name and a set of methods that clients can request. Figure 8.2 shows two interfaces named *Shape* (line 3) and *ShapeList* (line 5)*,* which are IDL versions of the interfaces.These are preceded by definitions of two *structs*, which are used as parameter types in defining the methods.

Note in particular that *GraphicalObject* is defined as *struct*, whereas it was a class in the Java RMI example. A component whose type is *struct* has a set of fields containing values of various types, like the instance variables of an object,but it has no methods.

CORBA IDL has the same lexical rules as C++ but has additional keywords to support distribution, for example, *interface*, *any*, *attribute*, *in*, *out*, *inout*, *readonly* and *raises*.

```
struct Rectangle{                                                              1
      long width;
      long height;
      long x;
      long y;
};
struct GraphicalObject {                                                       2
      string type;
      Rectangle enclosing;
      boolean isFilled;
};
interface Shape {                                                              3
      long getVersion();
      GraphicalObject getAllState();    // returns state of the GraphicalObject
};
typedef sequence <Shape, 100> All;                                             4
interface ShapeList {                                                          5
      exception FullException{ };                                              6
      Shape newShape(in GraphicalObject g) raises (FullException);             7
      All allShapes();              // returns sequence of remote object references 8
      long getVersion();
};
```

- **IDL modules:** The module construct allows interfaces and other IDL type definitions to be grouped in logical units. A *module* defines a naming scope, which prevents names defined within a module from clashing with names defined outside it. For example, the definitions of the interfaces *Shape* and *ShapeList* could belong to a module called *Whiteboard*, as shown in Figure 8.3.

Figure 8.3 IDL module *Whiteboard*

```
module Whiteboard {
    struct Rectangle{
    ...};
    struct GraphicalObject {
    ...};
    interface Shape {
    ...};
    typedef sequence <Shape, 100> All;
    interface ShapeList {
    ...};
};
```

IDL interfaces: As we have seen, an IDL interface describes the methods that are available in CORBA objects that implement that interface. Clients of a CORBA object may be developed just from the knowledge of its IDL interface.

IDL methods: The general form of a method signature is given below. where the expressions in square brackets are optional.

[oneway] <return_type> <method_name> (parameter1,..., parameterL)
[raises (except1,..., exceptN)] [context (name1,..., nameM)];

The parameters are labelled as ***in, out*** or ***inout***, where the value of an *in* parameter is passed from the client to the invoked CORBA object and the value of an *out* *p*arameter is passed back from the invoked CORBA object to the client.
Parameters labelled as *inout* are seldom used, but they indicate that the parameter value may be passed in both directions.

IDL types: IDL supports 15 primitive types, which include *short* (16-bit), *long* (32-bit), *unsigned short*, *unsigned long*, *float* (32-bit), *double* (64-bit), *char*, *boolean* (TRUE, FALSE), *octet* (8-bit) and *any* (which can represent any primitive or constructed type).

Constants of most of the primitive types and constant strings may be declared using the *const* keyword.

Figure 8.4 IDL constructed types

Type	*Examples*	*Use*

sequence	*typedef sequence <Shape, 100> All;*
	typedef sequence <Shape> All;
	Bounded and unbounded sequences of *Shapes*
		Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.

string	*string name;*
	typedef string<8> SmallString;
	Unbounded and bounded sequences of characters
		Defines a sequence of characters, terminated by the null character. An upper bound on the length may be specified.

array	*typedef octet uniqueId[12];*
	typedef GraphicalObject GO[10][8];
		Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.

record	*struct GraphicalObject {*
	  string type;
	  Rectangle enclosing;
	  boolean isFilled;
	};
		Defines a type for a record containing a group of related entities.

enumerated	*enum Rand* *(Exp, Number, Name);*	The enumerated type in IDL maps a type name onto a small set of integer values.

union	*union Exp switch (Rand) {*
	  case Exp: string vote;
	  case Number: long n;
	  case Name: string s;
	};
		The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterized by an *enum*, which specifies which member is in use.

Attributes: IDL interfaces can have attributes as well as methods. Attributes are like public class fields in Java. Attributes may be defined as *readonly* where appropriate. The attributes are private to CORBA objects, but for each attribute declared, a pair of accessor methods is generated automatically by the IDL compiler: one to retrieve the value of the attribute and the other to set it. For *readonly* attributes, only the getter method is provided.

Inheritance: IDL interfaces may be extended through interface inheritance. A value of an extended type is valid as the value of a parameter or result of the parent type. For example, the type *B* is valid as the value of a parameter or result of the type *A*. In addition, an IDL interface may extend more than one interface. For example,
interface *Z* here extends both *B* and *C*:
interface A { };
interface B: A{ };
interface C {};
interface Z : B, C {};

IDL type identifiers: type identifiers are generated by the IDL compiler for each type in an IDL interface.

IDL pragma directives: These allow additional, non-IDL properties to be specified for components in an IDL interface. These properties include, for example, specifying that an interface will be used only locally, or supplying the value of an interface repository ID.

CORBA language mappings

- The mapping from the types in IDL to types in a given programming language is quite straightforward. For example, in Java the primitive types in IDL are mapped to the corresponding primitive types in that language. *Structs*, *enums,* and *unions* are mapped to Java classes; sequences and arrays in IDL are mapped to arrays in Java.

- An IDL exception is mapped to a Java class that provides instance variables for the fields of the exception and constructors. The mappings in C++ are similarly straightforward.

- However, some difficulties arise with mapping the parameter-passing semantics of IDL onto those of Java. In particular, IDL allows methods to return several separate values via output parameters, whereas Java can have only a single result. The *Holder* classes are provided to overcome this difference, but this requires the programmer to make use of them, which is not altogether straightforward.

- Although C++ implementations of CORBA can handle *out* and *inout* parameters quite naturally, C++ programmers suffer from a different set of problems with parameters, related to storage management. These difficulties arise when object references and variable-length entities such as strings or sequences are passed as arguments.

Asynchronous RMI

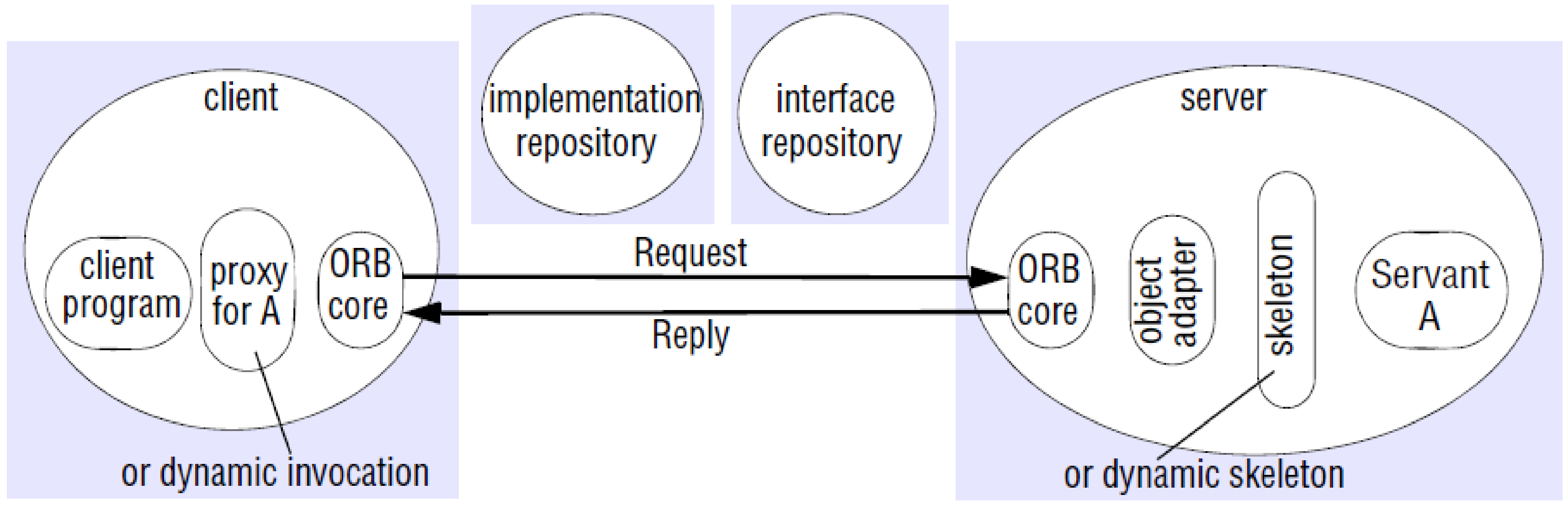
- CORBA supports a form of asynchronous RMI that allows clients to make non-blocking invocation requests on CORBA objects [OMG 2004e]. It is intended to be implemented in the client. Therefore a server is generally unaware of whether it is invoked synchronously or asynchronously.

- Asynchronous RMI adds two new variants to the invocation semantics of RMIs: *callback*, in which a client uses an extra parameter to pass a reference to a callback with each invocation so that the server can call back with the results; *polling*, in which the server returns a *valuetype* object that can be used to poll or wait for the reply.

- The architecture of asynchronous RMI allows an intermediate agent to be deployed to make sure that the request is carried out and, if necessary, to store the reply. Thus it is appropriate for use in environments where clients may become temporarily disconnected – such as, for example, a client using a laptop on a train

8.3.2 The architecture of CORBA

Figure 8.5 The main components of the CORBA architecture



CORBA architecture contains three essential components: the object adapter, the implementation repository and the interface repository.

CORBA provides for both static and dynamic invocations. Static invocations are used when the remote interface of the CORBA object is known at compile time, enabling client stubs and server skeletons to be used. If the remote interface is not known at compile time, dynamic invocation must be used.

Components of the CORBA architecture

- **ORB core •** The role of the ORB core includes all the functionality of the communication module In addition, an ORB core provides an interface that includes the following:

✓ operations enabling it to be started and stopped;

✓ operations to convert between remote object references and strings;

✓ operations to provide argument lists for requests using dynamic invocation

- **Object adapter •** The role of an *object adapter* is to bridge the gap between CORBA objects with IDL interfaces and the programming language interfaces of the corresponding servant classes. An object adapter has the following tasks:

✓ It creates remote object references for CORBA objects .

✓ It dispatches each RMI via a skeleton to the appropriate servant.

✓ It activates and deactivates servants.

- An object adapter gives each CORBA object a unique *object name*, which forms part of its remote object reference. The same name is used each time an object is activated. The object name may be specified by the application program or generated by the object adapter.
- Each CORBA object is registered with its object adapter, which keeps a remote object table that maps the names of CORBA objects to their servants. Each object adapter also has its own name, which forms part of the remote object references of all of the CORBA objects it manages. This name may either be specified by the application program or generated automatically

- **Portable object adapter -** The CORBA standard for object adapters is called the Portable Object Adapter (POA). It is called portable because it allows applications and servants to be run on ORBs produced by different developers [Vinoski 1998]. This is achieved by means of the standardization of the skeleton classes and of the interactions between the POA and the servants.
- The POA supports CORBA objects with two different sorts of lifetimes:
- ❑ those whose lifetimes are restricted to that of the process in which their servants are instantiated;
- ❑ those whose lifetimes can span the instantiations of servants in multiple processes.
- The POA allows CORBA objects to be instantiated transparently; and in addition it separates the creation of CORBA objects from the creation of th, *servants* that implement those objects. Server applications such as databases with large numbers of CORBA objects can create servants on demand, only when the objects are accessed. In this case, they may use database keys for the object names, or they may use a single servant to support all of these objects.
- Implementations of CORBA provide interfaces to the functionality of the POA and the ORB core through *pseudo-objects,* given this name because they cannot be used like regular CORBA objects; for example, they cannot be passed as arguments in RMIs. They do, though, have IDL interfaces and are implemented as libraries. The POA pseudo-object includes, for example, one method for activating a *POAmanager* and another method, *servant_to_reference*, for registering a CORBA object;

- **Skeletons-** Skeleton classes are generated in the language of the server by an IDL compiler. Remote method invocations are dispatched via the appropriate skeleton to a particular servant, and the skeleton unmarshals the arguments in request messages and marshals exceptions and results in reply messages.

- **Client stubs/proxies -**These are in the client language. The class of a proxy (for objectoriented languages) or a set of stub procedures (for procedural languages) is generated from an IDL interface by an IDL compiler for the client language. As before, the client stubs/proxies marshal the arguments in invocation requests and unmarshal exceptions and results in replies.

- **Implementation repository-** An implementation repository is responsible for activating registered servers on demand and for locating servers that are currently running. The object adapter name is used to refer to servers when registering and activating them.

- **Interface repository-The** role of the interface repository is to provide information about registered IDL interfaces to clients and servers that require it. For an interface of a given type it can supply the names of the methods and, for each method, the names and types of the arguments and exceptions. Thus, the interface repository adds a facility for reflection to CORBA.

- **Dynamic invocation interface -** The dynamic invocation interface allows clients to make dynamic invocations on remote CORBA objects. It is used when it is not practical to employ proxies. The client can obtain from the interface repository the necessary information about the methods available for a given CORBA object. The client may use this information to construct an invocation with suitable arguments and send it to the server.

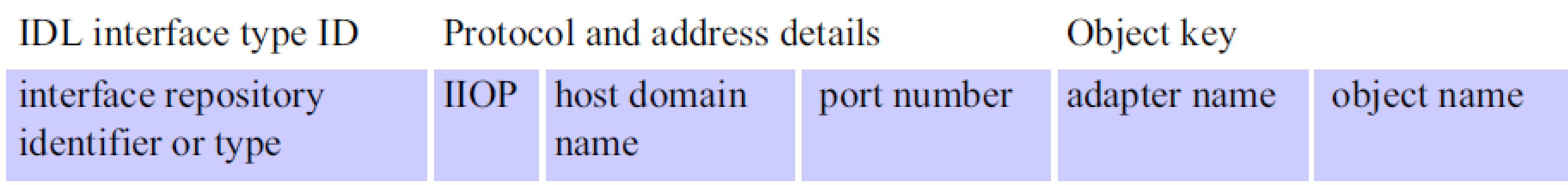
- **Dynamic skeletons -** If a server uses dynamic skeletons, then it can accept invocations on the interface of a CORBA object for which it has no skeleton. When a dynamic skeleton receives an invocation, it inspects the contents of the request to discover its target object, the method to be invoked and the arguments. It then invokes the target.

- **Legacy code -** The term *legacy code* refers to existing code that was not designed with distributed objects in mind. A piece of legacy code may be made into a CORBA object by defining an IDL interface for it and providing an implementation of an appropriate object adapter and the necessary skeletons.

8.3.3 CORBA remote object references

CORBA specifies a format for remote object references that is suitable for use whether or not the remote object is to be activated by an implementation repository. References using this format are called *interoperable object references* (IORs).

IOR format



IDL interface type ID	Protocol and address details			Object key	
interface repository identifier or type	IIOP	host domain name	port number	adapter name	object name

The first field of an IOR specifies the type identifier of the IDL interface of the CORBA object. Note that if the ORB has an interface repository, this type name is also the interface repository identifier of the IDL interface, which allows the IDL definition for the interface to be retrieved at runtime.

The second field specifies the transport protocol and the details required by that particular transport protocol to identify the server. In particular, the Internet Inter-ORB protocol (IIOP) uses TCP, in which the server address consists of a host domain name and a port number

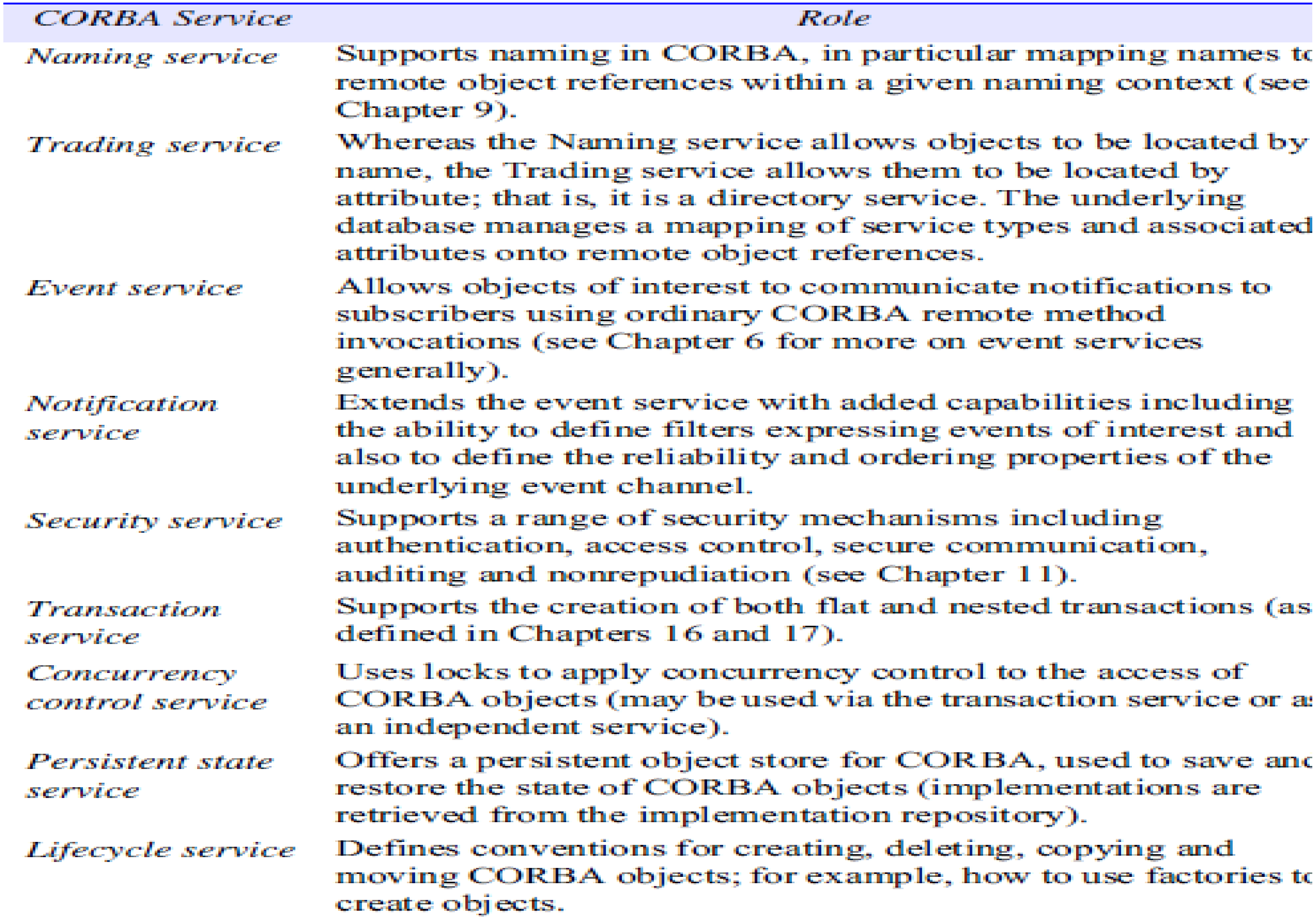
The third field is used by the ORB to identify a CORBA object. It consists of the name of an object adapter in the server and the object name of a CORBA object specified by the object adapter.

- *Transient IORs* for CORBA objects last only as long as the process that hosts those objects, whereas *persistent IORs* last between activations of the CORBA objects.
- A transient IOR contains the address details of the server hosting the CORBA object, whereas a persistent IOR contains the address details of the implementation repository with which it is registered. In both cases, the client ORB sends the request message to the server whose address details are given in the IOR.
- Here is how the IOR is used to locate the servant representing the CORBA object in the two cases:

Transient IORs**:** The server ORB core receives the request message containing the object adapter name and object name of the target. It uses the object adapter name to locate the object adapter, which uses the object name to locate the servant.

Persistent IORs**:** An implementation repository receives the request. It extracts the object adapter name from the IOR in the request. Provided that the object adapter name is in its table, it attempts if necessary to activate the CORBA object at the host address specified in its table.

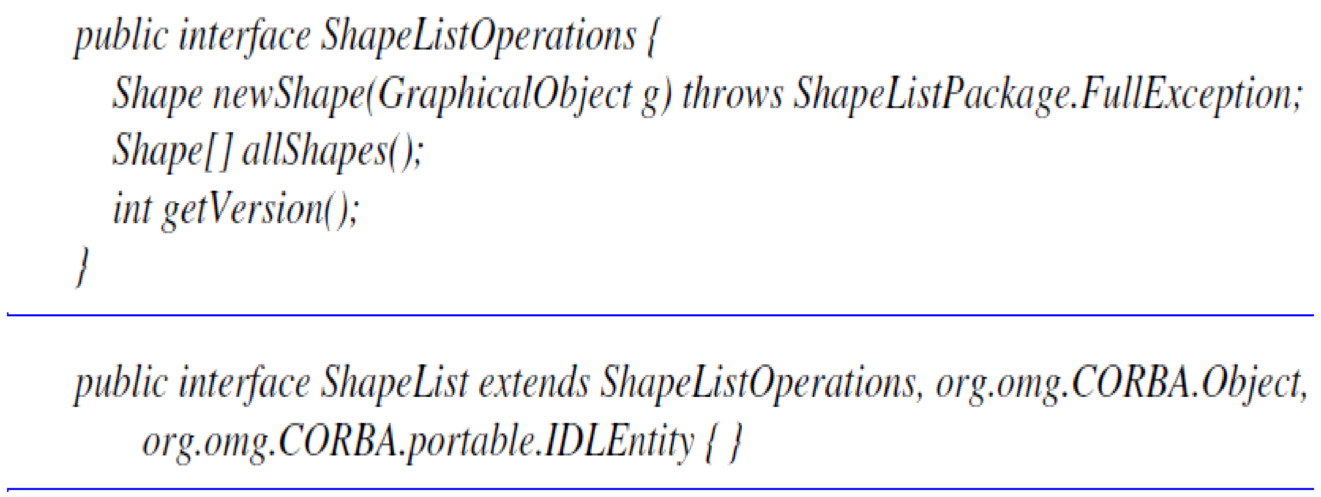
8.3.4 CORBA services



CORBA Service	*Role*
Naming service	Supports naming in CORBA, in particular mapping names t remote object references within a given naming context (see Chapter 9).
Trading service	Whereas the Naming service allows objects to be located by name, the Trading service allows them to be located by attribute; that is, it is a directory service. The underlying database manages a mapping of service types and associated attributes onto remote object references.
Event service	Allows objects of interest to communicate notifications to subscribers using ordinary CORBA remote method invocations (see Chapter 6 for more on event services generally).
Notification service	Extends the event service with added capabilities including the ability to define filters expressing events of interest and also to define the reliability and ordering properties of the underlying event channel.
Security service	Supports a range of security mechanisms including authentication, access control, secure communication, auditing and nonrepudiation (see Chapter 11).
Transaction service	Supports the creation of both flat and nested transactions (as defined in Chapters 16 and 17).
Concurrency control service	Uses locks to apply concurrency control to the access of CORBA objects (may be used via the transaction service or a an independent service).
Persistent state service	Offers a persistent object store for CORBA, used to save an restore the state of CORBA objects (implementations are retrieved from the implementation repository).
Lifecycle service	Defines conventions for creating, deleting, copying and moving CORBA objects; for example, how to use factories t create objects.

8.3.5 CORBA client and server example

Figure 8.7 Java interfaces generated by *idlj* from CORBA interface *ShapeList*



```
public interface ShapeListOperations {
    Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
    Shape[] allShapes();
    int getVersion();
}
```

```
public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,
        org.omg.CORBA.portable.IDLEntity { }
```

The equivalent Java interfaces – two per IDL interface. The name of the first Java interface ends in *Operations* – this interface just defines the operations in the IDL interface. The second Java interface has the same name as the IDL interface and implements the operations in the first interface as well as those in an interface suitable for a CORBA object. For example, the IDL interface *ShapeList* results in the two Java interfaces *ShapeListOperations* and *ShapeList*

- The server skeletons for each *idl* interface. The names of skeleton classes end in *POA* – for example, *ShapeListPOA.*
- The proxy classes or client stubs, one for each IDL interface. The names of these classes end in *Stub* – for example, *_ShapeListStub*
- A Java class to correspond to each of the *structs* defined with the IDL interfaces. In our example, classes *Rectangle* and *GraphicalObject* are generated. Each of these classes contains a declaration of one instance variable for each field in the corresponding *struct* and a pair of constructors, but no other methods.
- Classes called helpers and holders, one for each of the types defined in the IDL interface. A helper class contains the *narrow* method, which is used to cast down from a given object reference to the class to which it belongs, which is lower down the class hierarchy.

8.4 From objects to components

Issues with object-oriented middleware

- **Implicit dependencies:** A distributed object offers a *contract* to the outside world in terms of the interface (or interfaces) it offers to the distributed environment. The contract represents a binding agreement between the provider of the object and users of that object in terms of its expected behaviour. It is often assumed that such interfaces provide a complete contract for the deploying and use of this object. However, this is not the case. The problem arises from the fact that the internal (encapsulated) behaviour of an object is hidden.

- More generally, a given object can make arbitrary calls to other application-level objects or to distributed system services offering naming, persistence, concurrency control, transactions, security and so on, and this is not captured in the external view of the configuration. Implicit dependencies in the distributed configuration make it difficult to ensure the safe composition of a configuration, to replace one object with another, and for third-party developers to implement one particular element in a distributed configuration.

- From this, there is a clear requirement to specify not only the interfaces offered by an object but also the dependencies that object has on other objects in the distributed configuration

- **Interaction with the middleware**: Despite the goals of transparency, it is clear that in using distributed object middleware the programmer is exposed to many relatively low-level details associated with the middleware architecture.

- Despite this being a rather simple application, there are many CORBA-related calls that are absolutely essential to the operation of the application. These include calls associated with naming with the POA and to the ORB core.

- In more complex examples, this could include arbitrarily sophisticated code in terms of the creation and management of object references, management of object lifecycles, activation and passivation policies, management of persistent state and policies for mappings to underlying platform resources such as threads.

- All of this can very quickly become a distraction from the main purpose of the code, which is to create a particular application. This is all too evident from the example cited above, where the actual code related to the whiteboard application is minimal and interleaved with code related to distributed systems concerns

Lack of separation of distribution concerns: Programmers using distributed object middleware also have to deal explicitly with non-functional concerns related to issues such as security, transactions, coordination and replication. In technologies such as CORBA and RMI, this is achieved by inserting appropriate calls to the associated distributed system services within the objects.

This has two repercussions:

- Programmers must have an intimate knowledge of the full details of all the associated distributed system services.

- The implementation for a given object will contain application code alongside calls to distributed system services and to the underlying middleware interfaces The resultant tangling of concerns further increases the complexity of distributed systems programming.

No support for deployment: While technologies such as CORBA and Java RMI make it possible to develop arbitrary distributed configurations of objects, there is no support for the deployment of such configurations. Rather, objects must be deployed manually on individual machines.

This can become a tiresome and error-prone process, particularly with large-scale deployments consisting of many objects spread over a physical architecture with a large number of (potentially heterogeneous) nodes.

In addition to physically placing the objects, objects must also be activated and appropriate bindings created to other objects. Because of the lack of support for deployment, developers inevitably resort to ad hoc strategies for deployment, which are then not portable to other environments.

Essence of components –

- A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only.
- Software components are like distributed objects in that they are encapsulated units of composition, but a given component specifies both its interfaces provided to the outside world and its dependencies on other components in the distributed environment. The dependencies are also represented as interfaces.

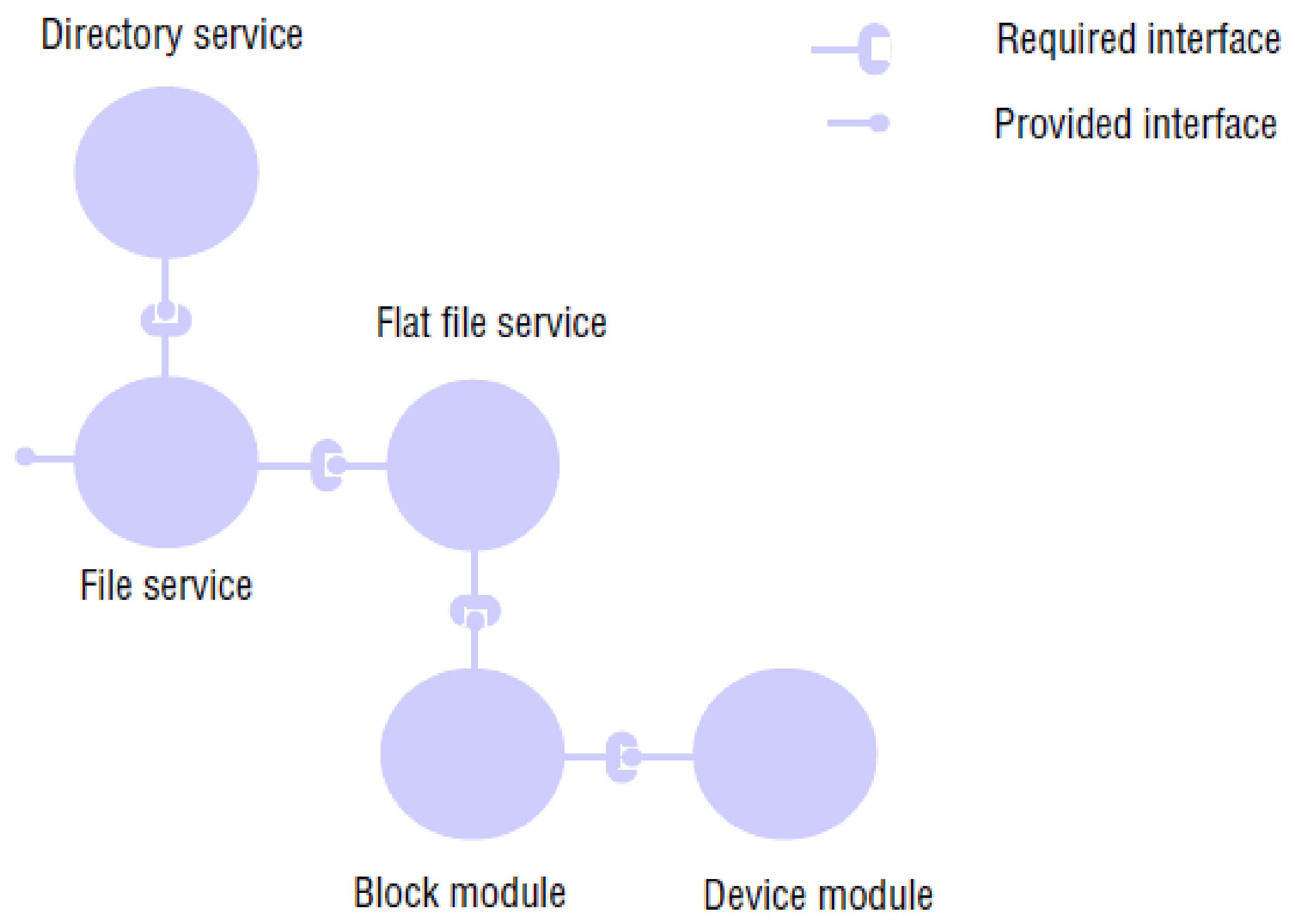
More specifically, a component is specified in terms of a *contract*, which includes:

- a **set of *provided interfaces*** – that is, interfaces that the component offers as services to other components;

- a **set of *required interfaces*** – that is, the dependencies that this component has in terms of other components that must be present and connected to this component for it to function correctly.

In a given component configuration, every required interface must be bound to a provided interface of another component. This is also referred to as a *software architecture* consisting of components, interfaces and connections between interfaces

Figure 8.11 An example software architecture



This example shows the architecture of a simple file system providing an interface to other users and in turn requiring connection to a directory service component and a flat file service component. The figure also shows additional connections to block and device modules, capturing the overall architecture of this particular file system.

Interfaces may be of different styles. In particular, many component-based approaches offer two styles of interface:

- interfaces supporting remote method invocation, as in CORBA and Java RMI;
- interfaces supporting distributed events.

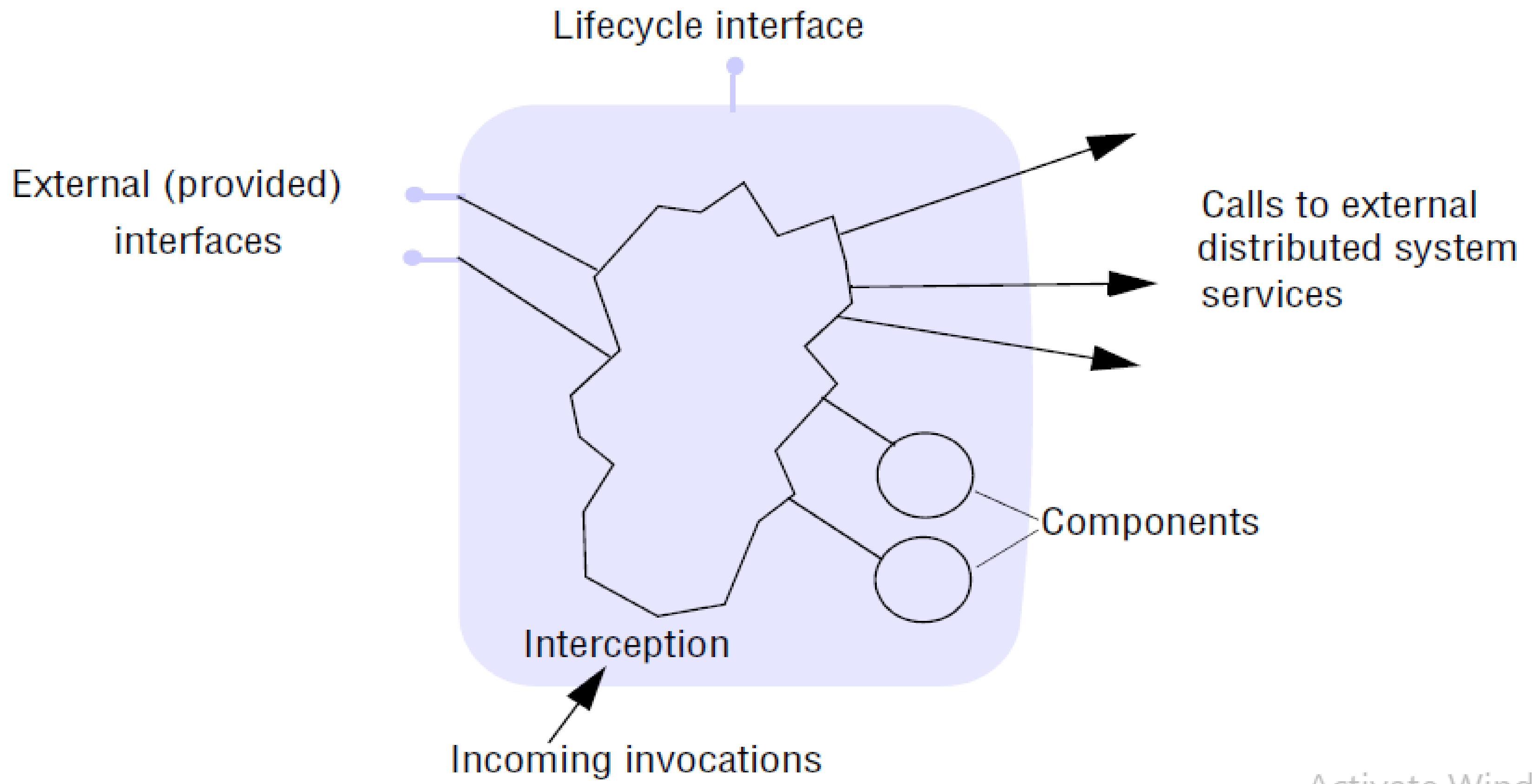
Programming in component-based systems is concerned with the development of components and their ***composition***. The goal is to support a style of software development that parallels hardware development in using off-the-shelf components and composing them together to develop more sophisticated

- services: a move from software development to software assembly.
- This approach therefore supports thirdparty development of software components and also makes it easier to adapt system configurations at runtime, by replacing one component with another that is a precise match in terms of provided and required interfaces.

- Note that advocates of component-based approaches place significant emphasis on this use of composition and see this as the cleanest approach to constructing complex software systems. In particular, they advocate composition over inheritance, viewing inheritance as creating additional forms of implicit dependency

Components and distributed systems : A range of component-based middleware technologies have emerged, including enterprise JavaBeans and the CORBA Component Model (CCM)

Figure 8.12 The structure of a container


Lifecycle interface
External (provided) interfaces
Calls to external distributed system services
Components
Interception
Incoming invocations

Containers: The concept of *containers* is absolutely central to component-based middleware. Containers support a common pattern often encountered in distributed applications, which consists of:

• front-end (perhaps web-based) client;

• a container holding one or more components that implement the application or business logic;

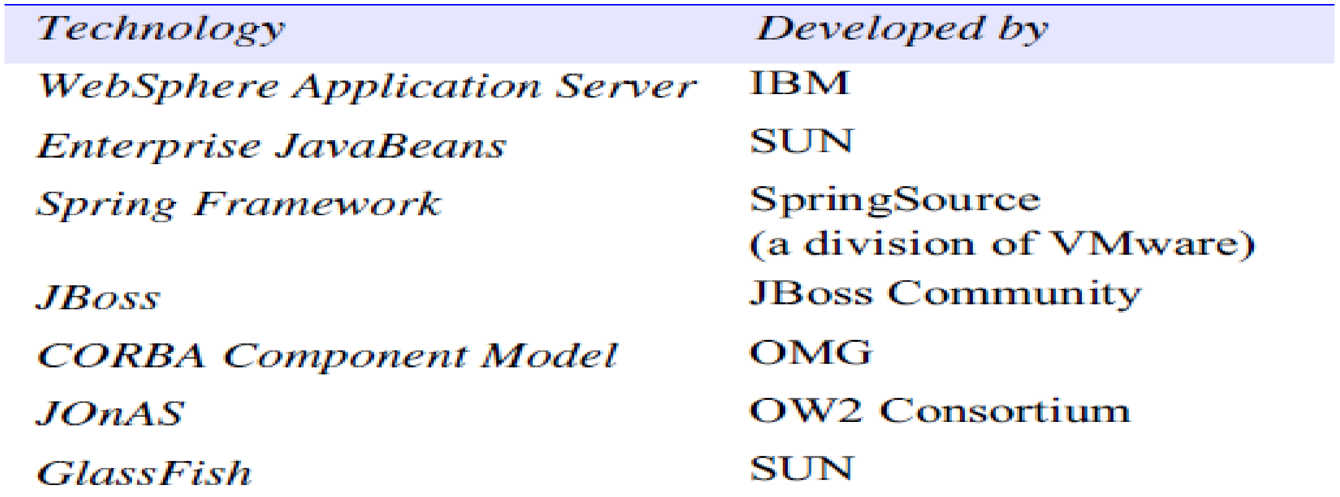
• system services that manage the associated data in persistent storage.

- The tasks of the container are to provide a ***managed* server-side** hosting environment for components and to provide the necessary separation of concerns alluded to above, where components deal with application concerns and the container deals with distributed systems and middleware issues, ensuring that non-functional properties are achieved.

If we take the case of CORBA, for example, this would include:

• managing the interaction with the underlying ORB core and POA functionality and hiding this entirely from application developers;
• managing calls to appropriate distributed system services, including security and transaction services, to provide the required non-functional properties of the application, again transparently to the programmer.

Figure 8.13 Application servers



Technology	*Developed by*
WebSphere Application Server	IBM
Enterprise JavaBeans	SUN
Spring Framework	SpringSource (a division of VMware)
JBoss	JBoss Community
CORBA Component Model	OMG
JOnAS	OW2 Consortium
GlassFish	SUN

Support for deployment:

Deployment descriptors are typically written in XML and include sufficient information to ensure that:

- components are correctly connected using appropriate protocols and associated middleware support;
- the underlying middleware and platform are configured to provide the right level of support to the component configuration (for example, in CORBA, this would include configuring the POA);
- the associated distributed system services are set up to provide the right level of security, transaction support and so on.

8.5 Case studies: Enterprise JavaBeans and Fractal

- The advantage of application servers is that they provide comprehensive support for one style of distributed programming – the three-tier approach as explained above – and most of the complexities associated with distributed programming are hidden from the user.

- The disadvantages are that the approach is both prescriptive and heavyweight: prescriptive in the sense that the approach mandates that particular style of systems architecture and heavyweight in that application servers are large and complex software systems that inevitably carry an overhead in terms of performance and resource requirements. The approach works best on high-end server machines.

- To counter this, a more stripped-down and minimal style of component programming is also adopted in distributed systems. We refer to this style as *lightweight component models* to distinguish them from the much more heavyweight application server architectures.

- Two case studies of component technologies are studied here: Enterprise JavaBeans, a leading example of the application server approach, and Fractal, an example of a lightweight component architecture.

8.5.1 Enterprise JavaBeans

- Enterprise JavaBeans (EJB) is a specification of a server-side, managed component architecture and a major element of the Java Platform, Enterprise Edition (Java EE), a set of specifications for client-server programming.

- EJB is defined as a ***server-side component model*** because it supports the development of the classic style of application, where potentially large numbers of clients interact with a number of services realized through components or configuration of components.

- The components, which are known as *beans* in EJB, are intended to capture the application (or business) logic, with EJB also supporting the separation between this application logic and its persistent storage in a back-end database. In other words, EJB provides direct support for the three-tier architecture.

- EJB is ***managed*** in the sense that the container pattern is used to provide support for key distributed systems services including transactions, security and lifecycle support. Typically, the container injects appropriate calls to the associated services to provide the required properties, and the use of a transaction manager or security services is completely hidden from the developer of the associated beans (***container-managed***). It is also possible for the bean developer to take more control over these operations (***bean-managed*****).**

- The goal of **EJB** is to maintain a strong separation of concerns between the various roles involved in developing distributed applications. The EJB specification identifies the following key roles:

- the ***bean provider***, who develops the application logic of the component(s);
- the ***application assembler***, who assembles beans into application configurations;
- the ***deployer***, who takes a given application assembly and ensures it can be correctly deployed in a given operational environment;
- the ***service provider***, who is a specialist in fundamental distributed system services such as transaction management and establishes the desired level of support in these areas;
- the ***persistence provider***, who is a specialist in mapping persistent data to underlying databases and in managing these relationships at runtime;
- the ***container provider***, who builds on the above two roles and is responsible for correctly configuring containers with the required level of distributed systems support in terms of non-functional properties related to, for example, transactions and security as well as the desired support for persistence;
- the ***system administrator***, who is responsible for monitoring a deployment at runtime and making any adjustments to ensure its correct operation.

The EJB component model • A *bean* in EJB is a component offering one or more *business interfaces* to potential clients of that component, where interfaces can be either *remote*, requiring the use of appropriate communication middleware (such as RMI or JMS), or *local*, in which case more direct, and hence efficient, bindings are possible.

- A given bean is represented by the set of remote and local business interfaces together with an associated *bean class* that implements the interfaces. Two main styles of bean are supported in the EJB 3.0 specification:
- ***Session beans*** *:* A session bean is a component implementing a particular task within the application logic of a service.
- ***Message-driven beans*** *:* Clients interact with session beans using local or remote invocation.

Session beans

- A session bean is a component implementing a particular task within the application logic of a service, for example to make a purchase in our *eShop* application.

- A session bean persists for the duration of a service and maintains a running conversation with the client for the duration of the session. Session beans can be either *stateful*, maintaining associated conversational state (such as the current status of the eCommerce transaction), or *stateless*, in which case no state is maintained.

- Stateful session beans imply a conversation with a single client and maintain the state of that conversation. In contrast, stateless beans can have many concurrent conversations with different clients. The state associated with stateful beans may or may not be persistent, as we discuss below.

Message-driven beans:

- Clients interact with session beans using local or remote invocation.
- The concept of a message-driven bean was introduced in EJB 2.0 to support indirect communication and, in particular, the possibility to interact with components using either message queues or topics, building directly on the functionality offered by JMS (remember that both queues and topics are firstclass entities in JMS representing alternative intermediaries for messages.
- In a message-drive bean, a business interface will be realized as a listener-style interface reflecting the event-driven nature of the associated bean.

POJOs and annotations

- The task of programming in EJB has been simplified significantly through the use of *Enterprise JavaBeanPOJOs* together with Java *Enterprise JavaBean annotations.* A bean is a plain old Java object: it consists of the application logic written simply in Java with no other code relating to it being a bean.

- Annotations were introduced in Java 1.5 as a mechanism for associating metadata with packages, classes, methods, parameters and variables. This metadata can then be used by frameworks to ensure the right behaviour or interpretation is associated with that part of the program.

Enterprise JavaBean containers in EJB

- EJB adopts a container-based approach. Beans are deployed to containers, and the containers provide implicit distributed system management using interception.

- In this way, the container provides the necessary policies in areas including transaction management, security, persistence and lifecycle management allowing the bean developer to focus exclusively on the application logic.

- Containers must therefore be configured with the necessary level of support. In the current version, EJB is preconfigured with common default policies and the developer need only take action if these defaults are insufficient.
- Transactions in EJB apply equally to any style of bean, including both session beans and message-driven beans.

- The first thing to declare is whether transactions associated with an enterprise bean should be bean-managed or container-managed. This is achieved by associating the following annotations with the associated class, respectively:

  @TransactionManagement (BEAN)

  @TransactionManagement (CONTAINER)

Note that, by default, transactions are container-managed in EJB.

Figure 8.14 Transaction attributes in EJB.

Attribute	*Policy*
REQUIRED	If the client has an associated transaction running, execute within this transaction; otherwise, start a new transaction.
REQUIRES_NEW	Always start a new transaction for this invocation.
SUPPORTS	If the client has an associated transaction, execute the method within the context of this transaction; if not, the call proceeds without any transaction support.
NOT_SUPPORTED	If the client calls the method from within a transaction, then this transaction is suspended before calling the method and resumed afterwards – that is, the invoked method is excluded from the transaction.
MANDATORY	The associated method must be called from within a client transaction; if not, an exception is thrown.
NEVER	The associated methods must not be called from within a client transaction; if this is attempted, an exception is thrown.

Dependency injection: The example above also illustrates a further important role of containers: *Enterprise JavaBean dependency injection*. Dependency injection is a common pattern in programming whereby a third party, in this case a container, is responsible for managing and resolving the relationships between a component and its dependencies.
In particular, in EJB 3.0, a component refers to a dependency using an annotation and the container is responsible for resolving this annotation and ensuring that, at runtime, the associated attribute refers to the right object. This is typically implemented by the container using reflection.

Enterprise JavaBean Interception • The Enterprise JavaBeans specification enables the programmer to intercept two types of operation on beans in order to alter their default behaviour:
- method calls associated with a business interface;
- lifecycle events.

Interception of methods: This mechanism is used where it is necessary to associate a particular action or set of actions with an incoming call on a business interface. This applies equally to incoming invocations on a session bean or incoming events on a message-driven bean.

Interception of lifecycle events: A similar mechanism can be used to intercept and react to lifecycle events associated with a component. In particular, the EJB specification allows a bean developer to associate interceptors with the creation and deletion of components using the following annotations, respectively:

@PostConstruct

@PreDestroy

The annotations are associated with given methods in the bean class, with the effect that these methods will be called when the associated lifecycle events happen

8.5.2 Fractal

- Fractal is a lightweight component model that brings the benefits of component-based programming to the development of distributed systems.

- Fractal provides support for *programming with interfaces*, with the associated benefits in terms of the separation of interface and implementation. Fractal goes further, though, and supports the *explicit representation* of the software architecture of the system, avoiding the problem of implicit dependencies.

- Fractal is used to construct more complex software systems (including middleware systems as discussed below) using the component model as the basic building block, resulting in software that has a clear component-based architecture and that is *configurable* and also *reconfigurable* at runtime to match the current operational environment and requirements.

- Fractal defines a programming model and, as such, is programming language–agnostic.

Implementations of this model are available in several different languages, including:
• Julia and AOKell (Java-based, with the latter also offering support for aspect oriented programming);
• Cecilia and Think (C-based);
• FracNet (.NET-based);
• FracTalk (Smalltalk-based);
• Julio (Python-based).
Julia and Cecilia are treated as the reference implementations of Fractal.

Fractal is supported by the **OW**2 consortium [www.ow2.org], an open source software community for distributed systems middleware that encourages and promotes the component-based philosophy for the construction of such software.

To date, Fractal has been used in the construction of a wide range of middleware platforms including **Think** (a configurable operating system kernel), **DREAM** (a middleware platform supporting various forms of indirect communication), **Jasmine** (a tool supporting the monitoring and management of SOA platforms), **GOTM** (offering flexible transaction management) and Proactive (a middleware platform for Grid computing).

Fractal is also the basis of the **Grid Component Model** (GCM), which has been influential in the development of associated ETSI standards . Further details of all these projects can be found on the OW2 web site.

The core component model • A component in Fractal offers one or more interfaces, with two types of interfaces available:

- ***server interfaces***, which support incoming operational invocations
- ***client interfaces***, which support outgoing invocations (equivalent to required interfaces).
- An interface is an implementation of an *interface type*, which defines the operations that are supported by that interface.

Bindings in Fractal: To enable composition, Fractal supports *bindings* between interfaces. Two styles of binding are supported by the model:

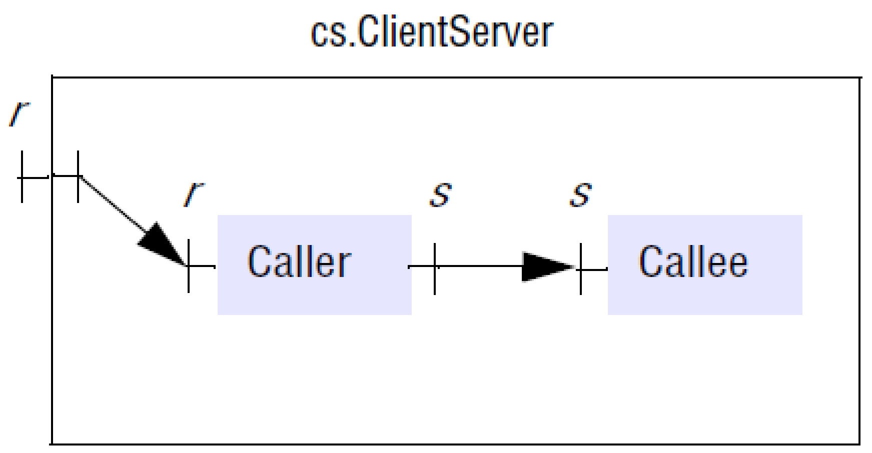
Primitive bindings**:** The simplest style of binding is a *primitive binding*, which is a direct mapping between one client interface and one server interface within the same address space, assuming the types are compatible. Primitive bindings can be implemented efficiently in a given language environment.

Composite bindings: Fractal also supports *composite bindings*, which are arbitrarily complex software architectures (that is consisting of components and bindings) implementing communication between a number of interfaces potentially on different machines.

Composite bindings are themselves components in Fractal, and this is important for two reasons:

- A system developed using Fractal is fully configurable in terms of the components *and* their interconnections.
- If a given communication paradigm is not already provided, it can be developed in Fractal and then made available to future developers as a component.
- Once established, any aspect of the software architecture can be reconfigured at runtime, including composite bindings. It is very useful to be able to adapt communication structures at runtime, for example to introduce added levels of security or to alter the implementation to be more scalable as a system grows in size.

Figure 8.16 An example component configuration in Fractal



Hierarchical composition: The component model is *hierarchical* in that a component consists of a series of subcomponents and associated bindings, where the subcomponents may themselves be composite. Composition is supported by a Fractal Architectural Description Language (ADL), which we introduce by a simple example showing the creation of a component containing two subcomponents that interact in a client-server manner:

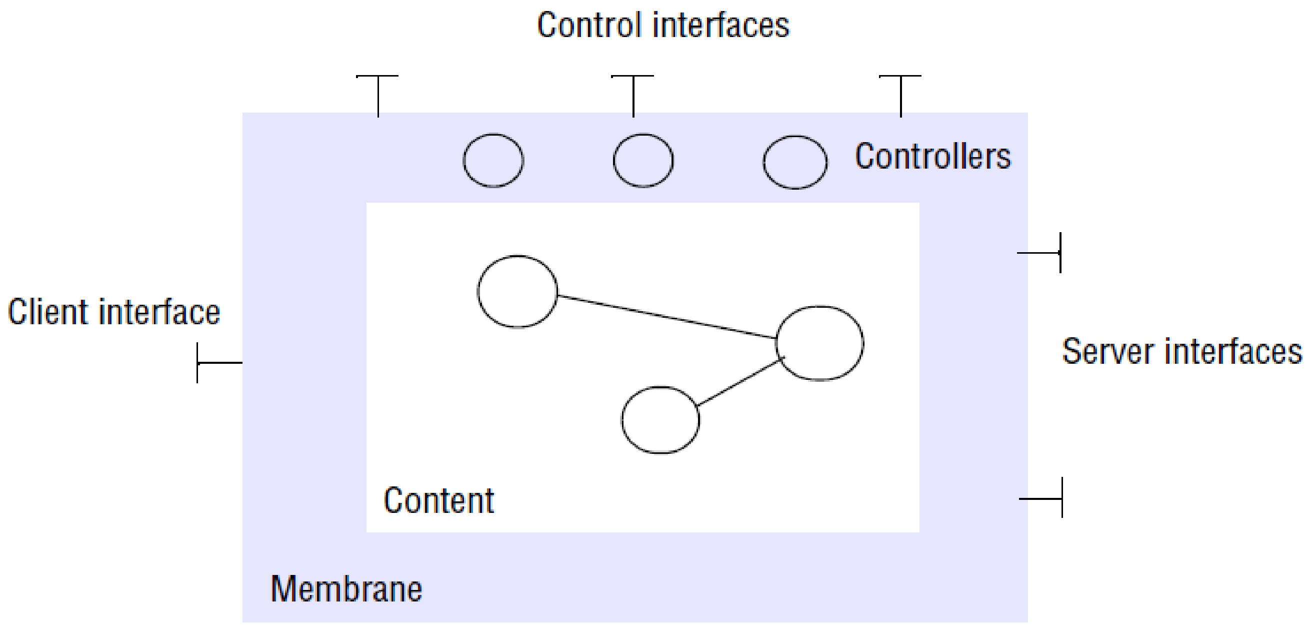
```
<definition name="cs.ClientServer">
<interface name="r" role="server"
signature="java.lang.Runnable" />
<component name="caller"
definition="hw.CallerImpl" />
<component name="callee"
definition="hw.CalleeImpl" />
<binding client="this.r" server="caller.r" />
<binding client="caller.s" server="callee.s" />
</definition>
```

Fractal ADL is based on XML. This example shows a component *cs.ClientServer* with two subcomponents, *caller* and *callee*; bindings are created between the client interface, *this.r* (that is, the *r* interface defined on the containing component *cs.ClientServer*), and the associated *caller.r* interface (the *r* interface defined on the *caller* component), and between the client interface *caller.s* and the corresponding server interface *callee.s*.

Fractal also supports ***sharing,*** whereby a given component may be shared across multiple software architectures. The developers of Fractal argue that this is necessary to faithfully represent system architectures including access to underlying resources that are fundamentally shared, such as a TCP connection.

Membranes and controllers -In implementation, a component consists of a *membrane*, which defines control capabilities associated with the component through a set of *controllers*, and also the associated *content* – the subcomponents (and bindings) that make up its architecture. Interfaces can be *internal* to the membrane and hence only

Figure 8.17 The structure of a Fractal component



The membrane concept is crucial to the Fractal approach; a membrane provides a configurable control regime for the encapsulated set of components (the content). In other words, the set of controllers defines the control capabilities and associated semantics for these components. By changing the set of controllers, we also change the capabilities.

Controllers can be used for various purposes:

- One of the key uses of controllers is to implement *lifecycle management*, including operations associated with activation and passivation such as *suspend*, *resume* and *checkpoint*. For example, Fractal supports a *LifeCycleController* supporting three methods, *startFc*, *stopFc* and *getFcState*, which implements these three functions, respectively. This is crucial in cases where reconfigurations of the underlying software architecture are being carried out at runtime.

- Controllers also offer reflection capabilities . In particular, *introspection* capabilities are provided through two interfaces, *Component* and *ContentController*, which support introspection (dynamic discovery) of the interfaces associated with a component and step through the architecture of a composite component structure respectively.

Figure 8.18 *Component* and *ContentController* Interfaces in Fractal

```
public interface Component {
    Object[] getFcInterfaces ();
    Object getFcInterface (String itfName);
    Type getFcType ();
}

public interface ContentController {
    Object[] getFcInternalInterfaces ();
    Object getFcInterfaceInterface(String itfName);
    Component[] getFcSubComponents ();
    void addFcSubComponent (Component c);
    void removeFcSubComponent(Component c);
}
```

Controllers can be introduced to offer *interception* capabilities mirroring the capability offered in EJB. Interception could be used in the client-server example to log all calls issued by the *caller* component in a manner that would be completely transparent to both the *caller* and *callee*. A further use of interception is to implement an access control policy only allowing an invocation to proceed if a given principal has rights to access a given resource

Membranes, like containers, provide a place for the deployment of components; both techniques also support implicit distributed systems management, containers by making implicit calls to distributed systems services and membranes through their constituent controllers. **Membranes, though, are significantly more flexible** :

- In terms of reflection, support can range from black-box components where internal structure is hidden, through approaches where limited introspection capabilities are offered (dynamically discovering interfaces), to advanced reflection features supporting full introspection and providing inherent support for subsequent adaptation of internal structures.

- In terms of supporting non-functional concerns, at one extreme, membranes can provide no more than a simple encapsulation of components at the other extreme, they can support fully fledged distributed systems management of components, including support for transactions and security as in application servers, but in a completely configurable and reconfigurable manner.