

APPENDIX D

OBJECT-ORIENTED DESIGN

William Stallings

Copyright 2014

D.1 MOTIVATION

D.2 OBJECT-ORIENTED CONCEPTS

Object Structure

Object Classes

Containment

D.3 BENEFITS OF OBJECT-ORIENTED DESIGN

D.4 CORBA

D.5 RECOMMEDED READING AND WEB SITE

Supplement to

Operating Systems, Eighth Edition

Pearson 2014

<http://williamstallings.com/OperatingSystems/>

Windows and several other contemporary operating systems rely heavily on object-oriented design principles. This appendix provides a brief overview of the main concepts of object-oriented design.

D.1 MOTIVATION

Object-oriented concepts have become quite popular in the area of computer programming, with the promise of interchangeable, reusable, easily updated, and easily interconnected software parts. More recently, database designers have begun to appreciate the advantages of an object orientation, with the result that object-oriented database management systems (OODBMS) are beginning to appear. Operating systems designers have also recognized the benefits of the object-oriented approach.

Object-oriented programming and object-oriented database management systems are in fact different things, but they share one key concept: that software or data can be "containerized." Everything goes into a box, and there can be boxes within boxes. In the simplest conventional program, one program step equates to one instruction; in an object-oriented language, each step might be a whole boxful of instructions. Similarly, with an object-oriented database, one variable, instead of equating to a single data element, may equate to a whole boxful of data.

Table D.1 introduces some of the key terms used in object-oriented design.

Table D.1 Key Object-Oriented Terms

Term	Definition
Attribute	Data variables contained within an object.
Containment	A relationship between two object instances in which the containing object includes a pointer to the contained object.
Encapsulation	The isolation of the attributes and services of an object instance from the external environment. Services may only be invoked by name and attributes may only be accessed by means of the services.
Inheritance	A relationship between two object classes in which the attributes and services of a parent class are acquired by a child class.
Interface	A description closely related to an object class. An interface contains method definitions (without implementations) and constant values. An interface cannot be instantiated as an object.
Message	The means by which objects interact.
Method	A procedure that is part of an object and that can be activated from outside the object to perform certain functions.
Object	An abstraction of a real-world entity.
Object class	A named set of objects that share the same names, sets of attributes, and services.
Object instance	A specific member of an object class, with values assigned to the attributes.
Polymorphism	Refers to the existence of multiple objects that use the same names for services and present the same interface to the external world but that represent different types of entities.
Service	A function that performs an operation on an object

D.2 OBJECT-ORIENTED CONCEPTS

The central concept of object-oriented design is the object. An object is a distinct software unit that contains a collection of related variables (data) and methods (procedures). Generally, these variables and methods are not directly visible outside the object. Rather, well-defined interfaces exist that allow other software to have access to the data and the procedures.

An object represents some thing, be it a physical entity, a concept, a software module, or some dynamic entity such as a TCP connection. The values of the variables in the object express the information that is known about the thing that the object represents. The methods include procedures whose execution affect the values in the object and possibly also affect that thing being represented.

Figures D.1 and D.2 illustrate key object-oriented concepts.

Object Structure

The data and procedures contained in an object are generally referred to as variables and methods, respectively. Everything that an object "knows" can be expressed in its variables, and everything it can do is expressed in its methods.

The **variables** in an object, also called **attributes**, are typically simple scalars or tables. Each variable has a type, possibly a set of allowable values, and may either be constant or variable (by convention, the term *variable* is used even for constants). Access restrictions may also be imposed on variables for certain users, classes of users, or situations.

The **methods** in an object are procedures that can be triggered from outside to perform certain functions. The method may change the state of

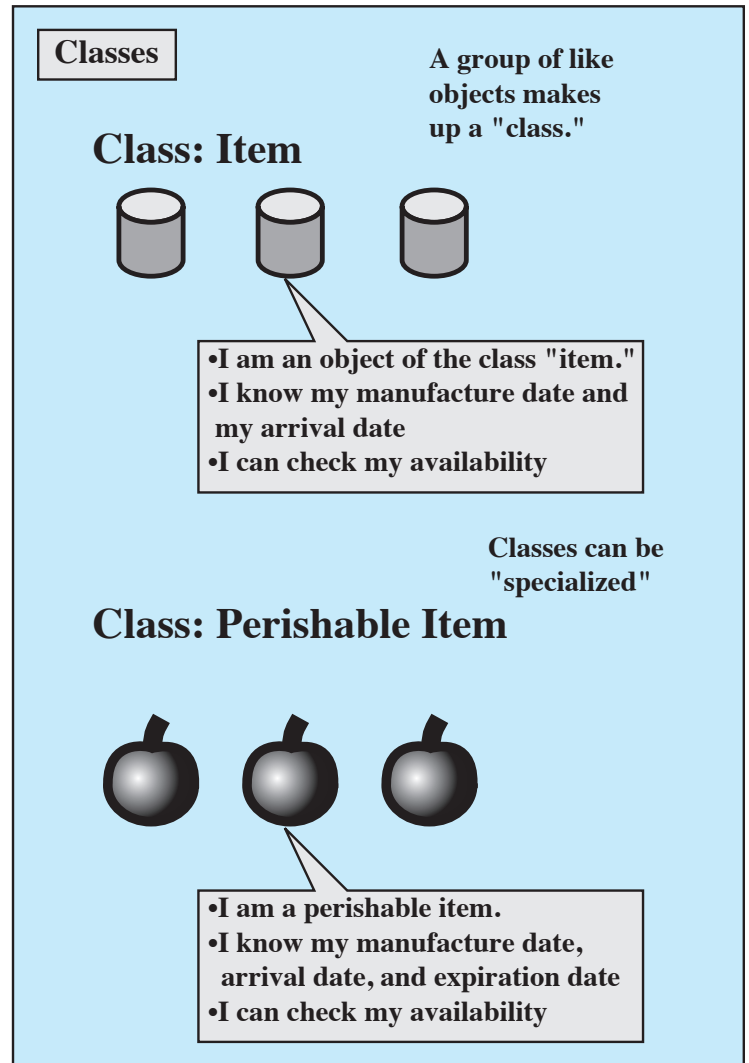
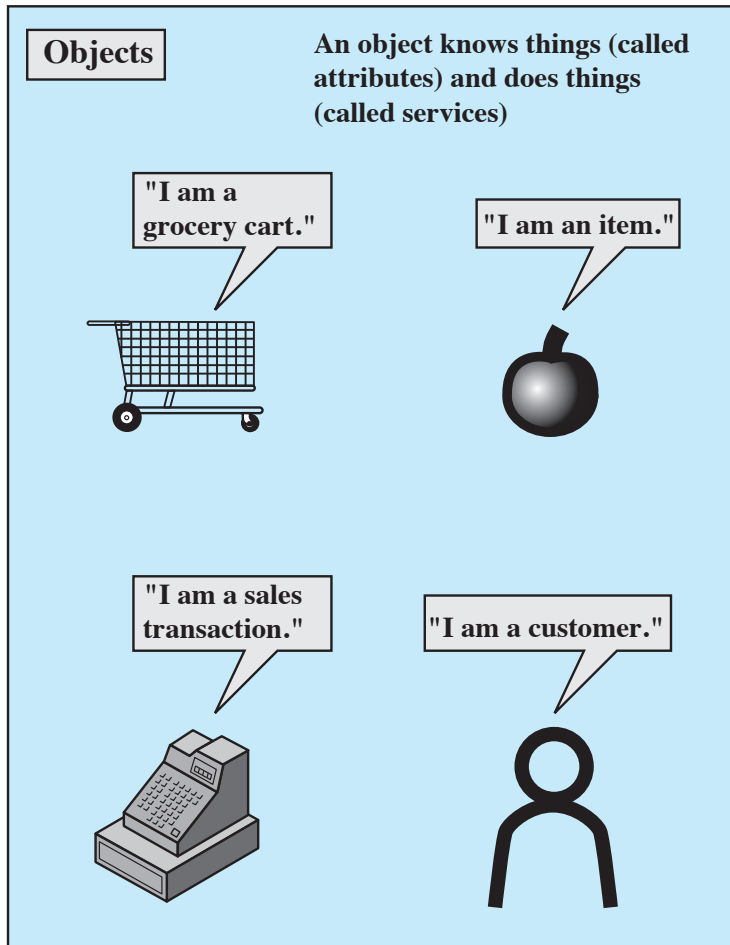
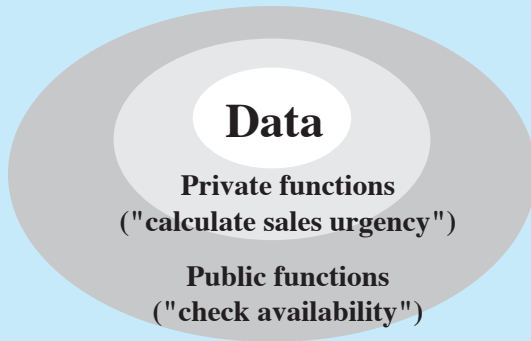


Figure D.1 Objects

Encapsulation

The principle that an object should hide things from other objects, limiting visibility about what "I know and do."



Inheritance

The principle that a class can extend from another previously defined class. The guiding principle is to organize the classes according to generalization/specialization.

Generalization



Class: item
"I am an item"

Specialization



Class: Perishable Item
"I am a perishable item"

Polymorphism

The principle that objects in different classes may understand the same message yet respond in different ways.

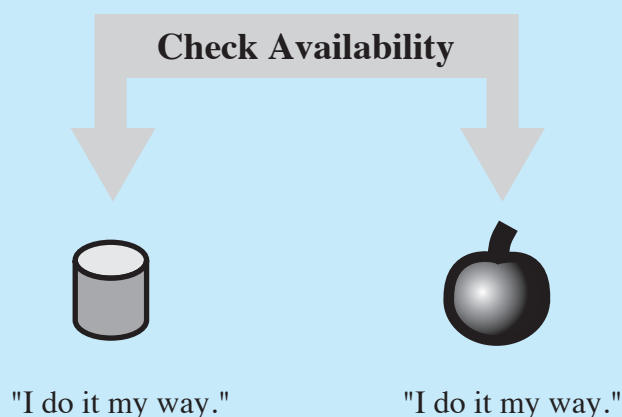


Figure D.2 Object Concepts

the object, update some of its variables, or act on outside resources to which the object has access.

Objects interact by means of **messages**. A message includes the name of the sending object, the name of the receiving object, the name of a method in the receiving object, and any parameters needed to qualify the execution of the method. A message can only be used to invoke a method within an object. The only way to access the data inside an object is by means of the object's methods. Thus, a method may cause an action to be taken or for the object's variables to be accessed, or both. For local objects, passing a message to an object is the same as calling an object's method. When objects are distributed, passing a message is exactly what it sounds like.

The interface of an object is a set of public methods that the object supports. An interface says nothing about implementation; objects in different classes may have different implementations of the same interfaces.

The property of an object that its only interface with the outside world is by means of messages is referred to as **encapsulation**. The methods and variables of an object are encapsulated and available only via message-based communication. Encapsulation offers two advantages:

1. It protects an object's variables from corruption by other objects. This protection may include protection from unauthorized access and protection from the types of problems that arise from concurrent access, such as deadlock and inconsistent values.
2. It hides the internal structure of the object so that interaction with the object is relatively simple and standardized. Furthermore, if the internal structure or procedures of an object are modified without changing its external functionality, other objects are unaffected.

Object Classes

In practice, there will typically be a number of objects representing the same types of things. For example, if a process is represented by an object, then there will be one object for each process present in a system. Clearly, every such object needs its own set of variables. However, if the methods in the object are reentrant procedures, then all similar objects could share the same methods. Furthermore, it would be inefficient to redefine both methods and variables for every new but similar object.

The solution to these difficulties is to make a distinction between an object class and an object instance. An **object class** is a template that defines the methods and variables to be included in a particular type of object. An **object instance** is an actual object that includes the characteristics of the class that defines it. The object contains values for the variables defined in the object class. **Instantiation** is the process of creating a new object instance for an object class.

INHERITANCE

The concept of an object class is powerful because it allows for the creation of many object instances with a minimum of effort. This concept is made even more powerful by the use of the mechanism of inheritance [TAIV96].

Inheritance enables a new object class to be defined in terms of an existing class. The new (lower level) class, called the **subclass**, or the **child class**, automatically includes the methods and variable definitions in the original (higher level) class, called the **superclass**, or **parent class**. A subclass may differ from its superclass in a number of ways:

1. The subclass may include additional methods and variables not found in its superclass.

2. The subclass may override the definition of any method or variable in its superclass by using the same name with a new definition. This provides a simple and efficient way of handling special cases.
3. The subclass may restrict a method or variable inherited from its superclass in some way.

Figure D.3, based on one in [KORS90], illustrates the concept.

The inheritance mechanism is recursive, allowing a subclass to become the superclass of its own subclasses. In this way, an **inheritance hierarchy** may be constructed. Conceptually, we can think of the inheritance hierarchy as defining a search technique for methods and variables. When an object receives a message to carry out a method that is not defined in its class, it automatically searches up the hierarchy until it finds the method. Similarly, if the execution of a method results in the reference to a variable not defined in that class, the object searches up the hierarchy for the variable name.

POLYMORPHISM

Polymorphism is an intriguing and powerful characteristic that makes it possible to hide different implementations behind a common interface. Two objects that are polymorphic to each other utilize the same names for methods and present the same interface to other objects. For example, there may be a number of print objects, for different output devices, such as `printDotmatrix`, `printLaser`, `printScreen`, and so forth, or for different types of documents, such as `printText`, `printDrawing`, `printCompound`. If each such object includes a method called `print`, then any document could be printed by sending the message `print` to the appropriate object, without concern for how that method is actually carried out. Typically, polymorphism is used to allow you have the same method in multiple subclasses of the same superclass, each with a different detailed implementation.

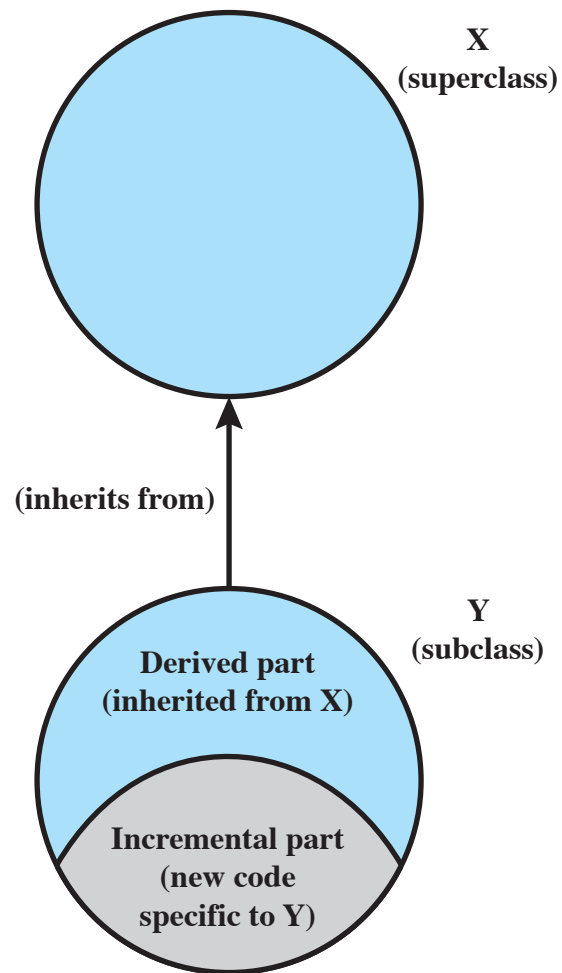


Figure D.3 Inheritance

It is instructive to compare polymorphism to the usual modular programming techniques. An objective of top-down, modular design is to design lower-level modules of general utility with a fixed interface to higher-level modules. This allows the one lower-level module to be invoked by many different higher-level modules. If the internals of the lower-level module are changed without changing its interface, then none of the upper-level modules that use it are affected. By contrast, with polymorphism, we are concerned with the ability of one higher-level object to invoke many different lower-level objects using the same message format to accomplish similar functions. With polymorphism, new lower-level objects can be added with minimal changes to existing objects.

INTERFACES

Inheritance enables a subclass object to use functionality of a superclass. There may be cases when you wish to define a subclass that has functionality of more than one superclass. This could be accomplished by allowing a subclass to inherit from more than one superclass. C++ is one language that allows such multiple inheritance. However, for simplicity, most modern object-oriented languages, including Java, C#, and Visual Basic .NET, limit a class to inheriting from only one superclass. Instead, a feature known as *interfaces* is used to enable a class to borrow some functionality from one class and other functionality from a completely different class.

Unfortunately, the term *interface* is used in much of the literature on objects with both a general-purpose and a specific functional meaning. An interface, as we are discussing it here, specifies an application-programming interface (API) for certain functionality. It does not define any implementation for that API. The syntax for an interface definition typically looks similar to a class definition, except that there is no code defined for the methods, just the method names, the arguments passed, and the type

of the value returned. An interface may be implemented by a class. This works in much the same way that inheritance works. If a class implements an interface, it must have the properties and methods of the interface defined in the class. The methods that are implemented can be coded in any fashion, so long as the name, arguments, and return type of each method from the interface are identical to the definition in the interface.

Containment

Object instances that contain other objects are called **composite objects**. Containment may be achieved by including the pointer to one object as a value in another object. The advantage of composite objects is that they permit the representation of complex structures. For example, an object contained in a composite object may itself be a composite object.

Typically, the structures built up from composite objects are limited to a tree topology; that is, no circular references are allowed and each "child" object instance may have only one "parent" object instance.

It is important to be clear about the distinction between an inheritance hierarchy of object classes and a containment hierarchy of object instances. The two are not related. The use of inheritance simply allows many different object types to be defined with a minimum of efforts. The use of containment allows the construction of complex data structures.

D.3 BENEFITS OF OBJECT-ORIENTED DESIGN

[CAST92] lists the following benefits of object-oriented design:

- **Better organization of inherent complexity:** Through the use of inheritance, related concepts, resources, and other objects can be

efficiently defined. Through the use of containment, arbitrary data structures, which reflect the underlying task at hand, can be constructed. Object-oriented programming languages and data structures enable designers to describe operating system resources and functions in a way that reflects the designer's understanding of those resources and functions.

- **Reduced development effort through reuse:** Reusing object classes that have been written, tested, and maintained by others cuts development, testing, and maintenance time.
- **More extensible and maintainable systems:** Maintenance, including product enhancements and repairs, traditionally consumes about 65% of the cost of any product life cycle. Object-oriented design drives that percentage down. The use of object-based software helps limit the number of potential interactions of different parts of the software, ensuring that changes to the implementation of a class can be made with little impact on the rest of the system.

These benefits are driving operating system design in the direction of object-oriented systems. Objects enable programmers to customize an operating system to meet new requirements, without disrupting system integrity. Objects also pave the road to distributed computing. Because objects communicate by means of messages, it matters not whether two communicating objects are on the same system or on two different systems in a network. Data, functions, and threads can be dynamically assigned to

workstations and servers as needed. Accordingly, the object-oriented approach to the design of operating systems is becoming increasingly evident in PC and workstation operating systems.

D.4 CORBA

As we have seen in this book, object-oriented concepts have been used to design and implement operating system kernels, bringing benefits of flexibility, manageability, and portability. The benefits of using object-oriented techniques extend with equal or greater benefit to the realm of distributed software, including distributed operating systems. The application of object-oriented techniques to the design and implementation of distributed software is referred to as distributed object computing (DOC).

The motivation for DOC is the increasing difficulty in writing distributed software: while computing and network hardware get smaller, faster, and cheaper, distributed software gets larger, slower, and more expensive to develop and maintain. [SCHM97] points out that the challenge of distributed software stems from two types of complexity:

- **Inherent:** Inherent complexities arise from fundamental problems of distribution. Chief among these are detecting and recovering from network and host failures, minimizing the impact of communication latency, and determining an optimal partitioning of service components and workload onto computers throughout a network. In addition, concurrent programming, with issues of resource locking and deadlocks, is still hard, and distributed systems are inherently concurrent.

- **Accidental:** Accidental complexities arise from limitations with tools and techniques used to build distributed software. A common source of accidental complexity is the widespread use of functional design, which results in nonextensible and non-reusable systems.

DOC is a promising approach to managing both types of complexity. The centerpiece of the DOC approach are object request brokers (ORBs), which act as intermediaries for communication between local and remote objects. ORBs eliminate some of the tedious, error-prone, and nonportable aspects of designing and implementing distributed applications. Supplementing the ORB must be a number of conventions and formats for message exchange and interface definition between applications and the object-oriented infrastructure.

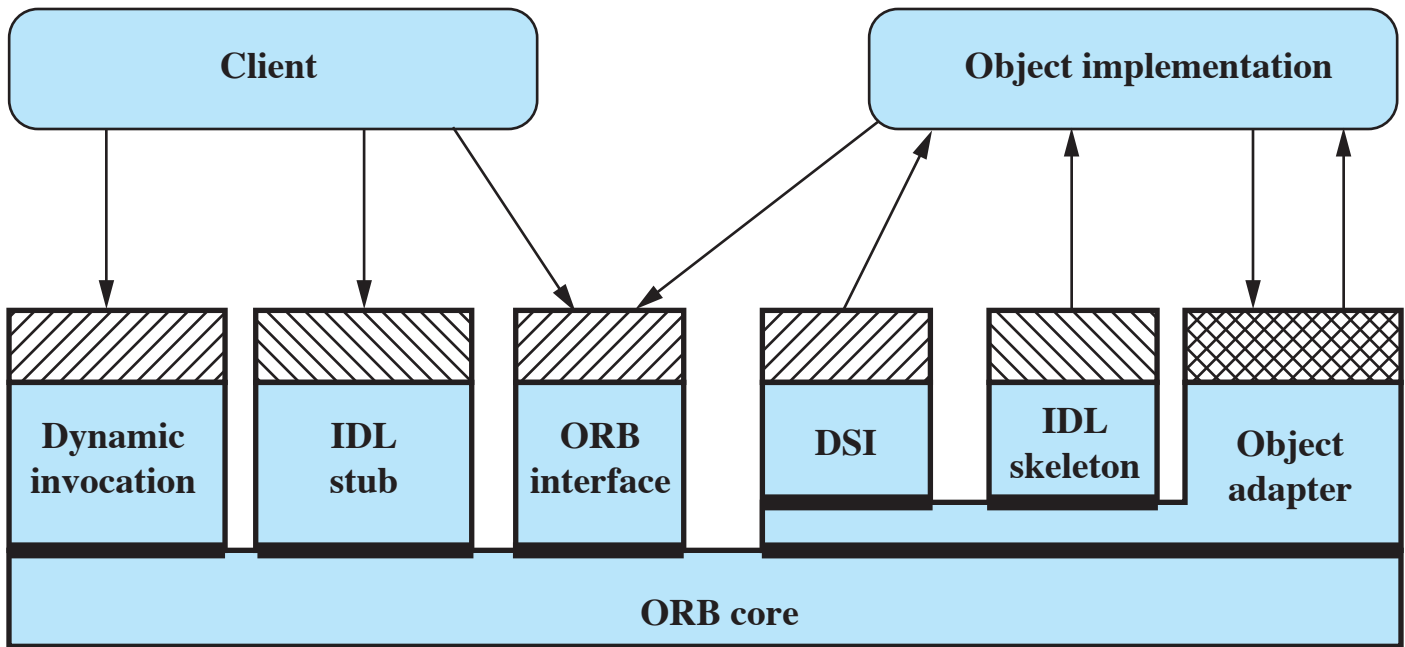
There are three main competing technologies in the DOC market: the object management group (OMG) architecture, called Common Object Request Broker Architecture (CORBA); the Java remote method invocation (RMI) system; and Microsoft's distributed component object model (DCOM). CORBA is the most advanced and well established of the three. A number of industry leaders, including IBM, Sun, Netscape, and Oracle, support CORBA, and Microsoft has announced that it will link its Windows-only DCOM with CORBA. The remainder of this appendix provides a brief overview of CORBA.

Table D.2 Key Concepts in a Distributed CORBA System

CORBA Concept	Definition
Client application	Invokes requests for a server to perform operations on objects. A client application uses one or more interface definitions that describe the objects and operations the client can request. A client application uses object references, not objects, to make requests.
Exception	Contains information that indicates whether a request was successfully performed.
Implementation	Defines and contains one or more methods that do the work associated with an object operation. A server can have one or more implementations.
Interface	Describes how instances of an object will behave, such as what operations are valid on those objects.
Interface definition	Describes the operations that are available on a certain type of object.
Invocation	The process of sending a request.
Method	The server code that does the work associated with an operation. Methods are contained within implementations.
Object	Represents a person, place, thing, or piece of software. An object can have operations performed on it, such as the promote operation on an employee object.
Object instance	An occurrence of one particular kind of object.
Object reference	An identifier of an object instance.
OMG Interface Definition Language (IDL)	A definition language for defining interfaces in CORBA.
Operation	The action that a client can request a server to perform on an object instance.
Request	A message sent between a client and a server application.
Server application	Contains one or more implementations of objects and their operations.

Table D.2 defines some key terms used in CORBA. The main features of CORBA are as follows (Figure D.4):

- **Clients:** Clients generate requests and access object services through a variety of mechanisms provided by the underlying ORB.
- **Object implementations:** These implementations provide the services requested by various clients in the distributed system. One benefit of the CORBA architecture is that both clients and object implementations can be written in any number of programming languages and can still provide the full range of required services.
- **ORB core:** The ORB core is responsible for communication between objects. The ORB finds an object on the network, delivers requests to the object, activates the object (if not already active), and returns any message back to the sender. The ORB core provides **access transparency** because programmers use exactly the same method with the same parameters when invoking a local method or a remote method. The ORB core also provides **location transparency**: Programmers do not need to specify the location of an object.
- **Interface:** An object's interface specifies the operations and types supported by the object, and thus defines the requests that can be made on the object. CORBA interfaces are similar to classes in C++ and interfaces in Java. Unlike C++ classes, a CORBA interface specifies methods and their parameters and return values but is silent about their



Same for all ORBs

ORB Object request broker



Interface-specific stubs and skeletons

IDL Interface definition language



May be multiple object adapters

DSI Dynamic skeleton interface



ORB-private interface

Figure D.4 Common Object Request Broker Architecture

implementation. Two objects of the same C++ class have the same implementation of their methods.

- **OMG interface definition language (IDL):** IDL is the language used to define objects. An example IDL interface definition is

```
//OMG IDL
interface Factory
    { Object create ( ) ;
    } ;
```

This definition specifies an interface named `Factory` that supports one operation, `create`. The `create` operation takes no parameters and returns an object reference of type `Object`. Given an object reference for an object of type `Factory`, a client could invoke it to create a new CORBA object. IDL is a programming-independent language and, for this reason, a client does not invoke directly any object operation. It needs a mapping to the client programming language to do that. It is possible, as well, that the server and the client are programmed in different programming languages. The use of a specification language is a way to deal with heterogeneous processing across multiple languages and platform environments. Thus, IDL enables **platform independence**.

- **Language binding creation:** IDL compilers map one OMG IDL file to different programming languages, which may or may not be object oriented, such as Java, Smalltalk, Ada, C, C++, and COBOL. That mapping includes the definition of the language-specific data types and procedure interfaces to access service objects, the IDL client stub

interface, the IDL skeleton, the object adapters, the dynamic skeleton interface, and the direct ORB interface. Usually, clients have a compile-time knowledge of the object interface and use client stubs to do a static invocation; in certain cases, clients do not have that knowledge and they must do a dynamic invocation.

- **IDL stub:** Makes calls to the ORB core on behalf of a client application. IDL stubs provide a set of mechanisms that abstract the ORB core functions into direct RPC (remote procedure call) mechanisms that can be employed by the end-client applications. These stubs make the combination of the ORB and remote object implementation appear as if they were tied to the same in-line process. In most cases, IDL compilers generate language-specific interface libraries that complete the interface between the client and object implementations.
- **IDL skeleton:** Provides the code that invokes specific server methods. Static IDL skeletons are the server-side complements to the client-side IDL stubs. They include the bindings between the ORB core and the object implementations that complete the connection between the client and object implementations.
- **Dynamic invocation:** Using the dynamic invocation interface (DII), a client application can invoke requests on any object without having compile-time knowledge of the object's interfaces. The interface details are filled in by consulting with an interface repository and/or other run-

time sources. The DII allows a client to issue one-way commands (for which there is no response).

- **Dynamic skeleton interface (DSI):** Similar to the relationship between IDL stubs and static IDL skeletons, the DSI provides dynamic dispatch to objects. Equivalent to dynamic invocation on the server side.
- **Object adapter:** An object adapter is CORBA system component provided by the CORBA vendor to handle general ORB-related tasks, such as activating objects and activating implementations. The adapter takes these general tasks and ties them to particular implementations and methods in the server.

D.5 RECOMMENDED READING AND WEB SITE


[KORS90] is a good overview of object-oriented concepts. [STRO88] is a clear description of object-oriented programming. An interesting perspective on object-oriented concepts is provided in [SYND93]. [VINO97] is an overview of CORBA.

KORS90 Korson, T., and McGregor, J. "Understanding Object-Oriented: A Unifying Paradigm." *Communications of the ACM*, September 1990.

STRO88 Stroustrup, B. "What is Object-Oriented Programming?" *IEEE Software*, May 1988.

SNYD93 Snyder, A. "The Essence of Objects: Concepts and Terms." *IEEE Software*, January 1993.

VINO97 Vinoski, S. "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments." *IEEE Communications Magazine*, February 1997.

 Recommended Web site:

- **Object Management Group:** Industry consortium that promotes CORBA and related object technologies