2 - Object Oriented Concepts, Java, and UML - A Short Tutorial

2.1 INTRODUCTION
2.2 BASIC OBJECT CONCEPTS
2.2.1 Object: The Basic Building Block
2.2.2 Messages: Activating Objects
2.2.3 Classes
2.2.4 Inheritance
2.3 OBJECT BASED VERSUS OBJECT ORIENTATION
2.3.1 Abstraction
2.3.2 Encapsulation (Information Hiding)
2.3.3 Polymorphism
2.4 OBJECT ORIENTED USER INTERFACES
2.5 OBJECT ORIENTED DESIGN AND PROGRAMMING
2.6 OBJECT ORIENTED DATABASES
2.7 OBJECT-ORIENTED MODELING
2.7.1 Object Oriented Analysis -- Building The Object Model
2.8 OBJECT MODELING BY USING UNIFIED MODELING LANGUAGE (UML)
2.8.1 What is UML
2.8.2 Importance of UML
2.8.3 Some important features of UML
2.8.4 UML Summary
2.9 A QUICK TOUR OF JAVA AND JAVA APPLETS
2.9.1 Java Overview
2.9.2 Java Applets Versus Java Applications
2.9.3 Key Java Features
2.9.4 Java Security Concerns
2.10 JAVA PROGRAMMING AND DEVELOPMENT ENVIRONMENTS
2.10.1 Getting Started
2.10.2 An Applet
2.10.3 Example of a Java Application
2.10.4 Java Development Tools and Environments
2.11 JAVA IN WEB ENVIRONMENTS -- A CLOSER LOOK AT JAVA APPLETS
2.11.1 Java-enabled Web Browsers
2.11.2 A more significant Java Applet
2.11.3 An Applet/Application combination
2.11.4 More Complex Applets
2.11.5 Java Servlets
2.11.6 The Java Virtual Machine (JVM)
2.11.7 Differences among JVM Implementations
2.12 STATE OF THE PRACTICE: GENERAL OBSERVATIONS
2.13 SUMMARY
2.14 ADDITIONAL INFORMATION

2.1 Introduction

At present, we are in the midst of object orientation (OO), with names like OO programming, OO design, OO databases, OO user interfaces, and so on. Most new software being developed at present and in the future will use some level of OO (at least in the user interface displays). The purpose of this very short tutorial is to give you the basic OO concepts and introduce you to the core OO technologies that are of relevance to this book. In particular, our objective is to answer questions such as the following:

- What are the underlying object concepts (Section 2.2)?
- What exactly is object orientation and how does it differ from object-based systems (Section 2.3)?
- How are OO concepts used in designing user interfaces (Section 2.4)?
- What is object oriented design and what type of OO programming languages are available to implement these designs (Section 2.5)?
- What are object oriented databases and how do they differ from the current genre of relational databases (Section 2.6)?

2.2 Basic Object Concepts

The basic object concepts comprise of object, message, class, and inheritance (see the side bar "Key Object Oriented Concepts"). Let us use an example of cars to quickly illustrate these concepts (programming type details can wait).

- A Buick car is an object. The attributes of this object are model, year, color, etc. The behavior of a car is given by start, stop, go-left, go-right, forward, reverse, etc. These are the methods of the object.
- Turning the ignition key and putting the car in reverse gear represent the messages which invoke the methods of start and reverse, respectively.
- All cars have similar properties. For example, Buicks, Toyotas, and Hondas have similar properties. Car can be used to represent the common properties of automobiles. Car is a class. Buicks, Toyotas, Hondas and so on are instances of class car.
- Classes can inherit common properties from other classes. For example, a vehicle class can be used to represent the common properties of classes such as cars, trucks, buses, and taxis. Vehicle is the superclass from which subclasses such as cars, taxis, trucks, and buses inherit common attributes (model, year, color) and methods (start, stop, go-left, go-right, forward, reverse). However, some properties are unique to each subclass and are not inherited (e.g., taxis have meters).

These key concepts can be of great value in developing reusable and flexible software systems. Let us describe these concepts in more detail.

Key Object Oriented Concepts

What are the Basic Object Concepts?

Object: An object is a piece of data surrounded by code (i.e., the data can only be accessed through code). This code is known as methods of an object. The data has certain attributes (e.g., name,

address, age) and the methods are used to operate on these attributes.

Messages. Objects communicate with each other through messages. A message invokes a method of the target object (the object internals are known to the method and not to the object).

Classes: A collection of like objects makes up a class. A class acts as a template for similar objects (e.g., a class representing all Chevrolets). An object is an instance of a class.

Inheritance: Objects can inherit properties from other objects. Technically, inheritance allows you to create subclasses from parent classes. Subclasses inherit properties from parent classes. Subclasses can inherit multiple properties from multiple parent classes. .

What is Object Orientation?

The following classic definition of object orientation is given by [Wegner 1987]:

Object oriented = objects + classes + inheritance

A more generalized definition, more suitable for distributed systems, is given by [Nicol 1993]:

Object oriented = encapsulation + abstraction + polymorphism

Encapsulation: The restriction of access to the object state via a well-defined interface (the operations). This involves a combination of two aspects: the grouping of object state and operation, and data hiding.

Abstraction: The ability to group associated entities according to common properties; for example, the set of instances belonging to a class.

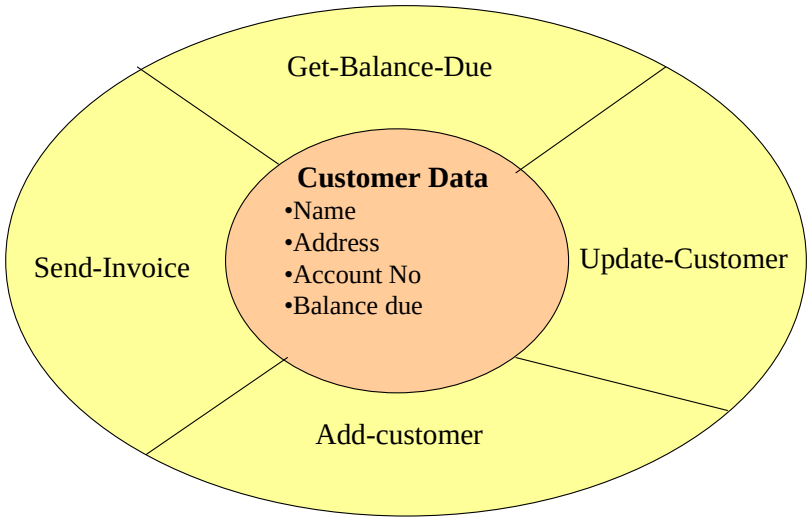
Polymorphism: The ability of abstractions to overlap and intersect. A popular form of polymorphism is inclusion polymorphism in which operations on a given type are also applicable to its subtype (for example, start a printer will start all types of printers). Inclusion polymorphism is often implemented via an inheritance mechanism.

Wegner, P., "Dimensions of Object-Based Language `Design", SIGPlan Notices, Vol. 22, No. 12, Dec. 1987, pp. 168-182.

Nicol, J., et al, "Object Orientation in Heterogeneous Distributed Computing Systems", IEEE Computer, June 1993. pp. 57- 67. ;

2.2.1 Object: The Basic Building Block

An object, according to the Webster's Dictionary, is something mental or physical toward which thought, feeling, or action is directed. In computing, an object is something that can be represented in computer memory. Examples of objects are a program variable, a robot, a car, an employee, a factory, and a company. Different objects can be defined in different problem areas. In distributed computing, for example, each file, printer, spreadsheet, program, user, communication line, workstation, and device can be viewed as an object.



•Customer object is customer data surrounded by methods
•The attributes of customer object are name, address, Account No, Balance due
•The methods on the customer object are Send-Invoice, Add-customer, etc.

Figure 2-1: Conceptual View of a Customer Object

Specifically, objects are data guarded by a protective layer of code. Figure 2-1 shows the conceptual view of a customer object that is the key to understanding object concepts. Basically, an object is represented by at least two properties:

- Attributes
- Methods

The data represents the attributes of the object and the code that surrounds the data represents the methods of the object. Methods are the only way any outsider can interact with the object (you cannot directly view or update customer balance without invoking the "Get-Balance-Due" or "Update-Customer" methods). Thus the outside world does not know how the customer data is internally stored (this leads to information hiding).

Attributes uniquely identify an object. For example, attributes of an employee object are name, social security number, etc. A method shows the object behavior (what it can do or what can be done to it) and contains the code (a list of detailed instructions) that define how the method will respond when invoked. A method hides the implementation details from the users of an object. A method typically receives a message, performs some operations, and sends the response back. An object essentially is a collection of attributes and the valid methods that manipulate the data elements.

Objects can represent small entities such as an icon on a display terminal or large business entities such as customers. For example, consider the following object definitions:

1. Object Name = invoice

   Attributes = customer name, items purchased, price per item, total invoice, etc.

   Methods = prepare, send, review status, update status

2. Object Name = employee

   Attributes = employee name, employee address, employee pay

   Methods = add employee, retrieve employee, update employee

3. Object Name = workstation

Attributes = workstation type, vendor name, OS used, assigned to, etc.

Methods = view, reassign, change operating system

4. Object Name = supplier

Attributes = supplier name, address, items supplied, cost per item

Methods = view, update items supplied, update item cost .

An object can be defined in a program or in a database stored on a permanent (persistent) medium such as disks. The objects in a database are referred to as persistent and the ones in a program storage are referred to as non-persistent. The data access is the same if the objects are non-persistent or persistent.

2.2.2 Messages: Activating Objects

Objects interact with each other through messages. These messages invoke particular methods. A message is simply the name of an object followed by the name of an appropriate method. In addition, the message may contain a set of parameters that the invoked method needs. Figure 2-2 illustrates two messages sent to the customer object, one with parameters the other without. The "send invoice to customer Donna" message does not have a parameter. However, the "get balance due for customer Pat" does have a parameter (date of 08/04/01). We are at present not following any programming language syntax to describe these messages (if you have not guessed, different programming languages use different syntaxes).

Each method of an object knows the list of messages to which it can respond to and how it will respond to each. The object that receives the message is responsible for providing the code that is needed for the invoked object. The receiver should also advertise its methods and the parameters to be invoked for each method so that the outside world can take full advantage of objects. In distributed systems, the advertised methods are known as interfaces and are typically defined by using Interface Definition Languages (IDLs). The sender of the message is responsible for sending the appropriate message and processing the response, including error codes, status codes, etc. In particular, the sender should know that the way a receiver object responds to a message may be affected by the value of its attributes.

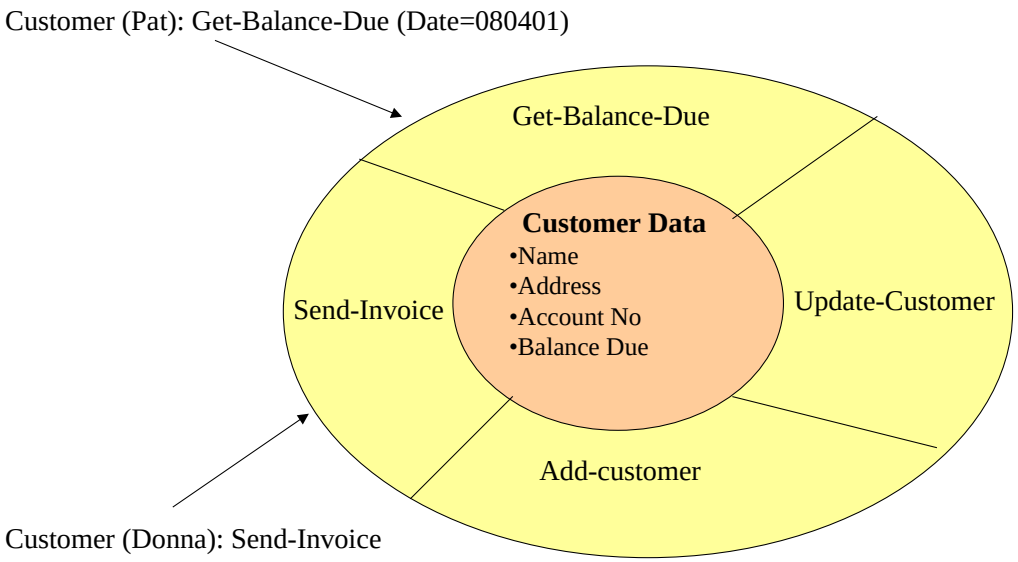


Figure 2-2: Messages to Customer Object

2.2.3 Classes

A class is a template which represents the properties that are common among similar objects. The basic purpose of class is to define a particular type of object. For example, the class employee can be used to represent the common properties of all company employees. The common properties include attributes such as name, address, employee id, grade, pay scale, etc. and methods such as view, update, terminate, etc. So, if in a programming system you have identified object types such as employees, customers, and products, then you will define three classes to correspond to these object types. Each class will define the properties of each object type.

The objects belonging to a particular class are known as instances of that class. For example, the objects Joe Smith, Harry Kline, and Pat Hemsath are instances of the class employee. Once you have created a class, you can create any number of instances of each class (object oriented programming languages provide statements to create instances). The instances contain the information that makes them unique. For example, the object instance of Joe Smith will have information unique to Joe Smith.

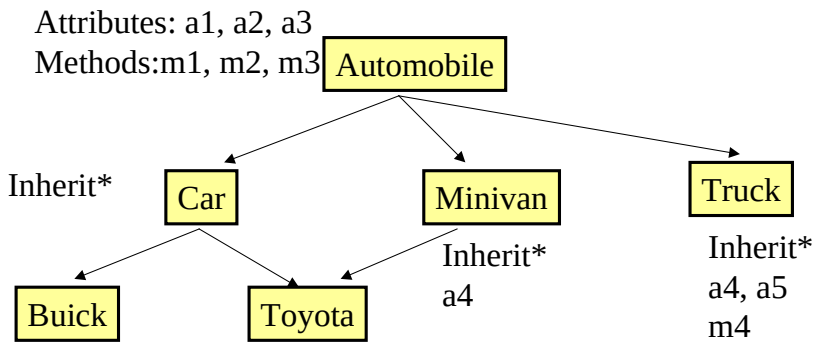
From a programming point of view, a class is similar to a structure that shows the layout of, say, an employee. Objects are instantiations or initializations of the structure to represent different employees.

2.2.4 Inheritance

Classes can be defined independently of each other. For example, the employee, customer, and product classes can be defined independently of each other. However, classes can inherit common properties from other classes. The properties to be inherited can be attributes as well as methods. A class hierarchy can be constructed where lower level classes (subclasses) inherit the properties from higher level classes (superclass). Figure 2-3 illustrates a class hierarchy for automobiles with lower level classes for cars, minivans, and trucks. Naturally, all automobiles have common properties but there are differences between cars, minivans, and trucks (e.g., number of seats, number of wheels). We can set up a class hierarchy for our employee class with subclasses such as managers, technical support, administrative support, consultants, visiting residents, and part-time staff.

A subclass can possess unique (specialized) properties which are not inherited from any class. For example, the resident visitor object has a predefined termination date (some others may also have a predefined termination date but they never know!). Inheritance comes in two flavors: single inheritance in which an object can only inherit from one class, and multiple inheritance in which an object can

inherit from many classes. Multiple inheritance allows creation of new objects from existing objects. Inheritance is based on the "is-a" relationship in artificial intelligence (see Figure 2-3).



- Class: object type (e.g., car, buick, minival)
- Object: instance of a class (e.g., my car)
- Class hierarchies support inheritance (inheritance can be from multiple classes)
- Inherit* means that all properties (attributes plus methods) are inherited from higher classes
- The lower classes can add own attributes and methods after inheriting
- Class library shows the attributes and methods (code) for certain domain (e.g., hospitals)
- Developers can build a system by inheritance

Figure 2-3: Class Hierarchies

2.3 Object Based Versus Object Orientation

The basic object concepts can be used to represent powerful programming systems. An object based system is defined as following [Chin 1991, Taylor 1994]:

Object- based = objects + classes

For example, an object-based programming language supports objects and classes as a language feature but does not support the concept of inheritance. ADA and Modula-2 are examples. Object-based concepts are quite general and can be found in application programs that existed before object technologies became fashionable. For example, many good structured programs developed in the 1970s decomposed each application into modules (i.e., objects) where each module hid the internal information and had well defined procedures that could be invoked from other modules. In addition, many such programs used templates of data definitions which were stored in libraries and used by different programs (this is somewhat similar to classes). Thus many programs could claim to be object-based.

Our interest is in object oriented systems that go beyond object-based systems. The following widely accepted definition of object orientation is given by [Wegner 1987]:

Object oriented = objects + classes + inheritance

Thus the object oriented programming languages (OOPLs) support inheritance. OOPLs support classes, objects, methods, messages, and inheritance (note that methods and messages are implicit in this definition). This definition works very well for programming language design. However, it is not well suited to distributed systems where objects could be stored on different machines in different formats. A more generalized definition, more suitable for distributed systems, is given by [Nicol 1993]:

Object oriented = encapsulation + abstraction + polymorphism

Let us review the three ingredients of this definition of OO. As we will see, this definition is quite similar to the Wegner's definition, albeit more general.

2.3.1 Abstraction

Abstraction focuses on the outside view of an object, i.e., how it will appear to the outside world. The following definition of abstraction has been given by [Booch 1994]:

> "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer".

For example, in an inventory control system of a grocery store, you may abstract different categories of food items into different objects (e.g. eggs, fruits, frozen foods, and juices are identified as different objects). This abstraction is based on the observation that these food items have different characteristics that distinguish them from one another. For each object, you can assign the necessary attributes and methods that outsiders (e.g., the store manager) need to know. This is one abstraction. Another abstraction may decompose the frozen foods and fruits to smaller objects.

Thus abstraction allows us to group associated entities according to common properties; for example, the set of instances belonging to a class. Abstraction represents the object properties without attention to implementation details. Methods as well as messages support abstraction by focusing on the outside view of an object.

2.3.2 Encapsulation (Information Hiding)

Encapsulation, also known as information hiding, prevents the clients from seeing the internals of an object. Encapsulation complements abstraction -- encapsulation focuses on hiding the internals while abstraction focuses on presenting an external view. Encapsulation has been defined by [Booch 1994] as following:

> "Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics".

Encapsulation as well as abstraction are important to the design of an OO system, especially when the objects can be at different sites. In particular, each object must have two parts: an interface and an implementation. The interface of an object only captures its external view (i.e., the object name, the method to be invoked, the parameters to be passed). The implementation of an object contains the mechanisms that achieve the desired behavior (i.e., the code for each method). The interface is also known as the public part of an object and the implementation is known as the private part of an object.

Interfaces are becoming a popular construct in modern programming languages. For example, Java supports an Interface statement. The following segment shows an example of Java Interface specification codee:

```
interface customer   { /* interface name is customer */
 Operation create_cust  (/*The  operation is create_cust */
   char customer_info;     /* input parameter */
   integer status )  /*  output parameter */
 Operation view-cust ( /*The operation is view_cust  */
   char cust-id;  /* input pparameter is cust-id */
   char customer_info;    /* output parameter 1*/
   integer status )  /*  output parameter 2 */
 }
```

In distributed systems, interfaces and implementations play an important role because the interface definitions of an object are created by using an interface definition language (IDL) and stored in directories. These directories are accessed by remote objects to send messages to each other. Different type of IDLs are provided by different middleware packages such as Open Software Foundation's Distributed Computing Environment (OSF DCE) and Object Management Group's Common Object Request Broker Architecture (OMG's CORBA). For example, the following IDL statements could be used to specify the interface for a customer object that supports three operations: create_customer, view_customer, and delete_customer (this IDL is a simplified version of actual IDLs): :

```
interface customer    /* interface name is customer */
 Operation create_cust  (/*The  operation is create_cust */
   [in]   char customer_info;     /* input parameter */
   [out]  integer status )  /*  output parameter */
 Operation view-cust ( /*The operation is view_cust  */
    [in]   char cust-id;  /* input pparameter is cust-id */
   [out]  char customer_info;    /* output parameter 1*/
   [out]  integer status )  /*  output parameter 2 */
Operation delete_cust  (/*The  operation is delet_cust */
   [in]   char customer_info;     /* input parameter */
   [out]  integer status )  /*  output parameter */
```

This interface definition is the rresult of an abstraction and shows the three operations (methods) and the parameters for each method. This interface restricts access to the customer object state via a well-defined interface that does not reveal any internal details about the customer object (i.e., information hiding). Encapsulation and abstraction lead to distinct border, well-defined interfaces and a protected internal representation.

2.3.3 Polymorphism

Polymorphism is a Greek term meaning "many forms". Polymorphism allows you to hide alternative procedures behind a common interface. Basically, you can use the same method name in more than one class. A popular form of polymorphism is inclusion polymorphism in which operations on a given type are also applicable to its subtype (for example, start a printer will start all types of printers). Inclusion polymorphism is often implemented via an inheritance mechanism.

Polymorphism is defined as the quality or state of being able to assume different forms [Taylor 1994]. Polymorphism can be displayed in messages and/or objects. An example is send the same message, "delete", to different objects which respond to it differently. For example, some objects may completely erase the information while the others may just flag it for erasure at a later stage.

Polymorphism is considered as one of the defining characteristics of object oriented technology. It distinguishes object oriented programming languages from more traditional programming. By allowing the same method name in different classes, you can use the same message (e.g., delete, print, draw) to get different results by simply pointing to a different class at run time. Thus polymorphism takes advantage of inheritance and "dynamic binding" (i.e., deciding at run time what object do you want to communicate with).

2.4 Object Oriented User Interfaces

The user interfaces provide the "look and feel" for the services provided by an application. The user interfaces have been by far most influenced by object orientation. At present, almost all new applications provide user interfaces that are either GUI (graphical user interface) or OOUI (object oriented user interface).

GUI provides graphic dialogues, menu bars, color, scroll boxes, pull-down and pop-up windows. GUI dialogs use the object/action model where users can select objects and then choose the actions to be performed on the selected objects. Most GUI dialogs at present are serial in nature (i.e., perform one operation at a time). GUIs are currently very popular and are used in Microsoft Windows 3.X and OSF Motif applications. In addition, the Web browsers use the GUI user interfaces.

OOUIs are highly iconic, object oriented user interfaces to support multiple, variable tasks whose sequence cannot be predicted. Examples include multimedia-based training systems, executive and decision-support systems, and stockbroker workstations. OOUIs are supported by commercial products such as NextStep, OS/2 Workplace Shell, and Macintosh.

Although GUIs and OOUIs are very fashionable at present, we should not forget the non-GUI/OOUI clients that generate server requests with a minimal amount of human interaction. Examples of such clients are daemon programs, barcode readers, robots, automatic teller machines, cellular phones, fax machines, intelligent metering equipment and automated testers.

Discussion of GUIs/OOUIs with numerous examples can be found in [Orfali 1994].

2.5 Object Oriented Design and Programming

Object oriented systems are intended to maximize information hiding (through methods) and reusability (through inheritance). The following steps are used to design a system by using an object oriented paradigm [Booch 1994, Booch 1986]:

- Decompose a given system in terms of objects and not the operations (functions) performed.
- For each object, identify the attributes and the methods. Make sure that the methods hide internal information.
- Develop an initial hierarchy of classes.
- Enforce reusability by inheriting attributes and methods from existing systems. Attempt to maximize inheritance.
- List the messages which invoke methods. These messages establish relationships between objects.
- Develop a prototype solution approach which maps each object into a module. Some of these modules may be implemented as program modules or as database objects (see Section 10.6 for a discussion of object oriented databases). .
- Evaluate and refine the design iteratively.

The object oriented design produced by these steps is translated into code by using object oriented or object-based programming languages. Object-based programming languages such as ADA and Modula support objects as a language feature but do not support the concept of inheritance. Our interest here is in the object oriented programming languages (OOPLs) which support inheritance. OOPLs support classes, objects, methods, messages, and inheritance. Other properties supported by OOPL include abstraction, encapsulation, and polymorphism.

Several object oriented programming languages have been introduced over the years. The first object oriented language, Simula, was introduced in the 1960s. Since then, OOPLs have evolved as follows:

  - Extensions of procedural languages to support inheritance and other features of OOP. Examples are C++ [Stroustrup 1986], Objective C [Pinsen 1991], and Object Pascal [Pascal 1989].
- Extensions of AI languages to include object oriented concepts. Examples are Lisp extensions such as Loops and the Common Lisp Object System [Alpert 1990].
- Languages which are fundamentally based on object oriented concepts and provide basic building blocks for object oriented systems. Eiffel [Meyer 1988], Smalltalk-80 [Goldberg 1995, Goldberg 1989], and Java [Hoff 1996] are examples.

An early review of some of these and many other OOPLs can be found in the OOPL survey by [Saunders 1989]. More recent reviews can be found in [Meade 1995, Pancake 1995]. For continued developments in OOPL, the Journal of Object Oriented Programming Languages should be consulted.

The main advantages of object oriented programming and design are:

- Most changes in software systems are related to objects in real life. Thus object oriented systems are easier to maintain.
- New objects can be created from existing objects by using inheritance thus allowing clustering of similar objects and reducing the complexity of the system.
- The modules can encapsulate (i.e., hide internal details) by allowing external objects to invoke appropriate methods (the external objects invoke methods but do not know how the methods are implemented).

The combined effect is that OOPL can improve code reusability and maintainability.

2.6 Object Oriented Databases

Simply stated, object oriented databases allow storage and retrieval of objects to/from persistent storage (i.e., disks). Object oriented databases, also known as object databases, allow you to store and retrieve non traditional data types such as bitmaps, icons, text, polygons, sets, arrays and lists. The stored objects can be simple or complex, can be related to each other through complex relationships, and can inherit properties from other objects. Object oriented database management systems (OODBMS), which can store, retrieve and manipulate objects, have been an area of active research and exploration since the mid 1980s. The initial work in OODBMSs was driven by the computer aided design and computer aided manufacturing (CAD/CAM) applications [Spooner 1986].

Relational databases are suitable for many applications and SQL use is widespread. However, it is not easy to represent complex information in terms of relational tables. For example, a car design, a computing network layout, and software design of large scale systems cannot be represented easily in terms of tables. For these cases, we need to represent complex interrelationships between data elements, retrieve several versions of design, represent the semantics (meaning) of relationships, and utilize the concepts of similarities to reduced redundancies.

The commonly known object oriented database management systems (OODBMSs) have been developed to support applications in computer aided design and computer-aided manufacturing (CAD/CAM), expert systems, computer-aided software engineering (CASE), and office automation. Simply stated, OODBMSs combine and extend the features of database management systems, artificial intelligence, and "object oriented programming" for these and other applications.

OODBMSs and RDBMSs both have their strengths and weaknesses. For example, RDBMSs are very mature and heavily used but cannot handle complex objects. OODBMSs, on the other hand lack the maturity and ease of use offered by the RDBMSs. A compromise, known as Object-Relational Databases, provide a hybrid solution where relational and object-oriented technologies are combined into a single product. Different vendors use different approaches to Object-Relational Databases. For

example, Odaptor from HP uses an underlying relational database with OO front-ends while UniSQL from UniSQL is an OO database that subsumes the relational model. Detailed discussion of the Object-Relational Database Managers can be found in [Davis 1995].

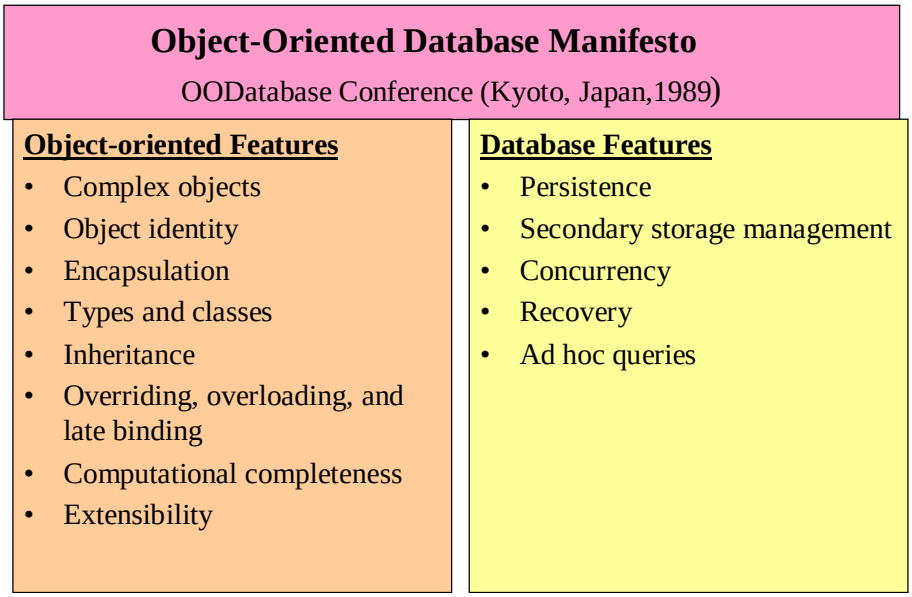


Figure 2-4: The Object Oriented Database Manifesto

Not everybody has always agreed on exactly what a OODBMS is. The debate over defining an OODBMS has continued since the mid 1980s. In 1989, a group of computer scientists got together and established "The Object-Oriented Database Manifesto" [Atkinson 1989]. This Manifesto, displayed in Figure 2-4, establishes the basic properties of OODBMS by combining conventional database functionalities with object-oriented functionalities. We have discussed most of these properties in our previous discussions. Let us highlight the most significant properties of OODBMSs in terms of the following features mentioned in the Manifesto:

Complex Objects. Data may be stored, retrieved and manipulated as complex objects which consist of sets, lists, arrays or relational tuples. For example, a relational table represents a simple object while a composite of many tables represents a complex object. A complex object may represent a plant which consists of simple objects such as personnel, buildings, and equipment. In addition, many plant objects can be combined to form a corporation object, and so on. OODBMSs provide data definition and manipulation facilities for complex objects.

Inheritance. OODBMSs allow creation of objects from existing objects by using inheritance of properties. This greatly simplifies the description of complex data. A DDL, for example, would allow creation of a new object which inherits its properties from existing objects in the database. Single or multiple inheritances may be used.

Procedural Encapsulation (Passive and Active Databases). Procedures can be stored as objects in the database. These procedures can be used as methods to encapsulate object semantics. Two types of object-oriented databases are commonly discussed: passive databases which only store the data attributes and active databases which store the data plus the methods associated with the object. Passive databases represent a more conventional view of the database, where the data is stored in the databases and the procedures are embedded in the programs. Procedural encapsulation implies active databases where the code associated with the object is stored in the database. Active databases have the attractive

feature that entire systems can be stored and retrieved as objects. Gemstone, Starburst, and POSTGRES are examples of active OODBMSs (see discussion later about these DBMSs).

Links. The relationships between objects can be complex, many to many relationships. In OODBMS, objects are related to other objects through relationships which carry semantic information. The syntax of relationships is much more convenient than the relational joins. For example, relationships between objects can be assigned names such as SUBPART_OF, CREATED_BY, COMPONENTS_ARE, DOCUMENTATION_IS, etc. The DDL allows definition of such relationships and the DML allows retrieval and manipulation of objects by using these relationships. For example, an OODBMS query could say: RETRIEVE SUBPARTS_OF CAR.

Multimedia data. Objects can contain very large values to store pictures, voice or text. Most OODBMS provide facilities to store and retrieve multimedia data. In some relational DBMS, multimedia data is referred to as BLOB (Binary Large Objects). A BLOB appears as any other object in the database and can be retrieved and displayed. This allows integration of multimedia applications around a database.

Versions. Most OODBMSs provide facilities to track multiple versions of an object. Many versions can be linked with one object. A user can issue queries such as retrieve all versions of design and retrieve the documentation associated with version 3.1 of the design.

Integration with programming languages. In many OODBMS, the data manipulation language is closely related to the programming language. In some object-oriented systems, it is difficult to say if a system is a database or a programming system. This leads to powerful DML capabilities such as use of AI and pattern matching for data manipulation. In addition, same operations can be used to operate on persistent or non-persistent data. For example, let us assume that we need to evaluate the expression C=A+B. In conventional programming systems, we will use a statement such as “C:=A+B;” in a program if A, B, and C were all in main memory (not-persistent). But if these three variables were on persistent storage, say a relational table, then different statements (e.g., SQL Select) would be needed before the addition. Thus the language syntax depends on where the data is. OODBMS programming languages attempt to eliminate this difference by providing a common syntax independent of where the data is: non-persistent, persistent, or remotely located.

Categories and Examples. The OODBMS systems generally fall into two categories:

- Extensions of the object-oriented programming languages (OOPL) to include the features of DBMS. Examples are O2 and Gemstone systems.
- Extensions of the relational DBMS to include the features of OOPL. Examples are Starburst and POSTGRESS.

OODBMS have moved from state of the art to state of the market, however they are not heavily used at the time of this writing (less than 5% of corporate data is stored in OODBMS). Among the earliest OODBMS are Gemstone (Servio Logic Corp) and Vbase (Antalogic Corp), announced in 1988. Objectstore is one of the most popular OODBMS. It is beyond the scope of this book to discuss detailed features of existing OODBMS.

Different databases are used for different applications in a distributed computing environment. As stated previously, relational database technology is state of the market and state of the practice. SQL, the query language for relational DBMS has become a de facto standard for enterprise-wide data access, even for non-relational data sources. However, relational DBMSs are not suitable for many engineering and other emerging applications discussed in the previous section. object oriented DBMSs are state of the market but not state of the practice at the time of this writing.

Additional information about object oriented databases can be found in [Davis 1995, Kim 1990, Kramer 1995, Kroha 1993, Loomis 1995].

2.7 Object-Oriented Modeling

2.7.1 Object Oriented Analysis -- Building The Object Model

2.7.1.1 Capture the Key Requirements

Capturing of key requirements is obviously important, independent of any methodology. In practice, the requirements are defined, refined, expanded, and modified based on additional insights gained through each iteration of the system. The requirements can be broadly viewed in terms of:

- Functional requirements that capture the capabilities to be provided by the application.
- Operational requirements represent a wide range of performance, security, and interconnectivity information.
- Management/organizational requirements establish the budgetary, time, and policy limits within which the application will be engineered/reengineered.

Specification of these requirements somewhat depends on the nature of applications being engineered/reengineered. We will discuss this topic in later modules of this book.

2.7.1.2 Develop an Object Model -- The Core Activity

An object model represents the functional requirements of a system in terms of objects and classes in a tool and implementation independent fashion. The object model is developed by using the following activities:

1. Identify object classes
2. Specify relationships between object classes
3. Define the attributes and operations of object classes
4. Specify interfaces of objects

Consistent and complete notation is needed for representation of knowledge and communication of knowledge gained about the object model. Many CASE (computer aided software engineering) tools are based on notations to generate code. Notations should allow different views to be represented such as logical models, physical models, static semantics, and dynamic semantics. Common notations used in OO analysis and design are due to Booch and Rumbaugh, among others. Although we do not recommend any specific notation (you can choose a notation of your choice), we will use the Booch's notation for illustration (see Figure 2-5).

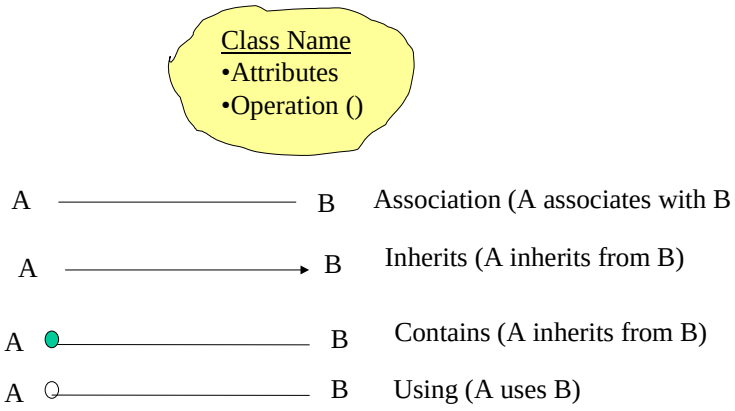


Figure 2-5: Notation Used in OO Analysis (Booch)

Step 1: Identify Object Classes

Development of object classes, commonly known as classes, is at the core of OO analysis and design. Basically, a class is a template which represents the properties that are common among similar objects. For example, the class customer can be used to represent the common properties of all customers. The common properties include attributes such as name, address, customer id, etc. and methods such as add, update, query, etc. The relationship between an object and a class is straightforward -- an object is an instance of a class. For example, if Buick is a class then all Buick cars can be viewed as objects (i.e., instances) of the class Buick. Due to this relationship, many people do not differentiate between objects and classes in this stage of analysis because either way you are specifying the core components of the object model. To avoid confusion, the term object class is used sometime to represent an object or a class (we will use the term class because it is more commonly used).

How do you identify classes? Classes can represent real-world things such as a person, a building, a laptop computer, or a text editor. It can also represent concepts such as schedules, pay scales, or algorithms. We strongly recommend to start with objects that represent business entities and not with programming objects such as GUI interfaces, stacks, queues, buttons and the like (see the sidebar "Business Objects: The Key to Reuse"). Here are some guidelines (see [Fingar 1996, Baster 1997, Gale 1997]):

- In the early iterations, concentrate on business objects (e.g., customer, order) which represent business things and do not get bogged down with programming objects (stacks, queues) in developing an object model.
- Develop the model of your business processes as business objects (business entities with policies as methods). For example, you can represent different business units as objects (i.e., purchasing department, shipping/receiving department)
- Identify objects within the context of an application (the application objects) which can be used by other applications. This helps in reuse.

A wide range of techniques to assist in the intuitive task of identifying classes have been discussed [Booch 1994, Rumbaugh 1994, Rymer 1995, Vasudeva 1997]. Here are the key ideas:

- Start with reviewing the problem description and listing the nouns you encounter (classes often correspond to nouns).
- First focus on the nouns that represent physical objects (people, things, places, devices, machines). It is easier to think of real objects first and postpone abstract things for later iterations.

  - Eliminate spurious classes by getting rid of redundancies (e.g., workers, employees and hired hands mean the same thing), irrelevant classes (e.g., classes that may not have anything to do with the application domain), attributes (e.g., names and addresses are attributes not classes), etc.
- Now extend the classes by adding Information sources/sinks (e.g., databases, files), concepts and events as potential classes if they are relevant to the application domain.
- It may also be advisable to identify the things used by the customers as objects.
- Go through several iterations and refinements by working through the design stage. In many cases, new object classes are discovered and existing classes are eliminated during the design stage.

For database designers, the process of identifying classes is very similar to identifying the entities in an entity-relationship data model.

Business Objects: The Key to Reuse

The main business motivation for OO is increased reuse. The best way to achieve high reuse is to build objects that represent business entities (i.e., business objects). The basic idea of business objects is that the users can construct objects that represent the real-world concepts of the business. Examples of business objects are customer, order, product, and regional office. If software could be structured around such objects and other business concepts, then organizations would be able to build software that simulates current business strategy. Moreover, businesses could reuse these objects to build new applications by using the OO paradigm.

You can represent business processes as objects (business things with policies as methods). For example, you can represent different business units as objects (i.e., purchasing department, shipping/receiving department). Since these objects represent business processes, these objects can change as businesses change. By using the OO concepts of encapsulation and abstraction, you can hide the internal details of business processes and build new business processes from these objects. After the business processes have been modeled as business objects, you can build applications by implementing business objects into lower level technical objects (e.g., GUIs, databases, etc.).

Business objects started appearing in the marketplace around the introduction of OLE 2.0, OpenDoc, and CORBA-based products in 1994. Since then, OO tools designed to support creation of business objects have emerged from vendors such as Easel and applications that employ business objects have appeared from vendors such as SAP and IMRS. The Object Management Group (OMG) has found the Business Object Management Special Interest Group (BOMSIG) to help the industry understand and utilize this technology effectively. System integrators such as Anderson Consulting are also committing to deliver software environments that can be used to build custom applications based on business objects.

Business objects area t the core of "Class-Based Reengineering (CBR)" that integrates business process reengineering with object technologies [Newman 1996]. This approach starts out by building a business model in terms of object classes and then uses these classes to architect and build systems. The object classes can represent different business entities such as customers, loans, accounts, etc. These classes can change to reflect business changes and tie into technology because these classes can be implemented by using OO technologies.

Classes At A Glance

Classes

- Class: set of objects with common properties (structure + behavior)
- Object: instance of a class
- Interface defines external view (i.e., what operations can be performed)
- Interface can consist of:
  Public: available to all classes

  Protected: available to class, subclass, and friends

  Private: available to class and friends only

Class Relationships

- ASSOCIATION: Links between classes.
  One to one

  One to many

  Many to many
- AGGREGATION: Part of (Containment relationships)
- INHERITANCE: inherit properties (data plus methods)
  Single inheritance

  Single polymorphism

  Multiple inheritance
- USING: Refinement of association
  Single direction (Client/server relationship)
- METACLASS: A class of classes
  Instances are classes

Figure 2-6 shows the classes identified in a customer information system in which customers buy a variety of products such as TVs, radios and their parts. We will expand this example as we go along.

Identification of classes in large scale problems is an extensive undertaking (you can easily come up with hundreds of classes). Most methodologies for identifying object classes utilize English descriptions (nouns are classes/objects) and “use-cases” that exercise various scenarios. Many start with domain analysis, i.e., rely on domain experts to help in identifying the classes and their behaviors. The domain analysis procedure for identifying classes starts with abstracting the things that are used in the domain and proceeds to representations of real-world objects such as an employee, machine, product, facility, or an organizational unit (e.g., a department). In addition, groups of users can be modeled as an object class. You may use the following guidelines:

- Develop a strawman general model of the domain by consulting with domain experts
- Examine existing systems within the domain before concentrating on the new system
- Note the similarities and differences between the existing and new systems
- Keep the classes as close to the notion of "business objects" as possible. Make sure that the classes you are defining can be used by other applications.
- Keep the following "implementation" issues in mind: First, calls should not be modeled as objects because they are specific to implementations. Second, “size” of objects (in terms of the number of

interactions) should be considered carefully. If objects are too small and involve too many interactions then too much intersystem communication can result. However, if objects are defined too broadly, then the memory requirements to handle such objects can be very large.

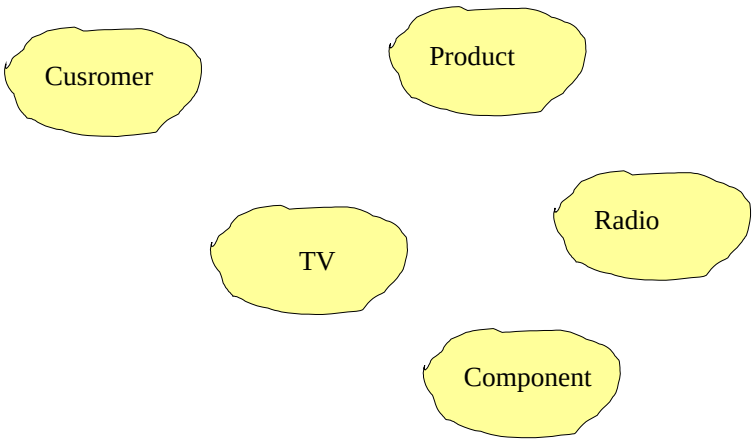


Figure 2-6: Customer Information Object Model -- Classes

The output produced by this step is a list of classes that may be stored in a dictionary for later use. Additional details to these classes will be added later. We will see later that once you have identified a class, then you can easily translate it into code and can create any number of instances (i.e., objects) of each class (object oriented programming languages provide statements to define classes and create objects).

Step 2: Specify Relationships (Associations) Between Object Classes

Relationships, also known as associations, between object classes show how the objects of an application collaborate with each other to satisfy user needs. Interacting objects collectively describe behavior of a system. Basically, any dependency between two or more object classes is an association. For example, the dependency between an employee and a company is an association (i.e., the works-for association) and the dependency between a customer and a product is also an association (i.e., the purchases association). Associations between objects can be one-to-one, one-to-many, and many-to-many.

As we will see in a moment, associations can be of different types. In the first iteration of building an object model, it is enough to identify associations without categorizing the association types. However, in later stages of an object model, the following types of associations (relationships) play a key role:

- Use relationships
- Contain relationships
- Inheritance relationships

A use relationship between two object classes indicates that the two classes collaborate by sending messages to each other. For example, the statement "sales clerk updates a customer account" shown in Figure 2-7, indicates three pieces of information: 1) the sales clerk class uses the customer account class, 2) an "update" operation is supported by the customer class, and 3) an "update" message is sent from the sales clerk class to the customer account class. Notice the use of notation in this example.

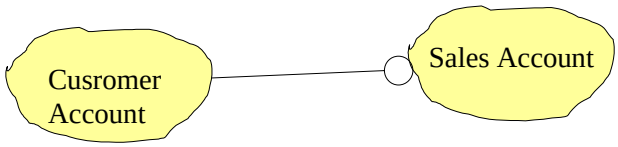


Figure 2-7: A Simple Using Relationship

Use relationships focus on the dynamic behavior of classes and objects. By collecting all the use relations to and from an object, you will have a good idea about behavior of the object and the system requirements. Given a collection of objects involved in use relationships, each object may play one of three roles:

- Actor- an object that can operate upon other objects, but that is never operated upon by other objects.
- Server - an object that never operates upon other objects, it is only operated upon by other objects.
- Agent - an object that can operate on other objects and can be operated on by other objects.

The containing relationship, also known as parent/child relationship, represents subparts of an object class. For example, container relationship exists between a car and the car engine, wheels, seats, etc. Container relationships are very useful in reducing the number of objects that are visible in a system. For example, if an application does not need it, all parts of a car can be contained in a car class. Thus, the details of car parts are encapsulated in the car class. However, containing relationships do lead to coupling between objects that cannot be distributed among machines, if needed.

The inheritance relationships, an important part of object model, represent the common properties (attributes, operations) between classes. For example, all cars have some common properties. The notion of inheritance is borrowed from artificial intelligence "ISA" relationship. For example, "Sun ISA computer" represents that the common properties of a computer are inherited by a Sun workstation. In some cases, only a subset of properties is inherited. This can be represented through "a kind of (AKO)" relationship. For example, "computer is a kind of electronic machine" shows that computer only inherits a small number of electronic machine properties. A class hierarchy can be constructed where lower level classes (subclasses) inherit the properties from higher level classes (superclass). For example, we can set up a class hierarchy for an employee class with subclasses such as managers, technical support, administrative support, consultants, visiting residents, and part-time staff.

Figure 2-8 shows a refinement of the customer information object model we introduced earlier. We have basically added use and contain relationships between the classes and also have added a class “product” from which the TV and Radio classes can inherit properties.

The main activities involved in developing the associations are as following:

- Choose a family of classes that appear to be related
- Identify dependencies between classes (each dependency is an association)
- Associations between classes are typically represented by verbs in a statement (e.g., “customer buys products” identifies an association “buys” between the two classes customer and products)
- Walk through various scenarios to make sure that all associations are identified
- Refine associations into inheritance, containment and use relationships
- Exit when most relationships have been identified

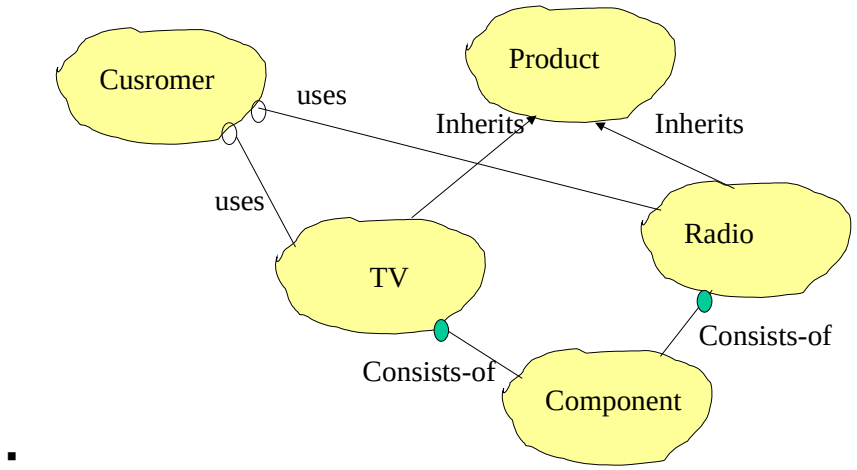


Figure 2-8: Customer Information Object Model - Relationships

Step 3: Identify Attributes and Operations

Attributes are properties of individual objects, such as name, age, weight, height, or color. Attributes should not be used for associations (e.g., do not use product-purchased as an attribute of a customer, use it as an association between a customer class and a product class). It is important to identify only those attributes that are important for the problem domain (i.e., use abstraction principles). For example, if you are designing an airline reservation system, then you only need to define a very few attributes of each city (e.g., number of airports, distance to the main city, etc.). However, if you are designing a travel guide, then many more attributes of each city are needed.

An operation is an activity performed by or on an object. The operations performed on each object are listed as a starting point. In addition, the information associated with each operation (i.e., inputs needed, outputs generated, preceding and/or subsequent activities, sequence restrictions) is specified. You should also list security restrictions associated with the operations being defined. The following guidelines can be used:

- You do not need to represent all operations on an object. You should primarily identify those operations that are needed by the users of the application at hand.
- Relationships between objects are a good starting point to specify the operations that can be performed, and the results that can be expected. For example, a “purchase-product” relationship between a customer and a product indicates that “purchase-product” operation should be supported by the product class.
- The operations external to the current application should only be considered if the external operations change the state of the object. If they do, you must synchronize the activities in order to ensure integrity of object data.

Table 2-1 illustrates the results of identifying attributes and operations for the simple customer information system we have been reviewing. We have shown an object (customer), four operations performed on this object, and the input/output of each operation. We have not specified any pre and post conditions and any other sequencing information that could have been identified here. A method is a set of software instructions that is invoked when an operation is invoked. Thus the customer object must provide four methods (i.e., software instructions) that are invoked when operations such as add a customer, view a customer, update customer information, and delete a customer are invoked. The information produced as a result of this step can be represented by any of your favorite representation tools (text, formal languages, graphical tools, Booch diagrams, etc.).

Table 2-1: The Customer Object (Operations, inputs, outputs)

Operations	Inputs	Outputs
1. Add a customer	Customer Information	Status
2. View a customer information	Account number	Customer information
3. Update a customer information	Account no, new information	Status
4. Delete a customer	Account number	Status

Figure 2-9 shows the customer information object model with attributes and operations identified. Recall that the notation we are using identifies class names with underlines, operations with "()" and attributes just as items. We have only identified a few key attributes and operations.

Figure 2-9 represents a complete object model -- it shows the classes, the relationships between classes, the attributes, and the operations. This object model can be easily translated to code by using OO programming languages (we will see this later).

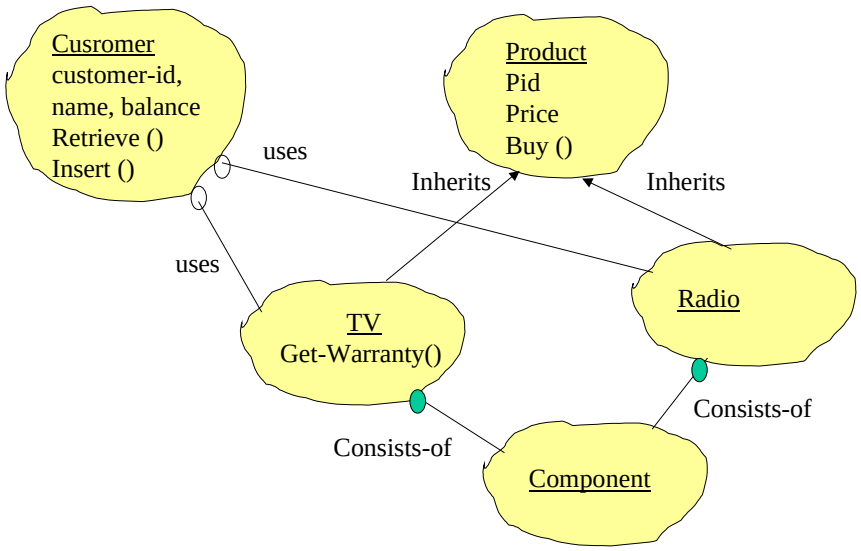


Figure 2-9: Customer Information Object Model -- Attributes and Operations

Step 4: Define Interfaces

Interface of an object is the collection of the operations and their signatures (signature of an operation defines the operation's name, it's arguments, and argument types). For example, the following statements could be used to specify the interface for a customer object that supports two operations: create customer and view customer (the language is illustrative):

```
interface customer    /* interface name is customer */
Operation create_cust  (/*The  operation is create_cust */
  [in]   char customer_info;     /* input parameter */
  [out]  integer status )  /*  output parameter */
Operation view-cust ( /*The operation is view_cust  */
   [in]   char cust-id;  /* input parameter is cust-id */
  [out]  char customer_info;    /* output parameter 1 */
  [out]  integer status )  /*  output parameter 2 */
```

The interface specification at first attempt can be at a high level by omitting some of the information about argument types.

One or more interfaces may be defined for an object. Interfaces are essentially declarations of how an object can be used by other objects. Although interfaces are defined for most object models, interface definitions have become an essential aspect of distributed object applications. This is because distributed objects can reside on different machines with different character and numeric representations. Interface definition languages (IDLs) are provided by middleware (e.g., OSF DCE IDL and CORBA IDL) for defining interfaces formally. The interface definitions are compiled by using vendor provided IDL compilers and can be stored in an Interface Repository for other objects to utilize. We will discuss IDLs in more detail in Chapter 4 of the "Third Generation Architecture" Module. In addition, modern OO languages such as Java provide constructs for interface specifications.

We should note that interface specifications are not typically considered as part of developing an object model at the time of this writing. However, as distributed objects become more popular, the notion of interface specification will become an important aspect of developing the object model. In essence, interface specification formally specifies certain aspects of the requirements of a system and should be considered as part of formal requirement specification -- an area of considerable research activity in software engineering [Siddiqi 1996, Ng 1990, Nicol 1993, Pankaj 1991].

2.7.1.3 Develop Dynamic Model

Dynamic behavior of objects deals with issues such as synchronization of messages and event notification. The objects involved in a using relationships must synchronize the messages. Synchronization is essential for the server object that may need to manage multiple threads of control. In general, objects can interact with each other in two manners:

- Synchronous - the sending object is blocked (suspended) until a response is received.
- Asynchronous - the sending object is not blocked and continues processing. The sending object typically checks for an event that indicates that the initiated interaction has been completed.

Synchronous communication may be sufficient for many real life systems. In addition, many middleware packages only support synchronous communications (e.g., DCE RPC is strictly synchronous). However, it may be desirable to exploit asynchronous interactions for some applications. Asynchronous communications require more effort by programmer's because code must be provided to handle unscheduled events. Asynchronous interactions can be achieved by generating

(local) concurrent synchronous interactions. For example, a client can initiate three "threads" for three synchronous interactions and thus achieve desirable parallelism.

Event handling is a crucial aspect of object-oriented applications. Most GUI applications are event-driven, i.e., they must respond to a large number of mouse clicks, window activations, function keys, key strokes, timers, and various other scheduled as well as unscheduled events. In particular, multimedia applications are intensely event driven with events from remote control, voice input, and mouse [Rosenberg 1995]. It is essential that a comprehensive event model be specified. A wide variety of techniques can be used to specify and analyze events (e.g, directed graphs, message sequence charts, timing diagrams, Petri nets, discrete event simulations). Details about these techniques can be found elsewhere [Booch 1994, Rumbaugh 1995, Rosenberg 1995].

2.8 Object Modeling by Using Unified Modeling Language (UML)

2.8.1 What is UML

UML (Unified Modeling Language) is an OMG (Object Management Group) standard diagrammatic notation used to analyze and design systems using object-oriented concepts. It should be noted that UML is not a modeling method, it is just a modeling notation that can be used with any process e.g., Rational Unified Process.

In general,

Modeling method = Process + Modeling Language

Process = steps that are needed to be followed to arrive at certain solution or design.

Many companies are using the UML as a standard in their development processes and products, which extends to areas such as business modeling, requirements management, analysis & design, programming and testing.

2.8.2 Importance of UML

The main objectives of UML are as follows:

- Making understanding of the problem domain and designing the solution easier and systematic.
- Ease in communicating ideas to other professionals using a standard notation that is independent of any programming language or development platform.

2.8.3 Some important features of UML

According to the UML specifications[1], following diagrams have been defined:

I. Use case diagram

II. class diagram

III. interaction diagrams:
   - sequence diagram
   - collaboration diagram

IV. behavior diagrams:
   - state diagram

[1] OMG Unified Modeling Language Specification, ver. 1.3, June 1999 (ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf)

- activity diagram

V. Implementation diagrams:

- component diagram
- deployment diagram

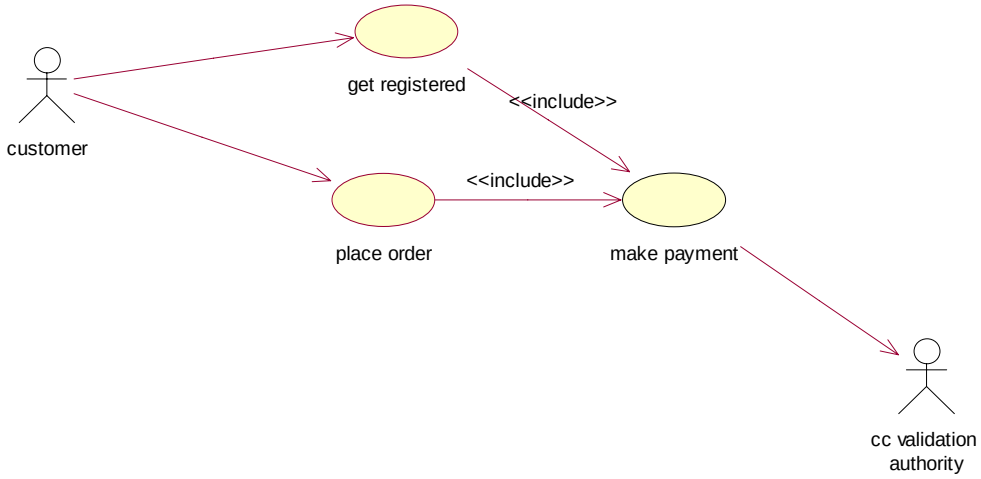
2.8.3.1 Use Cases

A use case is defined as the set of scenarios tied together by a common user goal[2]. It is a high level conceptual view of certain functionality of a system. On the other hand, a scenario is a sequence of interactions between a user and the system for a particular case. It should be noted that a use case is an activity that has a certain important goal and it may involve many steps. Also a use case can have many scenarios. For example, ordering an item on the shopping website is a use case which can have multiple scenarios that involve situations like

No desired item available in inventory, credit card accepted or rejected etc.

Use case diagram is one important aspect of UML and these diagrams can be used to identify objects and their interrelationships.

Example:



Use Case diagrams are divided into two categories:

- Business Level use case diagram

  These diagrams represent the system from the business point of view without worrying about how those business objectives will actually be implemented. This diagram is one level higher than the system use case diagram.

- System Level use case diagram

  In this diagram represents the actual functioning of the system in terms of use cases.

[2] Fowler,M., "UML Distilled", 2nd Edition, Addison-Wesley 2000 pp 40

Diff. notations and terminologies

Base use case -- Base use case is the main use case that has 'included', 'extended' or 'specialized use case.'

Included Use Case --- An included use case is a certain activity that is common among certain big use cases e.g., in the above diagram, making payment is required both for registration as a customer and for purchasing items. So we have made 'make payment' a separate use case for clarity. It could have been a part of the 'place orders' use case other wise.

Relationship between the base and included use case is shown as follows:

<<Include>> with an arrow pointing from base to included use case.

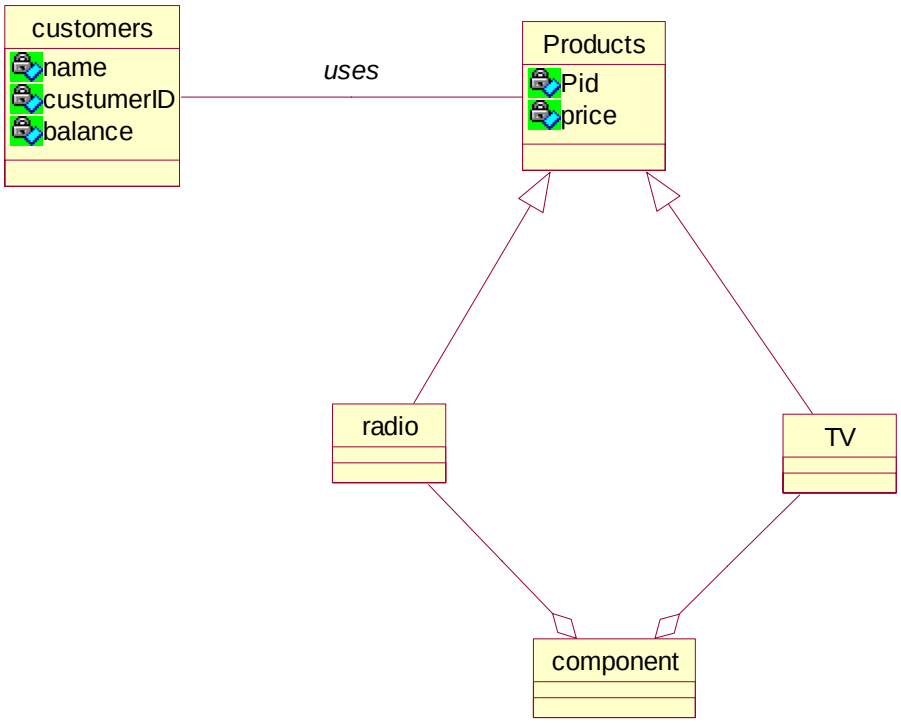
Generalization ---- Generalization is a relationship between a more general element (the parent) and a

more specific element (the child) that adds additional information. It is used for classes, packages, use cases, and other elements and is represented by a closed triangle arrow head pointing towards the parent.

Extended Use case ---- It is a certain activity that extends the basic job in the base use case i.e., it provides additional functionality for some special cases. It is very much like generalization but in this case, extension points are explicitly mentioned. It is represented as <<extend>> with arrow pointing towards the base use case.

2.8.3.2 Class Diagram

Class Diagram can be regarded as the heart of object-oriented modeling. This diagram shows the relationship between different classes and attributes and methods for each class. Class diagrams depict a static view of the system. These diagrams can be drawn at different stages of software development e.g., Conceptual class diagram gives a business level view of different objects of the system i.e., how the system will work. Design, specification and implementation class diagrams are developed at later stage and they reflect a more refined and implementation specific view of the system.



Each of the class identified in the design or implementation diagram is converted into code accordingly. Most of the other diagrams like interaction diagrams, behavior diagrams etc. are used to identify class methods, refining the interrelationships between classes etc.

2.8.3.3 Interaction Diagram

These diagrams are used to understand the functioning of a single use case, what objects are involved in that particular use case and how they are interacting. These diagrams give a dynamic view of the system. There are two kinds of interaction diagrams i.e.,

a) Sequence Diagrams

b) Collaboration Diagrams

Both the diagrams although different in graphical notation, serve the same purpose. Developers may choose any of these according their thinking style.

A sample sequence diagram

2.8.3.4 Behavior Diagrams

2.8.3.4.1 State diagrams

These diagrams shows the behavior of a single object across many use cases ie., it shows the life time states of a single object. This diagram is normally used only for those classes that have complex behavior and the developers or designers want to get a clearer picture of the functioning of that object.

2.8.3.4.2 Activity Diagram

It shows a general sequence of actions in the form of a control structure. Theses diagrams are useful in describing the workflow and the behavior that has lot of parallel processing. In this way, activity

diagrams are useful in multithreaded programming. Activity diagrams are not recommended to understand the collaboration of different objects or Interaction diagrams are better for this purpose.

2.8.3.5 Implementation Diagrams

Implementation or physical diagrams are developed mostly in cases where logical information is appears significantly different from physical information

2.8.3.5.1 Component Diagrams

A component diagram shows different components in a system and their dependencies.

2.8.3.5.2 Implementation or Deployment Diagram

Deployment diagram shows physical relationships among software and hardware components in the delivered system. It shows how components and objects actually function in the system especially in a distributed environment.

UML specifications are being updated and new features are continuously introduced. Some of other features are:

Package diagrams, CRC cards, Patterns etc.

2.8.4 UML Summary

UML has evolved into a powerful modeling language that is at the foundation of several CASE (computer-aided software engineering) environments such as Rational Rose (www.rational.com). UML is being used to model other systems besides software programs. For instance, UML can be used to model databases instead of ER diagrams. Currently, there are many notations available for ER modeling. Adoption of a standard modeling language such as UML might eliminate the confusion due to notational differences.

2.9 A Quick Tour of Java and Java Applets

2.9.1 Java Overview

Java is an object-oriented programming language that is playing a unique role in WWW. The Java programming language and environment was introduced by Sun Microsystems initially to develop advanced software for consumer electronics. Initially, Sun intended to use C++ for these devices that are small, reliable, portable, distributed, real-time embedded systems. It was found that the problems were best solved by introducing a new language that was similar to C++ but drew heavily from other object-oriented languages such as Eiffel and Smalltalk. The language, initially known as Oak, is now known as Java.

Why is Java so popular? The key is in supporting user interactions with Web pages that use Java. Simply stated, small Java programs, called Java applets, can be embedded in Web pages (these are called Java powered pages). Java powered Web pages can be downloaded to the Web client side and make the Web browsers a powerful user tool. Before Java, Web browsers were relatively dumb (i.e., most functionality resided in Web servers not in Web browsers). Java changed all that because Java applets can run on Java enabled browsers. When users access these pages, these pages along with the Java applets are downloaded to the Web browser. The Java applets run on the Web client side thus making the browser an intelligent component. Basically, the Java applets are downloaded to the Web browser site, when the user clicks on the Java powered pages, where they run doing whatever they were programmed to do. There are several implications of this:

- Java applets make Web applications really client/server because the Java code can run business logic on the Web client site (i.e., the Web browser houses the first tier).

  - Java applets exemplify "mobile code" that is developed at one site and is migrated to another site on demand. This introduces several security issues but also creates many interesting research opportunities.
- Back-end resources (databases and applications) can be accessed directly from the browser instead of invoking a gateway program that resides on the Web server site (security considerations may require a “proxy” server on the Web server site). The Java program can ask the user to issue a request and then send this request to back-end systems. A standard called JDBC (Java Database Connectivity) has been developed to allow Java programs to issue calls to relational databases.
- The Web screen layout can be changed dynamically based on the user type. A Java program can determine the user type and modify the screen layout. For example, different advertisements can be shown and highlighted to the user depending on the user characteristics (e.g., age, job type, education level, credit history, salary level).
- You can produce graphs and charts dynamically at your browser instead of fetching predefined graphs and images from the Web server (transferring images takes a very long time over the Internet). For example, I tried to access a home page from Europe. The home page had some image files that were somewhat complicated. I could simply not get the page even after hours of trying (the session timed out repeatedly).
- You can run animations, invoke business transactions, and run spreadsheets at your browser site. In essence, you can run almost any application on your Web browser that can interact with the user, display graphics on Web browser, and interact with back-end databases and applications.

Distributed Applications With Java

If you need to write a Java application where a Java applet on your Web browser invokes another Java applet on another machine, then you have the following main choices:

- Write your own low level code (e.g., TCP sockets) to invoke the remote Java code
- Utilize, if possible, distributed object middleware such as CORBA

The first choice is not very attractive (you are writing your own middleware). The second choice can be pursued through:

- CORBA calls
- DCOM calls
- Sun’s Remote Method Invocation (RMI)

See Section 2.11 for additional details.

2.9.2 Java Applets Versus Java Applications

What is the difference between a Java application and a Java applet? Basically, a Java application is a complete, stand-alone application that uses text input and output. Java applets, on the other hand, are not stand-alone applications and they run as part of a Java enabled browser. From a programming point of view, a Java application is Java code ("Java Class") that has the main () method. The Java interpreter looks for main () and executes it. Java applets do not execute main (). Instead, Java applets contain methods that are invoked by the Java enabled browsers.

A Java applet contains methods (subroutines) to initialize itself, draw itself, respond to clicks, etc. These methods are invoked by the Java enabled browser. How does a browser know to download Java applets. It is quite simple. A Java powered HTML page contains a tag (the APPLET Tag) that indicates the location of a Java applet. When the browser encounters this tag, it downloads it and runs it. See the Section "Downloading and Running Java Applets".

The Java applets are small enough so that they can be imbedded in Web pages but large enough to do something useful. The Java applets are transferred to the Web browser along with everything else imbedded in the Web page (e.g., text, images, video clips). Once transferred to the Web client, they execute on the client side and thus do not suffer from the issues of network traffic between the Web client and Web server. Because these applications run on your client machine, you see a much more natural and efficient execution (imagine running a multimedia application on a remote Web site versus running it on your own desktop).

Due to the popularity of Java applets, many plug-and-play Java applets are already available. Once built, the Java applets can run on many different machines. The Java code is first compiled into byte-codes (byte-codes are machine instructions that are machine independent). The byte-code of the applet is loaded into the browser where it runs efficiently on different machines by using a runtime interpreter. Due to the appeal of Java applet style programming, other programming languages such as C++ and COBOL have started producing byte code that can be invoked by Web browsers (the browsers do not know how the code was created).

Java applets have access to a wide range of libraries that allow Java applets to perform many operations such as graphics, image downloading, playing audio files, and user interface creation (i.e., buttons, scrollbars, windows, etc.). These libraries are included as part of the Java Applet API. This API is supported by all Java-compatible browsers. It is expected that these libraries will grow with time, thus making Java applets even more powerful and diversified.

2.9.3 Key Java Features

Java has emerged as a very popular language for developing Web applications. According to Sun, "Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language". The following paragraphs discuss these features of Java. The following discussion is an abbreviated version of the Java white paper that can be obtained from Sun's home page (http://www.sun.com). Although Java is very popular at present, it is presenting some security concerns (see the section on "Java Security Concerns").

- **Simplicity:** Java was designed to be similar to C++ in order to make the system more comprehensible to current practitioners. Java omits many features of C++ such as operator overloading (although the Java language does have method overloading), multiple inheritance, and extensive automatic coercions. The auto garbage collection was added thereby simplifying the task of Java programming but making the system somewhat more complicated. A good example of a common source of complexity in many C and C++ applications is storage management: the allocation and freeing of memory. By virtue of having automatic garbage collection the Java language makes the programming task easier and also cuts down on bugs. Java is designed so that it can run stand-alone in small machines. The size of the basic interpreter and class support is about 40K bytes; adding the basic standard libraries and thread support adds an additional 175K.
- **Object-Orientation.** The object-oriented facilities of Java are essentially those of C++, with extensions from Objective C for more dynamic method resolution. However, Java is more rigorous.
- **Distributed.** The main power of Java is that Java applications can open and access objects over the Internet via URLs in a manner similar to accessing a local file system. Java has an extensive library of routines for coping easily with TCP/IP protocols like HTTP and FTP.
- **Architecture Neutral.** Java was designed to support applications on networks. Java compiler generates an architecture neutral object file format that is executable on many processors, given

the presence of the Java runtime system. The Java compiler generates byte-code instructions that are independent of computer architecture. Byte-codes are designed to be easy to interpret on any machine and can be easily translated into native machine code on the fly.

- **Portable.** Java specifies the sizes of the primitive data types and the behavior of arithmetic on them. For example, "int" always means a signed two's complement 32 bit integer, and "float" always means a 32-bit IEEE 754 floating point number. The libraries that are a part of the system define portable interfaces. The Java system itself is also portable. The new compiler is written in Java and the runtime is written in ANSI C with a clean portability boundary. The portability boundary is essentially POSIX.
- **Interpreted.** The Java interpreter can execute Java byte-codes directly on any machine to which the interpreter has been ported. And since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.
- **Robust.** Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating error prone situations. Java requires declarations and does not support C-style implicit declarations. The single biggest difference between Java and C/C++ is that Java does not allow pointer arithmetic. Java has arrays that allow subscript checking to be performed. In addition, Java does not allow an arbitrary integer to be converted into a pointer.
- **Multithreaded.** Multithreading is important for performance but writing multithreaded programs is more difficult than writing in the conventional single-threaded programs. Java has a set of synchronization primitives that are based on the widely used monitor and condition variable paradigm.
- **Dynamic.** Java was designed to adapt to an evolving environment. It makes the interconnections between modules later. Java understands interfaces-- a concept that is used heavily in distributed systems through Interface Definition Languages (IDLs). An interface is simply a specification of a set of methods that an object responds to. Interfaces make it possible to use objects in a dynamic distributed environment (we will talk about this when we discuss CORBA).

The best source for additional information about Java is the home page (http://www.javasot.com). From this home page, you can find many white papers on different aspects of Java, Java applets, and Java components (Java Beans, Enterprise Java Beans). At present, more than 100 books are available on different aspects of Java (see, for example, Amazon.com, for the latest books).

Downloading and Running Java Applets

The Java applets are downloaded and executed on the Web browser by using the following steps:

- User selects an HTML page
- Browser locates the page and starts loading it.
- While loading, it starts to format text
- It loads graphics if indicated by IMG or FIG tags in HTML
- Java applets are indicated by an APPLET tag. For example, the tag indicates a Java applet called "myapplet.class" that is run in a window size of 110 by 150:

```
<APPLET CODE=myapplet.class WIDTH=110 HEIGHT=150>

</APPLET>
```

- The applet code is assumed to be on the same site where the HTML page is

- Browser loads the indicated class and other needed classes
- Java "enabled" browsers also keep local classes that may be used by the applets
- After the applet has been loaded, the browser asks it to initialize itself by invoking the init() method and draw a display area that is used for input/output.

2.9.4 Java Security Concerns

Several security flaws in Java are currently being discovered and addressed. The basic premise of the security concerns is that Java applets are essentially foreign applications that are brought into your environment and executed on your browser site. Such programs can contaminate your environment.

Java designers have taken reasonable precautions about Java security by introducing a Java verifier to make sure that the byte code was generated by a valid Java compiler before running it (Java compilers restrict pointers and typecodes to minimize security risks). However, several security flaws in Java are currently being discovered and addressed. The basic premise of the security concerns is that Java applets are essentially foreign applications that are brought into your environment and executed on your browser site. This opens the floodgate to unscrupulous code being brought in from other sites and do strange things. A quick remedy to this problem is to make sure that you download Java applets from trusted sites only (e.g., corporate Web servers within your firewalls).

The examples of how Java programs can contaminate your environment abound. For example, David Hopwood at Oxford University found that Java applets can load malicious class files and libraries onto a user's system. Many "hostile applets", such as the following, have been documented and are listed on the "Hostile Applet Home Page":

- A noisy bear who refuses to be quiet.
- A barking browser
- Popping up an untrusted applet window
- Forging email
- Obtaining a user ID

There are three different approaches to security for Java applets.

- Trusted servers
- Sandboxes
- Digital signatures

Trusting the server is a viable choice within the secure corporate intranet (files are downloaded regularly from corporate file servers). The corporate servers can be trusted not to deliver components that contain viruses or damage the system on which they're loaded and executed.

Sandboxing constrains the components themselves, making it impossible for them to execute unwanted functions. Sandboxing can guarantee security by dictating that the downloaded components are obligated to play only in their own sandbox. The disadvantage of this approach is that sandboxed components are prohibited from doing things that can sometimes be useful, like writing to a file on the client machine's local disk.

Digitally signing each downloaded component is an attractive approach. The digital signature can be checked by the browser that receives the component. If it is correct, the browser can be certain that the component was created by a specific trusted entity and that it has not been modified.

2.10 Java Programming and Development Environments

2.10.1 Getting Started

An example of the classic “Hello World” application is shown in Code-Example 1. The example is an application, not an applet. It writes the “Hello, World!” string to the system console. This example shows many of the basic features of the language.

- The file name, without the “.java” extension, is the same as the public class name, including capitalization.
- Java has three forms of comments:
  /** Comments between slashes and asterisks, begun with a double asterisk. These comments may be read by the javadoc utility for automatic incorporation into documentation, including specific values of the form ‘@keyword’. */

  /* Comments between slashes and asterisks. */

  // Comments after a double slash, continuing to the end-of-line.
- All Java code is located within the class definition.
- The declaration of the “main” method, which is “public” and “static”, or class-level, which returns “void”, i.e., nothing, and which takes an array of String arguments in the parameter “args”.
- The invocation of the “println” method of the “out” PrintStream attribute of the System class.

Code-Example 1: HelloWorld.java -- The Hello World Application in Java

```
/*
* This class prints out the phrase “Hello, World!”
* @author The author’s name
* @version 1.0 */
public class HelloWorld {
  /*   A simple Java application    */
   public static void main (String args []) {
        System.out..println ("Hello, World!");
   // comment:  System.out..println = output
         }
}
```

The most basic way to run a Java application is to use the Java Development Kit (JDK), which is widely available for many platforms, usually from the platform vendor, e.g., Sun Microsystems markets a JDK for Windows and Solaris platforms, HP markets a JDK for HP platforms, and, similarly, IBM for IBM platforms. Microsoft markets a Java SDK that is similar to a JDK for Windows platforms. The following steps show how to run the HelloWorld application using the JDK.

- Put the source statements in HelloWorld.java

  - Compile by typing: javac HelloWorld.java
  - This creates a file: HelloWorld.class (in byte-code)
- To run, type: java HelloWorld

2.10.2 An Applet

Code-Example 2 shows the code for a simple Java applet that writes “Hello, World!” to the browser’s Java console. It points out the basic differences between applications and applets.

- The java.applet libraries are imported. The HelloWorld application did not use any Java classes other than System, which is always available. In contrast, the HelloWorld applet uses the applet libraries. Practical examples usually import several libraries.
- An applet “extends Applet”, meaning that it is a subclass of the Applet class, usually the version shipped as java.applet.Applet.
- An application has a “main” method; an applet does not.
- An applet overrides one or more of the methods of the Applet class, especially:
  init() - how to set up.

  start() - begins the operation of the applet.

  stop() - ends or suspends the operation of the applet.

  paint() - draws the visible representation of the applet.

Code-Example 2: HelloWorld.java -- Hello World as an Applet

```
import java.applet.*;  // include applet classes (i.e., imports necessary classes)
/**
* This applet prints out the phrase “Hello, World!”
* @author The author’s name
* @version 1.0 */
public class HelloWorld extends Applet {
  /*   A simple Java applet    */
   start () {
        System.out..println ("Hello, World!");
   // comment:  System.out..println = output
         }
}
```

2.10.3 Example of a Java Application

Figure 2-1 shows a sample object model (class diagram) of a store that sells TVs and radios that the customer can buy. The following code segment translates the object model into a Java program (we have made a few simplifications to get the main points across. As you can see, the customer class and the product class are directly represented as class statements in Java. the customer class supports two methods (insert and retrieve) that can be used to add and query customer database. The product class supports a "buy" method that can be invoked by a customer to buy products. The product class is a superclass that is used to inherit properties for the TV and Radio derived classes. The "Extends" verb is used to show inheritance relationships. In the "main" part of the code, you see that RCA and Sony are created as instances of a TV and then a buy operation is invoked on object Sony (i.e., someone buys a Sony TV). As stated, this is a simplified view o Java programs that can be extended to add more programming aspects of Java.

```
class customer {          // class definition

int cid, accountno, balance;    // private variables

 public

  retrieve (int,  int);

 insert (int, int, int);

 }

class product  {                  // superclass

       int id, price, ,,,.    // parameters

       public buy  (cid)  // method indicating that a customer buys TV }

class TV Extends product {      derived class; inherits from product

         int warrantor, ..... //   parameters

     public getwarranty ()   // method

class Radio Extends product { derived class; }

//  Methods

 int retrieve (int,  int); {    .........  }

void insert (int, int, int); {    .........  }

Main  .... The code to initialize objects and perform operations

Sony, RCA = TV   //Sonny and RCA are instances of TV (objects)

Sony.buy         // invoke the method buy on TV Sony
```

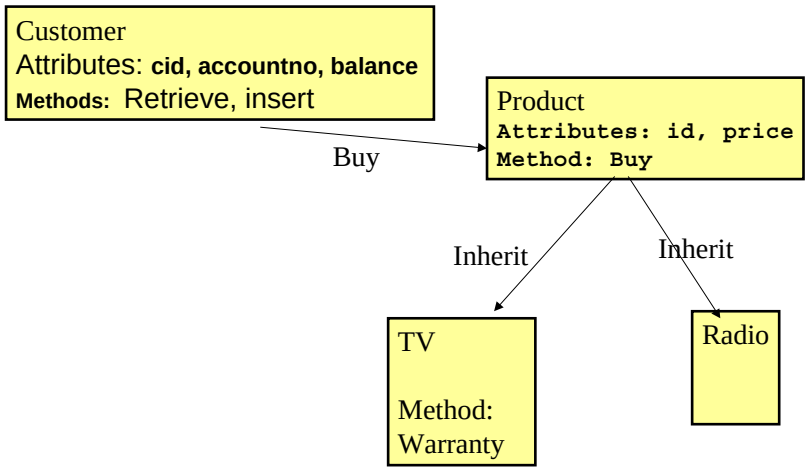


Figure 2-10: Object Model of a Store

2.10.4 Java Development Tools and Environments

The JDK contains a compiler, a run-time environment, and some basic utilities. A Java developer needs much more, however, and that is where the Integrated Development Environment (IDE) comes in.

An IDE minimally combines the JDK with a source code editor and a debugger. Additionally, the IDE may provide such features as color syntax display, code formatting, templates, a graphical class library browser, components, and assistance in building the results.

The IDE is not an original feature of the Java programmer's world. Before Java was created, IDE's existed for other languages, notably C++.

2.11 Java in Web Environments -- A Closer Look at Java Applets

2.11.1 Java-enabled Web Browsers

Web browsers are the end user interface to the Web servers. These browsers, also known as Web clients, typically reside on PCs, Macs and UNIX workstations. From an end users' point of view, the browsers give a GUI and easy to use view of the Internet and provide pull down/pop up menus and buttons for accessing remote servers, scrolling through documents, printing results, downloading code, saving retrieved documents on your local disk, performing searches, and surfing the net. Many browsers have been introduced since 1990 and are currently in use. Examples are the Netscape Navigator, Internet Explorer, HotJava, NCSA X-Mosaic, NCSA Mosaic for Windows, Spyglass, Air Mosaic, and Win-Tapestry.

Web browsers are designed to display information in HTML format and communicate with the Web servers through HTTP. As a matter of fact, you can develop your own browser if you provide the following two capabilities:

- HTML compliance, i.e., display information on the screen as specified by HTML tags.

- HTTP compliance, i.e., generate HTTP commands to connect to the Web server, initiate needed operations whenever a user clicks on a hyperlink, and receive/interpret the responses.

Many popular browsers, such as the Netscape Navigator and the Internet Explorer run on multiple platforms (PCs, Macs, UNIX). This is one of the many reasons for the popularity of WWW in the corporate world. While in the past a library system or a customer information system could have been developed by using a specially designed user interface, it seems much more natural for organizations today to use Web browsers for user interfaces. By using the Web browsers, users residing on different machines can use the same browser to interact with the corporate systems. The same browser can also allow the users to use Web for document searches. Thus Web browsers have the potential of becoming the only user interface for all information. This makes WWW unique in that it makes hypermedia a key enabler of business as well as non-business information that is becoming available through the Internet and Intranets.

Let us look at the Java-enabled Web browsers in some detail. As indicated previously, Java applets are indicated by an APPLET tag in HTML documents. For example, the following tag indicates a Java applet called "applet1.class" that is run in a window size of 100 by 150:

```
<APPLET CODE=myapplet.class WIDTH=100 HEIGHT=150>
```

We have discussed how the applet classes are loaded and executed on the Web browser. Let us see what goes into an applet class. Basically, all applets are subclasses of java.applet.Applet. This Applet class inherits from several classes of in the Java AWT (Advanced Window Toolkit) package. The Applet class inherits user interface capabilities such as displays and event handling from the AWT and adds a number of methods to interact with the Web browser. Examples of these methods are:

```
init() - the method where the applets initialize themselves

start() - the method called when the applet starts (i.e., applet page has been loaded or revisited)

stop() - the method called when the applet's page is no longer on the screen

mouseDown() - the method called to respond to when the mouse button is pressed down

paint() - the method called to paint and draw on the screen
```

Basically, a Java-enabled Web browser supports the libraries and methods needed by the Java applets (i.e., it supports a Java Virtual Machine). When writing Java applets, you need to invoke the init, start, mouseDown, paint, stop, and other such methods to interact with the users through the Web browsers.

Different browsers (e.g., Netscape Navigator and Microsoft Internet Explore) do not support the same features of Java because each browser supports its own default JVM that may differ from each other. This leads to compatibility problems for Java applets (i.e., some features are supported by one browser but not by the other). These compatibility issues between Web browsers cause significant problems for Web-based application developers (i.e., applets work for one browser but not for the other). The Sun Java Activator is designed to address this problem.

2.11.2 A more significant Java Applet

Code-Example 3 shows an applet with a paint () method, that operates on the Graphics object that it takes in as a parameter.

Java applets are not stand-alone applications and they run as part of a Java enabled browser. A Java applet may contains methods (subroutines) to initialize itself, draw itself, respond to clicks, etc. These methods are invoked by the Java enabled browser. A Java powered HTML page contains a tag (the APPLET Tag) that indicates the location of a Java applet. When the browser encounters this tag, it downloads it and runs it. Java applets are indicated by an APPLET tag. For example, the following tag indicates the Java applet called "LineApplet.class" that is run in a window size of 110 by 100:

```
<APPLET CODE=LineApplet.class WIDTH=110  HEIGHT=100> <APPLET>
```

Observe that this HTML code defines the size of the applet, and that the applet paint() code in Code-Example 3 works with the space that it is given. The HTML controls the size of the applet display: if the applet tries to create a larger display than the HTML allocated, the display is truncated.

The applet code is assumed to be on the same site where the HTML page resides. The browser loads the indicated class and other needed classes (Java "enabled" browsers also keep local classes that may be used by the applets). After the applet has been loaded, the browser asks it to initialize itself (the init() method) and draw a display area (the paint() method) that is used for input/output. Java applets have access to a wide range of libraries that allow Java applets to perform many operations such as graphics, image downloading, playing audio files, and user interface creation (i.e., buttons, scrollbars, windows, etc.).

Code-Example 3: LineApplet.java: A Simple Java Applet That Draws a Line

```
import java.awt.*;          // include the java tool classes and
import java.applet.*;  // include applet classes (i.e., imports necessary classes)
public class LineApplet extends Applet { // LineApplet is a subset of Applet
  public void paint (Graphics g)  {  // paint method is overridden
  Dimension r = size();     // find out how big the applet window is
  g.setColor (Color.green);     // set color  to green
  g.drawLine(0,0, r.width, r.height);    // draw the line from corner to corner
  }
}
```

2.11.3 An Applet/Application combination

The definitions of Applets and Applications are not exclusive: a single class definition can be both an applet and an application. To be both applet and application, a class is coded as an applet, but it is also provided with a main() method that allocates a browser-like environment, and then invokes the methods of the applet. This technique is sometimes used to create applets that can be unit tested from the command line, without the browser. The LineApplet seen earlier is turned into an example applet/application combination in Code-Example 4

Code-Example 4: An Applet/Application combination

```
import java.awt.*;
import java.applet.Applet;
public class TestableLineApplet extends Applet {
 public static void main (String args[]) {
   TestableLineApplet t = new TestableLineApplet();
   Frame f = new Frame("Testable LineApplet Test");
   f.resize(250, 250);
   f.show();
   t.init();
   t.start();
 }
   public void paint (Graphics g)  {  // paint method is overridden
  Dimension r = size();     // find out how big the applet window is
  g.setColor (Color.green);     // set color  to green
  g.drawLine(0,0, r.width, r.height);    // draw the line from corner to corner
  }
```

2.11.4 More Complex Applets

The applets discussed above have been simple and introductory. Considerably more complex applets with richer functionality are possible. An important way by which additional functionality is added is via database access.

By using Java applets, access to remote applications and databases can be invoked directly from the browser. The Java applet can ask the user to issue a query and then send this query to a remote application or database (Figure 2-11). This is especially interesting for database gateways where the database gateway functionality runs on the client side. A standard called JDBC (Java Database Connectivity) has been developed to allow Java programs to issue calls to relational databases.

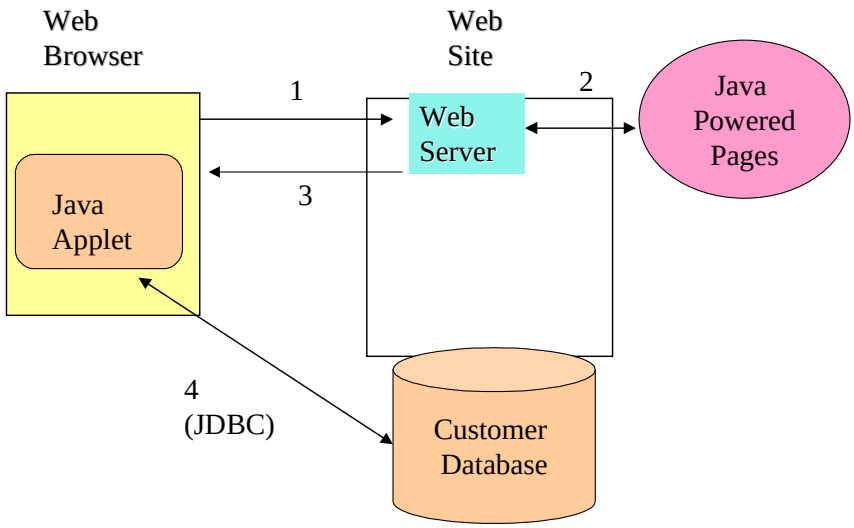


Figure 2-11: Java Applet-Based Application

If you need to write a Java application where a Java applet on your Web browser invokes another Java or C++ program on another machine, then you can use distributed object middleware such as CORBA. Sun has developed a new feature of Java that allows Java applets to talk to each other across machines without needing any middleware such as ORBs. This feature, known as Remote Method Invocation (RMI) allows Java applets to communicate with each other over the Internet. Figure 2-12 shows an example where a customer object on a remote web site is invoked by a Java applet by using CORBA or RMI. Details of this type of distributed applications can be found in the chapter on CORBA in the "Middleware" Module of this book. .

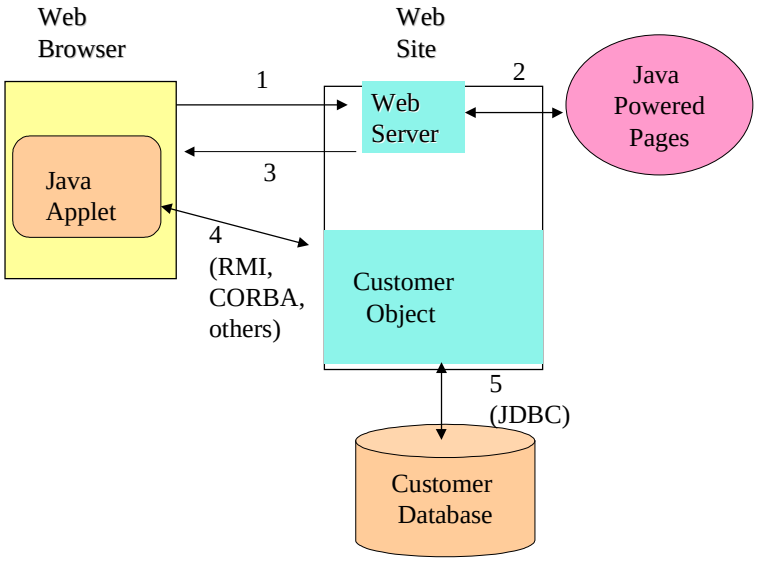


Figure 2-12: A Java Applet Invoking a Remote Object

2.11.5 Java Servlets

Servlets are Java programs that reside on the Web servers. Basically, servlets are to servers what applets are to browsers. Like applets, servlets cannot run as standalone programs -- they require a container .such as the Web server. The container receives a call, instantiates the servlet being invoked and passes control to the servlet. When the servlet is done, it issues a "destroy" call that tells the container to close up the servlet shop by shutting it down gracefully.

Servlets receive the calls issued by the client (browser), perform some computations, fetch some Web/non-Web content, and generate the response to be sent back to the client. For example, a servlet would receive SQL calls from a Web browser, send the SQL statements to the target databases, receive the results of the SQL query, build HTML pages from the results, and send the results back to the Web browsers. Figure 2-13 shows a conceptual view of servlets. .

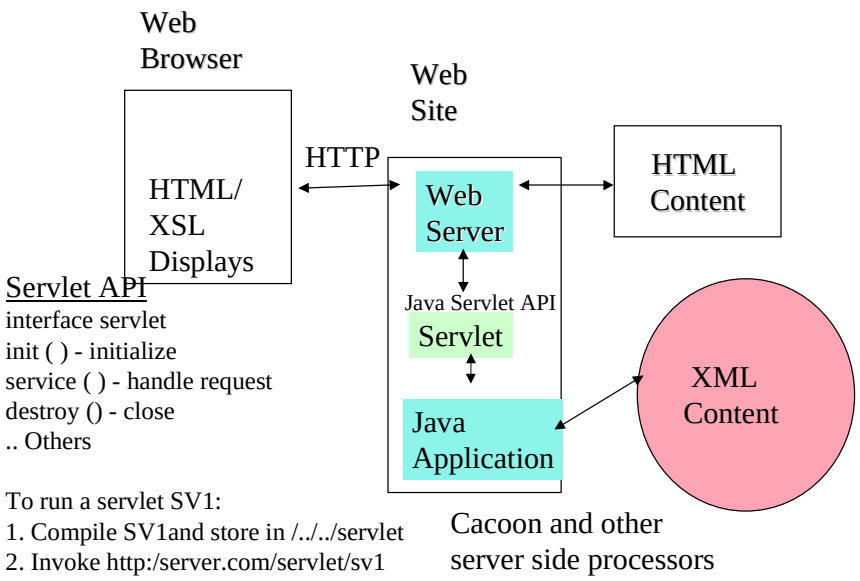


Figure 2-13: Servlets

Servlets are developed by using the Java Servlet API. This API is supported by numerous Web servers including the very popular Apache server. You invoke a servlet in a manner similar to CGI programs (see the notes in Figure 2-13). Many specialized servlets are becoming commercially available. An example is the Cocoon servlet engine that performs XML processing on the Tomcat and other Web servers.

2.11.6 The Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is an essential component of the Java environment. The JVM is a specification that defines whether and how Java byte-code class files should be executed. [Lindholm, 1996] The specification makes no mention as to how this will be accomplished, and it may be done via interpretation, compilation into binary code, or via hardware -- the proposed “Java chip”. The JVM specification provides many of Java’s features, notably its portability and security.

Java byte-code is run through a “JVM implementation”, which is often loosely referred to as a “JVM”. First the byte-code is validated, to ensure that it:

- Does not try to do anything illegal.
- Has not been altered since its compilation.

If the byte-code is valid, then it is executed. The specification of byte-code execution is machine-independent, so that the same behavior can be produced on any given machine by a JVM implementation that is specific to the machine. Although Java byte-code is portable, a JVM implementation is not portable. A JVM implementation is machine-specific.

The JVM is not the first virtual machine. Other VM's have preceded it, and others continue to be developed. The JVM Specification itself notes "the best-known virtual machine may be the P-Code machine of UCSD Pascal."

Java is not the only source of byte-codes to feed the JVM. Compilers have been written to generate Java byte-codes from other languages, e.g., Ada. However, Java source code provides a straightforward path to the generation of Java byte-codes.

2.11.7 Differences among JVM Implementations

The JVM Specification specifies what a JVM implementation should do, not how it should do it. Thus, there is a level of flexibility in how a JVM goes about executing Java byte-codes. One of the degrees of freedom is whether to execute the byte-codes via interpretation, compilation into machine code and execution of the resulting machine code, or in hardware, by creating a machine whose instructions correspond to the byte-code operands. This area is a source of distinction among competing JVM implementations, as different vendors try to create implementations that execute byte-codes quickly. Various techniques can be tried either separately or in combination. Among the popular techniques are:

- Optimized interpretation - Executing the byte-codes in an interpreter, but separately analyzing the byte-codes for improvements that can be made, e.g., in memory management or execution order.
- Just-In-Time (JIT) compilation - Compiling the byte-codes before and during execution. In contrast to compiling the entire applet or application before starting to execute them, this technique involves compiling and executing simultaneously, and the compiler is only a step ahead of the execution of the resulting machine code. The most obvious improvement caused by JIT compilation is when certain parts of the byte-code program are executed repeatedly. Because those sequences only have to be compiled once, the improvement in efficiency over interpretation can be significant.

Another degree of freedom provided by the specification is in what to optimize. While the most heavily publicized JVM implementations optimize for time, other implementations may optimize for other considerations, e.g., a JVM implementation intended for use in embedded devices might optimize for efficient memory usage.

2.12 State of the Practice: General Observations

Although OO technology has come into the limelight in the 1990s, it has been with us since the late 1960s when Simula programming language was introduced. OO technologies are at present becoming a de facto practice for developing new applications. Case studies and examples of using OO technologies are frequently discussed in the literature. For example, the October 1995 Communications of the ACM is a special issue on OO experiences and trends. This issue contains many insightful articles on topics such as lessons learned from the OS/400 OO Project, developing reusable object oriented communications software, developing an object oriented software testing and maintenance environment, and object oriented parallel computation. Similarly, a "virtual roundtable" on object technology [El-Rewini 1995] has many short assessments on vast aspects of OO technologies such as distributed objects, parallelism and objects, object databases, software quality and object orientation, and theoretical foundations. In addition, trade journals such as Object Magazine regularly publish case studies and experiences in OO technologies.

Let us make some general observations about the current state of the practice in OO technologies. First, almost all new applications are using OO technologies in user interfaces. Second, OO programming is steadily gaining ground especially due to the availability of off-the-shelf class libraries. In particular, OO programming in C++ is quite popular and the relatively new OO language Java has gained tremendous popularity for Web applications. Finally, OO databases still have a long way to go. Although, OODBMSs have been commercially available since the mid 1980s, only 3 to 5% of corporate data is currently being stored in OODBMS.

2.13 Summary

The object orientation (OO) revolution appears to be well entrenched at present in state of the practice as well as state of the market. The current emphasis on business objects [Sims 1994, Rymer 1995] and distributed objects [Orfali 1996] is a very promising area of current and future developments. This very short tutorial has attempted to give you the basic OO concepts and introduced you to the core OO technologies that are of relevance to this book.

2.14 Additional Information

Alpert, S., et al, "Guest Editor's Introduction: object oriented Programming in AI", IEEE Expert, Dec. 1990, Special Issue on object oriented Programming.

Atkinson, M., et al, "The Object Oriented Database Manifesto", Proceedings of the International Conference on Deductive and object oriented Databases, Kyoto, Japan, Dec. 1989.

Bic, L. and Gilbert, J., "Learning from AI: New Trends in Database Technology", IEEE Computer, March, 1986

Booch, G., "Object Oriented Design", IEEE Trans. on Software Engg, Feb. 1986.

Booch, G., "Object Oriented Design with Applications", Benjamins Cummings, Second edition, 1994.

Bray, O., "CIM: The Data Management Strategy", Digital Press, 1988.

Bretl, B., et al, "The Gemstone Data Management System: object oriented Concepts, Applications, and Databases", Addison-Wesley, 1989.

Butterworth, P., et al, "The Gemstone Object Database Management System", Comm. of ACM, Oct. 1991, pp. 64-77.

Catteli, R.G.G, "Intoduction to The Next Generation Database Systems", Comm. of ACM, Oct. 1991, pp. 33-33.

Davis, J., "Object-Rrelational Databases", Distributed Computing Monitor, Patricia Seybold Group, Feb. 1995.

Deux, G., et al, "The O2 System", Comm. of ACM, Oct. 1991, pp. 34-49. * Deux, G., et al, "The Story of O2", IEEE Trans. on Knowledge and Data Engineering, March 1990.

El-Rewini, H. and Hamilton, S., "Object Technology: A Virtual Roundatble", IEEE Computer, Oct. 1995, pp. 58-72.

Fayad, M. et al, "Introduction (Object Oriented Experiences)", Communications of the ACM, Oct. 1995, pp. 50-53.

Goldberg, A., "Why Smalltalk?", Communications of the ACM, Oct. 1995, pp. 105-107.

Goldberg, A., and Robson,D., "Smalltalk-80: The Language", Addison-Wesley, 1989.

Hardwick, M., and D.L. Spooner, "Comparison of Some Data Models for Engineering Objects", IEEE CG&A, March 1987

Haas, L., et al, "Starburst Mid-Flight: As the Dust Clears", IEEE Trans. on Knowledge and Data Engineering, March 1990.

Hoff, A., et al, "Hooked on Java; Creating Hot Web Sites with Java Applets", Addison-Wesley, 1996.

Ketabchi, M., and Berzins, V., "Modeling and Managing CAD Databases", IEEE Computer, Feb. 1987, pp.93-102.

Kemnitz, G. and Stonebraker, M., "The POSTGRES Tutorial", Electronics Research Laboratory, Memo M91/82, UC Berkeley, Feb. 1991.

Kim, W., "Introduction to Object Databases", MIT Press, 1990.

Kramer, M., "Object Databases", Distributed Computing Monitor, Patricia Seybold Group, April 1995.

Kroha, P., "Objects and Databases", McGraw Hill, 1993.

Lohman, G.M. et al, "Extensions to Starburst: Objects, Types, Functions, and Rules", Comm. of ACM, Oct. 1991, pp. 94-109.

Loomis, M., "Object Databases: The Essentials", Addison Wesley, 1995.

Martin, D., "Advanced Database Techniques", MIT Press, 1986.

Meade, D., "Object Lessons", Beyond Computing, July/August 1995, pp. 41-42.

Meyer, Bertrand, "Object Oriented Software Construction", Prentice Hall, 1988.

Nicol, J., et al, "Object Orientation in Heterogeneous Distributed Computing Systems", IEEE Computer, June 1993. pp. 57- 67. ;

Orfali, R., Harkey, D., and Edwards, J., "Essential Client/server survival Guide", John Wiley, 1994.

Orfali, R., Harkey, D., and Edwards, J., "Distributed Objects Survival Guide", John Wiley, 1996.

Pancake, C., "The Promise and the Cost of Object Technology: A Five-Year Forecast", Communications of the ACM, Oct. 1995, pp. 32-49.

Pascal, "Macintosh Programmer's Workshop Pascal 3.0 Reference", Apple Computer, 1989.

Pinsen, L.J., and Weiner, R.S., "Objective-C", Addson-Wesley, 1991.

Rasdorf, M., "Extending DBMSs for Engineering Applications", Computers in Mechanical Engineering, March 1987, pp. 62-69.

Rentsh, T., Sigplan Notices, Vol. 17, No. 9, 1982.

Ricciuti, M., "Object Databases Find Their Niche", Datamation, Sept. 15, 1993, pp. pp. 56-60.

Rumbaugh, J., et al, "Object-Oriented Modeling and Design", Prentice Hall, Second edition, 1994.

Rymer, J., "Buisness Objects", Distributed Computing Monitor, Patricia Seybold Group, Jan. 1995.

Saunders, J., "A Survey of object oriented Programming Languages", Journal of object oriented Programming Languages, March/April 1989.

Semich, W., "What's the Next Step After Client/Server?", Datamation, March 15, 1994, pp. 26-34.

Shan, Y., "Introduction (Smalltalk on the Rise)", Communications of the ACM, Oct. 1995, pp. 102-105.

Sheton, R., "Enterprise Reuse", Distributed Computing Monitor, Patricia Seybold Group, August 1995.

Silberschatz, A., et al, "Database Systems: Achievements and Opportunities", Comm. of ACM, Oct. 1991, pp. 110-120.

Simpon, D., "Objects May Appear Closer Than They Are", Client/Server Today, August 1995, pp. 59-69.

Sims, O., "Business Objects", McGraw Hill, 1994.

Spooner, D., "An Object Oriented Data Management System for Mechanical CAD", IEEE, 1986 Conf. on Graphics

Stroustrup, B., "The C++ Programming Language", Addison-Wesley, 1986.

Stonebraker, M. and Kemnitz, G., "The POSTGRES Next Generation Database System", Comm. of ACM, Oct. 1991, pp. 78-93.

Sudama, R., "Get Ready for Distributed Objects", Datamation, Oct. 1, 1995, pp. 67-72.

Taylor, D., "Object Oriented Technology: A Manager's Guide", Addison-Wesley, 1994.

Ullman, J., "Principles of Database Systems", John Wiley, 1982.

Ullman, J., "Principles of Database and Knowledge-Base Systems", Computer Science Press, 1988.

Umar, A., "Distributed Computing and Client/Server Systems", .Prentice Hall, 1993 (revised)

Wegner, P., "Dimensions of Object-Based Language Design", SIGPlan Notices, Vol. 22, No. 12, Dec. 1987, pp. 168-182.

Wirfs-Brock, R., Wilkerson, B. and Wiener, L., "Designing object oriented Software", Prentice-Hall, 1990.