

UNIT - I

LESSON – 1: OBJECT ORIENTED CONCEPTS

CONTENTS

1.0 AIMS AND OBJECTIVES

1.1 INTRODUCTION

1.2 OBJECT-ORIENTED SYSTEMS DEVELOPMENT METHODOLOGY

1.3 OVERVIEW OF THE UNIFIED APPROACH

1.4 LET US SUM UP

1.5 POINTS FOR DISCUSSION

1.6 LESSON – END ACTIVITIES

1.7 REFERENCES

1.0 AIMS AND OBJECTIVES

The main objective of this unit is to define and understand the

- The object oriented philosophy and why it is needed
- The unified approach , methodology used to study the object oriented concepts

1.1 INTRODUCTION

Software development is dynamic and always undergoing major change. The methods and tools will differ significantly from those currently in use. We can anticipate which methods and tools are going to succeed, but we cannot predict the future.

Today a vast number of tools and methodologies are available for systems development.

Systems development refers to all activities that go into producing an information systems solution.

Systems development activities consists of

- systems analysis
- modeling,
- design
- implementation,
- testing, and
- maintenance.

A *software development methodology* is a series of processes leads to the development of an application. The software processes describe how the work is to be carried out to achieve the original goal based on the system requirements. The software development process will continue to exist as long as the development system is in operation.

Object-oriented systems development methods differ from traditional development techniques in that the traditional techniques view software as a collection of programs (or functions) and isolated data.

A program can be defined as

Algorithms + Data Structures = Programs:

“A software system is a set of mechanisms for performing certain action on certain data.”

The main distinction between traditional system development methodologies and newer object-oriented methodologies depends on their primary focus

- traditional approach
 - focuses on the functions of the system
- object-oriented systems development
 - centers on the object, which combines data and functionality.

1.2 OBJECT-ORIENTED SYSTEMS DEVELOPMENT METHODOLOGY

Object-oriented development offers a different model from the traditional software development approach, which is based on functions and procedures. In simplified terms, object-oriented systems development is a way to develop software by building self-contained modules or objects that can be easily replaced, modified, and reused.

In an object-oriented environment,

- software is a collection of discrete objects that encapsulate their data as well as the functionality to model real-world "objects."

- An object orientation yields important benefits to the practice of software construction
- Each object has attributes (data) and methods (functions).
- Objects are grouped into classes; in object-oriented terms, we discover and describe the classes involved in the problem domain.
- everything is an object and each object is responsible for itself.

Example

Consider the Windows application needs Windows objects A Windows object is responsible for things like opening, sizing, and closing itself. Frequently, when a window displays something, that something also is an object (a chart, for example). A chart object is responsible for things like maintaining its data and labels and even for drawing itself.

Object-oriented methods enable us to create sets of objects that works together synergistically to produce software that better model their problem domains than similar systems produced by traditional techniques. The systems are easier to adapt to changing requirements, easier to maintain, more robust, and promote greater de-sign and code reuse. Object-oriented development allows us to create modules of functionality. Once objects are defined, it can be taken for granted that they will perform their desired functions and you can seal them off in your mind like black boxes. Your attention as a programmer shifts to what they do rather than how they do it. Here are some reasons why object orientation works

Importance of Object Orientation.

. Higher level of abstraction

The object-oriented approach supports abstraction at the object level. Since objects encapsulate both data (attributes) and functions (methods), they work at a higher level of abstraction. The development can proceed at the object level and ignore the rest of the system for as long as necessary. This makes designing, coding, testing, and maintaining the system much simpler.

. Seamless transition among different phases of software development.

The traditional approach to software development requires different styles and methodologies for each step of the process. Moving from one phase to another requires a complex transition of perspective between models that almost can be in different worlds. This transition not only can slow the development process but also increases the size of the project and the chance for errors introduced in moving from one language to another. The object-oriented approach, on the other hand, essentially uses the same language to talk about analysis, design, programming, and database design. This seamless approach reduces the level of complexity and redundancy and makes for clearer, more robust system development.

. Encouragement of good programming techniques.

A class in an object-oriented system carefully delineates between its interfaces the routines and attributes within a class are held together tightly. In a properly designed system, the classes will be grouped into subsystems but remain independent; therefore, changing one class has no impact on other classes, and so, the impact is minimized. However, the object-oriented approach is not a panacea; nothing is magical here that will promote perfect design or perfect code.

. Promotion of reusability.

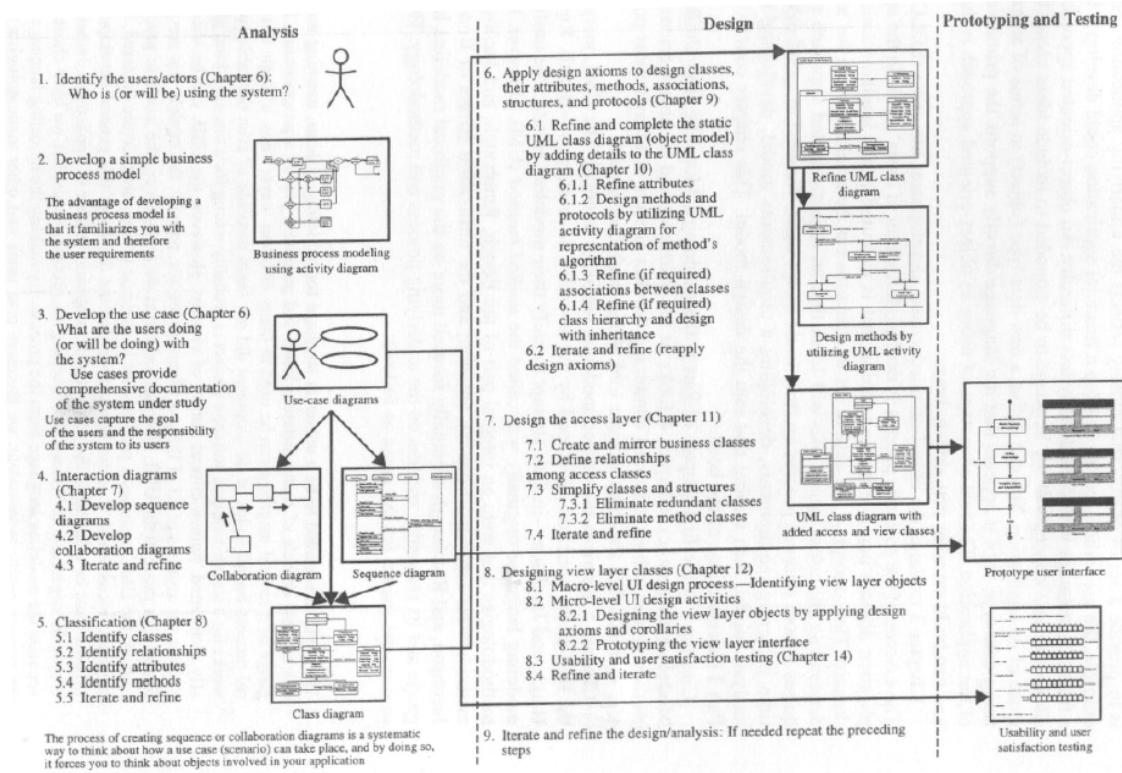
Objects are reusable because they are modeled directly out of a real-world problem domain. Each object stands by itself or within a small circle of peers (other objects). Within this framework, the class does not concern itself with the rest of the system or how it is going to be used within a particular system.

1.3 OVERVIEW OF THE UNIFIED APPROACH

The *unified approach* (UA) is a methodology for software development that is proposed by the author, and used in this book. The UA, based on methodologies by Booch, Rumbaugh, and Jacobson, tries to combine the best practices, processes, and guidelines along with the Object Management Group's unified modeling language.

The *unified modeling language* (UML) is a set of notations and conventions used to describe and model an application. But, the UML does not specify a methodology or what steps to follow to develop an application; that would be the task of the UA. Figure 1-1 depicts the essence of the unified approach. The heart of the UA is Jacobson's use case. The use case represents a typical interaction between a user and a computer system to capture the users' goals and needs.

The main advantage of an object-oriented system is that the class tree is dynamic and can grow. Your function as a developer in an object-oriented environment is to foster the growth of the class tree by defining new, more specialized classes to perform the tasks your applications require. After your first few projects, you will accumulate a repository or class library of your own, one that performs the operations your applications most often require. At that point, creating additional applications will require no more than assembling classes from the class library.



The unified approach road map.

1.4 LET US SUM UP

In an object-oriented environment, software is a collection of discrete objects that encapsulate their data and the functionality to model real-world "objects." Once objects are defined, you can take it for granted that they will perform their desired functions and so seal them off in your mind like black boxes. Your attention as a programmer shifts to what they do rather than how they do it. The object-oriented life cycle encourages a view of the world as a system of cooperative and collaborating agents.

An object orientation produces systems that are easier to evolve, more flexible, more robust, and more reusable than a top-down structure approach. An object orientation

- ❖ Allows working at a higher level of abstraction.
- ❖ Provides a seamless transition among different phases of software development.
- ❖ Encourages good development practices.
- ❖ Promotes reusability.
- ❖

The unified approach (UA) is the methodology for software development proposed and used in this book. Based on the Booch, Rumbaugh, and Jacobson methodologies, the UA consists of the following concepts:

- ❖ Use-case driven development.
- ❖ Utilizing the unified modeling language for modeling.
- ❖ Object-oriented analysis (utilizing use cases and object modeling).
- ❖ Object-oriented design.
- ❖ Repositories of reusable classes and maximum reuse.
- ❖ The layered approach.
- ❖ Incremental development and prototyping.
- ❖ Continuous testing.

1.5 POINTS FOR DISCUSSION

1. Evaluate two orthogonal views of the software
2. Justify why an object orientation

1.6 LESSON – END ACTIVITIES

1. Validate object oriented system development methodology
2. Analyze what is system development methodology?
3. Describe the components of the unified approach.

1.7 REFERENCES

1. Norman,Ronald- object oriented system analysis and design –prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 2: OBJECT ORIENTED CONCEPTS

CONTENTS

2.0 AIMS AND OBJECTIVES

2.1 INTRODUCTION

2.2 OBJECTS

 2.2.1 OBJECTS ARE GROUPED IN CLASSES

 2.2.2 ATTRIBUTES: OBJECT STATE AND PROPERTIES

 2.2.3 OBJECTS RESPOND TO MESSAGES

2.3 ENCAPSULATION AND INFORMATION HIDING

2.4 CLASS HIERARCHY

 2.4.1 INHERITANCE

 2.4.2 DYNAMIC INHERITANCE

 2.4.3 MULTIPLE INHERITANCE

2.5 POLYMORPHISM

2.6 OBJECT RELATIONSHIPS AND ASSOCIATIONS

2.7 AGGREGATIONS AND OBJECT CONTAINMENT

2.8 DYNAMIC BINDING

2.9 OBJECT PERSISTENCE

2.10 LET US SUM UP

2.11 POINTS FOR DISCUSSION

2.12 LESSON – END ACTIVITIES

2.12 REFERENCES

2.0 AIMS AND OBJECTIVES

The main aim of this lesson is know about the basic concepts of the objects and object oriented programming. At the end of this lesson you will be familiar with the concepts of

- Objects
- Classes
- Attributes
- Inheritance

- Polymorphism
- Messages
- Aggregations and Object Containment
- Object Persistence
- Meta-classes

2.1 INTRODUCTION

If there is a single motivating factor behind object-oriented system development, it is the desire to make software development easier and more natural by raising the level of abstraction to the point where applications can be implemented in the same terms in which they are described by users. Indeed, the name *object* was chosen because "everyone knows what an object is." The real question, then, is not so much "What is an object?" but "What do objects have to do with system development?" Let us develop the notion of an object through an example. A car is an *object* a real-world entity, identifiably separate from its surroundings. A car has a well-defined set of attributes in relation to other objects-such as color, manufacturer, cost, and owner-and a well-defined set of things you normally do with it-drive it, lock it, tow it, and carry passengers in it. In an object model, we call the former *properties* or *attributes* and the latter *procedures* or *methods*. *Properties* (or *attributes*) describe the state (data) of an object. *Methods* (procedures) define its behavior. Stocks and bonds might be objects for a financial investment application.

Parts and assemblies might be objects of a bill of materials application. Therefore, we can conclude that an object is whatever an application wants to "talk" about.

A fundamental characteristic of *object-oriented programming* is that it allows the base concepts of the language to be extended to include ideas and terms closer to those of its applications. New data types can be defined in terms of existing data types until it appears that the language directly supports the primitives of the application. In the financial investment example, a bond (data type) may be defined that has the same understanding within the language as a character data type. A buy operation on a bond can be defined that has the same understanding as the familiar plus (+) operation on a number. Using this data abstraction mechanism, it is possible to create new, higher-level, and more specialized data abstractions. You can work directly in the language, manipulating the kinds of "objects" required by you or your application, without having to constantly struggle to bridge the gap between how to conceive of these objects and how to write the code to represent them.

In an object-oriented system, the algorithm and the data structures are packaged together as an object, which has a set of attributes or properties. The state of these attributes is reflected in the values stored in its data structures. In addition, the object has a collection of procedures or methods-things it can do-as reflected in its package of methods. The attributes and methods are equal and inseparable parts of the object; one cannot ignore one for the sake of the other. For example, a car has certain attributes, such

as *color*, *year*, *model*, and *price*, and can perform a number of operations, such as *go*, *stop*, *turn left*, and *turn right*.

The traditional approach to software development tends toward writing a lot of code to do all the things that have to be done. The code is the plans, bricks, and mortar that you use to build structures. You are the only active entity; the code, basically, is just a lot of building materials. The object-oriented approach is more like employing a lot of helpers that take on an active role and form a community whose interactions become the application. Instead of saying, "System, write the value of this number to the screen," we tell the number object, "Write yourself." This has a powerful effect on the way we approach software development..

2.2 OBJECTS

The term *object* was first formally utilized in the Simula language, and objects typically existed in Simula programs to simulate some aspect of reality. The term *object* means a combination of data and logic that represents some real world entity. For example, consider a Saab automobile. The Saab can be represented in a computer program as an object. The "data" part of this object would be the car's name, color, number of doors, price, and so forth. The "logic" part of the object could be a collection of programs.

In an object-oriented system, everything is an object: A spreadsheet, a cell in a spreadsheet, a bar chart, a title in a bar chart, a report, a number or telephone number, a file, a folder, a printer, a word or sentence, even a single character all are examples of an object. Each of us deals with objects daily. Some objects, such as a telephone, are so common that we find them in many places. Other objects, like the folders in a file cabinet or the tools we use for home repair, may be located in a certain place.

2.2.1 OBJECTS ARE GROUPED IN CLASSES

Classes are used to distinguish one type of object from another. In the context of object-oriented systems, a *class* is a set of objects that share a common structure and a common behavior; a single object is simply an *instance* of a class. A class is a specification of structure (instance variables), behavior (methods), and inheritance for objects.) Classes are an important mechanism for classifying objects. The chief role of a class is to define the properties and procedures (the state and behavior) and applicability of its instances. The class *car*, for example, defines the *property* *color*. Each individual car (formally, 9th instance of the class *car*) will have a value for this property, such as maroon, yellow, or white. In an object-oriented system, a *method* or behavior of an object is defined by its class.

Each object is an instance of a class. There may be many different classes.

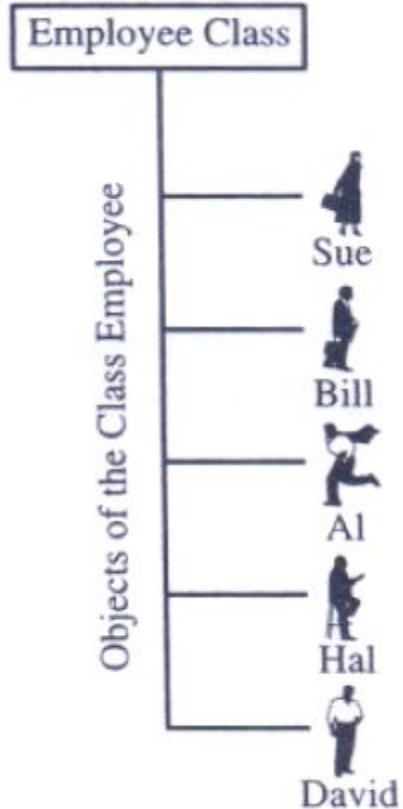


FIGURE 2.1 Sue, Bill, Al, Hal, and David are instances or objects of the class Employee.

2.2.2 ATTRIBUTES: OBJECT STATE AND PROPERTIES

Properties represent the state of an object. Often, we want to refer to the description of these properties rather than how they are represented in a particular programming language. In our example, the properties of a car, such as color, manufacturer, and cost, are abstract descriptions (Figure 2.2).

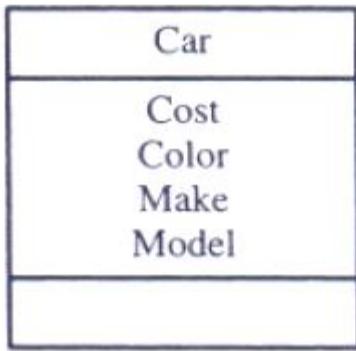


FIGURE 2.2 The attributes of a car object.

We could represent each property in several ways in a programming language. For color, we could choose to use a sequence of characters such as *red*, or the (stock) number for red paint, or a reference to a full-color video image that paints a red swatch on the screen when displayed. Manufacturer could be denoted by a name, a reference to a manufacturer object, or a corporate tax identification number. Cost could be a floating point number, a fixed point number, or an integer in units of pennies or even lira. The importance of this distinction is that an object's abstract state can be independent of its physical representation.

2.2.3 OBJECTS RESPOND TO MESSAGES

An object's capabilities are determined by the methods defined for it. Methods conceptually are equivalent to the function definitions used in procedural languages. For example, a draw method would tell a chart how to draw itself. However, to do an operation, a message is sent to an object. Objects perform operations in response to messages. For example, when you press on the brake pedal of a car, you send a *stop* message to the car object. The car object knows how to respond to the *stop* message, since brakes have been designed with specialized parts such as brake pads and drums precisely to respond to that message. Sending the same *stop* message to a different object, such as a tree, however, would be meaningless and could result in an unanticipated response.

Messages essentially are nonspecific function calls: We would send a *draw* message to a chart when we want the chart to draw itself. A message is different from a subroutine call, since different objects can respond to the same message in different ways. For example, cars, motorcycles, and bicycles will all respond to a *stop* message, but the actual operations performed are object specific. In the top example, depicted in Figure 2-3, we send a *Brake* message to the *Car* object. In the middle example, we send a *multiplication* message to 5 object followed by the number by which we want to multiply 5. In the bottom example, a *Compute Payroll* message is sent to the *Employee* object, where the employee object knows how to respond to the *Payroll* message. Note that the message makes no assumptions about the class of the receiver or the arguments; they are

simply objects. It is the receiver's responsibility to respond to a message in an appropriate manner. This gives you a great deal of flexibility, since different objects can respond to the same message in different ways. This is known as *polymorphism* (more on polymorphism later in this chapter), meaning "many shapes (behaviors)."

Polymorphism is the main difference between a message and a subroutine call. Methods are similar to functions, procedures, or subroutines in more traditional programming languages, such as COBOL, Basic, or C. The area where methods and functions differ, however, is in how they are invoked. In a Basic program, you call the subroutine (e.g., GOSUB 1000); in a C program, you call the function by name (e.g., draw chart). In an object-oriented system, you invoke a method of an object by sending an object a message. A message is much more general than a function call. It is important to understand the difference between methods and messages. Say you want to tell someone to make you French onion soup. Your instruction is the message, the way the French soup is prepared is the method and the French onion soup is the object.

Objects respond to messages according to methods defined in its class.

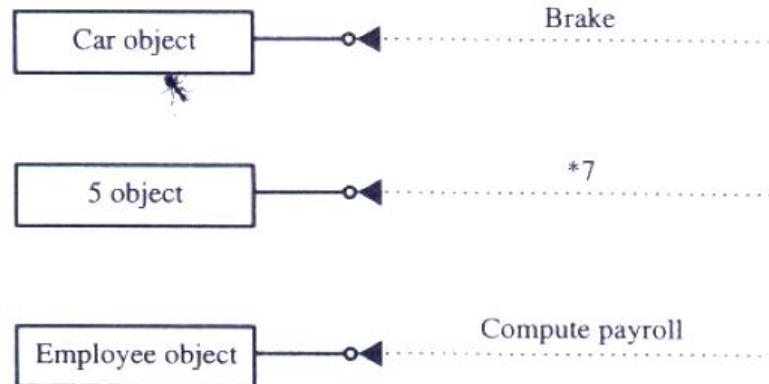


FIGURE 2.3 Objects respond to messages according to methods defined in its class.

A message has a name, just like a method, such as cost, set cost, cooking time. An object understands a message when it can match the message to a method that has a same name as the message. To match up the message, an object first searches the methods defined by its class. If found, that method is called up. If not found, the object searches the super class of its class. If it is found in a super class, then that method is called up. Otherwise, it continues the search upward. An error occurs only if none of the superclasses contains the method. A message differs from a function in that a function says how to do something and a message says what to do. Because a message is so general, it can be used over and over again in many different contexts. The result is a system more resilient to change and more reusable, both within an application and from one application to another.

2.3 ENCAPSULATION AND INFORMATION HIDING

Information hiding is the principle of concealing the internal data and procedures of an object and providing an interface to each object in such a way as to reveal as little as possible about its inner workings. As in conventional programming, some languages permit arbitrary access to objects and allow methods to be defined outside of a class. For example, Simula provides no protection, or information hiding, for objects, meaning that an object's data, or *instance variables*, may be accessed wherever visible. However, most object-oriented languages provide a well-defined interface to their objects through classes. For example, C++ has a very general *encapsulation* protection mechanism with public, private, and protected members. Public members (member data and member functions) may be accessed from anywhere. For instance, the *compute Payroll* method of an employee object will be public. Private members are accessible only from within a class. An object data representation, such as a list or an array, usually will be private. Protected members can be accessed only from subclasses.

An important factor in achieving encapsulation is the design of different classes of objects that operate using a common *protocol*, or object's user interface. This means that many objects will respond to the same message, but each will perform the message using operations tailored to its class. In this way, a program can send a generic message and leave the implementation up to the receiving object, which reduces interdependencies and increases the amount of interchangeable and reusable code.

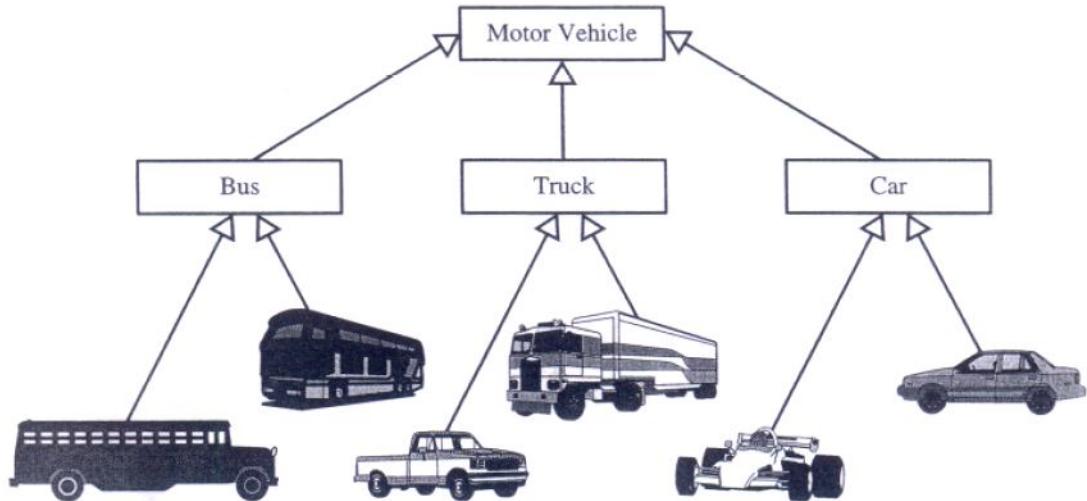
A car engine is an example of encapsulation. Although engines may differ in implementation, the interface between the driver and the car is through a common protocol: Step on the gas to increase power and let up on the gas to decrease power. Since all drivers know this protocol, all drivers can use this method in all cars, no matter what engine is in the car. That detail is insulated from the rest of the car and from the driver. This simplifies the manipulation of car objects and the maintenance of code.

Data abstraction is a benefit of the object-oriented concept that incorporates encapsulation and polymorphism. Data are abstracted when they are shielded by a full set of methods and only those methods can access the data portion of an object.

2.4 CLASS HIERARCHY

An object-oriented system organizes classes into a subclass-super class hierarchy. Different properties and behaviors are used as the basis for making distinctions between classes and subclasses. At the top of the *class hierarchy* are the most general classes and at the bottom are the most specific. The family car in Figure 2.4 is a subclass of car. A *subclass* inherits all of the properties and methods (procedures) defined in its *super class*. In this case, we can drive a family car just as we can drive any car or, indeed, almost any motor vehicle. Subclasses generally add new methods and properties specific to that class. Subclasses may refine or constrain the state and behavior inherited from its super class. In our example, race cars only have one occupant, the driver. In this manner, subclasses modify the attribute (number of passengers) of its super class, Car.

Superclass/subclass hierarchy.

**FIGURE 2.4** Super class/subclass hierarchies.

By contrast, super classes generalize behavior. It follows that a more general state and behavior is modeled as one moves up the super class-subclass hierarchy (or simply class hierarchy) and a more specific state is modeled as one move down. It is evident from our example that the notion of subclasses and super classes is relative. A class may simultaneously be the subclass to some class and a super class to other classes. Truck is a subclass of a motor vehicle and a super class of both 18-wheeler and pickup. For example, Ford is a class that defines Ford car objects (see Figure 2.5). However, more specific classes of Ford car objects are Mustang, Taurus, Escort, and Thunderbird. These classes define Fords in a much more specialized manner than the Ford car class itself. Since the Taurus, Escort, Mustang, and Thunderbird classes are more specific classes of Ford cars, they are considered subclasses of class Ford and the Ford class is their super class. However, the Ford class may not be the most general in our hierarchy. For instance, the Ford class is the subclass of the Car class, which is the subclass of the Vehicle class. The car class defines how a car behaves.

The Ford class defines the behavior of Ford cars (in addition to cars in general), and the Mustang class defines the behavior of Mustangs (in addition to Ford cars in general) wanted was a Ford Mustang object, you would write only one class, Mustang. The class would define exactly how a Ford Mustang car operates. This methodology is limiting because, if you decide later to create a Ford Taurus object, you will have to duplicate most of the code that describes not only how a vehicle behaves but also how a car, and specifically a Ford, behaves. This duplication occurs when using a procedural language, since there is no concept of hierarchy and inheriting behavior. An object-oriented system eliminates duplicated effort by allowing classes to share and reuse behaviors.

Class hierarchy for Ford class.

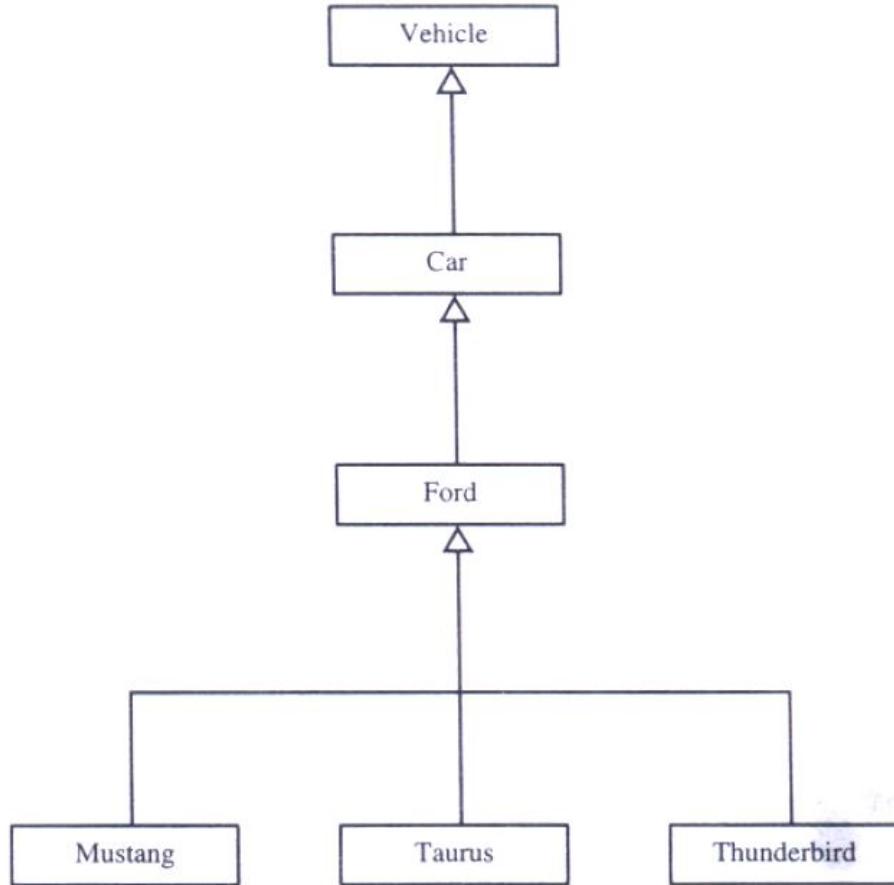


FIGURE 2.5 Class hierarchy for Ford class.

You might find it strange to define a Car class. After all, what is an instance of the Car class? There is no such thing as a generic car. All cars must be of some make and model. In the same way, there are no instances of Ford class. All Fords must belong to one of the subclasses: Mustang, Escort, Taurus, or Thunderbird. The Car class is a formal class, also called an abstract class. *Formal* or *abstract classes* have no instances but define the common behaviors that can be inherited by more specific classes.

In some object-oriented languages, the terms *super class* and *subclass* are used instead of *base* and *derived*. In this book, the terms *super class* and *subclass* are used consistently.

2.4.1 INHERITANCE

Inheritance is the property of object-oriented systems that allows objects to be built from other objects. Inheritance allows explicitly taking advantage of the commonality of objects when constructing new classes. Inheritance is a relationship between classes where one class is the parent class of another (derived) class. The parent class also is known as the *base class* or *super class*. Inheritance provides programming

by extension as opposed to programming by reinvention. The real advantage of using this technique is that we can build on what we already have and, more important, reuse what we already have. Inheritance allows classes to share and reuse behaviors and attributes. Where the behavior of a class instance is defined in that class's methods, a class also inherits the behaviors and attributes of all of its super classes.

For example, the Car class defines the general behavior of cars. The Ford class inherits the general behavior from the Car class and adds behavior specific to Fords. It is not necessary to redefine the behavior of the car class; this is inherited. Another level down, the Mustang class inherits the behavior of cars from the Car class and the even more specific behavior of Fords from the Ford class. The Mustang class then adds behavior unique to Mustangs. Assume that all Fords use the same braking system. In that case, the *stop* method would be defined in class Ford (and not in Mustang class), since it is a behavior shared by all objects of class Ford. When you step on the brake pedal of a Mustang, you send a *stop* message to the Mustang object. However, the *stop* method is not defined in the Mustang class, so the hierarchy is searched until a *stop* method is found. The *stop* method is found in the Ford class, a super class of the Mustang class, and it is invoked (Figure 2.6). In a similar way, the Mustang class can inherit behaviors from the Car and the Vehicle classes.

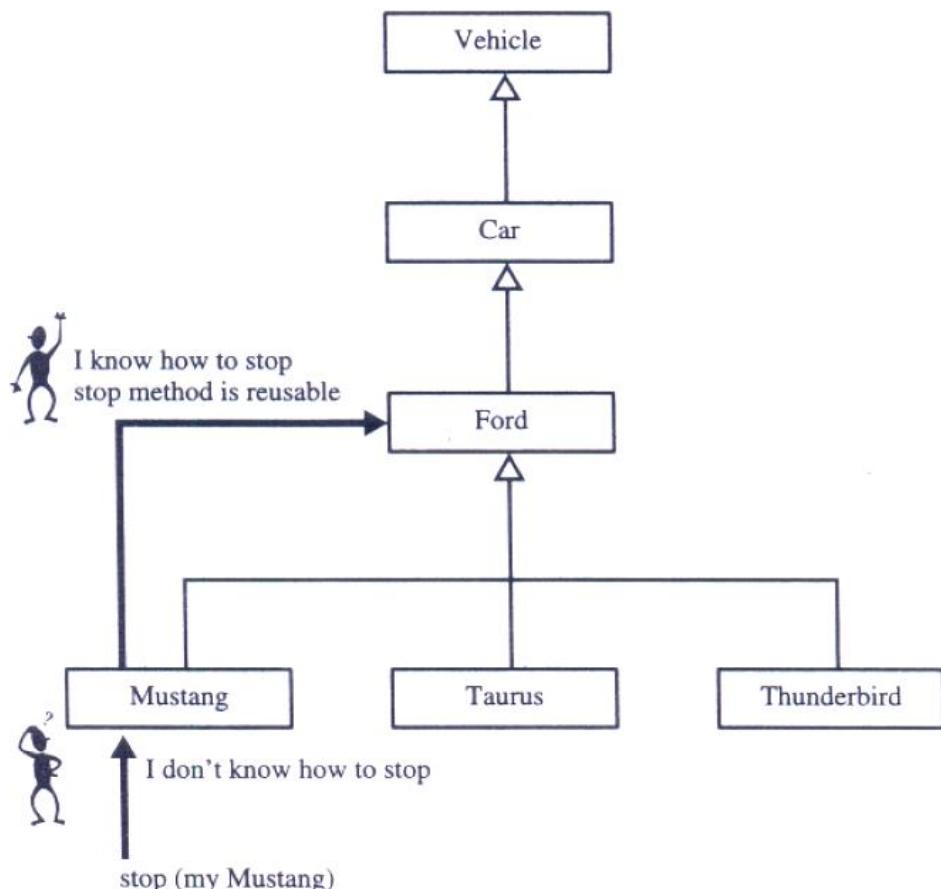


FIGURE 2.6 INHERITANCE ALLOWS REUSABILITY

2.4.2 DYNAMIC INHERITANCE

Dynamic inheritance allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, changing base classes changes the properties and attributes of a class. A previous example was a Windows object changing into an icon and then back again, which involves changing a base class between a Windows class and an Icon class. More specifically, *dynamic inheritance* refers to the ability to add, delete, or change parents from objects (or classes) at run time.

In object-oriented programming languages, variables can be declared to hold or reference objects of a particular class. For example, a variable declared to reference a motor vehicle is capable of referencing a car or a truck or any subclass of motor vehicle.

2.4.3 MULTIPLE INHERITANCE

Some object-oriented systems permit a class to inherit its state (attributes) and behaviors from more than one super class. This kind of inheritance is referred to as *multiple inheritance*. For example, a utility vehicle inherits attributes from both the Car and Truck classes.

Multiple inheritance can pose some difficulties. For example, several distinct parent classes can declare a member within a multiple inheritance hierarchy. This then can become an issue of choice, particularly when several super classes define the same method. It also is more difficult to understand programs written in multiple inheritance systems.

One way of achieving the benefits of multiple inheritance in a language with single inheritance is to inherit from the most appropriate class and then add an object of another class as an attribute.

2.5 POLYMORPHISM

Poly means "many" and *morph* means "form." In the context of object-oriented systems, it means objects that can take on or assume many different forms. *Polymorphism* means that the same operation may behave differently on different classes. Booch defines *polymorphism* as the relationship of objects of many different classes by some common super class; thus, any of the objects designated by this name is able to respond to some common set of operations in a different way. For example, consider how driving an automobile with a manual transmission is different from driving a car with an automatic transmission. The manual transmission requires you to operate the clutch and the shift, so in addition to all other mechanical controls, you also need information on when to shift gears. Therefore, although driving is a behavior we perform with all cars (and all motor vehicles), the specific behavior can be different, and depending on the kind of car we are driving. A car with an automatic transmission might implement its *drive* method to use information such as current speed, engine RPM, and current gear.

Polymorphism allows us to write generic, reusable code more easily, because we can specify general instructions and delegate the implementation details to the objects involved. Since no assumption is made about the class of an object that receives a message, fewer dependencies are needed in the code and, therefore, maintenance is easier. For example, in a payroll system, manager, office worker, and production worker objects all will respond to the *compute payroll* message, but the actual operations performed are object specific.

2.6 OBJECT RELATIONSHIPS AND ASSOCIATIONS

ASSOCIATIONS

Association represents the relationships between objects and classes. For example, in the statement "a pilot *can fly* planes" (Figure 2.7) the italicized term is an association. Associations are bidirectional; that means they can be traversed in both directions, perhaps with different connotations. The direction implied by the name is the forward direction; the opposite direction is the inverse direction. For example, *can fly* connects a pilot to certain airplanes. The inverse of *can fly* could be called *is flown by*.

An important issue in association is *cardinality*, which specifies how many instances of one class may relate to a single instance of an associated class. Cardinality constrains the number of related objects and often is described as being "one" or "many." Generally, the multiplicity value is a single interval, but it may be a set of disconnected intervals. For example, the number of cylinders in an engine is four, six, or eight. Consider a client-account relationship where one client can have one or more accounts and vice versa (in case of joint accounts); here the cardinality of the client-account association is many to many.

Consumer-Producer Association

A special form of association is a consumer-producer relationship, also known as a *client-server association* or a *use relationship*. The *consumer-producer relationship* can be viewed as one-way interaction: One object requests the service of another object. The object that makes the request is the consumer or client, and the object that receives the request and provides the service is the producer or



FIGURE 2.7 Association represents the relationship among objects, which is bidirectional.

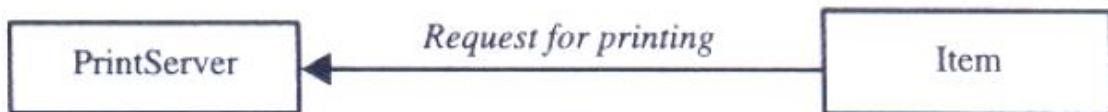


FIGURE 2.8 The consumer/producer association.

server. For example, we have a print object that prints the consumer object. The print producer provides the ability to print other objects. Figure 2.8 depicts the consumer-producer association.

2.7 AGGREGATIONS AND OBJECT CONTAINMENT

All objects, except the most basic ones, are composed of and may contain other objects. For example, a spreadsheet is an object composed of cells, and cells are objects that may contain text, mathematical formulas, video, and so forth. Breaking down objects into the objects from which they are composed is decomposition. This is possible because an object's attributes need not be simple data fields; attributes can reference other objects. Since each object has an identity, one object can refer to other objects. This is known as *aggregation*, where an attribute can be an object itself. For instance, a car object is an aggregation of engine, seat, wheels, and other objects (see Figure 2.9).

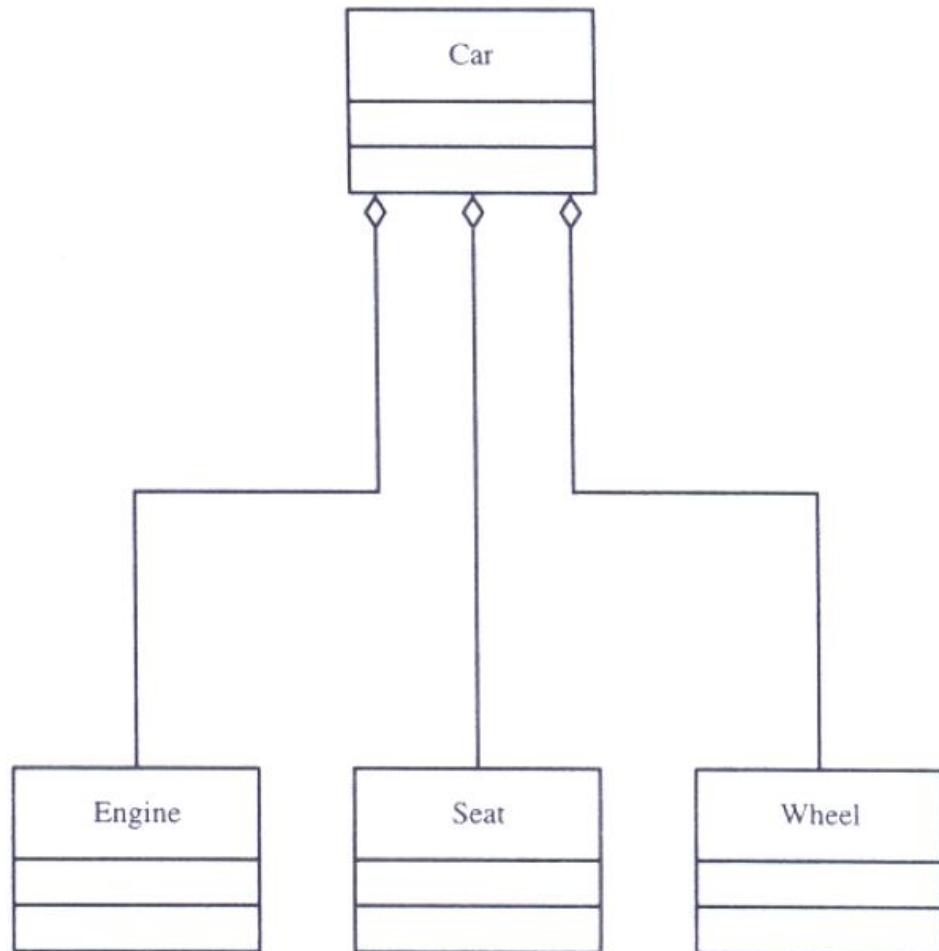


FIGURE 2.9 A Car object is an aggregation of other objects such as engine, seat, and wheel objects.

2.8 DYNAMIC BINDING

The process of determining (dynamically) at run time which function to invoke is termed *dynamic binding*. Making this determination earlier, at compile time, is called *static binding*. Static binding optimizes the calls; dynamic binding occurs when polymorphic calls are issued. Not all function invocations require dynamic binding. Dynamic binding allows some method invocation decisions to be deferred until the information is known. A run-time selection of methods often is desired, and even required, in many applications, including databases and user interaction (e.g., GUIs). For example, a cut operation in an Edit submenu may pass the cut operation (along with parameters) to any object on the Desktop, each of which handles the message in its own way. If an (application) object can cut many kinds of objects, such as text and graphic objects, many overloaded cut methods, one per type of object to be cut, are available in the receiving object; the particular method being selected is based on the actual type of object being cut (which in the *GUI* case is not available until run time).

2.9 OBJECT PERSISTENCE

Objects have a lifetime. They are explicitly created and can exist for a period of time that, traditionally, has been the duration of the process in which they were *created*. A file *or* a database can provide support for objects having a longer lifeline longer than the duration of the process for which they were created. From a language perspective, this characteristic is called *object persistence*. An object can persist beyond application session boundaries, during which the object is stored in a file *or* a database, in some file *or* database form. The object can be retrieved in another application session and will have the same state and relationship to other objects as at the time it was saved. The lifetime of an object can be explicitly terminated. After an object is deleted, its state is inaccessible and its persistent storage is reclaimed. Its identity, however, is never reused, not even after the object is deleted. Object storage and its access from the database will be covered in later chapters.

2.10 LET US SUM UP

The goal of object-oriented programming is to make development easier, quicker, and more natural by raising the level of abstraction to the point where applications can be implemented in the same terms in which they are described by the application domain. The main thrust of object-oriented programming is to provide the user with a set of objects that closely reflects the underlying application. The user who needs to develop a financial application could develop it in a financial language with considerably less difficulty. Object-oriented programming allows the base concepts of the language to be extended to include ideas closer to those of its application. You can define a new data type (object) in terms of an existing data type until it appears that the language directly supports the primitives of the application.

The real advantage of using the object-oriented approach is that you can build on what you already have. Object-oriented software development is a significant departure

from the traditional structured approach. The main advantage of the object-oriented approach is the ability to reuse code and develop more maintainable systems in a shorter amount of time. Additionally, object-oriented systems are better designed, more resilient to change, and more reliable, since they are built from completely tested and debugged classes.

Rather than treat data and procedures separately, object-oriented systems link both closely into objects. Events occur when objects respond to messages. The objects themselves determine the response to the messages, allowing the same message to be sent to many objects.

Each object is an instance *of* a class. Classes are organized hierarchically in a class tree, and subclasses inherit the behavior of their super classes. Good object-oriented programming uses encapsulation and polymorphism, which, when used in the definition of classes, result in completely reusable abstract data classes. Objects have a lifetime. They are explicitly created and Co-exist for a period of time that, traditionally, has been the duration of the process for which they were created. A file or a database can provide support for objects having a longer lifeline longer than the duration of the process for which they were created.

2.11 POINTS FOR DISCUSSIONS

1. Justify objects
2. Analyze how objects are grouped in classes
3. Discuss about the object behavior and methods

2.12 LESSON – END ACTIVITIES

1. Discuss about class hierarchy.
2. Critically analyze consumer – producer association.

2.12 REFERENCES

1. Norman,Ronald- object oriented system analysis and design –prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 3: SOFTWARE DEVELOPMENT PROCESS

CONTENTS

- 3.0 AIMS AND OBJECTIVES
- 3.1 INTRODUCTION
- 3.2 THE SOFTWARE DEVELOPMENT PROCESS
- 3.3 BUILDING HIGH-QUALITY SOFTWARE
- 3.4 OBJECT ORIENTED SYSTEMS DEVELOPMENT:
A USE-CASE DRIVEN APPROACH
- 3.5 LET US SUM UP
- 3.6 POINTS FOR DISCUSSION
- 3.7 LESSON – END ACTIVITIES
- 3.8 REFERENCES

3.0 AIMS AND OBJECTIVES

The main objectives of this lesson are

- To know in detail about the software development process.
- Building high-quality software
- Object-oriented systems development.
- Use-case driven systems development.

3.1 INTRODUCTION

The essence of the *software development process* that consists of analysis, design, implementation, testing, and refinement is to transform users' needs into a software solution that satisfies those needs. However, some people view the software development process as interesting but feel it has little importance in developing software. It is tempting to ignore the process and plunge into the implementation and programming phases of software development, much like the builder who would bypass the architect. Some programmers have been able to ignore the counsel of systems development in building a system.

The object-oriented approach requires a more rigorous process to do things right. This way, you need not see code until after about 25 percent of the development time, because you need to spend more time gathering requirements, developing a requirements model and an analysis model, then turning them into the design model. Now, you can develop code quickly—you have a recipe for doing it. However, you should construct a

prototype of some of the key system components shortly after the products are selected, to understand how easy or difficult it will be to implement some of the features of the system.

The prototype also can give users a chance to comment on the usability and usefulness of the design and let you assess the fit between the software tools selected, the functional specification, and the users' needs.

This lesson introduces you to the systems development life cycle in general and, more specifically, to an object-oriented approach to software development. The main point of this chapter is the idea of building software by placing emphasis on the analysis and design aspects of the software life cycle. The emphasis is intended to promote the building of high-quality software (meeting specifications and being adaptable for change). The software industry previously suffered from the lack of focus on the early stages of the life cycle.

3.2 THE SOFTWARE DEVELOPMENT PROCESS

System development can be viewed as a process. Furthermore, the development itself, in essence, is a process of change, refinement, transformation, or addition to the existing product. Within the process, it is possible to replace one sub process with a new one, as long as the new sub process has the same interface as the old one, to allow it to fit into the process as a whole. With this method of change, it is possible to adapt the new process. For example, the object-oriented approach provides us a set of rules for describing inheritance and specialization in a consistent way when a sub process changes the behavior of its parent process.

The process can be divided into small, interacting phases-sub processes. The sub processes must be defined in such a way that they are clearly spelled out, to allow each activity to be performed as independently of other sub processes as possible. Each sub process must have the following

- A description in terms of how it works
- Specification of the input required for the process
- Specification of the output to be produced

The software development process also can be divided into smaller, interacting sub processes. Generally, the software development process can be viewed as a series of transformations, where the output of one transformation becomes the input of the subsequent transformation (Figure 3-1):

Transformation 1 (analysis) translates the users' needs into system requirements and responsibilities. The way they use the system can provide insight into the users' requirements. For example, one use of the system might be analyzing an incentive payroll system, which will tell us that this capacity must be included in the system requirements.

Transformation 2 (design) begins with a problem statement and ends with a detailed design that can be transformed into an operational system. This transformation includes the bulk of the software development activity, including the

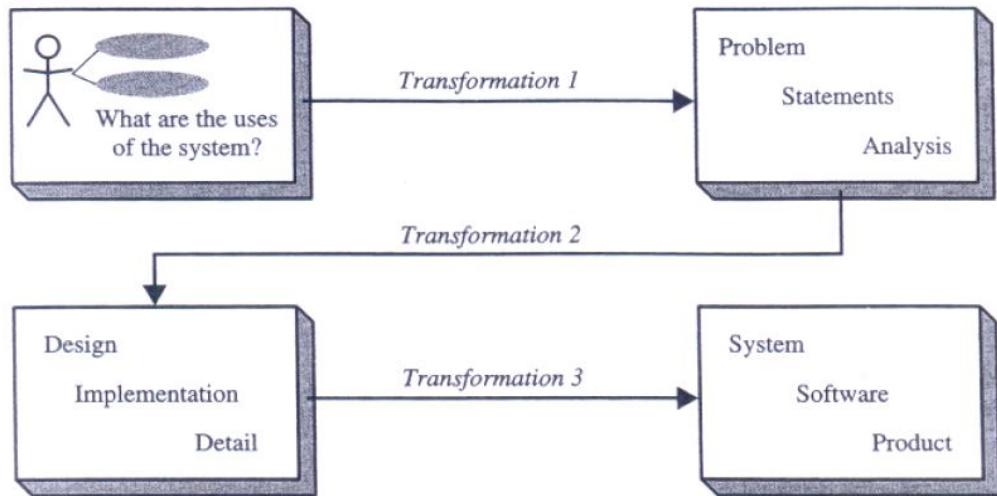


FIGURE 3-1

Software process reflecting transformation from needs to a software product that satisfies those needs.

definition of how to build the software, its development, and its testing. It also includes the design descriptions, the program, and the testing materials. . *Transformation 3 (implementation)* refines the detailed design into the system deployment that will satisfy the users' needs. This takes into account the equipment, procedures, people, and the like. It represents embedding the software product within its operational environment. For example, the new compensation method is programmed, new forms are put to use, and new reports now can be printed. Here, we try to answer the following question: What procedures and resources are needed to compensate the employees under the new accounting system?

An example of the software development process is the *waterfall approach*, which starts with deciding *what* is to be done (what is the problem). Once the requirements have been determined, we next must decide *how* to accomplish them. This is followed by a step in which we *do it*, whatever "it" has required us to do. We then must *test* the result to see if we have satisfied the users' requirements. Finally, we *use* what we have done (see Figure 3-2).

In the real world, the problems are not always well-defined and that is why the waterfall model has limited utility. For example, if a company has experience in building accounting systems, then building another such product based on the existing design is

best managed with the waterfall model, as it has been described. Where there is uncertainty regarding what is required or how it can be built, the waterfall model fails. This model assumes that the requirements are

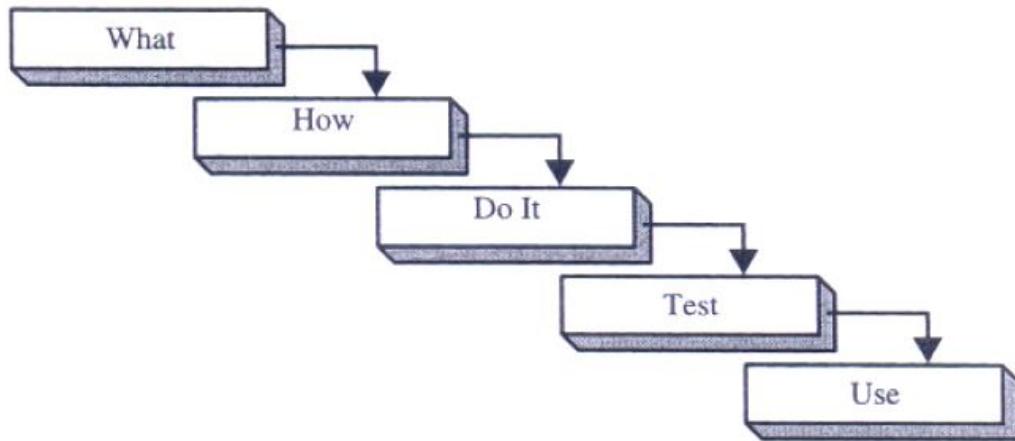


FIGURE 3-2 The waterfall software development process.

known before the design begins, but one may need experience with the product before the requirements can be fully understood. It also assumes that the requirements will remain static over the development cycle and that a product delivered months after it was specified will meet the delivery-time needs.

Finally, even when there is a clear specification, it assumes that sufficient design knowledge will be available to build the product. The waterfall model is the best way to manage a project with a well-understood product, especially very large projects. Clearly, it is based on well-established engineering principles. However, its failures can be traced to its inability to accommodate software's special properties and its inappropriateness for resolving partially understood issues; furthermore, it neither emphasizes nor encourages software reusability. After the system is installed in the real world, the environment frequently changes, altering the accuracy of the original problem statement and, consequently, generating revised software requirements. This can complicate the software development process even more. For example, a new class of employees or another shift of workers may be added or the standard workweek or the piece rate changed. By definition, any such changes also change the environment, requiring changes in the programs. As each such request is processed, system and programming changes make the process increasingly complex, since each request must be considered in regard to the original statement of needs as modified by other requests.

3.3 BUILDING HIGH-QUALITY SOFTWARE

The software process transforms the users' needs via the application domain to a software solution that satisfies those needs. Once the system (programs) exists, we must test it to see if it is free of bugs. High-quality products must meet users' needs and

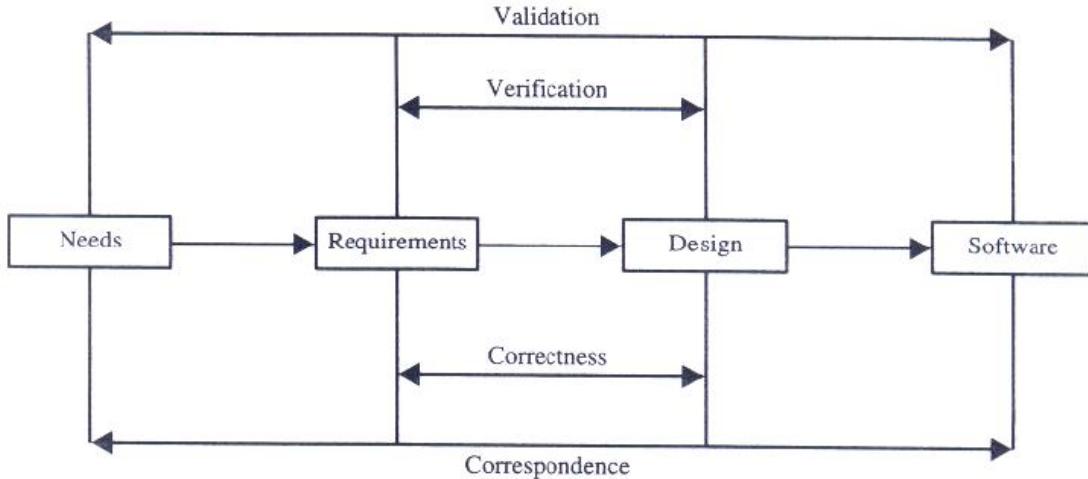
expectations. Furthermore, the products should attain this with minimal or no defects, the focus being on improving products (or services) prior to delivery rather than correcting them after delivery.

There are two basic approaches to systems testing. We can test a system according to how it has been built or, alternatively, what it should do. Blum describes a means of system evaluation in terms of four quality measures: correspondence, correctness, verification, and validation. *Correspondence* measures how well the delivered system matches the needs of the operational environment, as described in the original requirements statement.

Validation is the task of predicting correspondence. True correspondence cannot be determined until the system is in place (see Figure 3-3). *Correctness* measures the consistency of the product requirements with respect to the design specification. Blum argues that *verification* is the exercise of determining correctness. However, correctness always is objective. Given a specification and a product, it should be possible to determine if the product precisely satisfies the requirements of the specification. For example, does the payroll system accurately compute the amount of compensation? Does it report productivity accurately and to the satisfaction of the workers, and does it handle information as originally planned? Validation, however, is always subjective, and it addresses a different issue the appropriateness of the specification. This may be considered 20/20 hindsight: Did we uncover the true users' needs and therefore establish the proper design? If the evaluation criteria could be detailed, they would have been included in the specification. Boehm observes that these quality measures, verification and validation, answer the following questions:.

- *Verification*. Am I building the product right?
- *Validation*. Am I building the right product?

Validation begins as soon as the project starts, but verification can begin only after a specification has been accepted. Verification and validation are independent of each other. It is possible to have a product that corresponds to the specification, but if the specification proves to be incorrect, we do not have the right product; for example, say a necessary report is missing from the delivered product, since it was not included in the original specification. A product also may be correct but not correspond to the users' needs; for example, after years of waiting, a system is delivered that satisfies the initial design statement but no longer reflects current operating practices. Blum argues that, when the specification is informal, it is difficult to separate verification from validation. Future chapter looks at the issue of software validation and correspondence by proposing a way to measure user satisfaction

**FIGURE 3-3**

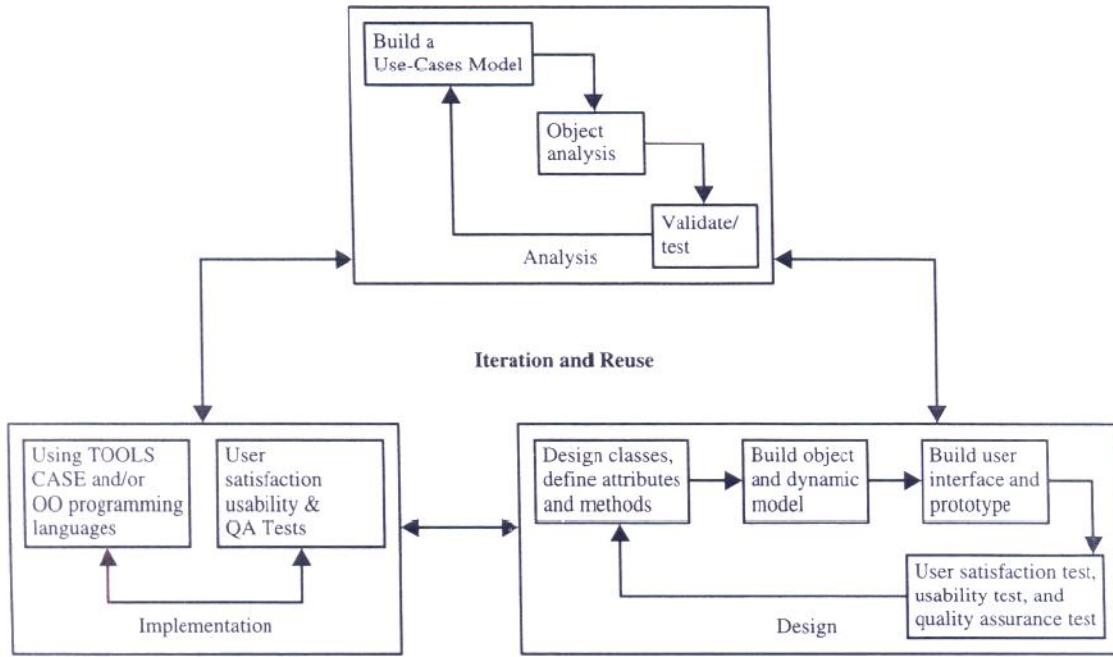
Four quality measures (correspondence, correctness, validation, and verification) for software evaluation.

3.4 OBJECT ORIENTED SYSTEMS DEVELOPMENT: A USE-CASE DRIVEN APPROACH

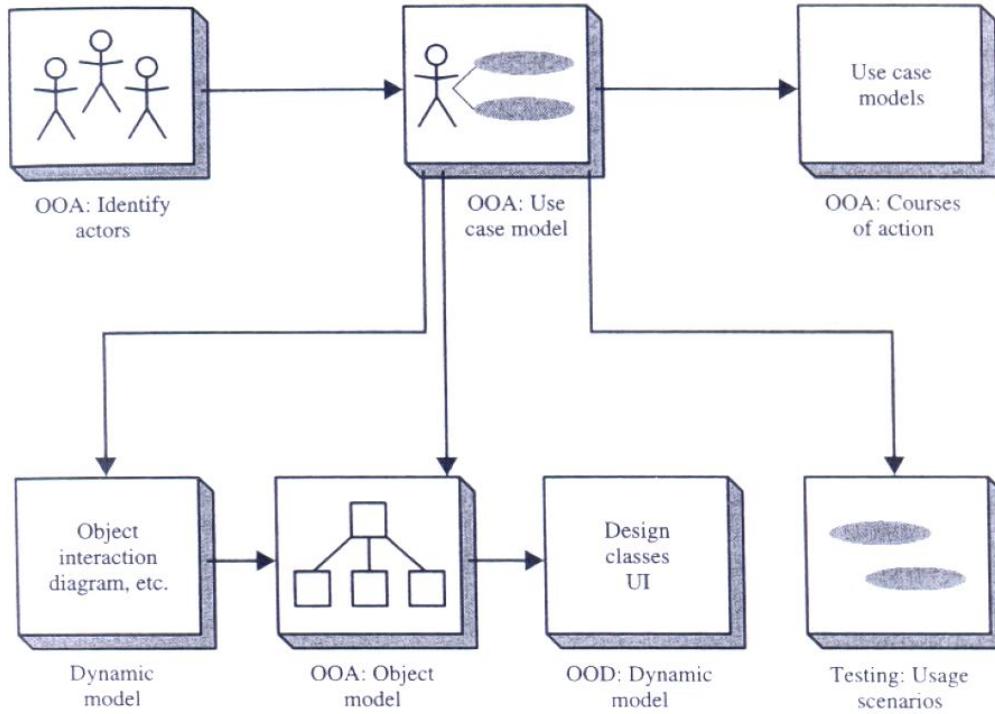
The object-oriented *software development life cycle* (SDLC) consists of three macro processes: object-oriented analysis, object-oriented design, and object-oriented implementation (see Figure 3-4).

The use-case model can be employed throughout most activities of software development. Furthermore, by following the life cycle model of Jacobson, Ericsson, and Jacobson one can produce designs that are traceable across requirements, analysis, design, implementation, and testing (as shown in Figure 3-5). The main advantage is that all design decisions can be traced back directly to user requirements. Usage scenarios can become test scenarios. Object-oriented system development includes these activities:

- Object-oriented analysis-use case driven
- Object-oriented design
- Prototyping
- Component-based development
- Incremental testing

**FIGURE 3-4**

The object-oriented systems development approach. Object-oriented analysis corresponds to transformation 1; design to transformation 2, and implementation to transformation 3 of Figure 3-1.

**FIGURE 3-5**

By following the life cycle model of Jacobson et al., we produce designs that are traceable across requirements, analysis, implementation, and testing.

Object-oriented software development encourages you to view the problem as a system of cooperative objects. Furthermore, it advocates incremental development. Although object-oriented software development skills come only with practice, by following the guidelines listed in this book you will be on the right track for building sound applications. We look at these activities in detail in subsequent chapters.

Object-Oriented Analysis-Use-Case Driven

The object-oriented analysis phase of software development is concerned with determining the system requirements and identifying classes and their relationship to other classes in the problem domain. To understand the system requirements, we need to identify the users or the actors. Who are the actors and how do they use the system? In object-oriented as well as traditional development, scenarios are used to help analysts understand requirements. However, these scenarios may be treated informally or not fully documented. Ivar Jacobson came up with the concept of the *use case*, his name for a scenario to describe the user computer system interaction. The concept worked so well that it became a primary element in system development.

The object-oriented programming community has adopted use cases to a remarkable degree. Scenarios are a great way of examining who does what in the interactions among objects and what *role* they play; that is, their interrelationships. This intersection among objects' roles to achieve a given goal is called *collaboration*. The scenarios represent only one possible example of the collaboration. To understand all aspects of the collaboration and all potential actions, several different scenarios may be required, some showing usual behaviors, others showing situations involving unusual behavior or exceptions. In essence, a use case is a typical interaction between a user and a system that captures users' goals and needs. In its simplest usage, you capture a use case by talking to typical users, discussing the various things they might want to do with the system.

Expressing these high-level processes and interactions with customers in a scenario and analyzing it is referred to as *use-case modeling*. The use-case model represents the users' view of the system or users' needs. For example, consider a word processor, where a user may want to be able to replace a word with its synonym or create a hyperlink. These are some uses of the system, or a system responsibility.

This process of developing use cases, like other object-oriented activities, is iterative-once your use-case model is better understood and developed you should start to identify classes and create their relationships.

Looking at the physical objects in the system also provides us important information on objects in the systems. The objects could be individuals, organizations, machines, units of information, pictures, or whatever else makes up the application and

makes sense in the context of the real-world system. While developing the model, objects emerge that help us establish a workable system. It is necessary to work iteratively between use-case and object models. For example, the objects in the incentive payroll system might include the following examples:

- The employee, worker, supervisor, office administrator.
- The paycheck.
- The product being made.
- The process used to make the product.

Of course, some problems have no basis in the real world. In this case, it can be useful to pose the problem in terms of analogous physical objects, kind of a mental simulation. the context of the application's domain. For example, the application domain might be a payroll system; and the tangible objects might be the paycheck, employee, worker, supervisor, office administrator; and the intangible objects might be tables, data entry screen, data structures, and so forth. Documentation is another important activity, which does not end with object-oriented analysis but should be carried out throughout the system development. However, make the documentation as short as possible. The 80-20 rule generally applies for documentation: 80 percent of the work can be done with 20 percent of the documentation. The trick is to make sure that the 20 percent is easily accessible and the rest (80 percent) is available to those (few) who need to know. Remember that documentation and modeling are not separate activities, and good modeling implies good documentation.

3.5 LET US SUM UP

This chapter introduces the system development life cycle (SOLC) in general and object-oriented and use-case driven SOLC specifically. The essence of the software process is the transformation of users' needs through the application domain into a software solution that is executed in the implementation domain. The concept of the use case, or a set of scenarios, can be a valuable tool for understanding the users' needs. The emphasis on the analysis and design aspects of the software life cycle is intended to promote building high-quality software (meeting the specifications and being adaptable for change).

3.6 POINTS FOR DISCUSSIONS

1. Analyze software development life cycle.
2. Evaluate building high quality software

3.6 LESSON – END ACTIVITIES

1. Discuss about Use case Driven approach

3.7 REFERENCES

1. Norman,Ronald- object oriented system analysis and design –prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 4: OBJECT ORIENTED DESIGN

CONTENTS

- 4.0 AIMS AND OBJECTIVES
- 4.1 INTRODUCTION
- 4.2 OBJECT-ORIENTED DESIGN
- 4.3 PROTOTYPING
- 4.4 IMPLEMENTATION: COMPONENT-BASED DEVELOPMENT
- 4.5 INCREMENTAL TESTING
- 4.6 REUSABILITY
- 4.7 LET US SUM UP
- 4.8 POINTS FOR DISCUSSION
- 4.9 LESSON – END ACTIVITIES
- 4.10 REFERENCES

4.0 AIM AND OBJECTIVES

The main objective of this lesson to know the details about the

- Object-Oriented Design
- Prototyping
- Implementation of the object oriented models

4.1 INTRODUCTION

The goal of *object-oriented design* (OOD) is to design the classes identified during the analysis phase and the user interface. During this phase, we identify and define additional objects and classes that support implementation of the requirements . For example, during the design phase, you might need to add objects for the user interface to the system (e.g., data entry windows, browse windows).

4.2 OBJECT-ORIENTED DESIGN

Object-oriented design and object-oriented analysis are distinct disciplines, but they can be intertwined. Object-oriented development is highly incremental; in other words, you start with object-oriented analysis, model it, create an object-oriented design, then do some more of each, again and again, gradually refining and completing models of the system. The activities and focus of object-oriented analysis and object-oriented design are intertwined-grown, not built (see Figure 3-4). First, build the object model based on

objects and their relationships, then iterate and refine the model: .Design and refine classes. .Design and refine attributes. .Design and refine methods. .Design and refine structures. .Design and refine associations.

Here are a few guidelines to use in your object-oriented design: .Reuse, rather than build, a new class. Know the existing classes. .Design a large number of simple classes, rather than a small number of complex classes. Design methods. Critique what you have proposed. If possible, go back and refine the classes.

4.3 PROTOTYPING

Although the object-oriented analysis and design describe the system features, it is important to construct a prototype of some of the key system components shortly after the products are selected. It has been said "a picture may be worth a thousand words, but a prototype is worth a thousand pictures"[author unknown]. Not only is this true, it is an understatement of the value of software prototyping. Essentially, a prototype is a version of a software product developed in the early stages of the product's life cycle for specific, experimental purposes. A prototype enables you to fully understand how easy or difficult it will be to implement some of the features of the system. It also can give users a chance to comment on the usability and usefulness of the user interface design and lets you assess the fit between the software tools selected, the functional specification, and the user needs. Additionally, prototyping can further define the use cases, and it actually makes use-case modeling much easier.

Building a prototype that the users are happy with, along with documentation of what you did, can define the basic courses of action for those use cases covered by the prototype. The main idea here is to build a prototype with uses-case modeling to design systems that users like and need. Traditionally, prototyping was used as a "quick and dirty" way to test the design, user interface, and so forth, something to be thrown away when the "industrial strength" version was developed. However, the new trend, such as using rapid application development, is to refine the prototype into the final product. Prototyping provides the developer a means to test and refine the user interface and increase the usability of the system. As the underlying prototype design begins to become more consistent with the application requirements, more details can be added to the application, again with further testing, evaluation, and rebuilding, until all the application components work properly within the prototype framework.

Prototypes have been categorized in various ways. The following categories are some of the commonly accepted prototypes and represent very distinct ways of viewing a prototype, each having its own strengths:

- Horizontal Prototype
- Vertical Prototype
- Analysis Prototype

A *horizontal prototype* is a simulation of the interface (that is, it has the entire user interface that will be in the full-featured system) but contains no functionality. This has the advantages of being very quick to implement, providing a good overall feel of the system, and allowing users to evaluate the interface on the basis of their normal, expected perception of the system.

A *vertical prototype* is a subset of the system features with complete functionality.

The principal advantage of this method is that the few implemented functions can be tested in great depth. In practice, prototypes are a hybrid between horizontal and vertical: The major portions of the interface are established so the user can get the feel of the system, and features having a high degree of risk are prototyped with much more functionality.

An *analysis prototype* is an aid for exploring the problem domain. This class of prototype is used to inform the user and demonstrate the proof of a concept. It is not used as the basis of development, however, and is discarded when it has served its purpose. The final product will use the concepts exposed by the prototype, not its code. A *domain prototype* is an aid for the incremental development of the ultimate software solution. It often is used as a tool for the staged delivery of subsystems to the users or other members of the development team. It demonstrates the feasibility of the implementation and eventually will evolve into a deliverable product.

The typical time required to produce a prototype is anywhere from a few days to several weeks, depending on the type and function of prototype. Prototyping should involve representation from all user groups that will be affected by the project, especially the end users and management members to ascertain that the general structure of the prototype meets the requirements established for the overall design. The purpose of this review is threefold:

1. To demonstrate that the prototype has been developed according to the specification and that the final specification is appropriate.
2. To collect information about errors or other problems in the system, such as user interface problems that need to be addressed in the intermediate prototype stage.
3. To give management and everyone connected with the project the first (or it could be second or third...) glimpse of what the technology can provide. The evaluation can be performed easily if the necessary supporting data is readily available. Testing considerations must be incorporated into the design and subsequent implementation of the system.

Prototyping is a useful exercise at almost any stage of the development. In fact, prototyping should be done in parallel with the preparation of the functional specification. As key features are specified, prototyping those features usually results in

modifications to the specification and even can reveal additional features or problems that were not obvious until the prototype was built.

4.4 IMPLEMENTATION: COMPONENT-BASED DEVELOPMENT

Manufacturers long ago learned the benefits of moving from custom development to assembly from prefabricated components. Component-based manufacturing makes many products available to the marketplace that otherwise would be prohibitively expensive. If products, from automobiles to plumbing fittings to PCs, were custom-designed and built for each customer, the way business applications are, then large markets for these products would not exist. Low-cost, high-quality products would not be available. Modern manufacturing has evolved to exploit two crucial factors underlying today's market requirements: reduce cost and time to market by building from pre-built, ready-tested components, but add value and differentiation by rapid customization to targeted customers

Today, software components are built and tested in-house, using a wide range of technologies. For example, computer-aided software engineering (CASE) tools allow their users to rapidly develop information systems. The main goal of CASE technology is the automation of the entire information system's development life cycle process using a set of integrated software tools, such as modeling, methodology, and automatic code generation. However, most often, the code generated by CASE tools is only the skeleton of an application and a lot needs to be filled in by programming by hand.

A new generation of CASE tools is beginning to support component-based development. *Component-based development* (CBD) is an industrialized approach to the software development process. Application development moves from custom development to assembly of prebuilt, pretested, reusable software components that operate with each other. Two basic ideas underlie component-based development.

First, the application development can be improved significantly if applications can be assembled quickly from prefabricated software components. Second, an increasingly large collection of interpretable software components could be made available to developers in both general and specialist catalogs. Put together, these two ideas move application development from a craft activity to an industrial process fit to meet the needs of modern, highly dynamic, competitive, global businesses. The industrialization of application development is akin to similar transformations that occurred in other human endeavors.

A CBD developer can assemble components to construct a complete software system. Components themselves may be constructed from other components and so on down to the level of prebuilt components or old-fashioned code written in a language such as C, assembler, or COBOL.

Visual tools or actual code can be used to "glue" together components. Although it is practical to do simple applications using only "visual glue" (e.g., by "wiring" components together as in Digitalk's Smalltalk PARTS, or IBM's VisualAge), putting together a practical application still poses some challenges. Of course, all these are "under the hood" and should be invisible to end users. The impact to users will come from faster product development cycles, increased flexibility, and improved customization features. CBD will allow independently developed applications to work together and do so more efficiently and with less development effort.

Existing (legacy) applications support critical services within an organization and therefore cannot be thrown away. Massive rewriting from scratch is not a viable option, as most legacy applications are complex, massive, and often poorly documented. The CBD approach to legacy integration involves application wrapping, in particular component wrapping, technology.

An application wrapper surrounds a complete system, both code and data. This wrapper then provides an interface that can interact with both the legacy and the new software systems (see Figure 3-6). Off-the-shelf application wrappers are not widely available. At present, most application wrappers are homegrown within organizations. However, with component-based development technology emerging rapidly, component wrapper technology will be used more widely.

The *software components* are the functional units of a program, building blocks offering a collection of reusable services. A software component can request a service from another component or deliver its own services on request. The delivery of services is independent, which means that components work together to accomplish a task. Of course, components may depend on one another without interfering with each other. Each component is unaware of the context or inner workings of the other components. In short, the object-oriented concept addresses analysis,

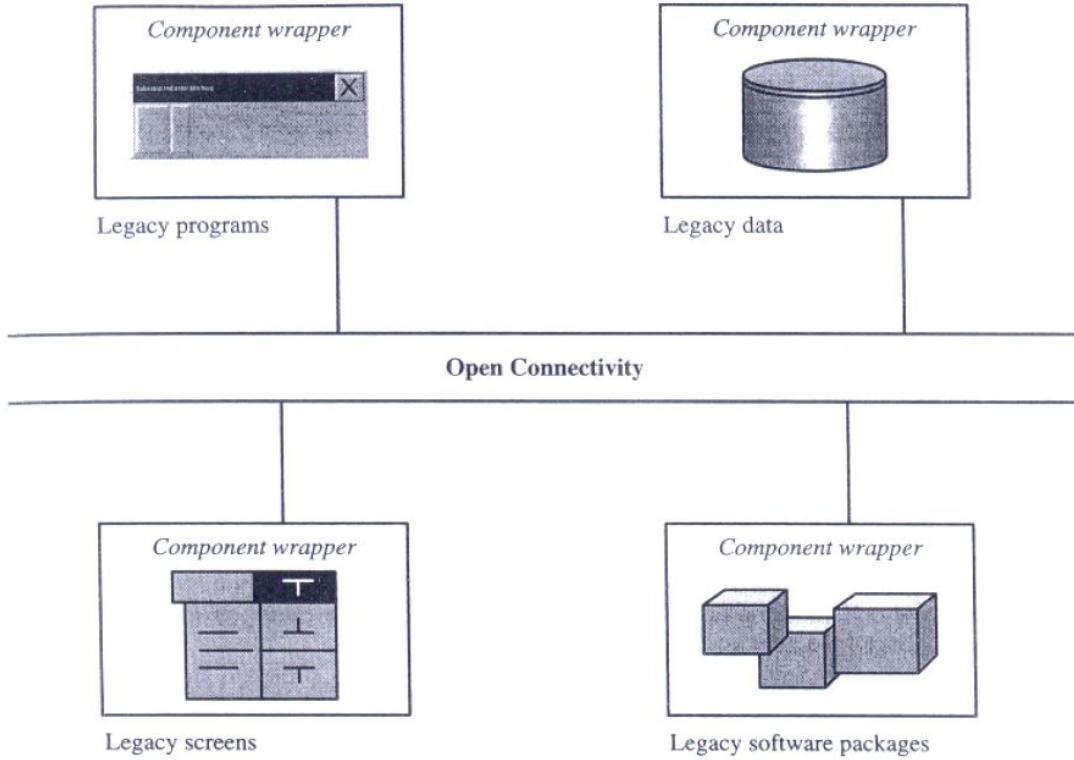


FIGURE 4.1
Reusing legacy system via component wrapping technology.

design, and programming, whereas component-based development is concerned with the implementation and system integration aspects of software development. *Rapid application development* (RAD) is a set of tools and techniques that can be used to build an application faster than typically possible with traditional methods. The term often is used in conjunction with software prototyping. It is widely held that, to achieve RAD, the developer sacrifices the quality of the product for a quicker delivery. This is not necessarily the case.

RAD is concerned primarily with reducing the "time to market," not exclusively the software development time. In fact, one successful RAD application achieved a substantial reduction in time to market but realized no significant reduction in the individual software cycles.

RAD does not replace the system development life cycle (see the Real-World case) but complements it, since it focuses more on process description and can be combined perfectly with the object-oriented approach. The task of RAD is to build the application quickly and incrementally implement the design and user requirements, through tools such as Delphi, VisualAge, Visual Basic, or PowerBuilder. After the overall design for an application has been completed, RAD begins. The main objective of RAD is to build a version of an application rapidly to see whether we actually have understood the problem (analysis). Further, it determines whether the system does what it

is supposed to do (design). RAD involves a number *of* iterations. Through each iteration we might understand the problem a little better make an improvement. RAD encourages the incremental development approach of “grow, do not Build” software.

4.5 INCREMENTAL TESTING

If you wait until after development to test an application for bugs and performance, you could be wasting thousands of dollars and hours of time. That's what happened at Bankers Trust in 1992: "Our testing was very complete and good, but it was costing a lot of money and would add months onto a project," says Glenn Shimamoto, vice president of technology and strategic planning at the New York bank . In one case, testing added nearly six months to the development of a funds transfer application. The problem was that developers would turn over applications to a quality assurance (QA) group for testing only after development was completed. Since the QA group wasn't included in the initial plan, it had no clear picture of the system characteristics until it came time to test.

4.6 REUSABILITY

A major benefit of object-oriented system development is reusability, and this is the most difficult promise to deliver on. For an object to be really reusable, much more effort must be spent designing it. To deliver a reusable object, the development team must have the up-front time to design reusability into the object. The potential benefits of reuse are clear: increased reliability, reduced time and cost for development, and improved consistency. You must effectively evaluate existing software components for reuse by asking the following questions as they apply to the intended applications

- Has my problem already been solved?
- Has my problem been partially solved?
- What has been done before to solve a problem similar to this one?

To answer these questions, we need detailed summary information about existing software components. In addition to the availability of the information, we need some kind of search mechanism that allows us to define the candidate object simply and then generate broadly or narrowly defined queries. Thus, the ideal system for reuse would function like a skilled reference librarian. If you have a question about a subject area, all potential sources could be identified and the subject area could be narrowed by prompting. Some form of browsing with the capability to provide detailed information would be required, one where specific subjects could be looked up directly. The reuse strategy can be based on the following:

- .Information hiding (encapsulation).
- .Conformance to naming standards.
- .Creation and administration of an object repository.
- .Encouragement by strategic management of reuse as opposed to constant redevelopment.

- .Establishing targets for a percentage of the objects in the project to be reused (i.e., 50 percent reuse of objects).

4.7 LET US SUM UP

High-quality software provides users with an application that meets their needs and expectations. Four quality measures have been described: correspondence, correctness, verification, and validation. Correspondence measures how well the delivered system corresponds to the needs of the problem. Correctness determines whether or not the system correctly computes the results based on the rules created during the system analysis and design, measuring the consistency of product requirements with respect to the design specification. Verification is the task of determining correctness (am I building the *product right?*). Validation is the task of predicting correspondence (am I building the *right product?*).

Object-oriented design requires more rigor up front to do things right. You need to spend more time gathering requirements, developing a requirements model and an analysis model, then turning them into the design model. Now, you can develop code quickly—you have a recipe for doing it. Object-oriented systems development consists of three macro processes: object-oriented analysis, object-oriented design, and object-oriented implementation.

Component-based development (CBO) is an industrialized approach to software development. Software components are functional units, or building blocks offering a collection of reusable services. A CBO developer can assemble components to construct a complete software system. Components themselves may be constructed from other components and so on down to the level of prebuilt components or old-fashioned code written in a language such as C, assembler, or COBOL. The object-oriented concept addresses analysis, design, and programming; whereas component-based development is concerned with the implementation and system integration aspects of software development.

The rapid application development (RAO) approach to systems development rapidly develops software to quickly and incrementally implement the design by using tools such as CASE.

4.8 POINTS FOR DISCUSSION

1. Discuss about reusability
2. Justify prototyping

4.9 LESSON – END ACTIVITIES

1. Validate object oriented design.
2. Discuss about component – based development approach.

4.10 REFERENCES

1. Norman, Ronald- object oriented system analysis and design –prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 5: METHODOLOGIES

CONTENTS

- 5.0 AIMS AND OBJECTIVES
- 5.1 INTRODUCTION
- 5.2 TOWARDS UNIFICATION-TOO MANY METHODOLOGIES
- 5.3 SURVEY OF SOME OF THE OBJECT-ORIENTED METHODOLOGIES
- 5.4 LET US SUM UP
- 5.5 POINTS FOR DISCUSSION
- 5.6 LESSON – END ACTIVITIES
- 5.7 REFERENCES

5.0 AIMS AND OBJECTIVES

The main objective of this lesson is to know about the concepts of Object-oriented methodology and the overview of the various object oriented methodology.

5.1 INTRODUCTION

Object-oriented methodology is a set of methods, models, and rules for developing systems. Modeling is the process of describing an existing or proposed system. It can be used during any phase of the software life cycle. A model is an abstraction of a phenomenon for the purpose of understanding it. Since a model excludes unnecessary details; it is easier to manipulate than the real object. Modeling provides a means for communicating ideas in an easy to understand and unambiguous form while also accommodating a system's complexity. In this part we will look at

5.2 TOWARD UNIFICATION-TOO MANY METHODOLOGIES

In the 1980s, many methodologists were wondering how analysis and design methods and processes would fit into an object-oriented world. Object-oriented methods suddenly had become very popular, and it was apparent that the techniques to help people execute good analysis and design were just as important as the object-oriented concept itself.

To get a feel for object-oriented methodologies, let us look at some of the methods developed in the 1980s and 1990s. This list by no means is complete

- 1986 . Booch developed the object-oriented design concept, the Booch method.
- 1987. Sally Shlaer and Steve Mellor created the concept of the recursive design approach.

- 1989. Beck and Cunningham produced class-responsibility-collaboration cards. .
- 1990. Wirfs-Brock, Wilkerson, and Wiener came up with responsibility driven design.
- 1991. Jim Rumbaugh led a team at the research labs of General Electric to develop the object modeling technique (OMT) .
- 1991. Peter Coad and Ed Yourdon developed the Coad lightweight and prototype-oriented approach to methods.
- 1994. Ivar Jacobson introduced the concept of the use case and object-oriented software engineering (OOSE).

These methodologies and many other forms of notational language provided system designers and architects many choices but created a much split, competitive, and confusing environment. Most of the methods were very similar but contained a number of often annoying minor differences, and each had a group of practitioners that liked its ideas. The same basic concepts appeared in very different notations, which caused confusion among the users . The trend in object-oriented methodologies, sometimes called *second-generation object-oriented methods*, has been toward combining the best aspects of the most popular methods instead of coming out with new methodologies, which was the tendency in first-generation object-oriented methods. In the next section, to give you a taste of object-oriented methodologies, we will look at some of the most popular ones.

5.3 SURVEY OF SOME OF THE OBJECT-ORIENTED METHODOLOGIES

Many methodologies are available to choose from for system development. Each methodology is based on modeling the business problem and implementing the application in an object-oriented fashion; the differences lie primarily in the documentation of information and modeling notations and language. An application can be implemented in many ways to meet the same requirements and provide the same functionality. The largest noticeable differences will be in the trade-offs and detailed design decisions made. Two people using the same methodology may produce application designs that look radically different. This does not necessarily mean that one is right and one is wrong, just that they are different. In the following sections, we look at the methodologies and their modeling notations developed by Rumbaugh et al., Booch, and Jacobson which are the origins of the Unified Modeling Language (UML).

Each method has its strengths. The Rumbaugh et al. method is well-suited for describing the object model or the static structure of the system. The Jacobson et al. method is good for producing user-driven analysis models. The Booch method produces detailed object-oriented design models.

5.4 LET US SUM UP

Object-oriented analysis requires building a use-case model and interaction diagrams to identify users' needs and the system's classes and their responsibility, then validating and testing the model, documenting each step along the way. Object-oriented design centers on establishing design classes and their protocol; building class diagrams, user interfaces, and prototypes; testing user satisfaction and usability based on usage and use cases. The use-case concept can be employed through most of the activities of software development. Furthermore, by following Jacobson's life cycle model, one can produce designs that are traceable across requirements, analysis, design, implementation, and testing.

5.5 POINTS FOR DISCUSSION

1. Discuss towards the unification-too many methodologies

5.6 LESSON - END ACTIVITIES

1. Evaluate the survey of some of the object-oriented methodologies

5.7 REFERENCES

1. Norman,Ronald- object oriented system analysis and design –prentice hall 1996.
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press.
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press.
4. Rumbaugh, James ,Michael –Object oriented Modelling and design.
5. Ali –Brahmi –Object oriented system development”.

UNIT – II

LESSON – 6: OBJECT ORIENTED METHODOLOGIES

CONTENTS

- 6.0 AIM AND OBJECTIVE
- 6.1 INTRODUCTION.
- 6.2 RUMBAUGH'S OBJECT MODELING TECHNIQUE
- 6.3 THE OBJECT MODEL
- 6.4 THE OMT DYNAMIC MODEL
- 6.5 THE OMT FUNCTIONAL MODEL
- 6.6 LET US SUM UP
- 6.7 POINTS FOR DISCUSSION
- 6.8 LESSON – END ACTIVITIES
- 6.9 REFERENCES

6.0 AIM AND OBJECTIVE

You should be able to define and understand object-oriented methodologies.

Object-oriented methodology is a set of methods, models, and rules for developing systems.

6.1 INTRODUCTION.

SURVEY OF SOME OF THE OBJECT ORIENTED METHODOLOGIES

Many methodologies are available to choose from for system development. Each methodology is based on modeling the business problem and implementing the application in an object-oriented fashion; the differences lie primarily in the documentation of information and modeling notations and language. An application can be implemented in many ways to meet the same requirements and provide the same functionality. The largest noticeable differences will be in the trade-offs and detailed design decisions made. Two people using the same methodology may produce application designs that look radically different. This does not necessarily mean that one is right and one is wrong, just that they are different. In the following sections, we look at the methodologies and their modeling notations developed by Rumbaugh Booch, and Jacobson which are the origins of the Unified Modeling Language (UML). Each method has its strengths. The Rum Baugh et al. method is well-suited for describing the object model or the static structure of the system. The Jacobson method is good for producing

user-driven analysis models. The Booch method produces detailed object-oriented design models.

6.2 RUMBAUGH'S OBJECT MODELING TECHNIQUE

The object modeling technique (OMT) presented by Jim Rumbaugh and his coworkers describes a method for the analysis, design, and implementation of a system using an object-oriented technique. OMT is a fast, intuitive approach for identifying and modeling all the objects making up a system. The dynamic behavior of objects within a system can be described using the OMT dynamic model. This model lets you specify detailed state transitions and their descriptions within a system. Finally, a process description and consumer-producer relationships can be expressed using OMT's functional model. OMT consists of four phases, which can be performed iteratively:

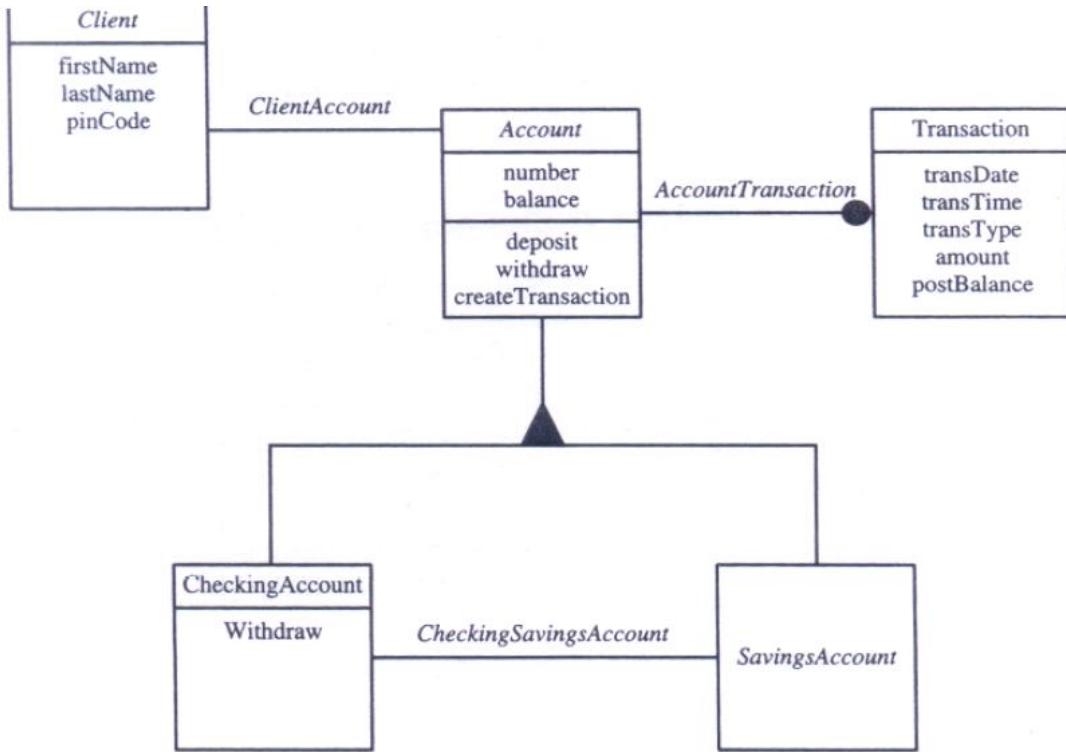
1. *Analysis*. The results are objects and dynamic and functional models.
2. *System design*. The results are a structure of the basic architecture of the system along with high-level strategy decisions.
3. *Object design*. This phase produces a design document, consisting of detailed objects static, dynamic, and functional models.
4. *Implementation*. This activity produces reusable, extendible, and robust code. OMT separates modeling into three different parts:
 1. An *object model*, presented by the object model and the data dictionary.
 2. A *dynamic model*, presented by the state diagrams and event flow diagrams.
 3. A *functional model*, presented by data flow and constraints.

6.3 THE OBJECT MODEL

The object model describes the structure of objects in a system: their identity, relationships to other objects, attributes, and operations. The object model is represented graphically with an object diagram (see Fig.). The object diagram contains classes interconnected by association lines. Each class represents a set of individual objects. The association lines establish relationships among the classes. Each association line represents a set of links from the objects of one class to the objects of another class.

6.4 THE OMT DYNAMIC MODEL

OMT provides a detailed and comprehensive dynamic model, in addition to letting you depict states, transitions, events, and actions. The OMT state transition diagram is a network of states and events (see Fig.). Each state receives one or more events, at which time it makes the transition to the next state. The next state depends on the current state as well as the events.



The OMT object model of a bank system. The boxes represent classes and the filled triangle represents specialization. Association between Account and transaction is one too many; since one account can have many transactions, the filled circle represents many (zero or more). The relationship between Client and Account classes is one to one: A client can have only one account and account can belong to only one person (in this model joint accounts are not allowed).

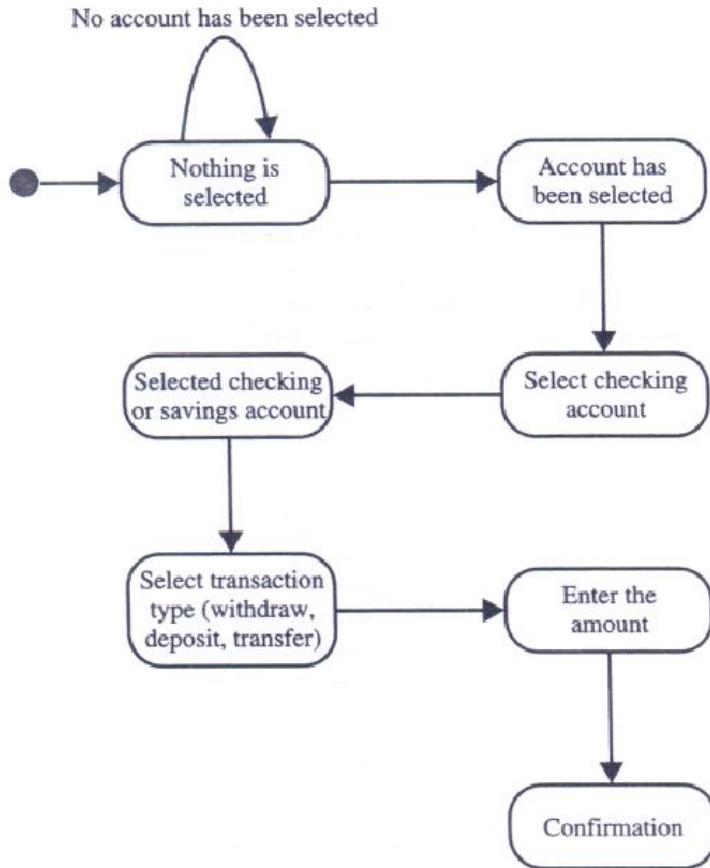
6.5 THE OMT FUNCTIONAL MODEL

The OMT data flow diagram (DFD) shows the flow of data between different processes in a business. An OMT DFD provides a simple and intuitive method for describing business processes without focusing on the details of computer systems.

Data flow diagrams use four primary symbols:

1. The *process* is any function being performed; for example, verify Password or PIN in the ATM system (see Fig).
2. The *data flow* shows the direction of data element movement; for example, PIN code.
3. The *data store* is a location where data are stored; for example, account is a data store in the ATM example.
4. An *external entity* is a source or destination of a data element; for example, the ATM card reader.

Overall, the Rumbaugh et al. OMT methodology provides one of the strongest tool sets for the analysis and design of object-oriented systems.



State transition diagram for the bank application user interface. The round boxes represent states and the arrows represent transitions.

6.6 LET US SUM UP

Each method has its strengths. Rumbaugh et al. have a strong method for producing object models (sometimes known as *domain object models*). Jacobson et al. have a strong method for producing user-driven requirement and object-oriented analysis models. Booch has a strong method for producing detailed object-oriented design models.

6.7 POINTS FOR DISCUSSION

1. Validate Object Model.
2. Evaluate OMT Dynamic Model.

6.8 LESSON – END ACTIVITIES

1. Discuss Rumbaugh's object modeling technique.
2. Justify OMT Functional Model.

6.9 REFERENCES

1. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
2. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
3. Rumbaugh, James ,Michael –Object oriented Modelling and design
- 4.Ali –Brahmi –Object oriented system development”

LESSON – 7: BOOCH METHODOLOGY AND DEVELOPMENT PROCESS

CONTENTS

- 7.0 AIMS AND OBJECTIVES
- 7.1 INTRODUCTION
- 7.2 THE BOOCH METHODOLOGY
- 7.3 THE MACRO DEVELOPMENT PROCESS
- 7.4 THE MICRO DEVELOPMENT PROCESS
- 7.5 THE JACOBSON METHODOLOGIES
- 7.6 USE CASES
- 7.7 OBJECT-ORIENTED SOFTWARE ENGINEERING: OBJECTORY
- 7.8 OBJECT-ORIENTED BUSINESS ENGINEERING
- 7.9 PATTERNS
- 7.10 LET US SUM UP
- 7.11 POINTS FOR DISCUSSION
- 7.12 LESSON – END ACTIVITIES
- 7.13 REFERENCES

7.0 AIMS AND OBJECTIVES

You should be able to define and understand Object-oriented methodologies.
 The Rumbaugh . OMT.
 The Booch methodology.
 Jacobson's methodologies.
 Patterns.
 Frameworks.
 Unified approach (UA).

7.1 INTRODUCTION

*Anyone who observes software development cannot but be impressed by its repetitive nature. Over and over again, programmers weave a number of basic patterns: sorting, searching, reading, writing, comparing, traversing, allocating, synchronizing, and so forth. Experienced programmers know the feeling of *deja vu* so characteristic of their trade*

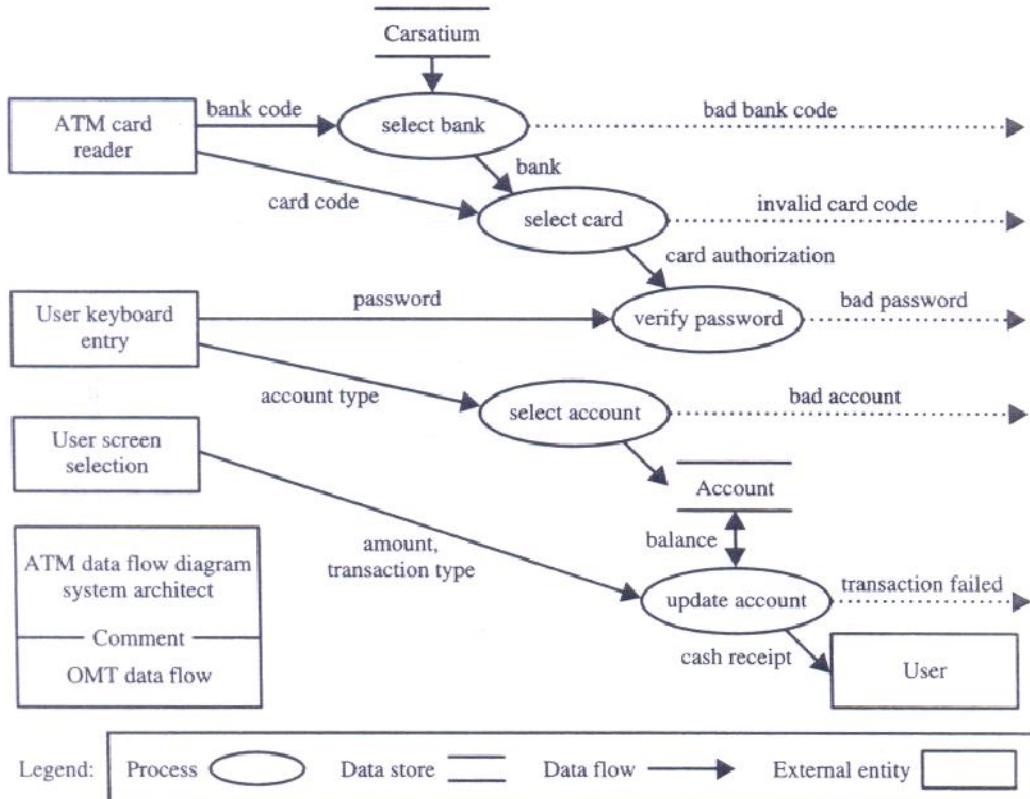
Modeling is the process of describing an existing or proposed system. It can be used during any phase of the software life cycle. A model is an abstraction of a phenomenon for the purpose of understanding it. Since a model excludes unnecessary details; it is easier to manipulate than the real object. Modeling provides a means for communicating ideas in an easy to understand and unambiguous form while also

accommodating a system's complexity. In this part we will look at Object-Oriented Methodologies

7.2 THE BOOCH METHODOLOGY

The Booch methodology is a widely used object-oriented method that helps you design your system using the object paradigm. It covers the analysis and design phases of an object-oriented system. Booch sometimes is criticized for his large set of symbols. Even though Booch defines a lot of symbols to document almost every design decision, if you work with his method, you will notice that you never use all these symbols and diagrams. You start with class and object diagrams (see Figs 4-4 and 4-5) in the analysis phase and refine these diagrams in various steps. Only when you are ready to generate code, do you add design symbols and this is where the Booch method shines, you can document your object-oriented code. The Booch method consists of the following diagrams:

- Class diagrams
- Object diagrams
- State transition diagrams
- Module diagrams
- Process diagrams
- Interaction diagrams



OMTDFDof the AT system. The data flowlines include arrows to show the direction of data element movement. The circles represent processes. The boxes represent external entities. A data store reveals the storage of data.

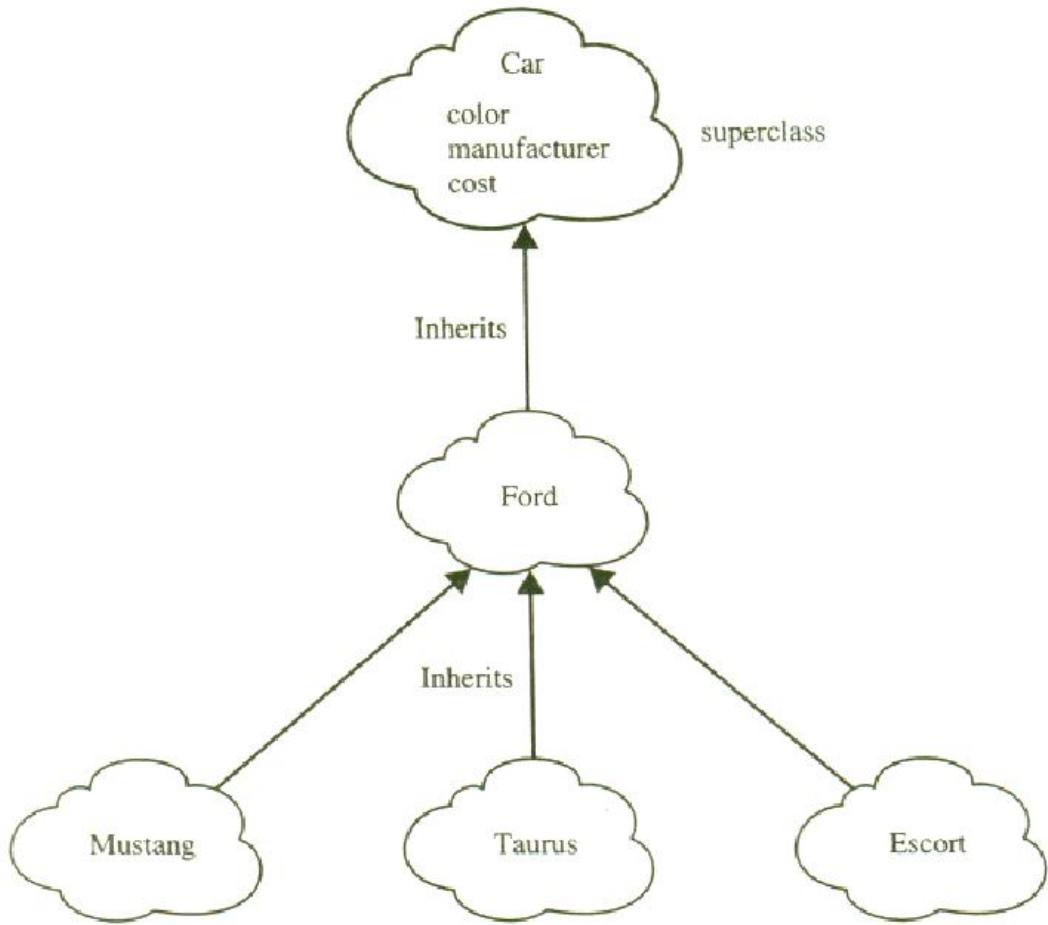
The Booch methodology prescribes a macro development process and a micro development process.

7.3 THE MACRO DEVELOPMENT PROCESS

The macro process serves as a controlling framework for the micro process and can take weeks or even months. The primary concern of the macro process is technical management of the system. Such management is interested less in the actual object-oriented design than in how well the project corresponds to the requirements set for it and whether it is produced on time. In the macro process, the traditional phases of analysis and design to a large extent are preserved.

The macro development process consists of the following steps:

1. *Conceptualization.* During conceptualization, you establish the core requirements of the system. You establish a set of goals and develop a prototype to prove the concept.
2. *Analysis and development of the model.* In this step, you use the class diagram to describe the roles and responsibilities objects are to carry out in performing



Object modeling using Booch notation. The arrows represent specialization; for example, the class Taurus is subclass of the class Ford.

the desired behavior of the system. Then, you use the object diagram to describe the desired behavior of the system in terms of scenarios or, alternatively, use the interaction diagram to describe behavior of the system in terms of scenarios.

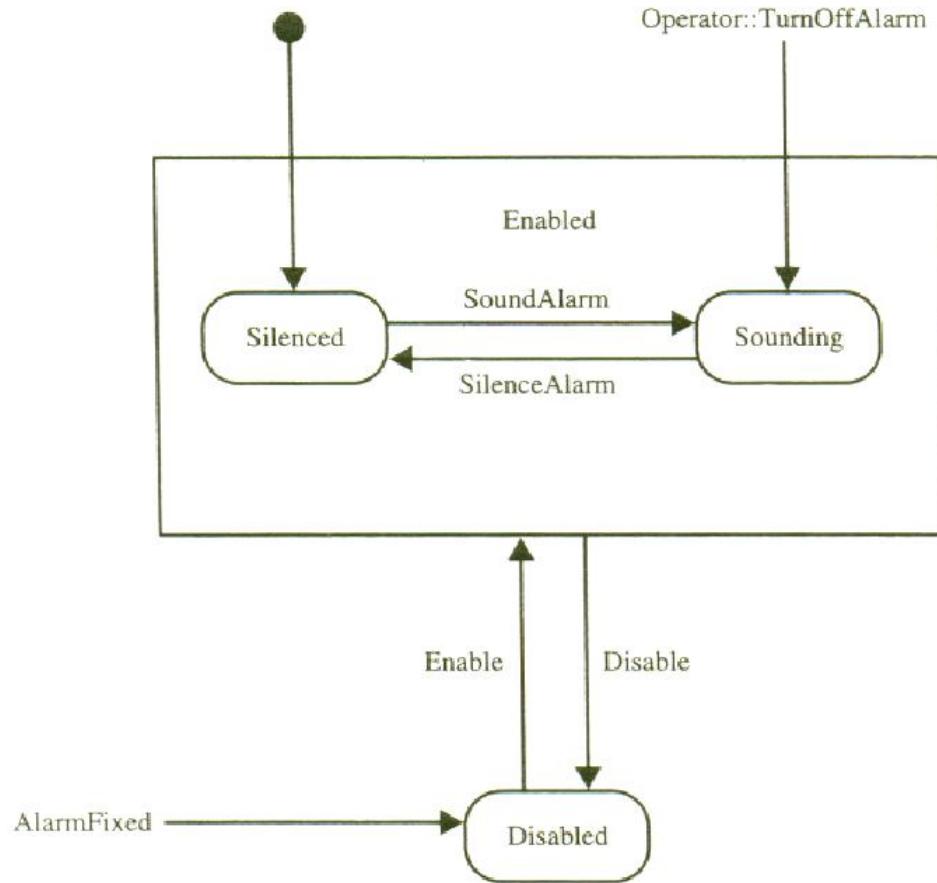
3. *Design or create the system architecture.* In the design phase, you use the class diagram to decide what classes exist and how they relate to each other. Next, you use the object diagram to decide what mechanisms are used to regulate how objects collaborate. Then, you use the module diagram to map out where each class and object should be declared. Finally, you use the process diagram to determine to which processor to allocate a process. Also, determine the schedules for multiple processes on each relevant processor.

4. *Evolution or implementation.* Successively refine the system through many iterations. Produce a stream of software implementations (or executable releases), each of which is a refinement of the prior one.

5. *Maintenance.* Make localized changes to the system to add new requirements and eliminate bugs.

7.4 THE MICRO DEVELOPMENT PROCESS

Each macro development process has its own micro development processes. The micro process is a description of the day-to-day activities by a single or small group of software developers, which could look blurry to an outside viewer, since the analysis and design phases are not clearly defined.



An alarm class state transition diagram with Booch notation. This diagram can capture the state of a class based on a stimulus. For example, a stimulus causes the class to perform some processing, followed by a transition to another state. In this case, the alarm silenced state can be changed to alarm sounding state and vice versa.

The micro development process consists of the following steps:

1. *Identify classes and objects.*
2. *Identify class and object semantics.*
3. *Identify class and object relationships.*
4. *Identify class and object interfaces and implementation.*

7.5 THE JACOBSON METHODOLOGIES

The Jacobson et al. methodologies (e.g., object-oriented Business Engineering (OOBE), object-oriented Software Engineering (OOSE), and Objectory) cover the entire

life cycle and stress traceability between the different phases, both forward and backward. This traceability enables reuse of analysis and design work, possibly much bigger factors in the reduction of development time than reuse of code.

At the heart of their methodologies is the use-case concept, which evolved with Objectory (Object Factory for Software Development).

7.6 USE CASES

Use cases are scenarios for understanding system requirements. A use case is an interaction between users and a system. The use-case model captures the goal of the user and the responsibility of the system to its users (see Fig). In the requirements analysis, the use cases are described as one of the following :

Nonformal text with no clear flow of events.

Text, easy to read but with a clear flow of events to follow (this is a recommended style).

Formal style using pseudo code.

The use case description must contain

How and when the use case begins and ends.

The interaction between the use case and its actors, including when the interaction occurs and what is exchanged.

How and when the use case will need data stored in the system or will store data in the system.

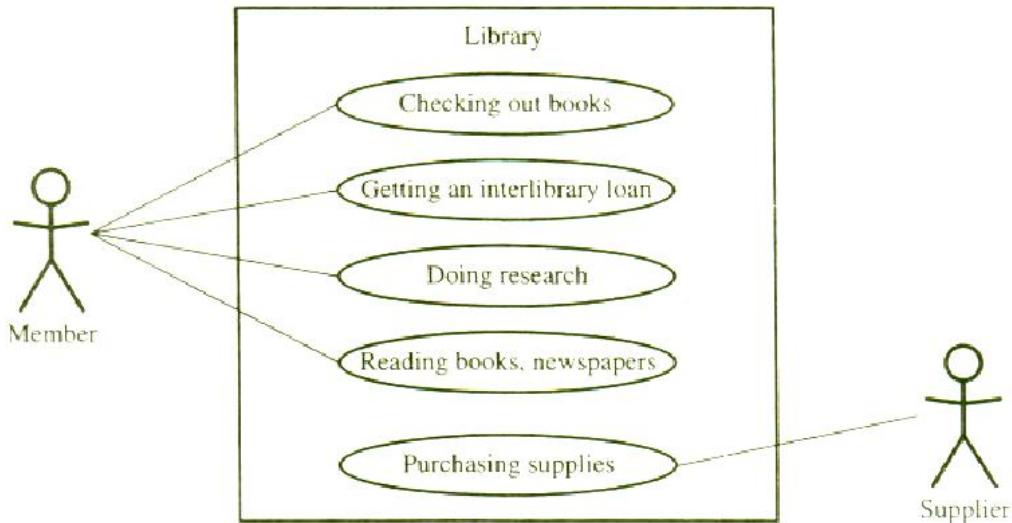
Exceptions to the flow of events.

How and when concepts of the problem domain are handled.

Every single use case should describe one main flow of events. An exceptional or additional flow of events could be added. The exceptional use case extends another use case to include the additional one. The use-case model employs extends and uses relationships. The extends relationship is used when you have one use case that is similar to another use case but does a bit more. In essence, it extends the functionality of the original use case (like a subclass). The uses relationship reuses common behavior in different use cases.

Use cases could be viewed as concrete or abstract. An *abstract use case* is not complete and has no actors that initiate it but is used by another use case. This inheritance could be used in several levels. Abstract use cases also are the ones that have uses or extends relationships.

Some uses of a library. As you can see, these are external views of the library system from an actor such as a member. The simpler the use case, the more effective it will be. It is unwise to capture all of the details right at the start; you can do that later.



the user and the responsibility of the system to its users (see Fig 4-6). In the requirements analysis, the use cases are described as one of the following:

Nonformal text with no clear flow of events.

Text, easy to read but with a clear flow of events to follow (this is a recommended style).

Formal style using pseudo code. The use case description must contain

How and when the use case begins and ends.

The interaction between the use case and its actors, including when the interaction occurs and what is exchanged.

How and when the use case will need data stored in the system or will store data in the system.

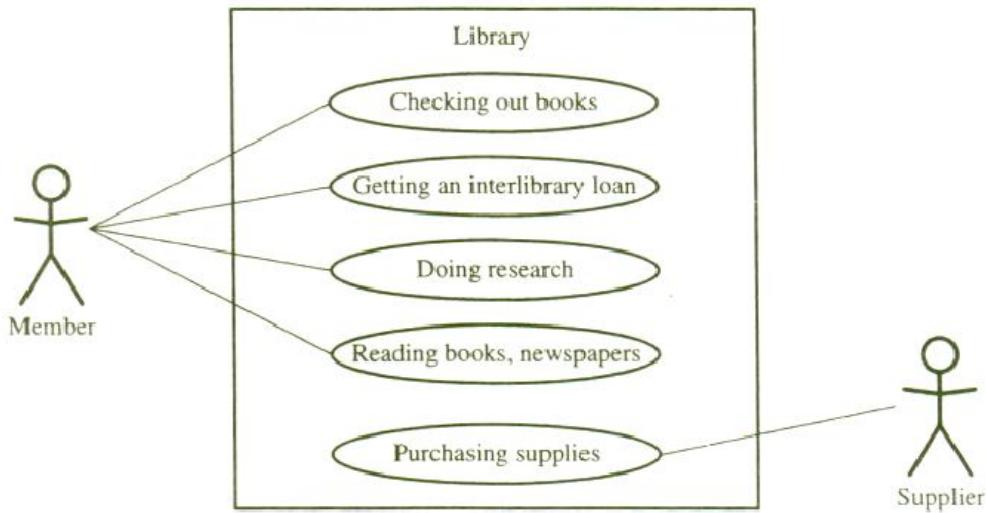
Exceptions to the flow of events.

How and when concepts of the problem domain are handled.

Every single use case should describe one main flow of events. An exceptional or additional flow of events could be added. The exceptional use case extends another use case to include the additional one. The use-case model employs extends and uses relationships. The extends relationship is used when you have one use case that is similar to another use case but does a bit more. In essence, it extends the functionality of the original use case (like a subclass). The uses relationship reuses common behavior in different use cases.

Use cases could be viewed as concrete or abstract. An *abstract use case* is not complete and has no actors that initiate it but is used by another use case. This inheritance could be used in several levels. Abstract use cases also are the ones that have uses or extends relationships.

Some uses of a library. As you can see, these are external views of the library system from an actor such as a member. The simpler the use case, the more effective it will be. It is unwise to capture all of the details right at the start; you can do that later.



7.7 OBJECT-ORIENTED SOFTWARE ENGINEERING: OBJECTORY

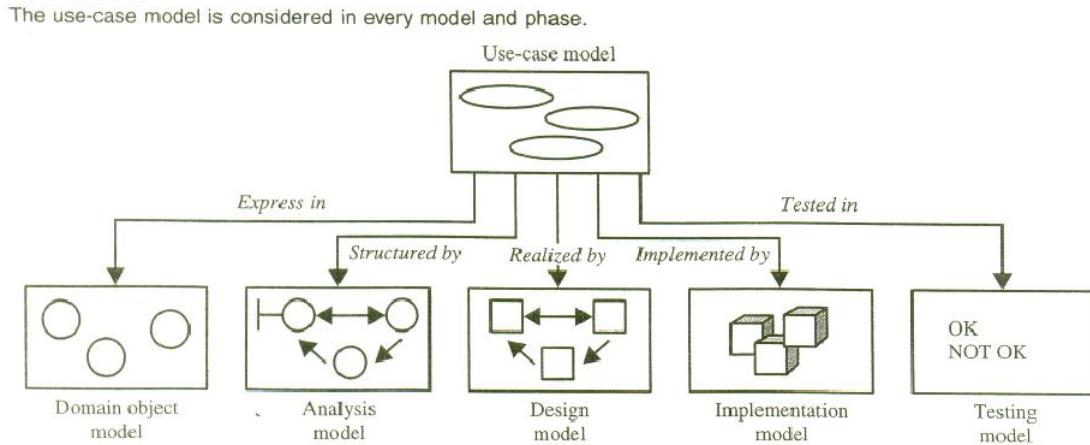
Object-oriented software engineering (OOSE), also called *Objectory*, is a method of object-oriented development with the specific aim to fit the development of large, real-time systems. The development process, called *use-case driven development*, stresses that use cases are involved in several phases of the development (see Fig), including analysis, design, validation, and testing. The use-case scenario begins with a user of the system initiating a sequence of interrelated events.

The system development method based on OOSE, Objectory, is a disciplined process for the industrialized development of software, based on a use-case driven design. It is an approach to object-oriented analysis and design that centers on understanding the ways in which a system actually is used. By organizing the analysis and design models around sequences of user interaction and actual usage scenarios, the method produces systems that are both more usable and more robust, adapting more easily to changing usage. Jacobson et al.'s Objectory has been developed and applied to numerous application areas and embodied in the CASE tool systems.

Objectory is built around several different models:

- . *Use case-model*. The use-case model defines the outside (actors) and inside (use case) of the system's behavior.
- . *Domain object model*. The objects of the "real" world are mapped into the domain object model.
- . *Analysis object model*. The analysis object model presents how the source code (implementation) should be carried out and written.
- . *Implementation model*. The implementation model represents the implementation of the system.

. *Test model.* The test model constitutes the test plans, specifications, and reports.



The maintenance of each model is specified in its associated process. A process is created when the first development project starts and is terminated when the developed system is taken out of service.

7.8 OBJECT-ORIENTED BUSINESS ENGINEERING

Object-oriented business engineering (OOBE) is object modeling at the enterprise level. Use cases again are the central vehicle for modeling, providing traceability throughout the software engineering processes. *Analysis phase.* The analysis phase defines the system to be built in terms of the problem-domain object model, the requirements model, and the analysis model. The analysis process should not take into account the actual implementation environment. This reduces complexity and promotes maintainability over the life of the system, since the description of the system will be independent of hardware and software requirements. Jacobson does not dwell on the development of the problem-domain object model, but refers the developer to Coad and Yourdon's or Booch's discussion of the topic, who suggest that the customer draw a picture of his view of the system to promote discussions. In their view, a full development of the domain model will not localize changes and therefore will not result in the most "robust and extensible structure." This model should be developed just enough to form a base of understanding for the requirements model. The analysis process is iterative but the requirements and analysis models should be stable before moving on to subsequent models. Jacobson suggest that prototyping with a tool might be useful during this phase to help specify user interfaces.

Design and implementation phases. The implementation environment must be identified for the design model. This includes factors such as Database Management System (DBMS), distribution of process, constraints due to the programming language, available component libraries, and incorporation of graphical user interface tools. It may be possible to identify the implementation environment concurrently with analysis. The

analysis objects are translated into design objects that fit the current implementation environment.

. *Testing phase.* Finally, Jacobson describes several testing levels and techniques. The levels include unit testing, integration testing, and system testing.

7.9 PATTERNS

An emerging idea in systems development is that the process can be improved significantly if a system can be analyzed, designed, and built from prefabricated and predefined system components. One of the first things that any science or engineering discipline must have is a vocabulary for expressing its concepts and a language for relating them to each other. Therefore, we need a body of literature to help software developers resolve commonly encountered, difficult problems and a vocabulary for communicating insight and experience about these problems and their solutions. The primary focus here is not so much on technology as on creating a culture to document and support sound engineering architecture and design .

In this section, we look at the concept of patterns; and in the next section, we look at another emerging method, frameworks. The use of design patterns originates in the work done by a building architect named Christopher Alexander during the late 1970s. Alexander wrote two books, *A Pattern Language* and *A Timeless Way of Building*], that, in addition to giving examples, described his rationale for documenting patterns. Alexander's articulation on pattern work was soon employed by object-oriented thinkers looking for ways to describe commonly occurring design solutions and programming paradigms.

As described in their seminal work in cataloging program design concepts, Gamma, Helm, Johnson, and Vlissides say that the design pattern identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. [Furthermore, it] identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

Another book that helped popularize the use of patterns is *Pattern-Oriented Software Architecture-A System* by Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal . Currently, patterns are being used largely for software architecture and design and, more recently, for organizations, specification models, and many other aspects of software development processes.

The main idea behind using patterns is to provide documentation to help categorize and communicate about solutions to recurring problems. The pattern has a name to facilitate discussion and the information it represents. A definition that more closely reflects its use within the patterns community is by Riehle and Ztillighoven:

A *pattern* is [an] instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces.

The documentation of a pattern, in essence, provides the contexts under which it is suitable and the constraints and forces that may affect a solution or its consequences. Communication about patterns is enabled by a vocabulary that describes the pattern and its related components such as name, context, motivation, and solution. By classifying these components and their nature (such as the structural or behavioral nature of the solution), we can categorize patterns. A pattern involves a general description of a solution to a recurring problem bundle with various goals and constraints. But a pattern does more than just identify a solution, it also explains why the solution is needed. For better or for worse, however, the meteoric rise in popularity of software patterns frequently has caused them to be overhyped. Patterns have achieved buzzword status: It is immensely popular to use the word *pattern* to gamer an audience. However, not every solution, algorithm, best practice, maxim, or heuristic constitutes a pattern (one or more key pattern ingredients may be absent). Even if something appears to have all the requisite pattern components, it should not be considered a pattern until it has been verified to be a recurring phenomenon (preferably found in at least three existing systems; this often is called the *rule of three*). A "pattern in waiting," which is not yet known to recur, sometimes is called a *proto-pattern*. Many also feel it is inappropriate to decisively call something a *pattern* until it has undergone some degree of peer scrutiny or review. Coplien explains that a good pattern will do the following:

- . *It solves a problem.* Patterns capture solutions, not just abstract principles or strategies.
- . *It is a proven concept.* Patterns capture solutions with a track record, not theories or speculation.
- . *The solution is not obvious.* The best patterns generate a solution to a problem indirectly-a necessary approach for the most difficult problems of design.
- . *It describes a relationship.* Patterns do not just describe modules, but describe deeper system structures and mechanisms.
- . *The pattern has a significant human component.* All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

The majority of the initial patterns developed focus on design problems and still design patterns represent most solutions. However, more recent patterns encompass all aspects of software engineering, including development organization, the software development process, project planning, requirements engineering, and software configuration management.

Generative and No generative Patterns

Generative patterns are patterns that not only describe a recurring problem, they can tell us how to generate something and can be observed in the resulting system architectures they helped shape. Nongenerative patterns are static and passive: They describe recurring phenomena without necessarily saying how to reproduce them. We should strive to document generative patterns because they not only show us the

characteristics of good systems, they teach us how to build them. Alexander explains that the most useful patterns are generative:

These patterns in our minds are, more or less, mental images of the patterns in the world: they are abstract representations of the very morphological rules which define the patterns in the world. However, in one respect they are very different. The patterns in the world merely exist. But the same patterns in our minds are dynamic. They have force. They are generative. They tell us what to do; they tell us how we shall, or may, generate them; and they tell us too, that under certain circumstances, we must create them. Each pattern is a rule which describes what you have to do to generate the entity which it defines.

Alexander wants patterns, and especially pattern languages, to be capable of generating whole, living structures. Part of the desire to create architectures that emulate life lies in the unique ability of living things to evolve and adapt to their ever-changing environments (not only for the sake of individual survival but also for survival of the species). Alexander wants to impart these same qualities into his architecture. Similarly, in software, good software architecture is all about being adaptable and resilient to change. So another aspect of generativity is about striving to create "living" architecture capable of dynamically adapting to fulfill changing needs and demands. The successive application of several patterns, each encapsulating its own problem and forces, unfolds a larger solution, which emerges indirectly as a result of the smaller solutions. It is the generation of such emergent behavior that appears to be what is meant by *generativity*. In this fashion, a pattern language should guide its users to generate whole architectures that possess the quality. This particular aspect of Alexander's paradigm seems a bit too mystical for some people's tastes .

Patterns Template

Every pattern must be expressed "in the form of a rule [template] which establishes a relationship between a context, a system of forces which arises in that context, and a configuration, which allows these forces to resolve themselves in that context" .

Currently, several different pattern templates have been defined that eventually will represent a pattern. Despite this, it is generally agreed that a pattern should contain certain essential components. The following essential components should be clearly recognizable on reading a pattern :

Name. A meaningful name. This allows us to use a single word or short phrase to refer to the pattern and the knowledge and structure it describes. Good pattern names form a vocabulary for discussing conceptual abstractions. Sometimes, a pattern may have more than one commonly used or recognizable name in the literature. In this case, it is common practice to document these nicknames or synonyms under the heading of aliases or also known as. Some pattern forms also provide a classification of the pattern in addition to its name.

.Problem. A statement of the problem that describes its intent: the goals and objectives it wants to reach within the given context and forces. Often the forces oppose these objectives as well as each other.

.Context. The *preconditions* under which the problem and its solution seem to recur and for which the solution is desirable. This tells us the pattern's applicability. It can be thought of as the initial configuration of the system before the pattern is applied to it.

.Forces. A description of the relevant *forces* and constraints and how they interact or conflict with one another and with the goals we wish to achieve (perhaps with some indication of their priorities). A concrete scenario that serves as the *motivation* for the pattern frequently is employed (see also Examples). Forces reveal the intricacies of a problem and define the kinds of *trade-offs* that must be considered in the presence of the tension or dissonance they create. A good pattern description should fully encapsulate all the forces that have an impact on it.

.Solution. Static relationships and dynamic rules describing how to realize the desired outcome. This often is equivalent to giving instructions that describe how to construct the necessary products. The description may encompass pictures, diagrams, and prose that identify the pattern's structure, its participants, and their collaborations, to show how the problem is solved. The solution should describe not only the *static structure* but also *dynamic behavior*. The static structure tells us the form and organization of the pattern, but often the behavioral dynamics is what makes the pattern "come alive." The description of the pattern's solution may indicate guidelines to keep in mind (as well as pitfalls to avoid) when attempting a concrete implementation of the solution. Sometimes, possible variants or specializations of the solution are described as well.

.Examples. One or more sample applications of the pattern that illustrate a specific initial context; how the pattern is applied to and transforms that context; and the resulting context left in its wake. Examples help the reader understand the pattern's use and applicability. Visual examples and analogies often can be very useful. An example may be supplemented by a *sample implementation* to show one way the solution might be realized. Easy-to-comprehend examples from known systems usually are preferred.

.Resulting context. The state or configuration of the system after the pattern has been applied, including the consequences (both good and bad) of applying the pattern, and other problems and patterns that may arise from the new context. It describes the *postconditions* and *side effects* of the pattern. This is sometimes called a *resolution of forces* because it describes which forces have been resolved, which ones remain unresolved, and which patterns may now be applicable. Documenting the resulting context produced by one pattern helps you correlate it with the initial context of other patterns (a single pattern often is just one step toward accomplishing some larger task or project).

.Rationale. A justifying explanation of steps or rules in the pattern and also of the pattern as a whole in terms of how and why it resolves its forces in a particular way to be in alignment with desired goals, principles, and philosophies. It explains how the forces and constraints are orchestrated in concert to achieve a resonant harmony. This tells us how the pattern actually works, why it works, and why it is "good." The solution component of a pattern may describe the outwardly visible structure and behavior of the pattern, but the rationale is what provides insight into the *deep structures* and *key mechanisms* going on beneath the surface of the system.

Related patterns. The static and dynamic relationships between this pattern and others within the same pattern language or system. Related patterns often share common forces. They also frequently have an initial or resulting context that is compatible with the resulting or initial context of another pattern. Such patterns might be predecessor patterns whose application leads to this pattern, successor patterns whose application follows from this pattern, alternative patterns that describe a different solution to the same problem but under different forces and constraints, and codependent patterns that may (or must) be applied simultaneously with this pattern.

Known uses. The known occurrences of the pattern and its application within existing systems. This helps validate a pattern by verifying that it indeed is a *proven solution* to a *recurring problem*. Known uses of the pattern often can serve as instructional examples (see also Examples).

Although it is not strictly required, good patterns often begin with an abstract that provides a short summary or overview. This gives readers a clear picture of the pattern and quickly informs them of its relevance to any problems they may wish to solve (sometimes such a description is called a *thumbnail sketch* of the pattern, or a *pattern thumbnail*). A pattern should identify its target audience and make clear what it assumes of the reader.

Antipatterns

A pattern represents a "best practice," whereas an antipattern represents "worst practice" or a "lesson learned." Anti patterns come in two varieties:

.Those describing a bad solution to a problem that resulted in a bad situation.
.Those describing how to get out of a bad situation and how to proceed from there to a good solution. Anti patterns are valuable because often it is just as important to see and understand bad solutions as to see and understand good ones.

Coplien explains that The study of anti-patterns is an important research activity. The presence of "good" patterns in a successful system is not enough; you also must show that those patterns are absent in unsuccessful systems. Likewise, it is useful to show the presence of certain patterns (anti-patterns) in unsuccessful systems, and their absence in successful systems.

Capturing Patterns

Writing good patterns is *very difficult*, explains Appleton . Patterns should provide not only facts (like a reference manual or users' guide) but also tell a story that captures the experience they are trying to convey. A pattern should help its users comprehend existing systems, customize systems to fit user needs, and construct new systems. The process of looking for patterns to document is called *pattern mining* (or sometimes *reverse architecting*). An interesting initiative started within the software community is to share experience with patterns and develop an ever-growing repository of patterns.

People can contribute new solutions, lessons learned (or antipatterns), and more examples within a variety of contexts.

How do you know a pattern when you come across one? The answer is you do not always know. You may jot down the beginning of some things you think are patterns, but it may turn out that these are not patterns at all, or they are only pieces of patterns, simply good principles, or general rules that may form part of the rationale for a particular pattern. It is important to remember that a solution in which no forces are present is *not* a pattern. These guidelines are summarized from Buschmann

- . *Focus on practicability.* Patterns should describe *proven solutions* to recurring problems rather than the latest scientific results.
- .*Aggressive disregard of originality.* Pattern writers do *not* need to be the original inventor or discoverer of the solutions that they document.
- .*Nonanonymous review.* Pattern submissions are *shepherded* rather than reviewed. The shepherd contacts the pattern author(s) and discusses with him or her how the patterns might be clarified or improved on.
- .*Writers' workshops instead of presentations.* Rather than being presented by the individual authors, the patterns are discussed in writers' workshops, open forums where all attending seek to improve the patterns presented by discussing what they like about them and the areas in which they are lacking.
- .*Careful editing.* The pattern authors should have the opportunity to incorporate all the comments and insights during the shepherding and writers' workshops before presenting the patterns in their finished form.

7.10 LET US SUM UP

Each method has a weakness, too. While Rumbaugh et al.'s OMT has strong methods for modeling the problem domain, OMT models cannot fully express the requirements. Jacobson et al. deemphasize object modeling and, although they cover a fairly wide range of the life cycle, they do not treat object-oriented design to the same level as Booch, who focuses almost entirely on design, not analysis.

Booch and Rumbaugh et al. are object centered in their approaches and focus more on figuring out what are the objects of a system, how are they related, and how do they collaborate with each other. Jacobson et al. are more user centered, in that everything in their approach derives from use cases or usage scenarios.

7.11 POINTS FOR DISCUSSION

1. Analyse Macro Development model by Booch methodology
2. Analyse Micro Development model by Booch methodology
3. Critically examine object oriented business engineering

7.12 LESSON – END ACTIVITIES

1. Discuss about object-oriented software – engineering.
2. Discuss about patterns.

7.13 REFERENCES

1. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
2. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
3. Rumbaugh, James ,Michael –Object oriented Modelling and design
4. Ali –Brahmi –Object oriented system development”

LESSON – 8: FRAMEWORKS

CONTENTS

8.0 AIMS AND OBJECTIVES

8.1 INTRODUCTION.

8.2 FRAMEWORKS

8.4 OBJECT-ORIENTED ANALYSIS

8.5 OBJECT-ORIENTED DESIGN

8.6 ITERATIVE DEVELOPMENT AND CONTINUOUS TESTING

8.7 MODELING BASED ON THE UNIFIED MODELING LANGUAGE

8.8 LET US SUM UP

8.9 POINTS FOR DISCUSSION

8.10 LESSON – END ACTIVITIES

8.11 REFERENCES

8.0 AIM AND OBJECTIVE

You should be able to develop application using patterns.

8.1 INTRODUCTION

Frameworks are a way of delivering application development patterns to support best practice sharing during application development-not just within one company, but across many companies-through an emerging framework market. This is not an entirely new idea.

8.2 FRAMEWORKS

An experienced programmer almost never codes a new program from scratch she'll use macros, copy libraries, and template like code fragments from earlier programs to make a start on a new one. Work on the new program begins by filling in new domain-specific code inside the older structures.

A seasoned business consultant who has worked on many consulting projects performing data modeling almost never builds a new data model from scratch he'll have a selection of model fragments that have been developed over time to help new modeling projects hit the ground running. New domain-specific terms will be substituted for those in his library models.

A *framework* is a way of presenting a generic solution to a problem that can be applied to all levels in a development . However, design and software frameworks are the most popular. A definition of an object-oriented software framework is given by Gamma:

A framework is a set of cooperating classes that make up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes a framework to a particular application by subclassing and composing instances of framework classes. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses you can put to work immediately. A single framework typically encompasses several design patterns. In fact, a framework can be viewed as the implementation of a system of design patterns.

Even though they are related in this manner, it is important to recognize that frameworks and design patterns are two distinctly separate beasts: A framework is executable software, whereas design patterns represent knowledge and experience about software. In this respect, frameworks are of a physical nature, while patterns are of a logical nature: Frameworks are the physical realization of one or more software pattern solutions; patterns are the instructions for how to implement those solutions .

Gamma et al. describe the major differences between design patterns and frameworks as follows:

Design patterns are more abstract than frameworks. Frameworks can be embodied in code, but only examples of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. In contrast, design patterns have to be implemented each time they are used. Design patterns also explain the intent, trade-offs, and consequences of a design.

Design patterns are smaller architectural elements than frameworks. A typical framework contains several design patterns but the reverse is never true.

. Design patterns are less specialized than frameworks. Frameworks always have a particular application domain. In contrast, design patterns can be used in nearly any kind of application. While more specialized design patterns are certainly possible, even these would not dictate an application architecture.

8.3 THE UNIFIED APPROACH

The approach promoted in this book is based on the best practices that have proven successful in system development and, more specifically, the work done by Booch, Rumbaugh, and Jacobson in their attempt to unify their modeling efforts. The unified approach (UA) (see Fig) establishes a unifying and unitary framework around their works by utilizing the unified modeling language (UML) to describe, model, and document the software development process. The idea behind the UA is not to introduce yet another methodology. The main motivation here is to combine the best practices,

processes, methodologies, and *guidelines* along with UML notations and diagrams for better understanding object-oriented concepts and system development.

The unified approach to software development revolves around (but is not limited to) the following processes and concepts (see Fig). The processes are:

- Use-case driven development
- Object-oriented analysis
- Object-oriented design
- Incremental development and prototyping
- Continuous testing

8.4 OBJECT-ORIENTED ANALYSIS

Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirements. The goal of object-oriented analysis is to first understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system. This is accomplished by constructing several models of the system. These models concentrate on describing what the system does rather than how it does it. Separating the behavior of a system from the way it is implemented requires viewing the system from the user's perspective rather than that of the machine. OOA Process consists of the following

Steps:

1. Identify the Actors.
2. Develop a simple business process model using UML Activity diagram.
3. Develop the Use Case.
4. Develop interaction diagrams.
5. Identify classes.

8.5 OBJECT-ORIENTED DESIGN

Bloch provides the most comprehensive object-oriented design method. Ironically, since it is so comprehensive, the method can be somewhat imposing to learn and especially tricky to figure out where to start. Rumbaugh et al.'s and Jacobson et al.'s high-level models provide good avenues for getting started. UA combines these by utilizing Jacobson et al.'s analysis and interaction diagrams, Bloch's object diagrams, and Rumbaugh et al.'s domain models. Furthermore, by following Jacobson et al.'s life cycle model, we can produce designs that are traceable across requirements, analysis, design, coding, and testing. OOD Process consists of:

- . Designing classes, their attributes, methods, associations, structures and protocols, apply design axioms
- . Design the Access Layer
- . Design and prototype User interface
- . User Satisfaction and Usability Tests based on the Usage/Use Cases
- . Iterate and refine the design

8.6 ITERATIVE DEVELOPMENT AND CONTINUOUS TESTING

You must iterate and reiterate until, eventually, you are satisfied with the system. Since testing often uncovers design weaknesses or at least provides additional information you will want to use, repeat the entire process, taking what you have learned and reworking your design or moving on to reprototyping and retesting. Continue this refining cycle through the development process until you are satisfied with the results. During this iterative process, your prototypes will be incrementally transformed into the actual application. The UA encourages the integration of testing plans from day I of the project. Usage scenarios can become test scenarios; therefore, use cases will drive the usability testing. Usability testing is the process in which the functionality of software is measured.

8.7 MODELING BASED ON THE UNIFIED MODELING LANGUAGE

The unified modeling language was developed by the joint efforts of the leading object technologists Grady Booch, Ivar Jacobson, and James Rumbaugh with contributions from many others. The UML merges the best of the notations used by the three most popular analysis and design methodologies: Booch's methodology, Jacobson et al.'s use case, and Rumbaugh et al.'s object modeling technique. The UML is becoming the universal language for modeling systems; it is intended to be used to express models of many different kinds and purposes, just as a programming language or a natural language can be used in many different ways. The UML has become the standard notation for object-oriented modeling systems. It is an evolving notation that still is under development. The UA uses the UML to describe and model the analysis and design phases of system development (UML notations will be covered in Chapter 5).

The UA Proposed Repository

In modern businesses, best practice sharing is a way to ensure that solutions to process and organization problems in one part of the business are communicated to other parts where similar problems occur. Best practice sharing eliminates duplication of problem solving. For many companies, best practice sharing is institutionalized as part of their constant goal of quality improvement. Best practice sharing must be applied to application development if quality and productivity are to be added to component reuse benefits. Such sharing extends the idea of software reusability to include all phases of software development such as analysis, design, and testing .

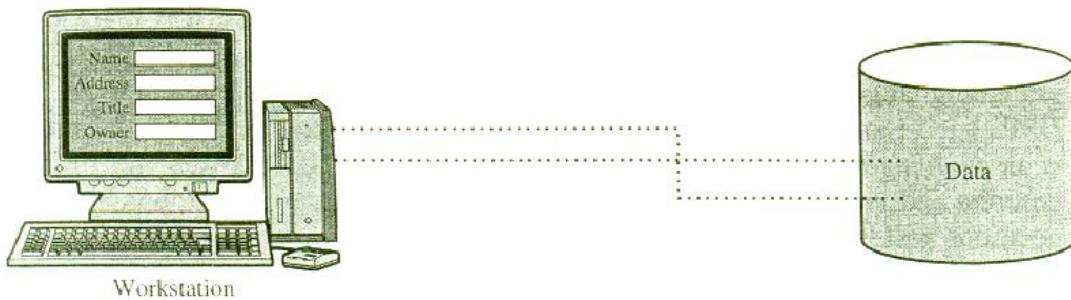
The idea promoted here is to create a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks, and user interfaces in an easily accessible manner with a completely available and easily utilized format. As we saw previously, central to the discussion on developing this best practice sharing is the concept of a pattern. Everything from the original user request to maintenance of the project as it goes to production should be kept in the repository. The advantage of repositories is that, if your organization has done projects in the past, objects in the repositories from those projects might be useful. You can select any piece

from a repository-from the definition of one data element, to a diagram, all its symbols, and all their dependent definitions, to entries- for reuse.

The UA's underlying assumption is that, if we design and develop applications based on previous experience, creating additional applications will require no more than assembling components from the library. Additionally, applying lessons learned from past developmental mistakes to future projects will increase the quality of the product and reduce the cost and development time. Some basic capability is available in most object-oriented environments, such as Microsoft repository, VisualAge, PowerBuilder, Visual C++ , and Delphi. These repositories contain all objects that have been previously defined and can be reused for putting together a new software system for a new application.

If a new requirement surfaces, new objects will be designed and stored in the main repository for future use. The same arguments can be made about patterns and frameworks. Specifications of the software components, describing the behavior of the component and how it should be used, are registered in the repository for future reuse by teams of developers.

The repository should be accessible to many people. Furthermore, it should be relatively easy to search the repository for classes based on their attributes, methods,



Two-layered architecture: interface and data.

or other characteristics. For example, application developers could select prebuilt components from the central component repository that match their business needs and assemble these components into a single application, customizing where needed. Tools to fully support a comprehensive repository are not accessible yet, but this will change quickly and, in the near future, we will see more readily available tools to capture all phases of software development into a repository for use and reuse.

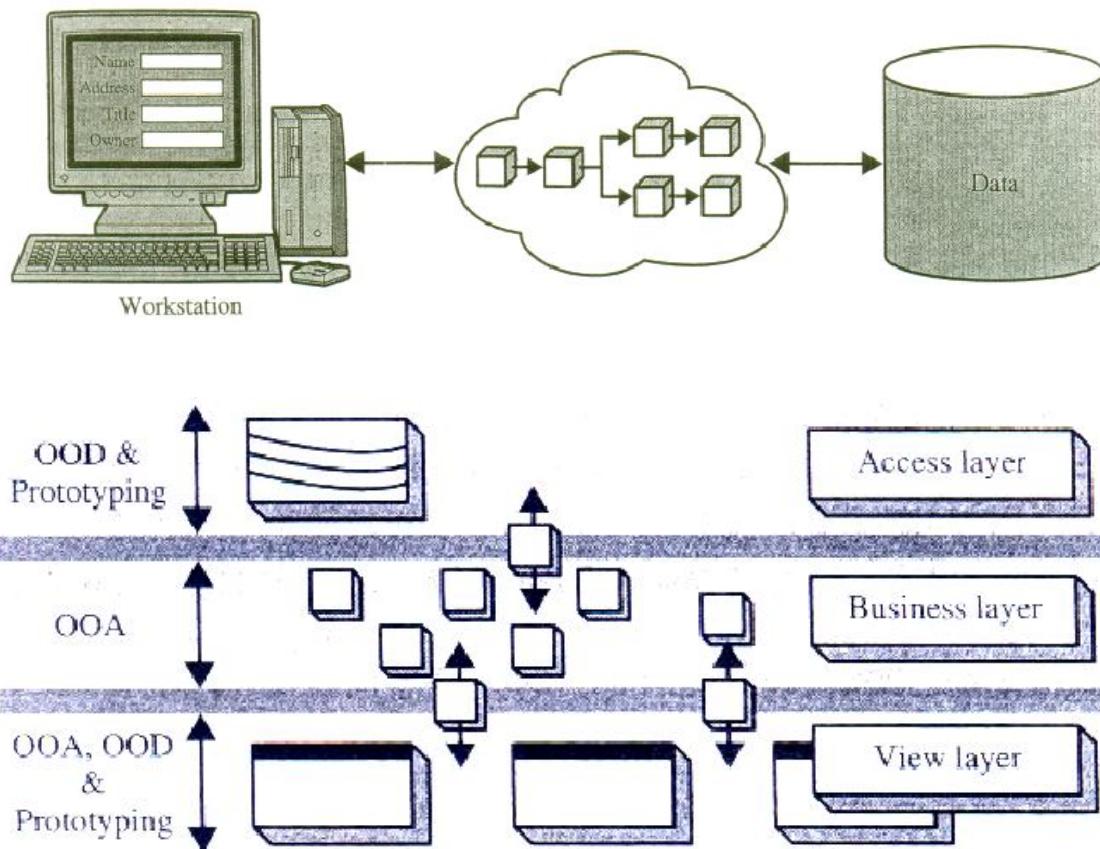
The Layered Approach to Software Development

Most systems developed with today's CASE tools or client-server application development environments tend to lean toward what is known as *two-layered architecture*: interface and data (see Fig).

In a two-layered system, user interface screens are tied to the data through routines that sit directly behind the screens; for example, a routine that executes when you click on a button. With every interface you create, you must re-create the business logic needed to run the screen. The routines required to access the data must exist within every screen. Any change to the business logic must be accomplished in every screen that deals with that portion of the business. This approach results in objects that are very specialized and cannot be reused easily in other projects.

A better approach to systems architecture is one that isolates the functions of the interface from the functions of the business. This approach also isolates the business from the details of the data access (see Fig.).

Objects are completely independent of how they are represented or stored.



Business objects represent tangible elements of the application. They should be completely independent of how they are represented to the user or how they are physically stored. In a three-layered approach, you are able to create objects that represent tangible elements of your business yet are completely independent of how they are represented to the user (through an interface) or how they are physically stored (in a database). The three-layered approach consists of a view or user interface layer, a business layer, and an access layer (see Fig.).

The Business Layer The business layer contains all the objects that represent the business (both data and behavior). This is where the real objects such as Order, Customer, Line item, Inventory, and Invoice exist. Most modern object-oriented analysis and design methodologies are generated toward identifying these kinds of objects.

The responsibilities of the business layer are very straightforward: Model the objects of the business and how they interact to accomplish the business processes. When creating the business layer, however, it is important to keep in mind a couple of things.

These objects should not be responsible for the following: *.Displaying details.* Business objects should have no special knowledge of how they are being displayed and by whom. They are designed to be independent of any particular interface, so the details of how to display an object should exist in the interface (view) layer of the object displaying it.

Data access details. Business objects also should have no special knowledge of "where they come from." It does not matter to the business model whether the data are stored and retrieved via SQL or file I/O. The business objects need to know only to whom to talk about being stored or retrieved. The business objects are modeled during the object-oriented analysis. A business model captures the static and dynamic relationships among a collection of business objects. Static relationships include object associations and aggregations. For example, a customer could have more than one account or an order could be aggregated from one or more line items. Dynamic relationships show how the business objects interact to perform tasks. For example, an order interacts with inventory to determine product availability. An individual business object can appear in different business models. Business models also incorporate control objects that direct their processes. The business objects are identified during the object-oriented analysis. Use cases can provide a wonderful tool to capture business objects.

The User Interface (View) Layer: The user interface layer consists of objects with which the user interacts as well as the objects needed to manage or control the interface. The user interface layer also is called the *view layer*. This layer typically is responsible for two major aspects of the applications:

. *Responding to user interaction.* The user interface layer objects must be designed to translate actions by the user, such as clicking on a button or selecting .from a menu, into an appropriate response. That response may be to open or close another interface or to send a message down into the business layer to start some business process; remember, the business logic does not exist here, just the knowledge of which message to send to which business object..

. *Displaying business objects.* This layer must paint the best possible picture of the business objects for the user. In one interface, this may mean entry fields and list boxes to display an order and its items. In another, it may be a graph of the total price of a customer's orders.

8.8 LET US SUM UP

The main idea behind a pattern is the documentation to help categorize, communicate about, and locate solutions to recurring problems. Frameworks are away of delivering application development patterns to support best practice sharing during application development. A single framework typically encompasses several design patterns. In fact, a framework can be viewed as the implementation of a system of design patterns. Writing good patterns is very difficult, since it should not only provide facts but also tell a story that captures the experience the pattern is trying to convey.

8.9 POINTS FOR DISCUSSION

1. Analyze object oriented analysis.
2. Evaluate object oriented design
3. Justify antipatterns

8.10 LESSON – END ACTIVITIES

1. Discuss about Frameworks.
2. Establish UA proposed Respiratory

8.11 REFERENCES

1. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
2. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
3. Rumbaugh, James ,Michael –Object oriented Modelling and design
4. Ali –Brahmi –Object oriented system development”

LESSON – 9: STATIC AND DYNAMIC MODELING

CONTENTS

- 9.0 AIMS AND OBJECTIVES
- 9.1 INTRODUCTION
- 9.2 STATIC AND DYNAMIC MODELS
- 9.3 WHY MODELING?
- 9.4 INTRODUCTION TO THE UNIFIED MODELING LANGUAGE
- 9.5 UML DIAGRAMS
- 9.6 UML CLASS DIAGRAM
- 9.7 LET US SUM UP
- 9.8 POINTS FOR DISCUSSION
- 9.9 LESSON – END ACTIVITIES
- 9.10 REFERENCES

9.0 AIM AND OBJECTIVE

Basics of Unified Modeling Language (UML) and its modeling diagrams.
 UML Class diagram.
 UML Use case diagram.
 UML Sequence diagram.
 UML Collaboration diagram.
 UML Statechart diagram.
 UML Activity diagram.
 UML Component diagram.
 UML Deployment diagram.

9.1 INTRODUCTION

A *model* is an abstract representation of a system, constructed to understand the system prior to building or modifying it. The term *system* is used here in a broad sense to include any process or structure. For Example, the organizational structure of a corporation, health services, computer software, instruction of any sort (including computers), the national economy, and so forth all would be termed *systems*. Efraim Turban describes a model as a simplified representation of reality. A model is simplified because reality is too complex or large and much of the complexity actually is irrelevant to the problem we are trying to describe or solve. A model provides a means for conceptualization and communication of ideas in a precise and unambiguous form. The

characteristics of simplification and representation are difficult to achieve in the real world, since they frequently contradict each other. Thus, modeling enables us to cope with the complexity of a system. Most modeling techniques used for analysis and design involve graphic languages.

These graphic languages are sets of symbols. The symbols are used according to certain rules of the methodology for communicating the complex relationships of information more clearly than descriptive text. The main goal of most CASE tools is to aid us in using these graphic languages, along with their associated methodologies.

Modeling frequently is used during many of the phases of the software life cycle, such as analysis, design, and implementation. For example, Objectory is built around several different models: . *Use-case model*. The use-case model defines the outside (actors) and inside (use case) of the system's behavior.

- . *Domain object model*. Objects of the "real" world are mapped into the domain object model.
- .*Analysis object model*. The analysis object model presents how the source code (i.e., the implementation) should be carried out and written.
- .*Implementation model*. The implementation model represents the implementation of the system.
- .*Test model*. The test model constitutes the test plans, specifications, and reports.

Modeling, like any other object-oriented development, is an iterative process.

As the model progresses from analysis to implementation, more detail is added, but it remains essentially the same. In this chapter, we look at unified modeling language (UML) notations and diagrams. The main idea here is to gain exposure to the UML syntax, semantics, and modeling constructs. Many new concepts will be introduced here from a modeling standpoint. We apply these concepts in system analysis and design contexts in later chapters.

9.2 STATIC AND DYNAMIC MODELS

Models can represent static or dynamic situations. Each representation has different implications for how the knowledge about the model might be organized and represented

Static Model

A *static model* can be viewed as a snapshot of a system's parameters at rest or at a specific point in time. Static models are needed to represent the structural or static aspect of a system. For example, a customer could have more than one account or an order could be aggregated from one or more line items. Static models assume stability and an absence of change in data over time. The unified modeling language class diagram is an example of a static model.

Dynamic Model

A *dynamic model*, in contrast to a static model, can be viewed as a collection of procedures or behaviors that, taken together, reflect the behavior of a system over time. Dynamic relationships show how the business objects interact to perform tasks. For example, an order interacts with inventory to determine product availability.

A system can be described by first developing its static model, which is the structure of its objects and their relationships to each other frozen in time, a baseline. Then, we can examine changes to the objects and their relationships over time. Dynamic modeling is most useful during the design and implementation phases of the system development. The UML interaction diagrams and activity models are examples of UML dynamic models.

9.3 WHY MODELING?

Building a model for a software system prior to its construction is as essential as having a blueprint for building a large building. Good models are essential for communication among project teams. As the complexity of systems increases, so does the importance of good modeling techniques. Many other factors add to a project's success, but having a rigorous modeling language is essential. A modeling language must include

- Model elements-fundamental modeling concepts and semantics.
- Notation-visual rendering of model elements.
- Guidelines-expression of usage within the trade.

In the face of increasingly complex systems, visualization and modeling become essential, since we cannot comprehend any such system in its entirety. The use of visual notation to represent or model a problem can provide us several benefits relating to clarity, familiarity, maintenance, and simplification.

- *Clarity.* We are much better at picking out errors and omissions from a graphical or visual representation than from listings of code or tables of numbers. We very easily can understand the system being modeled because visual examination of the whole is possible.
- *Familiarity.* The representation form for the model may turn out to be similar to the way in which the information actually is represented and used by the employees currently working in the problem domain. We, too, may find it more comfortable to work with this type of representation.
- *Maintenance.* Visual notation can improve the maintainability of a system. The visual identification of locations to be changed and the visual confirmation of those changes will reduce errors. Thus, you can make changes faster, and fewer errors are likely to be introduced in the process of making those changes.
- *Simplification.* Use of a higher level representation generally results in the use of fewer but more general constructs, contributing to simplicity and conceptual understanding. Turban cites the following advantages of modeling:

1. Models make it easier to express complex ideas. For example, an architect builds a model to communicate ideas more easily to clients.
2. The main reason for modeling is the reduction of complexity. Models reduce complexity by separating those aspects that are unimportant from those that are important. Therefore, it makes complex situations easier to understand.
3. Models enhance and reinforce learning and training.
4. The cost of the modeling analysis is much lower than the cost of similar experimentation conducted with a real system.
5. Manipulation of the model (changing variables) is much easier than manipulating a real system.

To summarize, here are a few key ideas regarding modeling: . A model is rarely correct on the first try. . Always seek the advice and criticism of others. You can improve a model by reconciling different perspectives. .Avoid excess model revisions, as they can distort the essence of your model. Let simplicity and elegance guide you through the process.

9.4 INTRODUCTION TO THE UNIFIED MODELING LANGUAGE

The unified modeling language is a language for specifying, constructing, visualizing, and documenting the software system and its components. The UML is a graphical language with sets of rules and semantics. The rules and semantics of a model are expressed in English, in a form known as *object constraint language* (OCL). OCL is a specification language that uses simple logic for specifying the properties of a system. The UML is not intended to be a visual programming language in the sense of having all the necessary visual and semantic support to replace programming languages. However, the UML does have a tight mapping to a family of object-oriented languages, so that you can get the best of both worlds.

The goals of the unification efforts were to keep it simple; to cast away elements of existing Booch, OMT, and OOSE methods that did not work in practice; to add elements from other methods that were more effective; and to invent new methods only when an existing solution was unavailable. Because the UML authors, in effect, were designing a language (albeit a graphical one), they had to strike a proper balance between minimalism (everything is text and boxes) and overengineering (having a symbol or fig for every conceivable modeling element).

To that end, they were very careful about adding new things: They did not want to make the UML unnecessarily complex. A similar situation exists with the problem of UML not supporting other diagrams. Booch et al. explain that other diagrams, such as the data flow diagram (DFD), were not included in the UML because they do not fit as cleanly into a consistent object-oriented paradigm. For example, activity diagrams accomplish much of what people want from DFDs and then some; activity diagrams also are useful for modeling work flow. The authors of the UML clearly are promoting the UML diagrams over all others for object-oriented projects but do not condemn all other diagrams. Along the way, however, some things were found that were advantageous to

add because they had proven useful in other modeling practice. The primary goals in the design of the UML were as follows :

1. Provide users a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts.
7. Integrate best practices and methodologies.

This section of the chapter is based on the *The Unified Modeling Language Notation Guide Version 1.1* written by Grady Booch, Ivar Jacobson, and James Rumbaugh .

9.5 UML DIAGRAMS

Every complex system is best approached through a small set of nearly independent views of a model; no single view is sufficient. Every model may be expressed at different levels of fidelity. The best models are connected to reality. The UML defines nine graphical diagrams:

1. Class diagram (static)
2. Use-case diagram
3. Behavior diagram (dynamic):
 - 3.1. Interaction diagram:
 - 3.1.1. Sequence diagram
 - 3.1.2. Collaboration diagram
 - 3.2. Statechart diagram
 - 3.3. Activity diagram
4. Implementation diagram:
 - 4.1. Component diagram
 - 4.2. Deployment diagram

The choice of what models and diagrams one creates has a great influence on how a problem is encountered and how a corresponding solution is shaped. We will study applications of different diagrams throughout the book. However, in this chapter we concentrate on the UML notations and its semantics.

9.6 UML CLASS DIAGRAM

The UML *class diagram*, also referred to as *object modeling*, is the main static analysis diagram. These diagrams show the static structure of the model. A class diagram is a collection of static modeling elements, such as classes and their relationships, connected as a graph to each other and to their contents; for example, the things that exist (such as classes), their internal structures, and their relationships to other classes. Class diagrams do not show temporal information, which is required in dynamic modeling.

Object modeling is the process by which the logical objects in the real world (problem space) are represented (mapped) by the actual objects in the program(logical or a mini

world). This visual representation of the objects, their relationships, and their structures is for ease of understanding. To effectively develop a model of the real world and to determine the objects required in the system, you first must ask what objects are needed to model the system. Answering the following questions will help you to stay focused on the problem at hand and determine what is inside the problem domain and what is outside it:

- .What are the goals of the system?
- .What must the system accomplish?

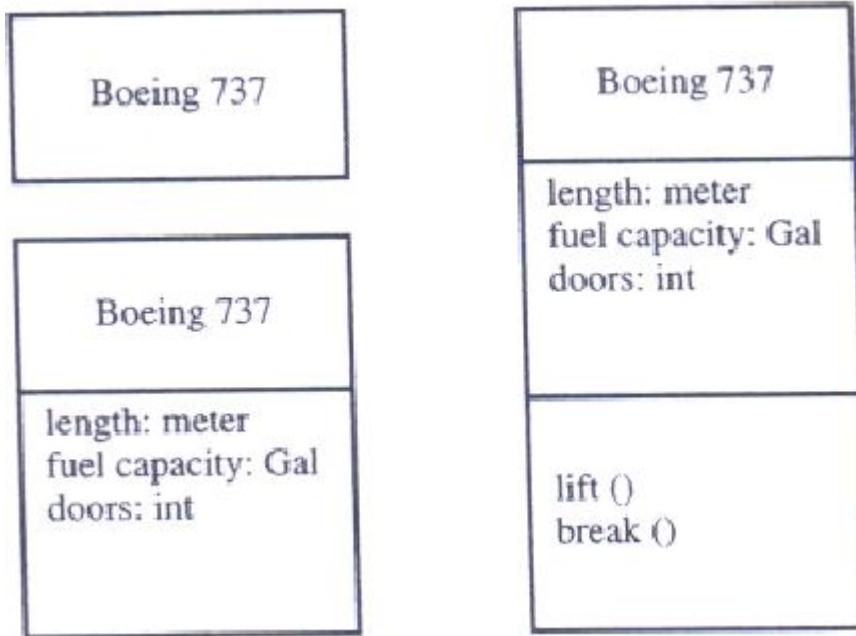
You need to know what objects will form the system because, in the object-oriented viewpoint, objects are the primary abstraction. The main task of object modeling is to graphically show what each object will do in the problem domain, describe the structure (such as class hierarchy or part-whole) and the relationships among objects (such as associations) by visual notation, and determine what behaviors fall within and outside the problem domain.

Class Notation: Static Structure

A class is drawn as a rectangle with three compartments separated by horizontal lines. The top name compartment holds the class name, other general properties of the class, such as attributes, are in the middle compartment, and the bottom compartment holds a list of operations (see Fig.). Either or both the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment if a compartment is suppressed; no inference can be drawn about the presence or absence of elements in it. The class name and other properties should be displayed in up to three sections. A stylistic convention of UML is to use an italic font for abstract classes and a normal (roman) font for concrete classes.

Object Diagram

A static object diagram is an instance of a class diagram. It shows a snapshot of the detailed state of the system at a point in time. Notation is the same for an object diagram and a class diagram. Class diagrams can contain objects, so a class diagram with objects and no classes is an object diagram.



In class notation, either or both the attributes and operation compartments may be suppressed.

Class Interface Notation

Class interface notation is used to describe the externally visible behavior of a class; for example, an operation with public visibility. Identifying class interfaces is a design activity of object-oriented system development. The UML notation for an interface is a small circle with the name of the interface connected to the class. A class that requires the operations in the interface may be attached to the circle by a dashed arrow. The dependent class is not required to actually use all of the operations. For example, a Person object may need to interact with the BankAccount object to get the Balance; this relationship is depicted in Fig with UML class interface notation.

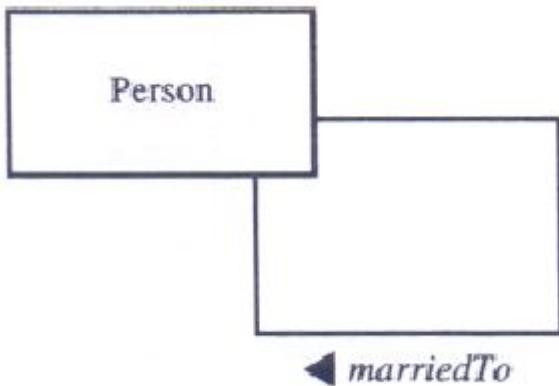
Binary Association Notation

A binary association is drawn as a solid path connecting two classes, or both ends may be connected to the same class. An association may have an association name. Furthermore, the association name may have an optional black triangle in it, the point of the triangle indicating the direction in which to read the name. The end of an association, where it connects to a class, is called the *association role* (seeFig).

Association Role

A simple association—the technical term for it is *binary association*—is drawn as a solid line connecting two class symbols. The end of an association, where it connects to a class, shows the association role. The role is part of the association, not

Interface notation of a class.



Association notation.

part of the class. Each association has two or more roles to which it is connected. In Fig 5-3, the association *worksFor* connects two roles, *employee* and *employer*. A Person is an employee of a Company and a Company is an employer of a Person.

The UML uses the term *association navigation* or *navigability* to specify a role affiliated with each end of an association relationship. An arrow may be attached to the end of the path to indicate that navigation is supported in the direction of the class pointed to. An arrow may be attached to neither, one, or both ends of the path. In particular, arrows could be shown whenever navigation is supported in a given direction. In the UML, association is represented by an open arrow, as represented in Fig . Navigability is visually distinguished from inheritance, which is denoted by an unfilled arrowhead symbol near the superclass.

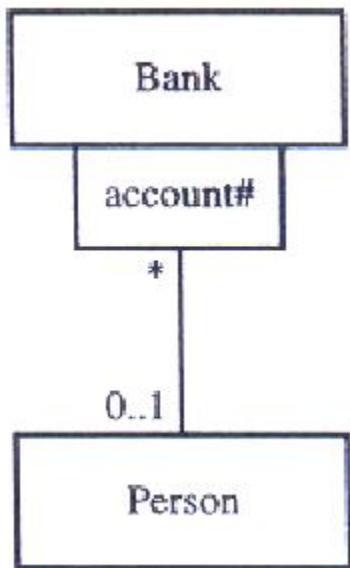
In Fig 5-4, the association is navigable in only one direction, from the BankAccount to Person, but not the reverse. This might indicate a design decision, but it

also might indicate an analysis decision, that the Person class is frozen and cannot be extended to know about the BankAccount class, but the BankAccount class can know about the Person class.

Qualifier

A *qualifier* is an association attribute. For example, a person object may be associated to a Bank object. An attribute of this association is the account#. The account# is the qualifier of this association

Association notation.



The fig depicts association qualifier and its multiplicity.

A qualifier is shown as a small rectangle attached to the end of an association path, between the final path segment and the symbol of the class to which it connects. The qualifier rectangle is part of the association path, not part of the class. The qualifier rectangle usually is smaller than the attached class rectangle (see Fig).

Multiplicity

Multiplicity specifies the range of allowable associated classes. It is given for roles within associations, parts within compositions, repetitions, and other purposes. A multiplicity specification is shown as a text string comprising a period-separated

sequence of integer intervals, where an interval represents a range of integers in this format (see Fig 5-5): lower bound.. upper bound.

The terms *lower bound* and *upper bound* are integer values, specifying the range of integers including the lower bound to the upper bound. The star character (*) may be used for the upper bound, denoting an unlimited upper bound. If a single integer value is specified, then the integer range contains the single values. For example, 0..1

0..*

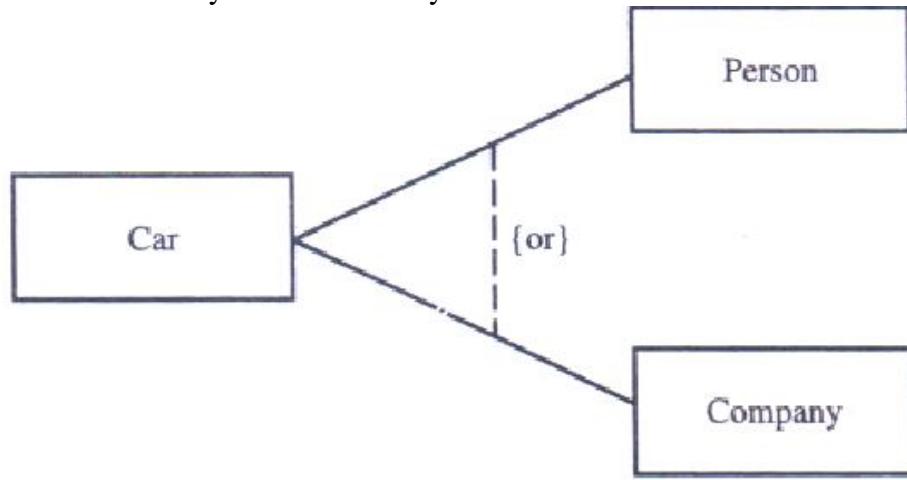
1..3, 7..10, 15, 19..*

OR Association

An *OR association* indicates a situation in which only one of several potential associations may be instantiated at one time for any single object. This is shown as a dashed line connecting two or more associations, all of which must have a class in common, with the constraint string {or} labeling the dashed line (see Fig). In other words, any instance of the class may participate in, at most, one of the associations at one time.

Association Class

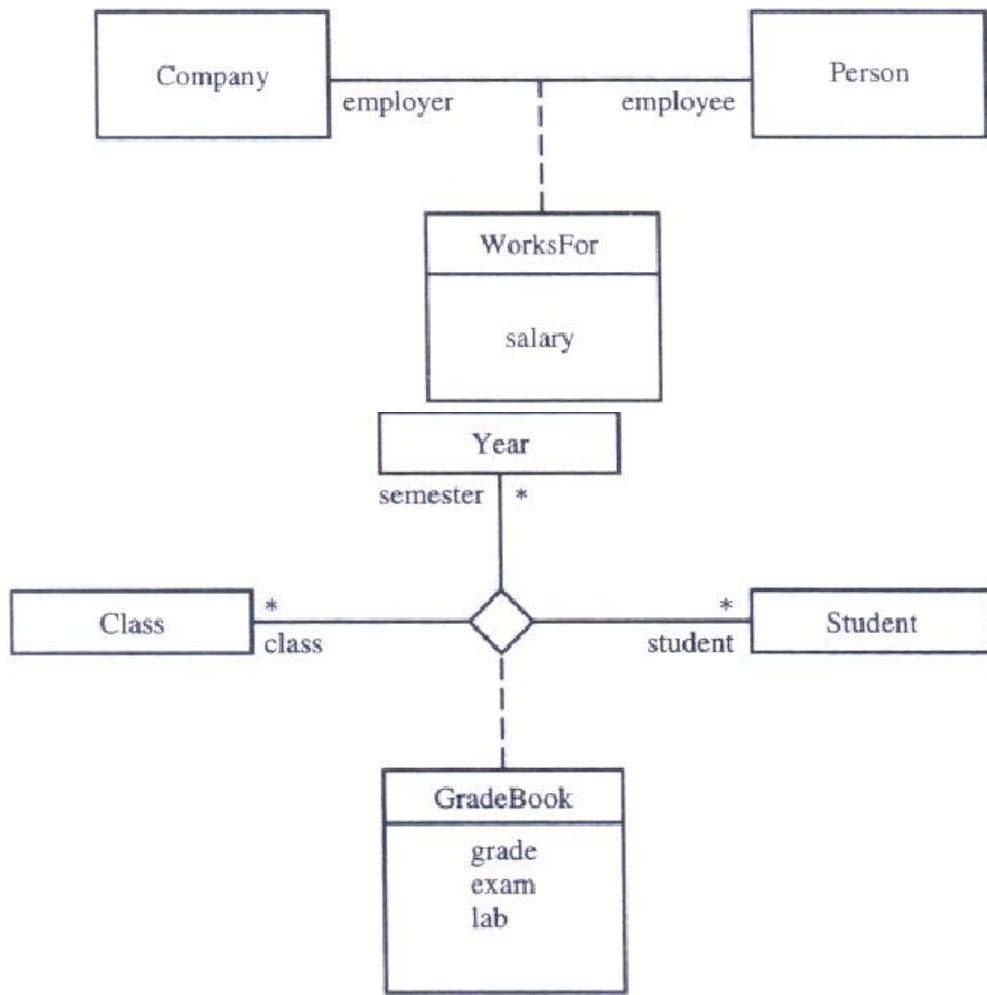
An *association class* is an association that also has class properties. An association class is shown as a class symbol attached by a dashed line to an association



An OR association notation. A car may associate with a person or a company. path. The name in the class symbol and the name string attached to the association path are the same (see Fig 5-7). The name can be shown on the path or the class symbol or both. If an association class has attributes but no operations or other associations, then the name may be displayed on the association path and omitted from the association class to emphasize its "association nature." If it has operations and attributes, then the name may be omitted from the path and placed in the class rectangle to emphasize its "class nature."

N-Ary Association

An *n-ary association* is an association among more than two classes. Since n-ary association is more difficult to understand, it is better to convert an n-ary association to binary association. However, here, for the sake of completeness, we cover the notation of n-ary association. An n-ary association is shown as a large diamond with a path from the diamond to each participant class. The name of the association (if any) is shown near the diamond. The role attachment may appear on each path as with a binary association. Multiplicity may be indicated; however, qualifiers and aggregation are not permitted. An association class symbol may be attached to the association class.



An n-ary (ternary) association that shows association among class, year, and student classes. The association class GradeBook which contains the attributes of the associations such as grade, exam, and lab. attached to the diamond by a dashed line, indicating an n-ary association that has attributes, operation, or associations. The example depicted in Fig shows the grade book of a class in each semester.

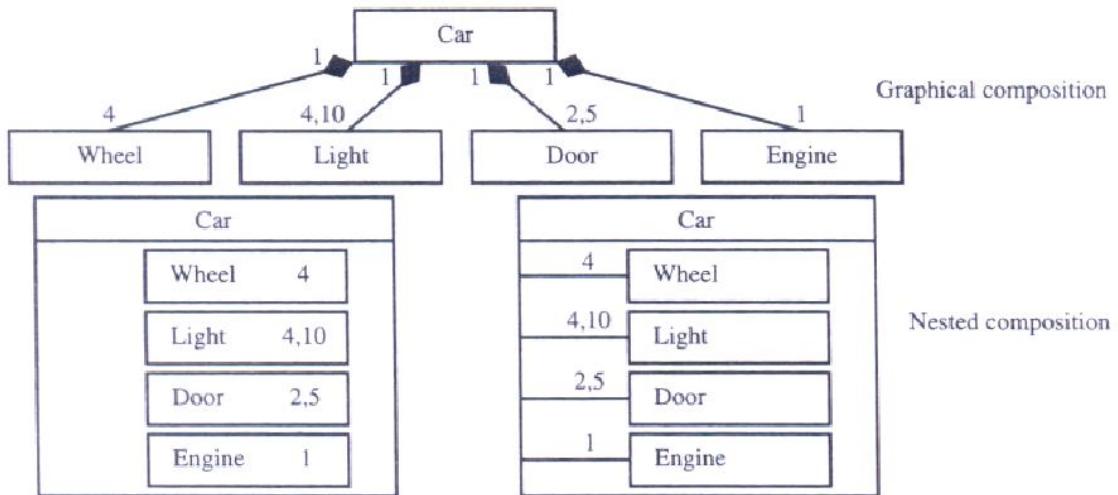
Aggregation and Composition (a.part.of)

Aggregation is a form of association. A hollow diamond is attached to the end of the path to indicate aggregation. However, the diamond may not be attached to both ends of a line, and it need not be presented at all (see Fig.). Composition, also known as the *a-part-of*, is a form of aggregation with strong ownership to represent the component of a complex object. Composition also is referred to as a *part-whole relationship*. The UML notation for composition is a solid diamond at the end of a path. Alternatively, the UML provides a graphically nested form that, in many cases, is more convenient for showing composition (see Fig.).

Parts with multiplicity greater than one may be created after the aggregate itself but, once created, they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.

Generalization

Generalization is the relationship between a more general class and a more specific class. Generalization is displayed as a directed line with a closed, hollow arrowhead

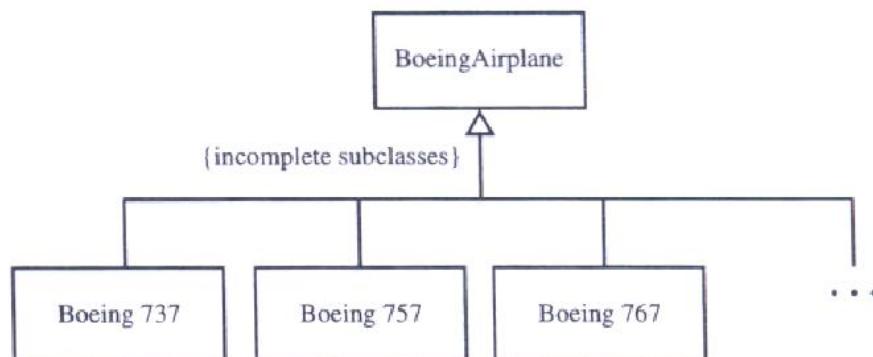
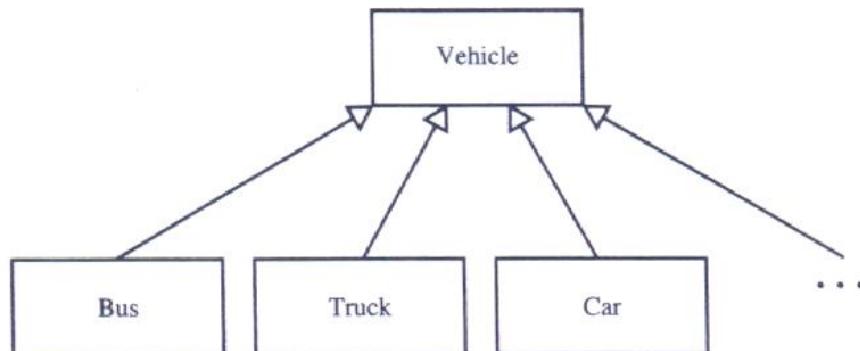
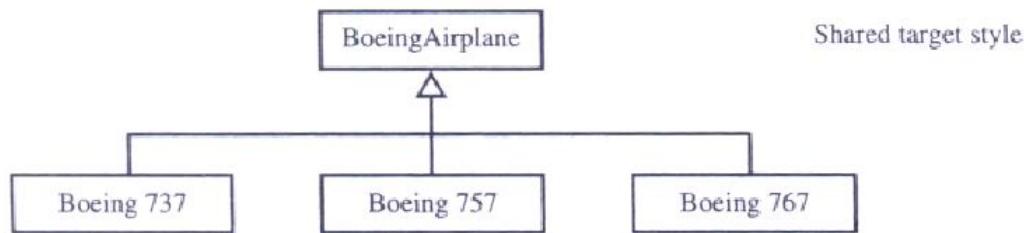
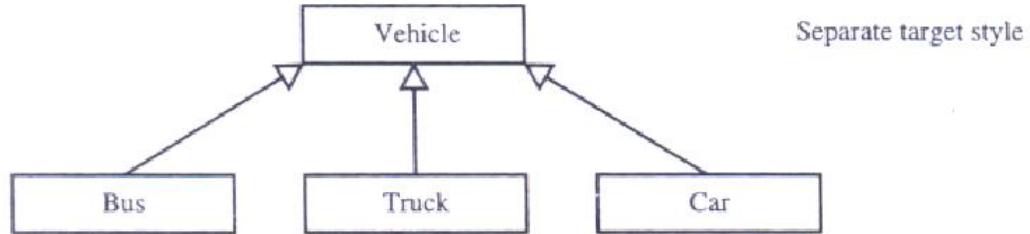


Different ways to show composition.

at the superclass end (see Fig.). The UML allows a *discriminator* label to be attached to a generalization of the superclass. For example, the class Boeing-Airplane has instances of the classes Boeing 737, Boeing 747, Boeing 757, and Boeing 767, which are subclasses of the class BoeingAirplane. Ellipses (...) indicate that the generalization is incomplete and more subclasses exist that are not shown (see Fig.). The constructor complete indicates that the generalization is complete and no more subclasses are needed. If a text label is placed on the hollow triangle shared by several generalization paths to

subclasses, the label applies to all of the paths. In other words, all subclasses share the given properties.

Generalization notation.



Ellipses (...) indicate that additional classes exist and are not shown.

UML CLASS DIAGRAM

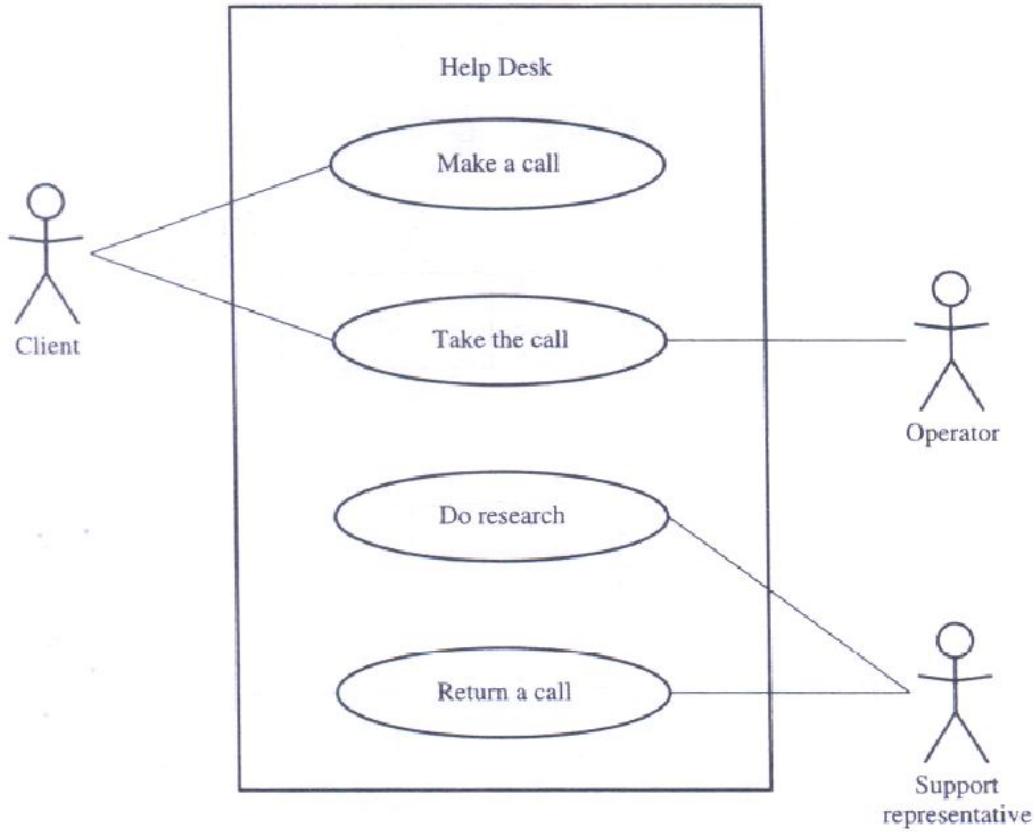
The use-case concept was introduced by Ivar Jacobson in the object-oriented software engineering (OOSE) method . The functionality of a system is described in a number of different use cases, each of which represents a specific flow of events in the system.

A use case corresponds to a sequence of transactions, in which each transaction is invoked from outside the system (actors) and engages internal objects to interact with one another and with the system's surroundings.

The description of a use case defines what happens in the system when the use case is performed. In essence, the use-case model defines the outside (actors) and inside (use case) of the system's behavior. Use cases represent specific flows of events in the system. The use cases are initiated by actors and describe the flow of events that these actors set off. An actor is anything that interacts with a use case:

It could be a human user, external hardware, or another system. An actor represents a category of user rather than a physical user. Several physical users can play the same role. For example, in terms of a Member actor, many people can be members of a library, which can be represented by one actor called *Member*.

A *use-case diagram* is a graph of actors, a set of use cases enclosed by a system boundary, communication (participation) associations between the actors and the use cases, and generalization among the use cases.

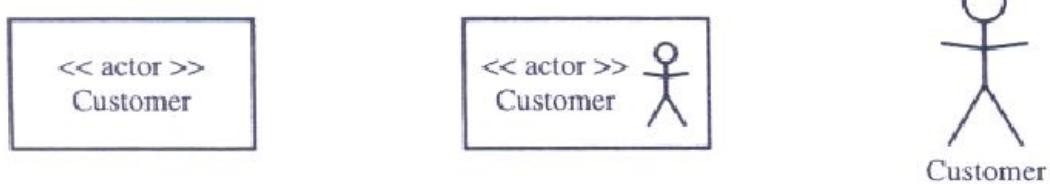


A use-case diagram shows the relationship among actors and use cases within a system. Fig. 1.10 diagrams use cases for a Help Desk. A use-case diagram shows the relationship among the actors and use cases within a system. A client makes a call that is taken by an operator, who determines the nature of the problem. Some calls can be answered immediately; other calls require research and a return call. A use case is shown as an ellipse containing the name of the use case. The name of the use case can be placed below or inside the ellipse. Actors' names and use case names should follow the capitalization and punctuation guidelines of the model.

An actor is shown as a class rectangle with the label <<actor>>, or the label and a stick figure, or just the stick figure with the name of the actor below the fig (see Fig.).

FIG

The three representations of an actor are equivalent.



These relationships are shown in a use-case diagram:

1. *Communication.* The communication relationship of an actor in a use case is shown by connecting the actor symbol to the use-case symbol with a solid path. The actor is said to "communicate" with the use case.
2. *Uses.* A uses relationship between use cases is shown by a generalization arrow from the use case.
3. *Extends.* The extends relationship is used when you have one use case that is similar to another use case but does a bit more. In essence, it is like a subclass.

9.7 LET US SUM UP

A model is a simplified representation of reality, simplified because reality is too complex or large and much of the complexity actually is irrelevant to the problem being described or solved. The unified modeling language was developed by Booch, Jacobson, and Rumbaugh. The UML encompasses the unification of their modeling notations. The UML class diagram is the main static structure analysis diagram for the system. It represents the class structure of a system with relationships between classes and inheritance structure. The class diagram is developed through use-case, sequence, and collaboration diagrams. The use-case diagram captures information on how the system or business works or how you wish it to work. It is a scenario-building approach in which you model the processes of the system. It is an excellent way to lead into object-oriented analysis of the system.

9.8 POINTS FOR DISCUSSION

1. Justify Static Model.
2. Justify Dynamic Model.

9.9 LESSON – END ACTIVITIES

1. Discuss N-ary Associations
2. Validate UML Diagrams.
3. Discuss OR Relation.

9.10 REFERENCES

1. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
2. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
3. Rumbaugh, James ,Michael –Object oriented Modelling and design
4. Ali –Brahmi –Object oriented system development”

LESSON – 10: UML DYNAMIC MODELING

CONTENTS

- 10.0 AIMS AND OBJECTIVES
- 10.1 INTRODUCTION.
- 10.2 UML DYNAMIC MODELING (BEHAVIOR DIAGRAMS)
- 10.3. UML INTERACTION DIAGRAMS
- 10.4 UML STATECHART DIAGRAM
- 10.5 UML ACTIVITY DIAGRAM
- 10.6 IMPLEMENTATION DIAGRAMS
- 10.7 MODEL MANAGEMENT: PACKAGES AND MODEL ORGANIZATION
- 10.8 LET US SUM UP
- 10.9 POINTS FOR DISCUSSION
- 10.10 LESSON – END ACTIVITIES
- 10.11 REFERENCES

10.0 AIMS AND OBJECTIVES

You can express the dynamic semantics of a problem with the following diagrams:

Behavior diagrams (dynamic):

Interaction diagrams:

Sequence diagrams

Collaboration diagrams

Statechart diagrams

Activity diagrams

10.1 INTRODUCTION

It is impossible to capture all details of a complex system in just one model or view. KIleyan and Gingrich explain:

One must understand both the structure and the function of the objects involved. One must understand the taxonomic structure of class objects, the inheritance and mechanisms used, the individual behaviors of objects, and the dynamic behavior of the system as a whole. The problem is somewhat analogous to that of viewing a sports event such as tennis or a football game. Many different camera angles are required to provide an understanding of the action taking place. Each camera reveals particular aspects of the action that could not be conveyed by one camera alone.

10.2 UML DYNAMIC MODELING (BEHAVIOR DIAGRAMS)

The diagrams we have looked at so far largely are static. However, events happen dynamically in all systems: Objects are created and destroyed, objects send messages to one another in an orderly fashion, and in some systems, external events trigger operations on certain objects. Furthermore, objects have states. The state of an object would be difficult to capture in a static model. The state of an object is the result of its behavior. Booch provides us an excellent example:

"When a telephone is first installed, it is in idle state, meaning that no previous behavior is of great interest and that the phone is ready to initiate and receive calls. When someone picks up the handset, we say that the phone is now off-hook and in the dialing state; in this state, we do not expect the phone to ring: we expect to be able to initiate a conversation with a party or parties on another telephone. When the phone is on-hook, if it rings and then we pick up the handset, the phone is now in the receiving state, and we expect to be able to converse with the party that initiated the conversation ." Booch explains that describing a systematic event in a static medium such as on a sheet of paper is difficult, but the problem confronts almost every discipline.

Each class may have an associated activity diagram that indicates the behavior of the class's instance (its object). In conjunction with the use-case model, we may provide a script or an interaction diagram to show the time or event ordering of messages as they are evaluated .

10.3. UML INTERACTION DIAGRAMS

Interaction diagrams are diagrams that describe how groups of objects collaborate to get the job done. *Interaction diagrams* capture the behavior of a single use case, showing the pattern of interaction among objects. The diagram shows a number of example objects and the messages passed between those objects within the use case . There are two kinds of interaction models: sequence diagrams and collaboration diagrams.

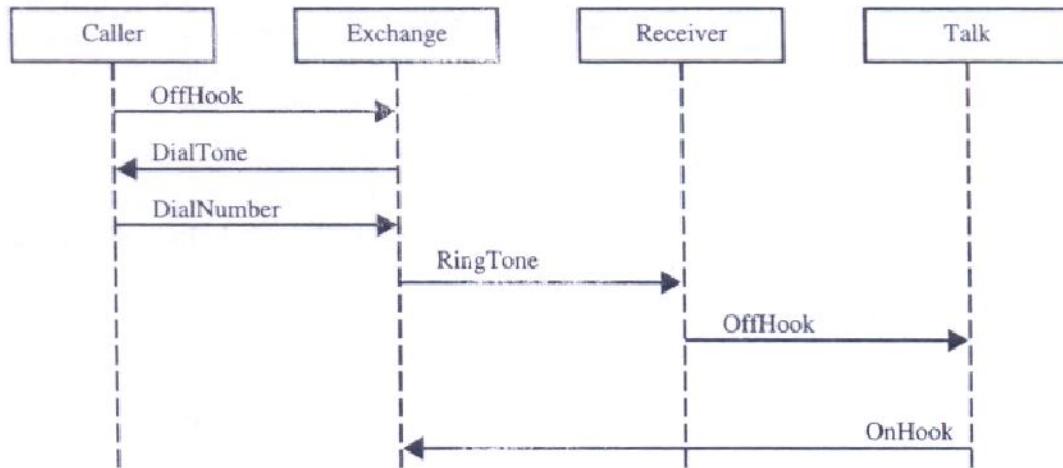
UML Sequence Diagram *Sequence diagrams* are an easy and intuitive way of describing the behavior of a system by viewing the interaction between the system and its environment. A sequence diagram shows an interaction arranged in a time sequence. It shows the objects participating in the interaction by their lifelines and the messages they exchange, arranged in a time sequence.

A sequence diagram has two dimensions: the vertical dimension represents time, the horizontal dimension represents different objects. The vertical line is called the object's *lifeline*. The *lifeline* represents the object's existence during the interaction. This form was first popularized by Jacobson. An object is shown as a box at the top of a dashed vertical line (see Fig). A role is a slot for an object within a collaboration that describes the type of object that may play the role and its relationships to other roles.

However, a sequence diagram does not show the relationships among the roles or the association among the objects. An object role is shown as a vertical dashed line, the lifeline.

An example of a sequence diagram.

Telephone Call



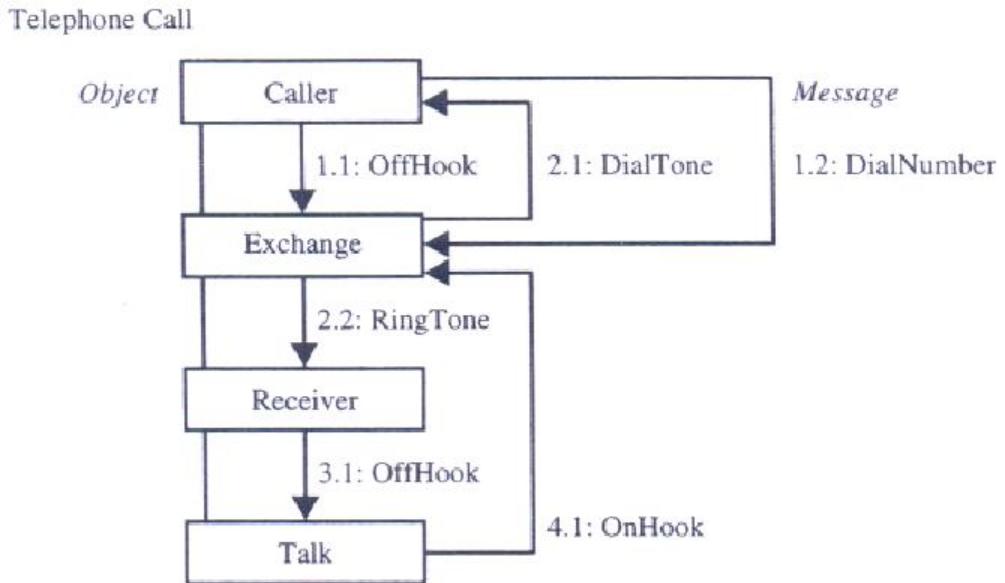
Each message is represented by an arrow between the lifelines of two objects. The order in which these messages occur is shown top to bottom on the page. Each message is labeled with the message name. The label also can include the argument and some control information and show self-delegation, a message that an object sends to itself, by sending the message arrow back to the same lifeline. The horizontal ordering of the lifelines is arbitrary. Often, call arrows are arranged to proceed in one direction across the page, but this is not always possible and the order conveys no information.

The sequence diagram is very simple and has immediate visual appeal-this is its great strength. A sequence diagram is an alternative way to understand the overall flow of the control of a program. Instead of looking at the code and trying to find out the overall sequence of behavior, you can use the sequence diagram to quickly understand that sequence .

UML Collaboration Diagram Another type of interaction diagram is the collaboration diagram. A *collaboration diagram* represents a collaboration, which is a set of objects related in a particular context, and interaction, which is a set of messages exchanged among the objects within the collaboration to achieve a desired outcome. In a collaboration diagram, objects are shown as figs. As in a sequence diagram, arrows indicate the message sent within the given use case. In a collaboration diagram, the sequence is indicated by numbering the messages.

Some people argue that numbering the messages makes it more difficult to see the sequence than drawing the lines on the page. However, since the collaboration diagram is more compressed, other things can be shown more easily-for example, how the objects are linked together-and the layout can be overlaid with packages or other information.

A collaboration diagram provides several numbering schemes. The simplest is illustrated in Fig 5-16. You can also use a decimal numbering scheme (see Fig-) A collaboration diagram with simple numbering.



A collaboration diagram with decimal numbering.

where 1.2: DialNumber means that the Caller
 (1) is calling the Exchange
 (2); hence, the number 1.2.

The UML uses the decimal scheme because it makes it clear which operation is calling which other operation, although it can be hard to see the overall sequence . Different people have different preferences when it comes to deciding whether to use sequence or collaboration diagrams. Fowler and Scott suggest that a sequence diagram is easier to read. Others prefer a collaboration diagram, because they can use the layout to indicate how objects are statically connected . Fowler and Scott argue that the main advantage of interaction diagrams (both collaboration and sequence) is simplicity. You easily can see the message by looking at the diagram. The disadvantage of interaction diagrams is that they are great only for representing a single sequential process; they begin to break down when you want to represent conditional looping behavior. However, conditional behavior can be represented in sequence or collaboration diagrams through two methods. The preferred method is to use separate diagrams for each scenario. Another way is to use conditions on messages to indicate the behavior. The main guideline in developing interaction diagrams is simplicity. The interaction diagram loses its clarity with more complex conditional behavior. If you want to capture complex behavior in a single diagram, use an activity diagram, which will be described in a later section.

An interaction diagram basically is used to examine the behavior of objects within a single use case. It is good at showing collaboration among the objects but not so good at precise definition of the behavior .

10.4 UML STATECHART DIAGRAM

A *statechart diagram* (also called a *state diagram*) shows the sequence of states that an object goes through during its life in response to outside stimuli and messages. The state is the set of values that describes an object at a specific point in time and is represented by state symbols and the transitions are represented by arrows connecting the state symbols. A statechart diagram may contain subdiagrams. A *state diagram* represents *the state of the method execution (that is, the state of the object executing the method)*, and *the activities in the diagram* represent *the activities of the object that performs the method*. The purpose of the state diagram is to understand the algorithm involved in performing a method. To complete an object-oriented design, the activities within the diagram must be assigned to objects and the control flows assigned to links in the object diagram.

A statechart diagram is similar to a Petri net diagram, where a token (shown by a solid black dot) represents an activity symbol. When an activity symbol appears within a state symbol, it indicates the execution of an operation. Executing a particular step within the diagram represents a state within the execution of the overall method. The same operation name may appear more than once in a state diagram, indicating the invocation of the same operation in a different phase. An outgoing solid arrow attached to a statechart symbol indicates a transition triggered by the completion of the activity. The name of this implicit event need not be written, but conditions that depend on the result of the activity or other values may be included. An event occurs at the instant in time when the value is changed.

A message is data passed from one object to another. At a minimum, a message is a name that will trigger an operation associated with the target object; for example, an Employee object that contains the name of an employee. If the Employee object received a message (*getEmployeeName*) asking for the name of the employee, an operation contained in the Employee class (e.g., *returnEmployeeName*) would be invoked. That operation would check the attribute Employee and then assign the value associated with that attribute back to the object that sent the message in the first place. In this case, the state of the Employee object would not have been changed. Now, consider a situation where the same Employee object received a message *updateEmployeeAddress*) that contained a parameter (2000 21st Street, Seattle, WA): *updateEmployeeAddress* (2000 21st Street, Seattle, WA) In this case the object would invoke an operation from its class that would modify the value associated with the attribute Employee, changing it from the old address to the new address; therefore, the state of the employee object has been changed.

A state is represented as a rounded box, which may contain one or more compartments. The compartments are all optional. The name compartment and the internal transition compartment are two such compartments:

The *name compartment* holds the optional name of the state. States without names are "anonymous" and all are distinct. Do not show the same named state twice in the same diagram, since it will be very confusing.

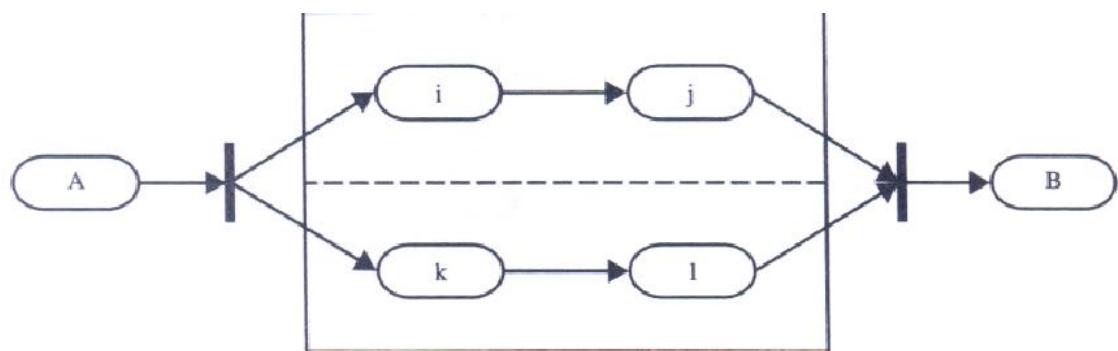
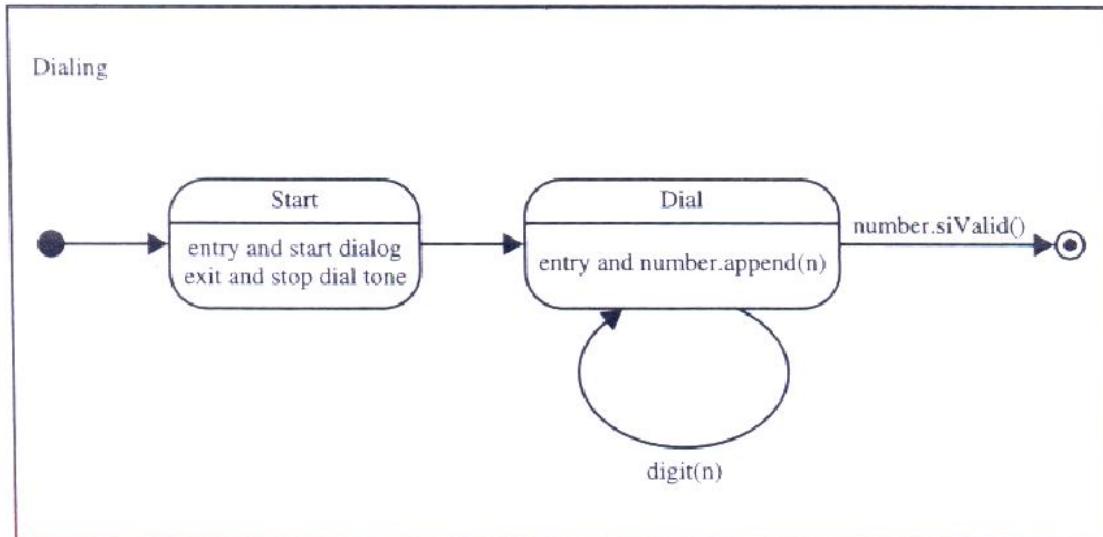
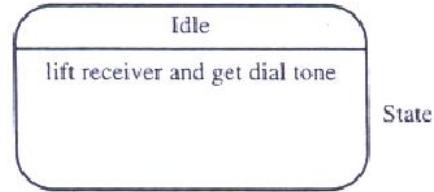
The *internal transition compartment* holds a list of internal actions or activities performed in response to events received while the object is in the state, without changing states:

The syntax used is this: event-name argument-list / action-expression; for example, help / display help.

Two special events are *entry* and *exit*, which are reserved words and cannot be used for event names. These terms are used in the following ways: entry *I* actionexpression (the action is to be performed on entry to the state) and exit *I* actionexpressed (the action is to be performed on exit from the state). The statechart supports nested state machines; to activate a substate machine use the keyword *do*: do *I* machine-name (argument-list). If this state is entered, after the entry action is completed, the nested (sub)state machine will be executed with its initial state. When the nested state machine reaches its final state, it will exit the action of the current state, and the current state will be considered completed. An initial state is shown as a small dot, and the transition from the initial state may be labeled with the event that creates the objects; otherwise, it is unlabeled. If unlabeled, it represents any transition to the enclosing state.

A final state is shown as a circle surrounding a small dot, a bull's-eye. This represents the completion of activity in the enclosing state and triggers a transition on the enclosing state labeled by the implicit activity completion event, usually displayed as an unlabeled transition (see Fig.). The transition can be simple or complex. A simple transition is a relationship between two states indicating that an object in the first state will enter the second state and perform certain actions when a specific event occurs; if the specified

A simple state Idle and a nested state. The dialing state contains substates, which consist of start and dial states.



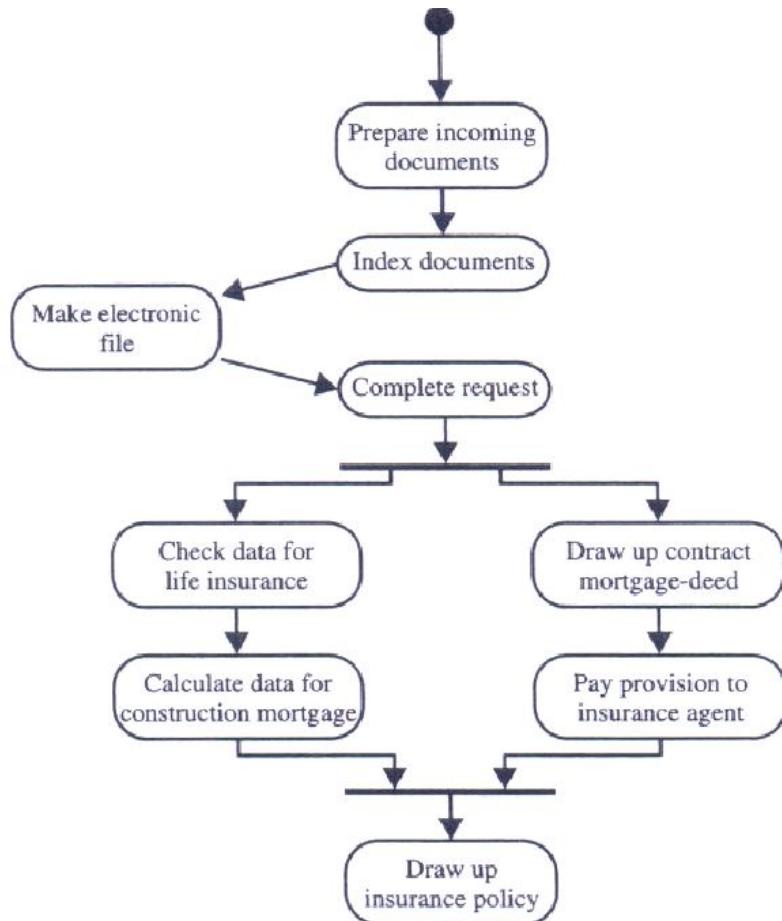
A complex transition.

conditions are satisfied, the transition is said to "fire." Events are processed one at a time. An event that triggers no transition is simply ignored. A complex transition may have multiple source and target states. It represents a synchronization or a splitting of control into concurrent threads. A complex transition is enabled when all the source states are changed, after a complex transition "fires" all its destination states. A complex transition is shown as a short heavy bar.! The bar may have one or more solid arrows from states to the bar (these are source states); the bar also may have one or more solid arrows from the bar to states (these are the destination states). A transition string may be shown near the bar. Individual arrows do not have their own transition strings (see Fig 5-19).

There certainly is no reason to prepare a state diagram for each class in your system. Indeed, many developers create rather large systems without bothering to create any state diagrams. However, state diagrams are useful when you have a class that is very dynamic. In that situation, it often is helpful to prepare a state diagram to be sure you understand each of the possible states an object of the class could take and what event (message) would trigger each transition from one state to another. In effect, state diagrams emphasize the use of events and states to determine the overall activity of the system.

10.5 UML ACTIVITY DIAGRAM

An *activity diagram* is a variation or special case of a state machine, in which the states are activities representing the performance of operations and the transitions are triggered by the completion of the operations. Unlike state diagrams that focus on the events occurring to a single object as it responds to messages, an activity diagram can be used to model an entire business process. The purpose of an activity diagram is to provide a view of flows and what is going on inside a use case or among several classes. However, activity diagram can also be used to represent a class's method implementation as we will see throughout the book.

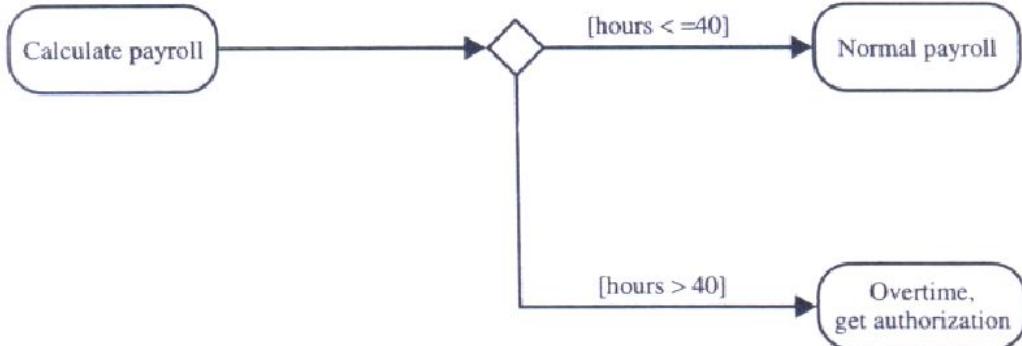


An activity diagram for processing mortgage requests (Loan: Processing Mortgage Request).

An activity model is similar to a statechart diagram, where a token (shown by a black dot) represents an operation. An activity is shown as a round box, containing the name of the operation. When an operation symbol appears within an activity diagram or other state diagram, it indicates the execution of the operation.

Executing a particular step within the diagram represents a state within the execution of the overall method. The same operation name may appear more than once in a state diagram, indicating the invocation of the same operation in different phases. An outgoing solid arrow attached to an activity symbol indicates a transition triggered by the completion of the activity. The name of this implicit event need not be written, but the conditions that depend on the result of the activity or other values may be included (see Fig.). Several transitions with different conditions imply a branching off of control. If conditions are not disjoint, then the branch is nondeterministic. The concurrent control is represented by multiple arrows leaving a synchronization bar, which is represented by a short thick bar with incoming and outgoing arrows. Joining concurrent control is expressed by multiple arrows entering the synchronization bar. The activity diagram

depicted in Fig 5-20, "Process Mortgage Request," is a multistep operation, all of which are completed before the single operation Draw up insurance policy.



A decision.

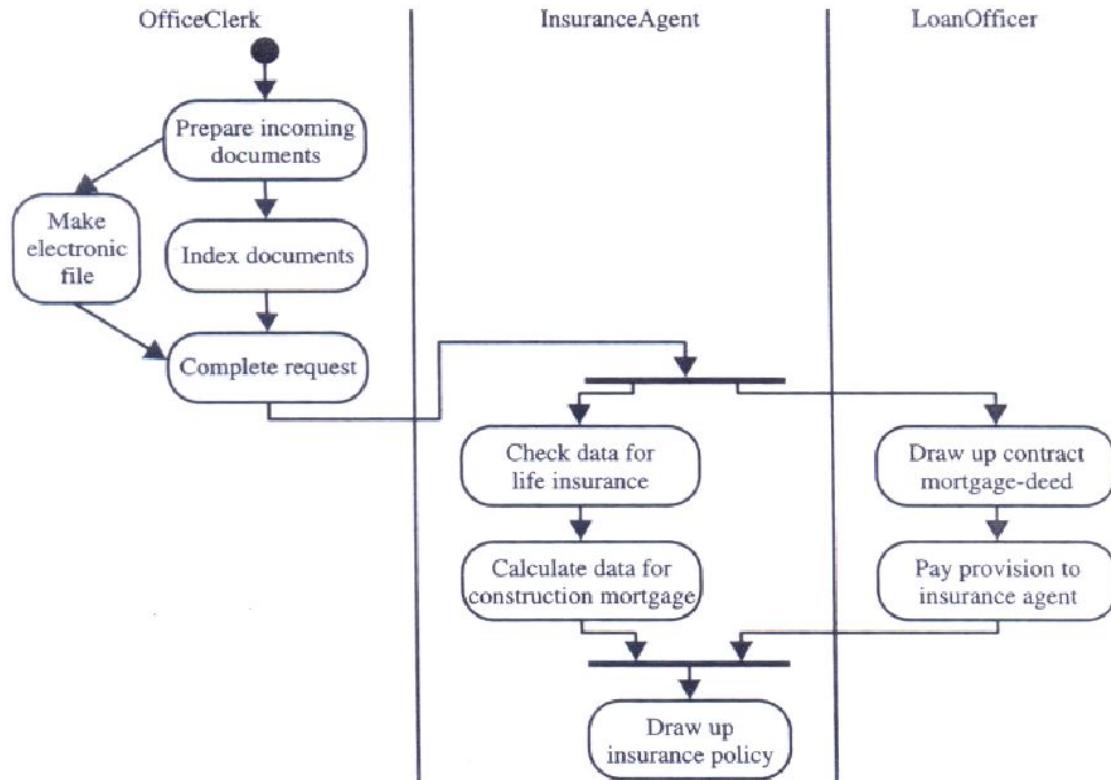
An activity diagram is used mostly to show the internal state of an object, but external events may appear in them. An external event appears when the object is in a "wait state," a state during which there is no internal activity by the object and the object is waiting for some external event to occur as the result of an activity by another object (such as a user input or some other signal). The two states are wait state and activity state. More than one possible event might take the object out of the wait state; the first one that occurs triggers the transition. A wait state is the "normal" state.

Activity and state diagrams express a decision when conditions (the UML calls them *guard conditions*) are used to indicate different possible transitions that depend on Boolean conditions of container object. The fig provided for a decision is the traditional diamond shape, with one or more incoming arrows and two or more outgoing arrows, each labeled by a distinct guard condition. All possible outcomes should appear on one of the outgoing transitions (see Fig). Actions may be organized into swimlanes, each separated from neighboring swimlanes by vertical solid lines on both sides. Each *swimlane* represents responsibility for part of the overall activity and may be implemented by one or more objects. The relative ordering of the swimlanes has no semantic significance but might indicate some affinity. Each action is assigned to one swimlane. A transition may cross lanes; there is no significance to the routing of the transition path (see Fig).

10.6 IMPLEMENTATION DIAGRAMS

Implementation diagrams show the implementation phase of systems development, such as the source code structure and the run-time implementation structure. There are two types of implementation diagrams: Component diagrams show the structure of the code itself, and deployment diagrams show the structure of the runtime system. These are relatively simple, high-level diagrams compared with the diagrams we have considered so far. Although we look at component-based development

later in this book, a full discussion of implementation is beyond the scope of this book. This section is included to show the place of implementation in the UML.



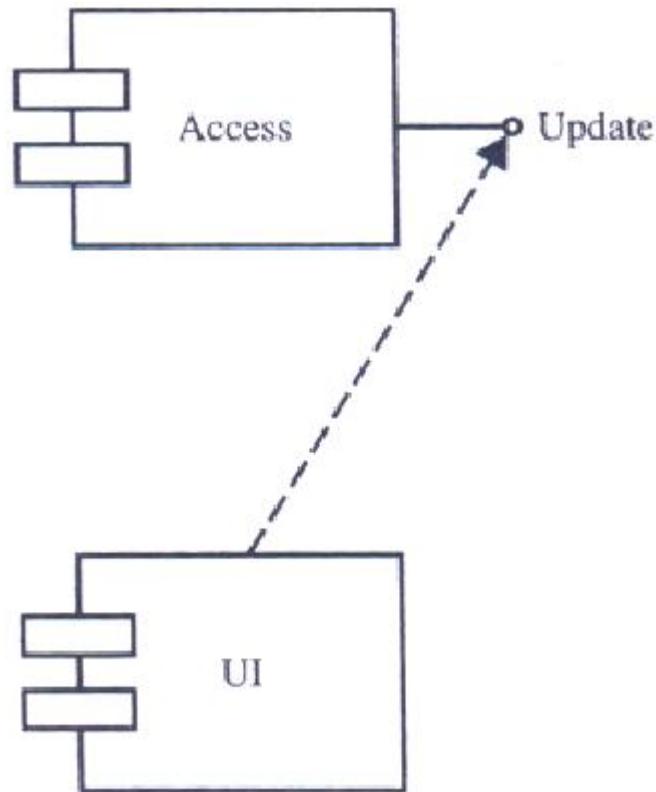
Swimlanes in an activity diagram.

Component Diagram *Component diagrams* model the physical components (such as source code, executable program, user interface) in a design. These high-level physical components may not be equivalent to the many smaller components you use in the creation of your application. For example, a user interface may contain many other off-the-shelf components purchased to put together a graphical user interface. Another way of looking at components is the concept of packages. A package is used to show how you can group together classes, which in essence are smaller scale components. Packages will be covered in the next section, but a point worth mentioning here is that a package usually will be used to group logical components of the application, such as classes, and not necessarily physical components.

However, the package could be a first approximation of what eventually will turn into physical grouping. In that case, the package will become a component .

A component diagram is a graph of the design's components connected by dependency relationships. A component is represented by the boxed figure shown in Fig .

Dependency is shown as a dashed arrow. 5.8.4.2 Deployment Diagram *Deployment diagrams* show the configuration of run-time processing elements and the software components, processes, and objects that live in them. Software component instances represent run-time manifestations

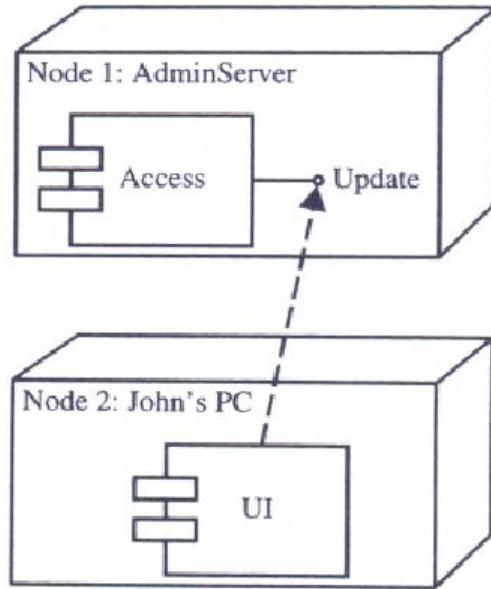


A component diagram.

of code units. In most cases, component diagrams are used in conjunction with deployment diagrams to show how physical modules of code are distributed on various hardware platforms. In many cases, component and deployment diagrams can be combined .

A deployment diagram is a graph of nodes connected by communication association. Nodes may contain component instances, which means that the component lives or runs at that node. Components may contain objects; this indicates that the object is part of the component. Components are connected to other components by dashed-arrow dependencies, usually through interfaces, which indicate one component uses the

services of another. Each node or processing element in the The basic UML notation for a deployment diagram.



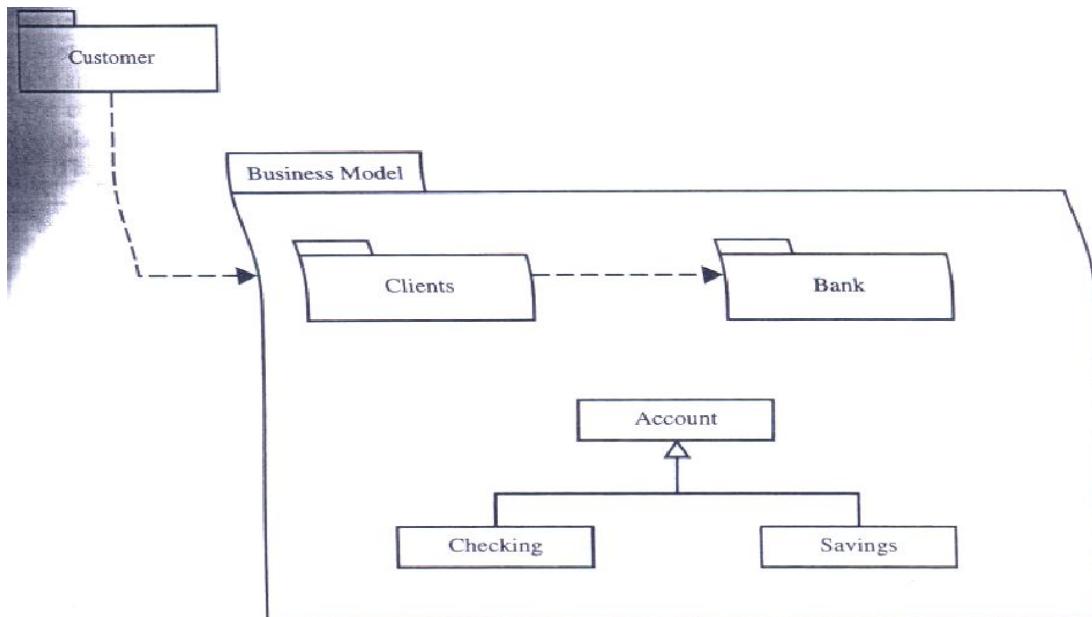
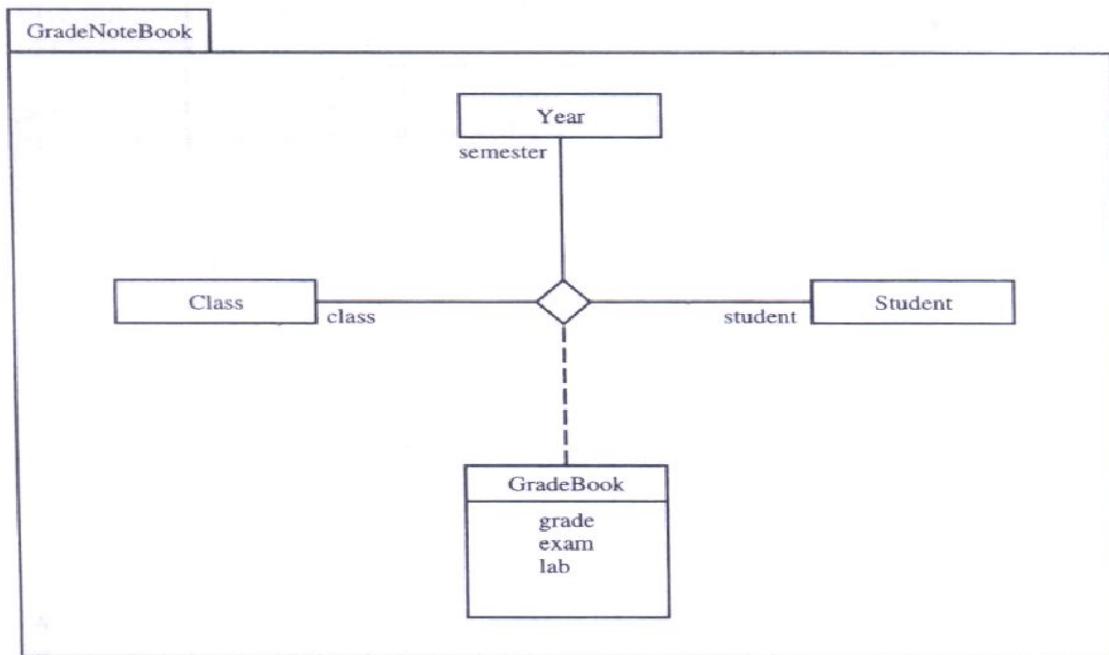
system is represented by a three-dimensional box. Connections between the nodes (or platforms) themselves are shown by solid lines (see Fig 5-24).

10.7 MODEL MANAGEMENT: PACKAGES AND MODEL ORGANIZATION

A *package* is a grouping of model elements. Packages themselves may contain other packages. A package may contain both subordinate packages and ordinary model elements. The entire system can be thought of as a single high-level package with everything else in it. All UML model elements and diagrams can be organized into packages.

A package is represented as a folder, shown as a large rectangle with a tab attached to its upper left corner. If contents of the package are not shown, then the name of the package is placed within the large rectangle. If contents of the package are shown, then the name of the package may be placed on the tab (see Fig). The contents of the package are shown within the large rectangle. Fig shows an example of several packages. This fig shows three packages (Clients, Bank, and Customer) and three classes (Account class, Savings class, and Checking class) inside the Business Model package. A real model would have many more classes in each package. The contents might be shown if they

A package and its contents.

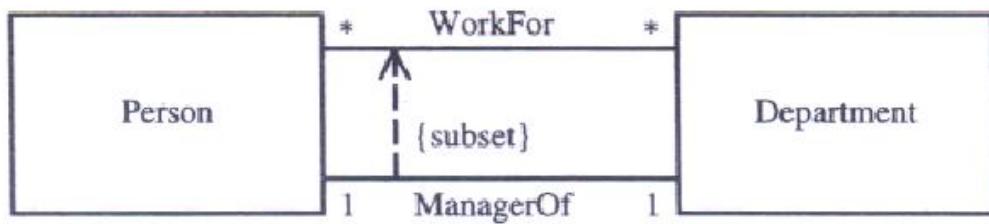


A package and its dependencies.

are small, or they might be suppressed from higher levels. The entire system is a package. Fig 5-26 also shows the hierarchical structure, with one package dependent on other packages. For example, the Customer depends on the package Business Model, meaning that one or more elements within Customer depend on one or more elements within the other packages. The package Business Model is shown partially expanded. In this case, we see that the package Business Model owns the classes Bank, Checking, and Savings as well as the packages Clients and Bank. Ownership may be shown by a graphic nesting of

the figs or by the expansion of a package in a separate drawing. Packages can be used to designate not only logical and physical groupings but also use-case groups. A use-case group, as the name suggests, is a package of use cases.

Model dependency represents a situation in which a change to the target element may require a change to the source element in the dependency, thus indicating the relationship between two or more model elements. It relates the model elements themselves and does not require a set of instances for its meaning. A dependency is shown as a dashed arrow from one model element to another on which the first element is dependent (see Fig 5-27).



An example of constraints. A person is a manager of people who work for the accounting department.

10.8 LET US SUM UP

In the UML sequence diagram is for dynamic modeling, where objects are represented by vertical lines and messages passed back and forth between the objects are modeled by horizontal vectors between the objects. The UML collaboration diagram is an alternative view of the sequence diagram, showing in a scenario how objects interrelate with one another. Statechart diagrams, another form of dynamic modeling, focus on the events occurring within a single object as it responds to messages; an activity diagram is used to model an entire business process. Thus, an activity model can represent several different classes.

10.9 POINTS FOR DISCUSSION

1. Establish UML interaction Diagrams
2. Establish UML sequence Diagrams
3. Evaluate UML Collaboration Diagram
4. Critically analyze Component Diagram
5. Critically analyze Deployment Diagram

10.10 LESSON – END ACTIVITIES

1. Establish UML Activity Diagram.
2. Discuss about a package.

10.11 REFERENCES

1. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
2. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
3. Rumbaugh, James ,Michael –Object oriented Modelling and design
- 4.Ali –Brahmi –Object oriented system development”

UNIT – III:

LESSON – 11: OBJECT ORIENTED ANALYSIS PROCESS

CONTENTS

- 11.0 AIMS AND OBJECTIVES
- 11.1 INTRODUCTION
- 11.2 WHY ANALYSIS IS A DIFFICULT ACTIVITY
- 11.3 BUSINESS OBJECT ANALYSIS: UNDERSTANDING THE BUSINESS LAYER
- 11.4 USE.CASE DRIVEN OBJECT.ORIENTED ANALYSIS: THE UNIFIED APPROACH
- 11.5 BUSINESS PROCESS MODELING
- 11.6 USE-CASE MODEL
- 11.7 LET US SUM UP
- 11.8 POINTS FOR DISCUSSION
- 11.9 LESSON – END ACTIVITIES
- 11.10 REFERENCES

11.0 AIMS AND OBJECTIVES

You should be able to define and understand
 The object-oriented analysis process.
 The use-case modeling and analysis.
 Identifying actors.
 Identifying use cases.

11.1 INTRODUCTION

Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirement. The goal of object oriented analysis is to understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system.

The first step in finding an appropriate solution to a given problem is to understand the problem and its domain. The main objective of the analysis is to capture a complete, unambiguous, and consistent picture of the requirements of the system and

what the system must do to satisfy the users' requirements and needs. This is accomplished by constructing several models of the system that concentrate on describing what the system does rather than how it does it. Separating the behavior of a system from the way that behavior is implemented requires viewing the system from the perspective of the user rather than that of the machine.

Analysis is the process of transforming a problem definition from a fuzzy set of facts and myths into a coherent statement of a system's requirements. In previous chapter we looked at the software development process as three basic transformations. The objective of this chapter is to describe Transformation 1, which is the transformation of the users' needs into a set of problem statements and requirements (also known as *requirement determination*). In this phase of the software process, you must analyze how the users will use the system and what is needed to accomplish the system's operational requirements. Analysis involves a great deal of interaction with the people who will be affected by the system, including the actual users and anyone else on whom its creation will have an impact. The analyst has four major tools at his or her disposal for extracting information about a system:

1. Examination of existing system documentation
2. Interviews
3. Questionnaire
4. Observation

In addition, there are minor methods, such as literature review. However, these activities must be directed by a use-case model that can capture the user requirements. The inputs to this phase are the users' requirements, both written and oral, which will be reduced to the model of the required operational capability of the system.

An object-oriented environment allows the same set of models to be used for analysis, design, and implementation. The analyst is concerned with the uses of the system, identifying the objects and inheritance, and thinks about the events that change the state of objects. The designer adds detail to this model, perhaps designing screens, user interaction, and database access. The thought process flows so naturally from analyst to designer that it may be difficult to tell where analysis ends and design begins.

11.2 WHY ANALYSIS IS A DIFFICULT ACTIVITY

Analysis is a creative activity that involves understanding the problem, its associated constraints, and methods of overcoming those constraints. This is an iterative process that goes on until the problem is well understood . Norman explains the three most common sources of requirement difficulties:

1. Fuzzy descriptions
2. Incomplete requirements
3. Unnecessary features

A common problem that leads to requirement ambiguity is a fuzzy and ambiguous description, such as "fast response time" or "very easy and very secure updating mechanisms." A requirement such as fast response time is open to interpretation, which

might lead to user dissatisfaction if the user's interpretation of a fast response is different from the systems analyst's interpretation. Incomplete requirements mean that certain requirements necessary for successful system development are not included for a variety of reasons. These reasons could include the users' forgetting to identify them, high cost, politics within the business, or oversight by the system developer. However, because of the iterative nature of object-oriented analysis and the unified approach most of the incomplete requirements can be identified in subsequent tries. When addressing features of the system, keep in mind that every additional feature could affect the performance, complexity, stability, maintenance, and support costs of an application. Features implemented by a small extension to the application code do not necessarily have a proportionally small effect on a user interface. For example, if the primary task is selecting a single object, extending it to support selection of multiple objects could make the frequent, simple task more difficult to carry out. A number of other factors also may affect the design of an application. For example, deadlines may require delivering a product to market with a minimal design process, or comparative evaluations may force considering additional features. Remember that additional features and shortcuts can affect the product.

There is no simple equation to determine when a design trade-off is appropriate. Analysis is a difficult activity. You must understand the problem in some application domain and then define a solution that can be implemented with software. Experience often is the best teacher. If the first try reflects the errors of an incomplete understanding of the problems, refine the application and try another run.

11.3 BUSINESS OBJECT ANALYSIS: UNDERSTANDING THE BUSINESS LAYER

Business object analysis is a process of understanding the system's requirements and establishing the goals of an application. The main intent of this activity is to understand users' requirements. The outcome of the business object analysis is to identify classes that make up the business layer and the relationships that play a role in achieving system goals.

To understand the users' requirements, we need to find out how they "use" the system. This can be accomplished by developing use cases. Use cases are scenarios for understanding system requirements.

In addition to developing use cases, which will be described in the next section, the uses and the objectives of the application must be discussed with those who are going to use it or be affected by the system. Usually, domain users or experts are the best authorities. Try to understand the expected inputs and desired responses.

Defer unimportant details until later. State *what* must be done, not *how* it should be done. This, of course, is easier said than done. Yet another tool that can be very useful for understanding users' requirements is preparing a prototype of the user interface. Preparation of a prototype usually can help you better understand how the system will be used, and therefore it is a valuable tool during business object analysis.

Having established what users want by developing use cases then documenting and modeling the application, we can proceed to the design and implementation. The unified approach (UA) steps can overlap each other. The process is iterative, and you may have to backtrack to previously completed steps for another try. Separating the *what* from the *how* is no simple process. Fully understanding a problem and defining how to implement it may require several tries or iterations. In this chapter, we see how a use-case model can assist us in capturing an application's requirements.

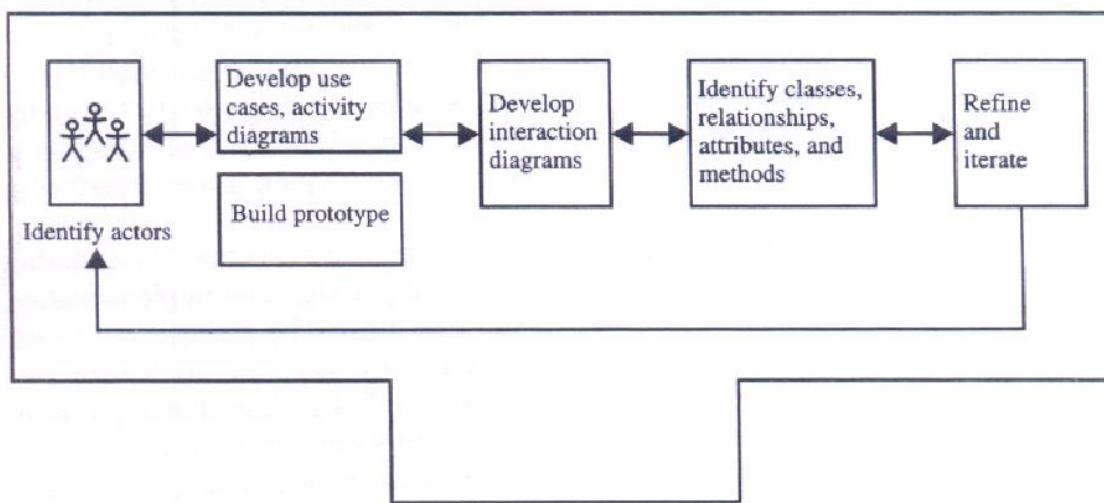
11.4 USE.CASE DRIVEN OBJECT.ORIENTED ANALYSIS: THE UNIFIED APPROACH

The object-oriented analysis (OOA) phase of the unified approach uses actors and use cases to describe the system from the users' perspective. The *actors* are external factors that interact with the system; *use cases* are scenarios that describe how actors use the system. The use cases identified here will be involved throughout the development process.

The OOA process consists of the following steps :

1. Identify the actors: .Who is using the system? Or, in the case of a new system, who will be using the system?
2. Develop a simple business process model using UML activity diagram.
3. Develop the use case: .What are the users doing with the system?. Or, in case of the new system, what will users be doing with the system? .Use cases provide us with comprehensive documentation of the system under study.
4. Prepare interaction diagrams: .Determine the sequence. .Develop collaboration diagrams.
5. Classification-develop a static UML class diagram: .Identify classes. . Identify relationships. .Identify attributes. .Identify methods.
6. Iterate and refine: If needed, repeat the preceding steps.

The object-oriented analysis process in the Unified Approach (UA).



11.5 BUSINESS PROCESS MODELING

This is not necessarily the start of every project, but when required, business processes and user requirements may be modeled and recorded to any level of detail. This may include modeling as-is processes and the applications that support them and any number of phased, would-be models of reengineered processes or implementation of the system. These activities would be enhanced and supported by using an activity diagram. Business process modeling can be very time consuming, so the main idea should be to get a basic model without spending too much time on the process. The advantage of developing a business process model is that it makes you more familiar with the system and therefore the user requirements and also aids in developing use cases. For example, let us define the steps or activities involved in using your school library. These activities can be represented with an activity diagram.

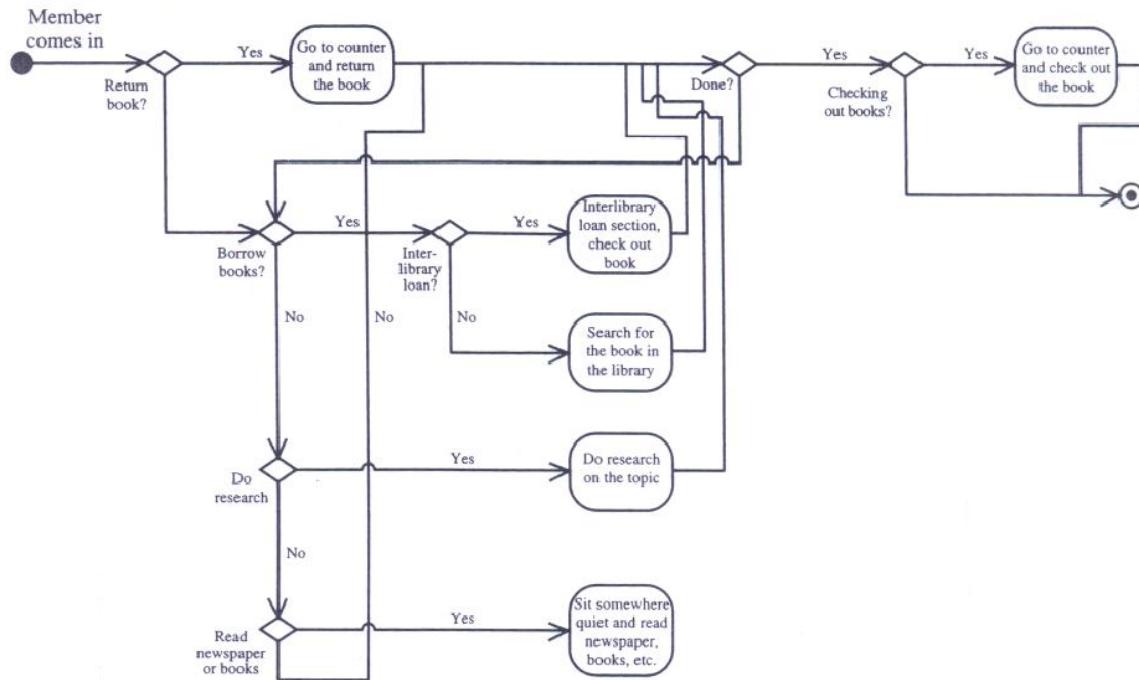
Developing an activity diagram of the business process can give us a better understanding of what sort of activities are performed in a library by a library member.

11.6 USE-CASE MODEL

Use cases are scenarios for understanding system requirements. A use-case model can be instrumental in project development, planning, and documentation of systems

FIGURE 6-2

This activity diagram (AD) shows some activities that can be performed by a library member.

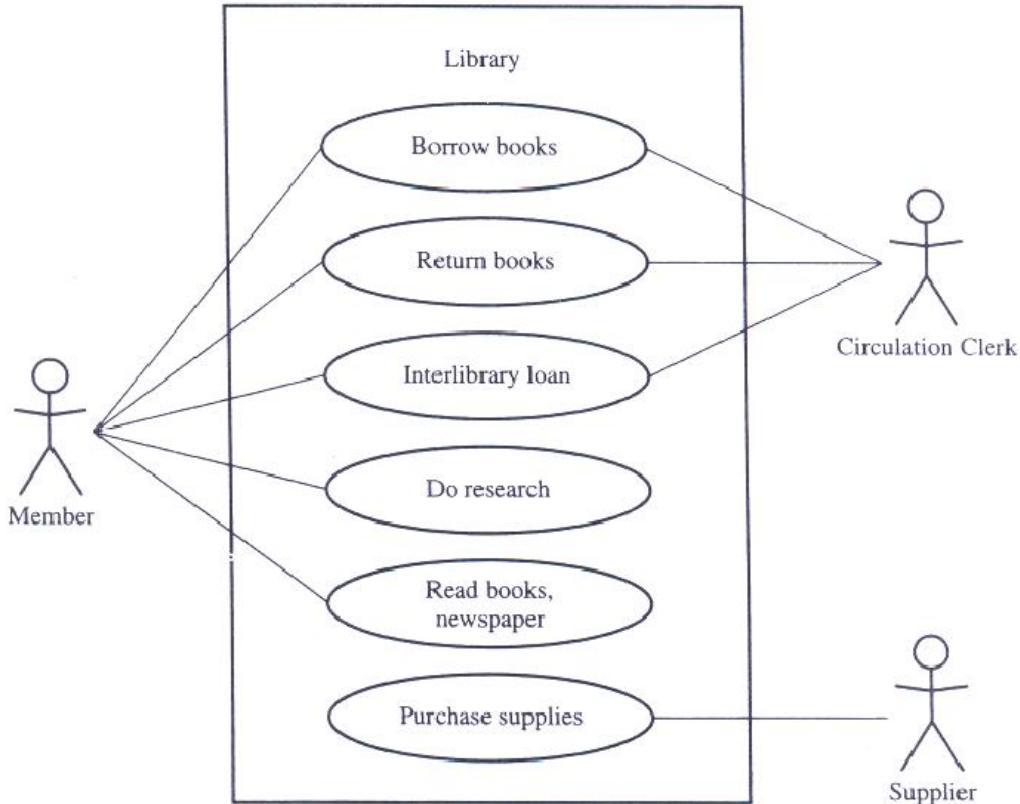


requirements. A use case is an interaction between users and a system; it captures the goal of the users and the responsibility of the system to its users)For example, take a car;

typical uses of a car include "take you different places" or "haul your stuff" or a user may want to use it "off the road." The use-case model describes the uses of the system and shows the courses of events that can be performed. In other words, it shows a system in terms of its users and how it is being used from a user point of view. Furthermore, it defines what happens in the system when the use case is performed. In essence, the use-case model tries to systematically identify uses of the system and therefore the system's responsibilities. A use-case model also can discover classes and the relationships among subsystems of the systems.

(A use-case model can be developed by talking to typical users and discussing the various things they might want to do with the application being prepared) Each use or scenario represents what the user wants to do. Each use case must have a name and short textual description, no more than a few paragraphs. Since the use-case model provides an external view of a system or application, it is directed primarily toward the users or the "actors" of the systems, not its implementers (see Figure).(The use-case model expresses what the business or application will do and not how; that is the responsibility of the UML class diagram-FIGURE

Some uses of a library. As you can see, these are uses of external views of the library system by an actor such as a member, circulation clerk, or supplier instead of a developer of the library system. The simpler the use-case model, the more effective it will be. It is not wise to capture all the details right at the start; you can do that later.



gram. The UML class diagram, also called an object model, represents the static relationships between objects, inheritance, association, and the like. The object model represents an internal view of the system, as opposed to the use-case model, which represents the external view of the system. The object model shows how the business is run. Jacobson, Ericsson, and Jacobson call the use-case model a "what model," in contrast to the object model, which is a "how model."¹

Use Cases under the Microscope

An important issue that can assist us in building correct use cases is the differentiation between user goals and system interactions. Use cases represent the things that the user is doing with the system, which can be different from the users' goals:)However, by focusing on users' goals first, we can come up with use cases to satisfy them. Let us take a closer look at the definition of use case b.y Jacobson et al., italics added to highlight the words that are discussed next]: "A Use Case is a sequence of transactions in a system whose task is to yield results of measurable value to an individual actor of the system."

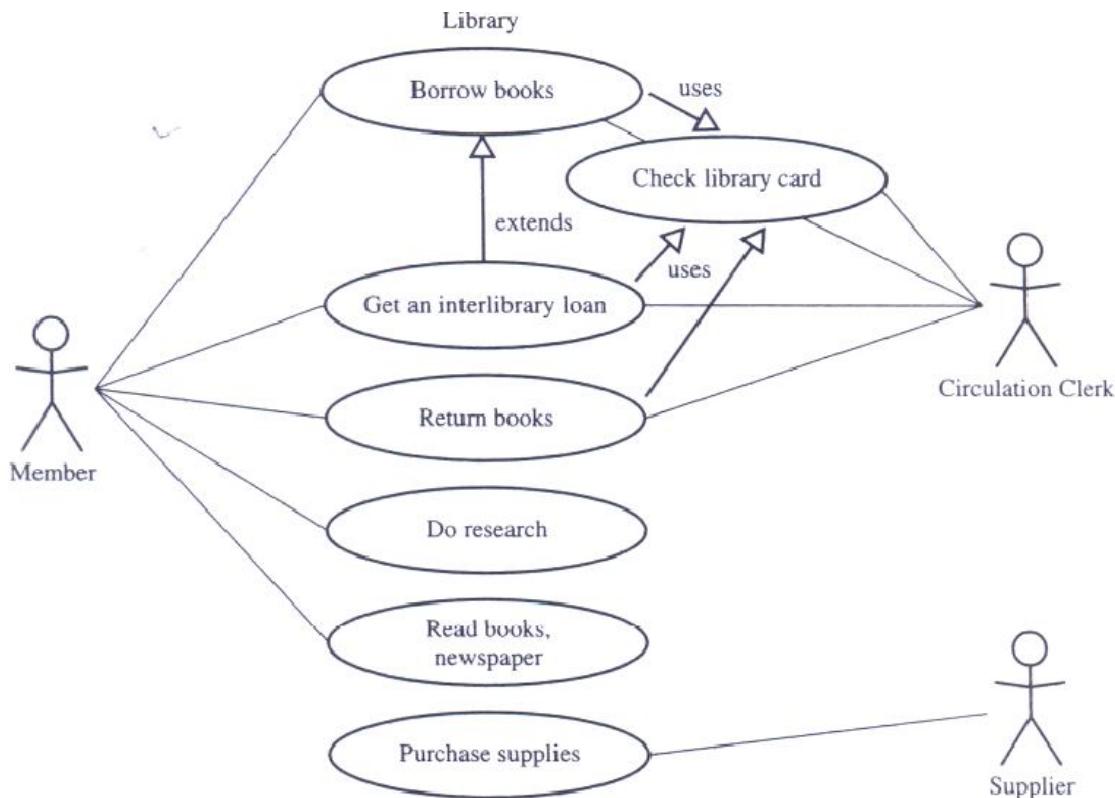
Now let us take a look at the key words of this definition: .Use casi Use case is a special flow of events through the system. By definition, many courses of events are possible and many of these are very similar. It is suggested that, to make a use-case model meaningful, we must group the courses of events and call each group a use-case

class; For example, how you would borrow a book from the library depends on whether the book is located in the library, whether you are the member of the library, and so on.

All these alternatives often are best grouped into one or two use cases, called Borrow books and Get an interlibrary loan (we will look at the relationships of these two use cases in the next section). By grouping the uses cases, we can manage complexities and reduce the number of use cases in a package. .Actors.iAn actor is a user playing a role with respect to the system. When dealing with actors, it is important to think about roles rather than just people and their job titles . For instance, a first-class passenger may play the role of business- class passenger. The actor is the key to finding the correct use cases.

Actors carry out the use cases. A single actor may perform many use cases; furthermore, a use case may have several actors performing it. Actor also can be an external system that needs some information from the current system. Actors can be the ones that get value from the use case, or they can just participate in the use case . .In a system. This simply means that the actors communicate with the system's use case.

A measurable value. A use case must help the actor to perform a task that has some identifiable value; for example, the performance of a use case in terms of price or cost. For example, borrowing books is something of value for a member of the library.



The use-case diagram depicts the extends and uses relationships, where the interlibrary loan is a special case of checking out books. Entering into the system is common to get an interlibrary loan, borrow books, and return books use cases, so it is being "used" by all these use cases.

. Transaction.A transaction is an atomic set of activities that are performed either fully or not at all. A transaction is triggered by a stimulus from an actor to the system or by a point in time being reached in the system. The following are some examples of use cases for the library (see Figure).

Three actors appear in Figure : a member, a circulation clerk, and a supplier.

.Use-case name: Borrow books. A member takes books from the library to read at home, registering them at the checkout desk so the library can keep track of its books. Depending on the member's record, different courses of events will follow.

.Use-case name: Get an interlibrary loan. A member requests a book that the library does not have. The book is located at another library and ordered through an interlibrary loan.

.Use-case name: Return books. A member brings borrowed books back to the library. .

.Use-case name: Check library card. A member submits his or her library card to the clerk, who checks the borrower's record. . Use-case name: Do research. A member comes to the library to do research. The member can search in a variety of ways (such as through books, journals, CDROM, WWW) to find information on the subjects of that research.

. Use-case name: Read books, newspaper. A member comes to the library for a quiet place to study or read a newspaper, journal, or book. . Use-case name: Purchase supplies. The supplier provides the books, journals, and newspapers purchased by the library.

In Figure , the library has an environment with three types of actors (member, circulation clerk, and supplier) and seven use cases (borrow books, return books, get an interlibrary loan, do research, read books or newspaper, and purchase supplies).

Uses and Extends Associations

A use-case description can be difficult to understand if it contains too many alternatives or exceptional flows of events that are performed only if certain conditions are met as the use-case instance is carried out. A way to simplify the description is to take advantage of extends and uses associations. The extends and uses associations often are sources of confusion, so let us take a look at these relationships.

The extends association is used when you have one use case that is similar to another use case but does a bit more or is more specialized; in essence, it is like a subclass.'}:n our example, checking out a book is the basic use case. This is the case that will represent what happens when all goes smoothly. However, many things can affect the flow of events. For example, the book already might be checked out or the library might not have the requested book. Therefore, we cannot always perform the usual behavior associated with the given use case and need to create other use cases to handle the new situations. Of course, one option is to put this variation within the use case. However, the use case quickly would become cluttered with lots of special logic, which would obscure the normal flow .

To remedy this problem, we can use the extends association. Here, you put the base or normal behavior in one use case and the unusual behaviors somewhere else; but instead of cutting and pasting the shared behavior between the base (common) and more specialized use cases, you utilize an extends association to expand the common behavior to fit the special circumstances. Figure1 "extends" Figure 2 to include extends and uses associations.

The uses association occurs when you are describing your use cases and notice that some of them have subflows in common. To avoid describing a subflow more than once in several use cases, you can extract the common subflow and make it a use case of its own. This new use case then can be used by other use cases. The relationships among the other use cases and this new extracted use case is called a uses association.) The uses association helps us avoid redundancy by allowing a use case to be shared. For example, checking a library card is common among the borrow books, return books, and interlibrary loan use cases (see Figure).

The similarity between extends and uses associations is that both can be viewed as a kind of inheritance. When you want to share common sequences in several use cases, utilize the uses association by extracting common sequences into a new, shared use case. The extends association is found when you add a bit more specialized, new use case that extends some of the use cases that you have.

(Use cases could be viewed as concrete or abstract. An abstract use case is not complete and has no initiation actors but is used by a concrete use case, which does interact with actors. This inheritance could be used at several levels. Abstract use cases also are the use cases that have uses or extends associations. All the use cases depicted in Figure are concrete, since they all have initiation actors.

Fowler and Scott provide us excellent(guidelines for addressing variations in use-case modeling :

1. Capture the simple and normal use case first.
2. For every step in that use case, ask .What could go wrong here? . How might this work out differently?
3. Extract common sequences into a new, shared use case with the uses association. If you are adding more specialized or exceptional uses cases, take advantage of use cases you already have with the extends association.

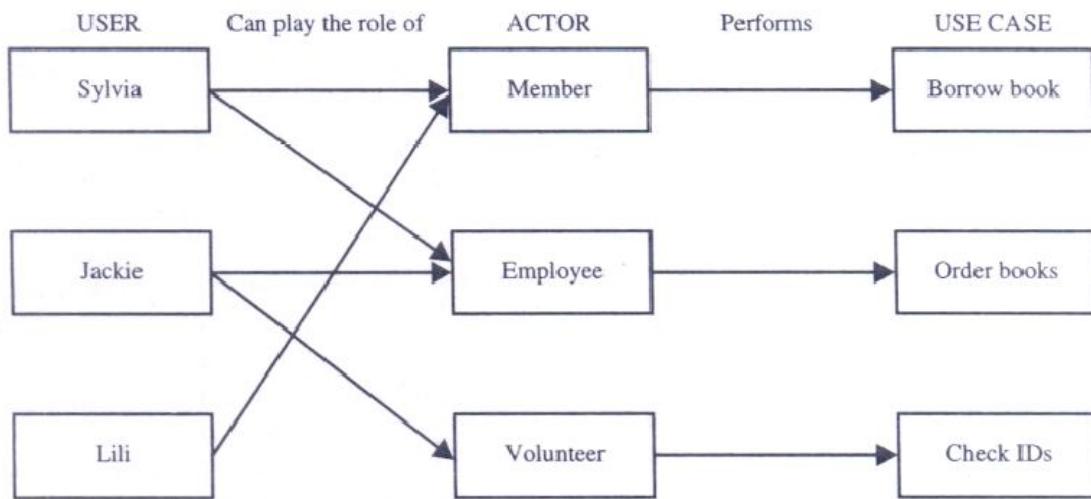
Identifying the Actors

Identifying the actors is (at least) as important as identifying classes, structures, associations, attributes, and behavior. The term actor represents the role a user plays with respect to the system)When dealing with actors, it is important to think about roles rather than people cfrjob titles . A user may play more than one role) For instance, a member of a public library also may play the role of volunteer at the help desk in the library. However, an actor should represent a single user; in the library example, the member can

perform tasks some of which can be done by others and others that are unique. However, try to isolate the roles that the users can play.

You have to identify the actors and understand how they will use and interact with the system. In a thought-provoking book on requirement analysis, Gause and Weinberg , explain what is known as the railroad paradox:

The difference between users and actors.



When trying to find all users, we need to beware of the Railroad Paradox. When railroads were asked to establish new stops on the schedule, they "studied the requirements," by sending someone to the station at the designated time to see if anyone was waiting for a train. Of course, nobody was there because no stop was scheduled, so the railroad turned down the request because there was no demand.

Gause and Weinberg concluded that the railroad paradox appears everywhere there are products and goes like this (which should be avoided):

1. The product is not satisfying the users.
2. Since the product is not satisfactory, potential users will not use it.
3. Potential users ask for a better product.
4. Because the potential users do not use the product, the request is denied.

Therefore, since the product does not meet the needs of some users, they are not identified as potential users of a better product. They are not consulted and the product stays bad . The railroad paradox suggests that a new product actually can create users where none existed before{ Candidates for actors can be found through the answers to the following questions:

Who is using the system? Or, who is affected by the system? Or, which groups need help from the system to perform a task?

.Who affects the system? Or, which user groups are needed by the system to perform its functions? These functions can be both main functions and secondary / functions, such as administration.

.Which external hardware or other systems (if any) use the system to perform tasks?

."What problems does this application solve (that is, for whom)? . And, finally, how do users use the system (use case)? What are they doing with 'the system.

When requirements for new applications are modeled and designed by a group that excludes the targeted users, not only will the application not meet the users' needs, but potential users will feel no involvement in the process and not be committed to giving the application a good try. Always remember Veblen's principle:

'There's no change, no matter how awful, that won't benefit some people; and no change, no matter how good, that won't hurt some." , Another issue worth mentioning is that actors need not be human, although actors are represented as stick figures within a use-case diagram. An actor also can be an external system. For example, an accounting system that needs information from a system to update its accounts is an actor in that system .

Guidelines for Finding Use Cases

When you have defined a set of actors, it is time to describe the way they interact with the system. This should be carried out sequentially, but an iterated approach may be necessary. Here are the steps for finding use cases :

1. For each actor, find the tasks and functions that the actor should be able to perform or that the system needs the actor to perform. The use case should represent a course of events that leads to a clear goal (or, in some cases, several distinct goals that could be alternatives for the actor or for the system).

2. Name the use cases (see Section 6.6.8).

3. Describe the use cases briefly by applying terms with which the user is familiar. This makes the description less ambiguous. Once you have identified the use-cases candidates, it may not be apparent that

all of these use cases need to be described separately; some may be modeled as variants of others. Consider what the actors want to do. It is important to separate actors from users. The actors each represent a role

that one or several users can play. Therefore, it is not necessary to model different actors that can perform the same use case in the same way. The approach should allow different users to be different actors and play one role when performing a particular actor's use case. Thus, each use case has only one main actor. To achieve this, you have to

. Isolate users from actors.

. Isolate actors from other actors (separate the responsibilities of each actor).

.Isolate use cases that have different initiating actors and slightly different behavior (if the actor had been the same, this would be modeled by a use-case alternative behavior) .

While finding use cases, you might have to make changes to your set of actors. All actor changes should be updated in the textual description of actors and use cases. The change should be carried out with care, since changes to the set of actors affect the use cases as well.

When specifying use cases, you might discover that some of them are variants of each other. If so, try to see how you can reuse the use case through extends or uses associations .

How Detailed Must a Use Case Be? When to Stop Decomposing and When to Continue

A use case, as already explained, describes the courses of events that will be carried out by the system. Jacobson et al. believe that, in most cases, too much detail may not be very useful. During analysis of a business system, you can develop one use-case diagram as the system use case and draw packages on this use case to represent the various business domains of the system. For each package, you may create a child usecase diagram (see the case in Section 6.7 for an example). On each child use-case diagram, you can draw all of the use cases of the domain, with actions and interactions. You can further refine the way the use cases are categorized. The extends and uses relationships can be used to eliminate redundant modeling of scenarios.

When should use cases be employed? Use cases are an essential tool in capturing requirements and planning and controlling any software development project. Capturing use cases is a primary task of the analysis phase. Although most use cases are captured at the beginning of the project, you will uncover more as you proceed.

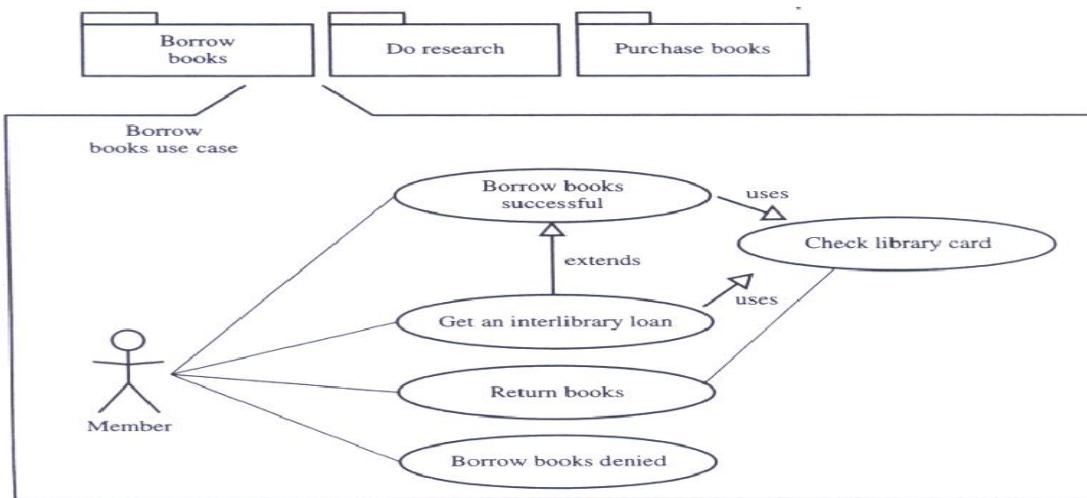
How many use cases do you need? Ivar Jacobson believes that, for a 10-personyear project, he would expect 20 use cases (not counting the uses and extends associations). Other researchers, such as Fowler and Scott, would come up with 100 use cases for a project of the same magnitude. Some prefer smaller grained, more detailed use cases. There is no magic formula; you need to be flexible and work with whatever magnitude you find comfortable . The UML specification recommends that at least one scenario be prepared for each significantly different kind of use case instance. Each scenario shows a different sequence of interactions between actors and the system, with all decisions definite. When you have arrived at the lowest use-case level, which cannot be broken down any further, you may create a sequence diagram and an accompanying collaboration diagram for the use case. With the sequence and collaboration diagrams, you can model the implementation of the scenario .

Dividing Use Casesinto Packages

Each use case represents a particular scenario in the system. You may model either how the system currently works or how you want it to work. Typically, a design is broken down into packages. You must narrow the focus of the scenarios in your system. For example, in a library system, the various scenarios involve a supplier providing books or a member doing research or borrowing books. In this case, there should be three separate packages, one each for Borrow books, Do research, and Purchase books. Many applications may be associated with the library system and one or more databases used to store the information (see Figure).

Naming a Use Case

Use-case names should provide a general description of the use-case function. The name should express what happens when an instance of the use case is performed." Jacobson et al. recommend that the name should be active, often expressed in the form of a verb (Borrow) or verb and noun (Borrow books). The naming should be done with care; the description of the use case should be descriptive and consistent. For example, the use case that describes what happens when a person deposits money into an ATM machine could be named either receive money or deposit money.



A library system can be divided into many packages, each of which encompasses multiple use cases.

11.7 LET US SUM UP

This chapter provides a detailed discussion of use-case driven object-oriented analysis process and how to develop use cases. The main objective of the analysis is to capture a complete, unambiguous, and consistent picture of the requirements of the system. This is accomplished by constructing several models of the system. These models concentrate on describing what the system does rather than how the system does it. Separating the behavior of a system from the way it is implemented requires viewing the system from the perspective of the users rather than that of the machine. Analysis is a creative activity that involves understanding the problem, its associated constraints, and methods of overcoming those constraints.

This is an iterative process that goes on until the problem is well understood. The main objective of object-oriented analysis is to find out *what* the problem is by developing a use-case model, which Jacobson et al. call the "what model." We saw that

use cases are an essential tool in capturing requirements. Capturing use cases is one of the first things to do in coming up with requirements.

Requirements must be traceable across analysis, design, coding, and testing. The unified approach follows Jacobson et al.'s life cycle to produce systems that can be traced across all phases of the developments.

11.8 POINTS FOR DISCUSSION

1. Justify analysis is a difficult activity
2. Critically analyze Usecases under the microscope

11.9 LESSON – END ACTIVITIES

1. Discuss Guidelines for finding use cases.
2. Evaluate to identify the actors.

11.10 REFERENCES

1. Norman,Ronald- object oriented system analysis and design –printice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 12: DEVELOPING EFFECTIVE DOCUMENTATION

CONTENTS

- 12.0 AIM AND OBJECTIVE
- 12.1 INTRODUCTION
- 12.2 DEVELOPING EFFECTIVE DOCUMENTATION
- 12.3 CLASSIFICATIONS THEORY
- 12.4 APPROACHES FOR IDENTIFYING CLASSES
- 12.5 LET US SUM UP
- 12.6 POINTS FOR DISCUSSION
- 12.7 LESSON – END ACTIVITIES
- 12.8 REFERENCES

12.0 AIMS AND OBJECTIVES

You should be able to define and understand
Developing effective documentation.

12.1 INTRODUCTION

An object-oriented environment allows the same set of models to be used for analysis, design, and implementation. The analyst is concerned with the uses of the system, identifying the objects and inheritance, and thinks about the events that change the state of objects. The designer adds detail to this model, perhaps designing screens, user interaction, and database access. The thought process flows so naturally from analyst to designer that it may be difficult to tell where analysis ends and design begins

12.2 DEVELOPING EFFECTIVE DOCUMENTATION

Documenting your project not only provides a valuable reference point and form of communication but often helps reveal issues and gaps in the analysis and design. A document can serve as a communication vehicle among the project's team members, or it can serve as an initial understanding of the requirements. Blum concludes that management has responsibility for resources such as software, hardware, and operational expenses. In many projects, documentation can be an important factor in making a decision about committing resources. Application software is expected to provide a solution to a problem. It is very difficult, if not impossible, to document a poorly understood problem. The main issue in documentation during the analysis phase is to determine what the system must do. Decisions about how the system works are delayed to the design phase. Blum raises the following questions for determining the importance of documentation: How . will a document be used? (If it will not be used, it is not

necessary.) What is the objective of the document? What is the management view of the document? Who are the readers of the document?

Organization Conventions for Documentation

The documentation depends on the organization's rules and regulations. Most organizations have established standards or conventions for developing documentation. However, in many organizations, the standards border on the nonexistent. In other cases, the standards may be excessive. Too little documentation invites disaster; too much documentation, as Blum put it, transfers energy from the problemsolving tasks to a mechanical and unrewarding activity. Each organization determines what is best for it, and you must respond to that definition and refinement . Bell and Evans provide us with guidelines and a template for preparing a document that has been adapted for documenting the unified approach's systems development (see Appendix A). Remember that your modeling effort becomes the analysis, design, and testing documentation. However this template which is based on the unified approach life cycle (see Figure) assists you in organizing and composing your models into an effective documentation.

Guidelines for Developing Effective Documentation

Bell and Evans provide us the following guidelines for making documents fit the needs and expectations of your audience:

Common cover. All documents should share a common cover sheet that identifies the document, the current version, and the individual responsible for the content. As the document proceeds through the life cycle phases, the responsible individual may change. That change must be reflected in the cover sheet .

Figure depicts a cover sheet template. .80-20 rule. As for many applications, the 80-20 rule generally applies for ocumentation 80 percent of the work can be done with 20 percent of the documentation. The trick is to make sure that the 20 percent is easily accessible and the rest (80 percent) is available to those (few) who need to know. .Familiar vocabulary. The formality of a document will depend on how it is used and who will read it. When developing a documentation use a vocabulary that your readers understand and are comfortable with. The main objective here is to communicate with readers and not impress them with buzz words.

Make the document as short as possible. Assume that you are developing a manual. The key in developing an effective manual is to eliminate all repetition; present surpmaries, reviews, organization chapters in less than three pages; and make chapter headings task oriented so that the table of contents also could serveas an index . Organize the document. Use the rules of good organization (such as the organization's standards, college handbooks, Strunk and White's Elements of Styleor the University of Chicago Manual of Style) within each section. Appendix

A provides a template for developing documentation for a project. Most CASE tools provide documentation capability by providing customizable reports. The purpose of these guidelines is to assist you in creating an effective documentation.

A ViaNet bank client can have two types of accounts: a checking account and savings account. For each checking account, one related savings account can exist. Access to the ViaNet bank accounts is provided by a PIN code consisting of four integer digits between 0 and 9. One PIN code allows access to all accounts held by a bank client. No receipts will be provided for any account transactions. The bank application operates for a single banking institution only. Neither a checking nor a savings account can have a negative balance. The system should automatically withdraw money from a related savings account if the requested withdrawal amount on the checking account is more than its current balance. If the balance on a savings account is less than the withdrawal amount requested, the transaction will stop and the bank client will be notified. In this chapter, we identify the actors and use cases of the ViaNet bank ATM system that will be used by subsequent chapters.

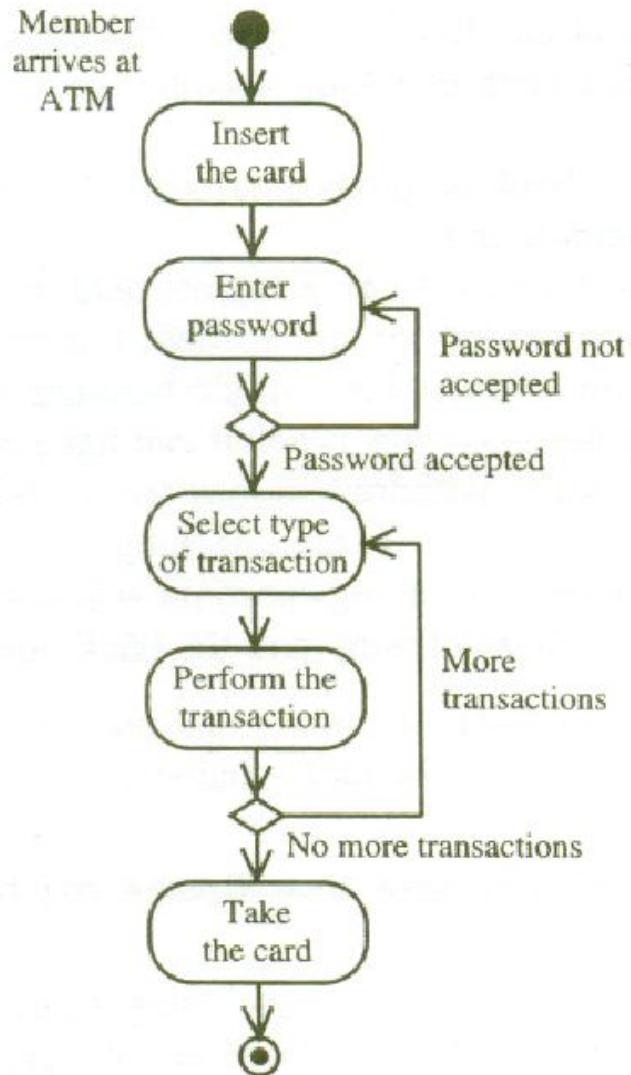
Identifying Actors and Use Cases for the ViaNet Bank ATM System

The bank application will be used by one category of users: bank clients. Notice that identifying the actors of the system is an iterative process and can be modified as you learn more about the system. The actor of the bank system is the bank client. The bank client must be able to deposit an amount to and withdraw an amount from his or her accounts using the bank application. The following scenarios show use-case interactions between the actor (bank client) and the bank. In real life application these use cases are created by system requirements, examination of existing system documentation, interviews, questionnaire, observation, etc.

. Use-case name: Bank ATM transaction. The bank clients interact with the bank system by going through the approval process. After the approval process, the bank client can perform the transaction. Here are the steps in the ATM transaction use case:

1. Insert ATM card.
2. Perform the approval process.
3. Ask type of transaction.
4. Enter type of transaction.
5. Perform transaction.
6. Eject card.
7. Request take card.
8. Take card.

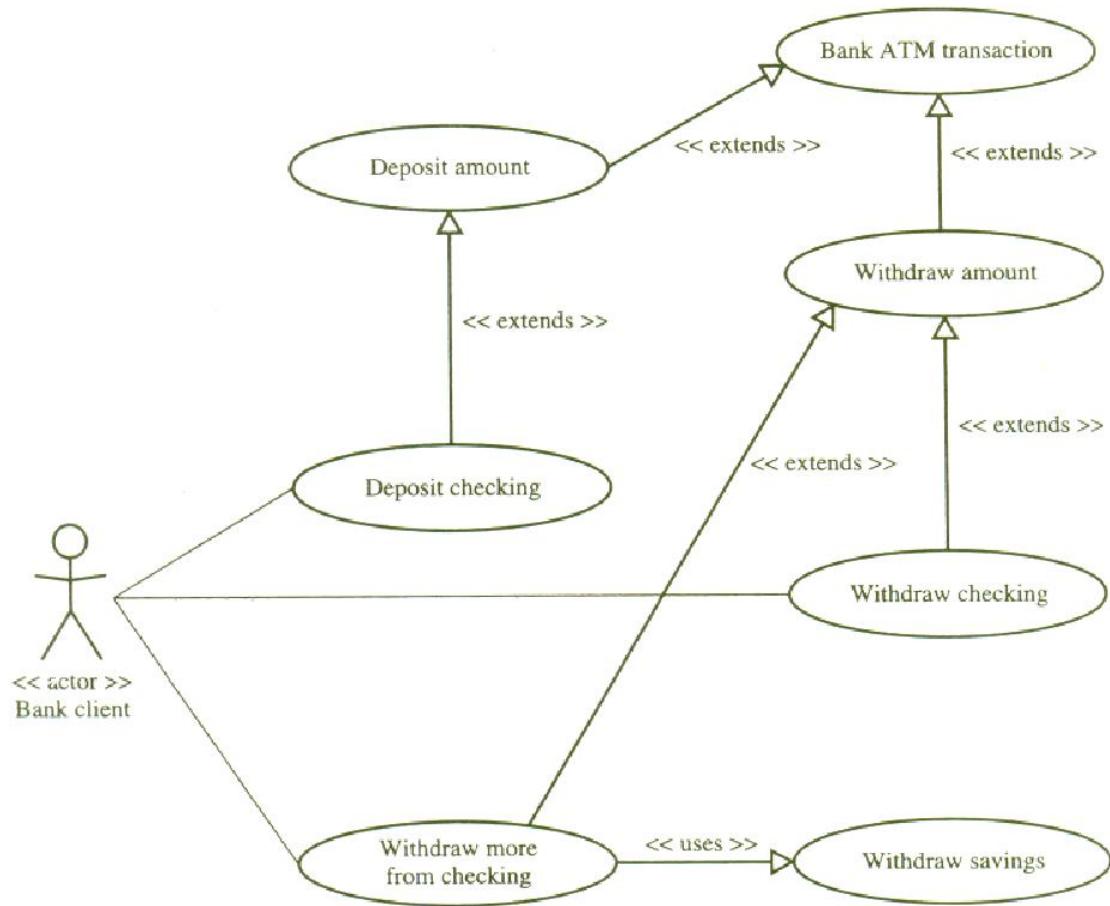
These steps are shown in the Figure activity diagram. Use-case name: Approval process. The client enters a PIN code that consists of digits.



Activities involved in an ATM transaction.

1. Request password.
2. Enter password.
3. Verify password. Use-case name: Invalid PIN. If the PIN code is not valid, an appropriate message is displayed to the client. This use case extends the approval process. (See Figure.) Use-case name: Deposit amount. The bank clients interact with the bank system after the approval process by requesting to deposit money to an account. The client selects the account for which a deposit is going to be made and enters an amount in dollar currency. The system creates a record of the transaction. (See Figure) This use case extends the bank ATM transaction use case. Here are the steps:
 1. Request account type.
 2. Request deposit amount.

3. Enter deposit amount.
4. Put the check or cash in the envelope and insert it into ATM. .Use-case name: Deposit savings. The client selects the savings account for which a deposit is going to be made. All other steps are similar to the deposit amount use case. The system creates a record of the transaction. This use case extends the deposit amount use case. (See Figure 6-11.)

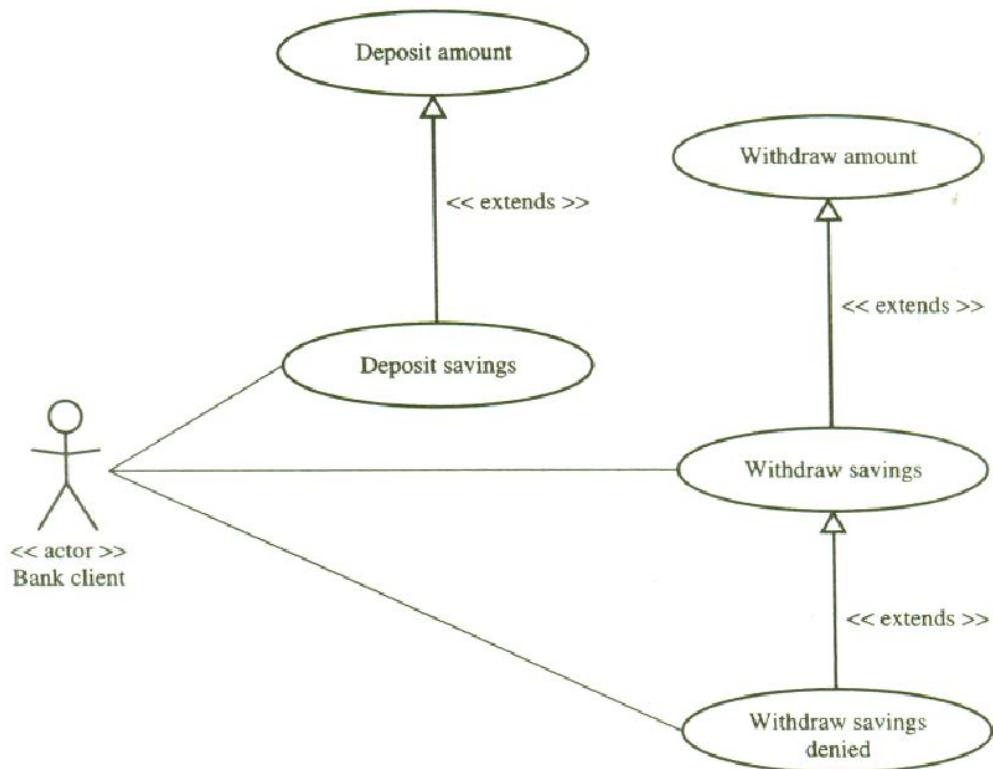


The checking account use-cases.

. Use-casename: Withdrawmorefrom checking. The client tries to withdraw an amount from his or her checking account. If the amount is more than the checking account's balance, the insufficient amount is withdrawn from the related savings account. The system creates a record of the transaction and the withdrawal is successful. This use case extends the withdraw checking use case and uses the withdraw savings use case. (See Figure)

.Use-case name: Withdraw savings. The client tries to withdraw an amount from a savings account. The amount is less than or equal to the balance and the transaction is performed on the savings account. The system creates a record of the transaction since the

withdrawal is successful. This use case extends the withdraw amount use case. (See Figure. Use-case name: Withdraw savings denied. The client withdraws an amount from a savings account. If the amount is more than the balance, the transaction is halted and a message is displayed.



The savings account use-cases package.

This use case extends the bank transaction use case. (See Figure. Use-case name: Savings transaction history. The bank client requests a history of transactions for a savings account. The system displays the transaction history for the savings account. This use case extends the bank transaction use case. (See Figure)

The use-case list contains at least one scenario of each significantly different kind of use-case instance. Each scenario shows a different sequence of interactions between actors and the system, with all decisions definite. If the scenario consists of an if statement, for each condition create one scenario.

Note that the extends association is used when you have a use case that is similar to another use case but does a bit more. In essence, it is a subclass. In the example, the Checking withdraw use case extends the Withdraw amount use case. The Withdraw amount use case represents the case when all goes smoothly. However, many things can affect the flow of events, such as when the withdrawal is for more than the amount of

money in the checking account. Withdraw more from checking is the use case that extends the Checking withdraw. You can put this variation within the Checking withdraw use case, too. However, this would clutter the use case with lots of special logic, which would obscure the normal flow. To review, the uses association occurs when a behavior is common to more than one use case and you want to avoid copying the description of that behavior. The Approval process is such a use case that is used by Bank transaction use case. As you can see, use cases are an essential tool for identifying requirements. Developing use cases is an iterative process. Although most use cases are generated at this phase of system development, you will uncover more as you proceed. Fowler and Scott advise us to keep an eye out for them at all times. Every use case represents a potential requirement.

The ViaNet Bank ATM Systems' Packages

Each use case represents a particular scenario in the system. As explained earlier, it is better to break down the use cases into packages. Narrow the focus of the scenarios in the system. In the bank system, the various scenarios involve checking account, savings account, and general bank transactions. (See Figure .)

Remember, use case is a method for capturing the way a system or business works. Use cases are used to model the scenarios. The scenarios are described textually or through a sequence of steps. Modeling with use cases is a recommended tool in finding the objects of a system. In the next chapter, we look at identifying classes based on the use cases identified here.

- Amount
- Approval Process
- ATM Card
- ATM Machine
- Bank
- Bank Client
- Card
- Cash
- Check
- Checking
- Checking Account
- Client
- Client's Account
- Currency
- Dollar
- Envelope
- Four Digits
- Fund
- Invalid PIN
- Message

Money
 Password
~~PIN .~~
~~PIN Code~~
 Record
~~Savings~~
~~Savings Account~~
~~Step.~~
~~System~~
~~Transaction~~
~~Transaction History~~

It is safe to eliminate the irrelevant classes. The candidate classes must be selected from relevant and fuzzy classes. The following irrelevant classes can be eliminated because they do not belong to the problem statement: Envelope, Four Digits, and Step. Strikeouts indicate eliminated classes.

Account
 Account Balance
 Amount
 Approval Process
 ATM Card
 ATM Machine
~~Bank .~~
 BankClient
 Card
 Cash
 Check
 Checking
 Checking Account
 Client
 Client's Account
 Currency
 Dollar
 EIWelape
 FeMeDigits
 Fund
 Invalid PIN
 Message
 Money,
 Password
 PIN
 PIN Code
 Record
 Savings
 Savings Account
 System
 Transaction -Transaction History

12.3 CLASSIFICATIONS THEORY

Classification, the process of checking to see if an object belongs to a category or a class, is regarded as a basic attribute of human nature.

Booch explains that, intelligent classification is part of all good science. Classification guides us in making decisions about modularization. We may choose to place certain classes and objects together in the same module or in different modules, depending upon the sameness we find among these declarations; coupling and cohesion are simply measures of this sameness.

Classification also plays a role in allocating processes to procedures. We place certain processes together in the same processor or different processors, depending upon packaging, performance, or reliability concerns.

Human beings classify information every instant of their waking lives. We recognize the objects around us, and we move and act in relation to them. A human being is sophisticated information system, partly because he or she possesses a superior classification capability . For example, when you see a new model of a car, you have no trouble identifying it as a car. What has occurred here, even though you may never have seen this particular car before, is that you not only can immediately identify it as a car, but you also can guess the manufacturer and model. Clearly, you have some general idea of what cars look like, sound like, do, and are good for-you have a notion of car-kind or, in object-oriented terms, the class car.

Classes are an important mechanism for classifying objects. The chief role of a class is to define the attributes, methods, and applicability of its instances. The class car, for example, defines the property color. Each individual car (formally, each instance of the class car) will have a value for this property, such as maroon, yellow, or white.

It is fairly natural to partition the world into objects that have properties (attributes)and methods (behaviors). It is common and useful partitioning or classification, but we also routinely divide the world along a second dimension: We distinguish classes from instances. A class is a specification of structure, behavior, and the description of an object. Classification is concerned more with identifying the class of an object than the individual objects within a system. Martin and Odell explain that classes are important because they create conceptual building blocks for designing systems:

In object-oriented programming, these building blocks guide the designer in defining the classes and their data structures. In addition, object types (classes) provide an index for system process. For instance, operations such as Hire, Promote, Retire, and Fire are intimately tied to the object type (class) Employee, because they change the state of an employee. In other words, an object should only be manipulated via the operations associated with its type. Without object types (classes), then, operations cannot be defined

properly. Tou and Gonzalez describe the recognition of concrete patterns or classes by humans as a psychophysiological problem that involves a relationship between a person and a physical stimulus .

When you perceive a real-world object, you make an inductive inference and associate this perception with some general concepts or clues that you have derived from your past experience. Human recognition, in reality, is a question of estimating the relative odds that the input data can be associated with a class from a set of known classes, which depend on our past experiences and clues for recognition. Intelligent classification is intellectually hard work and may seem rather arbitrary. That is how our minds work . Martin and Odell have observed in object-oriented analysis and design that, "In fact, an object can be categorized in more than one way. For example, in Figure one person may regard the object Betty as a Woman. Her boss regards her as an Employee. The person who mows her lawn classifies her as an Employer. The local animal control agency licenses her as a Pet Owner. The credit bureau reports that Betty is an instance of the object types called Good Credit Risk-and so on."

The problem of classification may be regarded as one of discriminating things, not between the individual objects but between classes, via the search for features or invariant attributes or behaviors among members of a class. Classification can be defined as the categorization of input data (things) into identifiable classes via the extraction of significant features of attributes of the data from a background of irrelevant detail. Another issue in relationships among classes is studied.

12.4 APPROACHES FOR IDENTIFYING CLASSES

In the following sections, we look at four alternative approaches for identifying classes: the noun phrase approach; the common class patterns approach; the usecase driven, sequence/collaboration modeling approach; and the Classes, Responsibilities, and Collaborators (CRC) approach.

The first two approaches have been included to increase your understanding of the subject; the unified approach uses the use-case driven approach for identifying classes and understanding the behavior of objects. However, you always can combine these approaches to identify classes for a given problem.

Another approach that can be used for identifying classes is Classes, Responsibilities, and Collaborators (CRC) developed by Cunningham, Wilkerson, and Beck. Classes, Responsibilities, and Collaborators, more technique than method, is used for identifying classes responsibilities and therefore their attributes and methods.

12.5 LET US SUM UP

The key in developing effective documentation is to eliminate all repetition; present summaries, reviews, organization chapters in less than three pages; and make chapter headings task oriented so that the table of contents also could serve as an index.

Object analysis is a process by which we can identify the classes that play a role in achieving the system goals and requirements. The problem of classification may be regarded as one of discriminating *things*, not between the individual objects, but between classes via the search for features or invariant attributes (or behaviors) among members of a class. Finding classes is one of the hardest activities in object-oriented analysis.

12.6 POINTS FOR DISCUSSION

1. Establish organizing conventions for documentation
2. Critically analyze identifying actors and use cases for that via net bank ATM System

12.7 LESSON – END ACTIVITIES

1. Discuss about classification theory.

12.8 REFERENCES

1. Norman,Ronald- object oriented system analysis and design –prinice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 13: NOUN PHRASE APPROACH

CONTENTS

- 13.0 AIMS AND OBJECTIVES
- 13.1 INTRODUCTION
- 13.2 NOUN PHRASE APPROACH
- 13.3 IDENTIFYING TENTATIVE CLASSES
- 13.4 LET US SUM UP
- 13.5 POINTS FOR DISCUSSION
- 13.6 LESSON – END ACTIVITIES
- 13.7 REFERENCES

13.0 AIMS AND OBJECTIVES

You should be able to identify classes and verbs.

13.1 INTRODUCTION

Object-oriented analysis is a process by which we can identify classes that play a role in achieving system goals and requirements. Unfortunately, classes rarely just are there for the picking. Identification of classes is the hardest part of object-oriented analysis and design. Booch argues that, "There is no such a thing as the perfect class structure, nor the right set of objects. As in any engineering discipline, our design choice is compromised by many competing factors." Gabriel, White, and Bobrow, the three designers of the object-oriented language CLOS, respond to the issue of how to identify classes: "That's a fundamental question for which there is no easy answer. I try things". In this chapter, we look at four approaches for identifying classes and their behaviors in the problem domain. Discovering classes is one of the hardest activities in object-oriented analysis. However, the process is incremental and iterative and, furthermore, as you gain experience it will become easier to identify the classes.

13.2 NOUN PHRASE APPROACH

The noun phrase approach was proposed by Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. In this method, you read through the requirements or use cases looking for noun phrases. Nouns in the textual description are considered to be classes and verbs to be methods of the classes (identifying methods will be covered in later chapter). All plurals are changed to singular, the nouns are listed, and the list divided into three categories (see Figure): relevant classes, fuzzy classes (the "fuzzy area," classes we are not sure about), and irrelevant classes. It is safe to scrap the

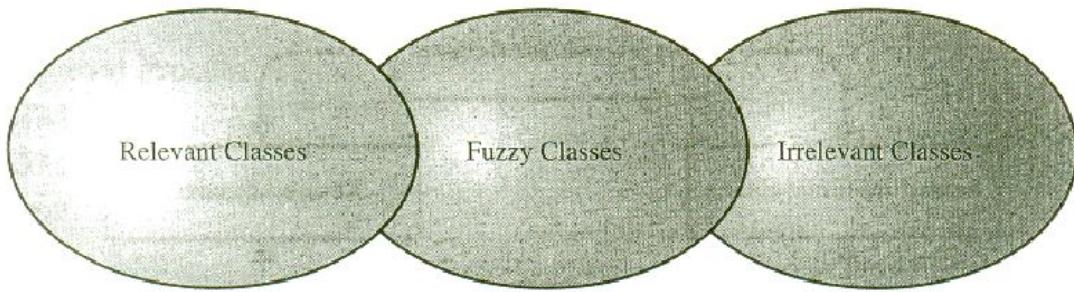
irrelevant classes, which either have no purpose or will be unnecessary. Candidate classes then are selected from the other two categories.

Keep in mind that identifying classes and developing a UML class diagram just like other activities is an iterative process. Depending on whether such object modeling is for the analysis or design phase of development, some classes may need to be added or removed from the model and, remember, flexibility is a virtue. You must be able to formulate a statement of purpose for each candidate class; if not, simply eliminate it.

13.3 IDENTIFYING TENTATIVE CLASSES

The following are guidelines for selecting classes in an application: .Look for nouns and noun phrases in the use cases. . Some classes are implicit or taken from general knowledge. . All classes must make sense in the application domain; avoid computer implementation classes-defer them to the design stage. . Carefully choose and define class names.

As explained before, finding classes is not easy. The more practice you have, the better you get at identifying classes. Finding classes is an incremental and iterative process. Booch explains this point elegantly: "Intelligent classification is intellectually hard work, and it best comes about through an incremental and iterative process. This incremental and iterative nature is evident in the development of such diverse software technologies as graphical user interfaces,



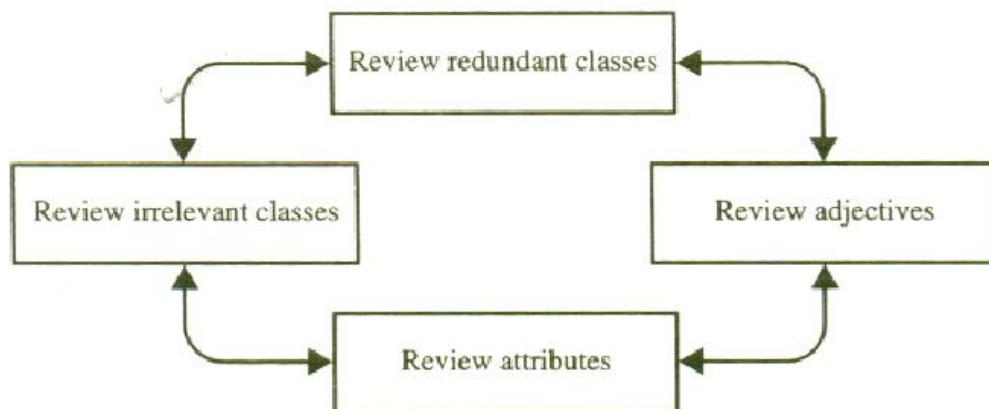
Using the noun phrase strategy, candidate classes can be divided into three categories: Relevant Classes, Fuzzy Area or Fuzzy Classes (those classes that we are not sure about), and Irrelevant Classes. database standards, and even fourth-generation languages." As Shaw observed in software engineering, the development of individual abstractions often follows a common pattern. First the problems are solved ad hoc. As experience accumulates, some solutions turn out to work better than others, and a sort of folklore is passed informally from person to person. Eventually, the useful solutions are understood more systematically, and they are codified and analyzed. This enables the development of models that support automatic implementation and theories that allow the generalization of the solution. This in turn enables a more sophisticated level of practice and allows us to tackle harder problems-which we often approach ad hoc, starting the cycle over again.

Selecting Classes from the Relevant and Fuzzy Categories

The following guidelines help in selecting candidate classes from the relevant and fuzzy categories of classes in the problem domain. .Redundant classes. Do not keep two classes that express the same information.

If more than one word is being used to describe the same idea, select the one that is the most meaningful in the context of the system. This is part of building a common vocabulary for the system as a whole . Choose your vocabulary carefully; use the word that is being used by the user of the system. .Adjectives classes. Wirls-Brock, Wilkerson, and Wiener warn us about adjectives:

"Be wary of the use of adjectives. Adjectives can be used in many ways. An adjective can suggest a different kind of object, different use of the same object, or it could be utterly irrelevant. Does the object represented by the noun behave differently when the adjective is applied to it? If the use of the adjective signals that the behavior of the object is different, then make a new class" . For example, Adult Members behave differently than Youth Members, so ,the two should be classified as different classes. .Attribute classes. Tentative objects that are used only as values should be defined or restated as attributes and not as a class. For example, Client Status and Demographic of Client are not classes but attributes of the Client class. .Irrelevant classes. Each class must have a purpose and every class should be clearly defined and necessary. You must formulate a statement of purpose for each candidate class. If you cannot come up with a statement of purpose, simply eliminate the candidate class.



The process of eliminating the redundant classes and refining the remaining classes is not sequential.

You can move back and forth among these steps as often as you like. Remember that this is an incremental process. Some classes will be missing, others will be eliminated or refined later. Unless you are starting with a lot of domain knowledge, you probably are missing more classes than you will eliminate. Although some classes ultimately may become superclasses, at this stage simply identify them as individual,

specific classes. Your design will go through many stages on its way to completion, and you will have adequate opportunity to revise it .

Like any other activity of software development, the process of identifying relevant classes and eliminating irrelevant classes is an incremental process. Each iteration often uncovers some classes that have been overlooked. The repetition of the entire process, combined with what you already have learned and the reworking of your candidate classes will enable you to gain a better understanding of the system and the classes that make up your application. Classification is the essence of good object-oriented analysis and design. You must continue this refining cycle through the development process until you are satisfied with the results. Remember that this process (of eliminating redundant classes, classes containing adjectives, possible attributes, and irrelevant classes) is not sequential. You can move back and forth among these steps as often as you like (see Figure)

The ViaNet Bank ATM System: Identifying Classes by Using Noun Phrase Approach

To better understand the noun phrase method, we will go through a case and apply the noun phrase strategy for identifying the classes. We must start by reading the use cases and applying the principles discussed in this chapter for identifying classes.

Initial List of Noun Phrases: Candidate Classes

The initial study of the use cases of the bank system produces the following noun phrases (candidate classes-maybe).

Account

Account Balance

Amount

Approval Process

ATM Card

ATM Machine

Bank

Bank Client

Card

Cash

Check

Checking

Checking Account

Client

Client's Account

Currency
 Dollar
 Envelope
 Four Digits
 Fund
 Invalid PIN
 Message
 Money
 Password
 PIN
 PIN Code
 Record
 Savings
 Savings Account
 Step.
 System
 Transaction
 Transaction History

It is safe to eliminate the irrelevant classes. The candidate classes must be selected from relevant and fuzzy classes. The following irrelevant classes can be eliminated because they do not belong to the problem statement: Envelope, Four Digits, and Step. Strikeouts indicate eliminated classes.

Account
 Account Balance
 Amount
 Approval Process
 ATM Card
 ATM Machine
 Bank .
 BankClient
 Card
 Cash
 Check
 Checking
 Checking Account
 Client
 Client's Account
 Currency
 Dollar
 Envelope
 Four Digits
 Fund
 Invalid PIN
 Message
 Money,

Password
 PIN
 PIN Code
 Record
 Savings
 Savings Account
 System
 Transaction -
 Transaction History

Reviewing the Redundant Classes and Building a Common Vocabulary

We need to review the candidate list to see which classes are redundant. If different words are being used to describe the same idea, we must select the one that is the most meaningful in the context of the system and eliminate the others. The following are the different class names that are being used to refer to the same concept:

Client, BankClient Account, Client's Account PIN, PIN Code Checking,

Checking Account = BankClient (the term chosen)

Checking Account = Account

Checking Account = PIN

Checking Account = Checking Account

Savings, Savings Account = Savings Account

Fund, Money = Fund

ATM Card, Card = ATM Card

Here is the revised list of candidate classes:

Account

Account Balance

Amount

Approval Process

ATM Card

Bank

BankClient

6affi

Cash

Check

Checking Account

Currency

Dollar

Fund digits

Fund

Invalid PIN

Message

Message

Password

PIN

PIN Case

Record
 Savings
 Savings Account
 System
 Transaction
 Transaction History

Reviewing the Classes Containing Adjectives

Veagain review the remaining list, now with an eye on classes with adjectives. The lain question is this: Does the object represented by the noun behave differently when the adjective is applied to it? If an adjective suggests a different kind of class or the class represented by the noun behaves differently when the adjective is applied to it, then we need to make a new class. However(it is a different use of the same object or the class is irrelevant, we must eliminate it) In this example, we have no classes containing adjectives that we can eliminate.

Reviewing the Possible Attributes

The next review focuses on identifying the noun phrases that are attributes, not classes. The noun phrases used only as values should be restated as attributes. This process also will help us identify the attributes of the classes in the system. Balance: An attribute of the Account class.Invalid PIN: It is only a value, not a class. Password: An attribute, possibly of the BankClient class.Transaction History: An attribute, possibly of the Transaction class. ..PIN: An attribute, possibly of the BankClientclass. Here is the revised list of candidate classes. Notice that the eliminated classes are strikeouts (they have a line through them).

Account
 Approval Process
 ATM Card
 Bank
 BankClient
 Cash
 Check
 CheelflRg
 Checking Account
 Currency
 Dollar
 Fund
 Message
 MaBey
 PIN
 Record
 Savings Account
 System
 Transaction

Reviewing the Class Purpose

Identifying the classes that play a role in achieving system goals and requirements is a major activity of object-oriented analysis) Each class must have a purpose. Every class should be clearly defined and necessary in the context of achieving the system's goals. If you cannot formulate a statement of purpose for a class, simply eliminate it. The classes that add no purpose to the system have been deleted from the list. The candidate classes are these:

- /1\TM Machine class: Provides an interface to the ViaNet bank.

ATMCard class: Provides a client with a key to an account. BankClient class: A client is an individual that has a checking account and, possibly, a savings account.

Bank class: Bank clients belong to the Bank. It is a repository of accounts and processes the accounts' transactions.

Account class: An Account class is a formal (or abstract) class, it defines the common behaviors that can be inherited by more specific classes such as CheckingAccount and SavingsAccount. CheckingAccount class: It models a client's checking account and provides more specialized withdrawal service.

.savingsAccount class: It models a client's savings account. Transaction class: Keeps track of transaction, time, date, type, amount, and 'balance'.

No doubt, some classes are missing from the list and others will be eliminated or refined later. Unless you are starting with a lot of domain knowledge, you probably will miss more classes than you will eliminate. After all, this is an incremental process; as you learn more about the problem, your design will go through many stages on its way to completion. Remember, there is no such thing as the "right" set of classes. However, the process of identifying classes can improve gradually through this incremental process.

The major problem with the noun phrase approach is that it depends on the completeness and correctness of the available document, which is rare in real life. On the other hand, large volumes of text on system documentation might lead to too many candidate classes. Even so, the noun phrase exercise can be very educational and useful if combined with other approaches, especially with use cases as we did here.

13.4 LET US SUM UP

In this chapter, we studied four approaches for identifying classes: the noun phrase, common class patterns, use-case driven, and Classes, Responsibilities, and Collaborators. The process of identifying classes can improve gradually through the incremental process. Some classes will be missing in the first few cycles of identification, and others will be eliminated or refined later. Unless you are starting with a lot of domain knowledge, you probably will miss more classes than you will eliminate. Your design will go through many stages on its way to completion as you learn more about the problem. To identify classes using the noun phrase approach, read through use cases, looking for noun phrases. Consider nouns in the textual description to be classes and verbs to be methods of the classes.

13.5 POINTS FOR DISCUSSION

1. Establish Noun Phrase Approach
2. Analyze to select classes from the relevant and fuzzy categories

13.6 LESSON – END ACTIVITIES

1. Discuss about identifying classes using Noun-Phrase approach.
2. Validate candidate class

13.7 REFERENCES

1. Norman,Ronald- object oriented system analysis and design –prinice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 14: USE CASE DRIVEN APPROACH

CONTENTS

- 14.0 AIMS AND OBJECTIVES
- 14.1 INTRODUCTION
- 14.2 USE-CASE DRIVEN APPROACH: IDENTIFYING CLASSES AND THEIR BEHAVIORS THROUGH SEQUENCE/COLLABORATION MODELING
- 14.3 IMPLEMENTATION OF SCENARIOS
- 14.4 THE VIANET BANK ATM SYSTEM: DECOMPOSING
- 14.5 CLASSES, RESPONSIBILITIES, AND COLLABORATORS
- 14.6 CLASSES, RESPONSIBILITIES, AND COLLABORATORS PROCESS
- 14.7 ASSOCIATIONS
- 14.8 GUIDELINES FOR IDENTIFYING ASSOCIATION
- 14.9 LET US SUM UP
- 14.10 POINTS FOR DISCUSSION
- 14.11 LESSON – END ACTIVITIES
- 14.12 REFERENCES

14.0 AIMS AND OBJECTIVES

You should be able to learn the scenarios.

14.1 INTRODUCTION

In an object-oriented environment, objects take on an active role in a system. Of course, objects do not exist in isolation but interact with each other. Indeed, these / interactions and relationships are the application.(All objects stand in relationship to others on whom they rely for services and control) The relationship among objects is based on the assumptions each makes about the other objects, including what operations can be performed and what behavior results three types of relationships among objects are Association. How are objects associated? This information will guide us in designing classes.

Super-sub structure (also known as generalization hierarchy). How are objects organized into super classes and subclasses? This information provides us the direction of inheritance.

.Aggregation and a-part-of structure. What is the composition of complex classes?

This information guides us in defining mechanisms that properly manage object within-object .

Generally speaking, the relationships among objects are known as associations. For example, a customer places an order for soup. The order is the association between the customer and soup objects. The hierarchical or super-sub relation allows the sharing of properties or inheritance. A-part-of structure is a familiar means of organizing components of a bigger object. For example, walls, windows, doors, and the like are part of a bigger object: a building.

In this chapter, we look at guidelines for identifying association, super-sub, and a-part-of relationships in the problem domain. We then proceed to identify attributes and methods. To do this we must first determine the responsibilities of the system. We saw that the system's responsibilities can be identified by analyzing use cases and their sequence and collaboration diagrams. Once you have identified the system's responsibilities and what information the system needs to remember, you can assign each responsibility to the class to which it logically belongs. This also aids in determining the purpose and role each class plays in the application.

14.2 USE-CASE DRIVEN APPROACH: IDENTIFYING CLASSES AND THEIR BEHAVIORS THROUGH SEQUENCE/COLLABORATION MODELING

The use-case driven approach is the third approach that we examine in this chapter and the one that is recommended. From the previous chapter, we learned that use cases are employed to model the scenarios in the system and specify what external actors interact with the scenarios. The scenarios are described in text or through a sequence of steps. Use-case modeling is considered a problem-driven approach to object-oriented analysis, in that the designer first considers the problem at hand and not the relationship between objects, as in a data-driven approach.

Modeling with use cases is a recommended aid in finding the objects of a system and is the technique used by the unified approach. Once the system has been described in terms of its scenarios, the modeler can examine the textual description or steps of each scenario to determine what objects are needed for the scenario to occur. However, this is not a magical process in which you start with use cases, develop a sequence diagram, and voila, classes appear before your eyes. The process of creating sequence or collaboration diagrams is a systematic way to think about how a use case (scenario) can take place; and by doing so, it forces you to think about objects involved in your application.

When building a new system, designers model the scenarios of the way the system of business should work. When redesigning an existing system, many modelers choose to first model the scenarios of the current system, and then model the scenarios of the way the system should work. Developing scenarios also requires us to think about class methods, which will be studied in later chapters.

14.3 IMPLEMENTATION OF SCENARIOS

The UML specification recommends that at least one scenario be prepared for each significantly different use-case instance. Each scenario shows a different sequence of interaction between actors and the system, with all decisions definite. In essence, this process helps us to understand the behavior of the system's objects. When you have arrived at the lowest use-case level, you may create a child sequence diagram or accompanying collaboration diagram for the use case. With the sequence and collaboration diagrams, you can model the implementation of the scenario .

Like use-case diagrams, sequence diagrams are used to model scenarios in the systems. Whereas use cases and the steps or textual descriptions that define them offer a high-level view of a system, the sequence diagram enables you to model a more specific analysis and also assists in the design of the system by modeling the interactions between objects in the system.

As explained in a sequence diagram, the objects involved are drawn on the diagram as a vertical dashed line, with the name of the objects at the top. Horizontal lines corresponding to the events that occur between objects are drawn between the vertical object lines. The event lines are drawn in sequential order, from the top of the diagram to the bottom. They do not necessarily correspond to the steps defined for a use-case scenario.

14.4 THE VIANET BANK ATM SYSTEM: DECOMPOSING

Scenario with a Sequence Diagram: Object Behavior Analysis A sequence diagram represents the sequence and interactions of a given use case or scenario. Sequence diagrams are among the most popular UML diagrams and, if used with an object model or class diagram, can capture most of the information about a system . Most object-to-object interactions and operations are considered events, and events include signals, inputs, decisions, interrupts, transitions, and actions to or from users or external devices. An event also is considered to be any action by an object that sends information. The event line represents a message sent from one object to another, in which the "from" object is requesting an operation be performed by the "to" object. The "to" object performs the operation using a method that its class contains. Developing sequence or collaboration diagrams requires us to think about objects that generate these events and therefore will help us in identifying classes.

To identify objects of a system, we further analyze the lowest level use cases with a sequence and collaboration diagram pair (actually, most CASE tools such as SA/Object allow you to create only one, either a sequence or a collaboration diagram, and the system generates the other one). Sequence and collaboration diagrams represent the order in which things occur and how the objects in the system send messages to one another. These diagrams provide a macro-level analysis of the dynamics of a system. Once you start creating these diagrams, you may find that objects may need to be added to satisfy the particular sequence of events for the given use case.

You can draw sequence diagrams to model each scenario that exists when a BankClient withdraws, deposits, or needs information on an account. By walking through the steps, you can determine what objects are necessary for those steps to take place. Therefore, the process of creating sequence or collaboration diagrams can assist you in identifying classes or objects of the system. This approach can be combined with noun phrase and class categorization for the best results. We identified the use cases for the bank system. The following are the low level (executable) use cases:

Deposit Checking

Deposit Savings

Invalid PIN

Withdraw Checking

Withdraw More from Checking

Withdraw Savings

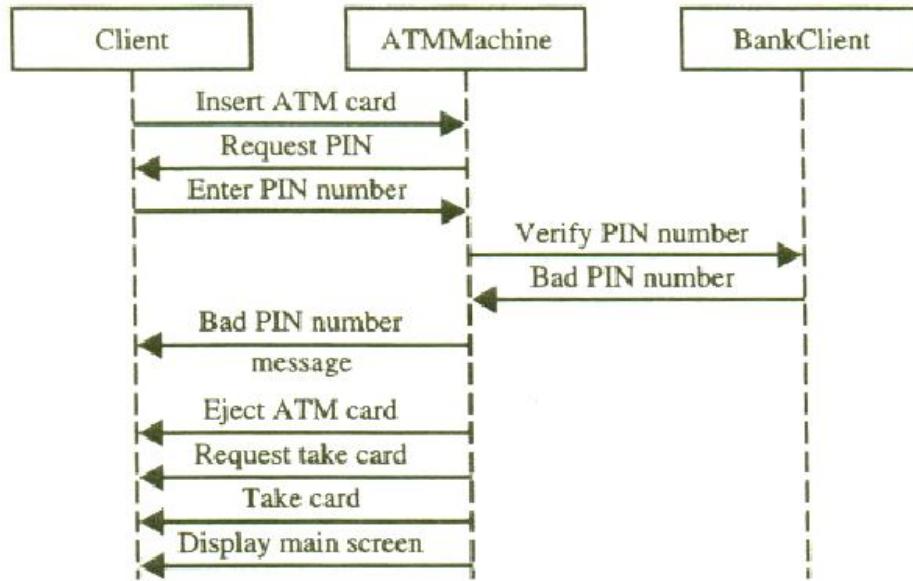
Withdraw Savings Denied

Checking Transaction History

Savings Transaction History

Let us create a sequence/collaboration diagram for the following use cases: .Invalid PIN use case .Withdraw Checking use case .Withdraw More from Checking use case Sequence/collaboration diagrams are associated with a use case. For example, to model the sequence/collaboration diagrams in SA/Object, you must first select a use case, such as the Invalid PIN use case, then associate a sequence or collaboration child process. To create a sequence you must think about the classes that probably will be involved in a use-case scenario. Keep in mind that use case refers to a process, not a class. However, a use case can contain many classes, and the same class can occur in many different use cases. Point of caution: you should defer the interfaces classes to the design phase and concentrate on the identifying business classes here. Consider how we would prepare a sequence diagram for the Invalid PIN use case. Here, we need to think about the sequence of activities that the actor BankClient performs: . Insert ATM Card. .Enter PIN number. . Remove the ATM Card. Based on these activities, the system should either grant the access right to the account or reject the card. Next, we need to more explicitly define the system. With what are we interacting? We are interacting with an ATMMachine and the BankClient. So, the other objects of this use case are ATMMachine and BankClient. Now that we have identified the objects involved in the use case, we need to list them in a line along the top of a page and drop dotted lines beneath each object (see Figure). The client in this case is whoever tries to access an account

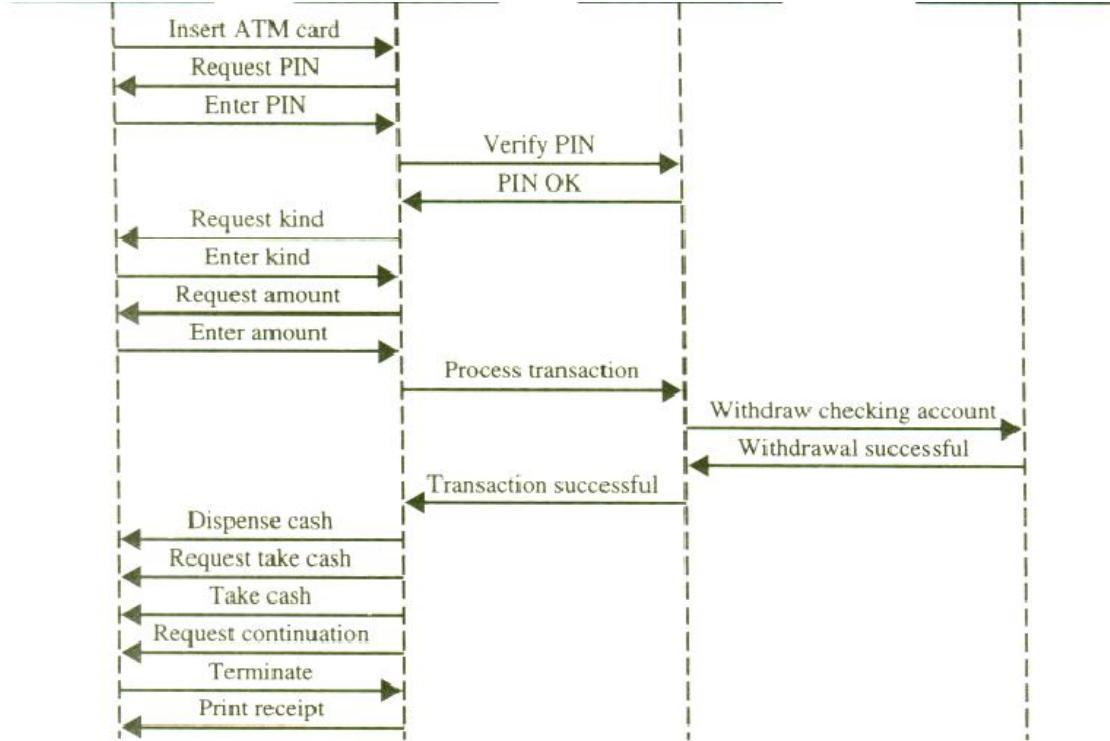
The sequence diagram for the Invalid PIN use case.



through the ATM, and mayor may not have an account. The BankClient on the other hand has an account.

The dotted lines are the lifelines. The line on the right represents an actor, in this case the BankClient, or an event that is outside the system boundary. Recall from previous chapter that an event arrow connect objects. In effect, the event arrow suggests that a message is moving between those two objects. An example of an event message is the request for a PIN. An event line can pass over an object without stopping at that object. Each event must have a descriptive name. In some cases, several objects are active simultaneously, even if they are only waiting for another object to return information to them. In other cases, an object becomes active when it receives a message and then becomes inactive as soon as it responds. Similarly, we can develop sequence diagrams for other use cases (as in Figures). Collaboration diagrams are just another view of the sequence diagrams and therefore can be created automatically; most UML modeling tools automatically create them (see Figures)

The following classes have been identified by modeling the UML sequence/collaboration diagrams: Bank, BankClient, ATMMachine, Account, Checking Account, and Savings Account. Similarly other classes can be identified by developing the remaining sequence/ collaboration diagrams. Developing the other sequence/collaboration diagrams has been left as an exercise; see problem



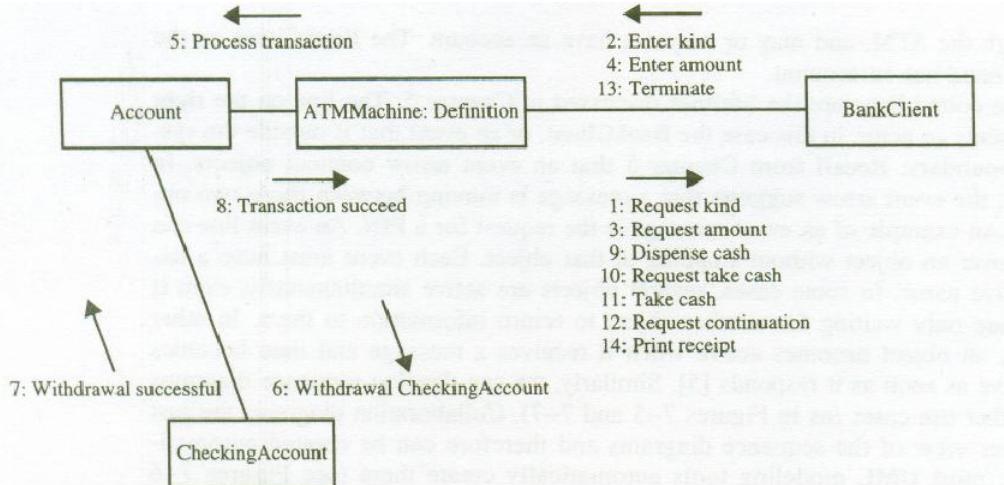
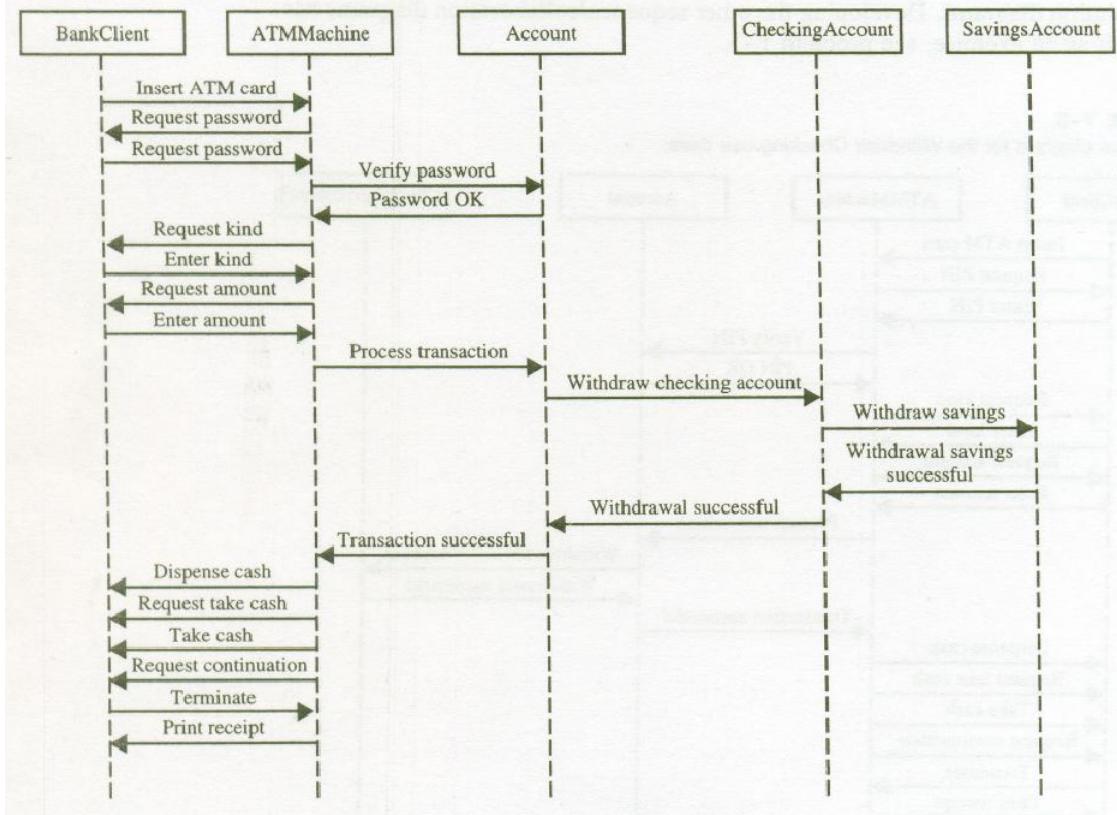
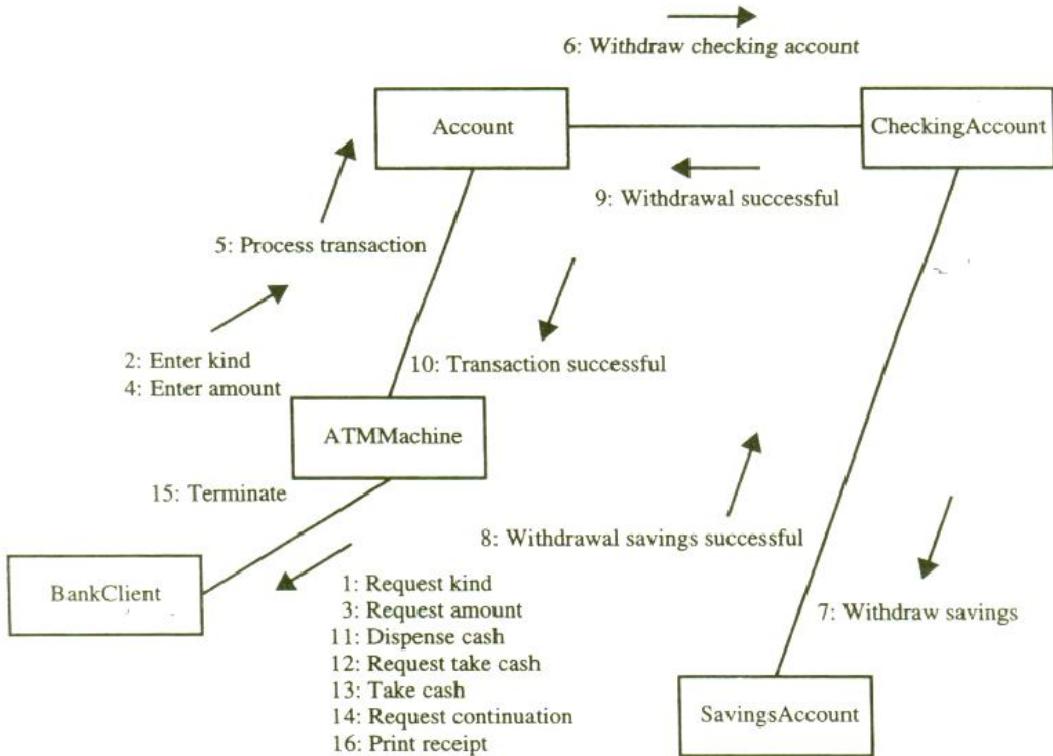


FIGURE 7-6
The collaboration diagram for the Withdraw Checking use case.

FIGURE 7-7
The sequence diagram for the Withdraw More from Checking use case.





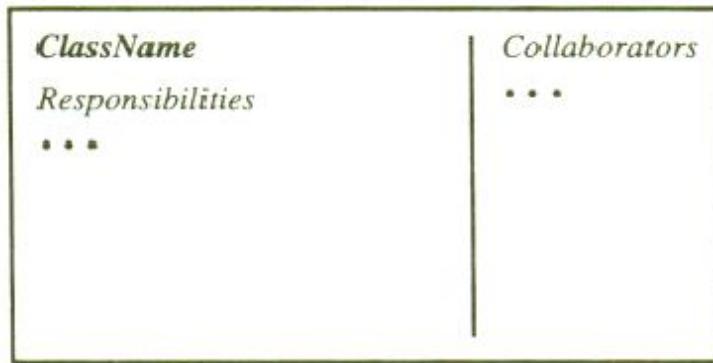
The collaboration diagram for the Withdraw More from Checkinguse case.

14.5 CLASSES, RESPONSIBILITIES, AND COLLABORATORS

Classes, responsibilities, and collaborators (CRC), developed by Cunningham, Wilkerson, and Beck, was first presented as a way of teaching the basic concepts of object-oriented development . Classes, Responsibilities, and Collaborators is a technique used for identifying classes' responsibilities and therefore their attributes and methods. Furthermore, Classes, Responsibilities, and Collaborators can help us identify classes. Classes, Responsibilities, and Collaborators is more a teaching technique than a method for identifying classes.

Classes, Responsibilities, and Collaborators is based on the idea that an object either can accomplish a certain responsibility itself or it may require the assistance of other objects. If it requires the assistance of other objects, it must collaborate with those objects to fulfill its responsibility . By identifying an object's responsibilities and collaborators (cooperative objects with which it works) you can identify its attributes and methods. Classes, Responsibilities, and Collaborators cards are 4" X 6" index cards. All the information for an object is written on a card, which is cheap, portable, readily available, and familiar. Figure shows an idealized card.

The class name should appear in the upper left-hand corner, a bulleted list of responsibilities should appear under it in the left two thirds of the card, and the list of collaborators should appear in the right third. However, rather than simply tracing the details of a collaboration in the form of message sending, Classes, Responsibilities, and Collaborators cards place the designer's focus on the motivation for collaboration by representing (potentially) many messages as phrases of English text.



A Classes, Responsibilities, and Collaborators (CRC) index card. Classes,

Responsibilities, and Collaborators stresses the importance of creating objects, not to meet mythical future needs, but only under the demands of the moment. This ensures that a design contains only as much information as the designer has directly experienced and avoids premature complexity. Working in teams helps here, because a concerned designer can influence team members by suggesting scenarios aimed specifically at suspected weaknesses or omissions.

14.6 CLASSES, RESPONSIBILITIES, AND COLLABORATORS PROCESS

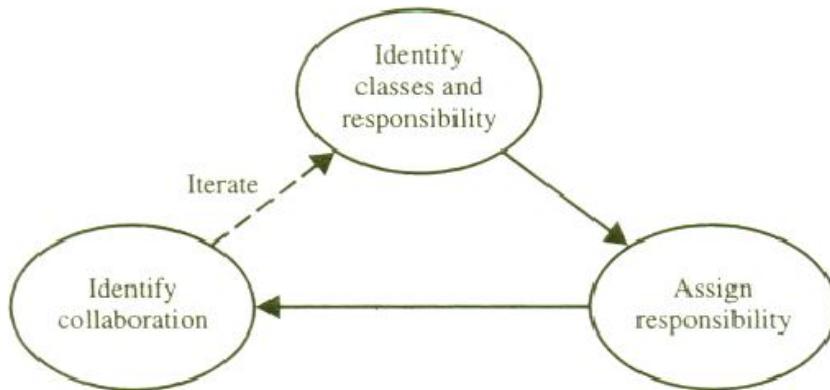
The Classes, Responsibilities, and Collaborators process consists of three steps (see Figure)

1. Identify classes' responsibilities (and identify classes).
2. Assign responsibilities.
3. Identify collaborators.

Classes are identified and grouped by common attributes, which also provides candidates for super classes. The class names then are written onto Classes, Responsibilities, and Collaborators cards. The card also notes sub- and super classes to show the class structure. The application's requirements then are examined for actions and information associated with each class to find the responsibilities of each class.

Next, the responsibilities are distributed; they should be as general as possible and placed as high as possible in the inheritance hierarchy. The idea in locating collaborators is to identify how classes interact. Classes (cards) that have a close collaboration are grouped together physically.

The Classes, Responsibilities, and Collaborators process.



The ViaNet Bank ATM System: Identifying Classes by Using Classes, Responsibilities, and Collaborators

We already identified the initial classes of the bank system. The objective of this example is to identify objects' responsibilities such as attributes and methods in that system.

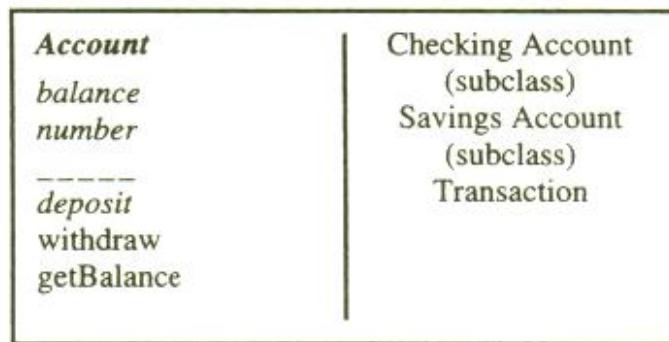
Account and Transaction provide the banking model. Note that Transaction assumes an active role while money is being dispensed and a passive role thereafter. The class Account is responsible mostly to the BankClient class and it collaborates with several objects to fulfill its responsibilities. Among the responsibilities of the Account class to the BankClient class is to keep track of the BankClient balance, account number, and other data that need to be remembered. These are the attributes of the Account class. Furthermore, the Account class provides certain services or methods, such as means for BankClient to deposit or withdraw an amount and display the account's Balance (see Figure).

Classes, Responsibilities, and Collaborators encourages team members to pick up the card and assume a role while "executing" a scenario. It is not unusual to see a designer with a card in each hand, waving them about, making a strong identification with the objects while describing their collaboration. Ward Cunningham writes: Classes, Responsibilities, and Collaborators cards work by taking people through programming episodes together.

As cards are written for familiar objects, all participants pick up the same context and ready themselves for decision making. Then, by waving cards and pointing fingers and yelling statements like, "no, this guy should do that," decisions are made. Finally, the group starts to relax as consensus has been reached and the issue becomes simply finding the right words to record a decision as a responsibility on a card.

In similar fashion other cards for the classes that have been identified earlier in this chapter must be created, with the list of their responsibilities and their collaborators. As you can see from Figure , this process is iterative. Start with few cards (classes) then proceed to play "what if." If the situation calls for a responsibility not already covered by one of the objects, either add the responsibility to an object or create a new object to address that responsibility. If one of the objects becomes too cluttered during this process, copy the information on

Classes, Responsibilities, and Collaborators for the Account object.

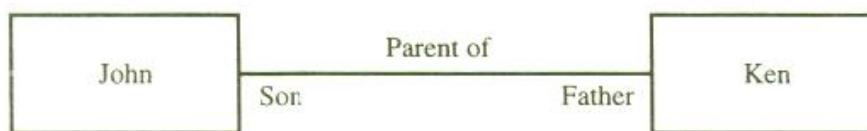


- .Analyzing relationships among classes.
- .Identifying association.
- .Association patterns.
- .Identifying super- and subclass hierarchies.
- .Identifying aggregation or a-part-of compositions.
- .Class responsibilities.
- .Identifying attributes and methods by analyzing use cases and other UML diagrams.

14.7 ASSOCIATIONS

Association represents a physical or conceptual connection between two or more objects) For example, if an object has the responsibility for telling another object that a credit card number is valid or invalid, the two classes have an association. In these chapters, we learnt that the binary associations are shown as lines connecting two class symbols(Ternary and higher-order associations are shown as diamonds connecting to a "class symbol by lines, and the association name is written above or below the line:

The association name can be omitted if the relationship is obvious. In some cases, you will want to provide names for the roles played by the individual classes making up the relationship. The role name on the side closest to each class describes the role that class plays relative to the class at the other end of the line, and vice versa .
Basic association. See Chapter 5 for a detailed discussion of association.



Identifying Associations

Identifying associations begins by analyzing the interactions between classes. After all, any dependency relationship between two or more classes is an association . You must examine the responsibilities to determine dependencies. In other words, if an object is responsible for a specific task (behavior) and lacks all the necessary knowledge needed to perform the task, then the object must delegate the task to another object that possesses such knowledge. Wirfs-Brock, Wilkerson,"ard Wiener provide the following questions that can help us to identify associations:

.As the class capable of fulfilling the required task by itself? .If not, what does it need? ! From what other class can it acquire what it needs?

. Answering these questions helps us identify association. The approach you should take to identify association is flexibility. First, extract all candidates' associations from the problem statement and get them down on paper. You can refine them later. Notice that a-part-of structures (aggregation) and associations are very similar. So, how do you distinguish one from the other? It depends on the problem domain; after all, a-part-of structure is a special case of association. Simply pick the one most natural for the problem domain. If you can represent the problem more easily with association, then select it; otherwise, use a-part-of structure, which is described later in the chapter.

14.8 GUIDELINES FOR IDENTIFYING ASSOCIATION

Following are general guidelines for identifying the tentative associations:

.A dependency between two or more classes may be an association. Association often corresponds to a verb or prepositional phrase, such as part of, next to, works for, or contained in.

. A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.

Common Association Patterns

The common association patterns are based on some of the common associations defined by researchers and practitioners: Rumbaugh et al. Coad and Yourdon , and others. These include .ycation association-next to, part of, contained in. For example, consider a soup object, cheddar cheese is a-part-of soup. The a-part-of relation is a special type of association, discussed in more detail later in the chapter. .Communication association-talk to, order to. For example, a customer places an order (communication association) with an operator person (see Figure).

These association patterns and similar ones can be stored in the repository and added to as more patterns are discovered. However, currently, this capability of the unified approach's repository is more conceptual than real, but it is my hope that CASE tool vendors in the near future will provide this capability.

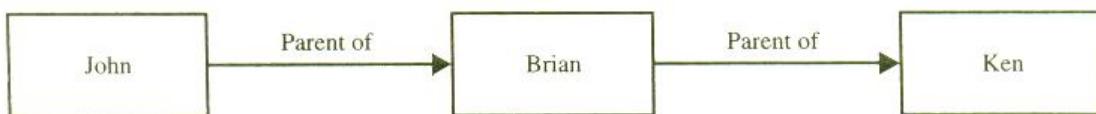
A customer places an order (communication association) with an operator person.
Eliminate Unnecessary Associations

.mplementation association. Defer implementation-specific associations to the design phase. Implementation associations are concerned with the implementation or design of the class within certain programming or development environments and not relationships among business objects.

. Ternary associations. Ternary or n-ary association is an association among more than two classes . Ternary associations complicate the representation. When possible, restate ternary associations as binary associations.

. Directed actions (or derived) association. Directed actions (derived) associations can be defined in terms of other associations. Since they are redundant, avoid these types of association. For example, Grandparent of can be defined in terms of the parent of association (see Figure). Choose association names carefully. Do not say how or why a situation came about; say what it is. Add role names where appropriate, especially to distinguish multiple associations. These often are discovered by testing access paths to objects.

Grandparent of Ken can be defined in terms of the parent association.



14.9 LET US SUM UP

The second method for identifying classes is the common class patterns approach based on the knowledge base of the common classes proposed by various researchers. These researchers compiled and listed several categories for finding the candidate classes and objects.

The third method we studied was use-case driven. To identify objects of a system and their behaviors, the lowest level of executable use cases is further analyzed with a sequence and collaboration diagram pair. By walking through the steps, you can determine what objects are necessary for the steps to take place. Finally, we looked at the Classes, Responsibilities, and Collaborators, which is a useful tool for learning about class responsibilities and identifying classes. These approaches can be mixed for identifying the classes of a given problem. Naming a class is an important activity, too.

The class should describe a single object, so it should be a singular noun or an adjective and a noun. A general rule for naming classes is that you use names with which the users or clients are comfortable. Choose the class names from standard vocabulary for the subject matter.

14.10 POINTS FOR DISCUSSION

1. Discuss class, responsibilities and collaborators process
2. Discuss object behavior analysis

14.11 LESSON – END ACTIVITIES

1. Establish the Guidelines for identifying associations.
2. Discuss about Associations.

14.12 REFERENCES

1. Norman,Ronald- object oriented system analysis and design –printice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 15: SUPER SUB CLASS RELATIONSHIPS

CONTENTS

- 15.0 AIMS AND OBJECTIVES
- 15.1 INTRODUCTION.
- 15.2 SUPER-SUB CLASS RELATIONSHIPS
- 15.3 GUIDELINES FOR IDENTIFYING SUPER-SUB RELATIONSHIP, A GENERALIZATION
- 15.4 A.PART.OF RELATIONSHIPS-AGGREGATION
- 15.5 A.PART-OF RELATIONSHIP PATTERNS
- 15.6 LET US SUM UP
- 15.7 POINTS FOR DISCUSSION
- 15.8 LESSON – END ACTIVITIES
- 15.9 REFERENCES

15.0 AIMS AND OBJECTIVES

You should be able to develop the relationship and its associations among classes.

15.1 INTRODUCTION.

The other aspect of classification is identification of super-sub relations among classes. For the most part, a class is part of a hierarchy of classes, where the top class is the most general one and from it descend all other, more specialized classes. The super-sub class relationship represents the inheritance relationships between related classes, and the class hierarchy determines the lines of inheritance between classes. Class inheritance is useful for a number of reasons. For example, in some cases, you want to create a number of classes that are similar in all but a few characteristics. In other cases, someone already has developed a class that you can use, but you need to modify that class. Subclasses are more specialized versions of their superclasses. The classes are not ordered this way for convenience's sake. Superclass-subclass relationships, also known as generalization hierarchy, allow objects to be built from other objects. Such relationships allow us to explicitly take advantage of the commonality of objects when constructing new classes.

15.2 SUPER-SUB CLASS RELATIONSHIPS

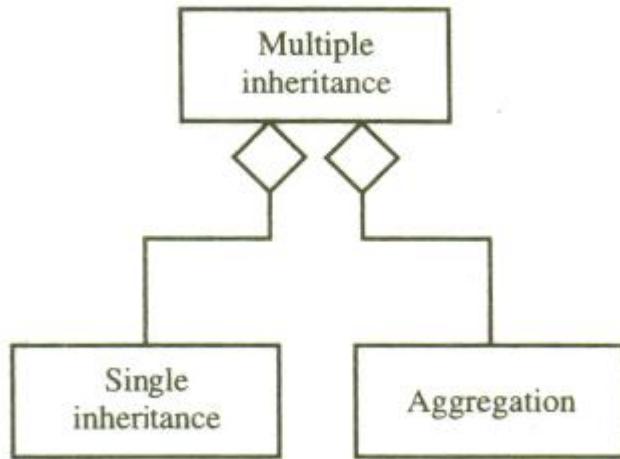
The super-sub class hierarchy is a relationship between classes, where one class is the parent class of another (derived) class. Recall from earlier chapter that the parent class also is known as the base or super class or ancestor. The super-sub class hierarchy is based on inheritance, which is programming by extension as opposed to programming by reinvention . The real advantage of using this technique is that we can build on what we already have and, more important, reuse what we already have. Inheritance allows classes to share and reuse behaviors and attributes. Where the behavior of a class instance is defined in that class's methods, a class also inherits the behaviors and attributes of all of its superclasses. Now let us take a look at guidelines for identifying classes.

15.3 GUIDELINES FOR IDENTIFYING SUPER-SUB RELATIONSHIP, A GENERALIZATION

The following are guidelines for identifying super-sub relationships in the application:

.Top-down. Look for noun phrases composed of various adjectives in a class name. Often, you can discover additional special cases. Avoid excessive refinement. Specialize only when the subclasses have significant behavior. For example, a phone operator employee can be represented as a cook as well as a clerk or manager because they all have similar behaviors.

.Bottom-up. Look for classes with similar attributes or methods. In most cases, you can group them by moving the common attributes and methods to an abstract class. You may have to alter the definitions a bit; this is acceptable as long as generalization truly applies. However, do not force classes to fit a preconceived generalization structure.
.jEusability. Move attributes and behaviors (methods) as high as possible in the hierarchy. At the same time, do not create very specialized classes at the top of the hierarchy. This is easier said than done. The balancing act can be achieved through several iterations. This process ensures that you design objects that can be reused in another application.



One way to achieve the benefits of multiple inheritance is to inherit from the most appropriate class and add an object of another class as an attribute. In essence, a multiple inheritance can be represented as an aggregation of a single inheritance and aggregation. This meta-model reflects this situation.

Multiple inheritance. Avoid excessive use of multiple inheritance. Multiple inheritance brings with it complications such as how to determine which behavior to get from which class, particularly when several ancestors define the same method. It also is more difficult to understand programs written in a multiple inheritance system. One way of achieving the benefits of multiple inheritance is to inherit from the most appropriate class and add an object of another class as an attribute. However, use multiple inheritance when it is appropriate. For example, if the owner of a restaurant prepares the soups, you can utilize multiple inheritance structure to define an OwnerOperator class that inherits its attributes and methods from both the Owner and Operator classes.

15.4 A PART OF RELATIONSHIPS-AGGREGATION

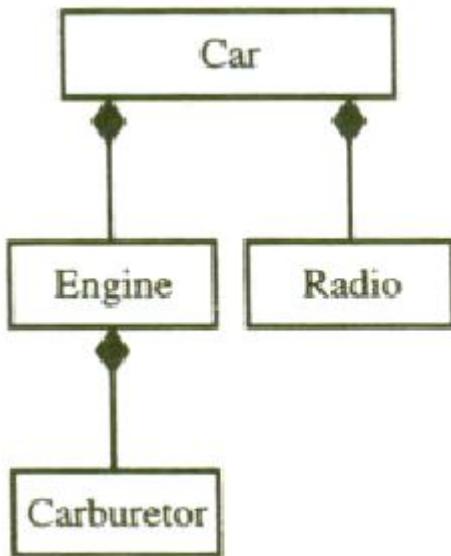
A-part-of relationship, also called aggregation, represents the situation where a class consists of several component classes. A class that is composed of other classes does not behave like its parts; actually, it behaves very differently. For example, a car consists of many other classes, one of which is a radio, but a car does not behave like a radio (see Figure).

Two major properties of a-part-of relationship are transitivity and antisymmetry

,Transitivity. The property where, if A is part of Band B is part of C, then A is part of C. For example, a carburetor is part of an engine and an engine is part of a car; therefore, a carburetor is part of a car. Figure shows a-part-of structure.

Antisymmetry. The property of a-part-of relation where, if A is part of B, then . B is not part of A. For example, an engine is part of a car, but a car is not part of an engine. A

clear distinction between the part and the whole can help us determine where responsibilities for certain behavior must reside. This is done mainly by asking the following questions :



A-part-ofcomposition.A carburetoris a part of an engineand an engineand a radioare parts ofa car.

. Does the part class belong to a problem domain?
. Is the part class within the system's responsibilities? .Does the part class capture more than a single value? (If it captures only a single value, then simply include it as an attribute with the whole class.)

.Does it provide a useful abstraction in dealing with the problem domain?
We saw that the UML uses hollow or filled diamonds to represent aggregations. A filled diamond signifies the strong form of aggregation, which is composition. For example, one might represent aggregation such as container and collection as hollow diamonds (see Figures) and use a solid diamond to represent composition, which is a strong form of aggregation (see Figure).

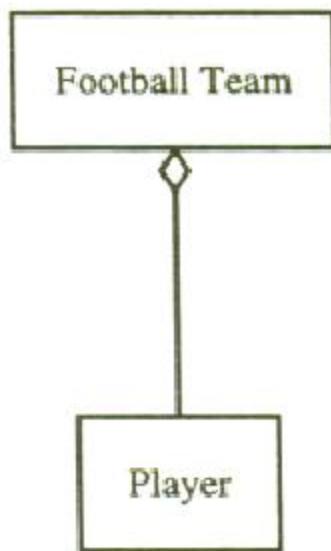
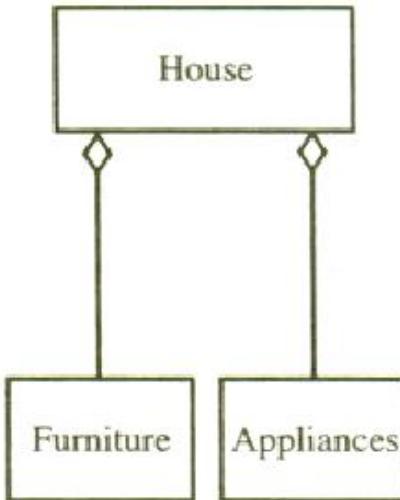
15.5 A.PART-OF RELATIONSHIP PATTERNS

To identify a-part-of structures, Coad and Yourdon provide the following guidelines:

.Assembly. An assembly is constructed from its parts and an assembly-part situation physically exists; for example, a French onion soup is an assembly of onion, butter, flour, wine, French bread, cheddar cheese, and so on.

. Container. A physical whole encompasses but is not constructed from physical parts; for example, a house can be considered as a container for furniture and appliances (see Figure).

A house is a container.



A football team is a collection of players.

.Collection-member. A conceptual whole encompasses parts that may be physical or conceptual; for example, a football team is a collection of players.

CASE STUDY: RELATIONSHIP ANALYSIS FOR THE VIANET BANK ATM SYSTEM

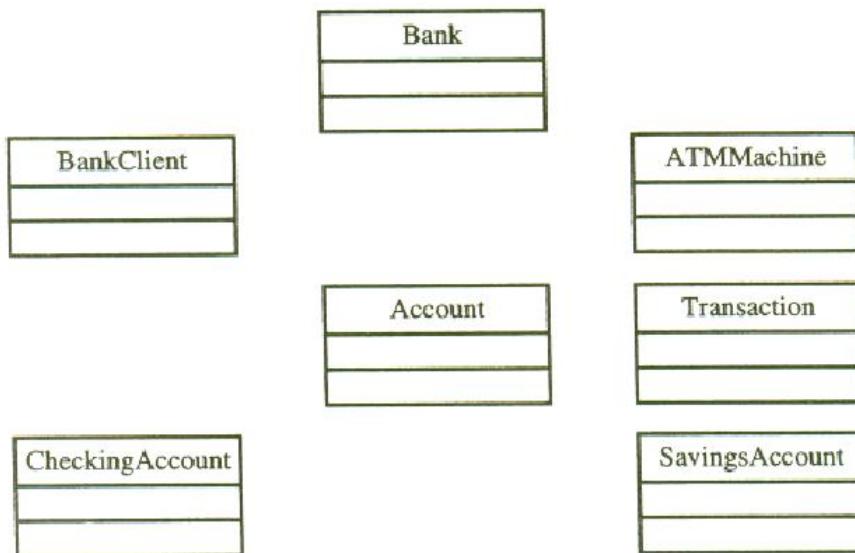
To better gain experience in object relationship analysis, we use the familiar bank system case and apply the concepts in this chapter for identifying associations, supersub relationships, and a-part-of relationships for the classes identified. As explained before, we must start by reading the requirement specification, which is presented here. Furthermore, object-oriented analysis and design are performed in an iterative process using class diagrams. Analysis is performed on a piece of the system, design details are added to this partial analysis model, and then the design is implemented. Changes can be made to the implementation and brought back into the analysis model to continue the cycle. This iterative process is unlike the traditional waterfall technique, in which all analysis is completed before design begins.

Identifying Classes' Relationships

One of the strengths of object-oriented analysis is the ability to model objects as they exist in the real world. To accurately do this, you must be able to model more than just an object's internal workings. You also must be able to model how objects relate to each other. Several different relationships exist in the ViaNet bank ATM system, so we need to define them

Developing a UML Class Diagram Based on the Use-Case Analysis

The UML class diagram is the main static analysis and design diagram of a system. The analysis generally consists of the following class diagrams .One class diagram for the system, which shows the identity and definition of classes in the system, their interrelationships, and various packages containing groupings of classes.



UML class diagram for the ViaNet bank ATM system. Some CASE tools such as the SA/Object Architect can automatically define classes and draw them from use cases or collaboration/ sequence diagrams. However, presently, it cannot identify all the classes. For this example, S/A Object was able to identify only the BankClient class.

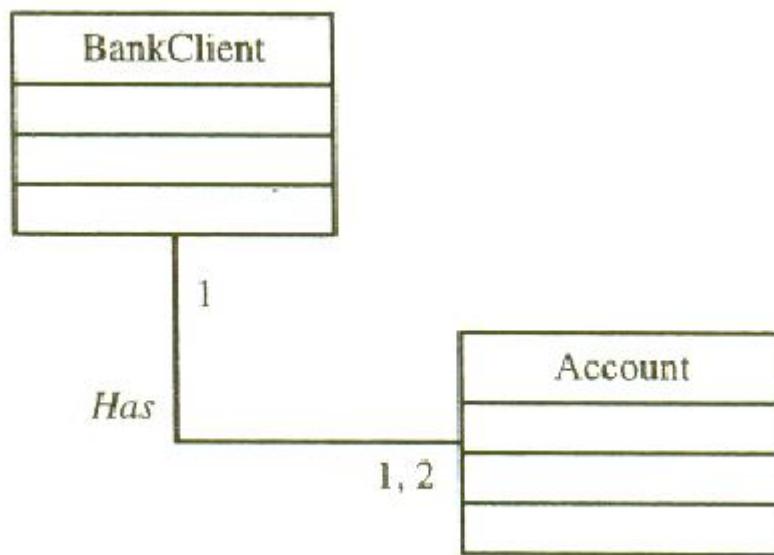
. Multiple class diagrams that represent various pieces, or views, of the system class diagram. .Multiple class diagrams, that show the specific static relationships between various classes.

First, we need to create the classes that have been identified in the previous chapter; we will add relationships later (see Figure).

Defining Association Relationships

Identifying association begins by analyzing the interactions of each class. Remember that any dependency between two or more classes is an association. The following are general guidelines for identifying the tentative associations, as explained in this chapter: .Association often corresponds to verb or prepositional phrases, such as part of, next to, works for, or contained in. .A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.

Some common patterns of associations are these: .Location association. For example, next to, part of, contained in (notice that apart- of relation is a special type of association). .Directed actions association. .Communication association. For example, talk to, order from. The first obvious relation is that each account belongs to a bank client since each BankClient has an account. Therefore, there is an association between the BankClient and Account classes. We need to establish cardinality among these



Defining the BankClient-Account association multiplicity. One Client can have one or more Accounts (checking and savings accounts). classes. By default, in most CASE tools such as SNOBJECT Architect, all associations are considered one to one (one client can have only one account and vice versa). However, since each BankClient can have one or

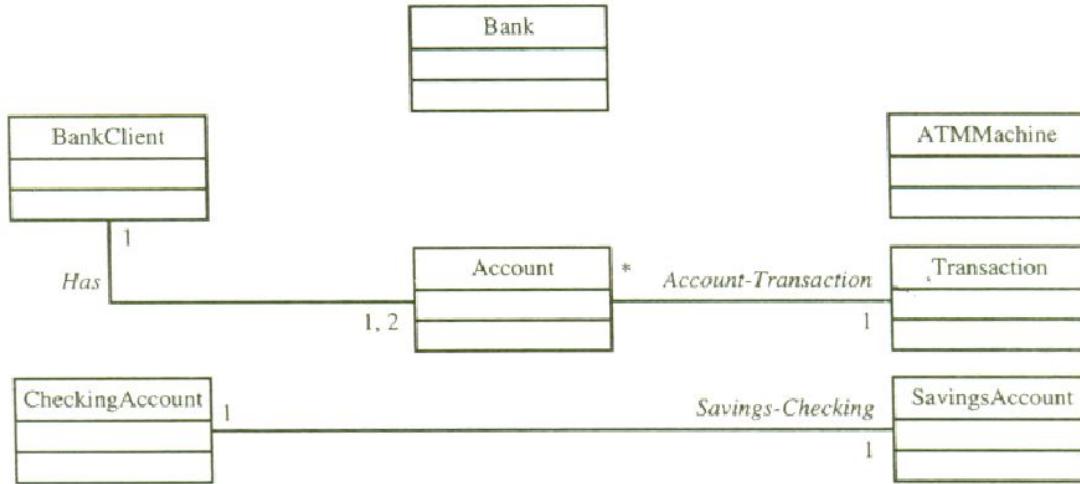
two accounts we need to change the cardinality of the association (see Figure). Other associations and their cardinalities are defined in Table 8-1 and demonstrated in Figure .

Defining Super-Sub Relationships

Let us review the guidelines for identifying super-sub relationships: . Top-down. Look for noun phrases composed of various adjectives in the class name. . Bottom-up. Look for classes with similar attributes or methods. In most cases, you can group them by moving the common attributes and methods to an abstract class. . Reusability. Move attributes and behaviors (methods) as high as possible in the hierarchy. . Multiple inheritance. Avoid excessive use of multiple inheritance. CheckingAccount and SavingsAccount both are types of accounts. They can be defined as specializations of the Account class. When implemented, the Account

SOME ASSOCIATIONS AND THEIR CARDINALITIES IN THE BANK SYSTEM

Class	Related class	Association name	Cardinality
Account	BankClient	Has	One
BankClient	Account		One or two
SavingsAccount	CheckingAccount		One
CheckingAccount	SavingsAccount		Zero or one
Account	Transaction	Account-Transaction	Zero or more
Transaction	Account		One



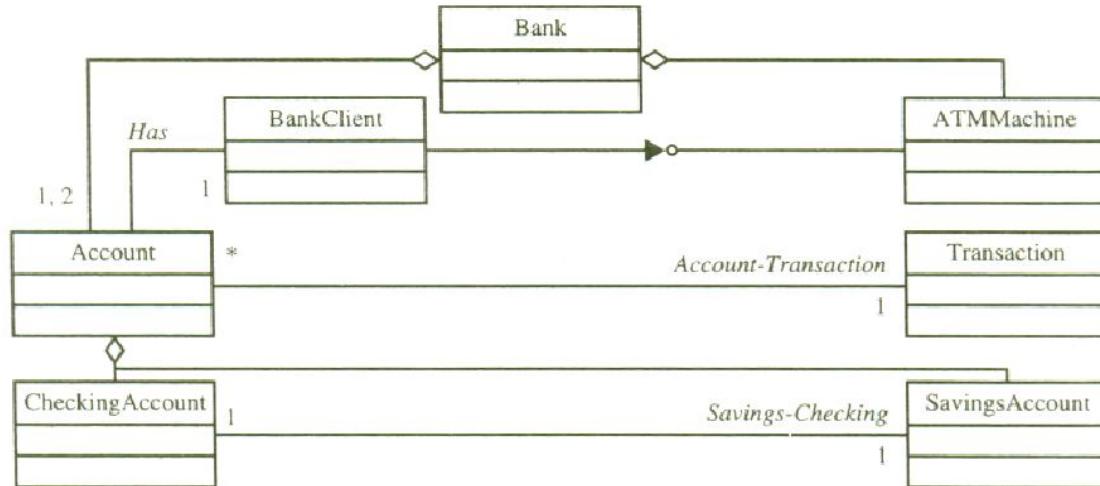
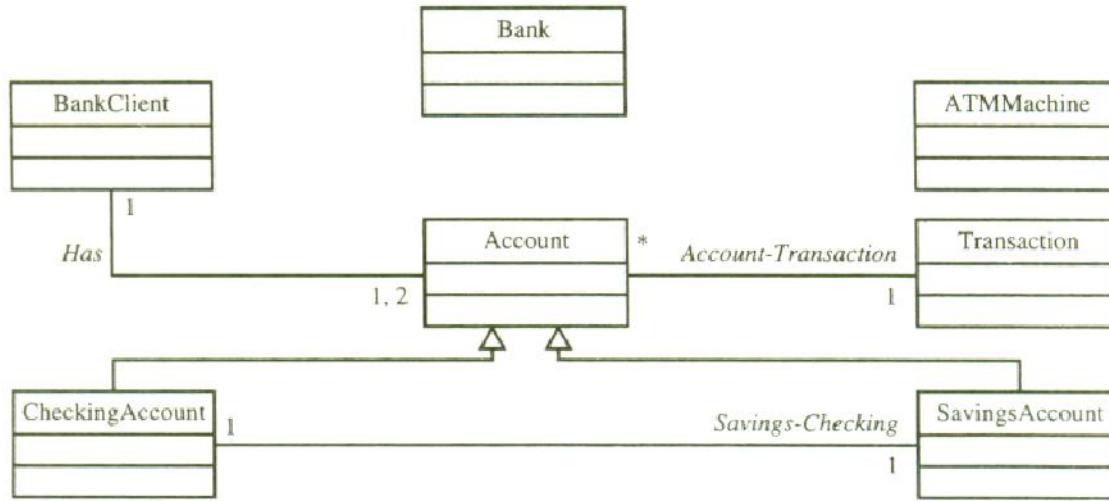
Associations among the ViaNet bank ATMsystem classes.

class will define attributes and services common to all kinds of accounts, with CheckingAccount and SavingsAccount each defining methods that make them more specialized. Figure depicts the super-sub relationships among Accounts, Savings Account, and CheckingAccount.

Identifying the Aggregation/a-Part-of Relationship

To identify a-part-of structures, we look for the following clues: .Assembly. A physical whole is constructed from physical parts. . Container. A physical whole encompasses but is not constructed from physical parts. . Collection-Member. A conceptual whole encompasses parts that may be physical or conceptual.

Super-sub relationships among the Account, SavingsAccount, and CheckingAccount classes.



Association, generalization, and aggregation among the ViaNet bank classes. Notice that the super-sub arrows for **CheckingAccount** and **SavingsAccount** have merged. The relationship between **BankClient** and **ATMMachine** is an interface.

A bank consists of ATM machines, accounts, buildings, employees, and so forth. However, since buildings and employees are outside the domain of this application, we define the **Bank** class as an aggregation of **ATMMachine** and **Account** classes. Aggregation is a special type of association. Figure depicts the association,

generalization, and aggregation among the bank systems classes. If you are wondering what is the relationship between the BankClient and ATMMachine, it is an interface. Identifying a class interface is a design activity of object-oriented system development.

15.6 LET US SUM UP

Some common associations patterns are *next to*, *part of*, and *contained in* a relation; directed actions and communication associations include *talk to* or *order from*.

To identify super-sub relationships in the application, look for noun phrases composed of various adjectives in the class name in top-down analysis. Specialize only when the subclasses have significant behavior. In bottom-up analysis, look for classes with similar attributes or methods. Group the classes by moving those classes with common attributes and methods as high as possible in the hierarchy. At the same time, do not create very specialized classes at the top of the hierarchy. This balancing act can be achieved through several iterations.

The process ensures that you design objects that can be reused in another application. Finally, avoid excessive use of multiple inheritance. It is more difficult to understand programs written in a multiple inheritance system. One way of achieving the benefits of multiple inheritance is to inherit from the most appropriate class and add an object of another class as an attribute. The a-part-of relationship, sometimes called *aggregation*, represents a situation where a class comprises several component classes. A class composed of other classes does not behave like its parts but very differently. For example, a car consists of many other classes, one of which is a radio, but a car does not behave like a radio. Some common aggregation/a-part-of patterns are assembly, container, and collection-member. The a-part-of structure is a special form of association, and similarly, association can be represented by the a-part-of relation.

Identifying attributes and methods is like finding classes, a difficult activity and an iterative process. Once again, the use cases and other UML diagrams will be a guide for identifying attributes, methods, and relationships among classes. Methods and messages are the workhorses of object-oriented systems. The sequence diagrams can assist us in defining services that the objects must provide.

An event is considered to be an action that transmits information; therefore, these actions are operations that the objects must perform. Additionally, operations (methods or behavior) in the object-oriented system usually correspond to queries about attributes and associations of the objects. Therefore, methods are responsible for managing the value of attributes such as query, updating, reading, and writing.

15.7 POINTS FOR DISCUSSION

1. Discuss super – sub class
2. Evaluate super – sub class relationship
3. Justify Aggregation Relationship

15.8 LESSON – END ACTIVITIES

1. Validate Association Relationship.
2. Discuss about A-part-of relationship patterns.

15.9 REFERENCES

1. Norman,Ronald- object oriented system analysis and design –prinice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

UNIT – IV

LESSON – 16: OBJECT ORIENTED AXIOMS

CONTENTS

- 16.0 AIMS AND OBJECTIVES
- 16.1 INTRODUCTION
- 16.2 OBJECT-ORIENTED DESIGN AXIOMS
- 16.3 COROLLARIES
- 16.4 LET US SUM UP
- 16.5 POINTS FOR DISCUSSION
- 16.6 LESSON – END ACTIVITIES
- 16.7 REFERENCES

16.0 AIMS AND OBJECTIVES

You should be able to understand the object oriented axioms

16.1 INTRODUCTION

It was explained in previous chapters that the main focus of the analysis phase of software development is on "what needs to be done." The objects discovered during analysis can serve as the framework for design . The class's attributes, methods, and associations identified during analysis must be designed for implementation as a data type expressed in the implementation language. New classes must be introduced to store intermediate results during program execution. Emphasis shifts from the application domain to implementation and computer concepts such as user interfaces or view layer and access layer (see Figures).

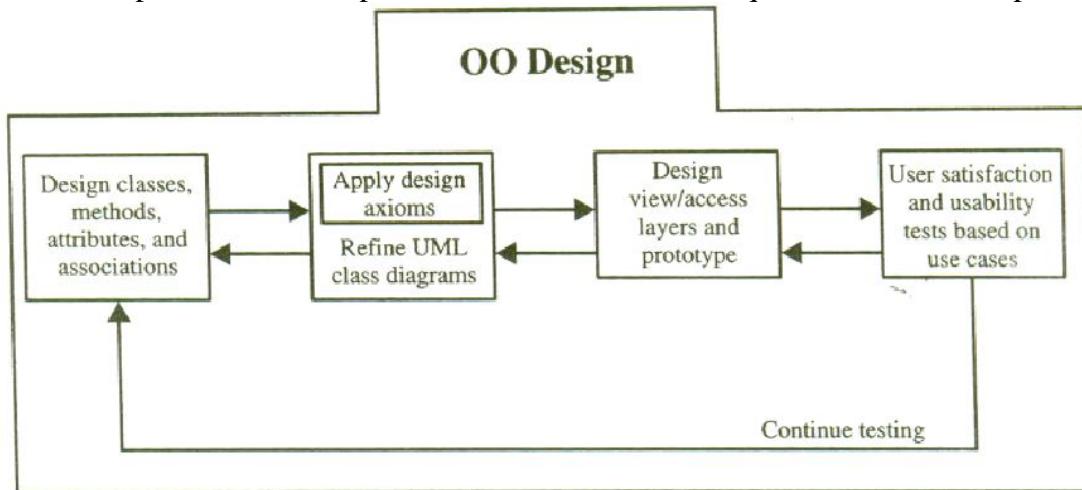
During the analysis, we look at the physical entities or business objects in the system; that is, who the players are and how 'they cooperate to do the work of the application. These objects represent tangible elements of the business. As we saw in Chapter 7, these objects could be individuals, organizations, machines, or whatever else makes sense in the context of the real-world system. During the design phase, we elevate the model into logical entities, some of which might relate more to the computer domain (such as user interfaces or the access layer) than the realworld or the physical domain (such as people or employees). This is where we begin thinking about how to actually implement the problem in a program. The goal here is to design the classes that we need to implement the system.

Fortunately, the design model does not look terribly different from the analysis model. The difference is that, at this level, we focus on the view and access classes, such as how to maintain information or the best way to interact with a user or present information. It also is useful, at this stage, to have a good understanding of the classes in a development environment that we are using to enforce reusability. In software development, it is tempting not to be concerned with design. After all, you (the designer) are so involved with the system that it might be difficult to stop and think about the consequences of each design choice. However, the time spent on design has a great impact on the overall success of the software development project. A large payoff is associated with creating a good design "up front," before writing a single line of code. While this is true of all programming, classes and objects underscore the approach even more. Good design usually simplifies the implementation and maintenance of a project.

In this chapter, we look at the object-oriented design process and *axioms*. The basic goal of the axiomatic approach is to formalize the design process and assist in establishing a scientific foundation for the object-oriented design process, to provide a fundamental basis for the creation of systems. Without scientific principles, the design field never will be systematized and so will remain a subject difficult to comprehend, codify, teach, and practice .

16.2 OBJECT-ORIENTED DESIGN AXIOMS

By definition, an *axiom* is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception. Such explains that axioms may be hypothesized from a large number of observations by noting the common phenomena shared by all cases; they cannot be proven or derived, but they can be invalidated by counterexamples or exceptions. A *theorem* is a proposition that may not be self-evident but can be proven from accepted axioms. It, therefore, is equivalent to a law or principle.



The object-oriented design process in the unified approach.

Consequently, a theorem is valid if its referent axioms and deductive steps are valid. A *corollary* is a proposition that follows from an axiom or another proposition that has been proven.

The author has applied Suh's design axioms to object-oriented design. Axiom 1 deals with relationships between system components (such as classes, requirements, and software components), and Axiom 2 deals with the complexity of design.

Axiom 1. *The independence axiom.* Maintain the independence of components.

Axiom 2. *The information axiom.* Minimize the information content of the design.

Axiom 1 states that, during the design process, as we go from requirement and use case to a system component, each component must satisfy that requirement without affecting other requirements. You been asked to design a refrigerator door, and there are two requirements: The door should provide access to food, and the energy lost should be minimal when the door is opened and closed. In other words, opening the door should be independent of losing energy. Is the vertically hung door a good design? We see that vertically hung door violates Axiom 1, because the two specific requirements (i.e., access to the food and minimal energy loss) are coupled and are not independent in the proposed design. When, for example, the door is opened to take out milk, cold air in the refrigerator escapes and warm air from the outside enters. What is an uncoupled design that somehow does not combine these two requirements? Once such uncoupled design of the refrigerator door is a horizontally hinged door, such as used in chest-type freezers. When the door is opened to take out milk, the cold air (since it is heavier than warm air) will sit at the bottom and not escape. Therefore, opening the door provides access to the food and is independent of energy loss. This type of design satisfies the first axiom.

Axiom 2 is concerned with simplicity. Scientific theoreticians often rely on a general rule known as *Occam's razor*, after William of Occam, a 14th century scholastic philosopher. Briefly put, Occam's razor says that, "The best theory explains the known facts with a minimum amount of complexity and maximum simplicity and straightforwardness."

Occam's razor has a very useful implication in approaching the design of an object-oriented application. Let us restate Occam's razor rule of simplicity in object-oriented terms:

The best designs usually involve the least complex code but not necessarily the fewest number of classes or methods. Minimizing complexity should be the goal, because that produces the most easily maintained and enhanced application. In an object-oriented system, the best way to minimize complexity is to use inheritance and the system's built in classes and to add as little as possible to what already is there.

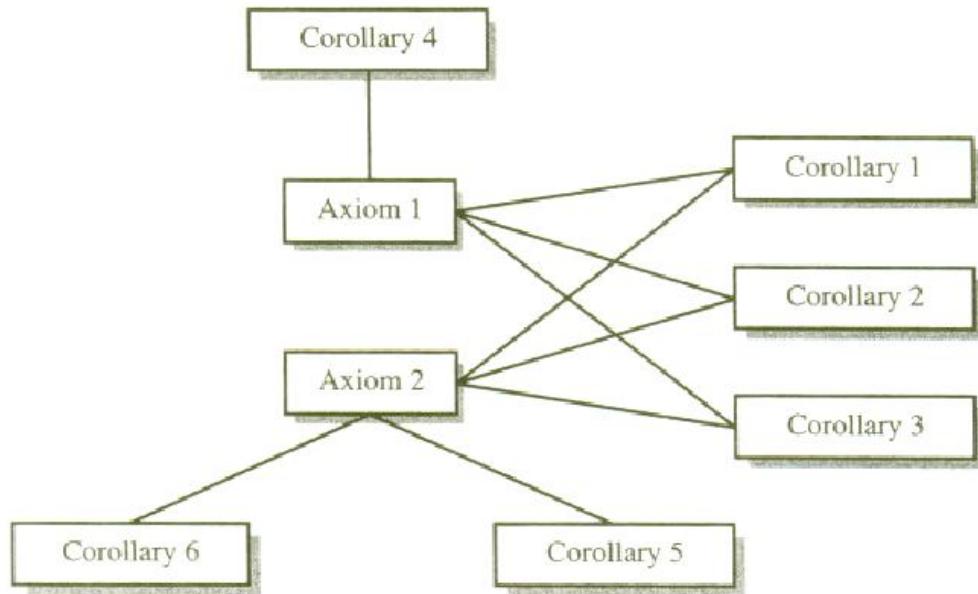
16.3 COROLLARIES

From the two design axioms, many corollaries may be derived as a direct consequence of the axioms. These corollaries may be more useful in making specific design decisions, since they can be applied to actual situations more easily than the original axioms. They even may be called *design rules*, and all are derived from the two basic axioms (see Figure):

.Corollary 1. *Uncoupled design with less information content.* Highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects.

FIGURE

The origin of corollaries. Corollaries 1, 2, and 3 are from both axioms, whereas corollary4 is from axiom 1 and corollaries 5 and 6 are from axiom 2.



.Corollary 2. *Single purpose.* Each class must have a single, clearly defined purpose. When you document, you should be able to easily describe the purpose of a class in a few sentences.

. Corollary 3. *Large number of simple classes.* Keeping the classes simple allows reusability.

.Corollary 4. *Strong mapping.* There must be a strong association between the physical system (analysis's object) and logical design (design's object).

. Corollary 5. *Standardization.* Promote standardization by designing interchangeable components and reusing existing classes or components.

.Corollary 6. *Design with inheritance.* Common behavior (methods) must be moved to super classes. The super class-subclass structure must make logical sense.

Corollary 1. Uncoupled Design with Less Information Content

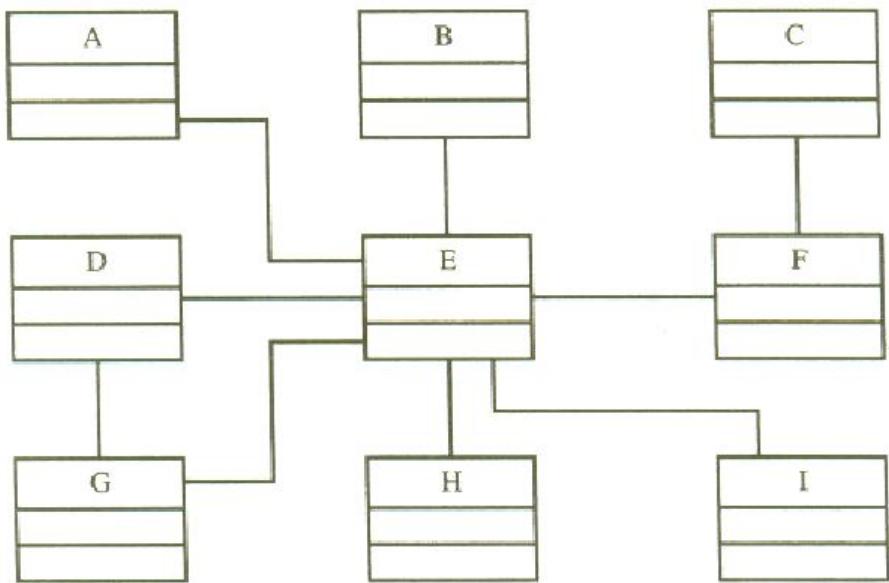
The main goal here is to maximize objects cohesiveness among objects and software components in order to improve coupling because only a minimal amount of essential information need be passed between components. *Coupling* is a measure of the strength of association established by a connection from one object or software component to another. Coupling is a binary relationship: A is coupled with B. Coupling is important when evaluating a design because it helps us focus on an important issue in design. For example, a change to one component of a system should have a minimal impact on other components. Strong coupling among objects complicates a system, since the class is harder to understand or highly interrelated with other classes. The degree of coupling is a function of

1. How complicated the connection is.
2. Whether the connection refers to the object itself or something inside it.
3. What is being sent or received.

The degree, or strength, of coupling between two components is measured by the amount and complexity of information transmitted between them. Coupling increases (becomes stronger) with increasing complexity or obscurity of the interface.

Coupling decreases (becomes lower) when the connection is to the component interface rather than to an internal component. Coupling also is lower for data connections than for control connections. Object-oriented design has two types of coupling: interaction coupling and inheritance coupling .

Interaction coupling involves the amount and complexity of messages between components. It is desirable to have little interaction. Coupling also applies to the complexity of the message. The general guideline is to keep the messages as simple and infrequent as possible. In general, if a message connection involves more than three parameters (e.g., in Method (X, Y, Z), the X, Y, and Z are parameters), examine it to see if it can be simplified. It has been documented that objects connected to many very complex messages are tightly coupled, meaning any change to one invariability leads to a ripple effect of changes in others (see Figure).



E is a tightly coupled object.

In addition to minimizing the complexity of message connections, also reduce the number of messages sent and received by an object . Table 9-1 contains different types of interaction couplings. Inheritance is a form of coupling between super- and subclasses. A subclass is coupled to its superclass in terms of attributes and methods. Unlike interaction coupling, high inheritance coupling is desirable. However, to achieve high inheritance

TYPES OF COUPLING AMONG OBJECTS OR COMPONENTS (shown from highest to lowest)

Degree of coupling	Name	Description
Very high	Content coupling	The connection involves direct reference to attributes or methods of another object.
High	Common coupling	The connection involves two objects accessing a "global data space," for both to read and write.
Medium	Control coupling	The connection involves explicit control of the processing logic of one object by another.
Low	Stamp coupling	The connection involves passing an aggregate data structure to another object, which uses only a portion of the components of the data structure.
Very low	Data coupling	The connection involves either simple data items or aggregate structures all of whose elements are used by the receiving object. This should be the goal of an architectural design.

coupling in a system, each specialization class should not inherit lots of unrelated and unneeded methods and attributes. For example, if the subclass is overwriting most of the

methods or not using them, this is an indication inheritance coupling is *low* and the designer should look for an alternative generalization specialization structure.

Cohesion Coupling deals with interactions between objects or software components. We also need to consider interactions within a single object or software component, called *cohesion*. Cohesion reflects the "single-purposeness" of an object. Highly cohesive components can lower coupling because only a minimum of essential information need be passed between components. Cohesion also helps in designing classes that have very specific goals and clearly defined purposes.

Method cohesion, like function cohesion, means that a method should carry only one function. A method that carries multiple functions is undesirable. Class cohesion means that all the class's methods and attributes must be highly cohesive, meaning to be used by internal methods or derived classes' methods. Inheritance cohesion is concerned with the following questions : .How interrelated are the classes? . Does specialization really portray specialization or is it just something arbitrary? See Corollary 6, which also addresses these questions.

Corollary 2. Single Purpose

Each class must have a purpose, as was explained in Chapter 7. Every class should be clearly defined and necessary in the context of achieving the system's goals. When you document a class, you should be able to easily explain its purpose in a sentence or two. If you cannot, then rethink the class and try to subdivide it into more independent pieces. In summary, keep it simple; to be more precise, each method must provide only one service. Each method should be of moderate size, no more than a page; half a page is better.

Corollary 3. Large Number of Simpler Classes, Reusability

A great benefit results from having a large number of simpler classes. You cannot possibly foresee all the future scenarios in which the classes you create will be reused. The less specialized the classes are, the more likely future problems can be solved by a recombination of existing classes, adding a minimal number of subclasses. A class that easily can be understood and reused (or inherited) contributes to the overall system, while a complex, poorly designed class is just so much dead weight and usually cannot be reused. Keep the following guideline in mind: The smaller are your classes, the better are your chances of reusing them in other projects. Large and complex classes are too specialized to be reused. Object-oriented design offers a path for producing libraries of reusable parts . The emphasis object-oriented design places on encapsulation, modularization, and polymorphism suggests reuse rather than building anew. Cox's description of a software IC library implies a similarity between object-oriented development and building hardware from a standard set of chips . The software IC library is realized with the introduction of design patterns, discussed later in this chapter. Coad and Yourdon argue that software reusability rarely is practiced effectively. But the organizations that will survive in the 21st century will be those that have

achieved high levels of reusability-anywhere from 70-80 percent or more . Griss argues that, although reuse is widely desired and often the...benefit of utilizing object technology, many object-oriented reuse efforts fail because Qf too narrow a focus on technology and not on the policies set forth by an organization. He recommended an institutionalized approach to software development, in which software assets intentionally are created or acquired to be reusable. These assets consistently are used and maintained to obtain high levels of reuse, thereby optimizing the organization's ability to produce high-quality software products rapidly and effectively .

Coad and Yourdon describe four reasons why people are not utilizing this concept:

1. Software engineering textbooks teach new practitioners to build systems from "first principles"; reusability is not promoted or even discussed.
2. The "not invented here" syndrome and the intellectual challenge of solving an interesting software problem in one's own unique way mitigates against reusing someone else's software component.
3. Unsuccessful experiences with software reusability in the past have convinced many practitioners and development managers that the concept is not practical.
4. Most organizations provide no reward for reusability; sometimes productivity is measured in tens of new lines of code written plus a discounted credit (e.g., 50 percent less credit) for reused lines of code.

The primary benefit of software reusability is higher productivity. Roughly speaking, the software development team that achieves 80 percent reusability is four times as productive as the team that achieves only 20 percent reusability. Another form of reusability is using a design pattern, which will be explained in the next section.

Corollary 4. Strong Mapping

Object-oriented analysis and object-oriented design are based on the same model. As the model progresses from analysis to implementation, more detail is added, but it remains essentially the same. For example, during analysis we might identify a class Employee. During the design phase, we need to design this classdesign its methods, its association with other objects, and its view and access classes. A strong mapping links classes identified during analysis and classes designed during the design phase (e.g., view and access classes). Martin and Odell describe this important issue very elegantly:

With OO techniques, the same paradigm is used for analysis, design, and implementation. The analyst identifies objects' types and inheritance, and thinks about events that change the state of objects. The designer adds detail to this model perhaps designing screens, user interaction, and client-server interaction. The thought process flows so naturally from analyst to design that it may be difficult to tell where analysis ends and design begins.

Corollary 5. Standardization

To reuse classes, you must have a good understanding of the classes in the object-oriented programming environment you are using. Most object-oriented systems, such as Smalltalk, Java, C++ , or PowerBuilder, come with several built-in class libraries. Similarly, object-oriented systems are like organic systems, meaning that they grow as

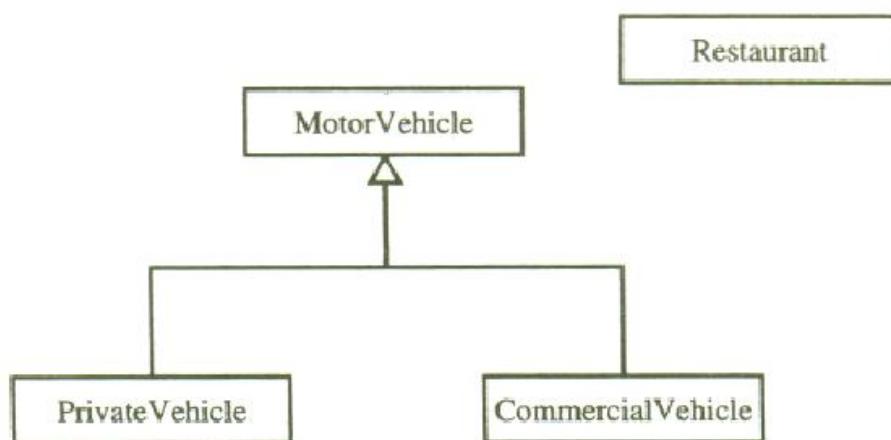
you create new applications. The knowledge of existing classes will help you determine what new classes are needed to accomplish the tasks and where you might inherit useful behavior rather than reinvent the wheel. However, class libraries are not always well documented or, worse yet, they are documented but not up to date. Furthermore, class libraries must be easily searched, based on users' criteria. For example, users should be able to search the class repository with commands like "show me all Facet classes." The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications.

Corollary 6. Designing with Inheritance

When you implement a class, you have to determine its ancestor, what attributes it will have, and what messages it will understand. Then, you have to construct its methods and protocols. Ideally, you will choose inheritance to minimize the amount of program instructions. Satisfying these constraints sometimes means that a class inherits from a superclass that may not be obvious at first glance.

For example, say, you are developing an application for the government that manages the licensing procedure for a variety of regulated entities. To simplify the example, focus on just two types of entities: motor vehicles and restaurants. Therefore, identifying classes is straightforward. All goes well as you begin to model these two portions of class hierarchy. Assuming that the system has no existing classes similar to a restaurant or a motor vehicle, you develop two classes, MotorVehicle and Restaurant.

Subclasses of the MotorVehicle class are Private Vehicle and CommercialVehicle. These are further subdivided into whatever level of specificity seems appropriate (see Figure). Subclasses of Restaurant are designed to reflect their own licensing procedures. This is a simple, easy to understand design, although somewhat limited in the reusability of the classes. For example, if in another project you must build a system that models a vehicle assembly plant, the classes from the licensing application are not appropriate, since these classes have instructions and data that deal with the legal requirements of motor vehicle license acquisition and renewal.



The initial single inheritance design.

In any case, the design is approved, implementation is accomplished, and the system goes into production. Now, here comes the event that every designer both knows well and dreads—when the nature of the real-world problem exceeds the bounds of the system, so far an elegant design. Say, six months later, while discussing some enhancements to the system with the right people (we learned how to identify right people in Chapter 6), one of them says, "What about coffee wagons, food trucks, and ice cream vendors? We're planning on licensing them as both restaurants and motor vehicles."

You know you need to redesign the application—but redesign how? The answer depends greatly on the inheritance mechanisms supported by the system's target language. If the language supports single inheritance exclusively, the choices are somewhat limited. You can choose to define a formal super class to both MotorVehicle and Restaurant, License, and move common methods and attributes from both classes into this License class (see Figure). However, the MotorVehicle and Restaurant classes have little in common, and for the most part, their attributes and methods are inappropriate for each other. For example, of what use is the gross weight of a diner or the address of a truck? This necessitates a very weak formal class (License) or numerous blocking behaviors in both MotorVehicle and Restaurant. This particular decision results in the least reusable classes and potentially extra code in several locations. So, let us try another approach. Alternatively, you could preserve the original formal classes, MotorVehicle and Restaurant. Next, define a FoodTruck class to descend from CommercialVehicle and copy enough behavior into it from the Restaurant class to support the application's requirements (see Figure).

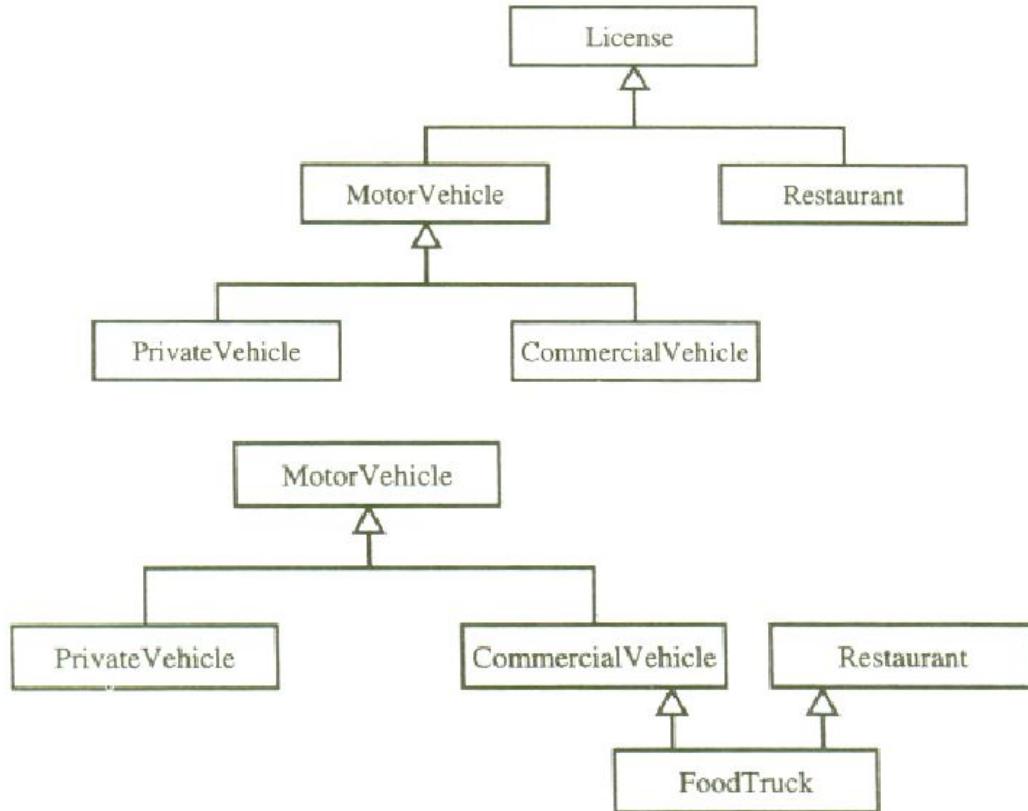
You can give FoodTruck copies of data and instructions from the Restaurant class that allow it to report on food type, health code categories, number of chefs and support staff, and the like. The class is not very reusable (Coad and Yourdon call it *cut-and-paste* reusability), but at least its extra code is localized, allowing simpler debugging and enhancement. Coad and Yourdon describe cut-and-paste type of reusability as follows :

This is better than no reuse at all, but is the most primitive form of reuse. The clerical cost of transcribing the code has largely disappeared with today's cut-and-paste text editors; nevertheless, the software engineer runs the risk of introducing errors during the copying (and modifications) of the original code. Worse is the configuration management problem: it is almost impossible for the manager to keep track of the multiple mutated uses of the original "chunk" of code.

If, on the other hand, the intended language supports multiple inheritance, another route can be taken, one that more closely models the real-world situation. In this case, you design a specialized class, FoodTruck, and specify dual ancestry. Our new class alternative seems to preserve the integrity and code bulk of both ancestors and does nothing that appears to affect their reusability.

In actuality, since we never anticipated this problem in the original design, there probably are instance variables and methods in both ancestors that share the same names. Most languages that support multiple inheritance handle these "hits" by giving precedence to the first ancestor defined. Using this mechanism, reworking will be required in the Food Truck descendant and, quite possibly, in both ancestors. It easily can become difficult to determine which method,

The single inheritance design modified to allow licensing food trucks.



Multiple inheritance design of the system structure.

in which class, affected an erroneously updated variable in an instance of a new descendant. The difficulties in maintaining such a design increase geometrically with the number of ancestors assigned to a given class. .

Achieving Multiple Inheritance in a Single Inheritance System *Single inheritance* means that each class has only a single superclass. This technique is used in Smalltalk and several other object-oriented systems. One result of using a single inheritance hierarchy is the absence of ambiguity as to how an object will respond to a given method; you simply trace up the class tree beginning with the object's class, looking for a method of the same name. However, languages like LISP or C++ have a multiple inheritance scheme whereby objects can inherit behavior from unrelated areas of the class tree. This could be desirable when you want a new class to behave similar to more than one existing class. However, multiple inheritance brings with it some complications, such as how to determine which behavior to get from which class, particularly when several

ancestors define the same method. It also is more difficult to understand programs written in a multiple inheritance system.

One way of achieving the benefits of multiple inheritance in a language with single inheritance is to inherit from the most appropriate class and add an object of another class as an attribute or aggregation. Therefore, as class designer, you have two ways to borrow existing functionality in a class. One is to inherit it, and the other is to use the instance of the class (object) as an attribute. This approach is described in the next section.

Avoiding Inheriting Inappropriate Behaviors Beginners in an object-oriented system frequently err by designing subclasses that inherit from inappropriate superclasses. Before a class inherits, ask the following questions: .Is the subclass fundamentally similar to its superclass (high inheritance coupling)? .Is it an entirely new thing that simply wants to borrow some expertise from its superclass (low inheritance coupling)?

Often you will find that the latter is true, and if so, you should add an attribute that incorporates the proposed superclass's behavior rather than an inheritance from the superclass. This is because inheritors of a class must be intimate with all its implementation details, and if some implementation is inappropriate, the inheritor's proper functioning could be compromised. For example, if FoodTruck inherits from both Restaurant and ConunercialVehicle classes, it might inherit a few inappropriate attributes and methods. A better approach would be to inherit only from CommercialVehicle and have an attribute of the type Restaurant (an instance of Restaurant class). In other words, Restaurant class becomes a-part -of FoodTruck class (see Figure).

16.4 LET US SUM UP

In this chapter, we looked at the object-oriented design process and design axioms. Integrating design axioms and corollaries with incremental and evolutionary styles of software development will provide you a powerful way for designing systems. During design, emphasis shifts from the application domain concept toward implementation, such as view (user interface) and access classes. The objects discovered during analysis serve as the framework for design.

The object-oriented design process consists of .Designing classes (their attributes, methods, associations, structures, and protocols) and applying design axioms. If needed, this step is repeated.

Designing the access layer. .Designing the user interface. .Testing user satisfaction and usability, based on the usage and use cases. .Iterating and refining the design.

The two design axioms are . Axiom 1. The independence axiom. Maintain the independence of components. .Axiom 2. The information axiom. Minimize the information content of the design. The six design corollaries are .Corollary 1. Uncoupled

design with less information content. .Corollary 2. Single purpose. .Corollary 3. Large number of simple classes. .Corollary 4. Strong mapping. .Corollary 5. Standardization. .Corollary 6. Design with inheritance.

16.5 POINTS FOR DISCUSSION

1. Evaluate the object oriented design axioms and corollaries.
2. Justify Coupling
3. Establish Corollary Designing with Inheritance

16.6 LESSON – END ACTIVITIES

1. Validate cohesion.
2. Evaluate achieving multiple inheritance in single inheritance system.

16.7 REFERENCES

1. Norman,Ronald- object oriented system analysis and design – Prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 17: CLASS VISIBILITY

CONTENTS

- 17.0 AIMS AND OBJECTIVES
- 17.1 INTRODUCTION
- 17.2 CLASS VISIBILITY: DESIGNING WELL-DEFINED PUBLIC, PRIVATE, AND PROTECTED PROTOCOLS
- 17.3 PRIVATE AND PROTECTED PROTOCOL LAYERS: INTERNAL
- 17.4 PUBLIC PROTOCOL LAYER: EXTERNAL
- 17.5 DESIGNING CLASSES: REFINING ATTRIBUTES
- 17.6 LET US SUM UP
- 17.7 POINTS FOR DISCUSSION
- 17.8 LESSON – END ACTIVITIES
- 17.9 REFERENCES

17.0 AIM AND OBJECTIVE

You should be able identify the class visibility.

17.1 INTRODUCTION

Object-oriented design requires taking the objects identified during object-oriented analysis and designing classes to represent them. As a class designer, you have to know the specifics of the class you are designing and be aware of how that class interacts with other classes. Once you have identified your classes and their interactions, you are ready to design classes.

Underlying the functionality of any application is the quality of its design. In this chapter, we look at guidelines and approaches to use in designing classes and their methods. Although the design concepts to be discussed in this chapter are general, we will concentrate on designing the business classes (see Chapter 6). The access and view layer classes will be described in the subsequent chapters. However, the same concepts will apply to designing access and view layer classes.

17.2 CLASS VISIBILITY: DESIGNING WELL-DEFINED PUBLIC, PRIVATE, AND PROTECTED PROTOCOLS

In designing methods or attributes for classes, you are confronted with two problems. One is the *protocol*, or interface to the class operations and its visibility; and

the other is how it is implemented. Often the two have very little to do with each other. For example, you might have a class Bag for collecting various objects that counts multiple occurrences of its elements. One implementation decision might be that the Bag class uses another class, say, Dictionary (assuming that we have a class Dictionary), to actually hold its elements. Bags and dictionaries have very little in common, so this may seem curious to the outside world. Implementation, by definition, is hidden and off limits to other objects. The class's protocol, or the messages that a class understands, on the other hand, can be hidden from other objects (private protocol) or made available to other objects (public protocol). Public protocols define the functionality and external messages of an object; private protocols define the implementation of an object (see Figure)

Public protocols define the functionality and external messages of an object, while private protocols define the implementation of an object. It is important in object-oriented design to define the public protocol between the associated classes in the application. This is a set of messages that a class of a certain generic type must understand, although the interpretation and implementation of each message is up to the individual class.

A class also might have a set of methods that it uses only internally, messages to itself. This, the *private protocol (visibility)* of the class, includes messages that normally should not be sent from other objects; it is accessible only to operations of that class. In private protocol, only the class itself can use the method. The *public protocol (visibility)* defines the stated behavior of the class as a citizen in a population and is important information for users as well as future descendants, so it is accessible to all classes. If the methods or attributes can be used by the class itself or its subclasses, a protected protocol can be used. In a *protected protocol (visibility)*, subclasses can use the method in addition to the class itself.

Lack of a well-designed protocol can manifest itself as encapsulation leakage. The problem of *encapsulation leakage* occurs when details about a class's internal implementation are disclosed through the interface. As more internal details become visible, the flexibility to make changes in the future decreases. If an implementation is completely open, almost no flexibility is retained for future carefully controlled. However, do not make such a decision lightly because that could impact the flexibility and therefore the quality of the design.

For example, public or protected methods that can access private attributes can reveal an important aspect of your implementation. If anyone uses these functions and you change their location, the type of attribute, or the protocol of the method, this could make the client application inoperable. Design the interface between a superclass and its subclasses just as carefully as the class's interface to clients; this is the contract between the super- and subclasses. If this interface is not designed properly, it can lead to violating the encapsulation of the superclass. The protected portion of the class interface can be accessed only by subclasses. This feature is helpful but cannot express the totality of the relationship between a class and its subclasses.

Other important factors include which functions might or might not be overridden and how they must behave. It also is crucial to consider the relationship among methods. Some methods might need to be overridden in groups to preserve the class's semantics. The bottom line is this: Design your interface to subclasses so that a subclass that uses every supported aspect of that interface does not compromise the integrity of the public interface. The following paragraphs summarize the differences between these layers.

17.3 PRIVATE AND PROTECTED PROTOCOL LAYERS: INTERNAL

Items in these layers define the implementation of the object. Apply the design axioms and corollaries, especially Corollary 1 (uncoupled design with less information content, see Chapter 9) to decide what should be private: what attributes (instance variables)? What methods? Remember, highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects.

17.4 PUBLIC PROTOCOL LAYER: EXTERNAL

Items in this layer define the functionality of the object. Here are some things to keep in mind when designing class protocols: Good design allows for polymorphism. Not all protocol should be public; again apply design axioms and corollaries 1.

The following key questions must be answered:

What are the class interfaces and protocols?

What public (external) protocol will be used or what external messages must the system understand? . What private or protected (internal) protocol will be used or what internal messages or messages from a subclass must the system understand?

17.5 DESIGNING CLASSES: REFINING ATTRIBUTES

Attributes identified in object-oriented analysis must be refined with an eye on implementation during this phase. In the analysis phase, the name of the attribute was sufficient. However, in the design phase, detailed information must be added to the model (especially, that defining the class attributes and operations). The main goal of this activity is to refine existing attributes (identified in analysis) or add attributes that can elevate the system into implementation.

17.5.1 Attribute Types

The three basic types of attributes are

1. Single-value attributes.
2. Multiplicity or multivalue attributes.
3. Reference to another object, or instance connection.

Attributes represent the state of an object. When the state of the object changes, these changes are reflected in the value of attributes. The single-value attribute is the most common attribute type. It has only one value or state. For example, attributes such as name, address, or salary are of the single-value type.

The multiplicity or multivalue attribute is the opposite of the single-value attribute since, as its name implies, it can have a collection of many values at any point in time . For example, if we want to keep track of the names of people who have called a customer support line for help, we must use the multivalues attributes. Instance connection attributes are required to provide the mapping needed by an object to fulfill its responsibilities, in other words, instance connection model association. For example, a person might have one or more bank accounts. A person has zero to many instance connections to Account{s}. Similarly, an Account can be assigned to one or more persbns (i.e., joint account). Therefore, an Account also has zero to many instance connections to Person{s}).

17.5.2 UML Attribute Presentation

As discussed in Chapter 5, OCL can be used during the design phase to define the class attributes. The following is the attribute presentation suggested by UML

: visibility name: type-expression =initial- value

Where visibility is one of the following:

+ public visibility (accessibility to all classes).

protected visibility (accessibility to subclasses and operations of the class).

- private visibility (accessibility only to operations of the class).

Type-expression is a language-dependent specification of the implementation type of an attribute.

Initial-value is a language-dependent expression for the initial value of a newly created object. The initial value is optional. For example, + size: length = 100

The UML style guidelines recommend beginning attribute names with a lowercase letter.

In the absence of a multiplicity indicator (array), an attribute holds exactly one value. Multiplicity may be indicated by placing a multiplicity indicator in brackets after attribute name; for example,

names[1..]: String

points[2.. *]: Point

The multiplicity of 0..1 provides the possibility of null values: the absence of a value, as opposed to a particular value from the range. For example, the following declaration permits a distinction between the null value and an empty string:

name[0..1]: String

REFINING ATTRIBUTES FOR THE VIANET BANK OBJECTS

In this section, we go through the ViaNet bank ATM system classes and refine the attributes identified during object-oriented analysis.

Refining Attributes for the BankClient Class

During object-oriented analysis, we identified the following attributes:

firstName

lastName

pinNumber
cardNumber

At this stage, we need to add more information to these attributes, such as visibility and implementation type. Furthermore, additional attributes can be identified during this phase to enable implementation of the class:

```
#firstName: String
#lastName: String
#pinNumber: String
#cardNumber: String
#account: Account (instance connection)
```

In Chapters we identified an association between the BankClient and the Account classes (see Figure). To design this association, we need to add an *account* attribute of type Account, since the BankClient needs to know about his or her account and this attribute can provide such information for the BankClient class. This is an example of instance connection, where it represents the association between the BankClient and the Account objects. All the attributes have been given *protected visibility*.

Refining Attributes for the Account Class

Here is the refined list of attributes for the Account class:

```
#number: String
#balance: float
#transaction: Transaction (This attribute is needed for implementing the association
between the Account and Transaction classes.)
#bankClient: BankClient (This attribute is needed for implementing the association
between the Account and BankClient classes.)
```

At this point we must make the Account class very general, so that it can be reused by the checking and savings accounts.

Refining Attributes for the Transaction Class

The attributes for the Transaction class are these:

```
#transID: String
#transDate: Date
#transTime: Time
#transType: String
#amount: float
#postBalance: float
```

Problem

Why do we not need the account attribute for the Transaction class? Hint: Do transaction objects need to know about account objects?

Refining Attributes for the ATM Machine Class

The ATMMachine class could have the following attributes:

```
#address: String
#state: String
```

Refining Attributes for the CheckingAccount Class

Add the *savings* attribute to the class. The purpose of this attribute is to implement the association between the CheckingAccount and SavingsAccount classes.

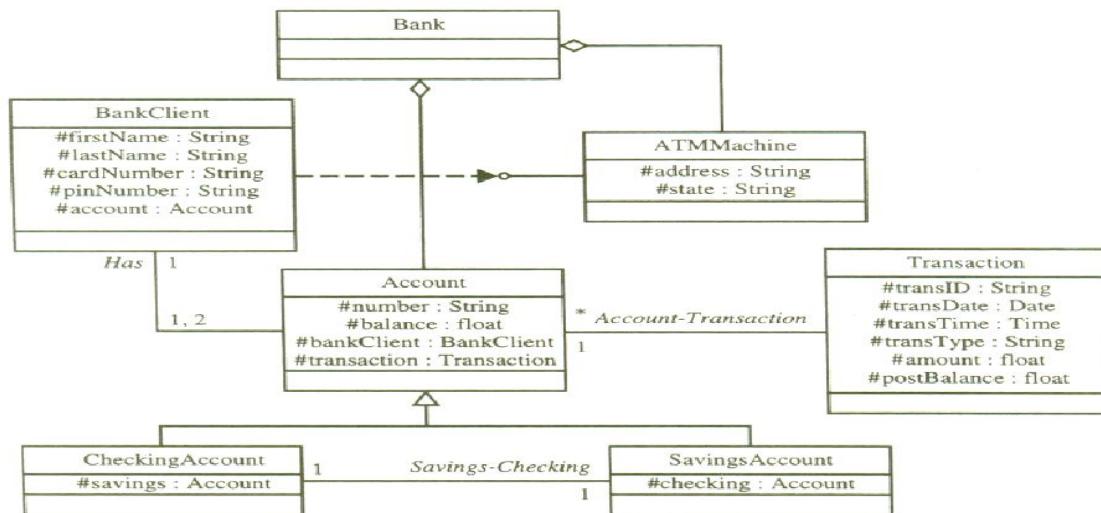
Refining Attributes for the SavingsAccount Class

Add the *checking* attribute to the class. The purpose of this attribute is to implement the association between the SavingsAccount and CheckingAccount classes. Figure 10-2 (see Chapter 8) shows a more complete UML class diagram for the bank system. At this stage, we also need to add a very short description of each attribute or certain attribute constraints. For example,

Class ATMMachine

`#address: String (The address for this ATM machine.)`

`#state: String (The state of operation for this ATM machine, such as running, off, idle, out of money, security alarm.)`



A more complete UML class diagram for the ViaNet bank system.

17.6 LET US SUM UP

The single most important activity in designing an application is coming up with a set of classes that work together to provide the needed functionality. After all, underlying the functionality of any application is the quality of its design. This chapter concentrated on the first step of the object-oriented design process, which consists of applying the design axioms and corollaries to design classes, their attributes, methods, associations, structures, and protocols; then, iterating and refining.

During the analysis phase, the name of the attribute should be sufficient. However, during the design phase, detailed information must be added to the model (especially, definitions of the class attributes and operations). The UML provides a language to do just that. The rules and semantics of the UML can be expressed in English, in a form known as *object constraint language* (OCL). OCL is a specification language that uses simple logic for specifying the properties of a system.

17.7 POINTS FOR DISCUSSION

1. Justify Private and Protected Protocol Layers Internal
2. Validate which corollary would you apply to design well defined public, private and protected protocols.
3. Justify Private and Protected Protocol Layers External
4. Valuate Attribute Types

17.8 LESSON – END ACTIVITIES

1. Discuss about the characteristics of bad design.
2. Analyse the UML attribute presentation.

17.9 REFERENCES

1. Norman,Ronald- object oriented system analysis and design –printice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press

LESSON – 18: DESIGNING METHODS AND PROTOCOLS

CONTENTS

- 18.0 AIMS AND OBJECTIVES
- 18.1 INTRODUCTION
- 18.2 DESIGNING METHODS AND PROTOCOLS
- 18.3 DESIGN ISSUES: AVOIDING DESIGN PITFALLS
- 18.4 UML OPERATION PRESENTATION
- 18.5 DESIGNING METHODS FOR THE VIANET BANK OBJECTS
- 18.6 BANKCLIENT CLASS VERIFYPASSWORD METHOD
- 18.7 ACCOUNT CLASS DEPOSIT METHOD
- 18.8 ACCOUNT CLASS WITHDRAW METHOD
- 18.9 ACCOUNT CLASS CREATETRANSACTION METHOD
- 18.10 CHECKING ACCOUNT CLASS WITHDRAW METHOD
- 18.11 PACKAGES AND MANAGING CLASSES
- 18.12 LET US SUM UP
- 18.13 POINTS FOR DISCUSSION
- 18.14 LESSON – END ACTIVITIES
- 18.15 REFERENCES

18.0 AIMS AND OBJECTIVES

You should be able to design methods and protocols.

18.1 INTRODUCTION

The main goal of this activity is to specify the algorithm for methods identified so far. Once you have designed your methods in some formal structure such as UML activity diagrams with an OCL description, they can be converted to programming language manually or in automated fashion

18.2 DESIGNING METHODS AND PROTOCOLS

A class can provide several types of methods :

.*Constructor*. Method that creates instances (objects) of the class.

.*Destructor*. The method that destroys instances.

.Conversion method. The method that converts a value from one unit of measure to another.

.Copy method. The method that copies the contents of one instance to another instance. .

.Attribute set. The method that sets the values of one or more attributes.

.Attribute get. The method that returns the values of one or more attributes.

.I/O methods. The methods that provide or receive data to or from a device.

.Domain specific. The method specific to the application.

Corollary 1, that in designing methods and protocols you must minimize the complexity of message connections and keep as low as possible the number of messages sent and received by an object. Your goal should be to maximize cohesiveness among objects and software components to improve coupling, because only a minimal amount of essential information should be passed between components. Abstraction leads to simplicity and straightforwardness and, at the same time, Increases class versatility. The requirement of simplification, while retaining functionality, seems to lead to increased utility. Here are five rules :

1. If it looks messy, then it's probably a bad design.
2. If it is too complex, then it's probably a bad design.
3. If it is too big, then it's probably a bad design.
4. If people don't like it, then it's probably a bad design.
5. If it doesn't work, then it's probably a bad design.

18.3 DESIGN ISSUES: AVOIDING DESIGN PITFALLS

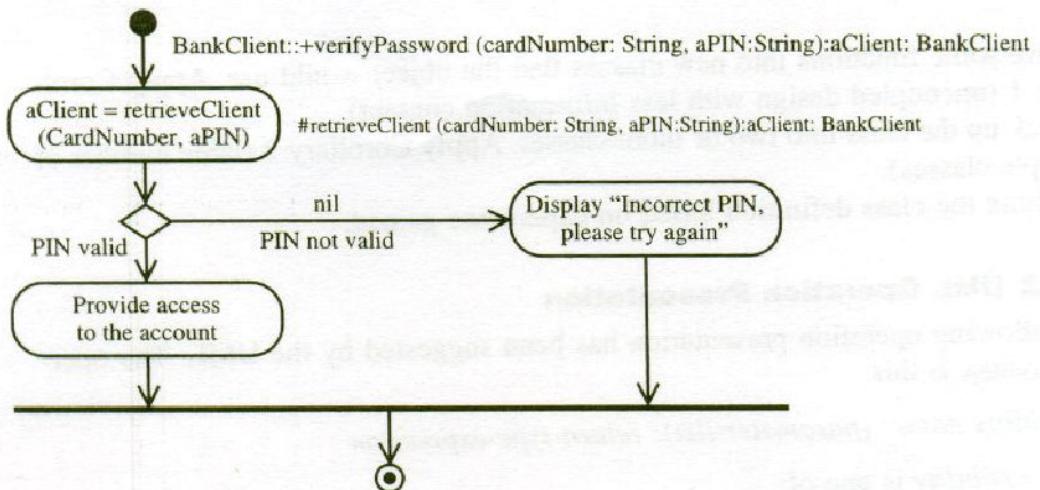
As described it is important to apply design axioms to avoid common design problems and pitfalls. For example, we learned that it is much better to have a large set of simple classes than a few large, complex classes. A common occurrence is that, in your first attempt, your class might be too big and therefore more complex than it needs to be. Take the time to apply the design axioms and corollaries, then critique what you have proposed. You may find you can gather common pieces of expertise from several classes, which in itself becomes another "peer" class that the others consult; or you might be able to create a superclass for several classes that gathers in a single place very similar code. Your goal should be maximum reuse of what you have to avoid creating new classes as much as possible.

Take the time to think in this way-good news, this gets easier over time. Lost object focus is another problem with class definitions. A meaningful class definition starts out simple and clean but, as time goes on and changes are made, becomes larger and larger, with the class identity becoming harder to state concisely (Corollary 2). This happens when you keep making incremental changes to an existing class. If the class does not quite handle a situation, someone adds a tweak to its description. When the next problem comes up, another tweak is added. Or, when a new feature is requested, another tweak is added, and so on.

Apply the design axioms and corollaries, such as Corollary 2 (which states that each class must have a single, clearly defined purpose). When you document, you easily

should be able to describe the purpose of a class in a few sentences. These problems can be detected early on. Here are some of the warning signs that something is going amiss. There are bugs because the internal state of an object is too hard to track and solutions consist of adding patches. Patches are characterized by code that looks like this: "If this is the case, then force that to be true" or "Do this just in case we need to" or "Do this before calling that function, because it expects this."

Some possible actions to solve this problem are these: .Keep a careful eye on the class design and make sure that an object's role remains well defined. If an object loses focus, you need to modify the design. Apply Corollary 2 (single purpose).



An activity diagram for the `BankClient` class `verifyPassword` method, using OCL to describe the diagram. The syntax for describing a class's method is Class name::methodName. We postpone design of the `retrieveClient` to Chapter 11, Section 11.10, Designing Access layer Classes.

BankClient Class VerifyPassword Method

The following describes the `verifyPassword` service in greater detail. A client PIN code is sent from the `ATMMachine` object and used as an argument in the `verifyPassword` method. The `verify Password` method retrieves the client record and checks the entered PIN number against the client's PIN number. If they match, it allows the user to proceed. Otherwise, a message sent to the `ATMMachine` displays "Incorrect PIN, please try again" (see Figure).

The `verifyPassword` methods' performs first creates a bank client object and attempts to retrieve the client data based on the supplied card and PIN numbers. At this stage, we realize that we need to have another method, `retrieveClient`. The `retrieveClient` method takes two arguments, the card number and a PIN number, and returns the client object or "nil" if the password is not valid. We postpone design of the `retrieveClient` method to Chapter 11 (Section 11.10, designing the access layer classes).

Account Class Deposit Method

The following describes the deposit service in greater detail. An amount to be deposited is sent to an account object and used as an argument to the deposit service. The account adjusts its balance to its current balance plus the deposit amount. The account object records the deposit by creating a transaction object containing the date and time, posted balance, and transaction type and amount (see Figure).

Once again we have discovered another method, updateClient. This method, as the name suggests, updates client data. We postpone design of the updateClient method to the (designing the access layer classes). .Move some functions into new classes that the object would use. Apply Corollary I (uncoupled design with less infonnation content). .Break up the class into two or more classes. Apply Corollary 3 (large number of simple classes). .Rethink the class definition based on experience gained.

18.4 UML OPERATION PRESENTATION

The following operation presentation has been suggested by the UML. The operation syntax is this:

visibility name: (parameter-list): return-type-expression

Where *visibility* is one of:

public visibility (accessibility to all classes).

protected visibility (accessibility to subclasses and operations of the class).

private visibility (accessibility only to operations of the class).

Here, *name* is the name of the operation.

Parameter-list: is a list of parameters, separated by commas, each specified by

name: type-expression = default value (where *name* is the name of the parameter, *type-expression* is the language-dependent specification of an implementation type, and *default-value* is an optional value).

Return-type-expression: is a language-dependent specification of the implementation of the value returned by the method. If *return-type* is omitted, the operation does not return a value; for example,

+

#

+getName(): aName

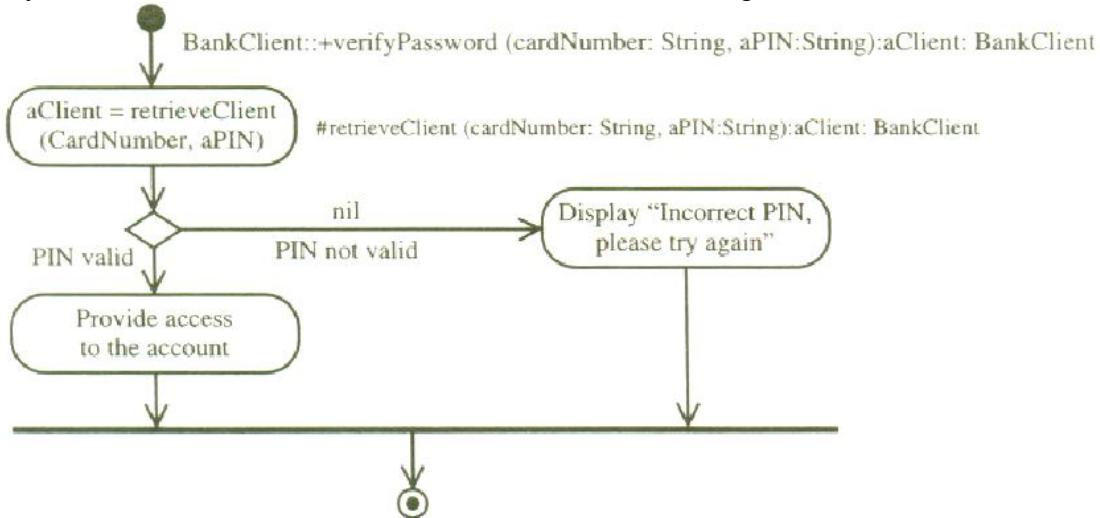
+getAccountNumber (account: type): account Number

The UML guidelines recommend beginning operation names with a lowercase letter.

18.5 DESIGNING METHODS FOR THE VIANET BANK OBJECTS

At this point, the design of the bank business model is conceptually complete. You have identified the objects that make up your business layer, as well as what services they provide. All that remains is to design methods, the user interface, database access, and implement the methods using any object-oriented programming language. To keep the book language independent, we represent the methods' algorithms with UML activity diagrams, which very easily can be translated into any language. this phase prepares the

system for the implementation. The actual coding and implementation (although they are beyond the scope of this book) should be relatively easy and, for the most part, can be automated by using CASE tools. This is because we know what we want to code. It is always difficult to code when we have no clear understanding of what we want to do.



An activity diagram for the BankClient class verifyPassword method, using OCL to describe the diagram. The syntax for describing a class's method is Class name::methodName. We postpone design of the retrieveClient to Chapter 11, Section 11.10, Designing Access Layer Classes.

18.6 BANKCLIENT CLASS VERIFYPASSWORD METHOD

The following describes the verifyPassword service in greater detail. A client PIN code is sent from the ATMMachine object and used as an argument in the verifyPassword method. The verify Password method retrieves the client record and checks the entered PIN number against the client's PIN number. If they match, it allows the user to proceed. Otherwise, a message sent to the ATMMachine displays "Incorrect PIN, please try again" (see Figure).

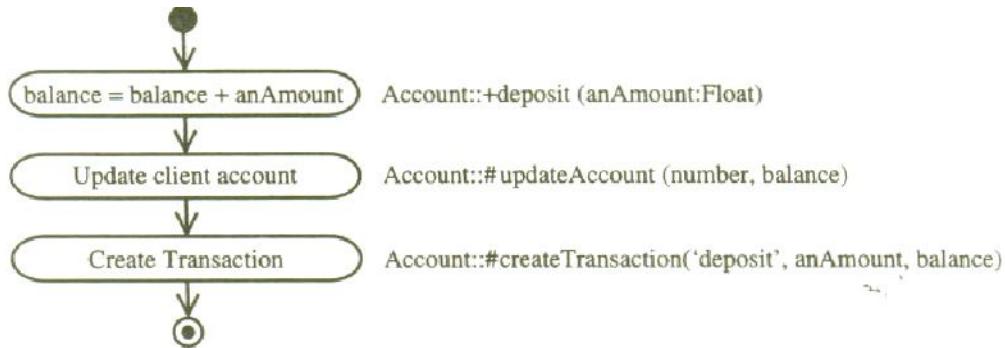
The verifyPassword methods' performs first creates a bank client object and attempts to retrieve the client data based on the supplied card and PIN numbers. At this stage, we realize that we need to have another method, retrieveClient. The retrieveClient method takes two arguments, the card number and a PIN number, and returns the client object or "nil" if the password is not valid. We postpone design of the retrieveClient method to Chapter 11 (Section 11.10, designing the access layer classes).

18.7 ACCOUNT CLASS DEPOSIT METHOD

The following describes the deposit service in greater detail. An amount to be deposited is sent to an account object and used as an argument to the deposit service. The account adjusts its balance to its current balance plus the deposit amount. The account object records the deposit by creating a transaction object

containing the date and time, posted balance, and transaction type and amount (see Figure).

Once again we have discovered another method, updateClient. This method, as the name suggests, updates client data. We postpone design of the updateClient method to the Chapter 11 (designing the access layer classes).



An activitydiagram for the Account class deposit method.

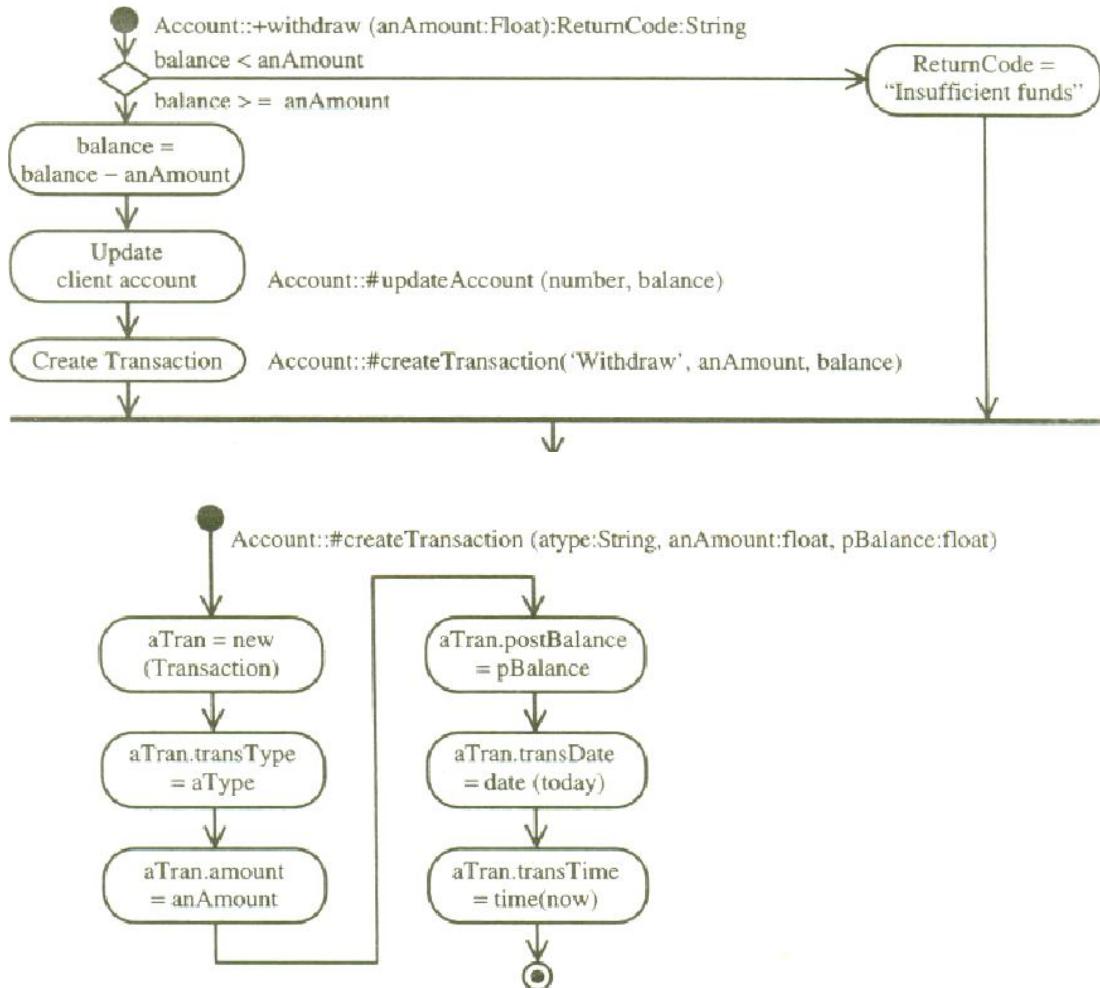
18.8 ACCOUNT CLASS WITHDRAW METHOD

This is the generic withdrawal method that simply withdraws funds if they are available. It is designed to be inherited by the CheckingAccount and SavingsAccount classes to implement automatic funds transfer. The following describes the withdraw method. An amount to be withdrawn is sent to an account object and used as the argument to the withdraw service. The account checks its balance for sufficient funds. If enough funds are available, the account makes the withdrawal and updates its balance; otherwise, it returns an error, saying "insufficient funds." If successful, the account records the withdrawal by creating a transaction object containing date and time, posted balance, and transaction type and amount (see Figure).

18.9 ACCOUNT CLASS CREATETRANSACTION METHOD

The createTransaction method generates a record of each transaction performed against it. The description is as follows. Each time a successful transaction is

An activity diagram for the Account class withdraw method.



An activity diagram for the Account class createTransaction method.

perfonned against an account, the account object creates a transaction object to record it. Arguments into this service include transaction type (withdrawal or deposit), the transaction amount, and the balance after the transaction. The account creates a new transaction object and sets its attributes to the desired infonnation. Add this description to the *create Transaction*'s description field (see Figure).

18.10 CHECKING ACCOUNT CLASS WITHDRAW METHOD

This is the checking account-specific version of the withdrawal service. It takes into consideration the possibility of withdrawing excess funds from a companion savings account. The description is as follows. An amount to be withdrawn is sent to a checking account and used as the argument to the withdrawal service. If the account has insufficient funds to cover the amount but has a companion savings account, it tries to withdraw the excess from there. If the companion account has insufficient funds, this method returns the appropriate error message. If the companion account has enough

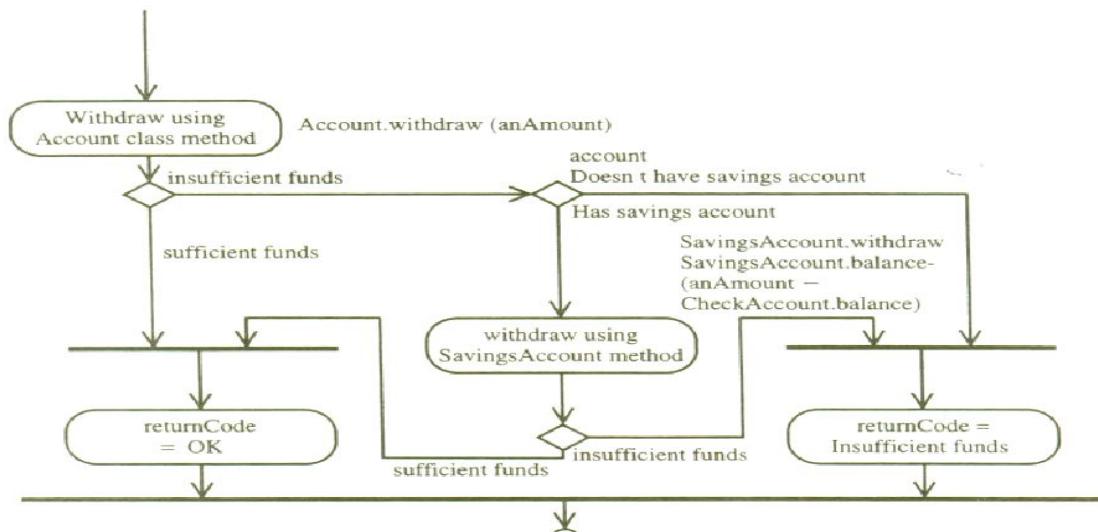
funds, the excess is withdrawn from there, and the checking account balance goes to zero (0). If successful, the account records the withdrawal by creating a transaction object containing the date and time, posted balance, and transaction type and amount (see Figure).

ATMMachine Class Operations

The ATMMachine class provides an interface (view) to the bank system. We postpone designing this class to Chapter 12.

18.11 PACKAGES AND MANAGING CLASSES

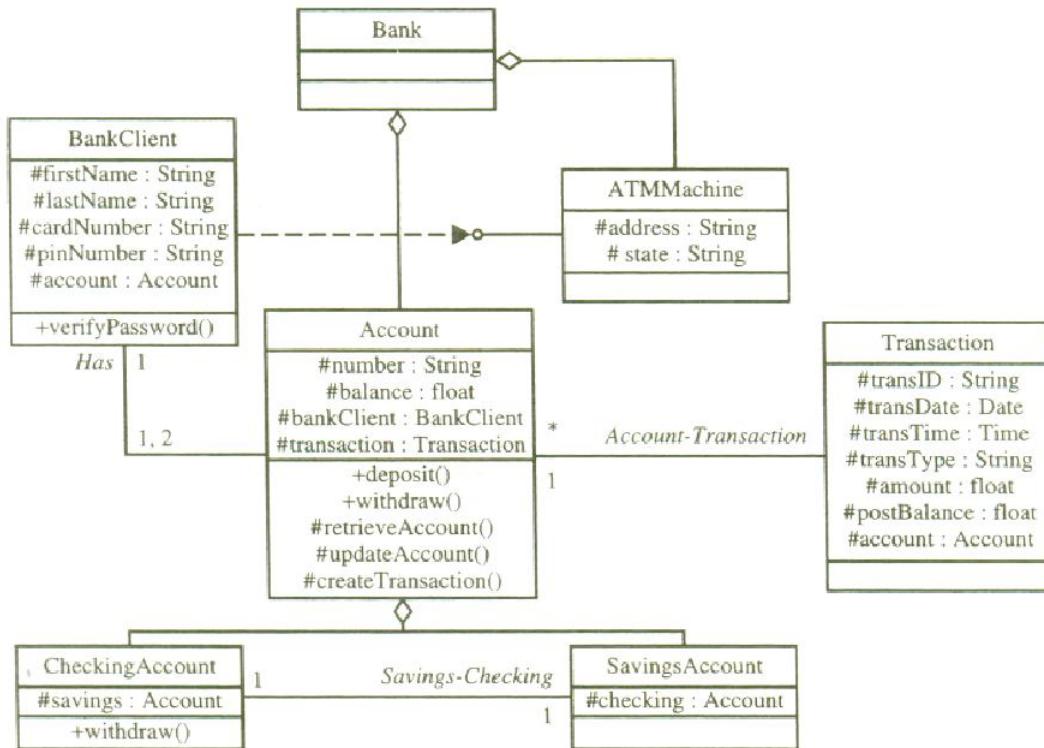
A package groups and manages the modeling elements, such as classes, their associations, and their structures. Packages themselves may be nested within other packages. A package may contain both other packages and ordinary model



elements. The entire system description can be thought of as a single high-level subsystem package with everything else in it. All kinds of UML model elements and diagrams can be organized into packages. For example, some packages may contain groups of classes and their relationships, subsystems, or models. A package provides a hierarchy of different system components and can reference other packages.

For example, the bank system can be viewed as a package that contains other packages, such as Account package, Client package, and so on. Classes can be packaged based on the services they provide or grouped into the business classes, access classes, and view classes (see Figure). Furthermore, since packages own model elements and model fragments, they can be used by CASE tools as the basic storage and access control.

In Chapter 5, we learned that a package is shown as a large rectangle with a *small rectangular tab*. If the contents' of the package are shown, then the name of the package may be placed within the tab. A keyword string may be placed above the package name. The keywords *subsystem* and *model* indicate that the package is a meta-model subsystem or model. The visibility of a package element outside the package may be indicated by preceding the name of the element by a visibility symbol (+ for public, - for private, # for protected). If the element is in an inner package, its visibility as exported by the outer package is obtained by combining the visibility of an element within the package with the visibility of the package itself: The most restrictive visibility prevails.



More complete UML class diagram for the ViaNet bank ATM system. Note that the method parameter list is not shown.

Relationships may be drawn between package symbols to show relationships between at least some of the elements in the packages. In particular, dependency between packages implies one or more dependencies among the elements. Figure depicts an even more complete class diagram the ViaNet bank ATM system.

18.12 LET US SUM UP

The UML package is a grouping of model elements. It can organize the modeling elements including classes. Packages themselves may be nested within other packages. A package may contain both other packages and ordinary model elements. The entire system description can be thought of as a single, high-level subsystem package with everything else in it.

Object-oriented design is an iterative process. Designing is as much about discovery as construction. Do not be afraid to change a class design, based on experience gained, and do not be afraid to change it a second, third, or fourth time. At each iteration, you can improve the design. However, the trick is to fix the design as early as possible; redesigning late in the development cycle is problematic and may be impossible.

18.13 POINTS FOR DISCUSSION

1. Validate how to avoiding design pitfalls
2. Validate UML operation presentation
3. Analyze account class deposit method

18.14 LESSON – END ACTIVITIES

1. Analyse account class create transaction method.
2. Discuss about packages and managing classes.

18.15 REFERENCES

1. Norman,Ronald- object oriented system analysis and design – Prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 19: OBJECT STORE AND PERSISTENCE

CONTENTS

- 19.0 AIMS AND OBJECTIVES
- 19.1 INTRODUCTION
- 19.2 OBJECT STORE AND PERSISTENCE: AN OVERVIEW
- 19.3 DATABASE MANAGEMENT SYSTEMS
- 19.4 LOGICAL AND PHYSICAL DATABASE ORGANIZATION AND ACCESS
CONTROL
- 19.5 OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS: THE PURE
WORLD
- 19.6 LET US SUM UP
- 19.7 POINTS FOR DISCUSSION
- 19.8 LESSON – END ACTIVITIES
- 19.9 REFERENCES

19.0 AIMS AND OBJECTIVES

To understand
object store and persistence
database management system

19.1 INTRODUCTION

A fundamental characteristic of the database approach is that the DBMS contains not only the data but a complete definition of the data formats it manages. This description is known as the *schema*, or *meta-data*, and contains a complete definition of the data formats, such as the data structures, types, and constraints.

19.2 OBJECT STORE AND PERSISTENCE: AN OVERVIEW

A program will create a large amount of data throughout its execution. Each item of data will have a different lifetime. Atkinson et al. describe six broad categories for the lifetime of data:

1. Transient results to the evaluation of expressions.
2. Variables involved in procedure activation (parameters and variables with a localized scope).
3. Global variables and variables that are dynamically allocated.

4. Data that exist between the executions of a program.
5. Data that exist between the versions of a program.
6. Data that outlive a program.

The first three categories are *transient data*, data that cease to exist beyond the lifetime of the creating process. The other three are nontransient, or *persistent*, data.

Typically, programming languages provide excellent, integrated support for the first three categories of transient data. The other three categories can be supported by a DBMS, or a file system.

The same issues also apply to objects; after all, objects have a lifetime, too. They are created explicitly and can exist for a period of time (during the application session). However, an object can persist beyond application session boundaries, during which the object is stored in a file or a database. A file or a database can provide a longer life for objects-longer than the duration of the process in which they were created. From a language perspective, this characteristic is called *persistence*. Essential elements in providing a persistent store are :

Identification of persistent objects or reachability (object ID).

Properties of objects and their interconnections. The store must be able to coherently manage nonpointer and pointer data (i.e., interobject references).

Scale of the object store. The object store should provide a conceptually infinite store.

Stability. The system should be able to recover from unexpected failures and return the system to a recent self-consistent state. This is similar to the reliability requirements of a DBMS, object-oriented or not.

Having separate methods of manipulating the data presents many problems.

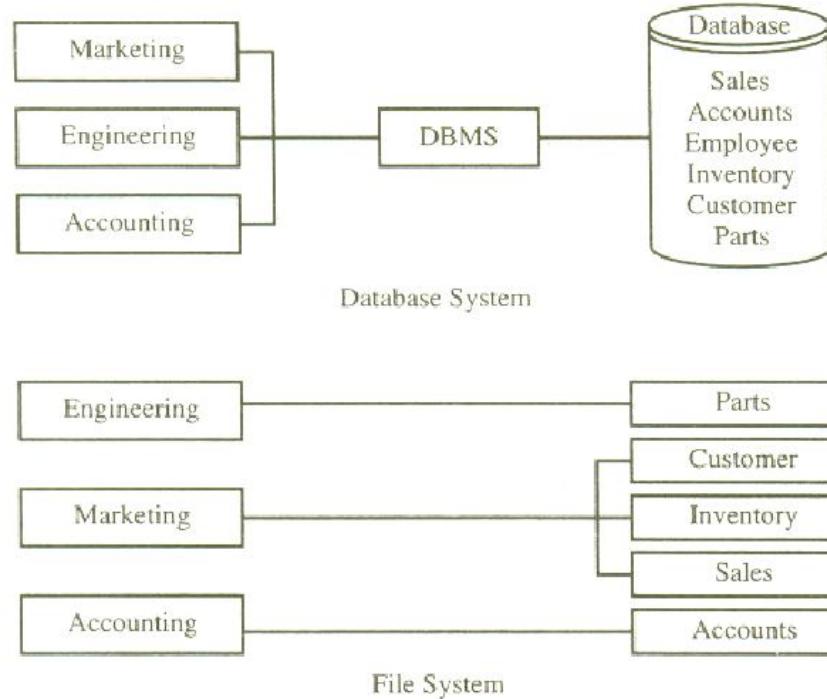
Atkinson et al. [I] claim that typical programs devote significant amounts of code to transferring data to and from the file system or DBMS. Additionally, the use of these external storage mechanisms leads to a variety of technical issues, which will be examined in the following sections. '

19.3 DATABASE MANAGEMENT SYSTEMS

Databases usually are large bodies of data seen as critical resources to a company. As mentioned earlier, a DBMS is a set of programs that enable the creation and maintenance of a collection of related data. DBMSs have a number of properties that distinguish them from the file-based data management approach. In traditional file processing, each application defines and implements the files it requires. Using a database approach, a single repository of data is maintained, which can be defined once and subsequently accessed by various users (see Figure).

In traditional file processing applications, such meta-data usually are encapsulated in the application programs themselves. In DBMS, the format of the metadata is independent of any particular application data structure; therefore, it will

Database system vs. file system.



provide a generic storage management mechanism. Another advantage of the database approach is program-data independence. By moving the meta-data into an external DBMS, a layer of insulation is created between the applications and the stored data structures. This allows any number of applications to access the data in a simplified and uniform manner.

Database Views

The DBMS provides the database users with a conceptual representation that is independent of the low-level details (physical view) of how the data are stored. The database can provide an abstract data model that uses logical concepts such as field, records, and tables and their interrelationships. Such a model is understood more easily by the user than the low-level storage concepts.

This abstract data model also can facilitate multiple views of the same underlying data. Many applications will use the same shared information but each will be interested in only a subset of the data. The DBMS can provide multiple virtual views of the data that are tailored to individual applications. This allows the convenience of a private data representation with the advantage of globally managed information.

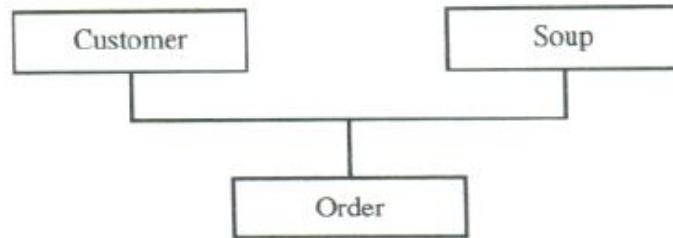
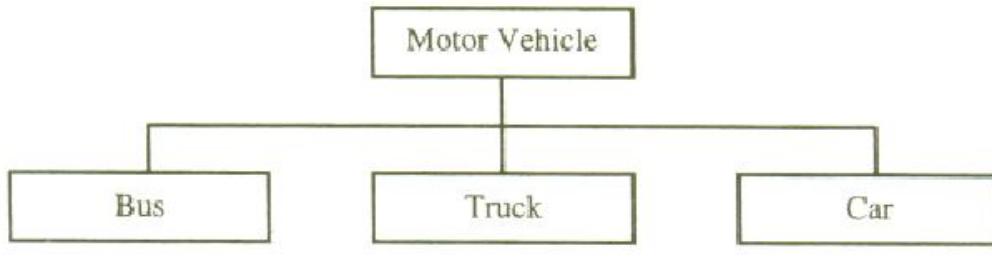
Database Models

A database model is a collection of logical constructs used to represent the data structure and data relationships within the database. Basically, database models may be grouped into two categories: conceptual models and implementation models. The

conceptual model focuses on the logical nature of that data presentation. Therefore, the conceptual model is concerned with *what* is represented in the database and the implementation model is concerned with *how* it is represented .

Hierarchical Model The hierarchical model represents data as a singlerooted tree. Each node in the tree represents a data object and the connections represent a parent-child relationship. For example, a node might be a record containing information about Motor vehicle and its child nodes could contain a record about Bus parts (see Figure). Interestingly enough, a hierarchical model resembles super-sub relationship of objects.

A hierarchical model. The top layer, the root, is perceived as the parent of the segment directly below it. In this case motor vehicle is the parent of Bus, Truck, and Car. A segment also is called a node. The segments below another node are the children of the node above them. Bus, Truck, and Car are the children of MotorVehicle.



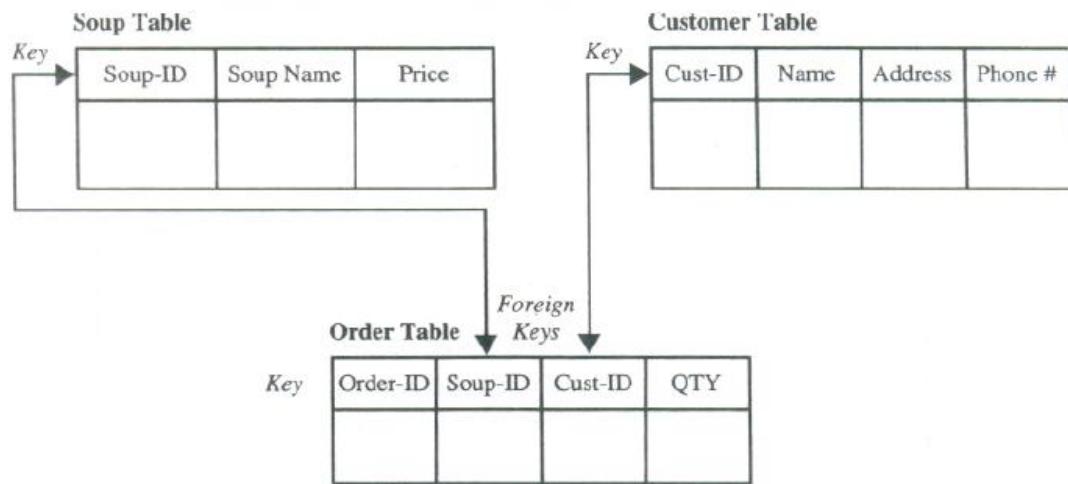
Network model. An Order contains data from both Customer and Soup.

Network Model A network database model is similar to a hierarchical database, with one distinction. Unlike the hierarchical model, a network model's record can have more than one parent. For example, in Figure , an Order contains data from the Soup and Customer nodes.

Relational Model Of all the database models, the relational model has the simplest, most uniform structure and is the most commercially widespread. The primary concept in this database model is the relation, which can be thought of as a table. The columns of each table are attributes that define the data or value domain for entries in that column. The rows of each table are *tuples* representing individual data objects being stored. A

relational table should have only one primary key. A *primary key* is a combination of one or more attributes whose *value* unambiguously locates each row in the table. In Figure , Soup-ID, Cust-ill, and Order-ill are primary keys in Soup, Customer, and Order tables. A *foreign key* is a primary key of one table that is embedded in another table to link the tables. In Figure , Soup-ill and Cust-ill are foreign keys in the Order table.

The figure depicts primary and foreign keys in a relation database. Soup-ID is a primary key of the Soup table, Cust-ID is a primary key of the Customer table, and Order-ID is a primary key of the Order table. Soup-ID and Cust-ID are foreign keys in the Order table.



Database Interface

The interface on a database must include a data definition language (DDL), a query, and data manipulation language (DML). These languages must be designed to fully reflect the flexibility and constraints inherent in the data model. Database systems have adopted two approaches for interfaces with the system. One is to embed a database language, such as structured query language (SQL), in the host programming language. This approach is a very popular way of defining and designing a database and its schema, especially with the popularity of languages such as SQL, which has become an industry standard for defining databases. The problem with this approach is that application programmers have to learn and use two different languages. Furthermore, the application programmers have to negotiate the differences in the data models and data structures allowed in both languages .

Another approach is to extend the host programming language with databaserelated constructs. This is the major approach, since application programmers need to learn only a new construct of the same language rather than a completely new language. Many of the currently operational databases and object-oriented database systems have adopted this approach; a good example is GemStone from Servio Logic, which has extended the Smalltalk object-oriented programming.

Database Schema and Data Definition Language To represent information in a database, a mechanism must exist to describe or specify to the database the entities of interest. A *data definition language* (DDL) is the language used to describe the structure of and relationships between objects stored in a database. This structure of information is termed the *database schema*. In traditional databases, the schema of a database is the collection of record types and set types or the collection of relationships, templates, and table records used to store information about entities of interest to the application.

For example, to create logical structure or schema, the following SQL command can be used:

```
CREATE SCHEMA AUTHORIZATION (creator)
```

```
CREATE DATABASE (database name)
```

For example,

```
CREATE TABLE INVENTORY (Inventory_Number CHAR(10)NOT NULL
DESCRIPTION CHAR(25) NOT NULL PRICE DECIMAL (9, 2));
```

where the boldface words are SQL keywords.

Data Manipulation Language and Query Capabilities Any time data are collected on virtually any topic, someone will want to ask questions about it. Someone will want the answers to simple questions like "How many of them are there?" or more intricate questions like "What is the percentage of people between ages 21 and 45 who have been employed for five years and like playing tennis?" Asking questions-more formally, making queries of the data-is a typical and common use of a database. A query usually is expressed through a query language.

The *structured query language* (SQL) is the standard DML for relational DBMSs. SQL is widely used for its query capabilities. The query usually specifies

- .The domain of the discourse over which to ask the query.

- .The elements of general interest.

- .The conditions or constraints that apply.

- .The ordering, sorting, or grouping of elements and the constraints that apply to the ordering or grouping. "-.

Query processes generally have sophisticated "engines" that determine the best way to approach the database and execute the query over it. They may use information in the database or knowledge of the whereabouts of particular data in the network to optimize the retrieval of a query. Traditionally, DML are either procedural or nonprocedural. A procedural DML requires users to specify what data are desired and how to get the data. A nonprocedural DML, like most databases' fourth generation programming language (4GLs), requires users to specify what data are needed but not how to get the data. Object-oriented query and data manipulation languages, such as Object SQL, provide object management capabilities to the data manipulation language. In a relational DBMS, the DML is independent of the host programming language.

A host language such as C or COBOL would be used to write the body of the application. Typically, SQL statements then are embedded in C or COBOL applications to manipulate data. Once SQL is used to request and retrieve database data, the results of

the SQL retrieval must be transformed into the data structures of the programming language. A disadvantage of this approach is that programmers code in two languages, SQL and the host language. Another is that the structural transformation is required in both database access directions, to and from the database.

For example, to check the table content, the SELECT command is used, followed by the desired attributes. Or, if you want to see all the attributes listed, use the (*) to indicate all the attributes: SELECT DESCRIPTION, PRICE FROM INVENTORY; where inventory is the name of a table.

19.4 LOGICAL AND PHYSICAL DATABASE ORGANIZATION AND ACCESS CONTROL

Logical database organization refers to the conceptual view of database structure and the relationships within the database. For example, object-oriented systems represent databases composed of objects, and many allow multiple databases to share information by defining the same object. Physical database organization refers to how the logical components of the database are represented in a physical form by operating system constructs (i.e., objects may be represented as files).

Shareability and Transactions

Data and objects in the database often need to be accessed and shared by different applications. With multiple applications having access to the object concurrently, it is likely that conflicts over object access will arise. The database then must detect and mediate these conflicts and promote the greatest amount of sharing possible without sacrificing the integrity of data. This mediation process is managed through concurrency control policies, implemented, in part, by transactions. A *transaction* is a unit of change in which many individual modifications are aggregated into a single modification that occurs in its entirety or not at all. Thus, either all changes to objects within a given transaction are applied to the database or none of the changes. A transaction is said to *commit* if all changes can be made successfully to the database and to *abort* if canceled because all changes to the database cannot be made successfully. This ability of transactions ensures atomicity of change that maintain the database in a consistent state.

Many transaction systems are designed primarily for short transactions (lasting on the order of seconds or minutes). They are less suitable for long transactions, lasting hours or longer. Object databases typically are designed to support both short and long transactions. A concurrence control policy dictates what happens when conflicts arise between transactions that attempt access to the same object and how these conflicts are to be resolved.

Concurrency Policy

As you might expect, when several users (or applications) attempt to read and write the same object simultaneously, they create a contention for object. The

concurrency control mechanism is established to mediate such conflicts by making policies that dictate how they will be handled.

A basic goal of the transaction is to provide each user with a consistent view of the database. This means that transactions must occur in serial order. In other words, a given user must see the database as it exists either before a given transaction occurs or after that transaction.

The most conservative way to enforce serialization is to allow a user to lock all objects or records when they are accessed and to release the locks only after a transaction commits. This approach, traditionally known as a *conservative* or *pessimistic policy*, provides exclusive access to the object, despite what is done to it. The policy is very conservative because no other user can view the data until the object is released.

However, by distinguishing between querying (reading or getting data from) the object and writing to it (which is achieved by qualifying the type of lock placed in the object-read lock or -write lock), somewhat greater concurrency can be achieved. This policy allows many readers of an object but only one writer.

Under an optimistic policy, two conflicting transactions are compared in their entirety and then their serial ordering is determined. As long as the database is able to serialize them so that all the objects viewed by each transaction are from a consistent state of the database, both can continue even though they have read and write locks on a shared object. Thus, a process can be allowed to obtain a read lock on an object already write locked if its entire transaction can be serialized as if it occurred either entirely before or entirely after the conflicting transaction. The reverse also is true: A process may be allowed to obtain a write lock on an object that has a read lock if its entire transaction can be serialized as if it occurred after the conflicting transaction. In such cases, the optimistic policy allows more processes to operate concurrently than the conservative policy.

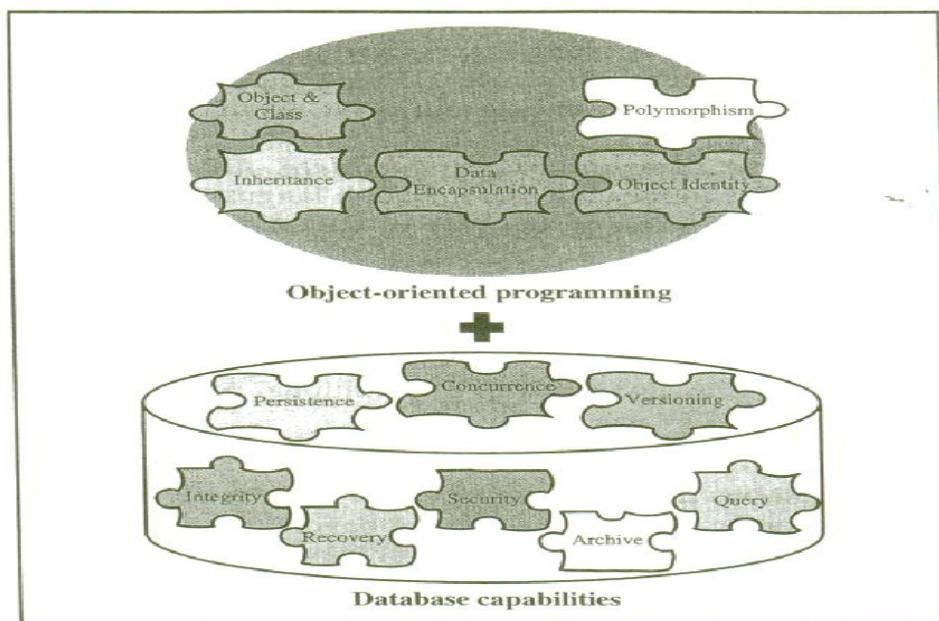
19.5 OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS: THE PURE WORLD

Database management systems have progressed from indexed files to network and hierarchical database systems to relational systems. The requirements of traditional business data processing applications are well met in functionality and performance by relational database systems focused on the needs of business data processing applications. However, as many researchers observed, they are inadequate for a broader class of applications with unconventional and complex data type requirements. These requirements along with the popularity of object-oriented programming have resulted in great demand for an object-oriented DBMS (OODBMS).

Therefore, the interest in OODBMS initially stemmed from the data storage requirements of design support applications (e.g., CAD, CASE, office information systems).

The *object-oriented database management system* is a marriage of object-oriented programming and database technology (see Figure) to provide what we now call *object-*

oriented databases. Additionally, object-oriented databases allow all the benefits of an object orientation as well as the ability to have a strong equivalence with object-oriented programs, an equivalence that would be lost if an alternative were chosen, as with a purely relational database. By combining object-oriented programming with database technology, we have an integrated application development system, a significant characteristic of object-oriented database technology. Many advantages accrue from including the definition of operations with the definition of data. First, the defined operations apply universally and are not dependent on the particular database application running at the moment. Second, the data types can be extended to support complex data such as multimedia by defining new object classes that have operations to support the new kinds of information.



The object-oriented database management system is a marriage of object-oriented programming and database technology.

The "Object-Oriented Database System Manifesto" by Malcom Atkinson et al. described the necessary characteristics that a system must satisfy to be considered an object-oriented database. These categories can be broadly divided into *object-oriented* language properties and *database* requirements.

First, the rules that make it an object-oriented system are as follows:

1. *The system must support complex objects.* A system must provide simple *atomic types of objects* (integers, characters, etc.) from which complex objects can be built by applying constructors to atomic objects or other complex objects or both.
2. *Object identity must be supported.* A data object must have an identity and existence independent of its values.
3. *Objects must be encapsulated.* An object must encapsulate both a program and its data. Encapsulation embodies the separation of interface and implementation and the need for modularity.

4. *The system must support types or classes.* The system must support either the type concept (embodied by C++) or the class concept (embodied by Smalltalk).
5. *The system must support inheritance.* Classes and types can participate in a class hierarchy. The primary advantage of inheritance is that it factors out shared code and interfaces.
6. *The system must avoid premature binding.* This feature also is known as *late binding* or *dynamic binding* (see Chapter 2, which shows that the same method name can be used in different classes). Since classes and types support encapsulation and inheritance, the system must resolve conflicts in operation names at run time.
7. *The system must be computationally complete.* Any computable function should be expressible in the data manipulation language (DML) of the system, thereby allowing expression of any type of operation.
8. *The system must be extensible.* The user of the system should be able to create new types that have equal status to the system's predefined types. These requirements are met by most modern object-oriented programming languages such as Smalltalk and C++. Also, clearly, these requirements are *not* met directly (more on this in the next section) by traditional relational, hierarchical, or network database systems. Second, these rules make it a DBMS:
9. *It must be persistent, able to remember an object state.* The system must allow the programmer to have data survive beyond the execution of the creating process for it to be reused in another process.
10. *It must be able to manage very large databases.* The system must efficiently manage access to the secondary storage and provide performance features, such as indexing, clustering, buffering, and query optimization.
11. *It must accept concurrent users.* The system must allow multiple concurrent users and support the notions of atomic, serializable transactions.
12. *It must be able to recover from hardware and software failures.* The system must be able to recover from software and hardware failures and return to a coherent state.
13. *Data query must be simple.* The system must provide some high-level mechanism for ad-hoc browsing of the contents of the database. A graphical browser might fulfill this requirement sufficiently. These database requirements are met by the majority of existing database systems. From these two sets of definitions it can be argued that an OODBMS is a DBMS with an underlying object-oriented model.

Object-Oriented Databases versus Traditional Databases

The scope of the responsibility of an OODBMS includes definition of the object structures, object manipulation, and recovery, which is the ability to maintain data integrity regardless of system, network, or media failure. Furthermore, OODBMSs like DBMSs must allow for sharing; secure, concurrent multiuser access; and efficient, reliable system performance.

Object-oriented databases also differ from the more traditional relational databases in that they allow representation and storage of data in the form of objects. Each object has its own identity, or *object-ID* (as opposed to the purely value-oriented

approach of traditional databases). The object identity is independent of the state of the object. For example, if one has a car object and we remodel the car and change its appearance, the engine, the transmission, and the tires so that it looks entirely different, it would still be recognized as the same object we had originally. Within an object-oriented database, one always can ask whether this is the same object I had previously, assuming one remembers the object's identity. Object identity allows objects to be related as well as shared within a distributed computing network.

All these advantages point to the application of object-oriented databases to information management problems that are characterized by the need to manage

- . A large number of different data types.
- .A large number of relationships between the objects.
- . Objects with complex behaviors.

Application areas where this kind of complexity exists include engineering, manufacturing, simulations, office automation, and large information systems

19.6 LET US SUM UP

A database management system (DBMS) is a collection of related data and associated programs that access, manipulate, protect, and manage data. the fundamental purpose of a DBMS is to provide a reliable persistent data storage facility and the mechanisms for efficient, convenient data access and retrieval.

Client-server computing is the logical extension of modular programming. The fundamental assumption of modular programming is that separation of a large piece of software into its constituent parts ("modules") creates the possibility for easier development and better maintainability.

The object-oriented database technology is a marriage of object-oriented programming and database technology. The programming and database come together to provide what we call object-oriented databases. By combining object-oriented .

Programming with database technology, we have an integrated application development system, a significant characteristic of object-oriented database technology. "The Object-Oriented Database system Manifesto" by Malcom Atkinson et al. describes the necessary characteristics a system must satisfy to be considered an object-oriented database. These categories can be broadly divided into object oriented language properties and database requirements.

19.7 POINTS FOR DISCUSSION

1. Justify OODBMS
2. Validate database views
3. Validate Hierarchical models
4. Evaluate network model
5. Validate relational model

19.8 LESSON – END ACTIVITIES

1. Evaluate database models.
2. Discuss about data base schemes and data definition languages.
3. Discuss about data manipulation languages and query capabilities.

19.9 REFERENCES

1. Norman,Ronald- object oriented system analysis and design – Prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 20: OBJECT RELATION MAPPING

CONTENTS

- 20.0 AIMS AND OBJECTRIVES
- 20.1 INTRODUCTION
- 20.2 OBJECT RELATION MAPPING
- 20.3 TABLE-CLASS MAPPING
- 20.4 TABLE-MULTIPLE CLASSES MAPPING
- 20.5 LET US SUM UP
- 20.6 POINTS FOR DISCUSSION
- 20.7 LESSON – END ACTIVITIES
- 20.8 REFERENCES

20.0 AIMS AND OBJECTRIVES

- To determine mapping between classes.

20.1 INTRODUCTION

Once the analysis is complete (and sometimes concurrently), we can start designing the user interfaces for the objects and determining how these objects are to be presented. The main goal of a user interface (VI) is to display and obtain needed information in an accessible, efficient manner. The design of the software's interface, more than anything else, affects how a user interacts and therefore experiences an application. It is important for a design to provide users the information they need and clearly tell them how to successfully complete a task. A well-designed VI has visual appeal that motivates users to use your application. In addition, it should use the limited screen space efficiently. In this chapter, we learn how to design the view layer by mapping the VI objects to the view layer objects, "Fe look at VI design rules based on the design corollaries, and finally, we look at the guidelines for developing a graphical user interface. A *graphical user interface* (GVI) uses icons to represent objects, a pointing device to select operations, and graphic imagery to represent relationships. See Appendix B for a review of Windows and graphical user interface basics and treatments.

20.2 OBJECT-RELATION MAPPING

In a relational database, the schema is made up of tables, consisting of rows and columns, where each column has a name and a simple data type. In an object model, the counterpart to a table is a class (or classes), which has a set of attributes (properties or data members). Object classes describe behavior with methods. A tuple (row) of a table contains data for a single entity that correlates to an object (instance of a class) in an

object-oriented system. In addition, a stored procedure in a relational database may correlate to a method in an object-oriented architecture. A *stored procedure* is a module of precompiled SQL code maintained within the database that executes on the server to enforce rules the business has set about the data. Therefore, the mappings essential to object and relational integration are between a table and a class, between columns and attributes, between a row and an object, and between a stored procedure and a method.

For a tool to be able to define how relational data maps to and from application objects, it must have at least the following mapping capabilities (note all these are two-way mappings, meaning they map from the relational system to the object and from the object back to the relational system):

.Table-class mapping.

Table-multiple classes mapping.

Table-inherited classes mapping.

Tables-inherited classes mapping.

Furthermore, in addition to mapping column values, the tool must be capable of interpretation of relational foreign keys. The tool must describe both how the foreign key can be used to navigate among classes and instances in the mapped object model and how referential integrity is maintained. *Referential integrity* means making sure that a dependent table's foreign key contains a value that refers to an existing valid tuple in another relation.

20.3 TABLE-CLASS MAPPING

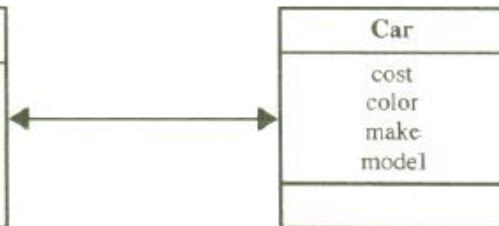
Table-class mapping is a simple one-to-one mapping of a table to a class and the mapping of columns in a table to properties in a class. In this mapping, a single table is mapped to a single class, as shown in Figure

In such mapping, it is common to map all the columns to properties. However, this is not required, and it may be more efficient to map only those columns for which an object model is required by the application(s). With the table-class

Table-class mapping. Each row in the table represents an object instance and each column in the table corresponds to an object attribute.

Car Table

cost	color	make	model



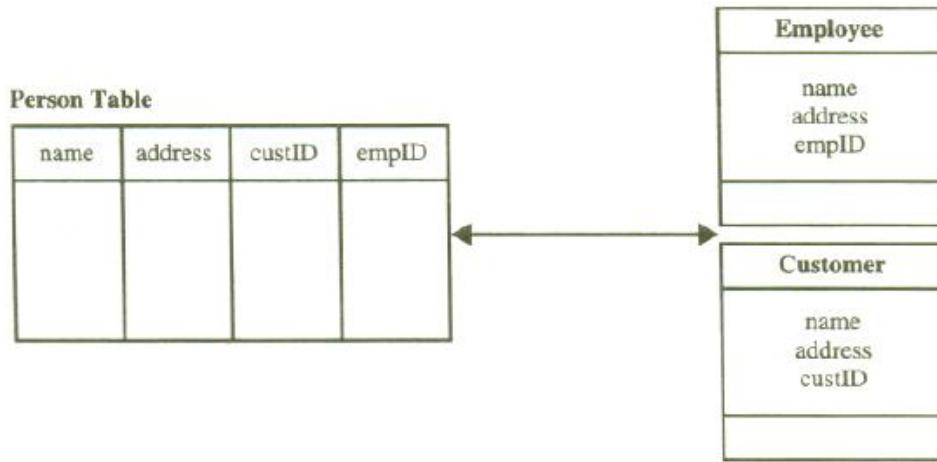


Table-multiple classes mapping. The custID column provides the discriminant. If the value for custID is null, an Employee instance is created at run time; otherwise, a Customer instance is created. In this approach, each row in the table represents an object instance and each column in the table corresponds to an object attribute. This one-to-one mapping of the table-class approach provides a literal translation between a relational data representation and an application object. It is appealing in its simplicity but offers little flexibility.

20.4 TABLE-MULTIPLE CLASSES MAPPING

In the table-multiple classes mapping, a single table maps to multiple noninheriting classes. Two or more distinct, noninheriting classes have properties that are mapped to columns in a single table. At run time, a mapped table row is accessed as an instance of one of the classes, based on a column value in the table. In Figure , the custID column provides the discriminant. If the value for custID is null, an Employee instance is created at run time; otherwise, a Customer instance is created.

In table-inherited classes mapping, a single table maps to many classes that have a common superclass. This mapping allows the user to specify the columns to be shared among the related classes. The superclass may be either abstract or instantiated. In Figure , instances of salariedEmployee can be created for any row in the Person table that has a non null value for the Salary column. If Salary is null, the row is represented by an hourly Employee instance.

.Another approach here is tables-inherited classes mapping, which allows the translation of *is-a* relationships that exist among tables in the relational schema into class inheritance relationships in the object model. In a relational database, an is-a relationship often is

modeled by a primary key that acts as a foreign key to another table. In the object model, *is-a* is another term for an inheritance relation-

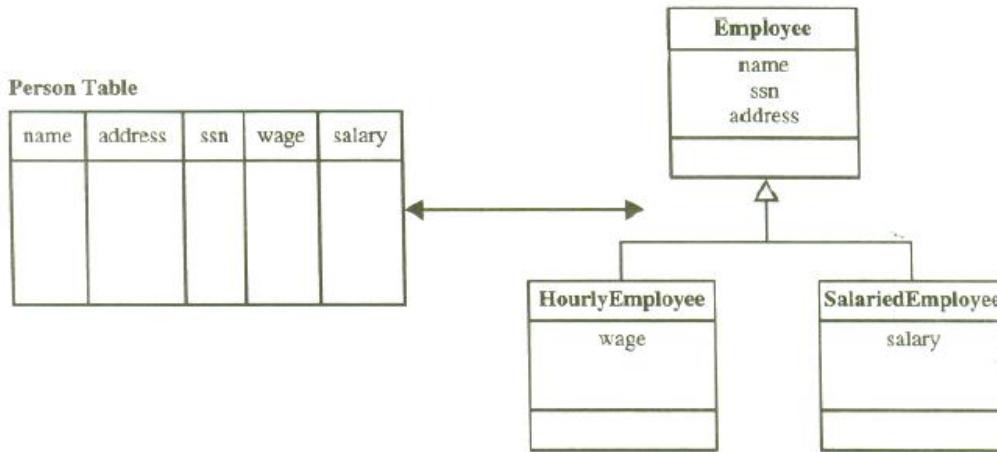


Table-inherited classes mapping. Instances of `SalariedEmployee` can be created for any row in the `Person` table that has a non null value for the `salary` column. If `salary` is null, the row is represented by an `HourlyEmployee` instance.

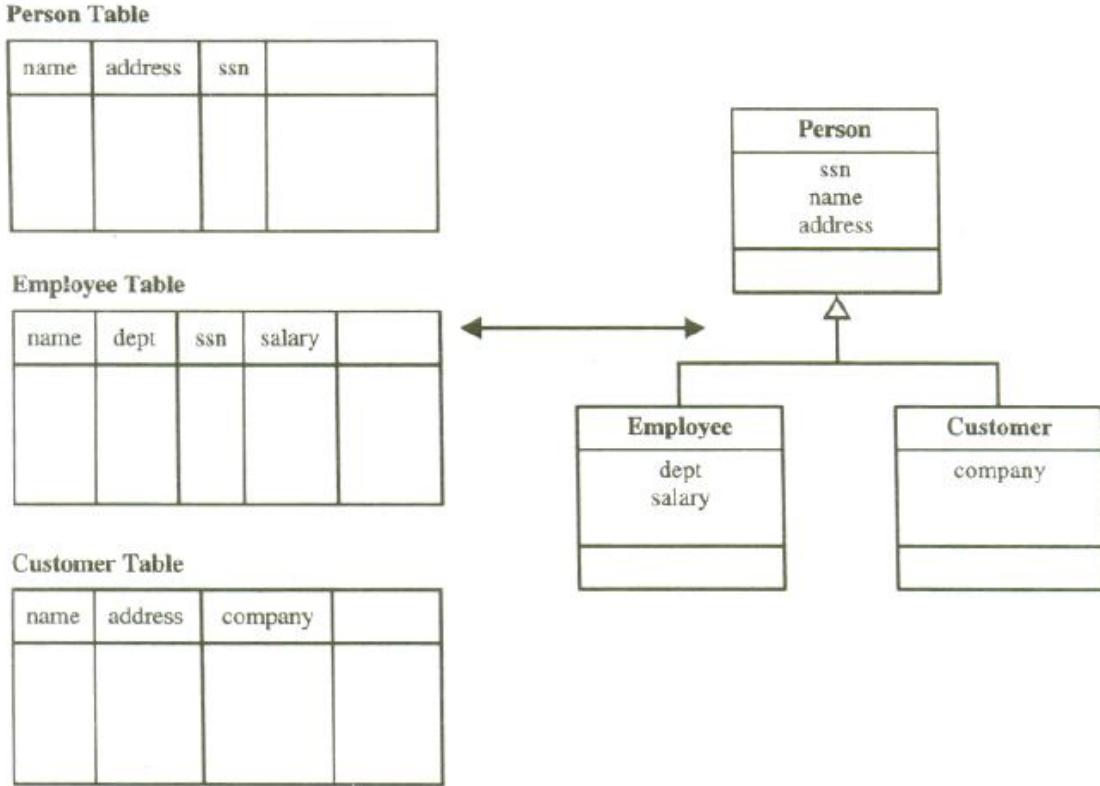
ship. By using the inheritance relationship in the object model, the mapping can express a richer and clearer definition of the relationships than is possible in the relational schema. Figure shows an example that maps a `Person` table to class `Person` and then maps a related `Employee` table to class `Employee`, which is a subclass of class `Person`. In this example, instances of `Person` are mapped directly from the `Person` table. However, instances of `Employee` can be created only for the rows in the `Employee` table (the joining of the `Employee` and `Person` tables on the `SSN` key).

Furthermore, `SSN` is used both as a primary key on the `Person` table for activating instances of `Person` and as a foreign key on the `Person` table and a primary key on the `Employee` table for activating instances of type `Employee`.

Keys for Instance Navigation

In mapping columns to properties, the simplest approach is to translate a column's value into the corresponding class property value. There are two interpretations of this mapping: Either the column is a data value or it defines a navigable relationship between instances (i.e., a foreign key). The mapping also should specify how to convert each data value into a property value on an instance.

In addition to simple data conversion, mapping of column values defines the interpretation of relational foreign keys. The mapping describes both how the foreign key can be used to navigate among classes and instances in the mapped object model and how referential integrity is maintained. A foreign key defines a relationship between tables in a relational database. In an object model, this association is where objects can have references to other objects that enable instance to instance navigation.



Tables-inherited classes mapping.

Instances of Person are mapped directly from the Person table. However, instances of Employee can be created only for the rows in the Employee table (the joining of the Employee and Person tables on the ssn key). The ssn is used both as a primary key on the Person table and as a foreign key on the Person table and a primary key on the Employee table for activating instances of type Employee.

In Figure , the departmentID property of Employee uses the foreign key in column Employee.departmentID. Each Employee instance has a direct reference of class Department (association) to the department object to which it belongs. A popular mechanism in relational databases is the use of stored procedures. As mentioned earlier, stored procedures are modules of precompiled SQL code stored in the database that execute on the server to enforce rules the business has set about the data. Mapping should support the use of stored procedures by allowing mapping of existing stored procedures to object methods.

DESIGNING VIEW LAYER CLASSES

An implicit benefit of three-layer architecture and separation of the view layer from the business and access layers is that, when you design the VI objects, you have to think more explicitly about distinctions between objects that are useful to users. A distinguishing characteristic of view layer objects or interface objects is that they are the

only exposed objects of an application with which users can interact. After all, view layer classes or interface objects are objects that represent the set of operations in the business that users must perform to complete their tasks, ideally in a way they find natural, easy to remember, and useful. Any objects that have direct contact with the outside world are visible in interface objects, whereas business or access objects are more independent of their environment. As explained in Chapter 4, the view layer objects are responsible for two major aspects of the applications:

1. *Input-responding to user interaction.* The user interface must be designed to translate an action by the user, such as clicking on a button or selecting from a menu, into an appropriate response. That response may be to open or close another interface or to send a message down into the business layer to start some business process. Remember, the business logic does not exist here, just the knowledge of which message to send to which business object.
2. *Output-displaying or printing business objects.* This layer must paint the best picture possible of the business objects for the user. In one interface, this may mean entry fields and list boxes to display an order and its items. In another, it may be a graph of the total price of a customer's orders.

The process of designing view layer classes is divided into four major activities:

The macro level VI design process-identifying view layer objects. This activity, for the most part, takes place during the analysis phase of system development. The main objective of the macro process is to identify classes that interact with human actors by analyzing the use cases developed in the 'analysis' phase. As described in previous chapters, each use case involves actors and the task they want the system to do. These use cases should capture a complete, unambiguous, and consistent picture of the interface requirements of the system.

After all, use cases concentrate on describing what the system does rather than how it does it by separating the behavior of a system from the way it is implemented, which requires viewing the system from the user's perspective rather than that of the machine. However, in this phase, we also need to address the issue of how the interface must be implemented. Sequence or collaboration diagrams can help by allowing us to zoom in on the actor-system interaction and extrapolate interface classes that interact with human actors; thus, assisting us in identifying and gathering the requirements for the view layer objects and designing them.

Micro level VI design activities:

Designing the view layer objects by applying design axioms and corollaries.

In designing view layer objects, decide how to use and extend the components so they best support application-specific functions and provide the most usable interface.

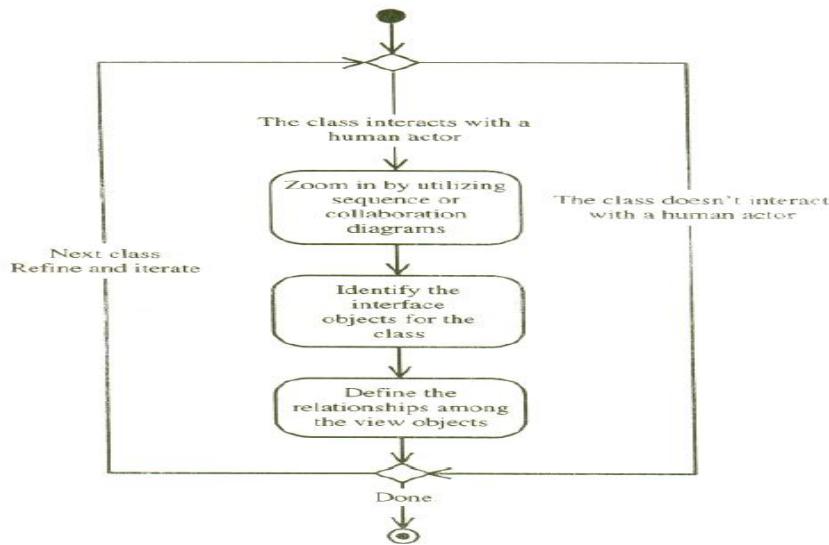
Prototyping the view layer interface. After defining a design model, prepare a prototype of some of the basic aspects of the design. Prototyping is particularly useful early in the design process.

Testing usability and user satisfaction. "We must test the application to make sure it meets the audience requirements. To ensure user satisfaction, we must measure user satisfaction and its usability along the way as the UI design takes form. Usability experts

agree that usability evaluation should be part of the development process rather than a post-mortem or forensic activity. Despite the importance of usability and user satisfaction, many system developers still fail to pay adequate attention to usability, focusing primarily on functionality". In too many cases, usability still is not given adequate consideration.

Macro-level process: identifying view classes by analyzing use cases

The interface object handles all communication with the actor but processes no business rules or object storage activities. In essence, the interface object will



Effective interface design is more than just following a set of rules. It also involves early planning of the interface and continued work through the software development process. The process of designing the user interface involves identifying the specific needs of the application, identifying the use cases and interface object and then devising a design that best meets users' needs. The remainder of this chapter describes the micro-level VI design process and the issues involved.

20.5 LET US SUM UP

In practice, even though many applications increasingly are developed using object-oriented programming technology, chances are good that the data those application need to access live in a very different universe – a relational database. To resolve such a mismatch, The application objects and the relational data must be mapped. Tools that can be used to establish the object-relational mapping processes have begun to emerge. The main process in relational – object integration is defining the relationships between the table structures (represented as schemata) in the relational database with classes (representing classes) in the object model.

20.6 POINTS FOR DISCUSSION

1. Justify object oriented databases versus traditional databases
2. Validate table class mapping

20.7 LESSON – END ACTIVITIES

1. Establish object relation mapping.
2. Discuss about Table – multiple classes mapping.

20.8 REFERENCES

1. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
2. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
3. Rumbaugh, James ,Michael –Object oriented Modelling and design
4. Ali –Brahmi –Object oriented system development”

UNIT – V:

LESSON – 21: SOFTWARE QUALITY ASSURANCE

CONTENTS

- 21.0 AIMS AND OBJECTIVES**
- 21.1 INTRODUCTION**
- 21.2 SOFTWARE QUALITY ASSURANCE**
- 21.3 QUALITY ASSURANCE TESTS**
- 21.4 TESTING STRATEGIES**
- 21.5 LET US SUM UP**
- 21.6 POINTS FOR DISCUSSION**
- 21.7 LESSON – END ACTIVITIES**
- 21.8 REFERENCES**

21.0 AIMS AND OBJECTIVES

You should be able to understand quality assurance and debugging.

21.1 INTRODUCTION.

To develop and deliver robust systems, we need a high level of confidence that Each component will behave correctly. . Collective behavior is correct. .No incorrect collective behavior will be produced. Not only do we need to test components or the individual objects, we also must examine collective behaviors to ensure maximum operational reliability. Verifying components in isolation is necessary but not sufficient to meet this end.

21.2 SOFTWARE QUALITY ASSURANCE

In the early history of computers, live bugs could be a problem (see Bugs and Debugging). Moths and other forms of natural insect life no longer trouble digital computers. However, bugs and the need to debug programs remain. In a 1966 article in *Scientific American*, computer scientist Christopher Strachey wrote,

Although programming techniques have improved immensely since the early years, the process of finding and correcting errors in programming—"debugging" still remains a most difficult, confused and unsatisfactory operation. The chief impact of this state of affairs is psychological. Although we are happy to pay lip service to the adage that to

err is human, most of us like to make a small private reservation about our own

Bugs and Debugging

The use of the term *bug* in computing has been traced to Grace Murray Hopper during the final days of World War II. On September 9, 1945, she was part of a team at Harvard University, working to build the Mark II, a large relay computer (actually a room-size electronic calculator). It was a hot summer evening and the Mark II's developers had the window open. Suddenly, the device stopped its calculations. The trouble turned out to involve a flip-flop switch (a relay). When the defective relay was

located, the team found a moth in it (the first case of a "bug"). "We got a pair of tweezers," wrote programmer Hopper. "Very carefully we took the moth out of the relay, and put it in the logbook, and put scotch tape over it."

After that, whenever Howard Aiken asked if a team was "making any numbers," negative responses were given with explanation "we are debugging the computer."

performance on special occasions when we really try. It is somewhat deflating to be shown publicly and incontrovertibly by a machine that even when we do try, we in fact make just as many mistakes as other people. If your pride cannot recover from this blow, you will never make a programmer.

Although three decades have elapsed since these lines were written, they still capture the essence and mystique of debugging. Precisely speaking, the elimination of the syntactical bug is the process of debugging, whereas the detection and elimination of the logical bug is the process of testing. Gruenberger writes, The logical bugs can be extremely subtle and may need a great deal of effort to eliminate them. It is commonly accepted that all large software systems(operating or application) have bugs remaining in them. The number of possible paths through a large computer program is enormous, and it is physically impossible to explore all of them. The single path containing a bug may not be followed in actual production runs for a long time (if ever) after the program has been certified as correct by its author or others.

In this previous chapter, we look at the testing strategies, the impact of an object orientation on software quality, and some guidelines for developing comprehensive test cases and plans that can detect and identify potential problems before delivering the software to its users. After all, defects can affect how well the software satisfies the users' requirements. Previous chapter addresses usability and user satisfaction tests.

21.3 QUALITY ASSURANCE TESTS

One reason why quality assurance is needed is because computers are infamous for doing what you tell them to do, not necessarily what you want them to do. To close this gap, the code must be free of errors or bugs that cause unexpected results, a process called debugging..

Scenario-based testing, also called *usage-based testing*, concentrates on what the user does, not what the product does. This means capturing use cases and the tasks users perform, then performing them and their variants as tests. These scenarios also can identify interaction bugs. They often are more complex and realistic than error-based tests. Scenario-based tests tend to exercise multiple subsystems in a single test, because

that is what users do. The tests will not find everything, but they will cover at least the higher visibility system interaction bugs .

21.4 TESTING STRATEGIES

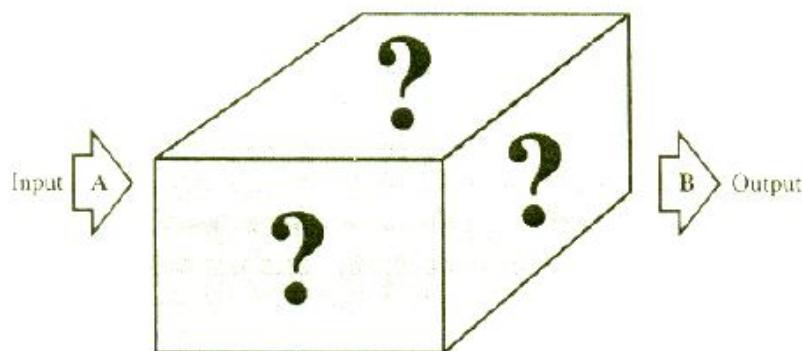
The extent of testing a system is controlled by many factors, such as the risks involved, limitations on resources, and deadlines. In light of these issues, we must deploy a testing strategy that does the "best" job of finding defects in a product within the given constraints. There are many testing strategies, but most testing uses a combination of these: black box testing, white box testing, top-down testing, and bottom-up testing. However, no strategy or combination of strategies truly can prove the correctness of a system; it can establish only its "acceptability."

Black Box Testing

The concept of the black box is used to represent a system whose inside workings are not available for inspection . In a black box, the test item is treated as "black," since its logic is unknown; all that is known is what goes in and what comes out, or the input and output (see Figure 13-1). Weinberg describes writing a user manual as an example of a black box approach to requirements. The user manual does not show the internal logic, because the users of the system do not care about what is inside the system.

In *black box testing*, you try various inputs and examine the resulting output; you can learn what the box does but nothing about how this conversion is implemented . Black box testing works very nicely in testing objects in an object-oriented environment. The black box testing technique also can be used for scenario-based tests, where the system's inside may not be available for inspection but the input and output are defined through use cases or other analysis information.

The black box is an imaginary box that hides its internal workings.



White Box Testing

White box testing assumes that the specific logic is important and must be tested to guarantee the system's proper functioning. The main use of the white box is in error-based testing, when you already have tested all objects of an application and all external or *public methods* of an object that you believe to be of greater importance (see Figure).

In white box testing, you are looking for bugs that have a low probability of execution, have been carelessly implemented, or were overlooked previously .

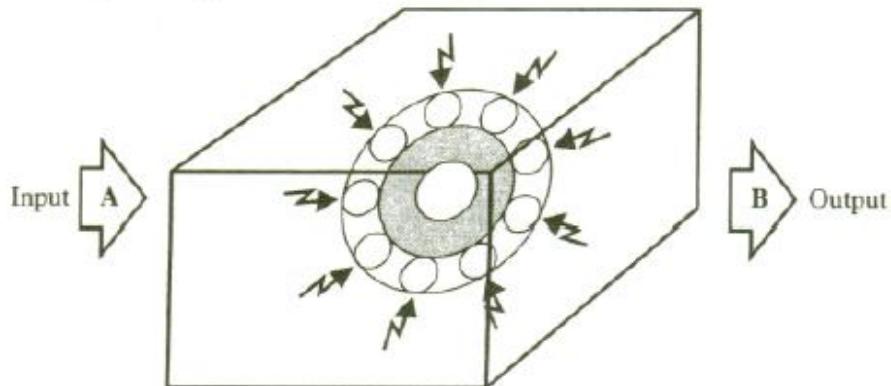
One form of white box testing, called *path testing*, makes certain that each path in a object's method is executed at least once during testing. Two types of path testing are statement testing coverage and branch testing coverage : *Statement testing coverage*. The main idea of statement testing coverage is to test every statement in the object's method by executing it at least once. Murray states, "Testing less than this for new software is unconscionable and should be criminalized" [quoted in 2]. However, realistically, it is impossible to test a program on every single input, so you never can be sure that a program will not fail on some input. *Branch testing coverage*. The main idea behind branch testing coverage is to perform enough tests to ensure that every branch alternative has been executed at least once under some test . As in statement testing coverage, it is unfeasible to fully test any program of considerable size.

Most debugging tools are excellent in statement and branch testing coverage. White box testing is useful for error-based testing.

Top-Down Testing

Top-down testing assumes that the main logic or object interactions and systems messages of the application need more testing than an individual object's methods or supporting logic. A top-down strategy can detect the serious design flaws early in the implementation.

In a white-box testing strategy, the internal workings are known.



In theory, top-down testing should find critical design errors early in the testing process and significantly improve the quality of the delivered software because of the iterative nature of the test . A top-down strategy supports testing the user interface and event-driven systems. Testing the user interface using a top-down approach means testing interface navigation. This serves two purposes, according to Conger. First, the top-down approach can test the navigation through screens and verify that it matches the requirements. Second, users can see, at an early stage, how the final application will look and feel . This approach also is useful for scenario-based testing. Topdown testing is useful to test subsystem and system integration.

Bottom-Up Testing

Bottom-up testing starts with the details of the system and proceeds to higher levels by a progressive aggregation of details until they collectively fit the requirements for the system. This approach is more appropriate for testing the individual objects in a system. Here, you test each object, then combine them and test their interaction and the messages passed among objects by utilizing the top-down approach.

In bottom-up testing, you start with the methods and classes that call or rely on no others. You test them thoroughly. Then you progress to the next level up: those methods and classes that use only the bottom level ones already tested. Next, you test combinations of the bottom two layers. Proceed until you are testing the entire program. This strategy makes sense because you are checking the behavior of a piece of code before it is used by another. Bottom-up testing leads to integration testing, which leads to systems testing.

21.5 LET US SUM UP

Testing may be conducted for different reasons. Quality assurance testing looks for potential problems in a proposed design. In this chapter, we looked at guidelines and the basic concept of test plans and saw that, for the most part, use cases can be used to describe the usage test cases. Also, some of the techniques, strategies, and approaches for quality assurance testing and the impact of object orientation on testing are discussed.

21.6 POINTS FOR DISCUSSION

1. Justify about quality assurance tests
2. Evaluate black box testing
3. Validate white box testing

21.7 LESSON – END ACTIVITIES

1. Discuss about software quality assurance.
2. Establish top-down and bottom-up testing.

21.8 REFERENCES

1. Norman,Ronald- object oriented system analysis and design – Prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 22: OBJECT ORIENTATION ON TESTING

CONTENTS

- 22.0 AIMS AND OBJECTIVES
- 22.1 INTRODUCTION
- 22.2 IMPACT OF OBJECT ORIENTATION ON TESTING
- 22.3 IMPACT OF INHERITANCE IN TESTING
- 22.4 REUSABILITY OF TESTS
- 22.5 TEST CASES
- 22.6 LET US SUM UP
- 22.7 POINTS FOR DISCUSSION
- 22.8 LESSON – END ACTIVITIES
- 22.9 REFERENCES

22.0 AIMS AND OBJECTIVES

To study about

Testing impacts and
Resusability tests.

22.1 INTRODUCTION

The problem of testing messages in an object orientation is the same as testing code that takes a function as a parameter and then invokes it. Marick argues that the process of testing variable uses in OOD essentially does not change, but you have to look in more places to decide what needs testing

22.2 IMPACT OF OBJECT ORIENTATION ON TESTING

The impact of an object orientation on testing is summarized by the following. Some types of errors could become less plausible (not worth testing for). .Some types of errors could become more plausible (worth testing for now). . Some new types of errors might appear.

For example, when you invoke a method, it may be hard to tell exactly which method gets executed. The method may belong to one of many classes. It can be hard to tell the exact class of the method. When the code accesses it, it may get an unexpected value. In a non-object-oriented system, when you looked at
`x = computePayrollO;`

you had to think about the behaviors of a single function. In an object-oriented environment, you may have to think about the behaviors of *base::computePayroll()*, of *derived: :computePayroll*, and so on. For a single message, you need to explore (or at least think about) the union of all distinct behaviors. The problem can be complicated if you have multiple inheritance. However, by applying the design axioms and corollaries of object-oriented design you can limit the differences in behavior between base and derived classes. The testing approach is essentially the same in both environments.. Has the plausibility of faults changed? Are some types of fault now more plausible or less plausible? Since object-oriented methods generally are smaller, these are easier to test. At the same time, there are more opportunities for integration faults. They become more likely, more plausible.

22.3 IMPACT OF INHERITANCE IN TESTING

Suppose you have this situation : The base class contains methods *inheritedO* and *redefinedO* and the derived class redefines the *redefinedO* method.

The *derived::redefined* has to be tested afresh since it is a new code. Does *derived:: inheritedO* have to be retested? If it uses *redefinedO* and the *redefinedO* has changed, the *derived::inheritedO* may mishandle the new behavior. So, it needs new tests even though the *derived: :inheritedO* itself has not changed.

If the *base::inherited* has been changed, the *derived::inheritedO* may not have to be completely tested. Whether it does depends on the base methods; otherwise, it must be tested again. The point here is that, if you do not follow the OOD guidelines, especially if you don't test incrementally, you will end up with objects that are extremely hard to debug and maintain.

22.4 REUSABILITY OF TESTS

If *base::redefinedO* and *derived::redefinedO* are two different methods with different protocols and implementations, each needs a different set of test requirements derived from the use cases, analysis, design, or implementation. But the methods are likely to be similar. Their sets of test requirements will overlap. The better the OOD, the greater is the overlap. You need to write new tests only for those *derived::redefined* requirements not satisfied by the *base::redefined* tests. If you have to apply the *base::redefined* tests to objects of the class "derived," the test inputs may be appropriate for both classes but the expected results might differ in the derived class .

Marick argues that the simpler is a test, the more likely it is to be reusable in subclasses. But simple tests tend to find only the faults you specifically target; complex tests are better at both finding those faults and stumbling across others by sheer luck. There is a trade-off here, one of many between simple and complex tests. The models developed for analysis and design should be used for testing as well. For example, the class diagram describes relationships between objects; that is, an object of one class may use or contain an object of another class, which is useful information for testing.

Furthermore, the use-case diagram and the highest level class diagrams can benefit the scenario-based testing. Since a class diagram shows the inheritance structure, which is important information for error-based testing, it can be used not only during analysis but also during testing.

22.5 TEST CASES

To have a comprehensive testing scheme, the test must cover all methods or a good majority of them. All the services of your system must be checked by at least one test. To test a system, you must construct some test input cases, then describe how the output will look. Next, perform the tests and compare the outcome with the expected output. The good news is that the use cases developed during analysis can be used to describe the usage test cases. After all, tests always should be designed from specifications and not by looking at the product! Myers describes the objective of testing as follows. Testing is the process of executing a program with the intent of finding errors. A good test case is the one that has a high probability of detecting an as-yet undiscovered error. A successful test case is the one that detects an as-yet undiscovered error.

Guidelines for Developing Quality Assurance Test Cases

Gause and Weinberg provide a wonderful example to highlight the essence of a test case. Say, we want to test our new and improved "Superchalk": Writing a geometry lesson on a blackboard is clearly normal use for Superchalk. Drawing on clothing is not normal, but is quite reasonable to expect. Eating Superchalk may be unreasonable, but the design will have to deal with this issue in some way, in order to prevent lawsuits. No single failure of requirements work leads to more lawsuits than the confident declaration.

Basically, a test case is a set of what-if questions. Freedman and Thomas have developed guidelines that have been adapted for the UA: .Describe which feature or service (external or internal) your test attempts to cover. .If the test case is based on a use case (i.e., this is a usage test), it is a good idea to refer to the use-case name. Remember that the use cases are the source of test cases. In theory, the software is supposed to match the use cases, not the reverse. As soon as you have enough of use cases, go ahead and write the test plan for that piece. . Specify what you are testing and which particular feature (methods). Then, specify what you are going to do to test the feature and what you expect to happen. .Test the normal use of the object's methods. .Test the abnormal but reasonable use of the object's methods. .Test the abnormal and unreasonable use of the object's methods.

Test the boundary conditions. For example, if an edit control accepts 32 characters, try 33, then try 40. Also specify when you expect error dialog boxes, when you expect some default event, and when functionality still is being defined. .Test objects' interactions and the messages sent among them. If you have developed sequence diagrams, they can assist you in this process. .When the revisions have been made, document the cases so they become the starting basis for the follow-up test. .Attempting

to reach agreement on answers generally will raise other what-if questions. Add these to the list and answer them, repeat the process until the list is stabilized, then you need not add any more questions.

.The internal quality of the software, such as its reusability and extendability, should be assessed as well. Although the reusability and extendability are more difficult to test, nevertheless they are extremely important. Software reusability rarely is practiced effectively. The organizations that will survive in the 21st century will be those that have achieved high levels of reusability-anywhere from 70-80 percent or more . Griss argues that, although reuse is widely desired and often the benefit of utilizing object technology, many object-oriented reuse efforts fail because of too narrow a focus on technology rather than the policies set forth by an organization. He recommends an institutionalized approach to software development, in which software assets intentionally are created or acquired to be reusable. These assets then are consistently used and maintained to obtain high levels of reuse, thereby optimizing the organization's ability to produce high-quality software products rapidly and effectively. Your test case may measure what percentage of the system has been reused, say, measured in terms of reused lines of code as opposed to new lines of code written. Specifying results is crucial in developing test cases. You should test cases that are supposed to fail. During such tests, it is a good idea to alert the person running them that failure is expected. Say, we are testing a File Open feature. We need to specify the result as follows:

1. Drop down the File menu and select Open.
2. Try opening the following types of files:
 - . A file that is there (should work).
 - .A file that is not there (should get an Error message).
 - .A file name with international characters (should work).
 - .A file type that the program does not open (should get a message or conversion dialog box).

22.6 LET US SUM UP

Testing is a balance of art, science, and luck. It may seem that everything will fall into place without any preparation and a bug-free product will be shipped. However, in the real world, we must develop a test plan for locating and removing bugs. A test plan offers a road map for testing activity; it should state test objectives and how to meet them. The plan need not be very large; in fact, devoting too much time to the plan can be counterproductive. There are no magic tricks to debugging; however, by selecting appropriate testing strategies and a sound test plan, you can locate the errors in your system and fix them by utilizing debugging tools.

22.7 POINTS FOR DISCUSSION

1. Analyze impact of inheritance in testing
2. Discuss reusability of tests

22.8 LESSON – END ACTIVITIES

1. Establish the guidelines for developing quality assurance test cases.
2. Analyse the impact of object orientation on testing.

22.9 REFERENCES

1. Norman,Ronald- object oriented system analysis and design – Prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”.

LESSON – 23: TEST PLANS

CONTENTS

- 23.0 AIMS AND OBJECTIVES
- 23.1 INTRODUCTION
- 23.2 TEST PLAN
- 23.3 GUIDELINES FOR DEVELOPING TEST PLANS
- 23.4 CONTINUOUS TESTING
- 23.5 MYERS'S DEBUGGING PRINCIPLES
- 23.6 LET US SUM UP
- 23.7 POINTS FOR DISCUSSION
- 23.8 LESSON – END ACTIVITIES
- 23.9 REFERENCES

23.0 AIM AND OBJECTIVE

To study the test plans

23.1 INTRODUCTION

A *test plan* is developed to detect and identify potential problems before delivering the software to its users. Your users might demand a test plan as one item delivered with the program. In other cases, no test plan is required, but that does not mean you should not have one.

23.2 TEST PLAN

On paper, it may seem that everything will fall into place with no preparation and a bug-free product will be shipped. However, in the real world, it may be a good idea to use a test plan to find bugs and remove them. A dreaded and frequently overlooked activity in software development is writing the test plan. A test plan offers a road map for testing activities, whether usability, user satisfaction, or quality assurance tests. It should state the test objectives and how to meet them. The test plan need not be very large; in fact, devoting too much time to the plan can be counterproductive. The following steps are needed to create a test plan:

1. *Objectives of the test.* Create the objectives and describe how to achieve them. For example, if the objective is usability of the system, that must be stated and also how to realize it. (Usability testing will be covered in Previous chapter.)

2. *Development of a test case.* Develop test data, both input and expected output, based on the domain of the data and the expected behaviors that must be tested (more on this in the next section).

3. *Test analysis.* This step involves the examination of the test output and the documentation of the test results. If bugs are detected, then this is reported and the activity centers on debugging. After debugging, steps 1 through 3 must be repeated until no bugs can be detected.

All passed tests should be repeated with the revised program, called *regression testing*, which can discover errors introduced during the debugging process. When sufficient testing is believed to have been conducted, this fact should be reported, and testing for this specific product is complete .

According to Tamara Thomas , the test planner at Microsoft, a good test plan is one of the strongest tools you might have. It gives you the chance to be clear with other groups or departments about what will be tested, how it will be tested, and the intended schedule. Thomas explains that, with a good, clear test plan, you can assign testing features to other people in an efficient manner. You then can use the plan to track what has been tested, who did the testing, and how the testing was done. You also can use your plan as a checklist, to make sure that you do not forget to test any features.

Who should do the testing? For a small application, the designer or the design team usually will develop the test plan and test cases and, in some situations, actually will perform the tests. However, many organizations have a separate team, such as a quality assurance group, that works closely with the design team and is responsible for these activities (such as developing the test plans and actually performing the tests). Most software companies also use *beta testing*, a popular, inexpensive, and effective way to test software on a select group of the actual users of the system. This is in contrast to *alpha testing*, where testing is done by inhouse testers, such as programmers, software engineers, and internal users. If you are going to perform beta testing, make sure to include it in your plan, since it needs to be communicated to your users well in advance of the availability of your application in a beta version.

23.3 GUIDELINES FOR DEVELOPING TEST PLANS

As software gains complexity and interaction among programs is more tightly coupled, planning becomes essential. A good test plan not only prevents overlooking a feature (or features), it also helps divide the work load among other people, explains Thomas.

The following guidelines have been developed by Thomas for writing test plans : .You may have requirements that dictate a specific appearance or format for your test plan. These requirements may be generated by the users. Whatever the appearance of your test plan, try to include as much detail as possible about the tests. .The test plan should contain a schedule and a list of required resources. List how many people will be needed, when the testing will be done, and what equipment will be required.

After you have determined what types of testing are necessary (such as black box, white box, top-down, or bottom-up testing), you need to document specifically what you are going to do. Document every type of test you plan to complete. The level of detail in your plan may be driven by several factors, such as the following: How much test time do you have? Will you use the test plan as a training tool for newer team members? . A configuration control system provides a way of tracking the changes to the code. At a minimum, every time the code changes, a record should be kept that tracks which module has been changed, who changed it, and when it was altered, with a comment about why the change was made. Without configuration control, you may have difficulty keeping the testing in line with the changes, since frequent changes may occur without being communicated to the testers. .

A well-thought-out design tends to produce better code and result in more complete testing, so it is a good idea to try to keep the plan up to date. Generally, the older a plan gets, the less useful it becomes. If a test plan is so old that it has become part of the fossil record, it is not terribly useful. As you approach the end of a project, you will have less time to create plans . If you do not take the time to document the work that needs to be done, you risk forgetting something in the mad dash to the finish line. Try to develop a habit of routinely bringing the test plan in sync with the product or product specification. . At the end of each month or as you reach each milestone, take time to complete routine updates. This will help you avoid being overwhelmed by being so out-of-date that you need to rewrite the whole plan. Keep configuration information on your plan, too. Notes about who made which updates and when can be very helpful down the road

23.4 CONTINUOUS TESTING

Software is tested to determine whether it conforms to the specifications of requirements. Software is maintained when errors are found and corrected, and software is extended when new functionality is added to an already existing program. There can be different reasons for testing, such as to test for potential problems in a proposed design or to compare two or more designs to determine which is better, given a specific task or set of tasks. A common practice among developers is to turn over applications to a quality assurance (QA) group for testing only after development is completed. Since it is not involved in the initial plan, the testing team usually has no clear picture of the system and therefore cannot develop an effective test plan.

Furthermore, testing the whole system and detecting bugs is more difficult than testing smaller pieces of the application as it is being developed. The practice of waiting until after the development to test an application for bugs and performance could waste thousands of dollars and hours of time. ' Testing often uncovers design weaknesses or, at least, provides information you will want to use. Then, you can repeat the entire process, taking what you have learned and reworking your design, or move onto reprototyping and retesting. Testing must take place on a continuous basis, and this refining cycle must continue

23.5 MYERS'S DEBUGGING PRINCIPLES

I conclude this discussion with the Myers's bug location and debugging principles

1. Bug Locating Principles . Think. . If you reach an impasse, sleep on it. . If the impasse remains, describe the problem to someone else. . Use debugging tools (this is slightly different from Myers's suggestion). .Experimentation should be done as a last resort (this is slightly different from Myers's suggestion).

2. Debugging Principles . Where there is one bug, there is likely to be another. .Fix the error, not just the symptom of it. .The probability of the solution being correct drops as the size of the program increases. .Beware of the possibility that an error correction will create a new error (this is less of a problem in an object-oriented environment).

CASE STUDY: DEVELOPING TEST CASES FOR THE VIANET BANK ATM SYSTEM

We identified the scenarios or use cases for the ViaNet bank ATM system. The ViaNet bank ATM system has scenarios involving Checking Account, Savings Account, and general Bank Transaction (see Figures. Here again is a list of the use cases that drive many object-oriented activities, including the usability testing: .Bank Transaction (see Figure). .Checking Transaction History (see Figure). .Deposit Checking (see Figure). .Deposit Savings (see Figure). .Savings Transaction History (see Figure). .Withdraw Checking (see Figure). .Withdraw Savings (see Figure). .Valid/Invalid PIN (see Figure).

The activity diagrams and sequence/collaboration diagrams created for these use cases are used to develop the usability test cases. For example, you can draw activity and sequence diagrams to model each scenario that exists when a bank client withdraws, deposits, or needs information on an account. Walking through the steps can assist you in developing a usage test case.

Let us develop a test case for the activities involved in the ATM transaction based on the use cases identified so far. (See the activity diagram in Figure and the sequence diagram of Figure to refresh your memory.)

23.6 LET US SUM UP

Once you have created fully tested and debugged classes of objects, you can put them into a library for use or reuse. The essence of an object-oriented system is that you can take for granted that these fully tested objects will perform their desired functions and seal them off in your mind like black boxes. Testing must take place on a continuous basis, and this refining cycle must continue throughout the development process until you are satisfied with the results.

23.7 POINTS FOR DISCUSSION

1. Generate test plans
2. Critically analyze Myers's debugging principles

23.8 LESSON – END ACTIVITIES

1. Discuss about continuous testing.
2. Establish guidelines for developing test plans.

23.9 REFERENCES

1. Norman,Ronald- object oriented system analysis and design – Prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
- 5.Ali –Brahmi –Object oriented system development”

LESSON – 24: SYSTEM USABILITY AND USER SATISFACTION

CONTENTS

- 24.0 AIMS AND OBJECTIVES
- 24.1 INTRODUCTION
- 24.2 USABILITY TESTING
- 24.3 GUIDELINES FOR DEVELOPING USABILITY TESTING
- 24.4 RECORDING THE USABILITY TEST
- 24.5 LET US SUM UP
- 24.6 POINTS FOR DISCUSSION
- 24.7 LESSON – END ACTIVITIES
- 24.8 REFERENCES

24.0 AIMS AND OBJECTIVES

You should be able to understand
 System usability and
 User satisfaction.

24.1 INTRODUCTION

Quality refers to the ability of products to meet the users' needs and expectations. The task of satisfying user requirements is the basic motivation for quality. Quality also means striving to do the things right the first time, while always looking to improve how things are being done. Sometimes, this even means spending more time in the initial phases of a project-such as analysis and design-making sure that you are doing the right things. Having to correct fewer problems means significantly less wasted time and capital. When all the losses caused by poor quality are considered, high quality usually costs less than poor quality.

Two main issues in software quality are *validation* or user satisfaction and *verification* or quality assurance (see Previous chapter). There are different reasons for testing. You can use testing to look for potential problems in a proposed design. You can focus on comparing two or more designs to determine which is better, given a specific task or set of tasks. Usability testing is different from quality assurance testing in that, rather than finding programming defects, you assess how well the interface or the software fits the use cases, which are the reflections of users' needs and expectations. To ensure user satisfaction, we must measure it throughout the system development with user satisfaction tests. Furthermore, these tests can be used as a communication vehicle between designers and end users . In the next section, we look at user satisfaction tests

that can be invaluable in developing high- Once the design is complete, you can walk users through the steps of the scenarios to determine if the design enables the scenarios to occur as planned.

24.2 USABILITY TESTING

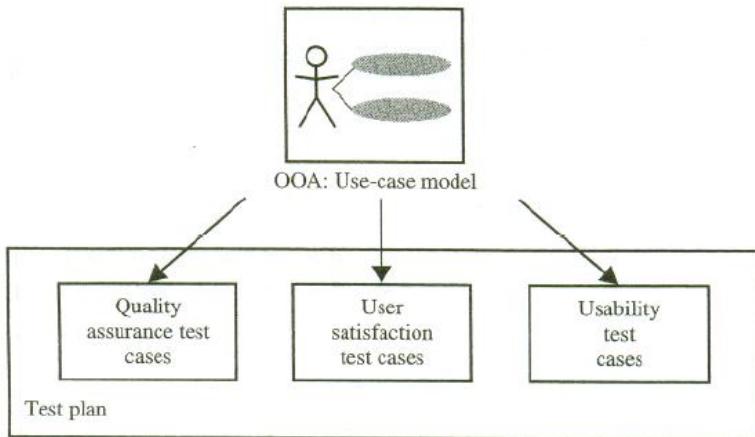
The International Organization for Standardization (ISO) defines *usability* as the effectiveness, efficiency; and satisfaction with which a specified set others can achieve a specified set of tasks in particular environments. The ISO definition requires .Defining tasks. What are the tasks? . Defining users. Who are the users? .*A means for measuring effectiveness, efficiency, and satisfaction*. How do we measure usability?

The phrase *two sides of the same coin* is helpful for describing the relationship between the usability and functionality of a system. Both are essential for the development of high-quality software . *Usability testing* measures the ease of use as well as the degree of comfort and satisfaction users have with the software. Products with poor usability can be difficult to learn, complicated to operate, and misused or not used at all. Therefore, low product usability leads to high costs for users and a bad reputation for the developers. Usability is one of the most crucial factors in the design and development of a product, especially the user interface. Therefore, usability testing must be a key part of the UI design process.

Usability testing should begin in the early stages of product development; for example, it can be used to gather information about how users do their work and find out their tasks, which can complement use cases. You can incorporate your findings into the usability test plan and test cases. As the design progresses, usability testing continues to provide valuable input for analyzing initial design concepts and, in the later stages of product development, can be used to test specific product tasks, especially the ill.

Usability test cases begin with the identification of use cases that can specify the target audience, tasks, and test goals. When designing a test, focus on use cases or tasks, not features. Even if your goal is testing specific features, remember that your users will use them within the context of particular tasks. It also is a good idea to run a pilot test to work the bugs out of the tasks to be tested and make certain the task scenarios, prototype, and test equipment work smoothly. Test cases must include all use cases identified so far. Recall from Previous chapter that the use case can be used through most activities of software development.

Furthermore, by following Jacobson's life cycle model, you can produce designs that are traceable across requirements, analysis, design, implementation, and testing. The main advantage is that all design traces directly back to the user requirements. Use cases and usage scenarios can become test scenarios; and therefore, the use case will drive the usability, user satisfaction, and quality assurance test cases (see Figure).



The use cases identified during analysis can be used in testing the design. Once the design is complete, walk users through the steps of the scenarios to determine if the design enables the scenarios to occur as planned.

24.3 GUIDELINES FOR DEVELOPING USABILITY TESTING

Many techniques can be used to gather usability information. In addition to use cases, focus groups can be helpful for generating initial ideas or trying out new ideas. A focus group requires a moderator who directs the discussion about aspects of a task or design but allows participants to freely express their opinions.

Usability tests can be conducted in a one-on-one fashion, as a demonstration, or as a "walk through," in which you take the users through a set of sample scenarios and ask about their impressions along the way. In a technique called the *Wizard of OZ*, a testing specialist simulates the interaction of an interface. Although these latter techniques can be valuable, they often require a trained, experienced test coordinator 9. Let us take a look at some guidelines for developing usability testing: .The usability testing should include aU of a software's components. . Usability testing need not be very expensive or elaborate, such as including trained specialists working in a soundproof lab with one-way mirrors and sophisticated recording equipment. Even the small investment of tape recorder, stopwatch, and notepad in an office or conference room can produce excellent results. . Similarly, all tests need not involve many subjects. More typically, quick, iterative tests with a small, well-targeted sample of 6 to 10 participants can identify 80-90 percent of most design problems. . Consider the user's experience as part of your software usability. You can study 80-90 percent of most design problems with as few as three or four users if you target only a single skill level of users, such as novices or intermediate level users. .

24.4 RECORDING THE USABILITY TEST

When conducting the usability test, provide an environment comparable to the target setting; usually a quiet location, free from distractions is best. Make participants

feel comfortable. It often helps to emphasize that you are testing the software, not the participants. If the participants become confused or frustrated, it is no reflection on them. Unless you have participated yourself, you may be surprised by the pressure many test participants feel. You can alleviate some of the pressure by explaining the testing process and equipment. . Tandy Trower, director of the Advanced User Interface group at Microsoft, explains that the users must have reasonable time to try to work through any difficult situation they encounter. Although it generally is best not to interrupt participants during a test, they may get stuck or end up in situations that require intervention. This need not disqualify the test data, as long as the test coordinator carefully guides or hints around a problem. Begin with general hints before moving to specific advice. For more difficult situations, you may need to stop the test and make adjustments. Keep in mind that less intervention usually yields better results. Always record the techniques and search patterns users employ when attempting to work through a difficulty and the number and type of hints you have to provide them.

Ask subjects to think aloud as they work, so you can hear what assumptions and inferences they are making. As the participants work, record the time they take to perform a task as well as any problems they encounter. You may want to follow up the session with the *user satisfaction* test (more on this in the next section) and a questionnaire that asks the participants to evaluate the product or tasks they performed. Record the test results using a portable tape recorder or, better, a video camera. Since even the best observer can miss details, reviewing the data later will prove invaluable. Recorded data also allows more direct comparison among multiple participants. It usually is risky to base conclusions on observing a single subject. Recorded data allows the design team to review and evaluate the results.

Whenever possible, involve all members of the design team in observing the test and reviewing the results. This ensures a common reference point and better design solutions as team members apply their own insights to what they observe. If direct observation is not possible, make the recorded results available to the entire team. To ensure user satisfaction and therefore high-quality software, measure user satisfaction along the way as the design takes form . In the next section, we look at the user satisfaction test, which can be an invaluable tool in developing highquality software.

24.5 LET US SUM UP

In this chapter, we looked at guidelines and the basic concept of test plans and saw that, for the most part, use cases can be used to describe the usage test cases.

The essence of an object-oriented system is that you can take for granted that these fully tested objects will perform their desired functions and seal them off in your mind like black boxes.

Testing must take place on a continuous basis, and this refining cycle must continue throughout the development process until you are satisfied with the results.

24.6 POINTS FOR DISCUSSION

1. Critically analyze usability test
2. Evaluate user satisfaction test

24.8 LESSON – END ACTIVITIES

1. Establish guidelines for developing usability testing.

24.9 REFERENCES

1. Norman, Ronald- object oriented system analysis and design – Prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”

LESSON – 25: USER SATISFACTION TEST

CONTENTS

- 25.0 AIMS AND OBJECTIVES
- 25.1 INTRODUCTION
- 25.2 USER SATISFACTION TEST
- 25.3 GUIDELINES FOR DEVELOPING A USER SATISFACTION TEST
- 25.4 A TOOL FOR ANALYZING USER SATISFACTION: THE USER SATISFACTION TEST TEMPLATE
- 25.5 CASE STUDY: DEVELOPING USABILITY TEST PLANS AND TEST CASES FOR THE VIANET BANK ATM SYSTEM
- 25.6 DEVELOP TEST OBJECTIVES
- 25.7 DEVELOP TEST CASES
- 25.8 ANALYZE THE TESTS
- 25.9 LET US SUM UP
- 25.10 POINTS FOR DISCUSSION
- 25.10 LESSON – END ACTIVITIES
- 25.11 REFERENCES

25.0 AIMS AND OBJECTIVES

To measuring the goals.

25.1 INTRODUCTION

A positive side effect of testing with a prototype is that you can observe how people actually use the software. In addition to prototyping and usability testing, another tool that can assist us in developing high-quality software is measuring and monitoring user satisfaction during software development, especially during the design and development of the user interface.

25.2 USER SATISFACTION TEST

User satisfaction testing is the process of quantifying the usability test with some measurable attributes of the test, such as functionality, cost, or ease of use. Usability can be assessed by defining measurable goals, such as .95 percent of users should be able to find how to withdraw money from the ATM machine without error and with no formal

training. .70 percent of all users should experience the new function as "a clear improvement over the previous one." .90 percent of consumers should be able to operate the VCR within 30 minutes. Furthermore, if the product is being built incrementally, the best measure of user satisfaction is the product itself, since you can observe how users are using it-or avoiding it . Gause and Weinberg have developed a user satisfaction test that can be used along with usability testing. Here are the principal objectives of the user satisfaction test : .

As a communication vehicle between designers, as well as between users and designers. .To detect and evaluate changes during the design process. .To provide a periodic indication of divergence of opinion about the current design. .To enable pinpointing specific areas of dissatisfaction for remedy. .To provide a clear understanding of just how the completed design is to be evaluated.

Even if the results are never summarized and no one fills out a questionnaire, the process of creating the test itself will provide useful information. Additionally, the test is inexpensive, easy to use, and it is educational to both those who administer it and those who take it.

25.3 GUIDELINES FOR DEVELOPING A USER SATISFACTION TEST

The format of every user satisfaction test is basically the same, but its content is different for each project. Once again, the use cases can provide you with an excellent source of information throughout this process. Furthermore, you must work with the users or clients to find out what attributes should be included in the test. Ask the users to select a limited number (5 to 10) of attributes by which the final product can be evaluated. For example, the user might select the following attributes for a customer tracking system: ease of use, functionality, cost, intuitiveness of user interface, and reliability.

A test based on these attributes is shown in Figure . Once these attributes have been identified, they can play a crucial role in the evaluation of the final product. Keep these attributes in the foreground, rather than make assumptions about how the design will be evaluated . The user must use his or her judgment to answer each question by selecting a number between 1 and 10, with 10 as the most favorable and 1 as the least. Comments often are the most significant part of the test. Gause and Weinberg raise the following important point in conducting a user satisfaction test : "When the design of the test has been drafted, show it to the clients and ask, 'If you fill this out monthly (or at whatever interval), will it enable you to express what you like and don't like?' If they answer negatively then find out what attributes would enable them to express themselves and revise the test."

25.4 A TOOL FOR ANALYZING USER SATISFACTION: THE USER SATISFACTION TEST TEMPLATE

Commercial off-the-shelf (COTS) software tools are already written and a few are available for analyzing and conducting user satisfaction tests. However, here, I have selected an electronic spreadsheet to demonstrate how it can be used to record and analyze the user satisfaction test. The user satisfaction test spreadsheet (USTS) automates many bookkeeping tasks and can assist in analyzing the user satisfaction test results. Furthermore, it offers a quick start for creating a user satisfaction test for a particular project.

Recall from the previous section that the tests need not involve many subjects. More typically, quick, iterative tests with a small, well-targeted sample of 6 to 10 participants can identify 80-90 percent of most design problems. The spreadsheet should be designed to record responses from up to 10 users. However, if there are inputs from more than 10 users, it must allow for that (see Figures).

One use of a tool like this is that it shows patterns in user satisfaction level. For example, a shift in the user satisfaction rating indicates that something is happening (see Figure . Gause and Weinberg explain that this shift is sufficient cause to follow up with an interview. The user satisfaction test can be a tool for

Measuring User Satisfaction

Project Name: Customer Tracking System

User	1	2	3	4	5	6	7	8	9	#	1	2	3	4	5	6	7	8	9	#	1	2	3	4	
Ease of Use	1	7								4	2									4	2				
Functionality	4	4								6	8									8	7				
Intuitiveness of	4	6								5	8									8	8				
Cost	1	1								5	5									5	5				
Reliability	3	4								6	8									8	7				
Comments																									



General Comments

File Edit View Contents UST Summary Chart Information CHART1

FIGURE 14-4

A user satisfaction test for a customer tracking system.

FIGURE 14-5

A user satisfaction summary sheet. The main thing to look for in the responses is changes over time. A shift in the user satisfaction rating indicates that something is happening.

Measuring User Satisfaction

Project Name: Customer Tracking System

	Test #1 Period 1	Test #2 Period 2	Test #3 Period 3	Test #4 Period 4	Test #5 Period 5	Test #6 Period 6
Overall Average	4	6	6	5	6	7
Overall High	7	8	8	7	8	9
Overall Low	1	2	2	2	2	4
Changes	Improvement	Improvement	Problem	Improvement	Improvement	
Comments						



Periodical plotting can reveal shifts in user satisfaction, which can pinpoint a problem. Plotting the high and low responses indicates where to go for maximum information (Gause and Weinberg)

finding out what attributes are important or unimportant. An interesting side effect of developing a user satisfaction test is that you benefit from it even if the test is never administered to anyone; it still provides useful information. However, performing the test regularly helps to keep the user involved in the system. It also helps you focus on user wishes. Here is the user satisfaction cycle that has been suggested by Gause and Weinberg:

1. Create a user satisfaction test for your own project. Create a custom form that fits the project's needs and the culture of your organization. Use cases are a great source of information; however, make sure to involve the user in creation of the test.
2. Conduct the test regularly and frequently.
3. Read the comments very carefully, especially if they express a strong feeling. Never forget that feelings are facts, the most important facts you have about the users of the system.
4. Use the information from user satisfaction test, usability test, reactions to prototypes, interviews recorded, and other comments to improve the product.

Another benefit of the user satisfaction test is that you can continue using it even after the product is delivered. The results then become a measure of how well users are learning to use the product and how well it is being maintained. They also provide a starting point for initiating follow-up projects.

25.5 CASE STUDY: DEVELOPING USABILITY TEST PLANS AND TEST CASES FOR THE VIANET BANK ATM SYSTEM

In previous previous chapter, we learned that test plans need not be very large; in fact, devoting too much time to the plans can be counterproductive. Having this in mind let us develop a usability test plan for the ViaNet ATM kiosk by going through the followings steps.

25.6 DEVELOP TEST OBJECTIVES

The first step is to develop objectives for the test plan. Generally, test objectives are based on the requirements, use cases, or current or desired system usage. In this case, ease of use is the most important requirement, since the ViaNet bank customers should be able to perform their tasks with basically no training and are not expected to read a user manual before withdrawing money from their checking accounts.

Here are the objectives to test the usability of the ViaNet bank ATM kiosk and its user interface:

95 percent of users should be able to find out how to withdraw money from the ATM machine without error or any formal training. .90 percent of consumers should be able to operate the ATM within 90 seconds.

25.7 DEVELOP TEST CASES

Test cases for usability testing are slightly different from test cases for quality assurance. Basically, here, we are not testing the input and expected output but how users interact with the system. Once again, the use cases created during analysis can be used to develop scenarios for the usability test. The usability test scenarios are based on the following use cases:

- Deposit Checking (see Figures).
- Withdraw Checking (see Figures).
- Deposit Savings (see Figures).
- Withdraw Savings (see Figures).
- Savings Transaction History (see Figures).
- Checking Transaction History(see Figures).

Next we need to select a small number of test participants (6 to 10) who have never before used the kiosk and ask them to perform the following scenarios based on the use case:

1. Deposit \$1056.65 to your checking account.
2. Withdraw \$40 from your checking account.
3. Deposit \$200 to your savings account.
4. Withdraw \$55 from savings account.
5. Get your savings account transaction history.
6. Get your checking account transaction history.

Start by explaining the testing process and equipment to the participants to ease the pressure. Remember to make participants feel comfortable by emphasizing that you are testing the software, not them. If they become confused or frustrated, it is no reflection on them but the poor usability of the system. Make sure to ask them to think aloud as they work, so you can hear what assumptions and inferences they are making. After all, if they cannot perform these tasks with ease, then the system is not useful.

As the participants work, record the time they take to perform a task as well as any problems they encounter. In this case, we used the kiosk video...camera to record the test results along with a tape recorder. This allowed the design team to review and evaluate how the participants interacted with the user interface, like those developed in Previous chapter . For example, look for things such as whether they are finding the appropriate buttons easily and the buttons are the right size. Once the test subjects complete their tasks, conduct a user satisfaction test to measure their level of satisfaction with the kiosk.

The format of the user satisfaction test is basically the same as the one we studied earlier in this previous chapter (see Figure), but its content is different for the ViaNet bank. The users, uses cases, and test objects should provide the attributes to be included in the test. Here, the following attributes have been selected, since the ease of use is the main issue of the user interface: .Is easy to operate. .Buttons are the right size and easily

located. . Is efficient to use. . Is fun to use. . Is visually pleasing. .Provides easy recovery from errors.

Based on these attributes, the test shown in Figure can be performed. Remember, as explained earlier, these attributes can play a crucial role in the evaluation of the final product.

25.8 ANALYZE THE TESTS

The final step is to analyze the tests and document the test results. Here, we need to answer questions such as these: What percentage were able to operate the ATM within 90 seconds or without error? Were the participants able to find out how to withdraw money from the ATM machine with no help? The results of the analysis must be examined.

We also need to analyze the results of user satisfaction tests. The USTS described earlier or a tool similar to it can be used to record and graph the results of user satisfaction tests. As we learned earlier, a shift in user satisfaction pattern indicates that something is happening and a follow-up interview is needed to find out the reasons for the changes. The user satisfaction test can be used as a tool for finding out what attributes are important or unimportant. For example, based on the user satisfaction test, we might find that the users do not agree that the system "is efficient to use," and it got a low score.

After the follow-up interviews, it became apparent that participants wanted, in addition to entering the amount for withdrawal, to be able to select from a list with predefined values (say, \$20, \$40).

How do you rate the ViaNet bank ATM kiosk interface?											
Is easy to operate:	Very Easy	<input type="checkbox"/>	Very Hard								
	10	9	8	7	6	5	4	3	2	1	
Buttons are right size and easily can be located:	Very Appropriate	<input type="checkbox"/>	Not Appropriate								
	10	9	8	7	6	5	4	3	2	1	
Is efficient to use:	Very Efficient	<input type="checkbox"/>	Very Inefficient								
	10	9	8	7	6	5	4	3	2	1	
Is fun to use:	Fun	<input type="checkbox"/>	No Fun								
	10	9	8	7	6	5	4	3	2	1	
Is visually pleasing:	Very Pleasing	<input type="checkbox"/>	Not Pleasing								
	10	9	8	7	6	5	4	3	2	1	
Provides easy recovery from errors:	Very Easy Recovery	<input type="checkbox"/>	Not at All								
	10	9	8	7	6	5	4	3	2	1	
Comments: <input type="checkbox"/> I have more to say; I would like to see you.											

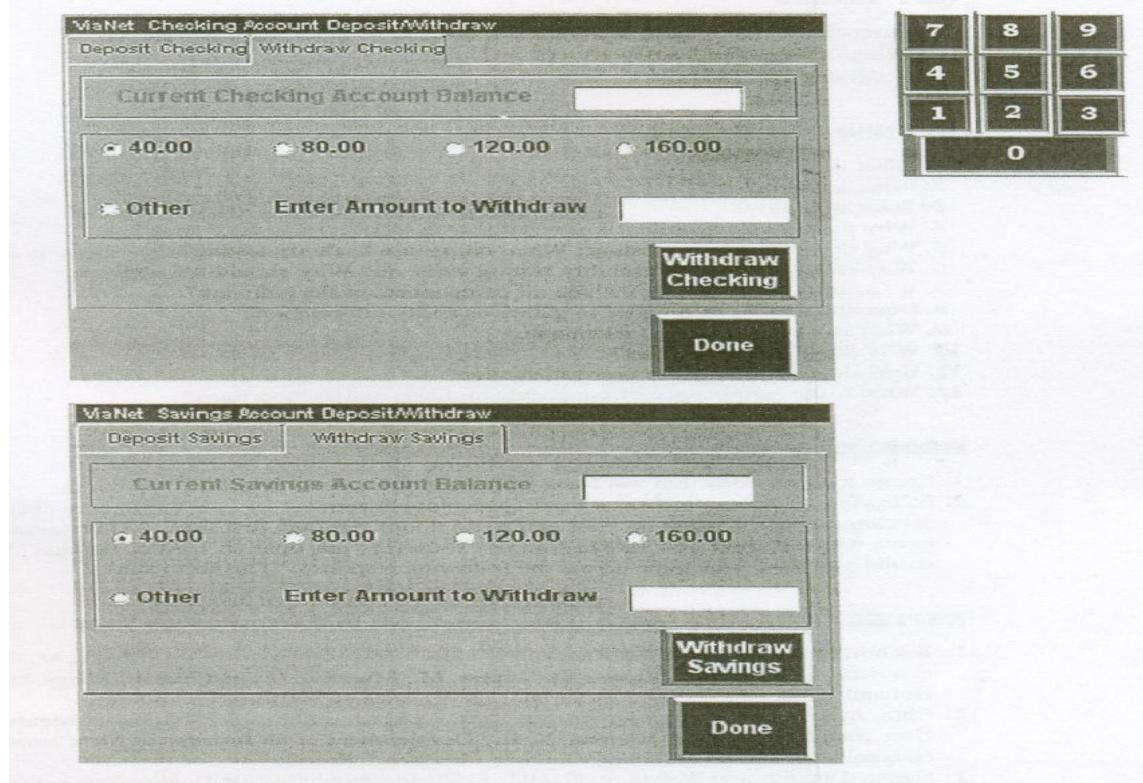
A form for the ViaNet bank ATM kiosk user satisfaction test.

\$60, \$80, \$100, and \$200). This would speed up the process at the ATM kiosk. Based on the result of the test, the UI was modified to reflect the wishes of the users. You need also to pay close attention to comments, especially if they express a strong feeling. Remember, feelings are facts, the most important facts you have about the users of the system.

25.9 LET US SUM UP

In this previous chapter, we looked at different dimensions of quality in software development. We discussed the importance of the usability and user satisfaction tests. The main point here is that you must focus on the users' perception. Many systems that are adequate technically have failed because of poor user perception. This can be prevented, or at least minimized, by utilizing usability and user satisfaction tests as part of your UI design process. Usability testing begins with defining the target audience and test goals. When designing a test, focus on tasks, not features.

Even if your goal is testing specific features, remember that your customers will use them within the context of particular tasks. The use cases identified during analysis can be used in testing your design. Once the design is complete, you can walk users through the steps of the scenarios to determine if the design enables the scenarios to occur as planned. An interesting side effect of developing user satisfaction tests is that you benefit from it even if the test is never administered to anyone; it still provides useful information. However, performing the test regularly helps keep the user actively involved in the system development. It also helps us stay focused on the users' wishes.



The CheckingAccountUI and SavingsAccountUI interface objects for withdrawal modified based on the usability and user satisfaction tests. The users have the option of selecting quick withdrawal or selecting *other option* and then entering the amount they want to withdraw.

25.10 POINTS FOR DISCUSSION

1. Develop test objectives
2. Develop test cases
3. Analyze the tests

25.11 LESSON – END ACTIVITIES

1. Discuss about used satisfaction test.
2. Establish guidelines for user satisfaction test.

25.12 REFERENCES

1. Norman,Ronald- object oriented system analysis and design – Prentice hall 1996
2. Coad.P and Yourdon .E – “Object Oriented Analysis” – Yourdon press
3. Coad.P and Yourdon .E – “Object Oriented Design” – Yourdon press
4. Rumbaugh, James ,Michael –Object oriented Modelling and design
5. Ali –Brahmi –Object oriented system development”