SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – I - OBJECT ORIENTED ANALYSIS AND SYSTEM ENGINEERING - SCSA1401

SCSA1401 - OBJECT ORIENTED ANALYSIS AND SYSTEM ENGINEERING

COURSE OBJECTIVES

- To understand the fundamentals of Object Oriented System Development.
- To understand the object oriented methodologies.
- To use UML in requirements elicitation and designing.
- To understand concepts of relationships and aggregations.
- To test the software against its requirements specification.

COURSE OUTCOMES

On completion of the course, student will be able to:

- Understand the basics object model for System development.
- Understand the object Oriented Methodologies.
- Express software design with UML diagrams.
- Understand the concept of Relationships.
- Design software applications using OO concepts.
- Understand the various testing methodologies for OO software.

UNIT 1 9 Hrs.

AN OVERVIEW OF OBJECT ORIENTED SYSTEM DEVELOPMENT

Introduction - Object Oriented System Development Methodology - Why Object Orientation - Overview of Unified Approach -Object Basics: Object Oriented Philosophy - Objects - Classes - Attributes - Object Behavior and Methods, Messages and Interfaces, Encapsulation and Information Hiding - Class Hierarchy - Polymorphism - Object Relationships and Associations - Aggregations and Object Containment - Object Identity - Static and Dynamic Binding - Persistence. Object-oriented CASE tools, Object Oriented Systems Development Life Cycle: Software Development Process - Building High Quality Software - Use case Driven Approach – Reusability.

UNIT 2 9 Hrs.

OBJECT ORIENTED METHODOLOGIES

Rumbaugh et al.'s Object Modeling Technique - Booch Methodology - Jacobson et al. Methodologies – Patterns - Framework - Unified approach - Unified Modeling Language: Static and Dynamic Model - UML Diagrams - UML Class Diagram – UML Use Case –Case study- Use case Modelling – Relating Use cases – include, extend and generalization – When to use Use-cases- UML Dynamic Modeling – Case study- UML Extensibility - UML Metamodel.

UNIT 3 9 Hrs

OBJECT ORIENTED ANALYSIS.

Business Object Analysis - Use Case Driven Object Oriented Analysis - Business Process Modeling - Use Case model - Developing Effective Documentation - Object Analysis Classification: Classification Theory - Noun Phrase Approach - Common Class Patterns Approach - Use-Case Driven Approach - Classes Responsibilities and Collaborators – Naming Classes - Identifying Object Relationships, Attributes and Methods: Association – SuperSubclass Relationships - A-part of Relationships.

UNIT 4 9 Hrs.

OBJECT ORIENTED DESIGN

Object Oriented Design Process - Object Oriented Design Axioms - Corollaries - Designing Classes: Object Constraint Language - Process of Designing Class - Class Visibility - Refining Attributes - Access Layer: Object Store and Persistence - Database Management System - Logical and Physical Database Organization and Access Control - Distributed Databases and Client Server Computing - Object Oriented Database Management System – Object Relational Systems – Designing Access Layer Classes - View Layer: Designing View Layer Classes - Macro Level Process - Micro Level Process – Purpose of View Layer Interface - Prototyping the user interface.

UNIT 5 9 Hrs.

SOFTWARE QUALITY

Software Quality Assurance- Impact of Object Orientation on Testing - Develop Test Cases and Test Plans – System Usability and Measuring User Satisfaction: Usability Testing - User Satisfaction Testing.

UNIT 1

AN OVERVIEW OF OBJECT ORIENTED SYSTEM DEVELOPMENT

Introduction - Object Oriented System Development Methodology - Why Object Orientation - Overview of Unified Approach -Object Basics: Object Oriented Philosophy - Objects - Classes - Attributes - Object Behavior and Methods, Messages and Interfaces, Encapsulation and Information Hiding - Class Hierarchy - Polymorphism - Object Relationships and Associations - Aggregations and Object Containment - Object Identity - Static and Dynamic Binding - Persistence. Object-oriented CASE tools, Object Oriented Systems Development Life Cycle: Software Development Process - Building High Quality Software - Use case Driven Approach – Reusability.

An Overview of Object Oriented System and Development 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.

INTRODUCTION

Software development is dynamic and always undergoing major change. The methods and tools will differ significantly from those currently in use.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 that, if followed 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.

The original goal based on the system requirements. Further we study about the unified approach, which is the methodology used for learning about object oriented system development.Object-Oriented (OO) systems development is a way to develop software by building self-contained modules that can be more easily:

- Replaced
- Modified and
- Reused

Orthogonal View of the Software:

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.

OBJECT-ORIENTED SYSTEMS DEVELOPMENTMETHODOLOGY

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 aswell 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.

Why an Oject Orientation?

To create sets of objects that work together concurrently to produce s/w that better, model their problem domain that similarly system produced by traditional techniques.
- It adapts to

1. Changing requirements
2. Easier to maintain
3. More robust
4. Promote greater design
5. Code reuse

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 anotherrequires a complex transition of perspective between models that almost can bein different worlds. This transition not only can slow the development processbut also increases the size of the project and the chance for errors introduced inmoving 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 therest of the system or how it is going to be used within a particular system.

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 unifiedmodeling language.

- The UA, based on methodologies by Booch, Rumbaugh, Jacobson, and others, tries to combine the best practices, processes, and guidelines.
- UA based on methodologies by Booch, Rumbaugh and Jacobson tries to combine the best practices, processes and guidelines along with the object management groups in unified modelling language.
- UML is a set of notations and conventions used to describe and model an application.
- UA utilizes the unified modeling language (UML) which is a set of notations and conventions used to describe and model an application.

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.

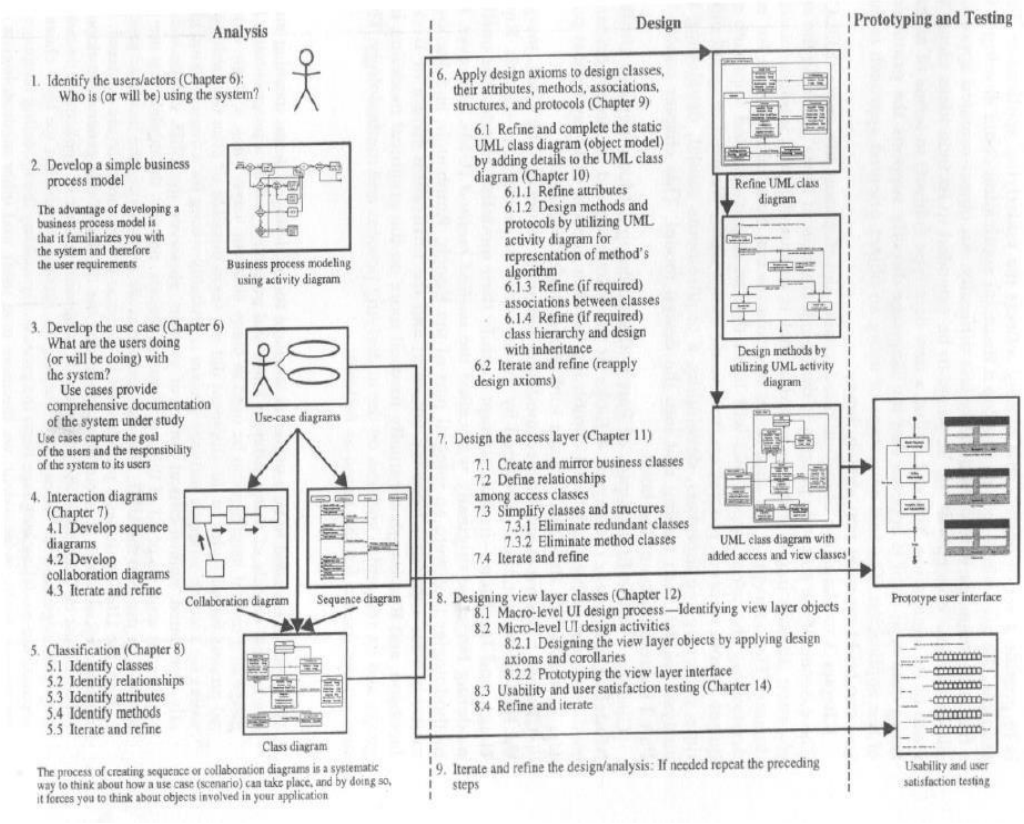


Fig 1.1 The unified approach road map.

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.

OBJECT BASICS:

Goals:

- Define Objects and classes
- Describe objects‘ methods, attributes and how objects respond to messages,
- Define Polymorphism, Inheritance, data abstraction, encapsulation, and protocol,
- Describe objects relationships,
- Describe object persistence,
- Understand meta-classes.

What is an object?

• The term object was first formally utilized in the Similar language to simulatesome aspect of reality.

• An object is an entity.

- It knows things (has attributes)
- It does things (provides services or has methods)

Example: *It Knows things (attributes)*

- I am an Employee.
- I know my name,
- social security number and
- my address.

Attributes

- I am a Car.
- I know my color,
- manufacturer, cost,
- owner and model.

It does things (methods)

- I know how to
- compute
- my payroll.

Attributes or properties describe object‘s state (data) and methods define itsbehavior.

Object:

- In an object-oriented system, everything is an object: numbers, arrays, records, fields, files, forms, an invoice, etc.
- An Object is anything, real or abstract, about which we store data and those methods that manipulate the data.
- Conceptually, each object is responsible for itself.
- A window object is responsible for things like opening, sizing, and closing itself.
- A chart object is responsible for things like maintaining its data andlabels, and even for drawing itself.

Two Basic Questions When developing an O-O application, two basic questions always arise.

- What objects does the application need?
- What functionality should those objects have?

Traditional Approach

• The traditional approach to software development tends toward writing a lot of code to do all the things that have to be done.

• You are the only active entity and the code is just basically a lot of building materials.

Object-Oriented Approach OO approach is more like creating a lot of helpers that take on an active role, a spirit, that form a community whose interactions become the application.

Object's Attributes

- Attributes represented by data type.
- They describe objects states.
- In the Car example the car's attributes are:
- color, manufacturer, cost, owner, model, etc.

Object's Methods

- Methods define objects behavior and specify the way in which an Object's data are manipulated.
- In the Car example the car's methods are:
- drive it, lock it, tow it, carry passenger in it.

Objects are Grouped in Classes

- 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 role of a class is to define the attributes and methods (the state and behavior) and applicability of its instances.
- The class car, for example, defines the property color.
- Each individual car (object) will have a value for this property, such as "maroon," "yellow" or "white."
- Each object is an instance of a class. There may be many different classes.

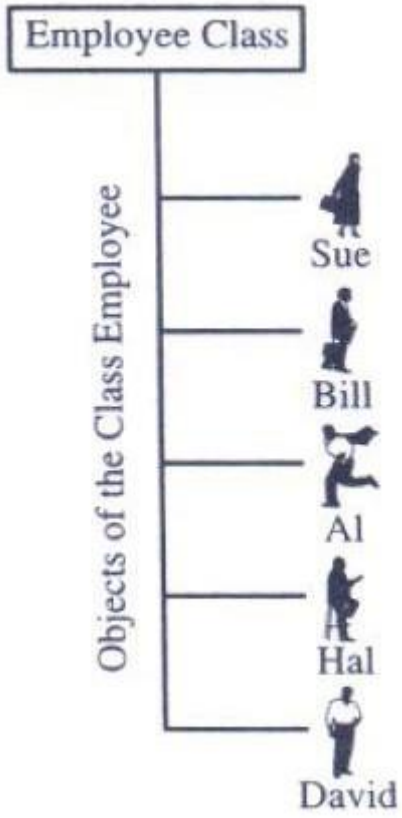


FIGURE 1.2 Sue, Bill, AI, Hal, and David are instances or objects of the classEmployee.

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 (Figure1.3).

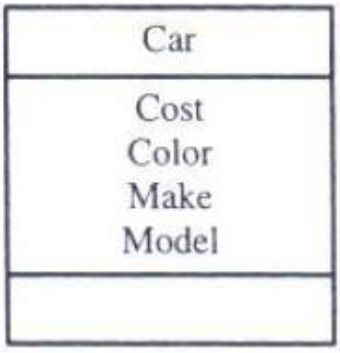


FIGURE 1.3 The attributes of a car object.

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. The importance distinction is that an object's abstract state can be independent of its physical representation.

OB.JECTS BEHAVIOR AND METHODS

- Behavior denotes the collection of methods that abstractly describes whatan object is capable of doing.
- Each produre defines and describes a particular behavior of an object.
- The object , called the receiver, is that on which the method operates.
- Methods encapsulate the behavior of the object, provide interface to the object, and hide any of the internal strctures and states maintained by theobject.
- Procedure provide us the means to communicate with an object andaccess its properties.
- Objects take responsibility for their own behavior.

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 an 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 objectspecific.

In the top example, depicted in Figure 1.4, 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.

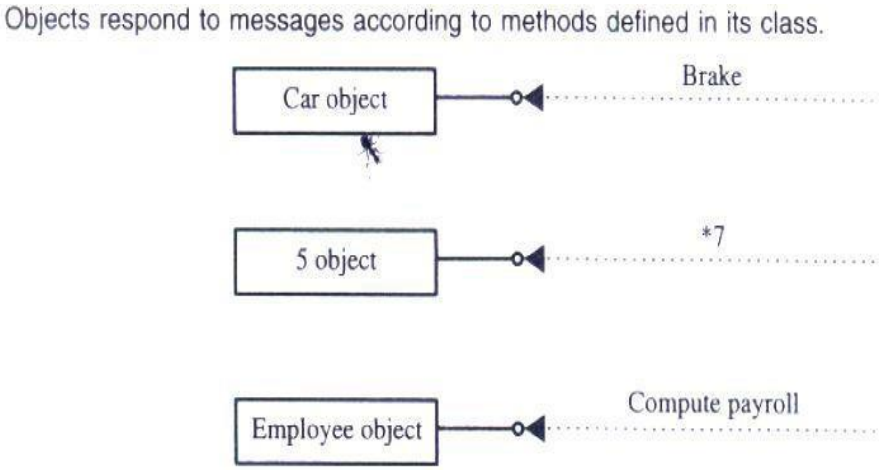


FIG 1.4

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.

ENCAPSULATION AND INFORMATION HIDING

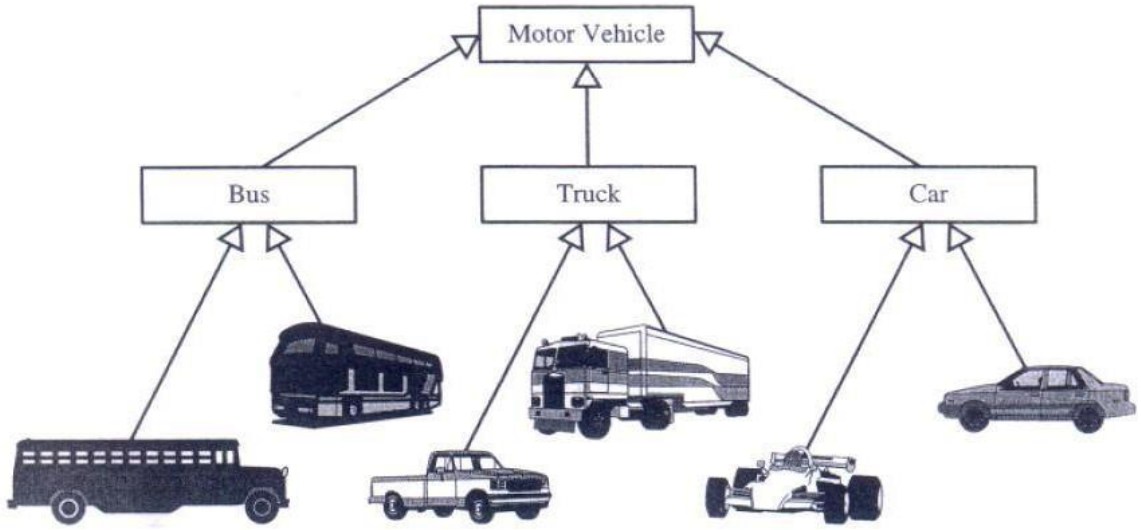
- Information hiding is the principle of concealing the internal data and procedures of an object and providing an interface to each object in sucha way as to reveal as little as possible about its inner workings.
- As in conventional programming, some languages permit arbitrary access to objects and 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.
- 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.

Class Hierarchy

- An object-oriented system organizes classes into subclass-super hierarchy.
- At the top of the hierarchy are the most general classes and at the bottomare the most specific
- A subclass inherits all of the properties and methods (procedures) definedin its super class.

Superclass/subclass hierarchy.



The family car in Figure 1.5 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.

Inheritance (programming by extension)

- Inheritance is a relationship between classes where one class is the parent class of another (derived) class.
- Inheritance allows classes to share and reuse behaviors and attributes.
- The real advantage of inheritance is that we can build upon what we already have and,
- Reuse what we already have.

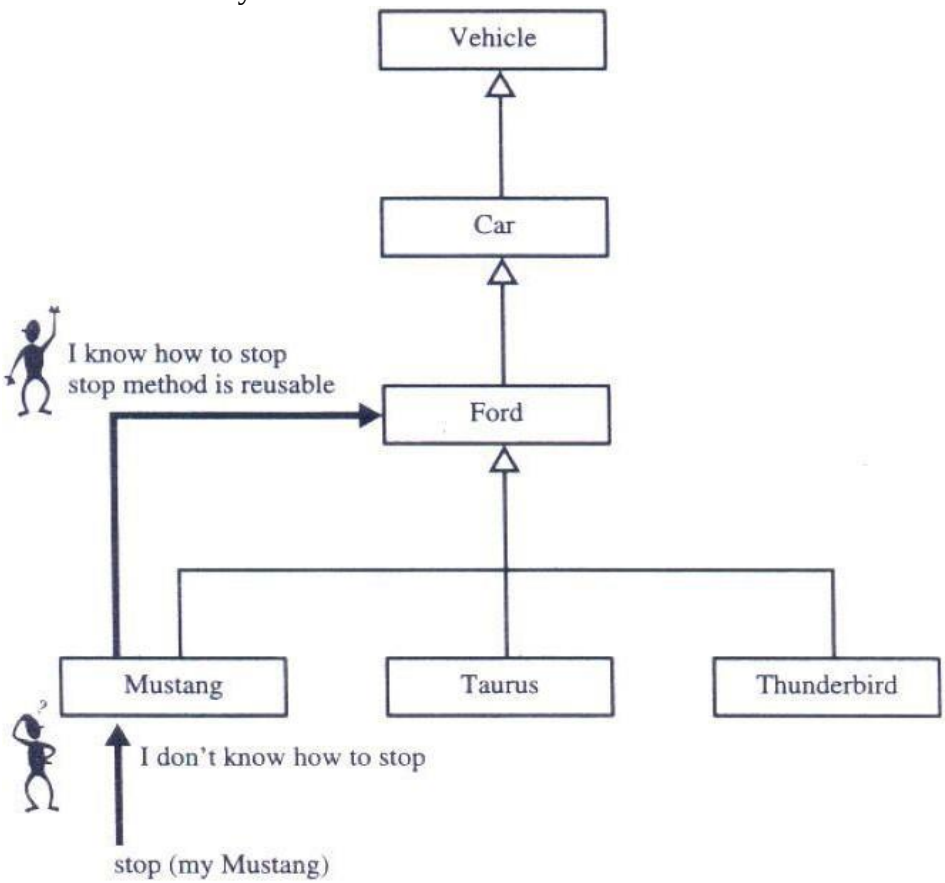


FIGURE 1.6 INHERITANCE ALLOWS REUSABLITY

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.

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.

For example utility vehicle inherent from Car and Truck classes.

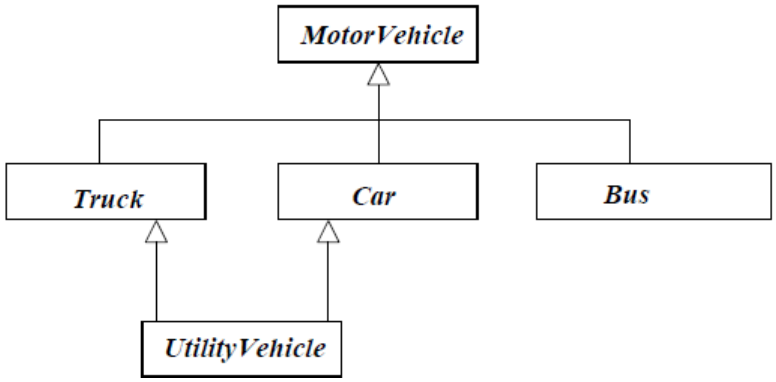


Fig 1.7 Utility vehicle inherent from car and truck classses.

POLYMORPHISM

Poly means "many" and *morph* means "form."

Polymorphism means that the same operation may behave differently on different classes.

Booch defines *polymorphism* as the relationship of objects of manydifferent 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 andthe shift, so in addition to all other mechanical controls, you also need information on when to shift gears. Therefore, although driving is a behavior weperform 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 ofan 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.

OBJECT RELATIONSHIPS AND ASSOCIATIONS

ASSOCIATIONS

The concept of ***association*** **represents relationships between objects and classes.** For example a pilot *can fly* planes.

FIGURE 1.8 Association represents the relationship among objects, whichis bidirectional.

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. Forexample, *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 server. For example, we have a print object that prints the consumer object. The print producer provides the ability to print other objects. Figure 1.9 depicts the consumerproducer association.

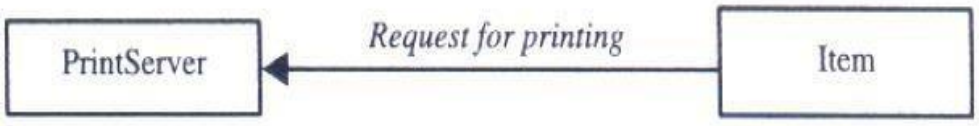


FIGURE 1.9 The consumer/producer association.

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 somple 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 example a car object is an aggregation of engine, seat, wheels and other objects.

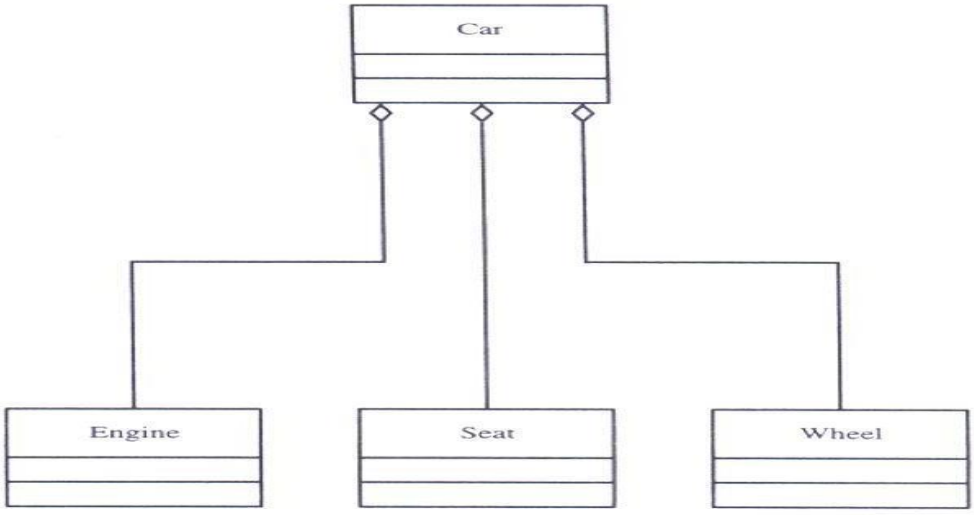


Fig 1.10 – A car object is an aggregation of other objects such as engine, seat and wheelobjects.

A Case Study - A Payroll Program

Consider a payroll program that processes employee records at a small manufacturing firm. Thiscompany has three types of employees:

- *Managers*: Receive a regular salary.
- *Office Workers*: Receive an hourly wage and are eligible for overtime after 40 hours.
- *Production Workers*: Are paid according to a piece rate.

Structured Approach

```
FOR EVERY EMPLOYEE
DOBEGIN
 IF employee = manager
THENCALL
computeManagerSalary
IF employee = office worker
THENCALL
computeOfficeWorkerSalary
IF employee = production worker
THEN CALL
computeProductionWorkerSalaryEND
```

What if we add two new types of employees?

Temporary office workers ineligible for overtime, junior production workers who receivean hourly wage plus a lower piece rate.

```
FOR EVERY EMPLOYEE
DOBEGIN
IF employee = manager
THENCALL
computeManagerSalary
IF employee = office worker
THEN CALL
computeOfficeWorker_salary
IF employee = production worker
THENCALL
computeProductionWorker_salary
IF employee = temporary office worker
THEN CALL
computeTemporaryOfficeWorkerSalary IF
employee = junior production worker THEN
CALL
computeJuniorProductionWorkerSalary END
```

An Object-Oriented Approach

What objects does the application need?

• The goal of OO analysis is to identify objects and classes that support the problem domain andsystem's requirements.

• Some general candidate classes are:

• *Persons*

• *Places*

• *Things*

• ***Class Hierarchy***

- Identify class hierarchy
- Identify commonality among the classes
- Draw the general-specific class hierarchy.

Class Hierarchy

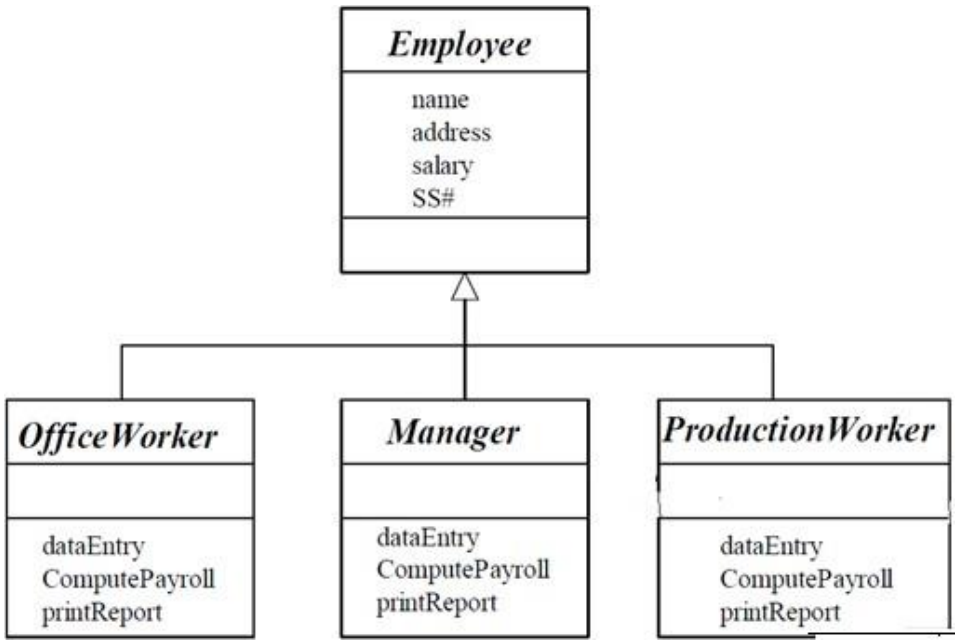


Fig: 1.11 Class hierarchy for the payroll application

OO Approach

FOR EVERY EMPLOYEE
DOBEGIN
employee
computePayrollEND

ADVANCE TOPICS
DYNAMIC BINDING

The process of detennining (dynamically) at run time which function to invoke is termed ***dynamic binding***. Making this detennination 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) .

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 *cre*ated. 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.This characteristic is called O***bject 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.

Meta-Classes

- Everything is an object.
- How about a class?
- Is a class an object?
- Yes, a class is an object! So, if it is an object, it must belong to a class.
- Indeed, class belongs to a class called a Meta-Class or a class' class.
- Meta-class used by the compiler. For example, the meta-classes handle messages to classes, such as constructors and "new."

Rather than treat data and procedures separately, object-oriented programming packages them into "objects." O-O system provides you with the set of objects that closely reflects the underlying application. Advantages of object-oriented programming are:

- The ability to reuse code,
- develop more maintainable systems in a shorter amount of time.
- more resilient to change, and
- more reliable, since they are built from completely tested and debugged classes.

Object Oriented Systems Development Life Cycle

Goals

- **The software development process**
- **Building high-quality software**
- **Object-oriented systems development**
- **Use-case driven systems development**
- **Prototyping**
- **Rapid application development**
- **Component-based development**
- **Continuous testing and reusability**

SOFTWARE PROCESS

The essence of the software process consists of analysis, design, implementation, testing, and refinement is to transform users' needs into a software solution that satisfies those needs.

THE SOFTWARE DEVELOPMENT PROCESS

System development can be viewed as a process. Furthermore, the development itself 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.

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 1.12):

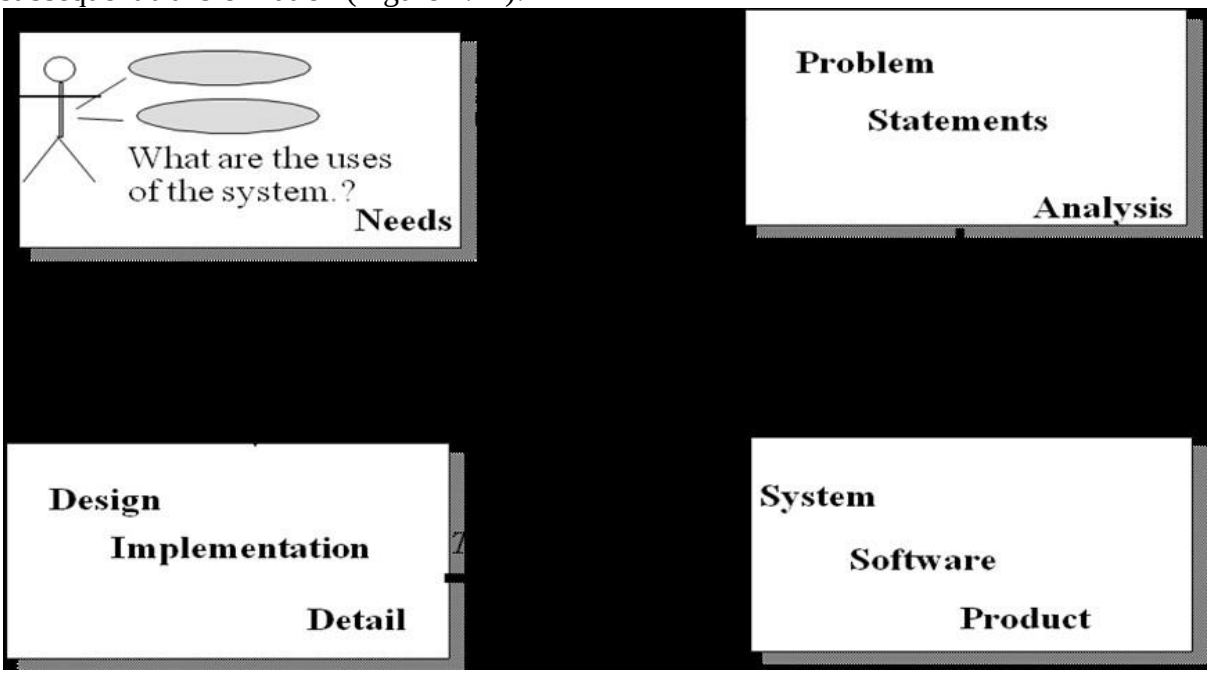


Fig 1.12: Software process reflecting transformation from needs to a software productthat satisfies those needs.

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 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.

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 1.13).

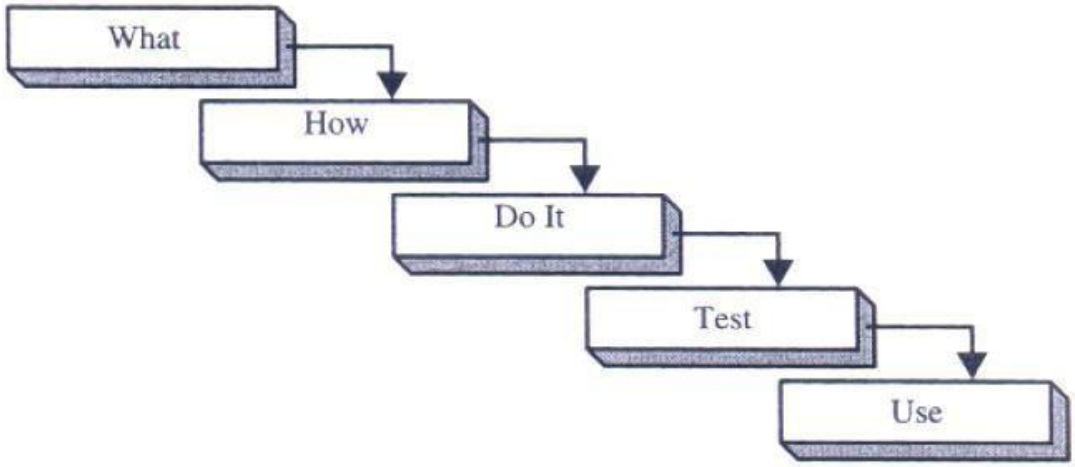


FIGURE 1.13 The waterfall software development process.

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 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 tracedto 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.

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.

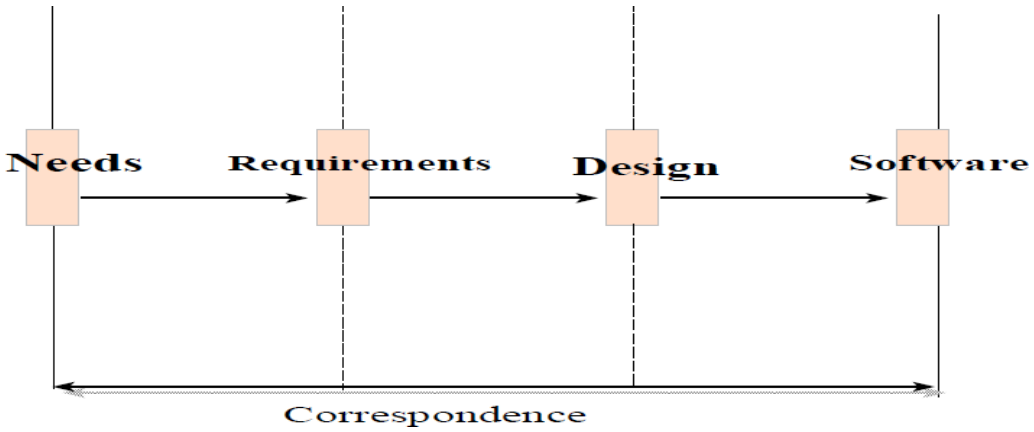
There are two basic approaches to systems testing.

- We can test a system according to how it has been built
- or, alternatively, we can test the system with respect to what it should do.

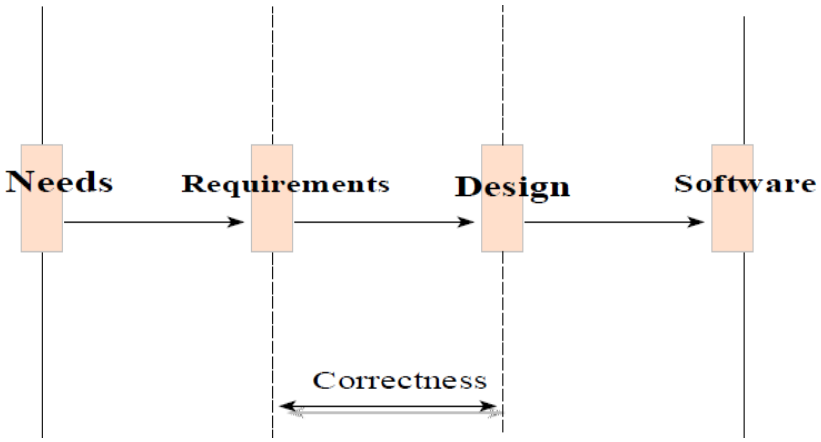
Blum describes a means of **system evaluation in terms of four quality measures:**

- **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. *It cannot be determined until the system is in place.*

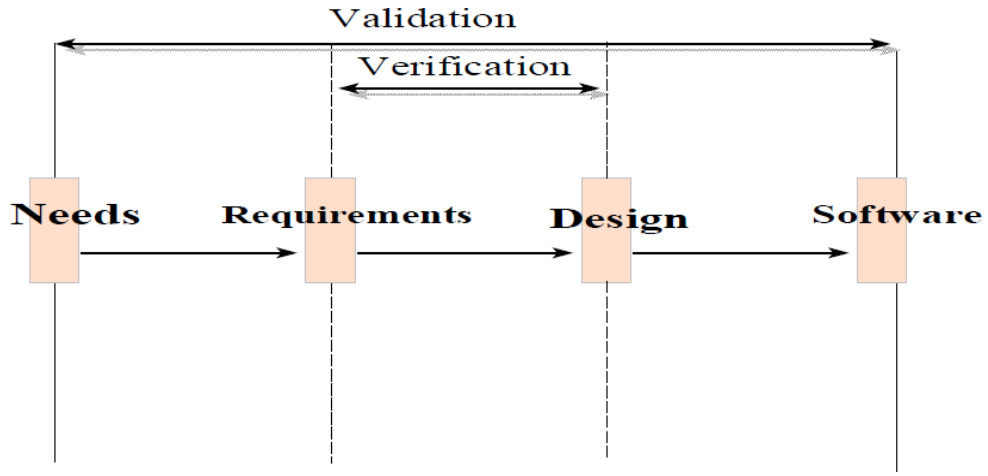


Correctness measures the consistency of the product requirements with respect to the design specification.



***Verification* is to predict the 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.

Validation is to predict the correspondence. True correspondence cannot be determined until the system is in place.



Verification - "Am I building the product right?"
Validation - "Am I building the right product?"

OB.JECT ORIENTED SYSTEMS DEVELOPMENT: A USE-CASE DRIVEN APPROACH

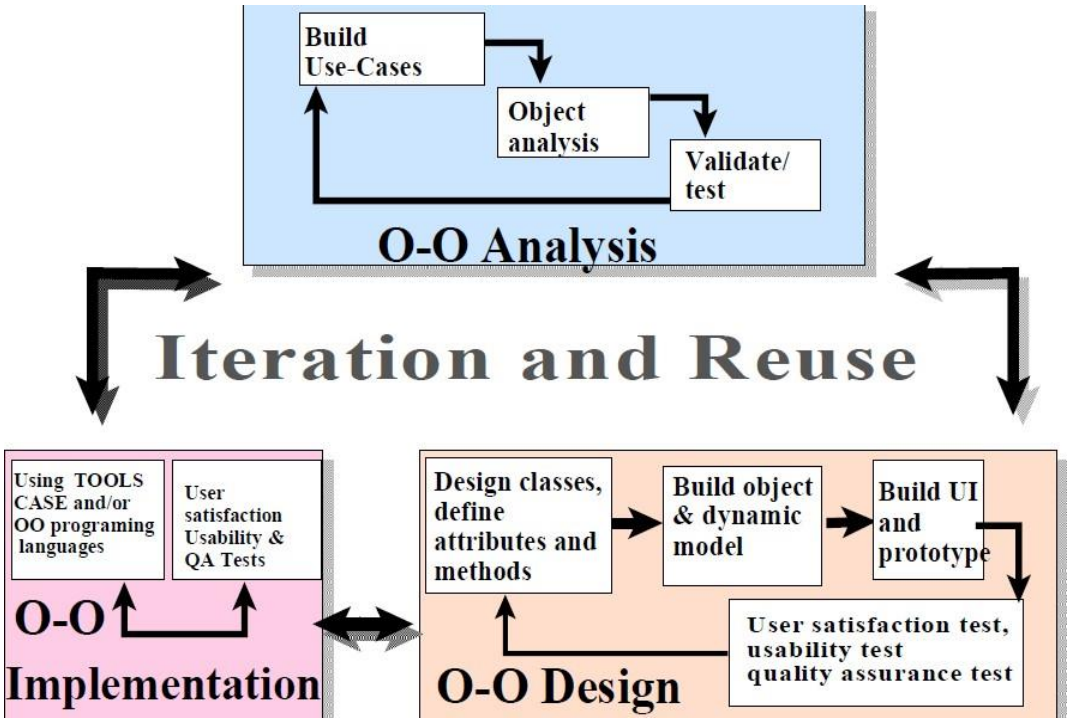


FIGURE 1.14 The object-oriented systems development approach. Object- oriented analysis corresponds to transformation 1;design to transformation 2, and implementation to transformation 3 of Figure 1.13.

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

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 1.15). The main advantage is that all design decisions can be traced back directly to user requirements. Usage scenarios can become test scenarios.

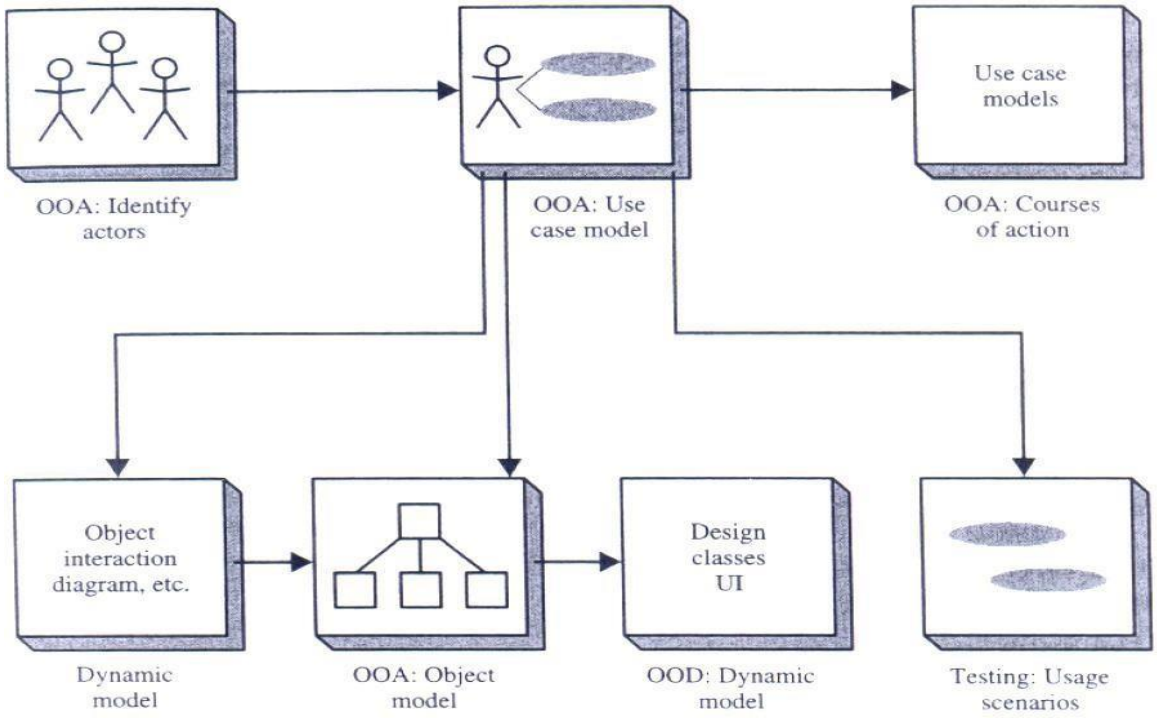


Fig 1.15: By following the life cycle model of Jacobson et aI., we produce designs that are traceable across requirements, analysis, implementation, and testing.

Object-Oriented Systems Development activities

- Object-oriented analysis.
- Object-oriented design.
- Prototyping.
- Component-based development.
- Incremental testing.

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 re1ationship 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.

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.*

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.

This process of developing uses 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.

Object-Oriented Analysis

OO analysis concerns with determining the system requirements and identifying classes and their relationships that make up an application.

Object-Oriented Design

The goal of object-oriented design (OOD) is to design

• The classes identified during the analysis phase,

• The user interface and

• Data access.

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 ofthe system. The activities and focus of object-oriented analysis and object-oriented design are intertwined-grown, not built.

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 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.

PROTOTYPING

- 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 can also give users a chance to comment on the usability and usefulness of the design.
- The main idea here is to build a prototype with uses-case modeling to design systems that users like and need.

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

Types of Prototypes

- **A *horizontal prototype*** is a simulation of the interface. 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.
- 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 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.

IMPLEMENTATION: COMPONENT-BASED DEVELOPMENT

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.

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.

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, design, and programming, whereas component-based development is concerned with the implementation and system integration aspects *of* software development.

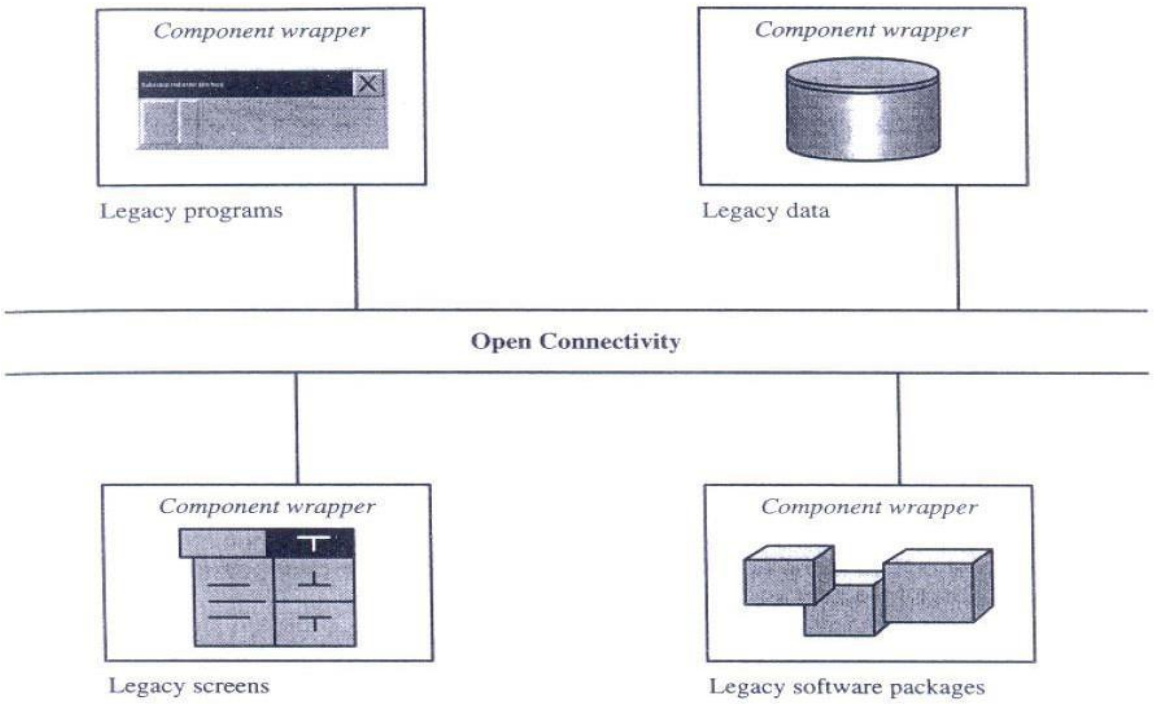


Fig. 1.16 Reusing legacy system via component wrapping technology.

Rapid Application Development (RAD)

RAD is a set of tools and techniques that can be used to build an application faster than typically possible with traditional methods. To achieve RAD, the developer sacrifices the quality *of* the product for a quicker delivery.

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.

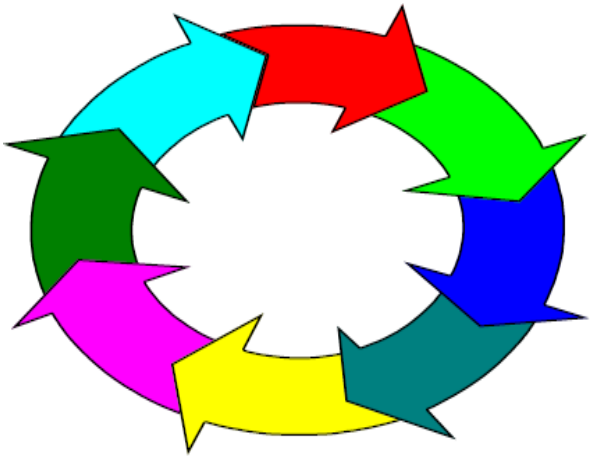
RAD does not replace SDLC but complements it, since it focuses more on process description and can be combined perfectly with the object-oriented approach. The task *of* RAD is to build the application quickly and incrementally implement the design and user requirements, through tools such as Delphi, 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.

Incremental Testing

• Software development and all of its activities including testing are an iterative process.

• 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.



Reusability A major benefit of object-oriented systems development is reusability, and this is the most difficult promise to deliver on.

Reuse strategy

- 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., 50percent reuse of objects).

. The essence of the software process is the transformation of users' needs into a software solution. The O-O SDLC is an iterative process and is divided into analysis, design, prototyping/implementation,and testing.

Questions

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Justify the need of object orientation.	Evaluate	BTL 5
2.	Explain the importance of unified approach	Remember	BTL 1
3.	Distinguish Static and Dynamic binding.	Understand	BTL 2
4.	List out the various System development activities?	Remember	BTL 1
5.	What is meant by Inheritance.	Remember	BTL 1
6.	Define Classes	Remember	BTL 1
7.	What is Analysis and Design?	Remember	BTL 1
8.	What is the main advantage of Object Oriented Development?	Remember	BTL 1
9.	Illustrate the concepts of Association Relationship.	Apply	BTL 3
10.	Compare Verification and Validation.	Analysis	BTL 4
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Explain object oriented system with reference to class, object, encapsulation, abstraction, message, inheritance, interface and polymorphism with suitable examples.	Remember	BTL 1
2.	Demonstrate in detail about Object Relationships and Associations.	Apply	BTL 3
3.	Explain in detail about object oriented system development life cycle.	Remember	BTL 1
4.	(i)Describe the concept of Prototyping and explain the types of prototyping? (ii)Explain about Class Hierarchy.	Remember	BTL 1
5.	Write short notes on a. RAD b. CBD	Remember	BTL 1
6.	(i)Compare and Contrast Traditional development methodologies and Object Oriented System. (ii)Explain about Inheritance.	Evaluate	BTL5
7.	(i)List out the Quality measures in building High quality Software and explain. (ii)Discuss about Aggregation.	Remember	BTL 1

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – II - OBJECT ORIENTED ANALYSIS AND SYSTEM ENGINEERING - SCSA1401

UNIT 2

OBJECT ORIENTED METHODOLOGIES

Rumbaugh et al.'s Object Modeling Technique - Booch Methodology - Jacobson et al. Methodologies – Patterns - Framework - Unified approach - Unified Modeling Language: Static and Dynamic Model - UML Diagrams - UML Class Diagram – UML Use Case –Case study- Use case Modelling – Relating Use cases – include, extend and generalization – When to use Use-cases- UML Dynamic Modeling – Case study- UML Extensibility - UML Metamodel.

OBJECT –ORIENTED METHODOLOGIES

INTRODUCTION

- Object-oriented methodology is a set of methods, models, and rulesfor 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.

TOWARD UNIFICATION-TOO MANY METHODOLOGIES

- 1986 - Booch developed t e object-oriented design concept, the Boochmethod.
- 1987 - Sally Shlaer and Steve Mellor created the concept of therecursive 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 GeneralElectric to develop the object modeling technique (OMT) .
- 1991 -Peter Coad and Ed Yourdon developed the Coad lightweightand Prototypeoriented approach to methods.
- 1994. Ivar Jacobson introduced the concept of the use case andobject-oriented software engineering (OOSE).

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.
- In the following sections, we look at the methodologies and their modeling notations developed by Rumbaugh et aI., Booch, and Jacobson which are the origins of the Unified Modeling Language (UML).
- **Each method has its strengths. The Rumbaugh et ai. 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**.

RUMBAUGH'S OB.JECTMODELING 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 asystem 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 (*Object Modeling Technique)* describes a method for the analysis, design, and implementation of a system using an object-oriented technique. Class attributes, method, inheritance, and association also can be expressed easily

- 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 ofthe system along* with high-level strategy decisions.

3. ***Object design****. This phase produces a design document, consisting ofdetailed objects* static, dynamic, and functional models.

4. ***Implementation.*** *This activity produces reusable, extendible, androbust 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 flowdiagrams.*

3. A *functional model, presented by data flow and constraints.*

2.3.1 THE OBJECT MODEL

- The object model describes the structure of objects in a system: theiridentity, relationships to other objects, attributes, and operations.
- The object model is represented graphically with an object diagram(see Fig: 1).
- 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 oneclass to the objects of another class.

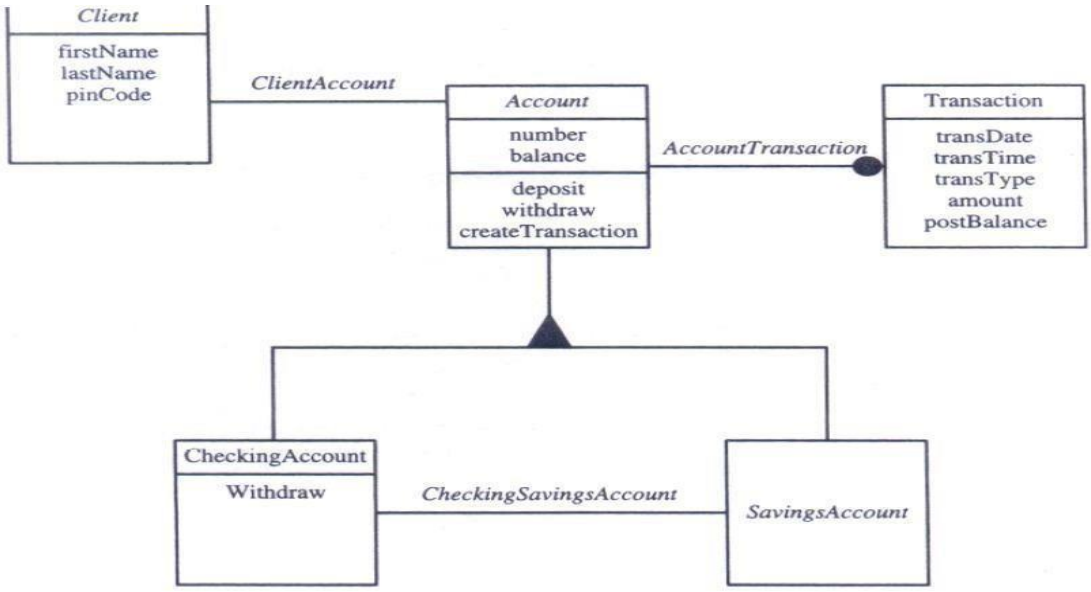


Fig 1: OMT object model of a bank system

Boxes- represents classes, Filled Triangle – represents Specialization, Association between account and transaction represents one to many, Filled Circle – represents many(zero or more) . Association between Client and Account represents one to one.

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. 2).
- Each state receives one or more events, at which time it makes thetransition to the next state.
- The next state depends on the current state as well as the events.

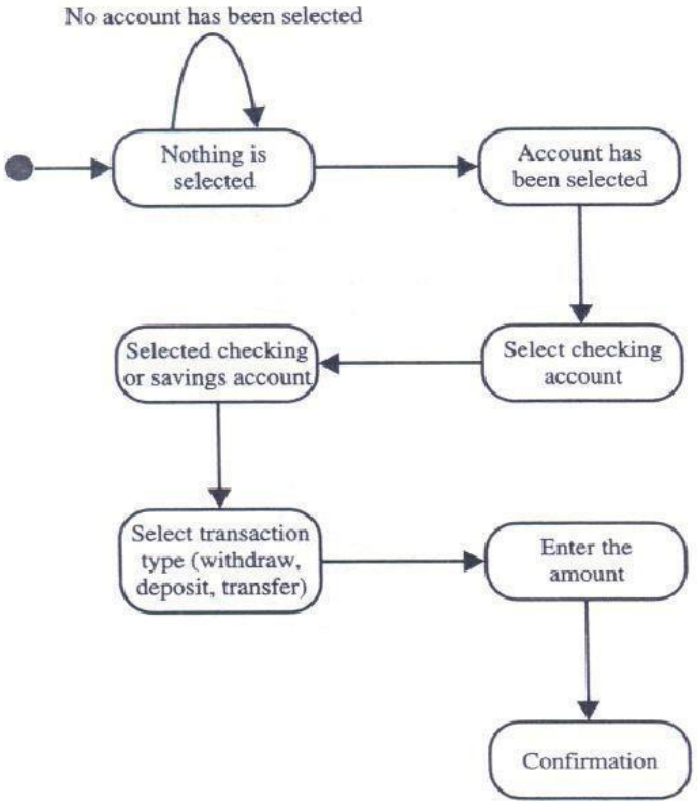


Fig. 2 : State transition diagram for the bank application user interface.The round boxes represent states and the arrows represent transitions.

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 simpleand 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, verifyPassword or* PIN in the ATM system (see Fig .3).
2. The ***data flow*** *shows the direction of data element movement; forexample, PIN code.*
3. The ***data store*** *is a location where data are stored; for example, accountis a data store* in the ATM example.
4. An ***external entity*** *is a source or destination of a data element; forexample, the ATM* card reader.

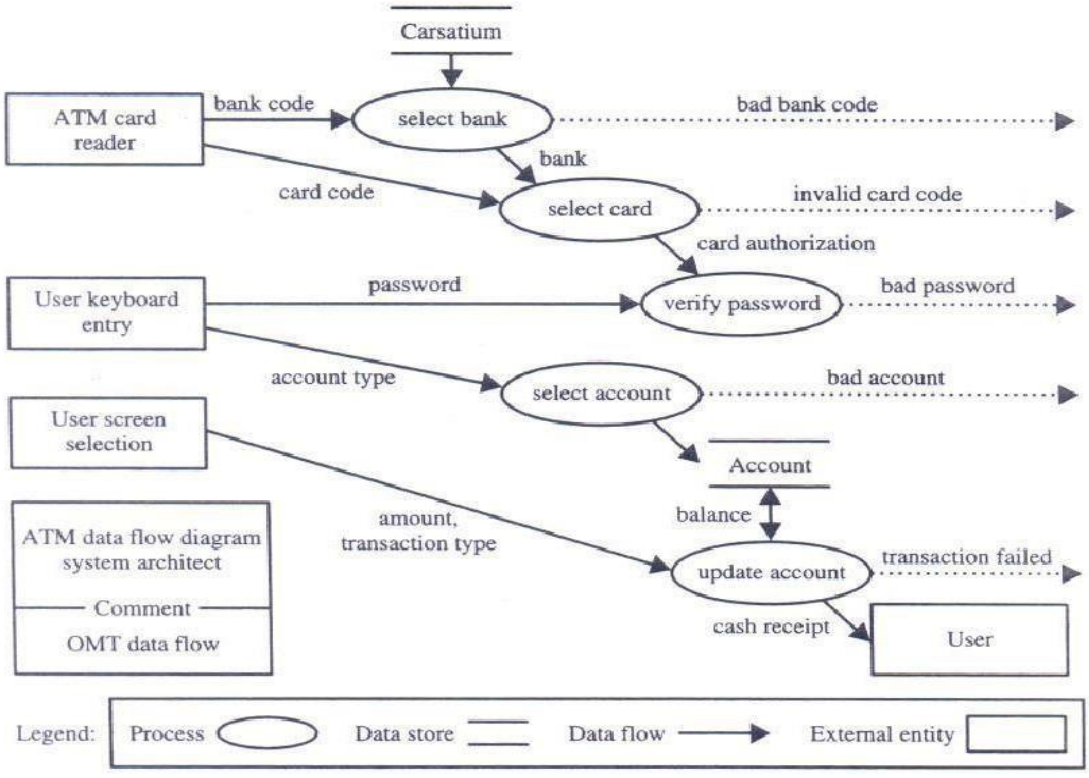


Fig 3: OMT DFD of the ATM system.

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

BOOCH METHODOLOGY

- The Booch methodology is a widely used object-oriented method thathelps you design your system using the object paradigm.
- It covers the analysis and design phases of an object-oriented system.
- The Booch method consists of the following diagrams:
  - Class diagrams
  - Object diagrams
  - State transition diagrams
  - Module diagrams
  - Process diagrams
  - Interaction diagrams
- The Booch methodology prescribes a **macro development process** and **a micro development process**.

THE MACRO DEVELOPMENT PROCESS

- The macro process serves as a controlling framework for the microprocess and can take weeks or even months.
- The primary concern of the macro process is technical management ofthe system.
- The macro development process consists of the following steps**:**

1. ***Conceptualization.*** *During conceptualization, 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, use the class diagram to* describe the roles and responsibilities objects are to carry out in performing the desired behavior of the system. Then,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, use the class* diagram to decide what classes exist and how they relate to each other. Next, use the object diagram to decide what mechanisms are used to regulate how objects collaborate. Then, use the module diagram to map out where each class and object should be declared. Finally, 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 throughmany 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.

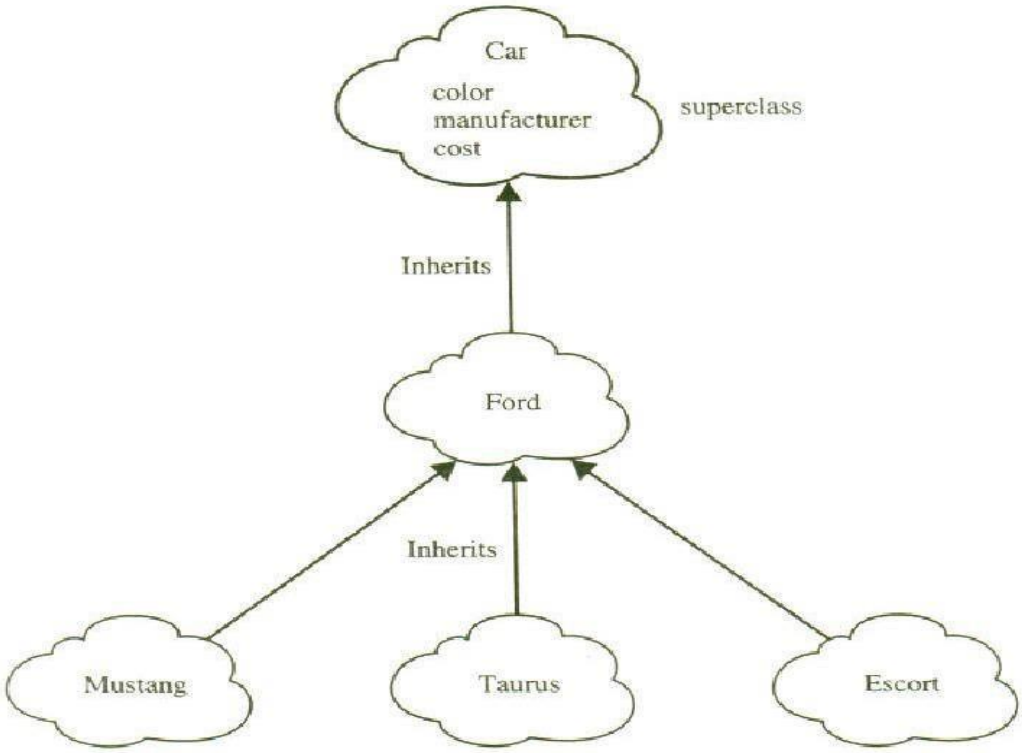


Fig .4 : Object modeling using Booch notation

The arrows represent specialization; for example, the class Taurus issubclass of the class Ford.

THE MICRO DEVELOPMENT PROCESS

- Each macro development process has its own micro developmentprocesses.
- 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.
- 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.***

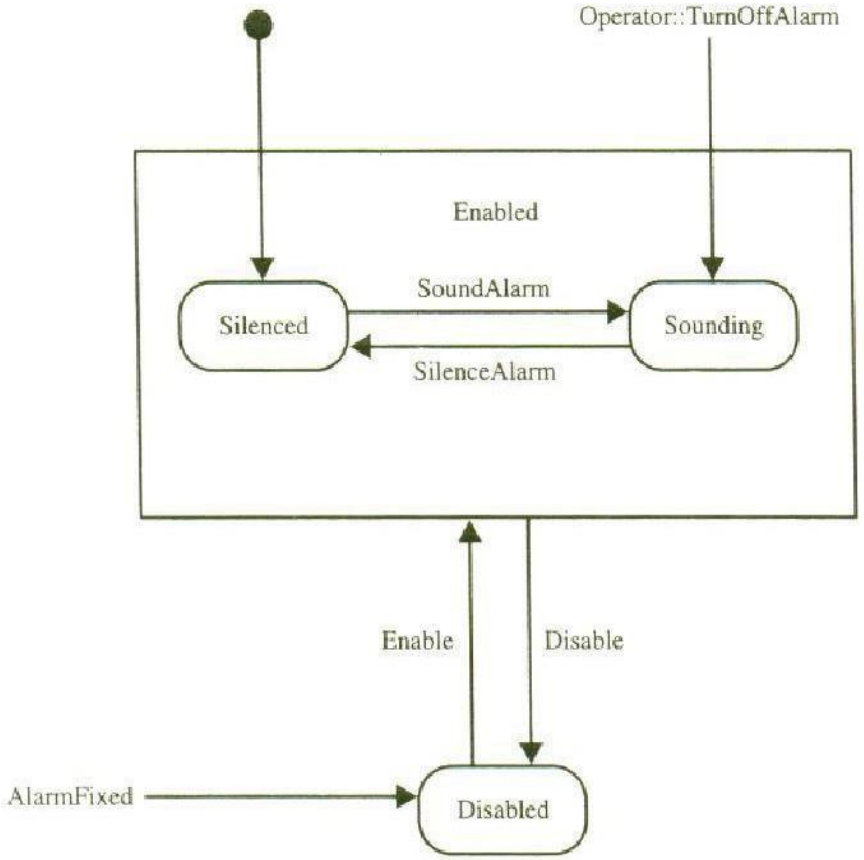


Fig. 5: An alarm class state transition diagram with Booch notation.

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 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).

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.(Fig.6)

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.

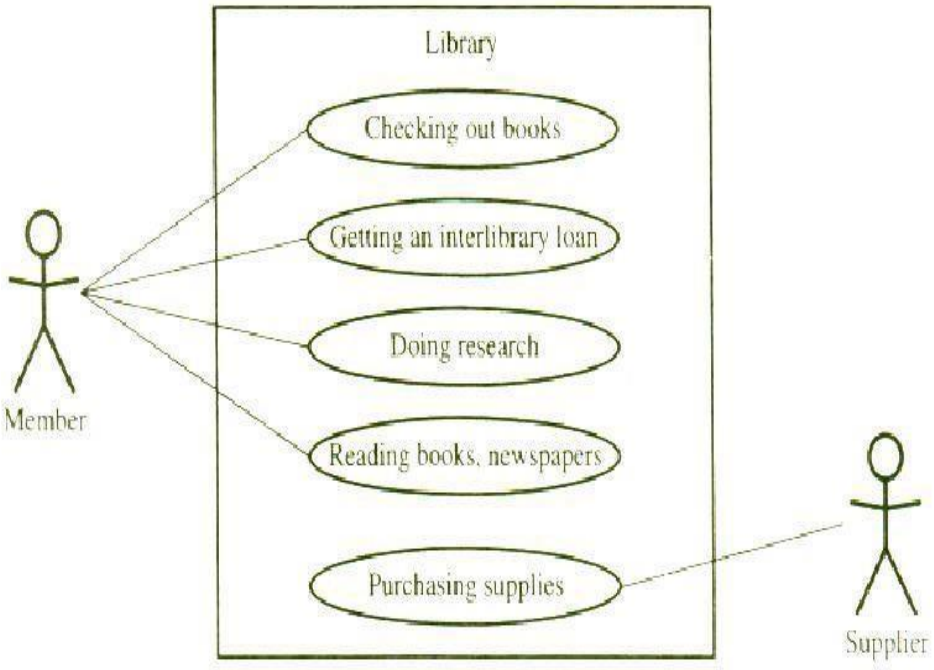


Fig.6 : Some Uses of A Library.

In the requirements analysis, the use cases are described as one of thefollowing :

- ✓ 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 whenthe 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 additionalone.
- The use-case model employs extends and uses relationships. The extends relationship is used when you have one use case that is similarto 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.

OBJECT-ORIENTED SOFTWARE ENGINEERING: OBJECTORY

- Object-oriented software engineering (OOSE), also called ***Objectory, is a method*** of object-oriented development with the specific aim tofit the development of large, realtime systems.

- The development process, called ***use-case driven development, stresses that*** use cases are involved in several phases of the

development (see Fig. 7), including analysis, design, validation, andtesting.

- The use-case scenario begins with a user of the system initiating a sequence of interrelated events.

The use-case model is considered in every model and phase.

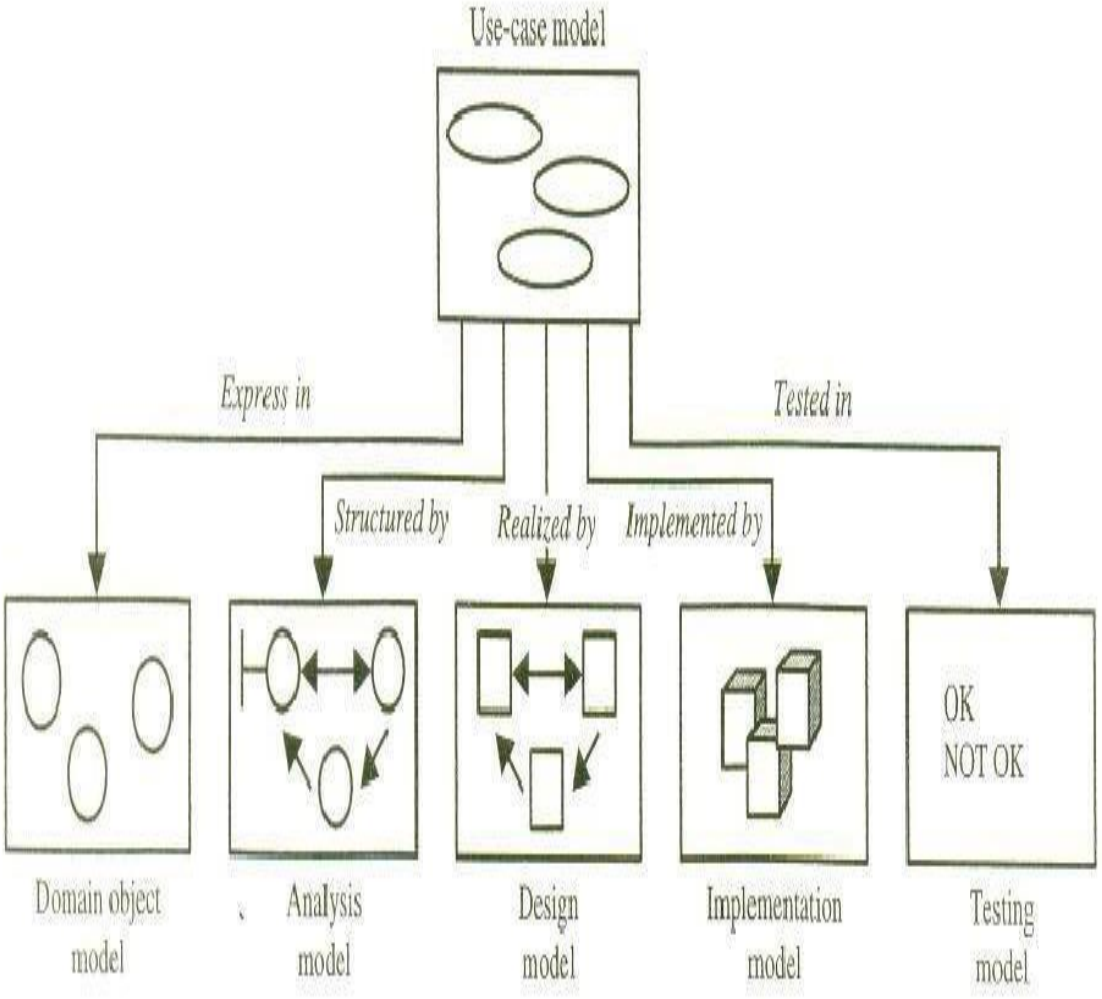


Fig. 7: The use case model is considered in every model and phase.

- 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.
- 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) andinside (use case) of* the system's behavior.
- ***Domain object model.*** *The objects of the "real" world are mappedinto the domain object* model.
- ***Analysis object model.*** *The analysis object model presents how thesource 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.*

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 process

1. ***Analysis phase.*** It 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 helpspecify user interfaces.

2. ***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 fitthe current implementation environment.

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

PATTERNS

- 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.
- 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 objectoriented 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.

- The main idea behind using patterns is to provide documentation to help categorize and communicate about solutions to recurringproblems.
- The pattern has a name to facilitate discussion and the information it represents.
- Riehle and Ztillighoven:- **A *pattern is [an] instructive information that captures the essential structure and* insight of a successfulfamily of proven solutions to a recurring problem that ariseswithin a certain context and system of forces.**
- 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.
- 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***.
- **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.

Generative and Non generative Patterns

- **Generative patterns** are patterns that not only describe a recurring problem, they can tell us how to generate something and can beobserved 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.
- 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**.**

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" .

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.
- ***Problem.*** A statement of the problem that describes its intent: the goals and objectives it wants to reach within the given context and forces.
- ***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 wishto achieve (perhaps with some indication of their priorities). A concrete scenario that serves as the motivation for the pattern frequently is employed.
- ***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. Thedescription may encompass pictures, diagrams, and prose that identifythe 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.

- ***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.
- ***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.
- ***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."

- ***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.
- **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.

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 seeand understand bad solutions as to see and understand good ones.

Capturing Patterns

- Writing good patterns is very difficult, explains Appleton. Patterns should provide not only facts 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).**

Guidelines

- ***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. -*** *T*o 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.

FRAMEWORKS

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.

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 severaldesign patterns.

In fact, a framework can be viewed as the implementation of a systemof design patterns.

Differences between frameworks and design patterns:

*. A framework is executable software, whereas design patterns representknowledge and experience about software.

*. Frameworks are of a physical nature, while patterns are of a logicalnature.

*. Frameworks are the physical realization of one or more software patternsolution; patterns are the instructions for how to implement those solution.

Gamma et al. describe the major differences between design patterns andframeworks 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. Astrength 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.

THE UNIFIED APPROACH

- The unified approach (UA) 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 the following processes and concepts (see Fig.8). The processes are:
    - Use-case driven development
    - Object-oriented analysis
    - Object-oriented design
    - Incremental development and prototyping
    - Continuous testing The

methods and technology employed include

1. Unified modeling language used for modeling.
2. Layered approach.
3. Respository for object oriented system development patterns and frameworks.
4. Component based development.

The UA allows iterative development by allowing us to go back andforth between the design and the modeling or analysis phases.

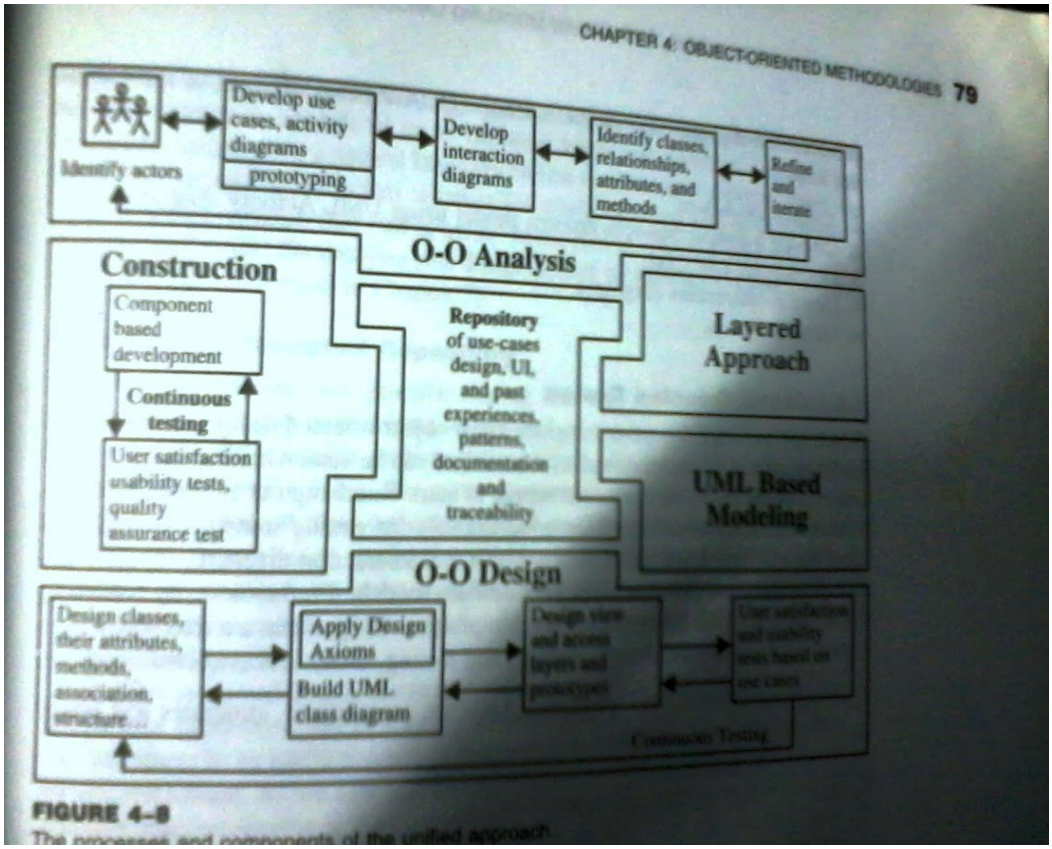


Fig 8. The Process and components of the unified approach2.8.1

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.

OBJECT-ORIENTED DESIGN

Booch 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 fig 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,

Booch'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.

Process consists of:

- Designing classes, their attributes, methods, associations, structuresand 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

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.

MODELING BASED ON THE UNIFIED MODELINGLANGUAGE

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 theanalysis and design phases of system development.

The UA Proposed Repository

In modem 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 witha 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 objectoriented 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.

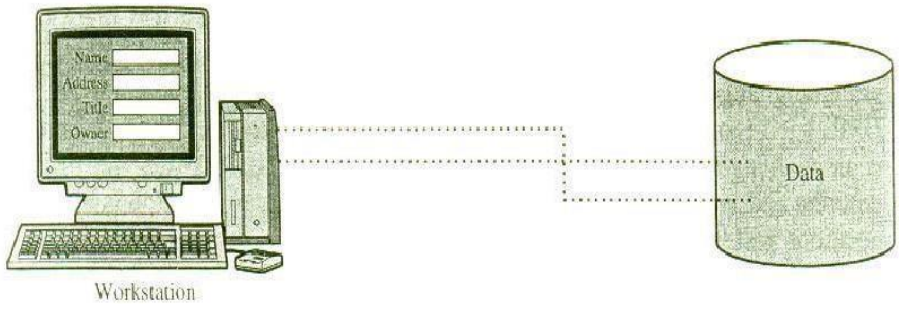


Fig. 9 : Two-layered architecture: interface and data.

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.

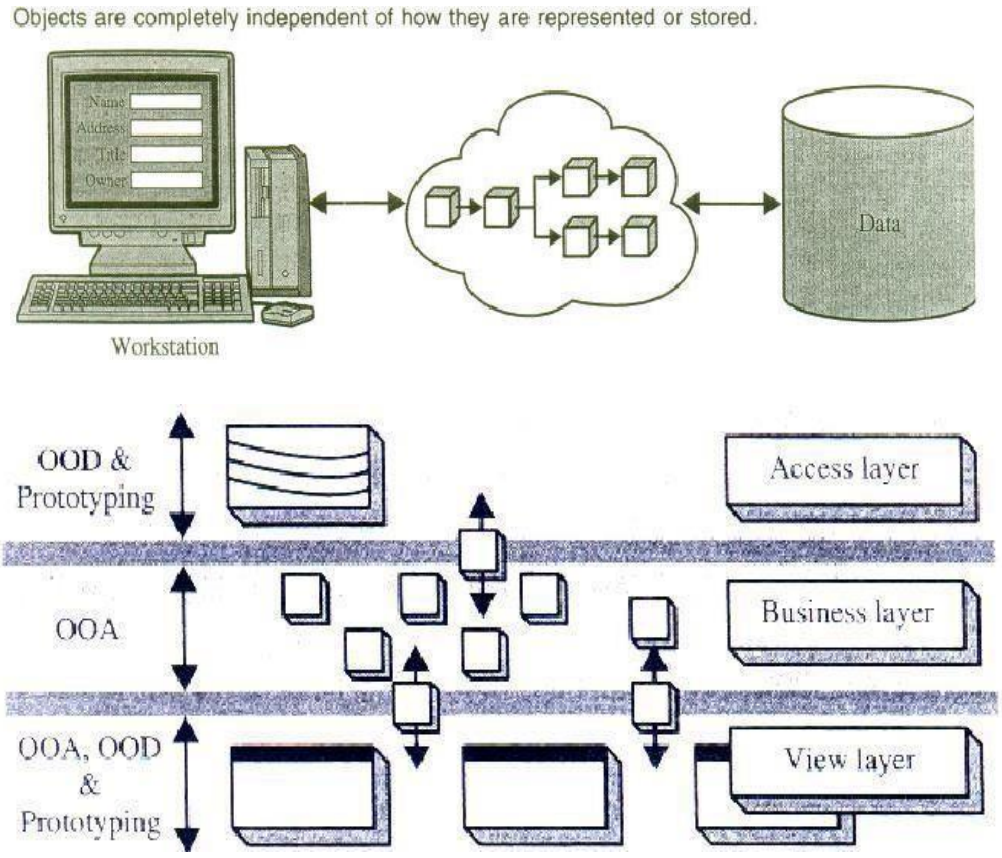
The repository should be accessible to many people. Furthermore, it should be relatively easy to search the repository for classes based on their attributes, methods, Two-layered architecture: interface and data, or other characteristics. For example, application developers could select prebuilt components 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 softwaredevelopment 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 thebusiness logic must be accomplished in every screen that deals with that portion of the business. This approach resultsin 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).



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. 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 modem objectoriented 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 businessobjects.

The User Interface (View) Layer: The user interface layer consists ofobjects 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.

The Access layer : The access layer contains objects that know how to communicate with the palce where the data actually reside, whether it be a relational databasem mainframe, internetm or file. The Access layer has 2 major responsiblities.

1. Translate request : The access layer must be able to translate any dat- related requests from the business layer into the appropriate protocol for data access.
2. Translate results/: tje access ;ayer also must ne able to translate the dat retrieved back into the appropriate business objects and pass those objects back up into the business layer.

   Access objets are indentified during object oriented design.

UNIFIED MODELING LANGUAGE

A ***model*** is an abstract representation of a system, constructed to understand the system prior to building or modifying it. Most of the modeling techniques involve graphical languages.

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) andinside (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 theimplementation of the system.

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

Modeling is an iterative process.

Static or Dynamic Models

Static Model	*Dynamic Model*
• A static model can be viewed as "snapshot" of a system's parameters at rest or at a specific point in time. • The classes'| structure and their relationships to each other frozen in time are examples of static models.	• Is a collection of procedures or behaviors that, taken together, reflect the behavior of a system over time. • For example, an order interacts with inventory to determine product availability.

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 graphicalor 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 main tainability 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 ofsimilar experimentation conducted with a real system.
5. Manipulation of the model (changing variables) is much easier than manipulating a real system.

key ideas regarding modeling:

- A model is rarely correct on the first try.
- Always seek the advice and criticism of others. You canimprove a model by reconciling different perspectives.
- Avoid excess model revisions, as they can distort the essence ofyour model.

What Is the UML?

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.

What it is/isn't? Is NOT

- A process
- A formalism

Is

- A way to describe your software
- more precise than English
- less detailed than code

What is UML Used For?

- Trace external interactions with the software
- Plan the internal behavior of the application
- Study the software structure
- View the system architecture
- Trace behavior down to physical components

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 extendthe 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.

UML DIAGRAMS

The UML defines nine graphical diagrams:

1. Class diagram (static)

2. Use-case diagram

3. Behavior diagrams (dynamic):

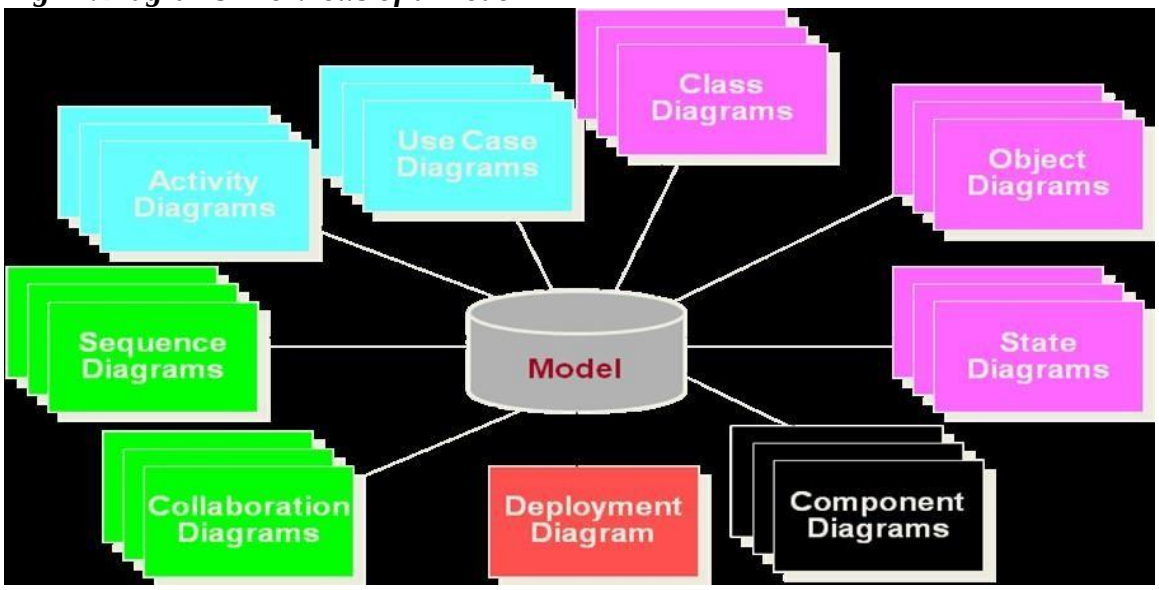
- **3.1. Interaction diagram:**
  - **3.1.1. Sequence diagram**
  - **3.1.2. Collaboration diagram**
- **3.2. State chart diagram**
- **3.3. Activity diagram**

4. Implementation diagram:

Component diagram

Deployment diagram

Fig 11.Diagrams Are Views of a Model



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 themodel.

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 requiredin dynamic modeling.

- A class diagram describes the types of objects in the system and thevarious kinds of static relationships that exist among them.
- A graphical representation of a static view on declarative staticelements.
- A central modeling technique that runs through nearly all object-oriented methods.
- The richest notation in UML.
- A class diagram shows the existence of classes and their relationshipsin the logical view of a system

Class Notation: Static Structure

A class is drawn as a rectangle with three components 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 Fig12.).

Either or both the attribute and operation compartments may be suppressed. Aseparator line is not drawn for a missing compartment if a compartment is suppressed; no inference can be drawn about the presence orabsence of elements in it.

The class name and other properties should be displayed in up to threesections. A stylistic convention of UML is to use an italic font for abstract classes and a normal (roman) font for concrete classes.

Essential Elements of a UML Class Diagram

- Class
- Attributes
- Operations
- Relationships
  - Associations
  - Generalization
- Dependency
- Realization

Constraint Rules and Notes

A class is the description of a set of objects having similar attributes,operations, relationships and behavior.

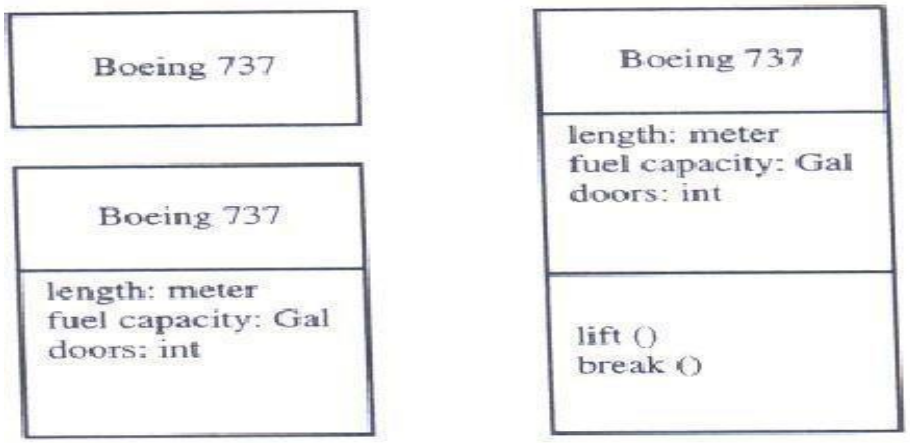


Fig.12 In class notation, either or both the attributes and operation compartments may be suppressed.

Attributes

– Classes have attributes that describe the characteristics of their objects.

– Attributes are atomic entities with no responsibilities.

– Attribute syntax (partial):

○ [visibility] name [: type] [= defaultValue]

– Class scope attributes are underlined

Visibility

• Visibility describes whether an attribute or operation is visible and can bereferenced from classes other than the one in which they are defined.

• language dependent

○ Means different things in different languages

• UML provides four visibility abbreviations: + (public) – (private) #(protected) ~ (package)

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.

UML modeling elements in class diagrams

• Classes and their structure, association, aggregation, dependency, andinheritance relationships

• Multiplicity and navigation indicators, etc.

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 dependentclass 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 Fig13. with UML class interface notation.

Fig 13. Interface notation of a Class

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. 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*** **(**see Fig14).

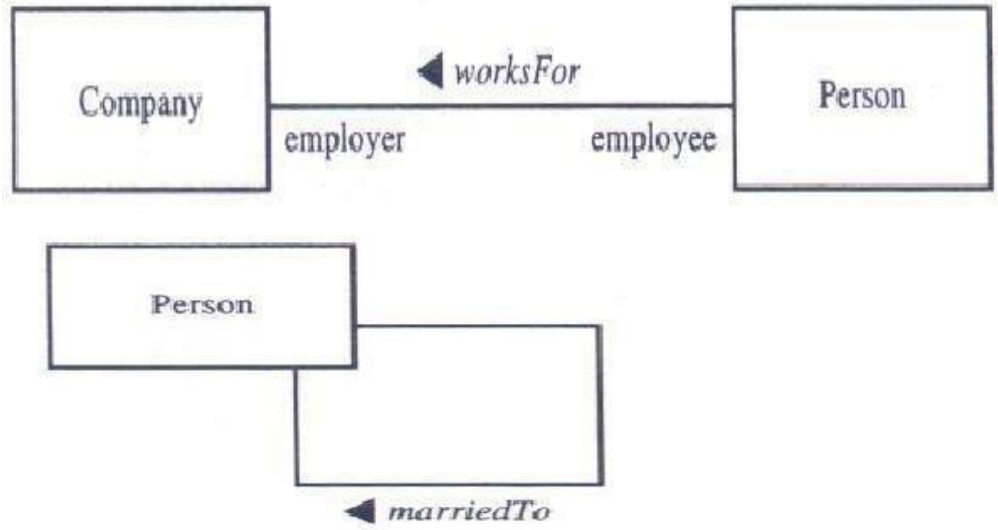


Fig 14: Association notation.

Association Role

A simple association- ***binary association****-is* drawn as a solid lineconnecting 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 part of the class. Each association has two or more roles to which it is connected.

In above Fig14. 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.15. Navigability is visually distinguished from inheritance, which is denoted by an unfilled arrowhead symbol near the superclass.

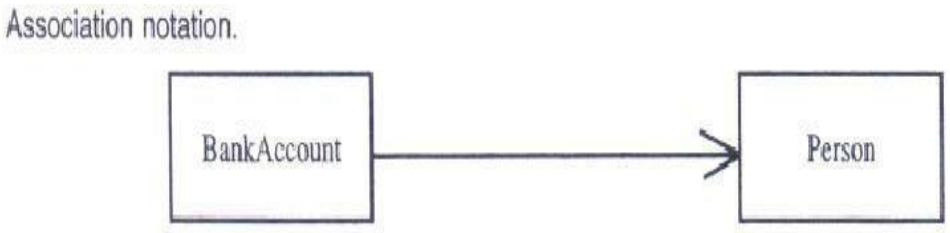


Fig 15. Association Notation

In this example, 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.(Fig 16)

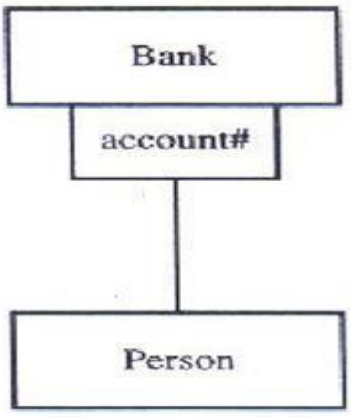


Fig 16 Association Qualifier.

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 above 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 17):

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..*

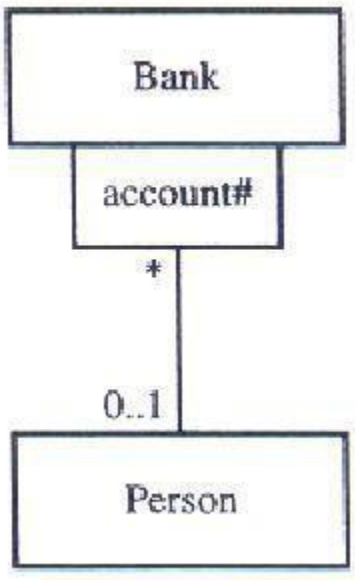


Fig 17. Assocaition Qualifier and its multiplicity.

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 18). In other words, any instance of the class may participate in, at most, one of the associations at one time.

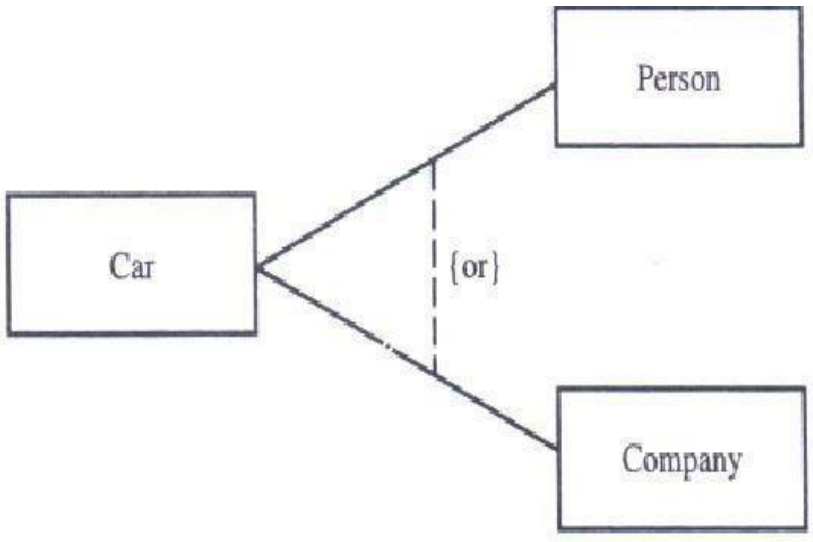


Fig 18. An OR association notation. A car may associate with a person or a company.

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 path. The name in the class symbol and the name string attached to the association path are the same (see Fig 19). The name can be shown on the path or the class symbol or both. If an ssociation 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."

Association class.

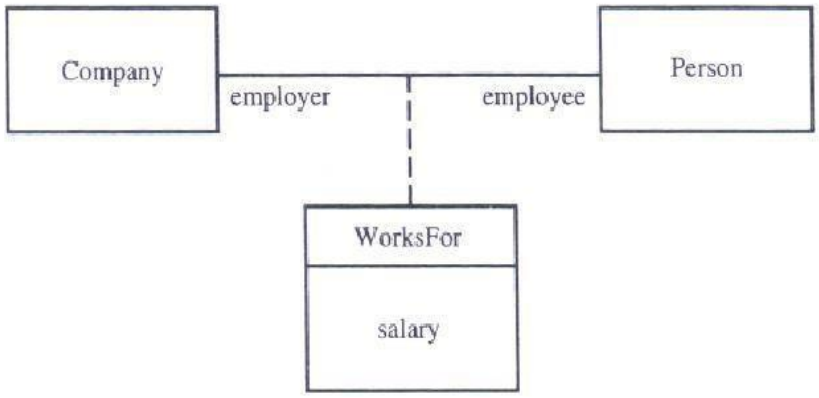


Fig 19. Association Class

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.

An n-ary association is shown as a large diamond with a path from thediamond 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 diamond by a dashed line, indicating an n-ary association thathas attributes, operation, or associations. The example depicted in Fig 20 shows the grade book of a class in each semester.

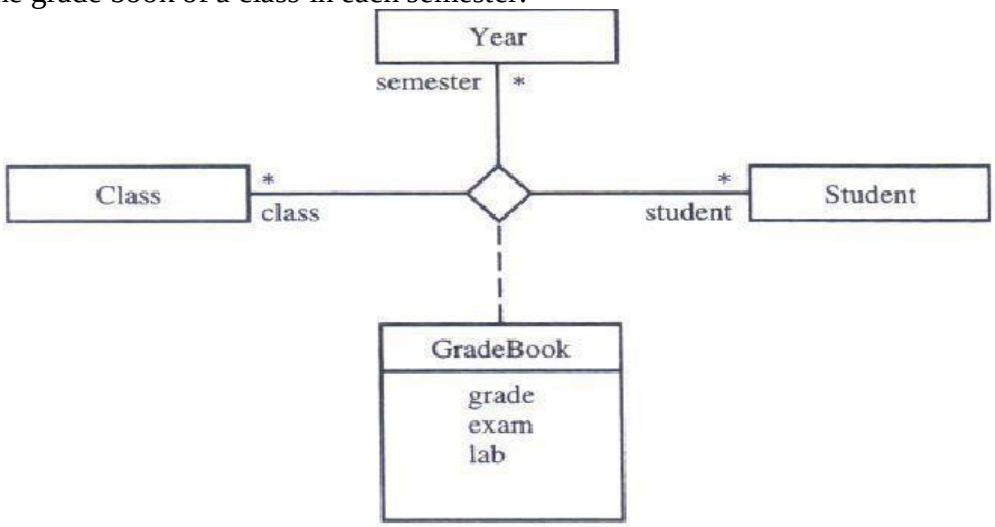


Fig 20 . An n-ary (ternary) association that shows association among class, year, and student classes. The association class GradeBook which contains the attributes of the associations such as grade, exam, and lab.

Aggregation and Composition (a.part.of)

Aggregationis 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 neednot be presentedat all (see Fig 21).

Composition, also known as the ***apart-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 nestedform that, in many cases,is more convenientfor showingcomposition (see Fig 22).

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.

Fig 21. Association Path

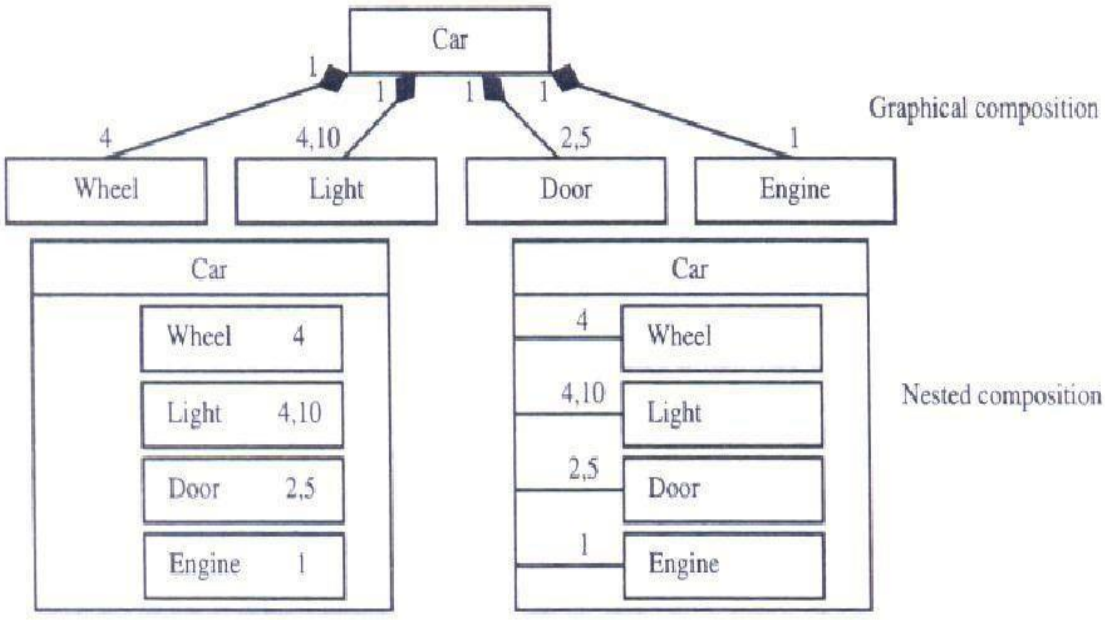


Fig 22. Different ways to show Composition.

Generalization

***Generalization* is** the relationshipbetweena more generalclass and a more specific class. Generalization is displayed as a directed line with a closed, hollow arrowhead at the superclass end (see Fig 23). 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 24).

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.

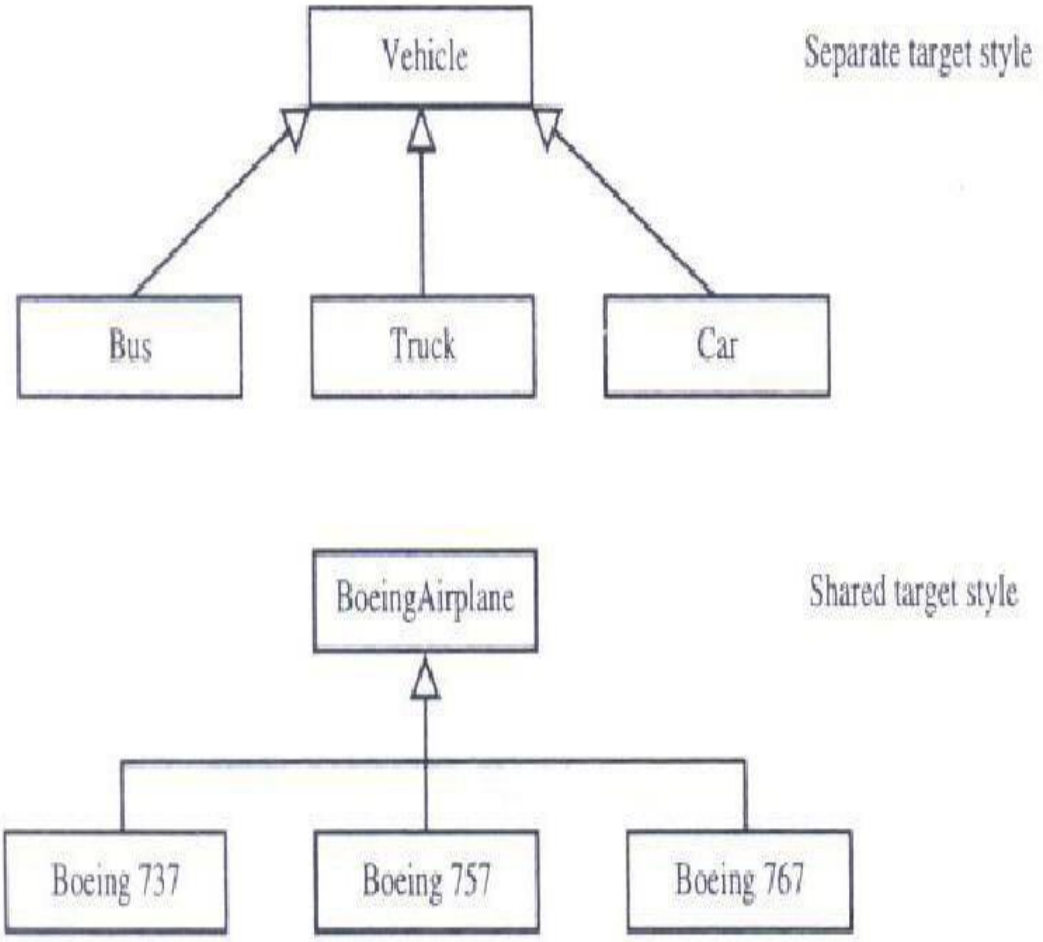


Fig 23. Generalization Notation

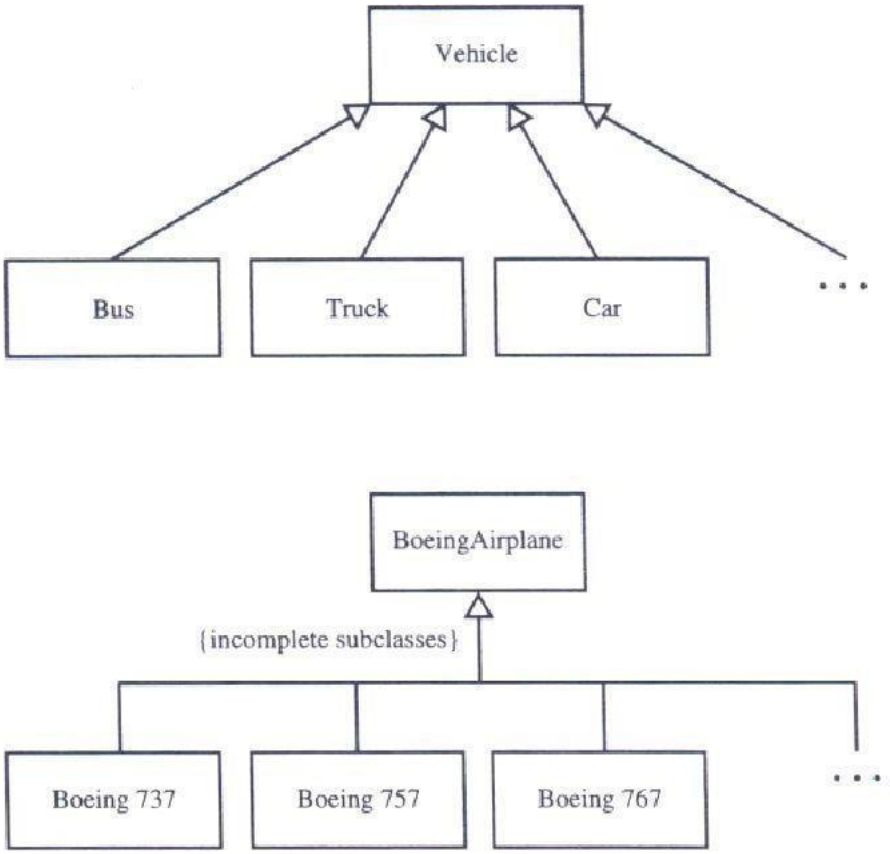


Fig 24. Ellipses(...) indicate that additional classes exist and are notshown.

USE-CASE 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.

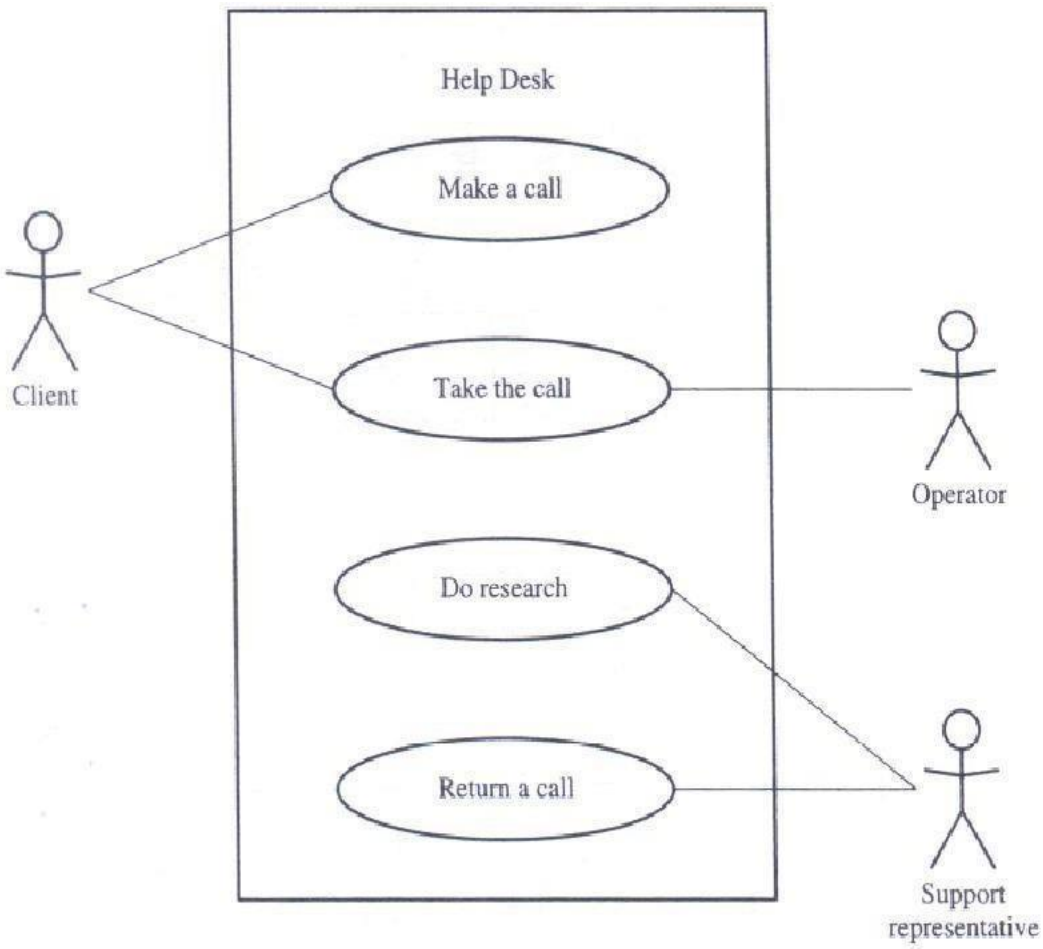


Fig 25. use-case diagram shows the relationship among actors and usecases within a system.

Fig 25. 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 fig, or just the stick fig with the name of the actor belowthe fig (see Fig 26).

Fig 26. 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.

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 mediumsuch as on a sheet of paper is difficult, but the problem confronts almost every discipline.

The Dynamic semantics of a problem with the following diagrams:

Behavior diagrams (Dynamic)

- Interaction Diagrams:
  - ✓ Sequence diagrams
  - ✓ Collaboration diagrams
- State Chart diagrams
- Activity diagrams

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

UML INTERACTION DIAGRAMS

Interaction diagrams are diagrams that describe how groups of objectscollaborate 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 interactionbetween 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 ina time sequence.

A sequence diagram has two dimensions: the **vertica**l dimension represents time, the **horizontal** dimension represents different objects. The vertical line is called the **object's *lifeline***. The ***lifeline*** represents the object'sexistence 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 27). 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. 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.

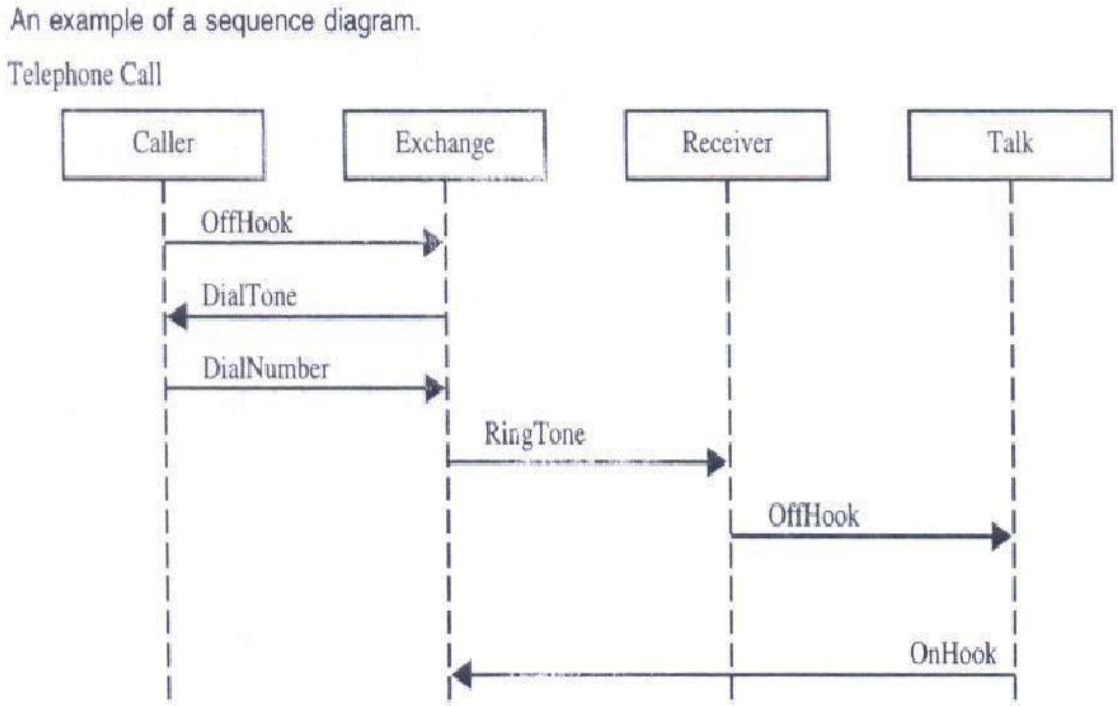


Fig 27. An example of a Sequence Diagram

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 caninclude 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 atthe 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 themessage sent within the given use case. In a collaboration diagram, the sequence is indicated by numbering the messages.

A collaboration diagram provides several numbering schemes. The simplest is illustrated in Fig 28. You can also use a decimal numbering scheme (see Fig- 28 a) where 1.2: DialNumber means that the Caller (1) is calling the Exchange (2); hence, the number 1.2.

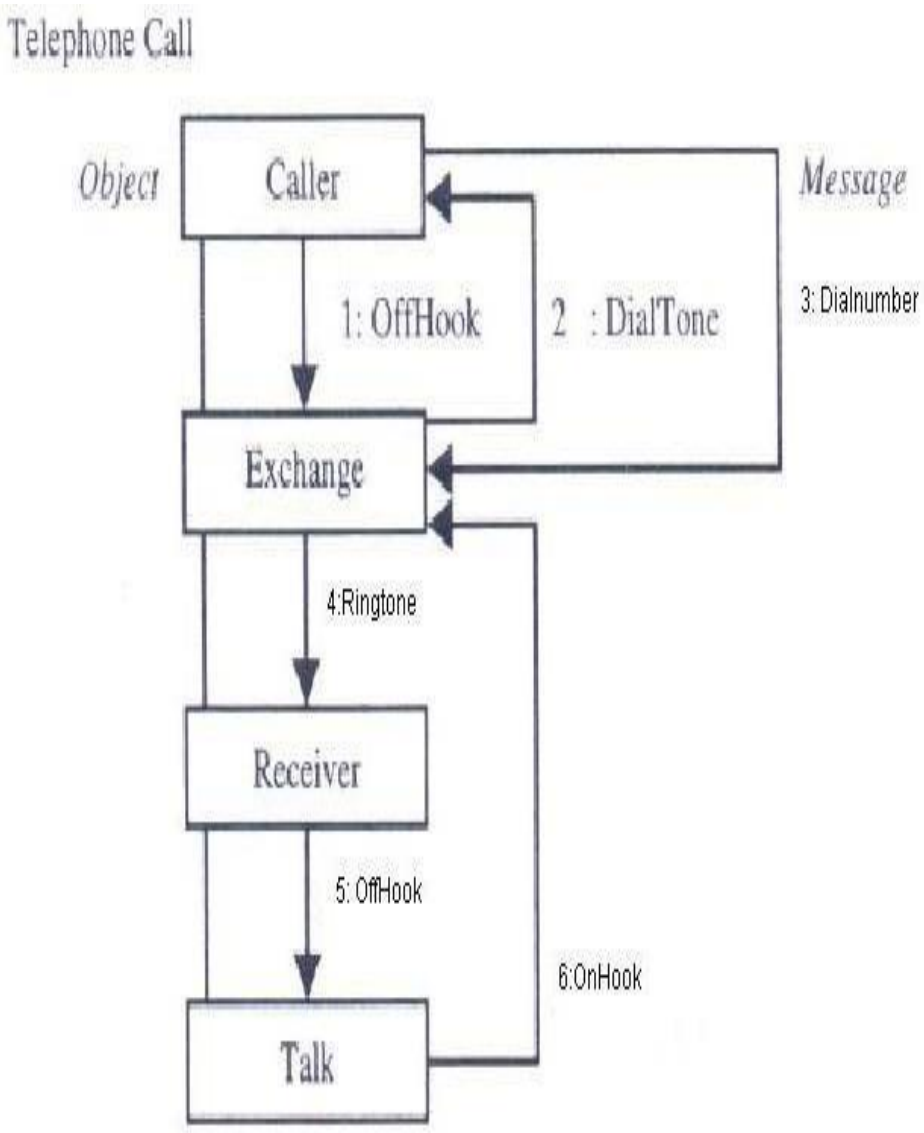


Fig.28. A collaboration diagram with simple numbering.

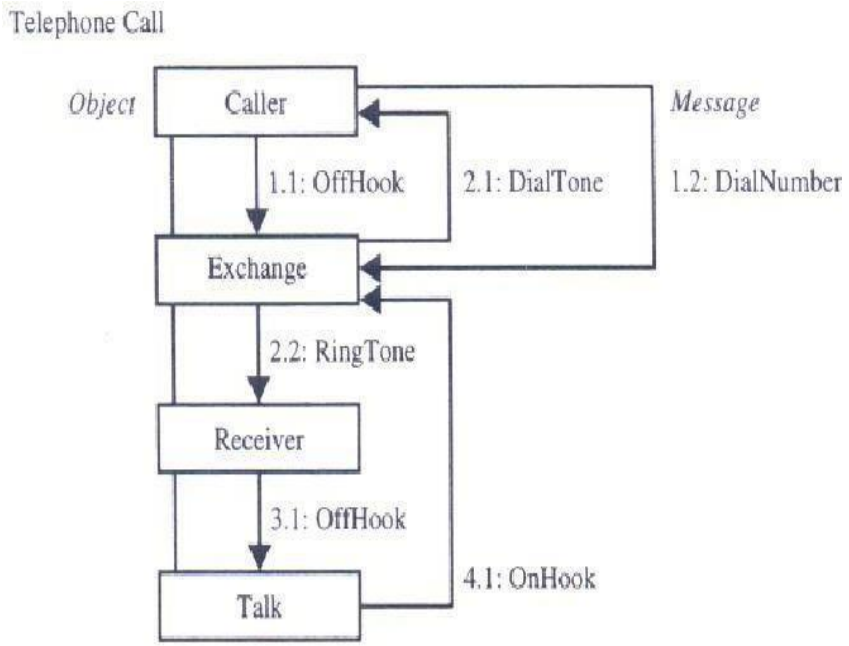


Fig.28 a. A collaboration diagram with decimal numbering.

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 argue that the **main advantage** of interaction diagrams (both collaboration and sequence) is **simplicity**. You easily can seethe 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.

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 thebehavior.

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 pointin 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 activitysymbol 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, amessage 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 suchcompartments:

- 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 willbe 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 statechartsupports nested state machines; to activate a substate machine use the keyword *do:* do *I* machine-name (argument-list). If this state is entered, afterthe 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 triggersa transition on the enclosing state labeled by the implicit activity completion event, usually displayed as an unlabeled transition (see Fig 29).

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 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 5-30).

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

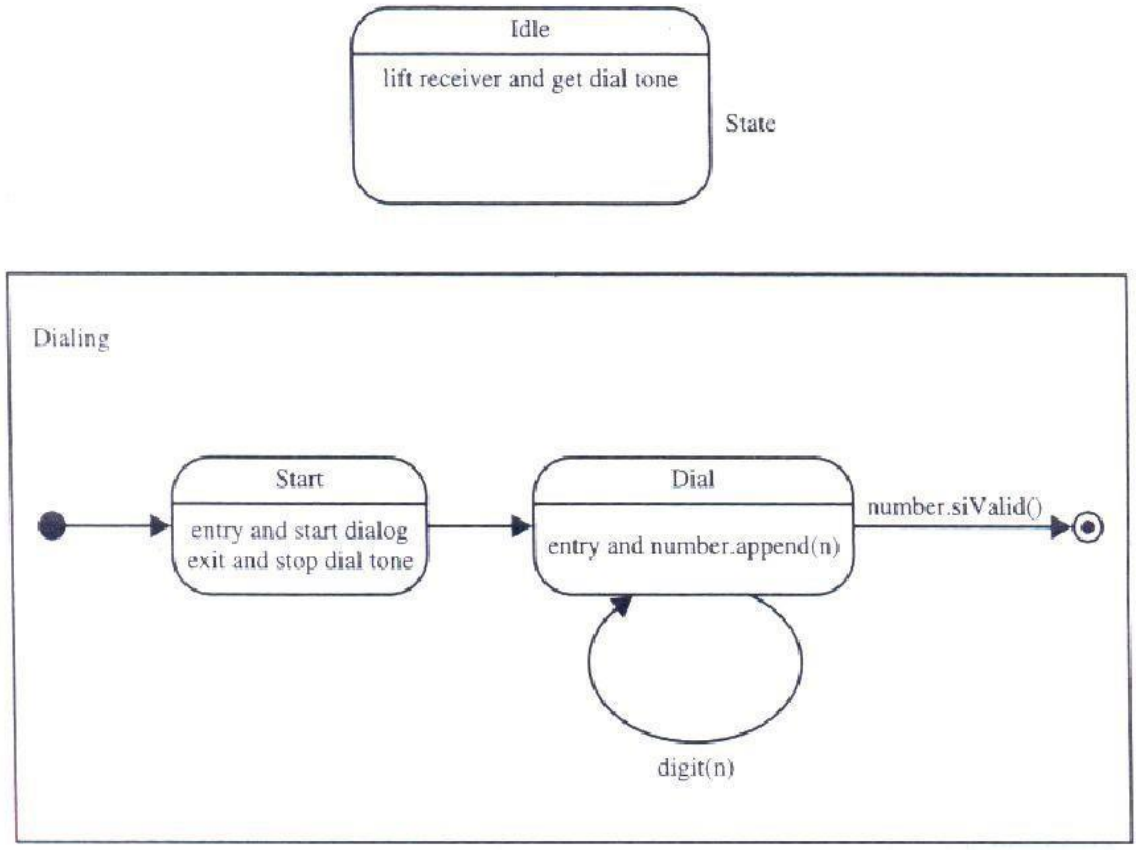


Fig 29: A simple idele and a nested state.

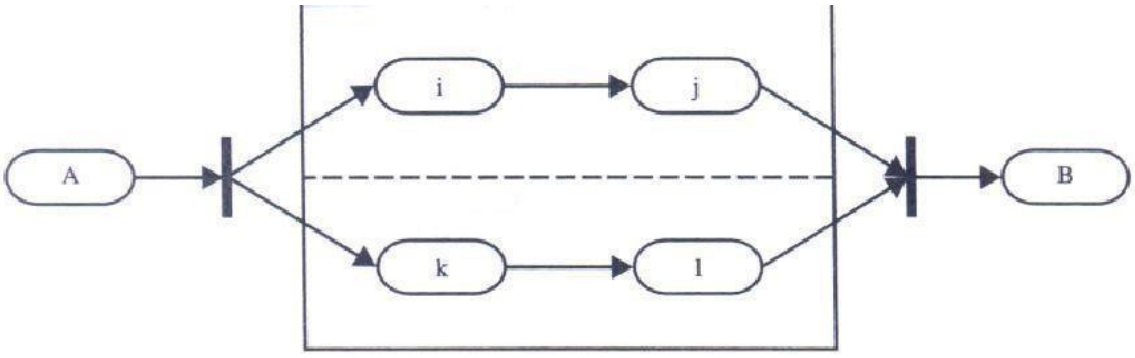


Fig 30: A complex Transition

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. An activity diagram can also be used to represent a class's method implementation.

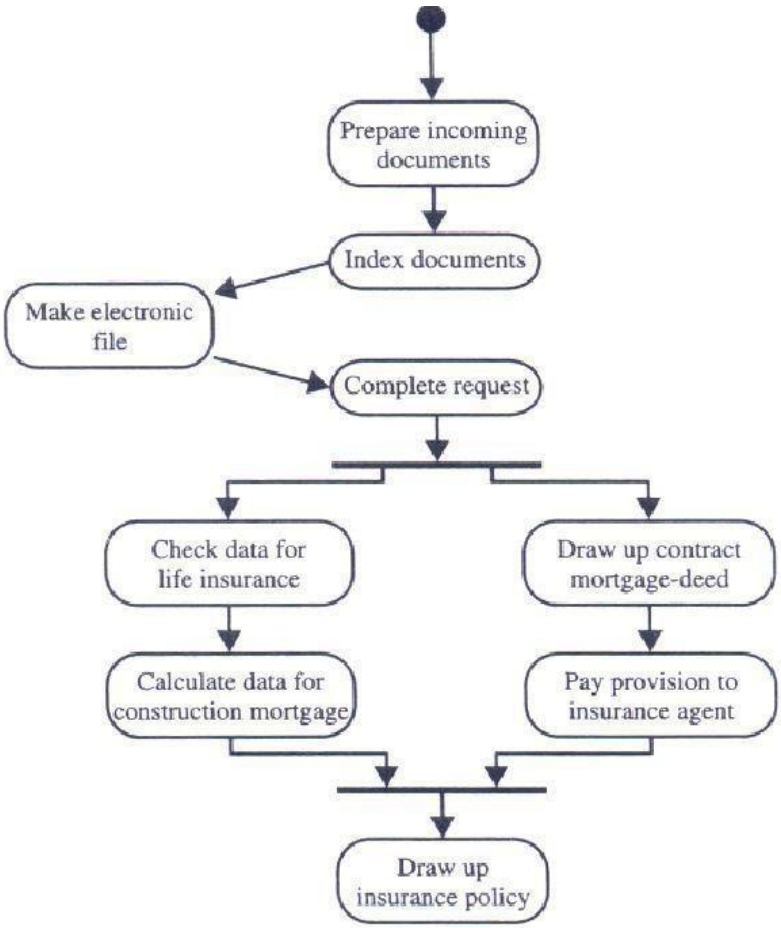


Fig 31. An activity diagram for processing mortgage requests (Loan:Processing Mortgage Request).

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 31).

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.

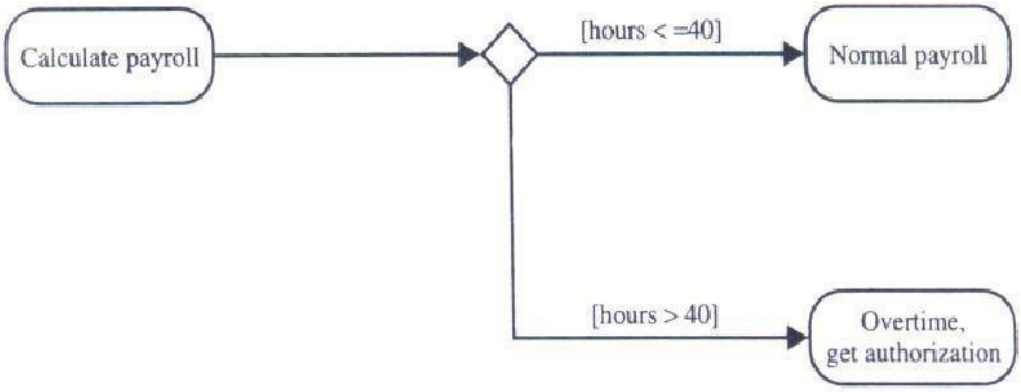


Fig 32: A decision.

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 thanone 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 32 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 32).

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 isassigned to one swimlane. A transition may cross lanes; there is no significance to the routing of the transition path (see Fig 33).

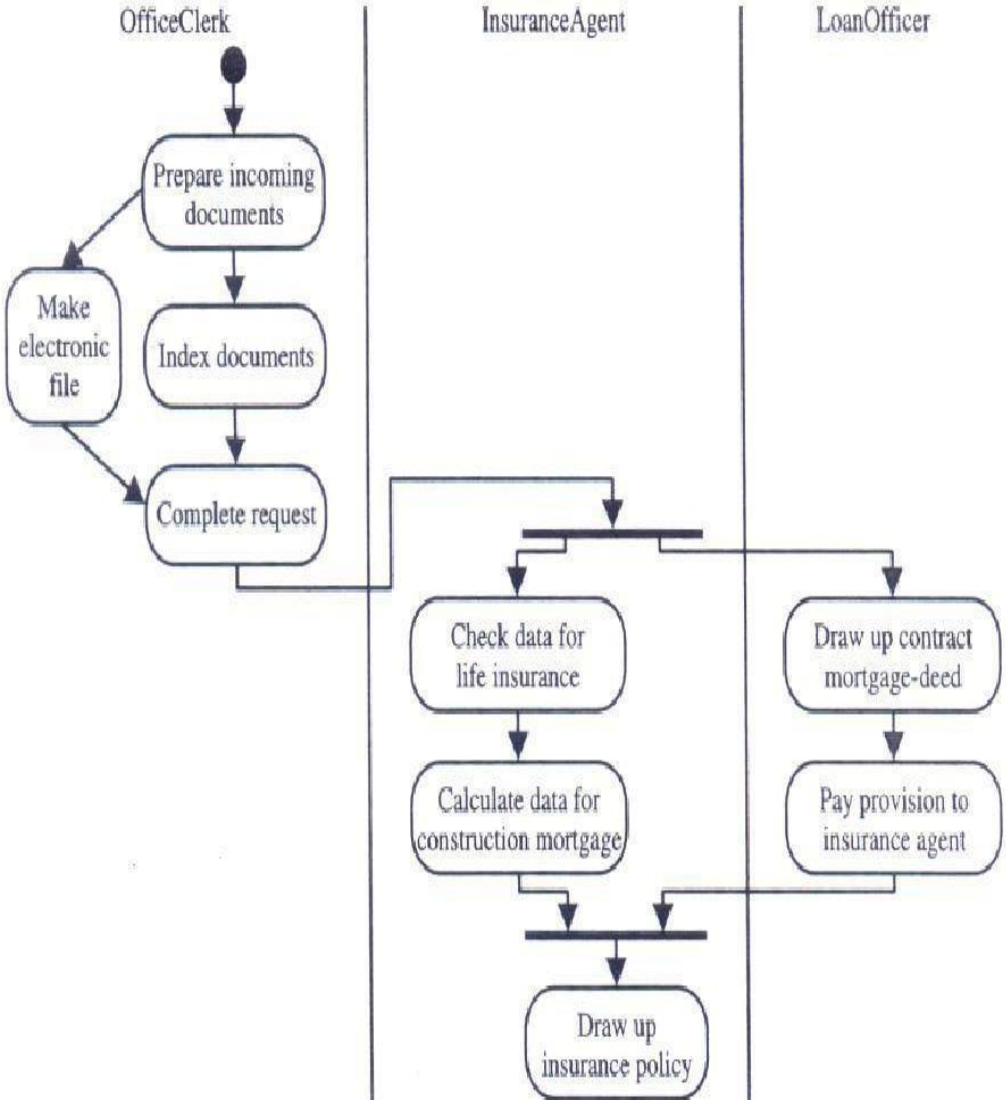


Fig 33. Swimlanes in an activity diagram.

UML 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 2 types of implementation diagrams:

*. Component diagrams – Its Show the stucture of the code itself.

*. Deployment Diagrams – Its show the structure of the runtime system.

These are relatively simple, high-level diagrams compared with other UML diagrams.

Component Diagram *:*

Component diagrams model the physical components (such as source code, executable program, user interface) in a design. These high-level physical components mayor 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 offtheshelf 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. 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 fig shown in Fig 34 Dependency is shown as a dashed arrow.

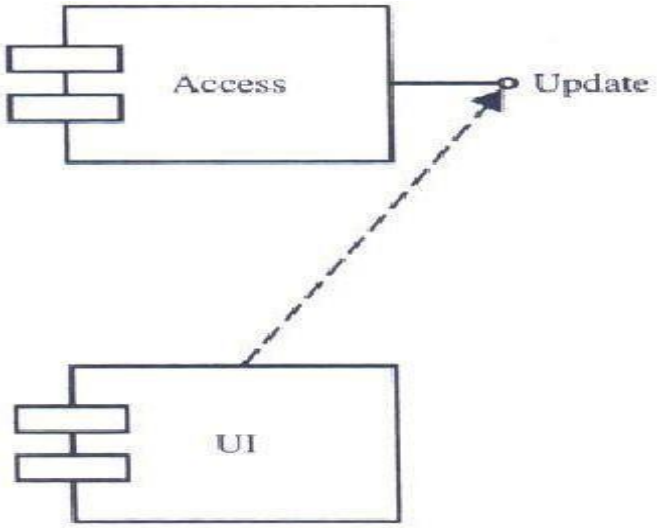


Fig 34: A Component Diagram

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 of code units. In most cases, component diagrams are used in conjunction with deployment diagrams to show how physical modules of code are distributedon 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 mean 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 dashedarrow dependencies, usually throughinterfaces, which indicate one component uses the services of another. Each node or processing element in the system is represented by a three-dimensional box. Connections between the nodes (or platforms) themselves are shown by solid lines (see Fig 35).

The basic UML notation for a deployment diagram.

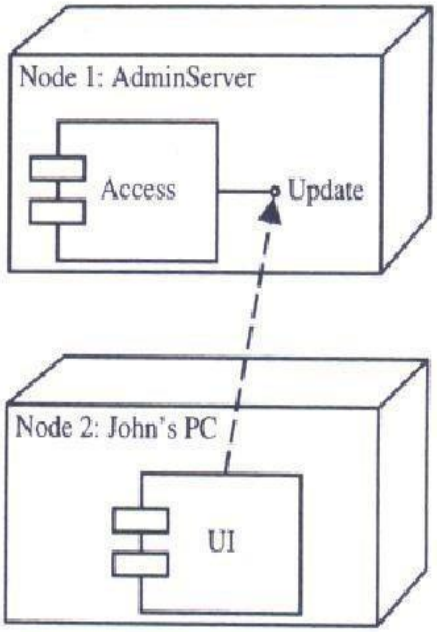


Fig 35. The Basic UML notation for a deployment diagram.

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 36). 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.

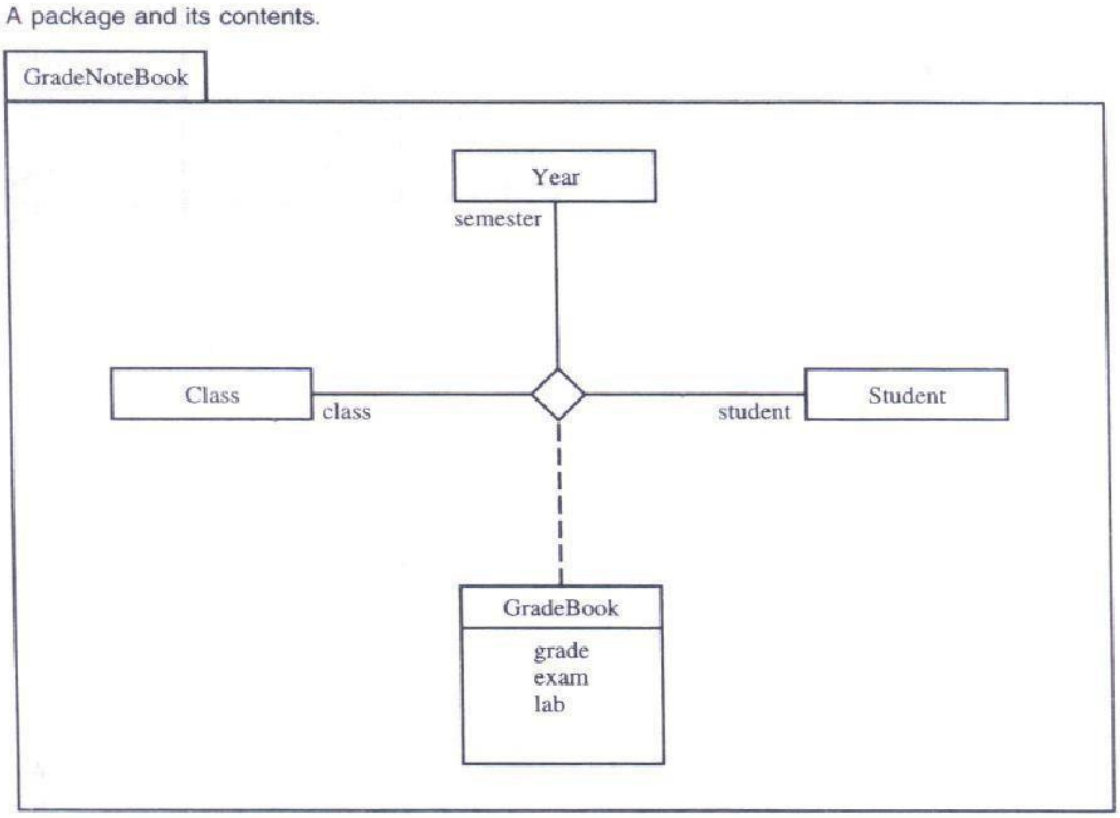


Fig 36: A package and its contents

A real model would have many more classes in each package. The contents might be shown if they are small, or they might be suppressed from higher levels. The entire system is a package. Fig 37 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

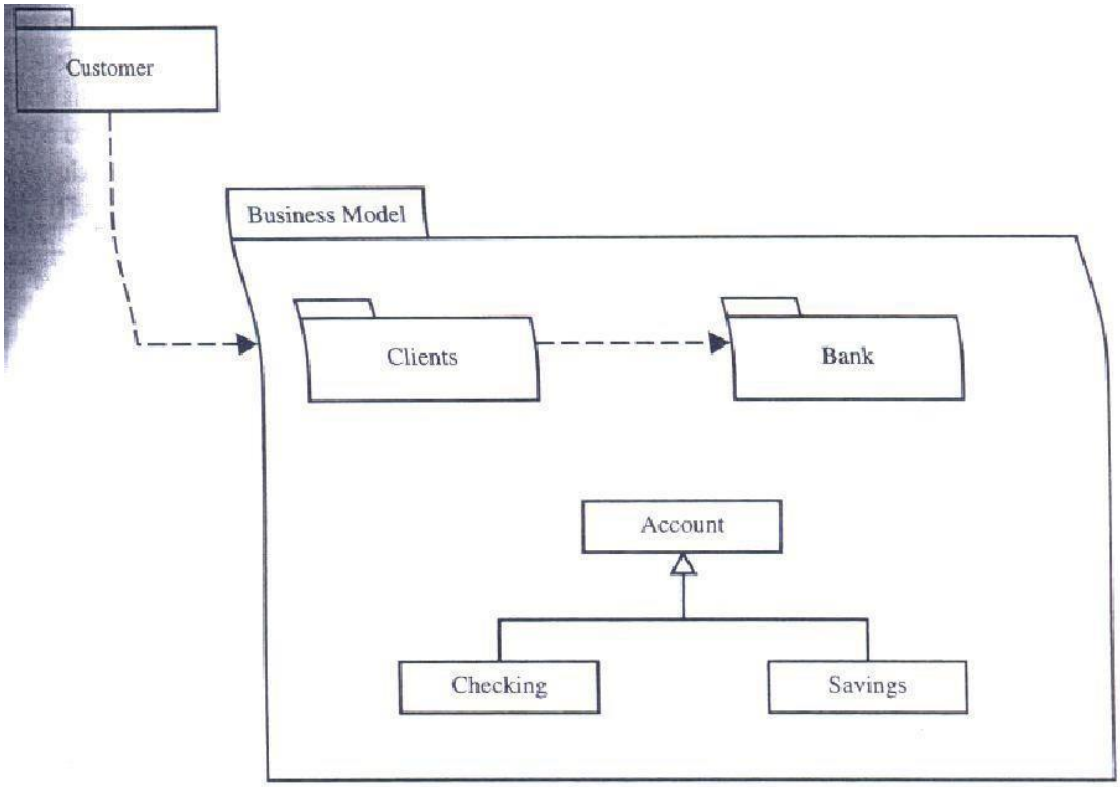


Fig 37: A package and its dependencies

relationship between two or more model elements. It relates the model elements themselves and does not require a setof 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 (seeFig 38).

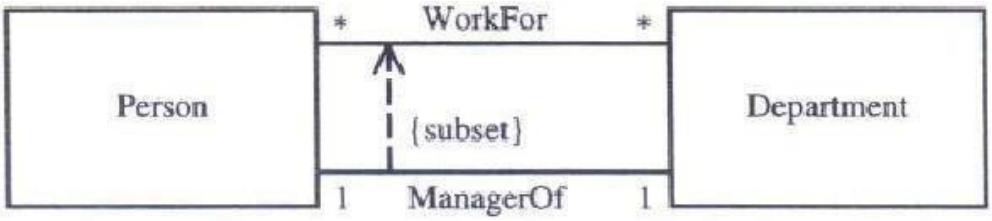


Fig 38: An e ample of constraints. A person is a manager of people whowork for the accounting department.

UML EXTENSIBILITY

1. Model Constraints and comments

Constraints are assumptions or relationship among model elements specifying conditions and propositions that must be maintained as true; otherwise the system described by the model would be invalid. Some constraints, such as association OR constraints are predefined in the UML; others may be defined by users.

Constraints are shown as text in braces (ref. Fig 38). The UML also provides language for writing constraints in the OCL. The constraints may be written in a natural language.

A constraint may be a "Comment", in which case it is written in text. For an element whose notation is a text string such as an attribute, the constraint string may follow the element text string. For a list of elements whose notation is a list of text strings, such as the attributes within class, the constraint string may appear as an element in the list. The constraint applies to all succeeding elements of the list until reaching another constraint string list element or the end of the list. A constraint attached to an individual list element does not supersede the general constraints but may modify individual constraints string may be placed near the symbol name.

The above example fig 38, shows two classes and two associations. The constraint is shown as a dashed arrow from one element to the other, labeled by the constraints string in brace. The direction of the arrow is relevant information within the constraint.

2. Note

A Note is a graphic symbol containing textual information; it also could contain embedded images. It is attached to the diagram rather than to a model element. A note is shown as a rectangle with "Bent Corner" in the upper right corner. It can contains any length text. (ref. Fig 39).

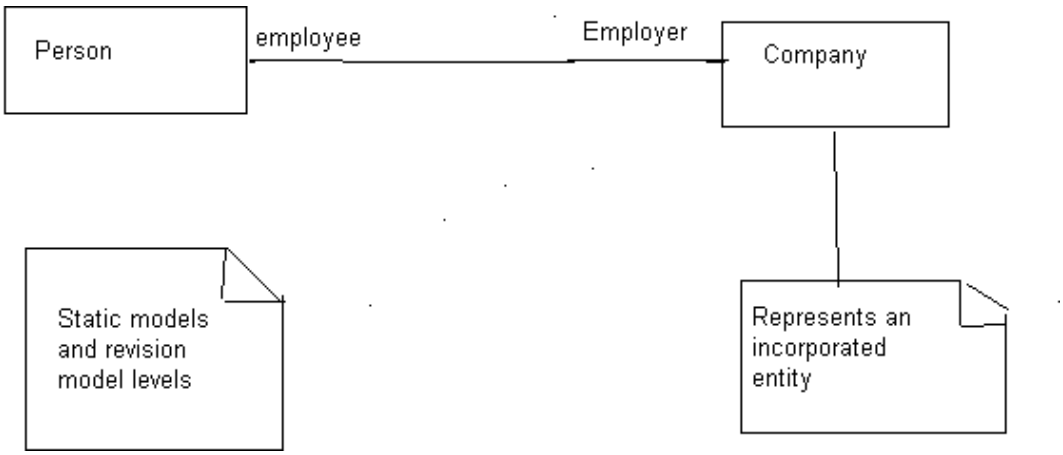


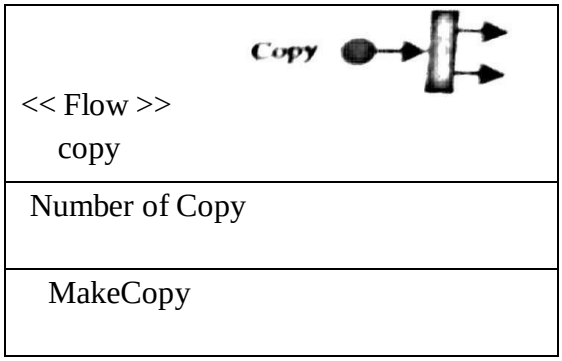
Fig 39. Note

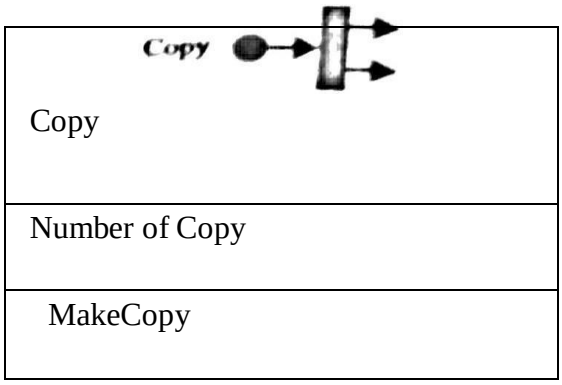
3. Stereotype

Stereotype represent a built-in extensibility mechanism of the UML. User-defined extensions of the UML are enabled through the use of stereotypes and constraints. A stereotype is a new class of modeling element introduced during modeling, It represents a subclass of an existing modeling element with the same form (attributed and relationships) but a different intent. UML stereotype extend and tailor the UML for a specific domain or process.

The general presentation of a stereotype is to use a figure for the base element but place a keyword string above the name of the element (if used, the keyword string is the name of a stereotype within matched guillemets, "<<",">>", such as <<flow>.. Note that a guillemet looks like a angle- bracket, but it is a single character in most fonts.

<< Flow >> copy
Number of Copy
MakeCopy





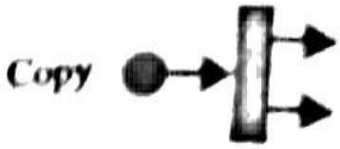


Fig 40: Various forms of Stereotype notation

The stereotype allows extension of UML notation as well as a graphic figure, texture, and color. The figure can be used in one of two ways: (1) instead of or in addition to the stereotype keyword string as part of the symbol for the base model element or (2) as the entire base model element. Other information contained by the base model element symbol is suppressed.

3. UML Meta-model

The UML defined notations as well as a meta model. UML graphic notations can be used not only to describe the system's components but also to describe a model itself. This is known as a **meta-model.**

In other words, **a meta-model** is a model of modeling elements. The purpose of the UML meta-model is to provide a single,common, and definitive statement of the syntax and semantics of the elements of the UML.

The meta model provides us a means to commect different UML diagrams. The connection between the different diagrams is very important, and the UML attempts to make these coupling more explict through definingthe underlying model while imposing no methodology.

The presence of this meta model has made it possible for its developers to agree on semantics and how those semantics would be best rendered. This is an important step forword, since it can assure consistency among diagrams. The meta-model aslomcan serve as a means to exchange data between different cas tools. The fig 41 is an example fo the UML meta- model that describes relationship with association and generalization; association is depicted as a composition of association roles.

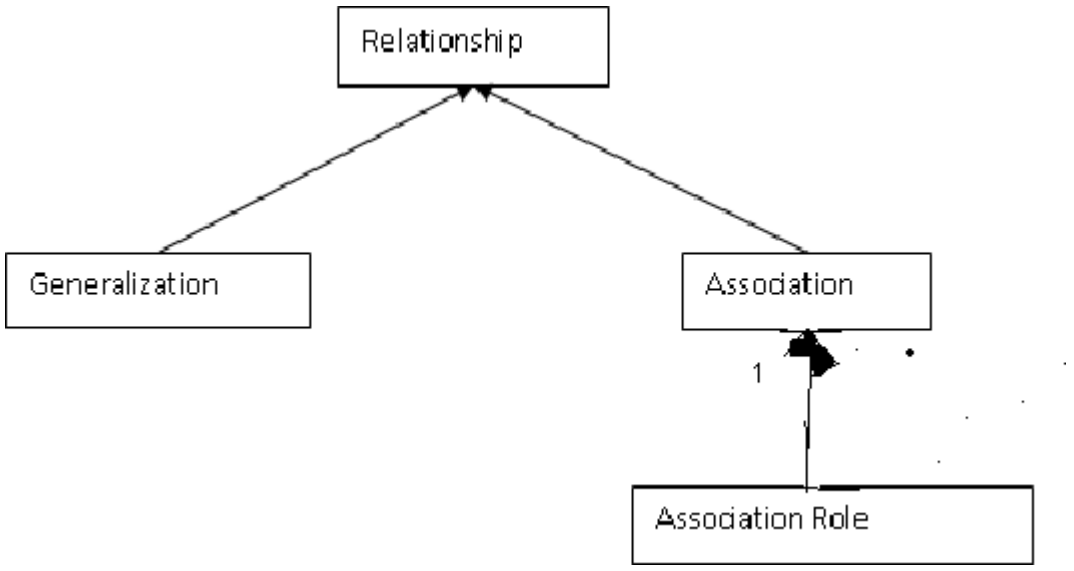


Fig 41. The UML meta-model describing the relationship between association and generalization.

Questions

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	What is antipattern?	Remember	BTL 1
2.	Define framework.	Remember	BTL 1
3.	Distinguish Static and Dynamic Models	Understand	BTL 2
4.	Compare patterns and frameworks.	Analysis	BTL 4
5.	Define Design Pattern	Remember	BTL 1
6.	Define UML.	Remember	BTL 1
7.	List out the various UML diagrams	Remember	BTL 1
8.	What are the advantages and disadvantages of interaction diagrams?	Remember	BTL 1
9.	What is UML class diagram?	Remember	BTL 1
10.	Classify the kinds of actors in use case.	Analysis	BTL 4
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Discuss in detail about Rumbaugh method.	Understand	BTL 2
2.	Design the Use case, Sequence and Activity diagram for an ATM application.	Create	BTL 6
3.	Discuss briefly about the usecase diagram with example.	Understand	BTL 2
4.	Explain in detail about Jacobson methodology?	Remember	BTL 1
5.	Design the Class diagram for Library Management System. Specify the attributes, methods and relationship among classes.	Create	BTL 6
6.	Describe patterns and the various pattern templates?	Remember	BTL 1
7.	Discuss in detail about BOOCH methodology	Understand	BTL 2

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – III - OBJECT ORIENTED ANALYSIS AND SYSTEM ENGINEERING - SCSA1401

UNIT 3

OBJECT ORIENTED ANALYSIS.

Business Object Analysis - Use Case Driven Object Oriented Analysis - Business Process Modeling - Use Case model - Developing Effective Documentation - Object Analysis Classification: Classification Theory - Noun Phrase Approach - Common Class Patterns Approach - Use-Case Driven Approach - Classes Responsibilities and Collaborators – Naming Classes - Identifying Object Relationships, Attributes and Methods: Association – SuperSubclass Relationships - A-part of Relationships.

OBJECT ORIENTED ANALYSIS: USE-CASE DRIVEN

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 tile 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: 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 which 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.

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 - 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

2. 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.

3. Unnecessary features - 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.

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.

BUSINESS OB.JECT 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 playa role in achieving system goals.

To understand the users' requirements, we need to find out how they "use" the system. This can be accomplish 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.

USECASE 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, whowill 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.

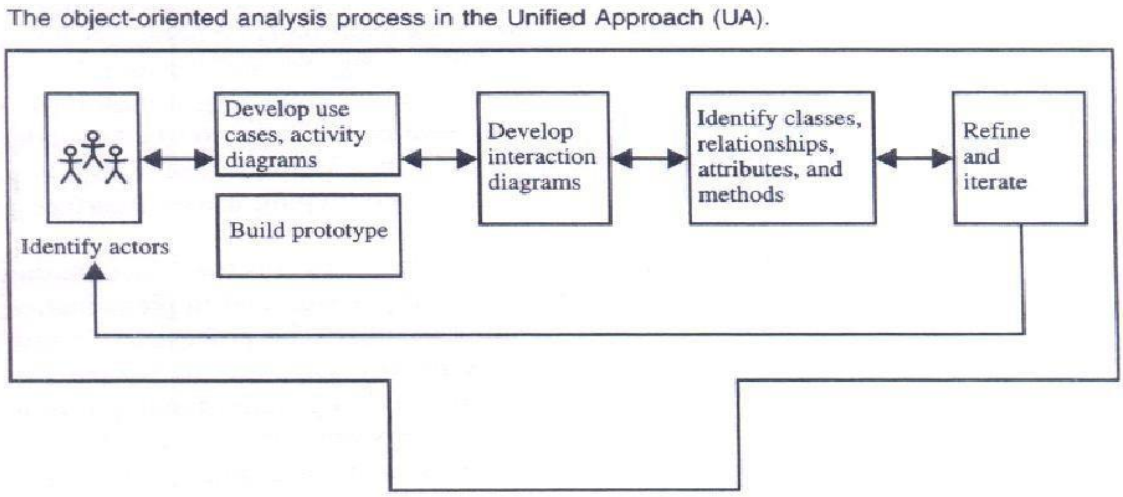


Fig 3.1 : Object Oriented analysis process in the Unified Approach(UA)

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. (see Fig- 3.2)

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.

FIGURE

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

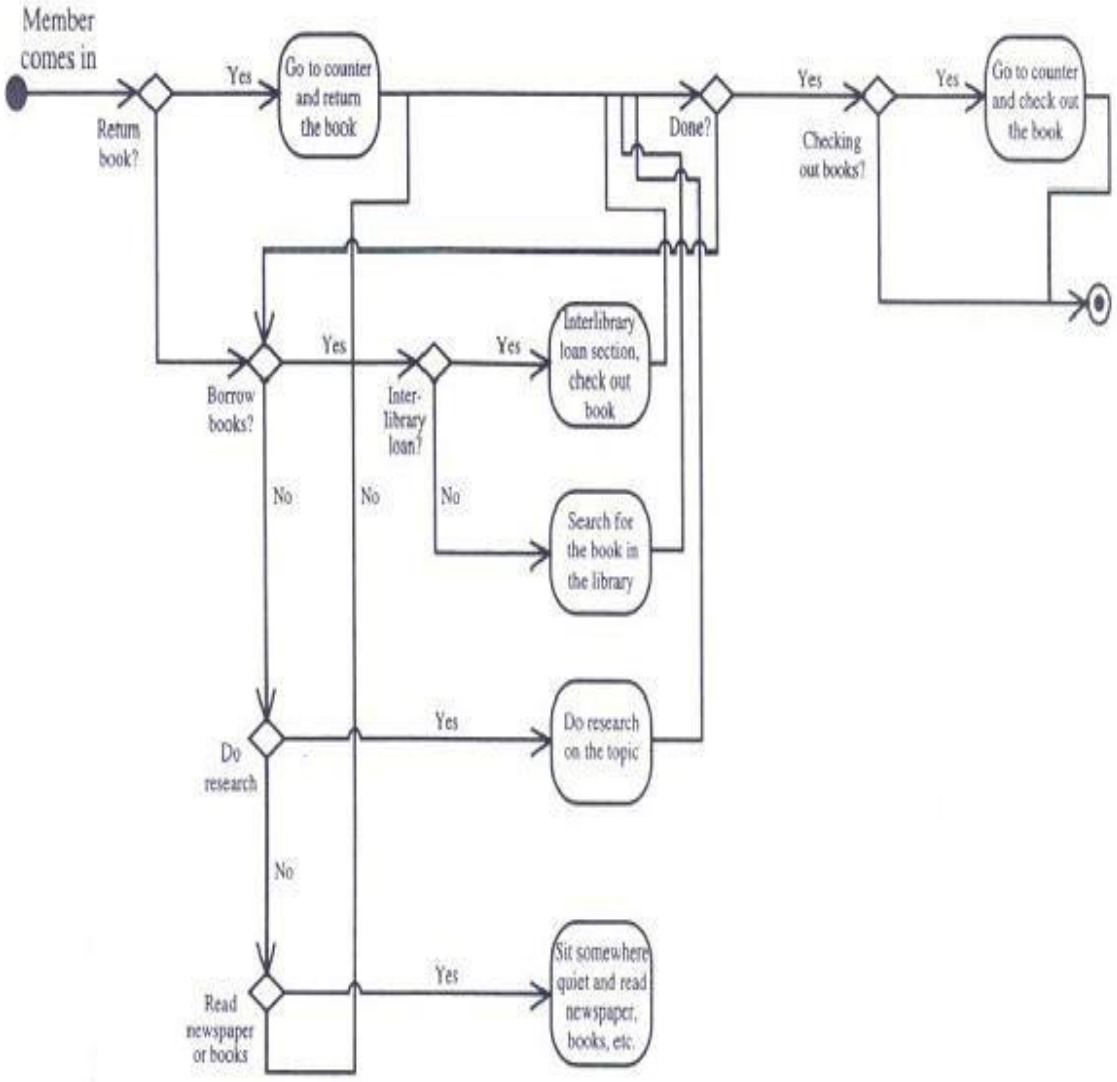


Fig 3.2 : This activity diagram shows some activities that can be performed by alibrary member.

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 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.

*A use-case model also can discover classes and the relationships among subsystems of the systems.

*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 3.3).

*The use-case model expresses what the business or application will do and not how; that is the responsibility of the UML class diagram

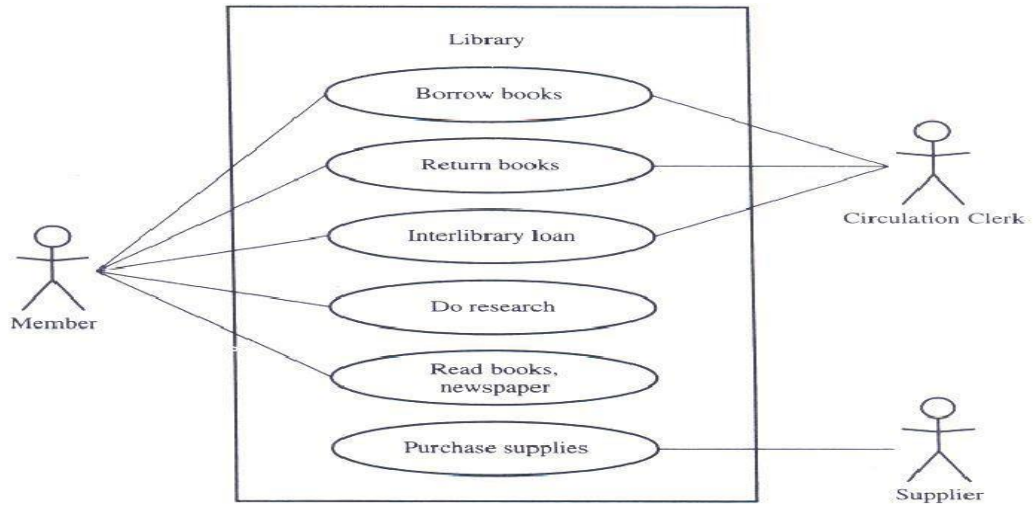


Fig 3.3 Use Case Diagram – Library System

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.".

Guidelines for developing Use Case Models:

1. **Use Cases under the Microscope**
2. **Uses and Extends Associations**
3. **Identifying the Actors**
4. **Guidelines for Finding Use Cases**
5. **How Detailed Must a Use Case Be? When to Stop Decomposing and When to Continue**
6. **Dividing Use Cases into Packages**
7. **Naming a Use Case**

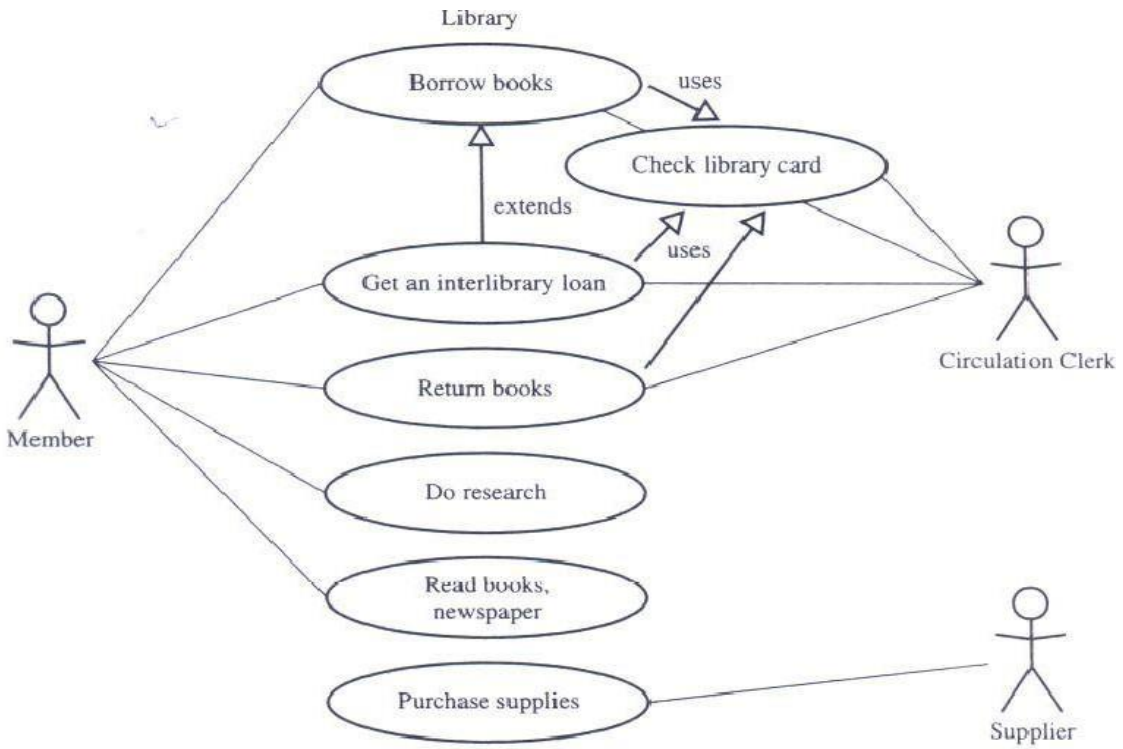
1. Use Cases under the Microscope:

Use cases represent the things that the user is doing with the system, which can be different from the users' goals.

Definition of use case by Jacobson "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 case** - 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.
***Actors** - An 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.
***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.
***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 3.4).



Three actors appear in Figure 3.4: a member, a circulation clerk, and a supplier.

1. **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.
2. **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.
3. **Use-case name: Return books**. A member brings borrowed books back to the library. .
4. **Use-case name: Check library card.** A member submits his or her library card to the clerk, who checks the borrower's record.
5. **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.
6. **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.
7. **Use-case name: Purchase supplies**. The supplier provides the books, journals, and newspapers purchased by the library.

2. 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 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.

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 are 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 3.4).

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.

Fowler and Scott provide us excellent(guidelines for addressing variations in usecase 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.

3. 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 or job 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.(Fig 3.5)

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:

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 usecase 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 .

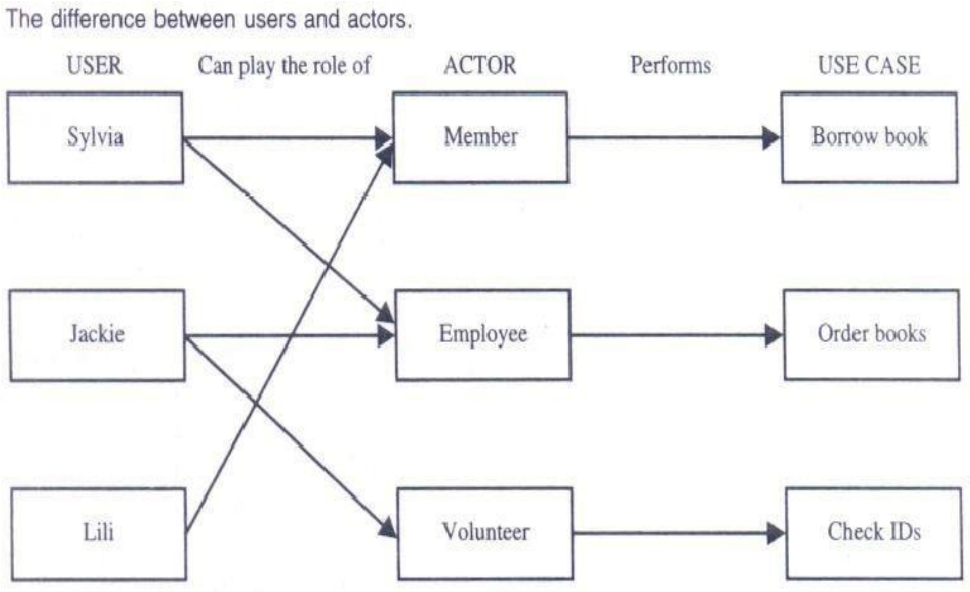


Fig 3.5 : The difference between users and actors

4. 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
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).

5. 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. 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.

The UML specification recommends that at least one scenario be prepared for each significantly different kind of use case instance

6. Dividing Use Cases into 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.

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 3.6).

7. 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.

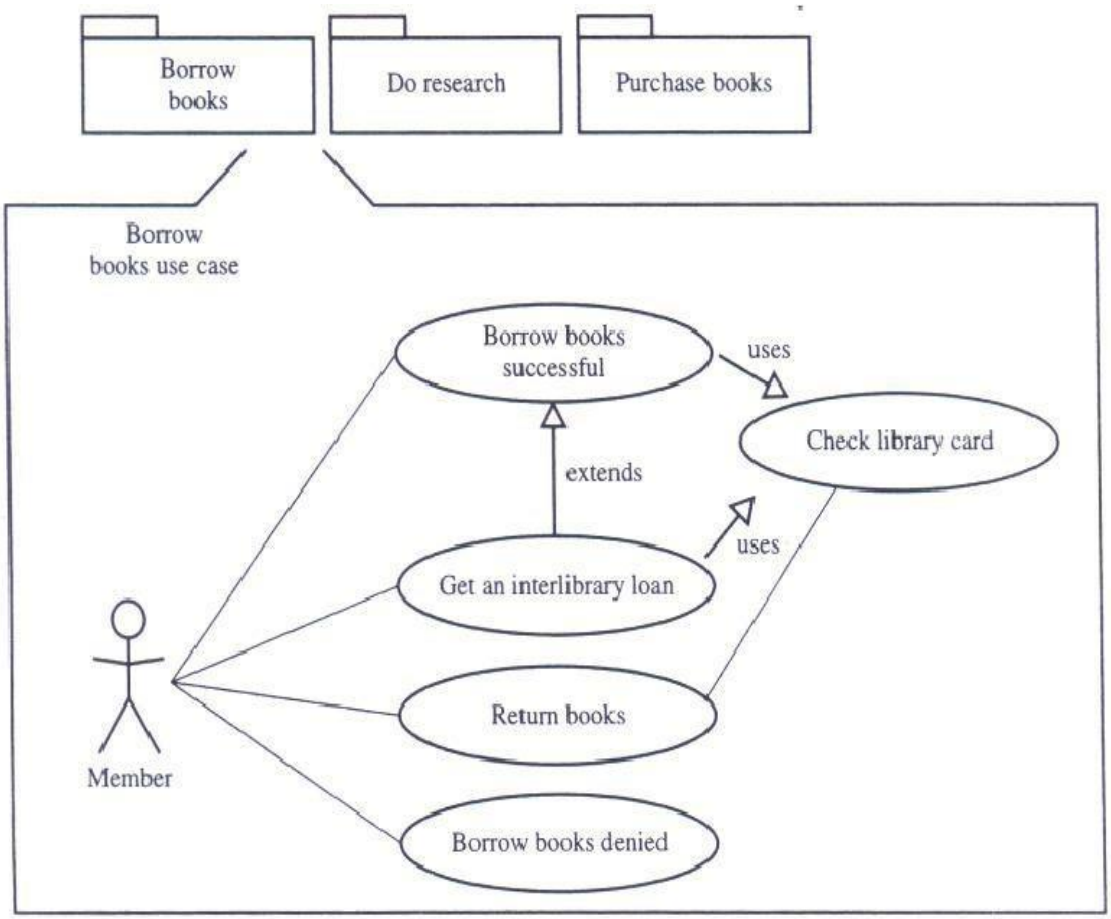


Fig 3.6 A libray system can be divided into many packages, each of whichencompasses mulitple use cases.

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?

1. 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 problem solving 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 Remember that your modeling effort becomes the analysis, design, and testing documentation. However this template which is based on the unified approach life cycle assists you in organizing and composing your models into an effective documentation.

2. Guidelines for Developing Effective Documentation

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

1.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 .

2.80-20 rule. As for many applications, 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.

3.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.

4.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 summaries, 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 .

5.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.

(Document Name)
For
(Product)
(Version no)

Responsible individual
Name :
Title :

Fig : 3.7 Cover Sheet template.

Case Study : ANALYSING THE VIANET BANK ATM – THE USE CASE DRIVEN PROCESS

The Following section provides the description of the vianet bank atm system's requirement.

*The Bank client must be able to deposit an amount to and withdraw an amount from his or her accounts using the touch screen at the vianet bank atm. Each transaction must be recorded, and the client must be able to review all transactions performed against the given account. Recorded transactions must include the date, time, Transaction type, amount and account balance after the transactions.
*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 rovided 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.

i) 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. .

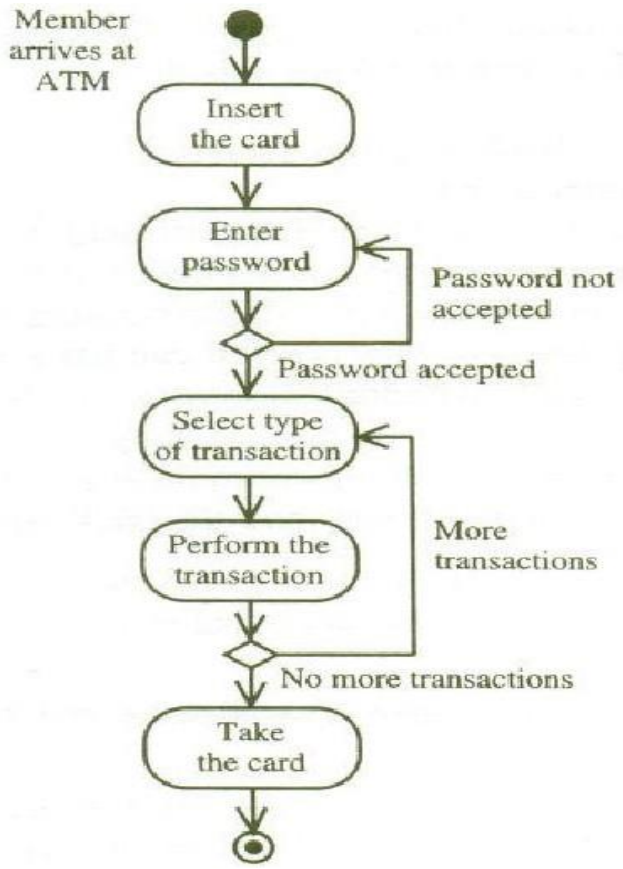


Fig 3.8 : Activities involved in an ATM transction

*. Use-case name: Approval process. The client enters a PIN code that consists of 4digits. 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.

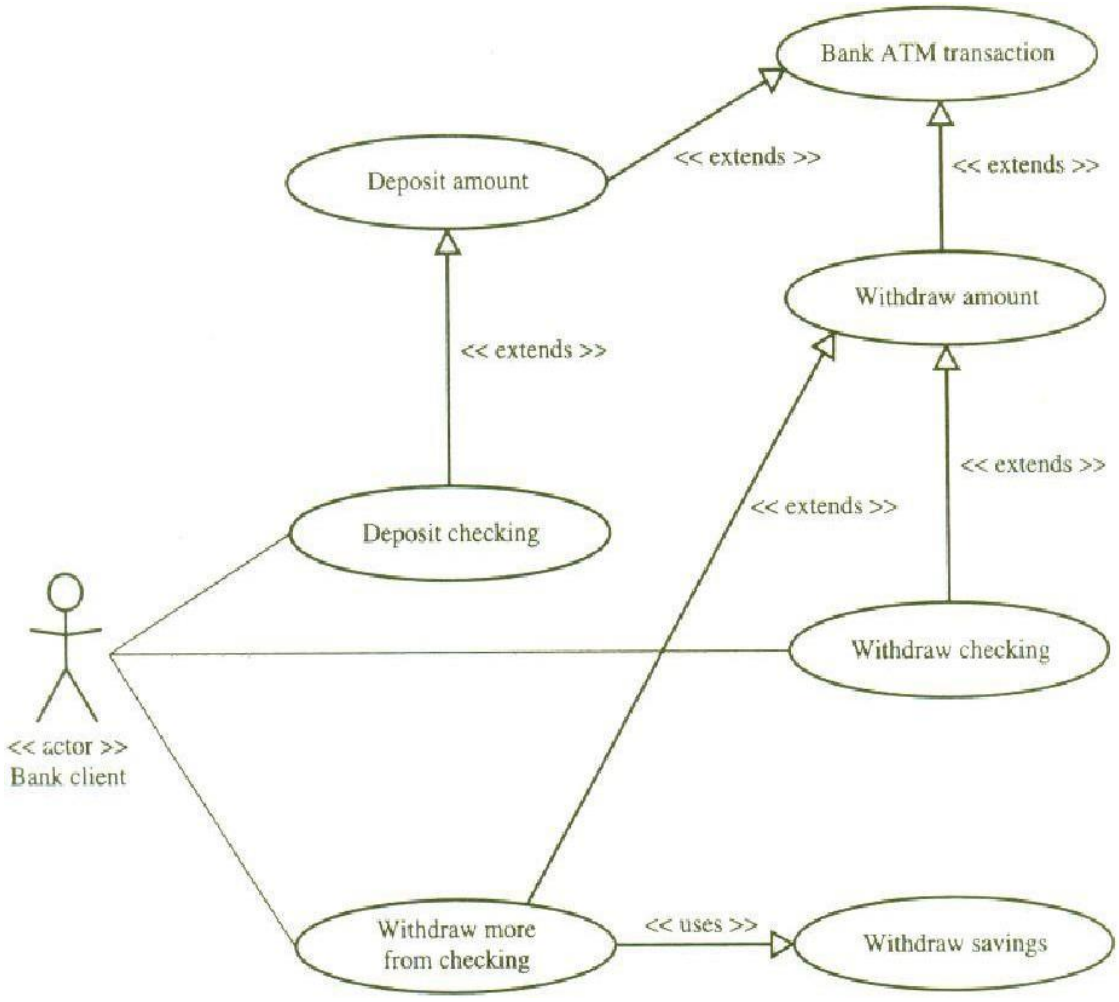


Fig : 3.9 Transaction use cases

* 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.)

*Use-case name: Withdraw 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.

*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 ishalted and a message is displayed. The savings account use-cases package. This use case extends the bank transaction use case. (See Figure 3.10))
*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.

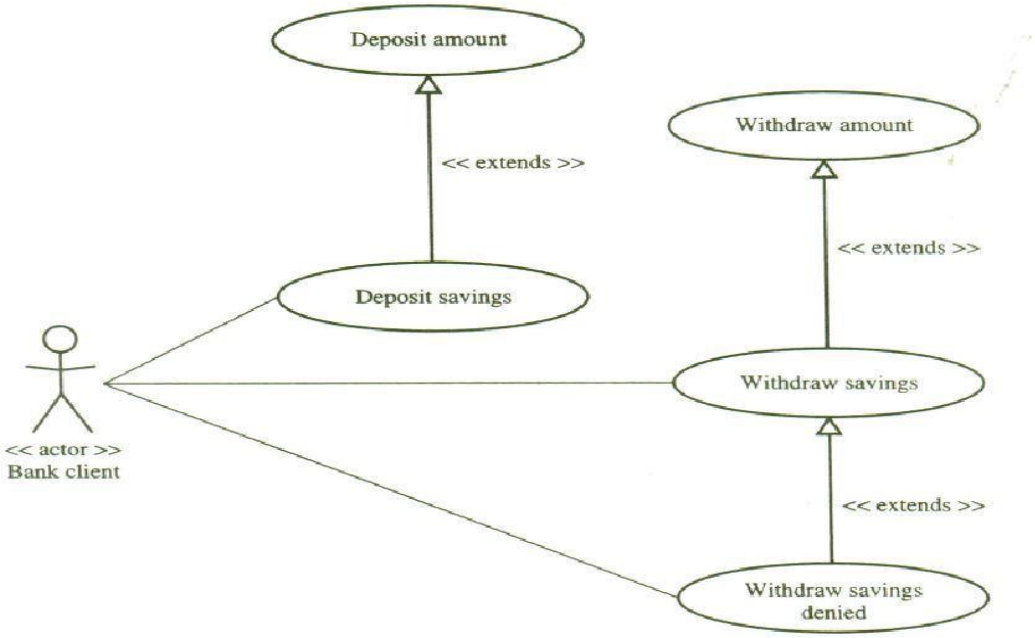


Fig 3.10 The checking account use-cases

OBJECT ANALYSIS :

CLASSIFICATION

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 airly 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.

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.

APPROACHES FOR IDENTIFYING CLASSES

In the following sections, we look at four alternative approaches for identifying classes:

1. The **Noun Phrase** approach;
2. The **Common Class Patterns** approach;
3. The **Usecase Driven, Sequence/Collaboration Modeling** approach;
4. 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.

3.10. 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 3.11): **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.

Fig 3.11: Using the noun phrase strategy, candidate classes can be divided into 3 categories.

i) 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.

ii) 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**. 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.

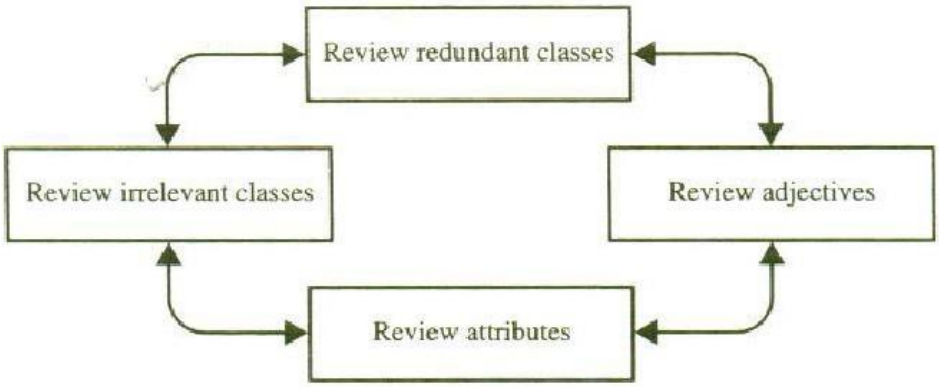


Fig 3.12: The process of eliminating the redundant classes and refining the remaining classes is not sequential. You can move back and forth among these steps as often as you like.

Example: 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

~~Step~~
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 **meaningfull 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
~~Card~~
Cash
Check
~~Checking~~
Checking Account
~~Client~~
~~Client's account~~
Currency
Dollar
~~Envelope~~
~~Fund digits~~
Fund
Invalid PIN
Message
Message
~~Money~~
Password
PIN

~~PIN Code~~
Record
~~Savings~~
Savings Account
~~Step~~
System
Transaction
Transaction History

Reviewing the Classes Containing Adjectives

We again review the remaining list, now with an eye on classes with adjectives. The main 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.

Amount : a value, not a class.
Account 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
~~Account Balance~~
~~Amount~~
Approval Process
ATM Card
Bank
BankClient
Cash
~~Card~~
Check
~~Checking~~
Checking Account
Currency
Dollar

~~Envelope~~
~~Fund digits~~
Fund
Message
MaBey
PIN
~~PIN Code~~
Record
Savings Account
System
~~step~~
Transaction
~~Transaction History~~

Reviewing the Class Purpose

Identifying the classes that playa 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:

ATM 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.

COMMON CLASS PATTERN APPROACH

The second method for identifying classes is using *common class patterns,* which is **based on a knowledge base of the common classes that have been proposed by various researchers, such as Shlaer and Mellor [10], Ross[8], and Coad and Yourdon [3].** They have compiled and listed the following patterns for finding the candidate class and object :

- Name. **Concept class**

Context: A concept is a particular idea or understanding that we have of our world. The concept class encompasses principles that are not tangible but used to organize or keep track of business activities or communications. Marin and Odell describe concepts elegantly, "Privately held ideas or notions are called **conceptions**. When an understanding is shared by another, it becomes a **concept.** To **communicate** with others, we must share our individually held conceptions and arrive at agreed concepts." Furthermore, Martin and Odell explain that, without concepts, mental life would be total chaos since every item we encountered would be different.
Example. Performance is an example of concept class object.

- Name. **Events class**

Context: Events classes are **points in time that must be recorded**. Things happen, usually to something else at a given date and time or as a step in an ordered sequence. Associated with things remembered are attributes (after all, the things to remember are objects) such as who, what, when, where, how, or why.
Example. Landing, interrupt, request, and order are possible events.

- Name. ***Organization class***

Context: . An organization class is a **collection of people, resources, facilities, or groups to which the users belong; their capabilities** have a defined mission, whose existence is largely independent of the individuals.
Example. An accounting department might be considered a potential class.

- Name. ***People class*** (also known as person, roles, and roles played class)

Context. The people class **represents the different roles users play in interacting with the application**. People carry out some function. What roles does a person play in the system? Coad and Yourdon [3] explain that a class which is represented by a person can be divided into two types: those representing users of the system, such as an operator or clerk who interacts with the system; and those representing people who do not use the system but about whom information is kept by the system.
Example. Employee, client, teacher, and manager are examples of people.

- Name. ***Places class***

Context. Places are **physical locations** that the system must keep information about.
Example. Buildings, stores, sites, and offices are examples of places.

- Name. ***Tangible things and devices class***

Context. This class **includes physical objects or groups of objects** that are tangible and devices with which the application interacts.

Example. Cars are an example of tangible things, and pressure sensors are an example of devices.

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

The 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.

i) 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 usecase scenario.

CASE STUDY : 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 3.13). The client in this case is whoever tries to access an account through the ATM, and mayor may not have an account. The BankClient on the other hand has an account.

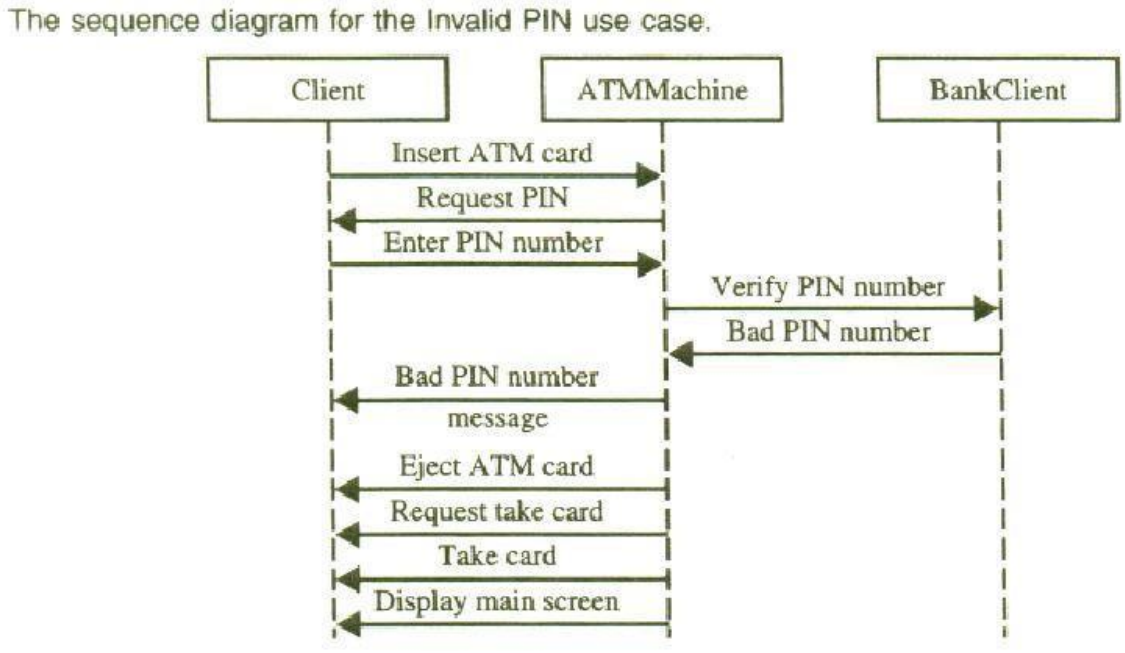


Fig.3.13: The sequence diagram for the INVALID PIN use case

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 ha"\'e'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 3.14 and 3.16). 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 3.15)

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.

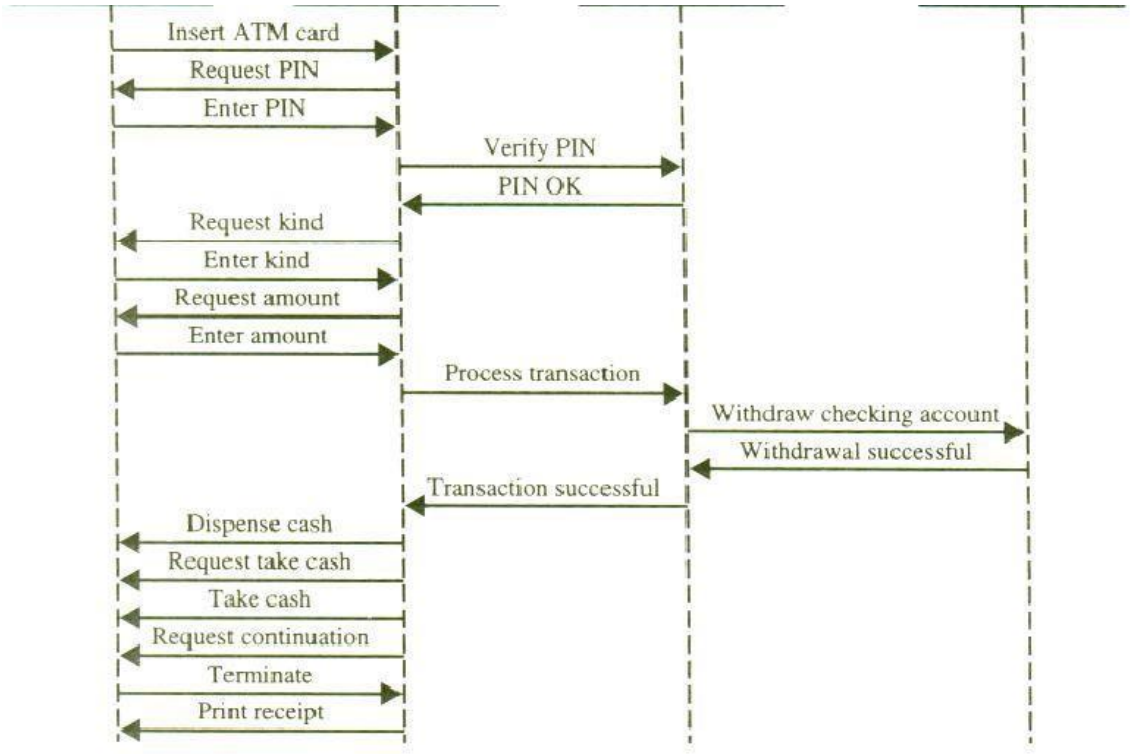


Fig 3.14 : Sequence Diagram for the Withdraw Checking use case

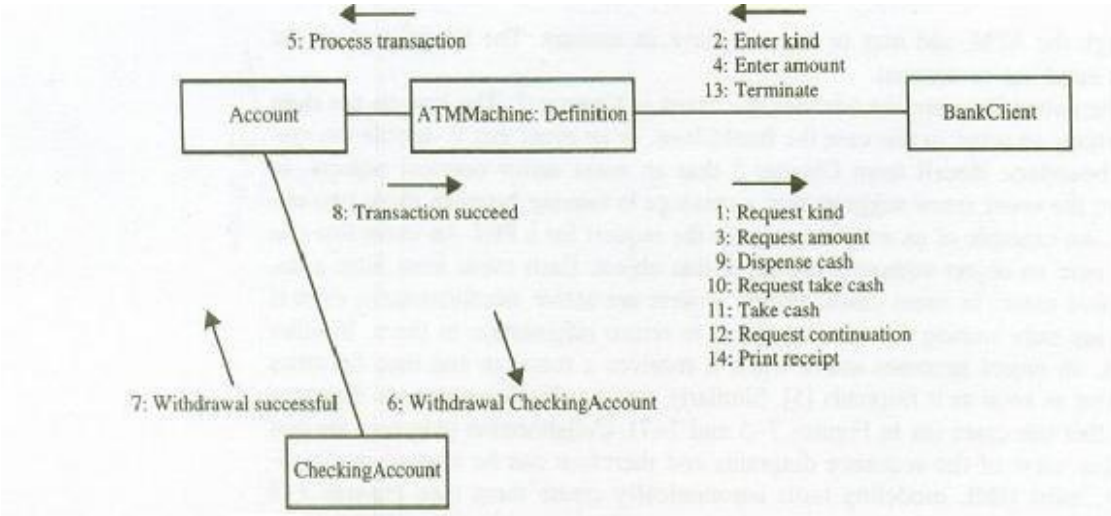
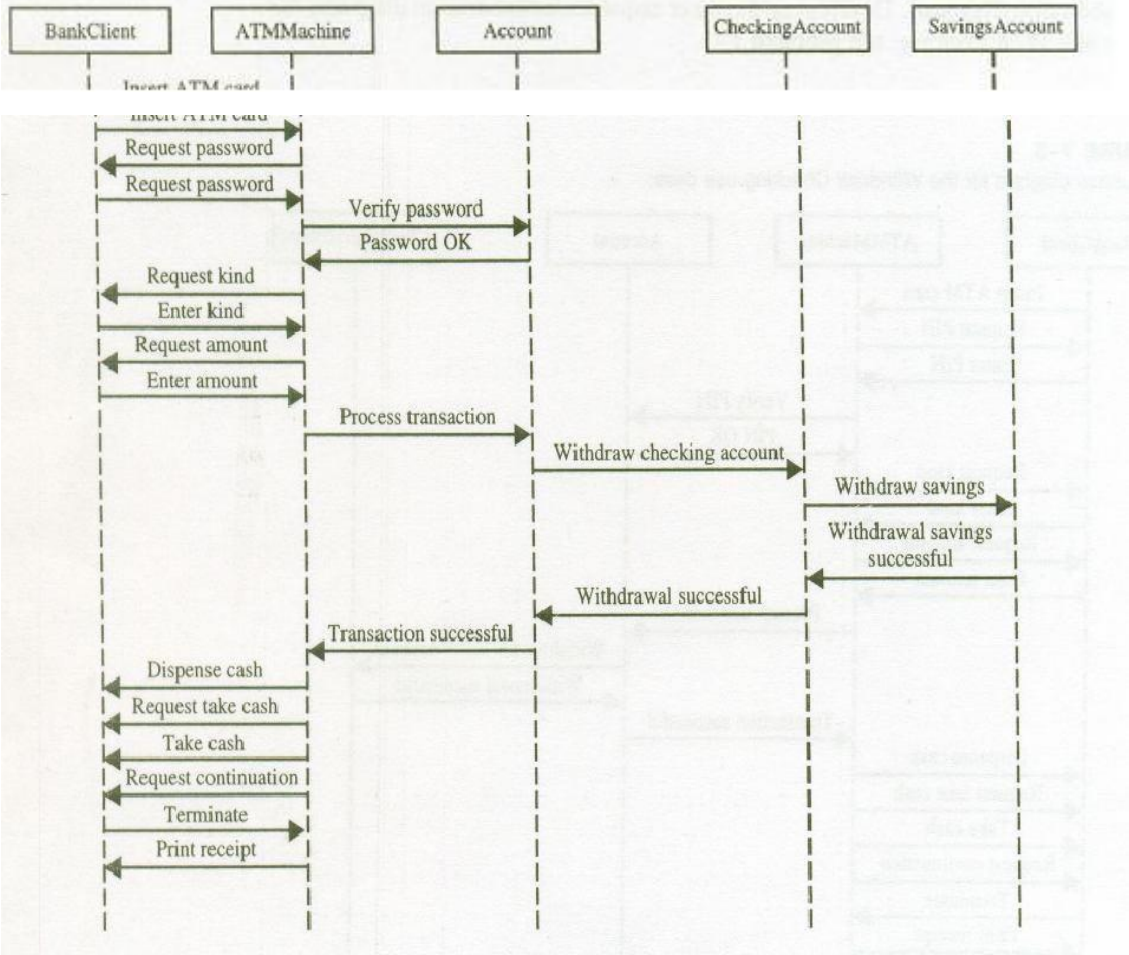


Fig 3.15 : The Collaboration diagram for the Withdraw Checking use case

Fig 3.16
The sequence diagram for the Withdraw More from Checking use case.



CLASSES, RESPONSIBILITIES, AND COLLABORATORS (CRC) APPROACH

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. 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 3.17 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.

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.

.

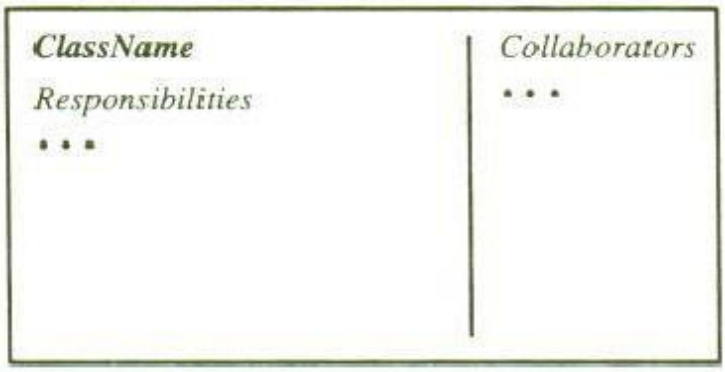


Fig 3.16 A Classess, Responsibilities and Collaborators (CRC) Index Card

CRC starts with only one or two obvious cards. If the situation calls for a responsibility not already covered by one of the objects: Add, or create a new object to address that responsibility.

– Finding classes is not easy.

– The more practice you have, the better you get at identifying classes.

– There is no such thing as the ―right set of classes.‖

– Finding classes is an incremental and iterative process.

i) Classes, Responsibilities, And Collaborators Process

The Classes, Responsibilities, and Collaborators process consists of three steps (Figure 3.17)
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, Resposibilities, and Collaborators process.

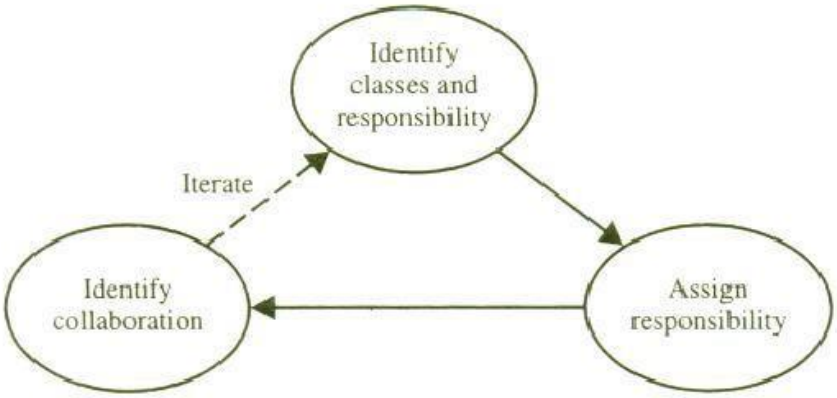


Fig 3.17 : The Classes, Responsiblities and Collaborators process.

CASE STUDY : 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 3.18).

Classes, Responsibilities, and Collaborators for the Account object.

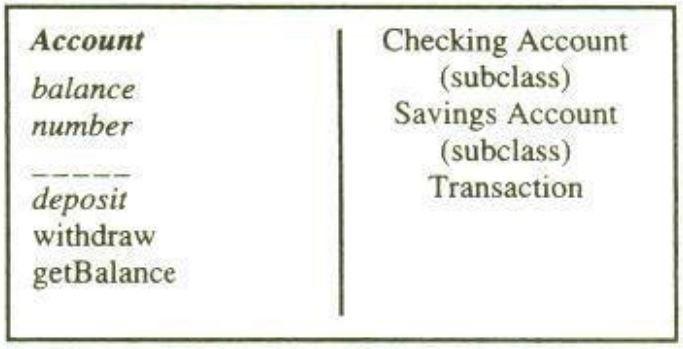


Fig 3.18: Classes, Responsibilities and Collaborators for the Account Object.

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.

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.

NAMING CLASSES

Naming a class in an important activity.

Guidelines for Naming Classes

- The class should describe a single object, so it should be the singular form of noun.
- Use names that the users are comfortable with.
- The name of a class should reflect its intrinsic nature.
- By the convention, the class name must begin with an upper case letter.
- For compound words, capitalize the first letter of each word - for example, Loan Window.

IDENTIFYING OBJECT, RELATIONSHIPS, ATTRIBUTES, & METHODS

In an object oriented environment, objects take on an active role in a system. All objects stand in relationship to others on whom they rely for services and control. The relationship among objects is based on the assumption 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 orgainsed 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, The relationships among objects are known as **associations.**
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.

.

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 previous 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.

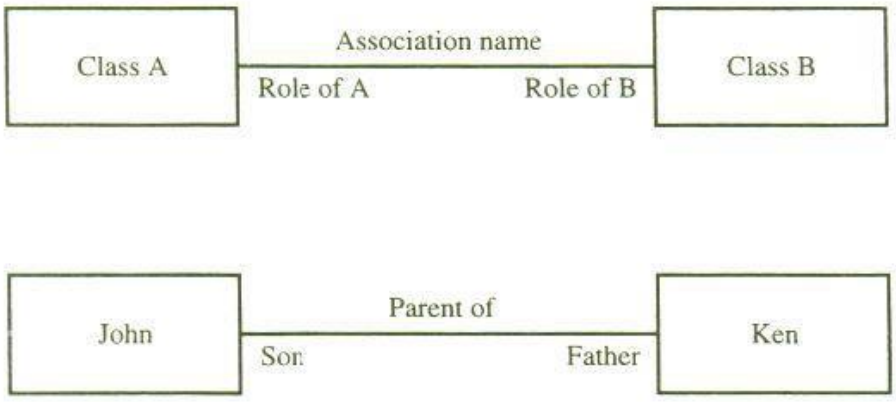


Fig 3.19: Basic Associations

i) 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,"'arid 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.**

ii) Guidelines For Identifying Association

The 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.

iii) Common Association Patterns

The common association patterns are based on some of the common associations defined by researchers and practioners: Rumbaugh et al. Coad and Yourdon , and others.

These include .

Location 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.

Communication association – talk to, order to. For example, a customer places an order (communication association) with an operator person (see Figure 3.20).

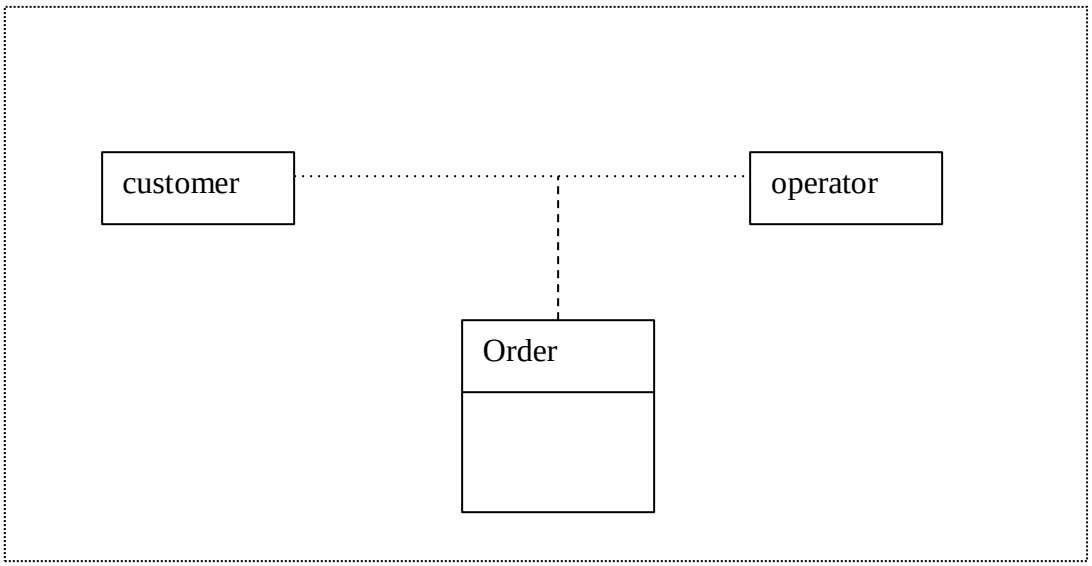


Fig 3.20: A customer places an order (communication association) with an operator person.

These association patterns and similar ones can be stored in the repository and added to as more patterns are discovered.

IV) Eliminate Unnecessary Associations

- **Implementation 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 3.21).

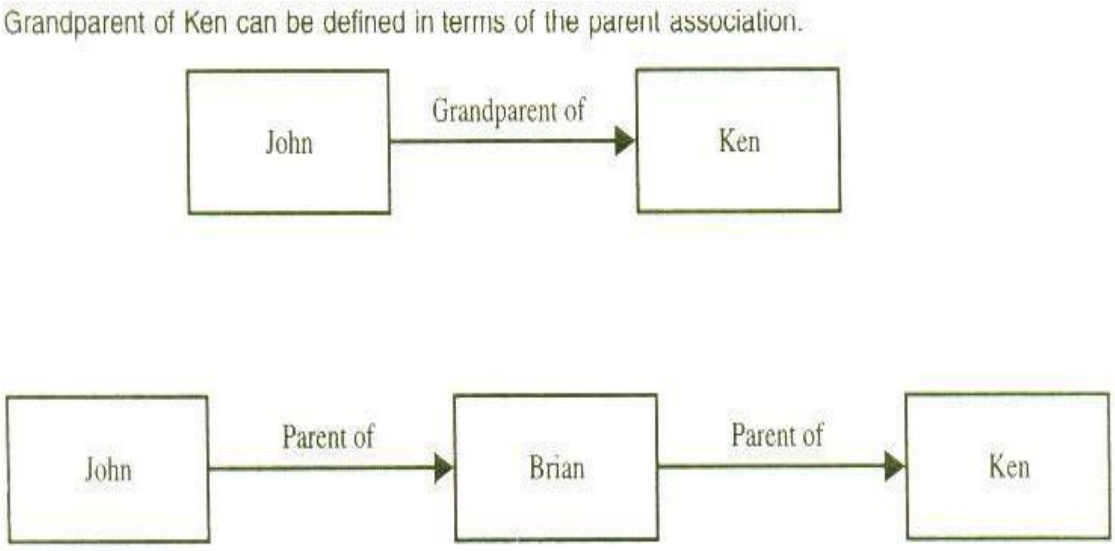


Fig 3.21 – Grandparent of Ken can be defined in terms of the parent association.

SUPER-SUB CLASS RELATIONSHIPS

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. 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.

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.**

i) <u>Guidelines For Identifying Super-Sub Relationship, A Generalization</u>

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.
- **Reusability.** 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.
- **Multiple inheritance**. Avoid excessive use of multiple inheritances. 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. (See fig 3.22)

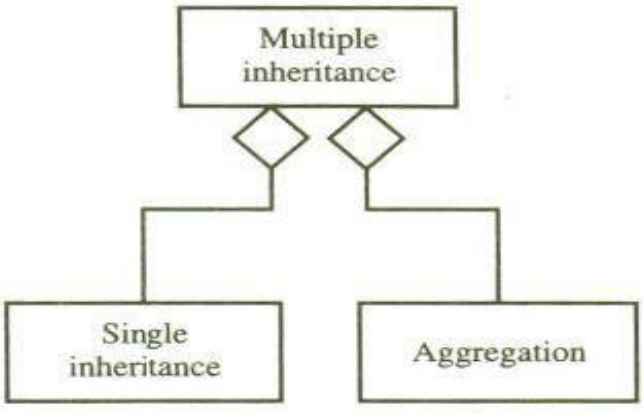


Fig. 3.22 – One way of achieving the benefits of multiple inheritancefrom the most appropriate class.

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 3.23).

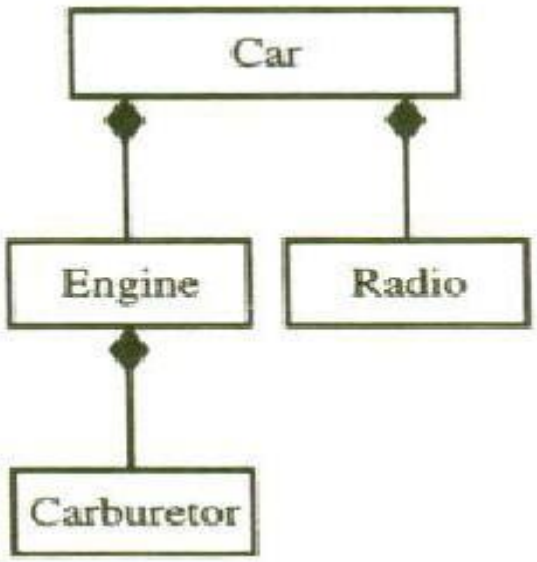


Fig 3.23 - A-part-ofcomposition.A carburetoris a part of an engine and an engine and aradio are parts of a car.

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. Figure3.23 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 caris 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 :

- . 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 3.24, 3.25) and use a solid diamond to represent composition, which is a strong form of aggregation (see Figure 3.23).

A house is a container.

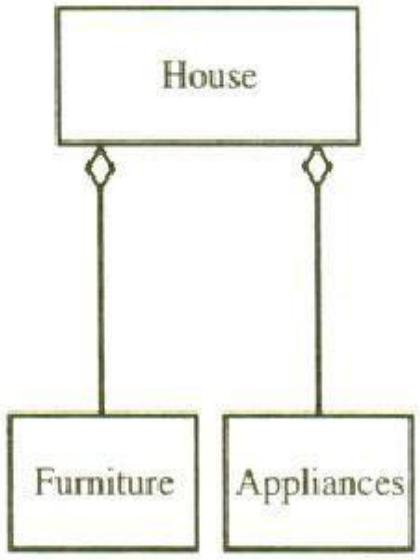


Fig 3.24 A house is a container

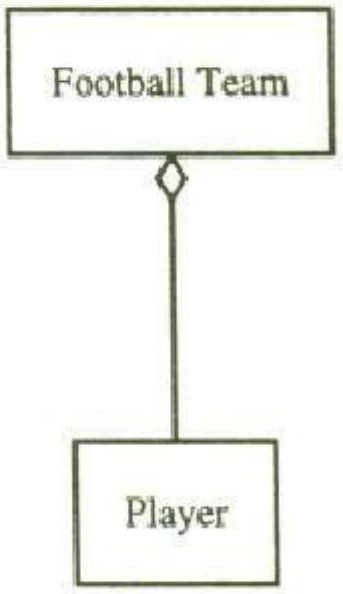


Fig 3.25. A football team is a collection of players.

i) 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 3.24).
- **Collection-member**. A conceptual whole encompasses parts that may be physical or conceptual; for example, a football team is a collection of players.(fig.3.25).

CASE STUDY: RELATIONSHIP ANALYSIS FOR THE VIANET BANK ATM SYSTEM

1) 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.

2) 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.

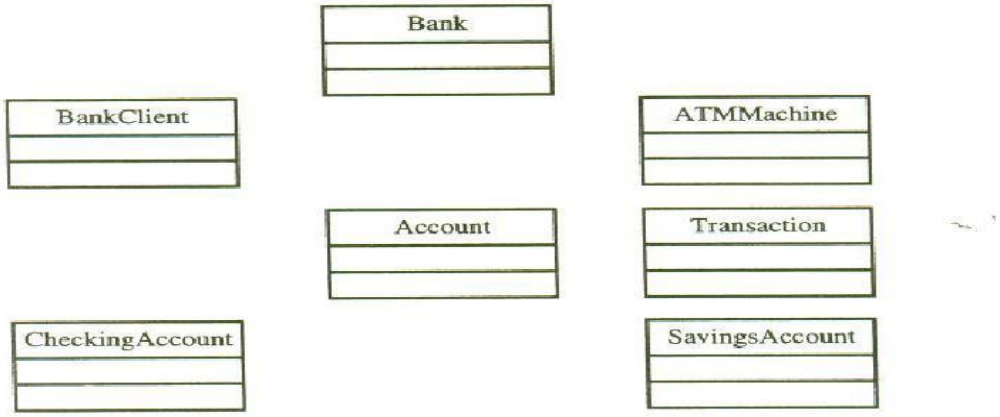


Fig 3.26. UML class diagram for the ViaNet bank ATM system.

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 3.26).

3) 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:

- 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.

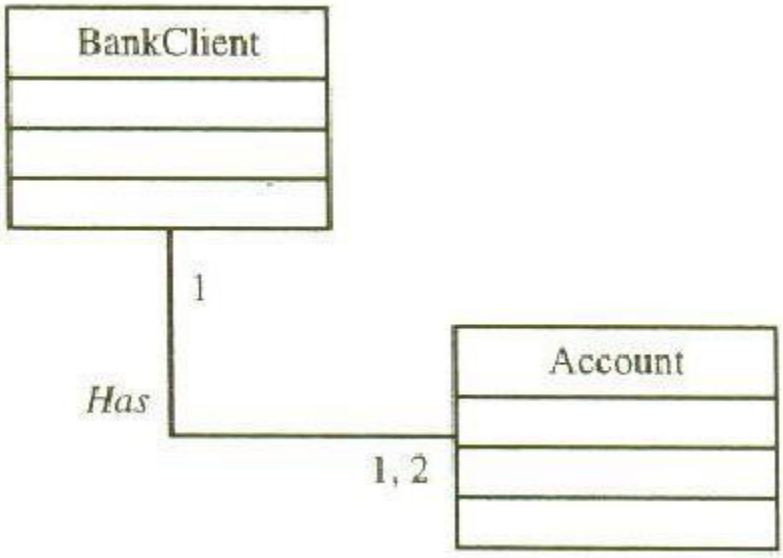


Fig 3.27. Defining the BankClient-Accountassociation multiplicity.One Client can haveone or more Accounts (checking and savings accounts).

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 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 3.27). Other associations and their cardinalities are defined in Table 8-1 and demonstrated in Figure 3.28 .

Table 8.1

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	Savings-Checking	One
CheckingAccount	SavingsAccount		Zero or one
Account	Transaction	Account-Transaction	Zero or more
Transaction	Account		One

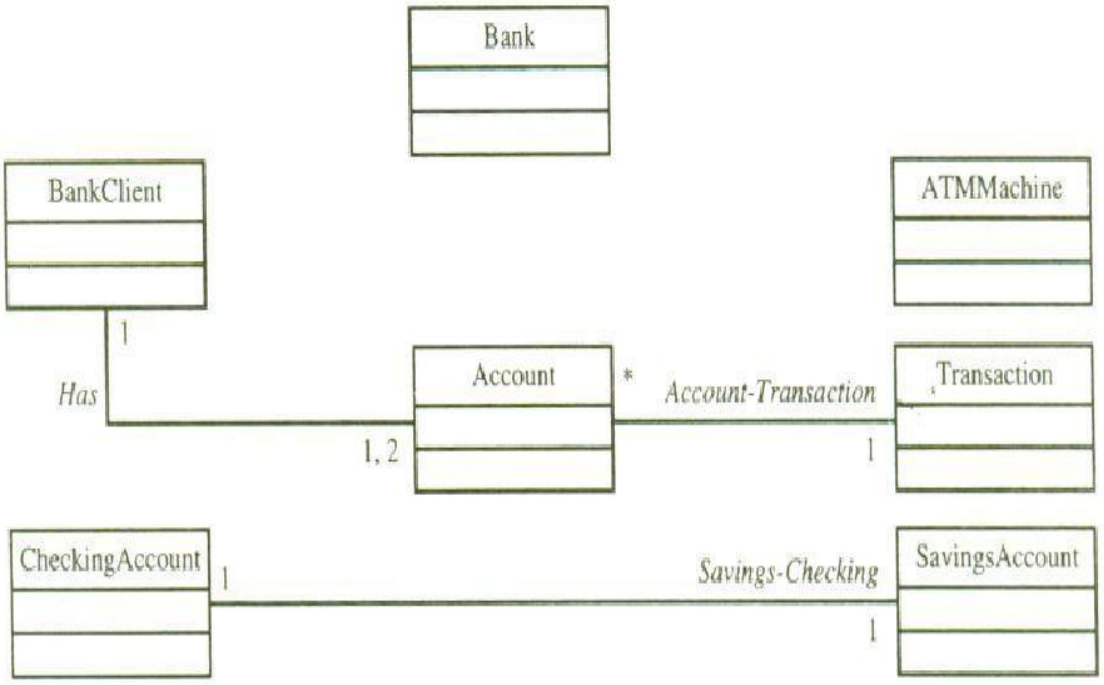


Fig 3.28. Associations among the ViaNet bank ATMsystem classes.

4) 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, class will define attributes and services common to all kinds of accounts, with CheckingAccount and SavingsAccount each defining methods that make them more specialized. Fig 3.29.

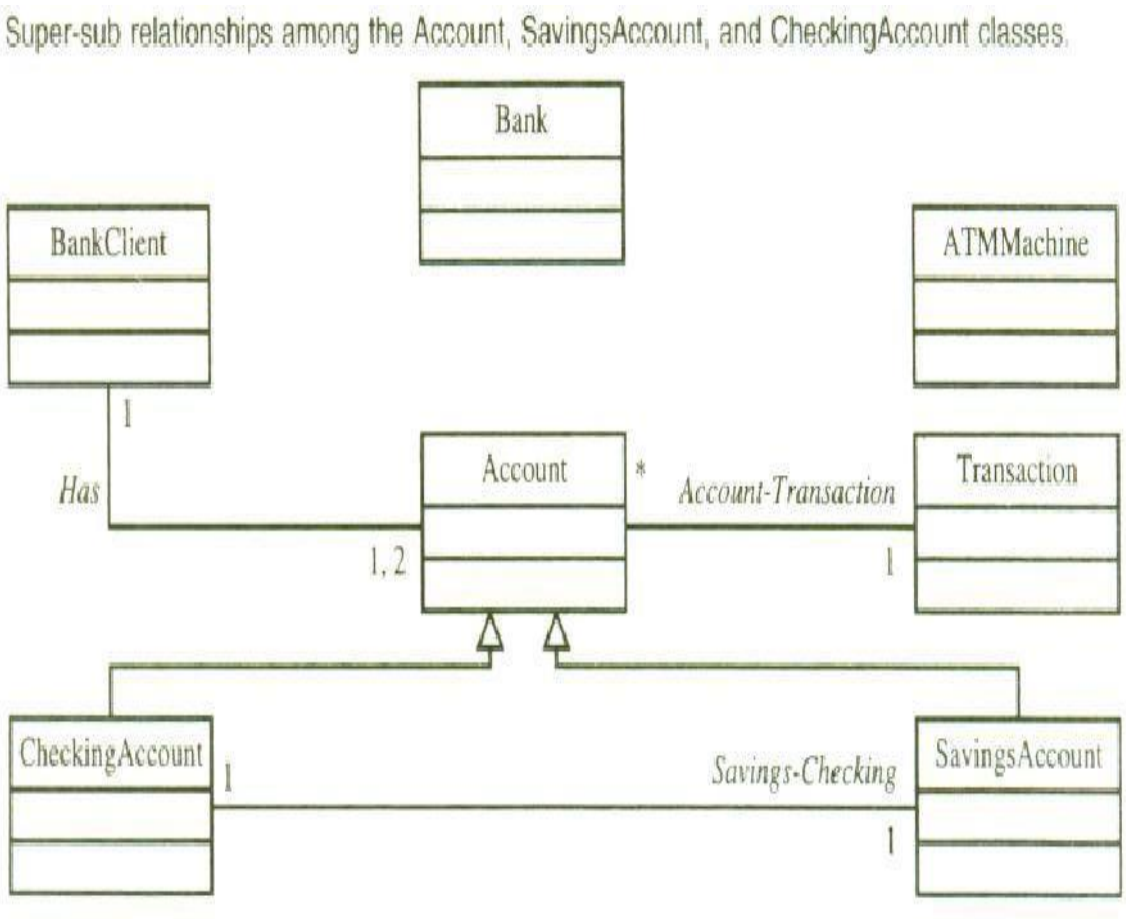


Fig 3.29- Super-sub relationships among the Account, SavingsAccount and CheckingAccount Classes

5) 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.

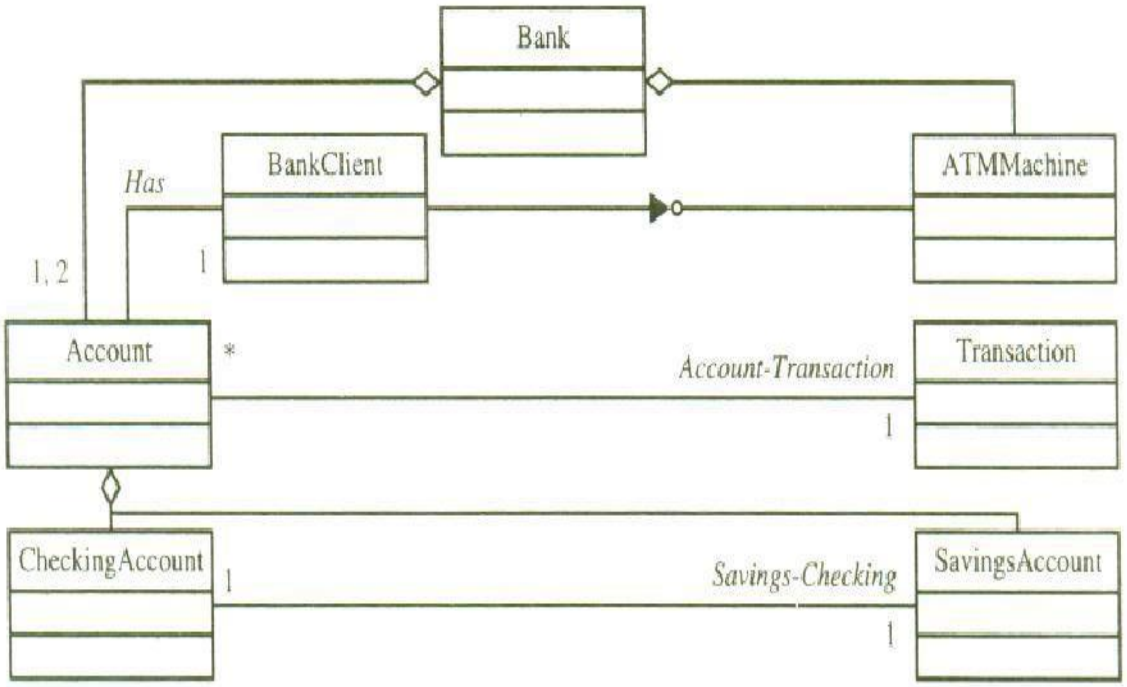


Fig 8.30. Association, generalization, and aggregation among the ViaNet bank classes. Notice that the super-sub arrows for CheckingAccount and SavingsAccount have merged. The relationship between BankClient and ATMMachine is an interface.

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 3.30 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.

CLASS RESPONSIBILITY : IDENTIFYING ATTRIBUTES AND METHODS

Identifying attributes and methods is like finding Classes, still a difficult activity and an iterative process.

Responsibilities identify problems to be solved.

Attributes are things an object must remember such as color, cost and manufacturer. Identifying attributes of a system's classes starts with understanding the system' s responsibilities.

The following questions help in identifying the responsibilities of classes and deciding what data elements to keep track of :

#. What information about an object should we keep track of?

What services must a class provide?

CLASS RESPONSIBILITY : DEFING ATTRIBUTES BY ANALYZINGUSE CASES AND OTHER UML DIAGRAMS

Attributes can be derived from scenario testing ; therefore, by analyzing the use cases and sequence/collaboration, activity and state diagrams, you can begin to understand classes responsibilities and how they must interact to perform their tasks.

The main goal is to understand what the class is responsible for knowledging.

Responsibility is the issue.

What kind of questions what kind of question would you like ask;

- ✓ How am I going to be used?
- ✓ How am I going to collaborate with other classes?
- ✓ How am I described in the context of this system's responsibility?

Guidelines for Defining Attributes.

- **Attributes usually correspond to nouns followed by prepositional phrases such as cost of the soup. Attributes also may correspond to adjectives or adverbs.**
- **Keep the class simple; state only enough attributes to define the object state.**
- **Attributes are less to be fully described in the problem statement.**
- **Omit derived attributes.**
- **Do not carry discovery of attributes to excess. You can add more attributes in subsequent iterations.**

Important point to remember is that you may think of many attributes that can be associated with a class. You must careful to add only those attributes necessary to the design at hand.

OBJECT RESPONSIBILITY : METHODS AND MESSAGES.

Objects not only describe abstract data but also must provide some services.

Methods and messages are the workhorses of object oriented systems.

In an object oriented environment, every pieces of data or object is surrounded by a rich set of routines called **methods.**

These methods do everything from printing the object to initializing its variables.

Every class is responsible for storing certain information from the domain knowledge. It also is logical to assign the responsibility for performing any operation necessary on that information.

Operations (methods or Behavior) in the o-o-system usually correspond to queries about attributes.

Methods are responsible for managing the value of attributes such as query, updating , reading and writing;

CASE STUDY : DEFINING ATTRIBUTES FOR VIANET BANK OBJETCTS.

1. **Defining Attributes for the BankClient Class**

   By analysing the use cases, the sequence/collaboration diagrams and activity diagram of bank atm process, it is apparent that, for the BankClient Class, the problem domain and system dictate certain attributes. In essence, what does the system need to know about the BankClient?

By looking at the activity diagram (See Fig 3.9) we notice that the BankClient must have a PIN number and CardNumber. Therefore, the PIN number and CardNumber are appropriate attributes for the BankClient.

The Attributes of the BankClient are
firstName
lastName
pinNumber
cardNumber
account:Account

At this stage of the design we are concerned with the functionality of the BankClinet object and not with implementation attributes.

2. **Defining Attributes for the AccountClass**.

Similarly, what information does the system need to know about an account? Based on the ATM Usecase diagram, Sequence/Collaboration diagram and activity diagram, BankClient can interact with its account by entering the account number and the could deposit money, get an account history, or get the balance. Therefore, we have defined the following attributes for the Account Class : number, balance.

CASE STUDY: DEFINING METHODS BY ANALYZING UML DIAGRAMS AND USE CASES.

We know that, in a sequence diagram, the objects involved are drawn on the diagram as vertical dashed lines. Furthermore, the events that occur between objects are drawn between the vertical object lines. An Event is considered tobe an action that transmits information.

For example, to define methods for the Account class, we look at sequence diagrams for the following use cases.

Deposit Checking
Deposit Savings
Withdraw Checking
Withdraw More from Checking
Withdraw Savings
Withdraw Savings Denied
Checking Transaction History
Savings Transaction History

CASE STUDY: DEFINING METHODS BY BANK OBJECTS

.

Operations (methods or behavor) in the object-oriented system usually correspond to events or actions that transmit information in the sequence diagram or queries about attributes of the objects. In other words, methods are responsible for managing the value fo attributes such as query, updating, reading and writing.

1) Defining Account Class operations.

Deposit and withdrawal operations are available to the Client through the bank application, but they are provided as services by the Account Class, since the accout objects must be able to manipulate their internal attributes. Accout objects also must be able to create transaction records of any deposit or withdrawal they perform.

Here are the methods that we need to define for the Account Class:

deposit
withdraw
createTransaction.

The services added to the Account class are thopse that apply to all subclasses of Account; namely, CheckingAccount and SavingsAccount. The subclass will either inherit these generic services without chage or enhace them to suit their own needs.

2) Defining BankClient Class Operation

Analyzing the sequence diagram(fig 3.13), it is apparent that the BankClient requires a method to validate client's passwords.

3) Defining CheckingAccount Class Operations

The requirement specification states that, when a checking account has insufficient funds to cover a withdrawal, it must try to withdraw the insufficient amount from its related saving account. To provide the service, the CheckingAccount class needs a withdrawal service that enables the transfer. Similarly, we must add the withdrawal service to the CheckingAccount class.

Questions

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	List out the steps in OOA process.	Remember	BTL 1
2.	List out the guidelines for developing effective documentation.	Remember	BTL 1
3.	Analyze the following approaches a) CRC b) noun phrase	Analyze	BTL 4
4.	List out the guidelines for developing use case models.	Remember	BTL 1
5.	What is Aggregation?	Remember	BTL 1
6.	Define 'uses' and 'extends' association.	Remember	BTL 1
7.	Define classification	Remember	BTL 1
8.	List the approaches for identifying classes	Remember	BTL 1
9.	Explain how we can identify actors for a system.	Understand	BTL 2
10.	Compare Aggregation and Composition.	Evaluate	BTL 5
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Discuss in detail about Object Analysis Classification.	Understand	BTL 2
2.	Explain noun phrase approach with an example.	Analyze	BTL 4
3.	Analyze the approaches for identifying classes	Analyze	BTL 4
4.	Discuss in brief about a)CRC approach b)Associations	Understand	BTL 2
5.	Explain the guidelines for developing effective documentation in detail	Remember	BTL 1
6.	Explain use-case driven approach with example.	Analyze	BTL 4
7.	Explain in detail about Commom class patterns approach.	Remember	BTL 1

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – IV - OBJECT ORIENTED ANALYSIS AND SYSTEM ENGINEERING - SCSA1401

UNIT 4

OBJECT ORIENTED DESIGN

Object Oriented Design Process - Object Oriented Design Axioms - Corollaries - Designing Classes: Object Constraint Language - Process of Designing Class - Class Visibility - Refining Attributes - Access Layer: Object Store and Persistence - Database Management System - Logical and Physical Database Organization and Access Control - Distributed Databases and Client Server Computing - Object Oriented Database Management System – Object Relational Systems – Designing Access Layer Classes - View Layer: Designing View Layer Classes - Macro Level Process - Micro Level Process – Purpose of View Layer Interface - Prototyping the user interface.

The Object Oriented Design Process and Design Axioms

Designing systems using self-contained objects and object classes

- To explain how a software design may be represented as a set of interacting objects that manage their own state and operations
- To describe the activities in the object-oriented design process
- To introduce various models that describe an object-oriented design
- To show how the UML may be used to represent these models

Characteristics of OOD

- Objects are abstractions of real-world or system entities and manage themselves
- Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services
- Shared data areas are eliminated. Objects communicate by message passing
- Objects may be distributed and may execute sequentially or in parallel

THE OBJECT ORIENTED DESIGN PROCESS

During the design phase the Classes in o-o- analysis must be revisited with a shift in focus to their implementation. New Classes or attributes and methods must be added for implementation purposes and their user interfaces.

The o-o design process consists of the following activities (Fig 4.1):

1. Apply design axioms to design classes, their attributes, methods, associations, structures and protocols.

   Refine and complete the static UML Class diagram by adding details

to the uml class diagram. This step consists of the following activities.

Refine attributes.

Design methods and protocols by utilizing a UML Activity diagram to represent the method's algorithm.

Refine associations between classes (if required)

Refine class hierarchy and design with inheritance (if required)

Iterate and refine again.

2. Design the access layer

Create mirror classes. For every business class identified and created, Create one access class.

Identify access layer class relationships.

Simplify Classes and the relationships. – The main goal is to eliminate redundant classes and structures.

Redundant Classes: Do not keep 2 classes that perform similar translate request and translate results activities. Simply select one and eliminate the other.

Method Classes: Revisit the classes that consist of only one or two methods to see if they can be eliminated or combine with existing classes.

Iterate and refine again

3. Design the view layer classes.

Design the macro level user interface, identifying view layer objects.

Design the micro level user interface, which includes these activities:

Design the view layer objects by applying the design axioms and corollaries.

Build a prototype of the view layer interface.

Test usability and user satisfaction

Iterate and refine.

4. Iterate and refine the whole design. Reapply the design axioms and if needed, repeat the proceeding steps.

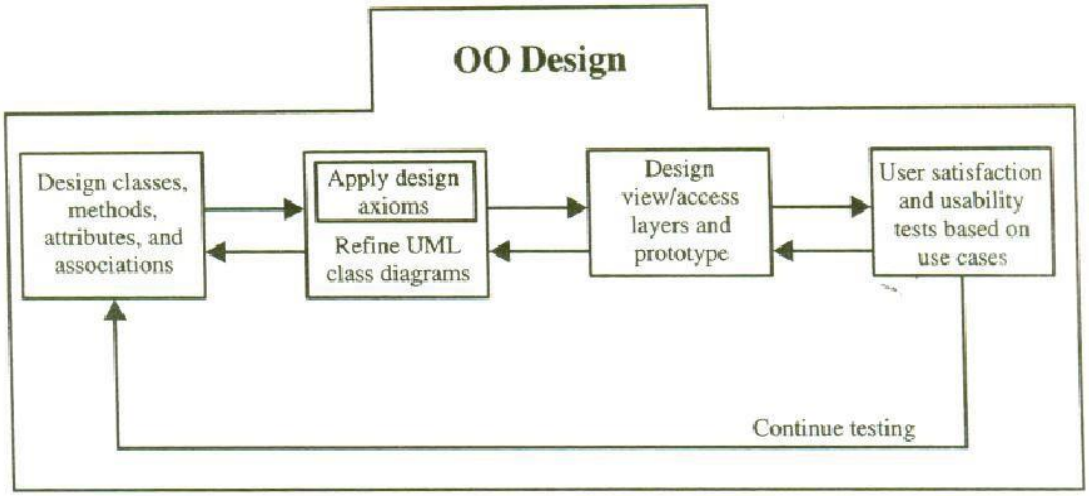


Fig 4.1 The object-oriented design process in the unified approach.

OBJECT-ORIENTED DESIGN AXIOMS

- Main focus of the analysis phase of SW development → "what needs to be done"?
- Objects discovered during analysis serve as the framework for design.
- Class's attributes, methods, and associations identified during analysis must be designed for implementation as a data type expressed in the implementation language.
- During the design phase, we elevate the model into logical entities, some of which might relate more to the computer domain (such as user interface, or the access layer) than the real world or the physical domain (such as people or employees). Start thinking how to actually implement the problem in a program.
- The goal → to design the classes that we need to implement the system.
- Design is about producing a solution that meets the requirements that have been specified during analysis.
- Analysis Versus Design.

• Analysis :	• Design:
○ *Focus on understanding the problem*	○ *Focus on understanding the solution*
○ *Idealized design*	○ *Operations & attributes*
○ *Behavior*	○ *performance*
○ *System structure*	○ *close to real code*
○ *Functional requirements*	○ *object lifecycles*
○ *A small model*	○ *non-functional requirements*
	○ *a large model*

- An **axiom** = is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception.
- A **theorem** = is a proposition that may not be self-evident but can be proven from accepted axioms. Therefore, is equivalent to a law or principle?
- A **theorem** is valid if its referent axioms & deductive steps are valid.
- A **corollary** = is a proposition that follows from an axiom or another proposition that has been proven.
- **Suh's design axioms to OOD** :

- 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
- Axiom 2 → concerned with simplicity. Rely on a general rule known as ***Occam's razor.***
  - *Occam's* razor rule of simplicity in OO 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.

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 4.2):

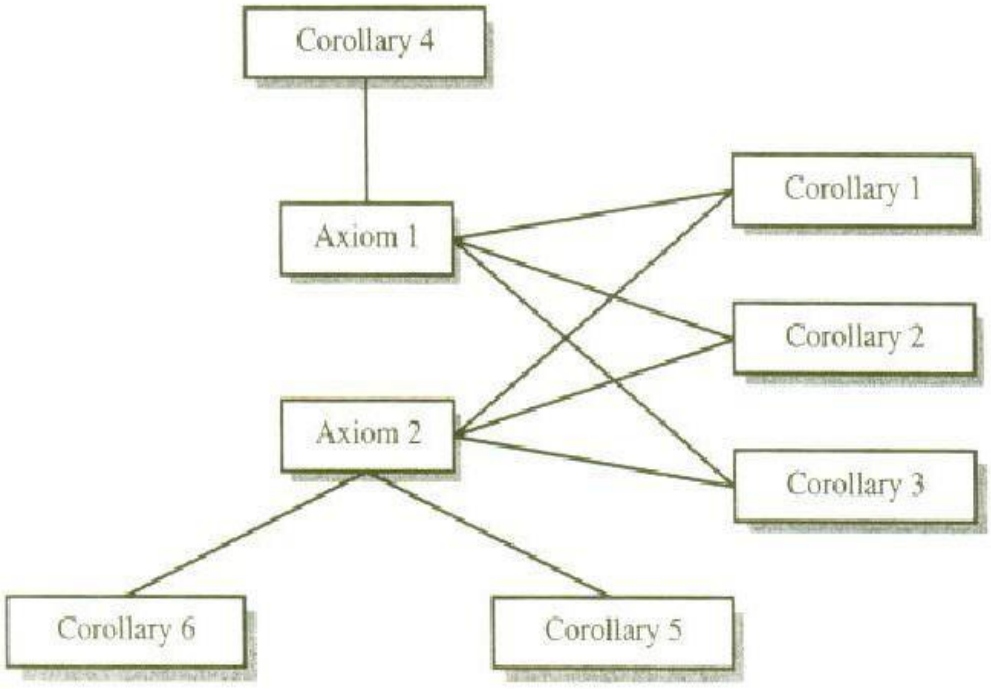


Fig 4.2 The origin of corollaries. Corollaries 1, 2, and 3 are from both axioms, whereasCorollary 4 is from axiom 1 and corollaries 5 and 6 are from axiom 2.

- 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.
- 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

- Highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects.
- The main goal 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

- ***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 ➔ *the amount & complexity of messages between components.*

  - Desirable to have a 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.
  - Minimize the number of messages sent & received by an object
  - Types of coupling among objects or components , refer

. 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 4.3).

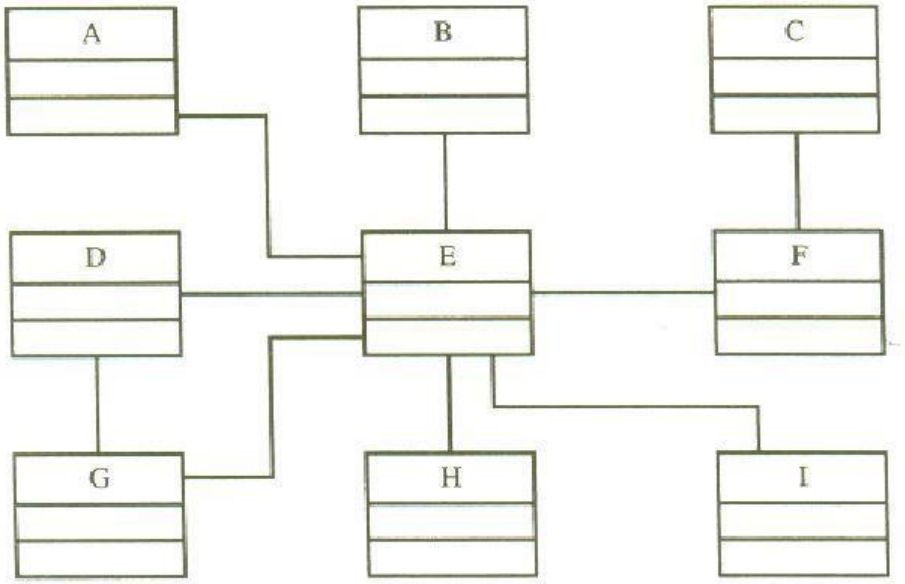


Fig 4.3 E is a tightly coupled object.

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.

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.

- ***Inheritance coupling -> coupling between super-and subclasses***
  - A subclass is coupled to its superclass in terms of attributes & methods
  - High inheritance coupling is desirable
  - Each specialization class should not inherit lots of unrelated & unneeded methods & attributes.

Cohesion

- **Cohesion** deals with interaction within a single object or software component.
- Need to consider interaction within a single object or sw component ➔ **Cohesion**
- Cohesion ➔ reflects the '**single-purposeness**' of an object (see corollaries 2 & 3)
- 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 ➔ a method should carry only one function.
- A method 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: 1.How interrelated are the classes? 2.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. 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

- Keeping the classes simple allows reusability
- A class that easily can be understood and reused (or inherited) contributes to the overall system
- Complex & poorly designed class usually cannot be reused
- Guideline ➔ The smaller are your classes, the better are your chances of reusing them in other projects. Large & complex classes are too specialized to be reused
- The emphasis OOD places on encapsulation, modularization, and polymorphism suggests reuse rather than building anew
- Primary benefit of sw reusability ➔Higher productivity

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 terms of new lines of code written plus a discounted credit (e.g., 50 percent less credit) for reused lines of code.

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 class design 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).

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.

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 CommercialVehicleo These are further subdivided into whatever level of specificity seems appropriate (see Figure 4.4).

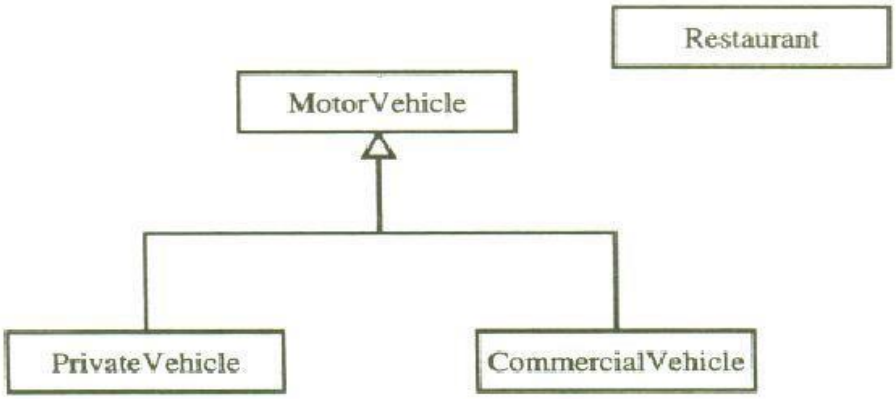


Fig 4.4 The initial single inheritance design.

Subclasses of Restaurant are designed to reflect their own licensing procedures. This is a simple, easy to understand design,

In any case, the design is approved, implementation is accomplished, and the system goes into production.

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 Figure4.5).

The single inheritance design modified to allow licensing food trucks.

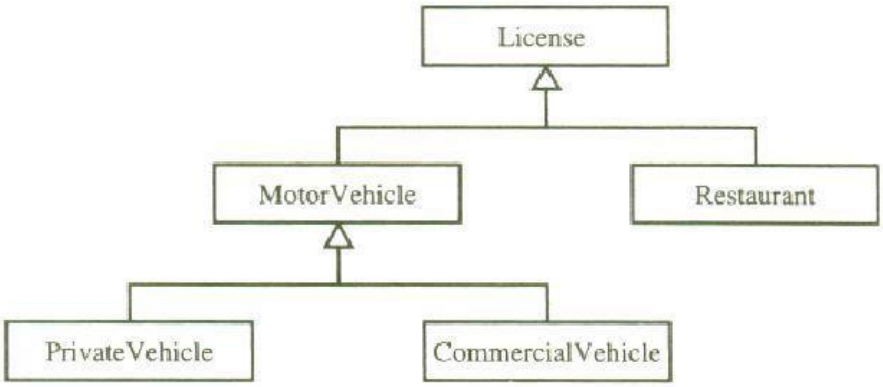


Fig 4.5 The single inheritance design modified to allow licensing food trucks.

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.

DESIGN PATTERNS

- Provides a scheme for refining the subsystems or components of a sw system or the relationships among them
- Are devices that allow systems to share knowledge about their design, by describing commonly recurring structures of communicating components that solve a general design problem within a particular context
- The main idea ➔ to provide documentation to help categorize & communicate about solutions to recurring problems
- The pattern has a name to facilitate discussion and the information it represents .

  For example , refer the book . page no 212.

Designing Classes

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.

Objectives To explain how a software design may be represented as a set of interacting objects that manage their own state and operations To describe the activities in the object-oriented design process To introduce various models that describe an object-oriented design To show how the UML may be used to represent these models

Characteristics of OOD :

- Characteristics of OOD Objects are abstractions of real-world or system entities and manage themselves Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services Shared data areas are eliminated.
- Objects communicate by message passing Objects may be distributed and may execute sequentially or in parallel

Advantages of OOD :

- Advantages of OOD Easier maintenance.
- Objects may be understood as stand-alone entities Objects are appropriate reusable components For some systems, there may be an obvious mapping from real world entities to system objects

Object-oriented development :

Object-oriented development Object-oriented analysis, design and programming are related but distinct OOA is concerned with developing an object model of the application domain OOD is concerned with developing an object-oriented system model to implement requirements OOP is concerned with realising an OOD using an OO programming language such as Java or C++

Objects and object classes :

Objects and object classes Objects are entities in a software system which represent instances of real-world and system entities Object classes are templates for objects. They may be used to create objects Object classes may inherit attributes and services from other object classes

Objects :

Objects An object is an entity which has a state and a defined set of operations which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required. Objects are created according to some object class definition. An object class definition serves as a template for objects. It includes declarations of all the attributes and services which should be associated with an object of that class.

UML OBJECT CONSTRAINT LANGUAGE

The UML is a graphical language with a set of rules and semantics. The rules and semantics of the UML are expressed in English, a form known as **OBJECT CONSTRAING LANGUAGE. OCL** is a specification language that uses simple logic for specifying the properties of a system.

Many UML modeling constructs require expression: For eg; there are expressions for types, Boolean values and numbers.

Expressions are stated as strings in ocl. The syntax for some common navigational expressions is shown here. These forms ca be chained together. The Leftmost element must be an expression for an object or a set of objects. The expressions are meant to work on sets of values when applicable.

- Item-Selector : - The selector is the name of an attribute in the item. The result is the value of the attribute. For eg: John.age(the age is attribute of the object john, and john.age represents the value of the attribute).
- Item-selector [qualifier-value]. The selector indicates a qualified association that qualifies the item. The result is the related object selected by an qualifier. For eg; John.phone[3], assuming john has several phone.
- Set -> select (Boolean-expression). The Boolean expression is written in terms of objects within the set. The result is the subset of objects in the set for which the Boolean expression is true.

  For eg; company.employee->salary->50000. This represents employees with salaries over $50,000.

DESINGING CLASSES : THE PROCESS

In this section we concentrate on step 1 of the design process described in chap 9, which consists of the following activities:

1. Apply design axioms to design classes, their attributes, methods, associations, structures and protocols.

   Refine and complete the static UML Class diagram by adding details to the uml class diagram. This step consists of the following activities.

   Refine attributes.

   Design methods and protocols by utilizing a UML Activity diagram to represent the method's algorithm.

   Refine associations between classes (if required)

   Refine class hierarchy and design with inheritance (if required)

   Iterate and refine again.

   O-O design is an iterative process. After all, design is as much about discovery as construction.

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.

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 the 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.

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.

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

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?

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.

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).

UML Attribute Presentation

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[lO]: 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[O..lj: String

REFINING ATTRIBUTES FOR THE VIANET BANK OB.JECTS

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
lastN ame
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 Chapter 8 we identified an association between the BankClient and the Account classes. (see Figure 3.27). 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 Transactionclass are these:
#transID: String
#transDate: Date
#trans Time: Time
#transType: String
#amount: float
#postBalance: float

Refining Attributes for the ATMMachine 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 10-1 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.)

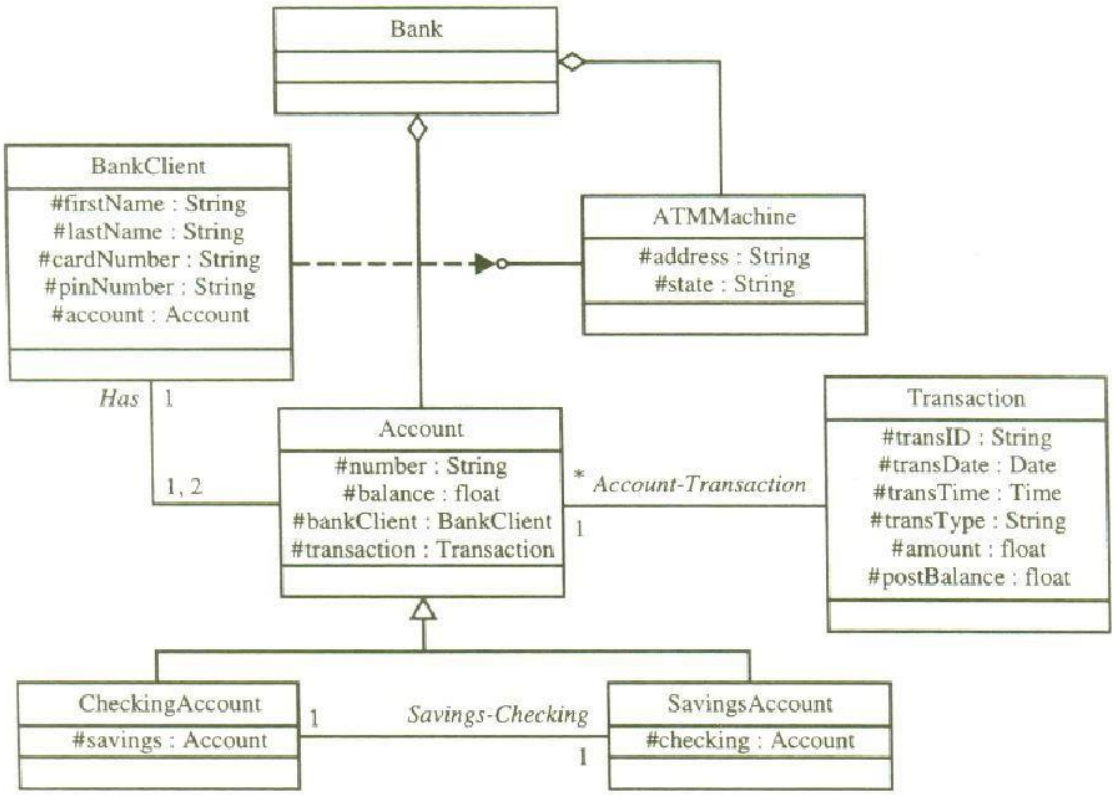


Fig 4.6 A more complete UML class diagram for the ViaNet bank system.

DESIGNING METHODS AND PROTOCOLS

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.

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.

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.

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).

*Move some function into new classes that the object would use. Apply corrolary 1 (Uncoupled design with less information content).

*Break up the class into 2 or 3 classes. Apply corollary 3 (large number of simple classes).

*Rethink the class definition based on experience gained.

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 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).

Here, The name is the name of the operation.

Parameter list is a list of parameter, separated by commas, each specified by name:type-expression=default 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.

DESIGNING METHODS FOR THE VIANET BANK OBJECTS.

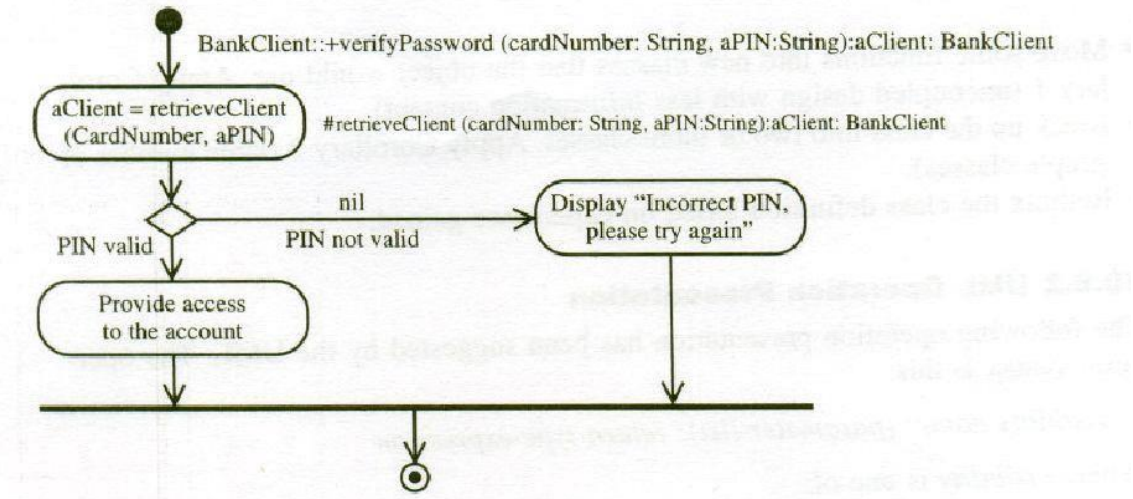


Fig 4.7An activity diagram for the BankClient class verifyPassword method, using OCl to describe the diagram. The syntax for describing a class's method is Class name::methodName.

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 verify-Password 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 4.7).

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 retrieveClientmethod 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 next chapter.

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 4.8).

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).

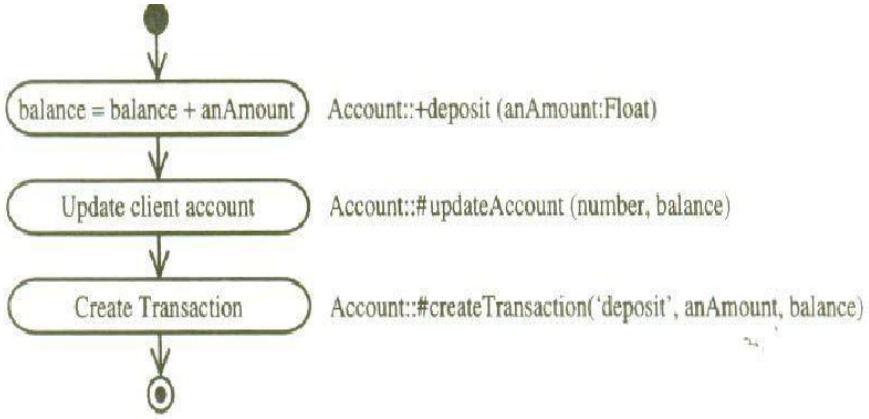


Fig 4.8 An activity diagram for the Account class deposit method.

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 Figure4.9).

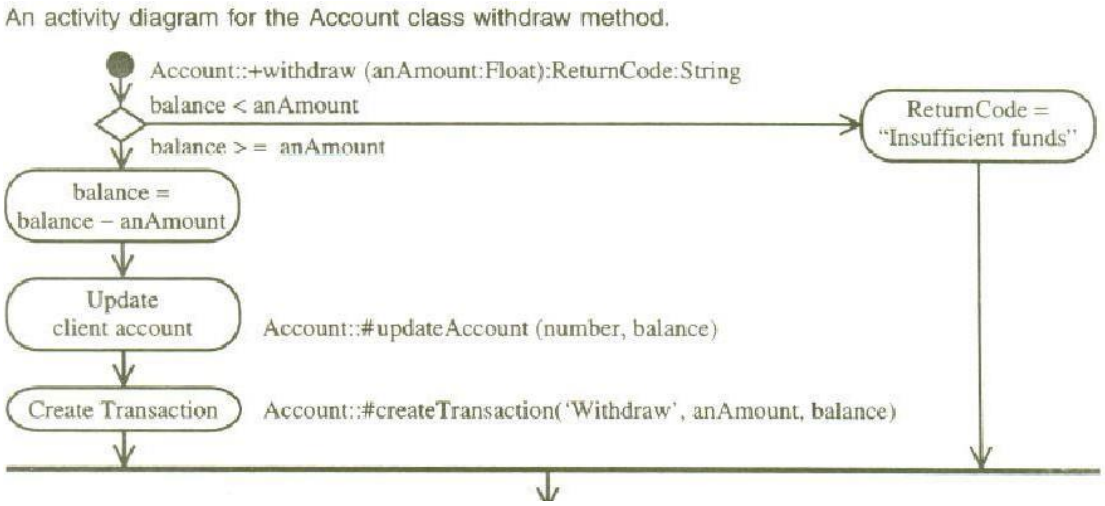


Fig 4.9 An activity diagram for the account class withdraw method.

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 performed 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 information. Add this description to the *create Transaction* 's description field (see Figure 4.10).

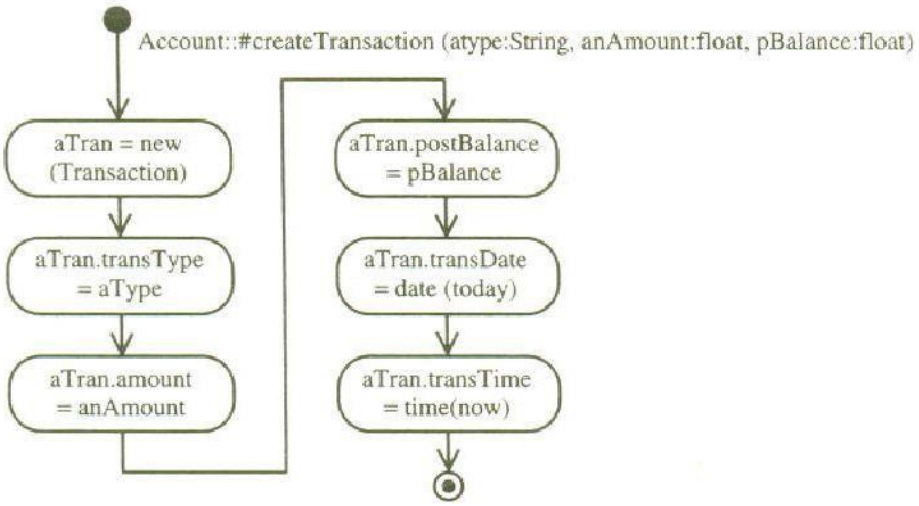


Fig 4.10An activity diagram for the Account class createTransaction method.

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.

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 4.11). 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.

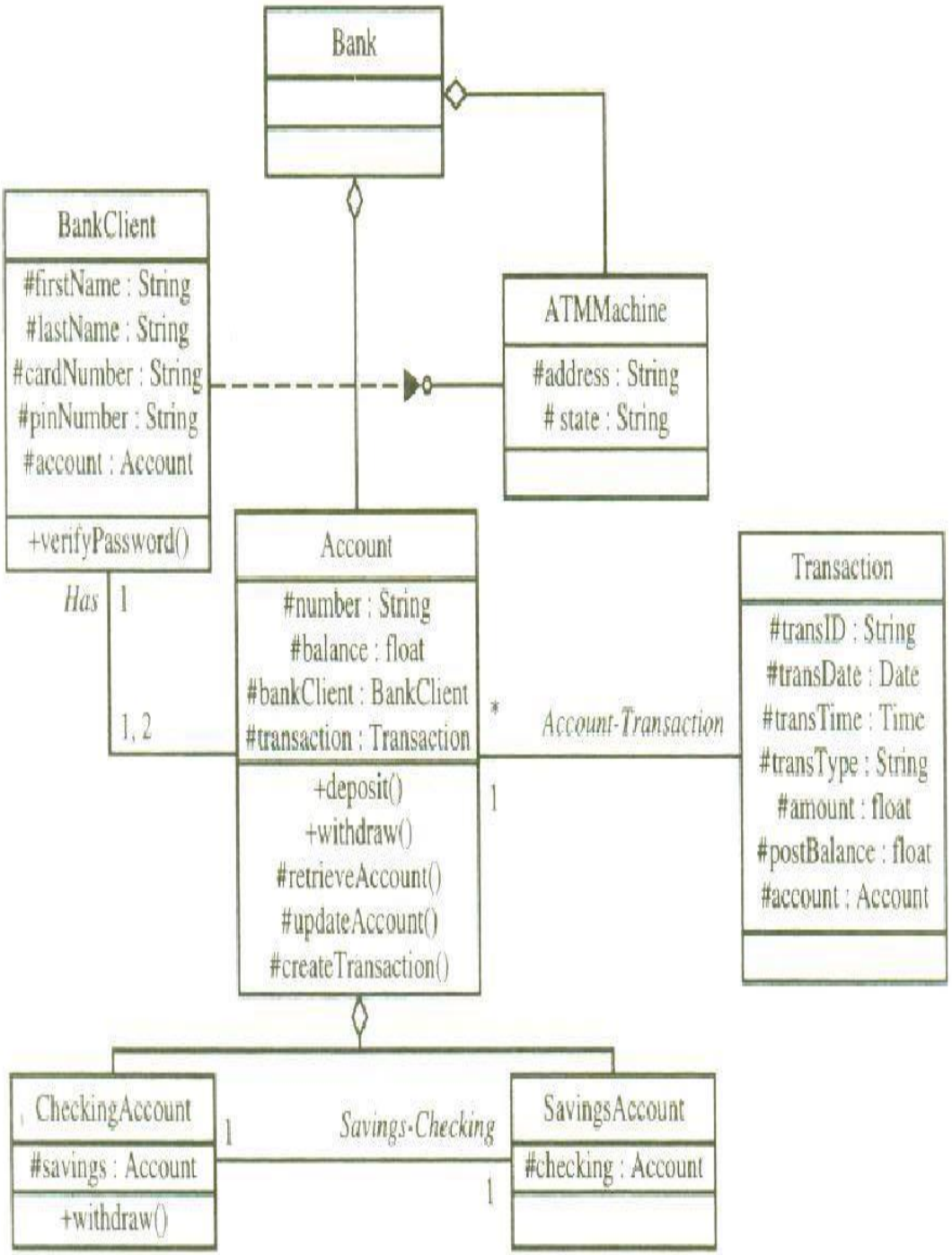


Fig 4.11 More complete UML class diagram for the ViaNet bank ATM system. Note that the method parameter list is not shown.

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.

ACCESS LAYER : OBJECT STORAGE & OBJECT INTEROPERABILITY

A **DataBase Management System (DBMS**) is a set of programs that enables the creation and maintenance of a collection of related data. A DBMS and associated programs access, manipulate, protect and manage the data.

The fundamental purpose of a DBMS is to provide a reliable, persistent data storage facility and mechanisms for efficient, convenient data access and retrieval.

Persistence refers to the ability of some objects to outlive the programs that created them.

Object lifetimes can be short for local objects (called **transient objects**) or long for objects stored indefinitely in a database (called **persistent objects**).

Most object-oriented languages do not support serialization or object persistence, which is the process of writing or reading an object to and from a persistence storage medium, such as disk file.

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.**
- 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.

DATABASEMANAGEMENT SYSTEMS

Databases usually are large bodies of data seen as critical resources to a company. 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).

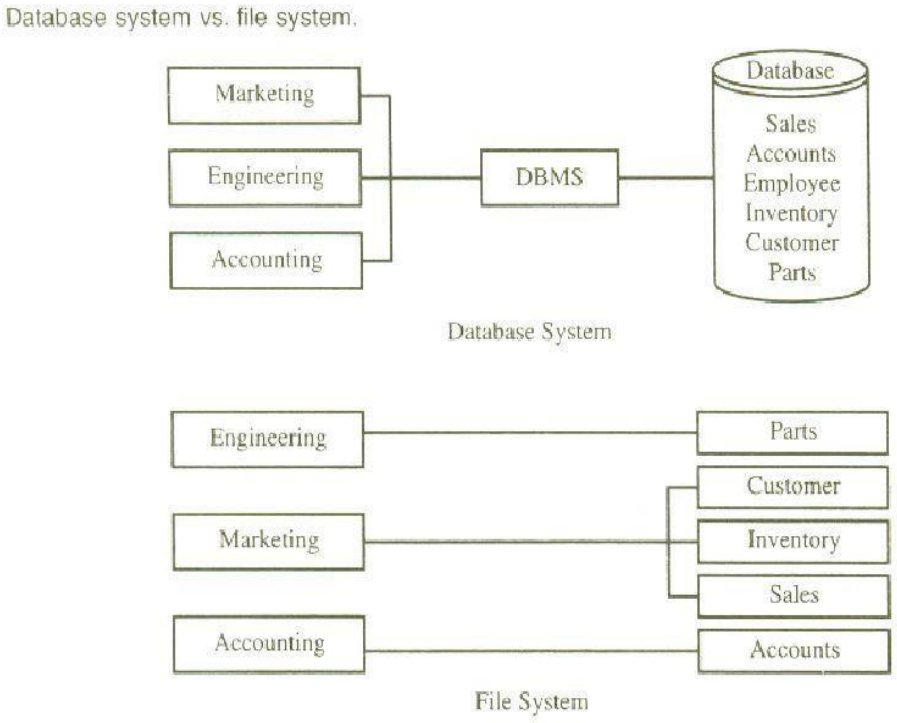


Fig.4.11: Database system vs file system.

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.

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 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.

Database models may be grouped into two categories: **conceptual models and implementation models.**

The **conceptual mode**l focuses on the **logical nature of that data presentation**. Therefore, the conceptual model is concerned with ***what* is represented in the database.**

The **implementation model** is concerned with *how* it is represented.

***Hierarchical Model*:** The hierarchical model represents **data** as a **single rooted 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 4.12).

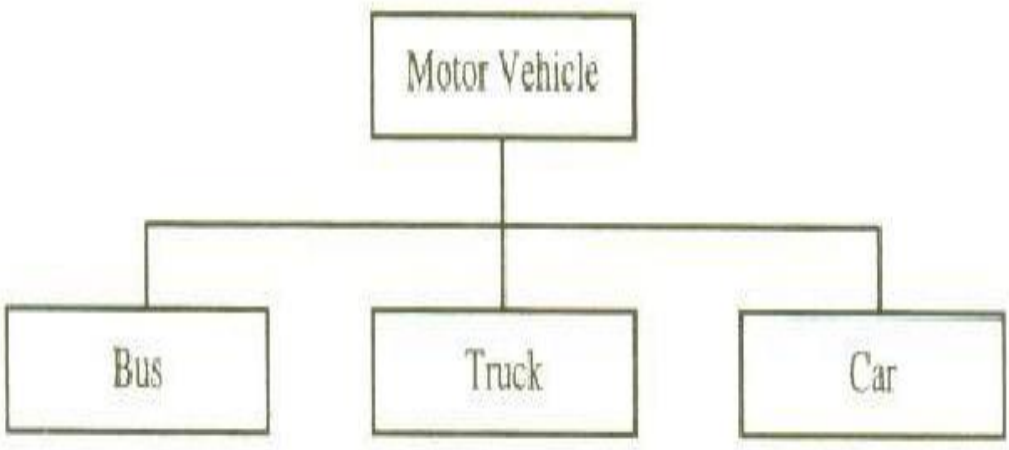


Fig. 4.12 A hierarchical Model

Network Model : A network database model is its' record can have more than one parent. For example, in Figure 4.13 , an Order contains data from the Soup and Customer nodes.

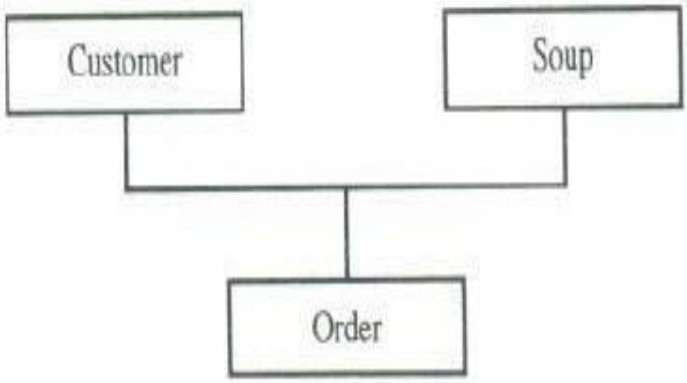


Fig 4.13: An order contains data from both customer and soup

Relational Model: 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 4.14 , Soup-ill and Cust-ill are foreign keys in the Order table.

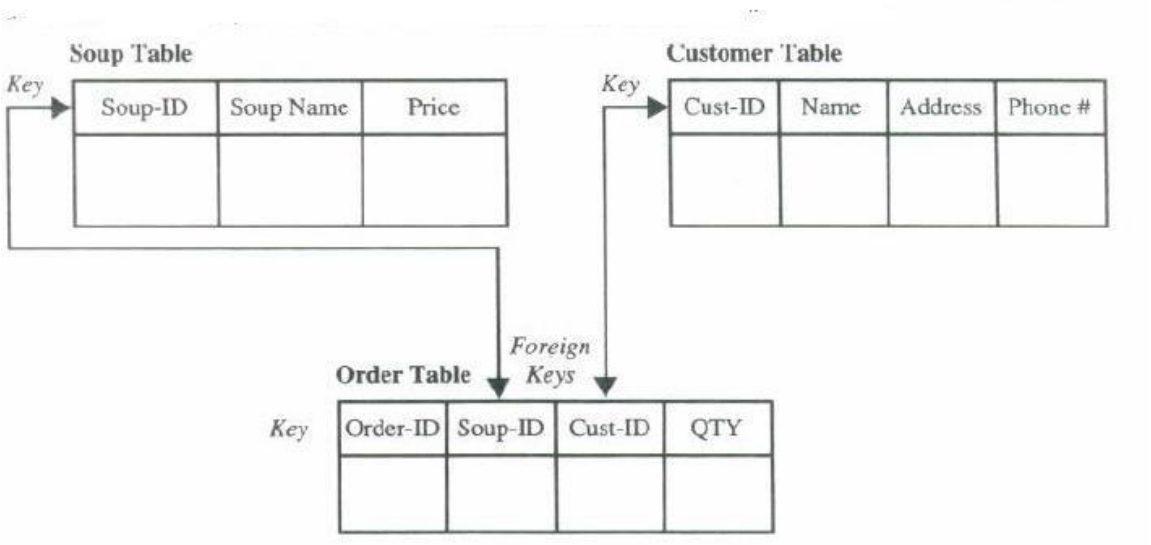


Fig 4.14 : The fig depicts primary and foreign key in a relation database.

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.

1. S**tructured query language (SQL**) - 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.
2. **To extend the host programming language** with database related 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 :

A **data Manipulation Language (DML)** is the language that allows users to access and manipulate(such as, create, save, or destroy) data organization.
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.

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.

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*** **i**f 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 maintains the database in a consistent state.

Concurrency Policy

*. 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.

*. 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**.

*. 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.

DISTRIBUTED DATABABSES AND CLIENT-SERVER COMPUTING

Many modern databases are **distributed databases**, which imply that portions of the database reside on different nodes (computers) and disk drives in the network. Usually, each portion of the database is managed by a server, a process responsible for controlling access and retrieval of data from the database portion.

The server dispenses information to client applications and makes queries or data requests to these client applications or other servers.

Clients generally reside on nodes in the network other than those on which the servers execute. However, both can reside on the same node, too.

What Is Client-Server Computing?

*. 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.

*.Client-server computing extends this theory a step further by recognizing that all those modules need not be executed within the same memory space or even on the same machine. With this architecture, **the calling module becomes the "client" (that which requests a service) and the called module becomes the "server" (that which provides the service).**

*. Another **important component** of client-server computing is **connectivity,** which allows applications **to communicate transparently** with other programs or processes, regardless of their locations. The **key element** of connectivity is the **Network Operating System (NOS),** also known as ***middleware***. The NOS provides services such as routing, distribution, messages, filing and printing, and network management.

*. The client is a process (program) that sends a message to a server process (program) requesting that the server perform a task (service).

*. Client programs usually manage the user interface portion of the application, validate data entered by the user, dispatch requests to server programs, and sometimes execute business logic.

*. The business layer contains all the objects that represent the business (real objects), such as Order, Customer, Lineitem, Inventory.

*. The client-based process is the front-end of the application, which the user sees and interacts with.

*. The client process contains solution-specific logic and provides the interface between the user and the rest if the application system. It also manages the local resources with which the user interacts, such as the monitor, keyboard, workstation, CPU, and peripherals.

*. A key component of a client workstation is the graphical user interface (GUI), which normally is a part of the operating system (i.e., the Windows manager). It is responsible for detecting user actions, managing the Windows on the display, and displaying the data in the Windows.

*. A server process (program) fulfills the client request by performing the task requested.

*. Server programs generally receive requests from client programs, execute database retrieval and updates, manage data integrity, and dispatch responses to client requests.

*. Sometimes, server programs execute common or complex business logic. The server-based process “may” run on another machine on the network. This server could be the host operating system or network file server; the server then is provided both file system services and application services.

*. In some cases, another desktop machine provide the application services. Their server process acts as a software engine that manages shared resources such as databases, printers, communication links, or high-powered processors. The server process performs the back-end tasks that are common to similar applications.

*. The server can take different forms. The **simplest form of server is a file server**.

*. With a file server, the client passes requests for files or file records over a network to the file server. This form of data service requires large bandwidth (the range of data that can be sent over a given medium simultaneously) and can considerably slow down a network with many users.

*. Traditional LAN computing allows users to share resources, such as data files and peripheral devices.

*. More **advanced forms of servers are database servers**, transaction servers, application servers, and more recently object servers.

*. With database servers, clients pass SQL requests as message to the server and the results of the query are returned over the network. Both the code that process the SQL request and the data reside on the server, allowing it to use its own processing power to find the requested data. This is in contrast to the file server, which requires passing all the records back to the client and then letting the client find its own data.

*. With transaction servers, clients invoke remote procedures that reside on servers, which also contain an SQL database engine. The server has procedural statements to execute a group of SQL statements (transactions), which either all succeed or fail as a unit.

*. The applications based on transaction servers, handled by on-line transaction processing (OLTP) tend to be mission-critical applications that always require a 1-3 second response time and tight control over the security and the integrity of the database. The communication overhead of a single request and reply (as opposed to multiple SQL statements in database servers).

*. Application servers are not necessarily database centered but are used to serve user needs, such as downloading capabilities from Dow Jones or regulating an electronic mail process. Basing resources on a server allows users to share data, while security and management services, also based on the server, ensure data integrity and security.

* The logical extension of this is to have clients and servers running on the appropriate hardware and software platforms for their functions. For example, database management system servers should run on platforms especially with special elements for managing files.

*. In a ***two-tier*** **architecture**, a client talks directly to a server, with no intervening server. This type of architecture typically is used in small environments with less than 50 users. A common error in client-server development is to prepare a prototype of an application in a small two-tier environment then scale up by simply adding more users to the server. This approach usually will result in an ineffective system, as the server becomes overwhelmed. To properly scale up to hundreds or thousands of users, it usually is necessary to move to three-tier architecture.

*. A ***three-tier*** architecture introduces a server (application or Web server) between the client and the server. The role of the application or Web server is manifold. It can provide translation services (as in adapting a legacy application on a mainframe to a client-server environment), meeting services (as in acting as a transaction monitor to limit the number of simultaneous requests to a given server), or intelligent agent services (as in mapping a request to a number of different servers, collating the results, and returning a single response to the client).

Ravi Kalakota describes the basic characteristics of client-server architectures as follows:

1. A combination of a client or front-end portion that interacts with the user and a server or back end portion that interacts with the shared resource. The client process contains solution-specific logic and provides the interface between the user and the rest of the application system. The server process acts as a software engine that manages shared resources such as databases, printer, modems, or high-powered processors.
2. The front-end task and back-end task have fundamentally different requirements for computing resources such as processor speeds, memory, disk speeds and capacities, and input/output devices.
3. The environment is typically heterogeneous and multivendor. The hardware platform and operating system of client and server are not usually the same. Client

and server processes communicate through a well-defines set of standard application program interfaces(APIs)

4. An important characteristic of client-server systems is scalability. They can be scaled horizontally or vertically. Horizontal scaling means adding or removing client workstations with only a slight performance impact. Vertical scaling means migrating to a larger and faster server machine or multi-servers.

Client-server and distributed computing have arisen because of a change in business needs. Unfortunately, most business have existing systems, based on older technology, that must be incorporated into the new, integrated environment; that is, mainframes with a great deal of legacy (older application) software.

A typical client-server application consists of the following components:

1. User interface. This major component of the client-server application interacts with users, screens, windows, Windows managements, keyboard, and mouse handling.

2. Business processing. This part of the application uses the user interface data to perform business tasks. In this book, we look at how to develop this component by utilizing an object-oriented technology.

3. Database Processing. This part of the application code manipulates data within the application. The data are managed by a database management system, object oriented or not. Data manipulation language , such as SQL or a dialect of SQL (perhaps, an object – oriented query language). Ideally , the DBMS processing is transparent to the business processing layer of the application.

The development and implementation of client-server computing is more complex, more difficult, and more expensive that traditional, single process applications. However, utilizing an object-oriented methodology, we can manage the complexity of client-server applications.

DISTRIBUTED AND COOPERATIVE PROCESSING

The distributed processing means **distribution of applications and business logic across multiple processing platforms.**

*. Distributed processing implies that processing will occur on more than one processor in order for a transaction to be completed.

*. In other words, processing is distributed across two or more machines, where each process performs part of an application in a sequence. These processes may not run at the same time. For example, in processing an order from a client, the client information may process at one machine and the account information then may process on a different machine.

*. Often, the object used in a distributed processing environment also is distributed across platforms.

*. Cooperative processing is computing that requires two or more distinct processors to complete a single transaction.

*. Cooperative processing is related to both distributed and client-server processing. Cooperative is a form of distributed computing in which two or more distinct processes are required to complete a single business transaction.

*. Usually, these programs interact and execute concurrently on different processors.

*. Cooperative processing also can be considered to be a style of distributed processing, if communication between processors is performed through a message-passing architecture.

DISTRIBUTED OBJECTS COMPUTING : THE NEXT GENERATION OF CLIENT-SERVER COMPUTING

Software technology is in the midst of a major computational shift toward distributed object computing (DOC). Distributed computing is poised for a second generation client-server era. In this new client-server model, servers are plentiful instead of scarce (because every client can be a server) and proximity no longer matters.

In the **first generation client-server era**, which still is very much is **progress, SQL database, transaction processing (TP) monitors, and groupware have begun to displace file servers as client-server application models.**

In the **new client-server era, distributed object technology is expected to dominate other client-server application models.**

Distributed object computing promises the most flexible client-server systems, because it utilized reusable software components that can roam anywhere on networks, run on different platforms, communicate with legacy applications by means of object wrappers, and manage themselves and the resources they control. Objects can help break monolithic applications into more manageable components that coexist on the expanded bus.

Distributed objects are reusable software components that can be distributed and accessed by users across the network. These objects can be assembled into distributed applications. Distributed object computing introduces a higher level of abstraction into the world of distributed applications. Applications no longer need to know which server process performs a given function. All information about the function is hidden inside the encapsulated object. A message requesting an operation is sent to the object, and the appropriate method is invoked.

Distributed object computing will be the key part of tomorrow's information systems. **DOC resulted from the need to integrate mission-critical applications and data residing on systems that are geographically remote, sometimes from users and often from each other, and running on many different hardware platforms**. Furthermore, the information systems must link applications developed in different languages, use data from object and relational databases and from mainframe systems, and he optimized from use across the Internet and thorough departmental intranets. Historically, business have had to integrate applications and data by writing custom

interfaces between systems, forcing developers to spend their time building and maintaining an infrastructure rather than adding new business functionality.

Distributed object technology has been tied to standards from the early stage. Since 1989, the **Object Management Group (OMG**), with over 500 member companies, has been specifying the architecture for an open software bus on which object components written by different vendors can operate across networks and operating systems. The OMG and the object bus are well on their way to becoming the universal client-server middleware.

Currently, there are several competing DOC standards, including the object Management Group's COBRA, OpenDoc, standards, and Microsoft's ActiveX/DCOM. Although DOC technology offers unprecedented computing power, few organizations have been able to harness it as yet. The main reasons commonly cited for slow adoption of DOC include closed legacy architecture, incompatible protocols, inadequate network bandwidths, and security issues. In the next subsections, we look at Microsoft's DCOM and OMG's CORBA.

Common Object Request Broker Architecture

Many Organizations are now adopting the **object Management Group's Common object request broker architecture (CORBA), a standard proposed as a means to integrate distributed, heterogeneous business applications and data.**

> **The CORBA interface definition language (IDL) allows developers to specify language-neutral, object-oriented interfaces for application and system components.**
>
> **IDL Definitions** are stored in an interface repository, a sort of phone book that offers object interfaces and services. For distributed enterprise computing, the interface repository is central to communication among objects located on different systems.

CORBA object request brokers (ORBs) implement a communication channel though which applications can access object interfaces and request data and services. The CORBA common object environment (COE) provides system-level services such as life cycle management for objects accessed through CORBA, event notification between objects, and transaction and concurrency control.

Microsoft's ActiveX/DCOM

Microsoft's **component object model** (COM) and its successor the distributed component object model (DCOM) are Microsoft's alternatives to OMG's distributed object architecture CORBA.

Microsoft and the OMG are competitors, and few can say for sure which technology will win the challenge.

Although CORBA benefits from wide industry support, DCOM is supported mostly by one enterprise, Microsoft.

However, Microsoft is no small business concern and hold firmly a huge part of the microcomputer population, so DCOM has appeared a very serious competitor to

CORBA. DCOM was bundled with Windows NT 4.0 and there is a good chance to see DCOM in all forthcoming Microsoft products.

The ***distributed component object model,*** **Microsoft's alternative to OMG's CORBA, is an Internet and component strategy where ActiveX (formerly known as object linking and embedding, or OLE) plays the role DCOM object. DCOM also is backed by a very efficient Web browser, the Microsoft Internet Explorer.**

OB.JECT.ORIENTED DATABASE MANAGEMENT SYSTEMS: THE PURE WORLD

The ***object-oriented database management system*** is a marriage of object oriented programming and database technology (see Figure 17) 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**.

1. The defined operations apply universally and are not dependent on the particular database application running at the moment.
2. 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.

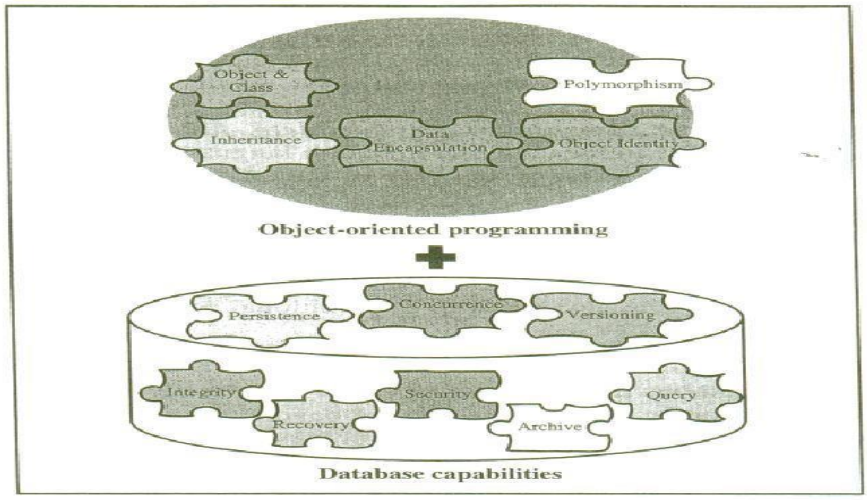


Fig 4.17 The object-oriented database management system is a marriage of object-oriented programming and database technology.

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 existenceindependent 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*** 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 modem 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.

4.17.1 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.

	Traditional / Relational database	**Obj-Oriented data base**
1	Records play passive role.	These databases are derived from the object's ability to interact with other objects itself. The objects are an "active" component in an o-o database,
2.	Relational Database systems do not explicitly provide inheritance of attributes and methods.	It represent relationships explicitly, support both navigational and associative access to information. So data access performance is improved.
3.	This is purely value-oriented approach.	They allow representation and storage of data in the form of objects. Each object has its own identity, or *object-ID*. The object identity is independent of the state of the object.

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.

OBJECT – RELATIONAL SYSTEMS: THE PRACTIVAL WORLD.

Many applications increasingly are developed in an o-o-programming technology, chances are good that the data those applications need to access live in a very different universe – a relational database. In such environment, the introduction of o-o-development creates a fundamental mismatch between the programming model (objects) and the way in which existing data are stored.

*. To resolve the mismatch, a mapping tool between the application objects and the relational data must be established.

*. Creating an object model from the existing relational database layout (schema) is referred to as Reverse engineering.

*. Creating a relational schema from an existing object model is referred to as forward engineering.

*. In Practice, over the life cycle of an application, forward and reverse engineering need to be combined in an iterative process to maintain the relationship between the object and relational data representations.

*. Tools that can be used to establish the object-relational mapping processes have begun to emerge. The main process in relational and object integration is defining the relationships between the table structures in the relational database with classes in the object model.

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):

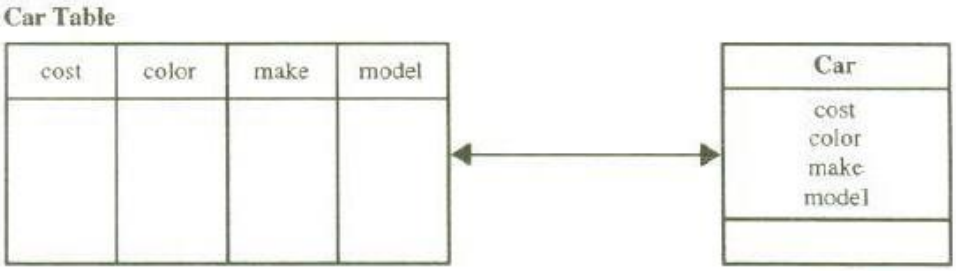
1. Table-class mapping.
2. Table-multiple classes mapping.
3. Table-inherited classes mapping.
4. Tables-inherited classes mapping.

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.

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 4.18

Table-class mapping. Each row in the table represents an object instance and each column in the table corresponds to an object attribute.



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).

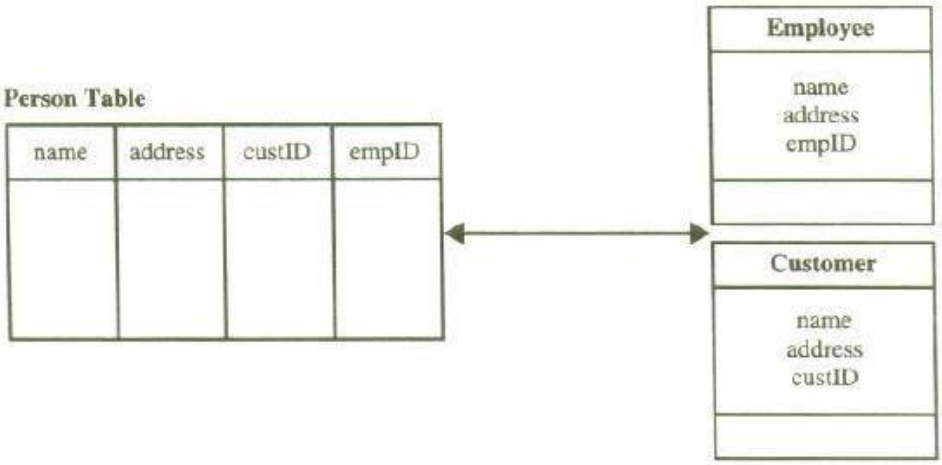


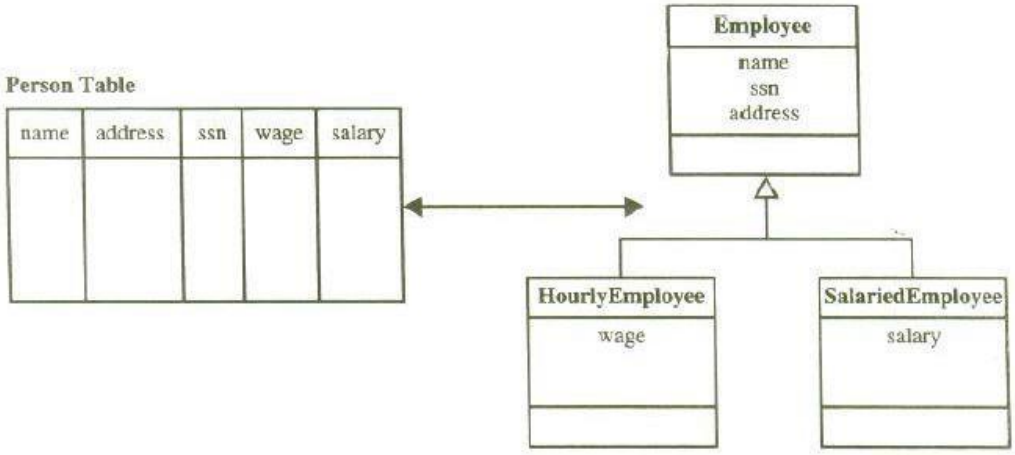
Fig 4.19 Table-multiple classes mapping. The custiD column provides the discriminant. If the value for custlD is null, an Employee instance is created at run time; otherwise, a Customer instance is created.

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.

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 4.20 , 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.



4.20 : Table –multiple classes mapping

Table-inherited Classes mapping

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 4.21 , 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.

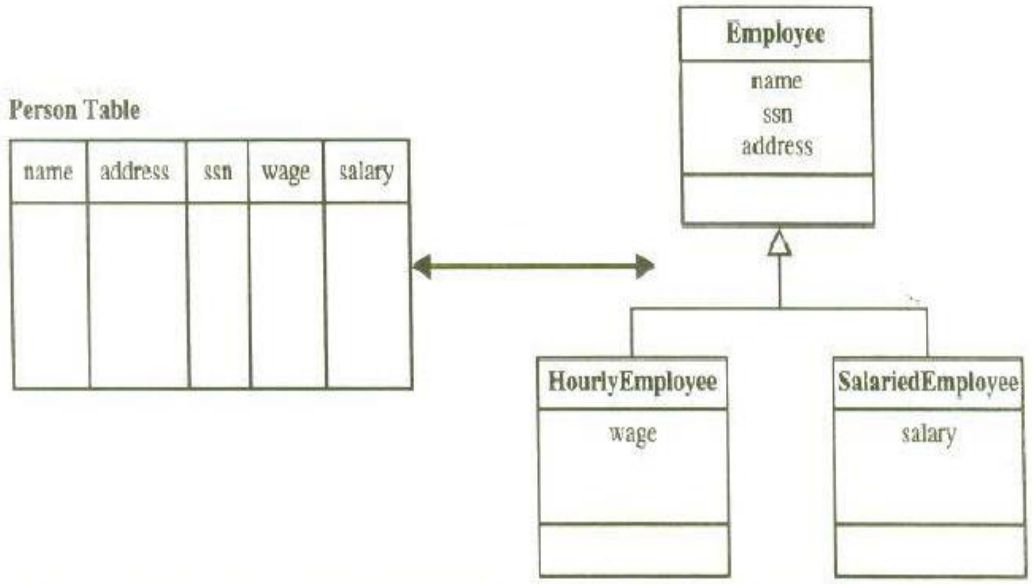


Fig 4.21: Table Inherited classes mapping

Table-inherited Classes mapping.

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 relationship.

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.

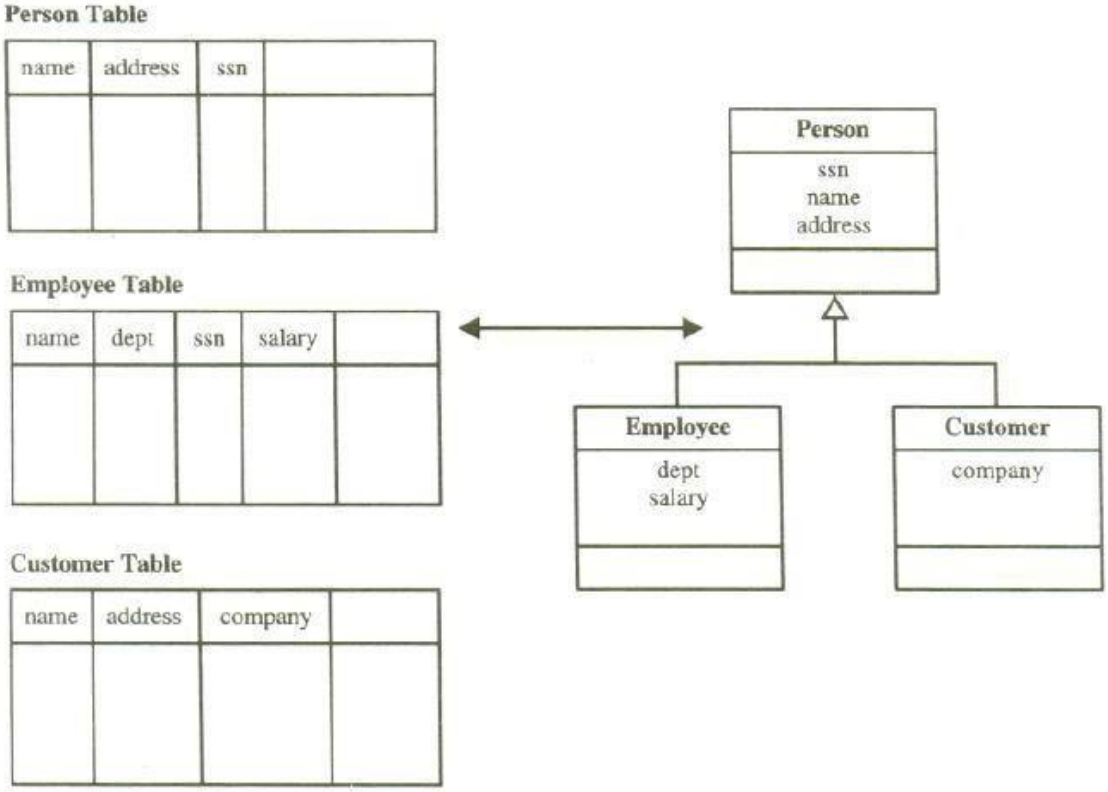


Fig 4.22 *Tables-inherited classes mapping. Instances of Person are mapped directly from the Person table. However, instances of Employee can be created only for the rows in the Employee table (the joining of the Employee and Person tables on the ssn key).The ssn is used both as a primary key on the 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.*

Figure 4.22 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.

MULTIDATABASE SYSTEMS

- A different approach for integration object-oriented applications with relational data environments is multidatabase systems or heterogeneous database systems, which facilitate the integration of heterogeneous databases and other information sources.
- Heterogeneous information systems facilitate the integration of heterogeneous information sources, where they can be structured (having regular schema), semi-structured and sometimes even unstructured. Some heterogeneous information systems are constructed on a global schema over several databases. So users can have the benefits of a database with a schema to access data stored in different databases and cross database functionality. Such heterogeneous information systems are referred to as federated multidatabase systems**.**

Federated Multi Data Base

Federated multidatabase systems provide uniform access to data stored in multiple databases that involve several different data models. A multidatabase system (MDBS) is a database system that resides unobtrusively on top of, existing relational and object databases and file systems (local database systems) and presents a single database illusion to its users. The MDBS maintains a single global database schema and local database systems maintain all user data. The schematic differences among local databases are handled by neutralization (homogenization), the process of consolidating the local schemata**.**

MultiDatabase Systems (MDBS)

- The MDBS translates the global queries and updates for dispatch to the appropriate local database system for actual processing, merges the results from them and generates the final result for the user. MDBS coordinates the committing and aborting of global transactions by the local database systems that processed them to maintain the consistency of the data within the local databases. An MDBS controls multiple gateways (or drivers). It manages local databases through gateways, one gateway for each local database.

To summarize the distinctive characteristic of MDBS

***.** Automatic generation of a unified global database schema from local databases, in addition to schema capturing and mapping for local databases.

*. Provision of cross-database functionality by using unified schemata

*. Integration of heterogeneous database systems with multiple databases.

*. Integration of data types other than relational data through the use of such tools as driver generators.

*. Provision of a uniform but diverse set of interfaces to access and manipulate data stored in local databases**.**

OPEN DATABASE CONNECTIVITY : MULTIDATABASE APPLICATION PROGRAMMING INTERFACES

- Open Data Base Connective (ODBC) is an application programming interface that provides solutions to the multidatabase programming problem. It provides a vendor-neutral mechanism for independently accessing multiple database hosts.
- ODBC and other APIs provide standard database access through a commonclient-side interface. It avoids the burden of learning multiple database APIs. Here one can store data for various applications or data from different sources in any database and transparently access or combing the data on an as needed basis. Details of back-end data structure are hidden from the user**.**

ODBC is similar to Windows print model, where the application developer writes to a generic printer interface and a loadable driver maps that logic to hardware-specific commands. This approach virtualizes the target printer or DBMS because the person with the specialized knowledge to make the application logic work with the printer or database is the driver developer and not the application programmer. The application interacts with the ODBC driver manager, which sends the application calls (such as SQL statements) to the database. The driver manager loads and unloads drivers, perform status checks and manages multiple connections between applications and data sources

Refer Text Book –page no. 262 – 263

Designing Access Layer Classes

The main idea behind creating an access layer is to create a set of classes that know how to communicate with the place(s) where the data actually reside. Regardless of where the data reside whether it be a file, relational database, mainframe, Internet, DCOM or via ORB, the access classes must be able to translate any data-related requests from the business layer into the appropriate protocol for data access. These classes also must be able to translate the data retrieved back into the appropriate business objects. The access layer‘s main responsibility is to provide a link between business or view objects and data storage. Three-layer architecture is similar to 3-tier architecture. The view layer corresponds to the client tier, the business layer to the application server tier and the access layer performs two major tasks:

- Translate the request: The access layer must be able to translate any data related requests from the business layer into the appropriate protocol for data access.

- Translate the results: The access layer also must be able to translate the data retrieved back into the appropriate business objects and pass those objects back into the business layer.

- Here design is tied to any base engine or distributed object technology such as CORBA or DCOM. Here we can switch easily from one database to another with no major changes to the user interface or business layer objects. All we need to change are the access classes‘ methods.

• Unlike object oriented DBMS systems, the persistent object stores do not support query or interactive user interface facilities.

• Controlling concurrent access by users, providing ad-hoc query capability and allowing independent control over the physical location of data are not possible with persistent objects.

• The access layer (AL), which is a key part of every n-tier system, is mainly consist of a simple set of code that does basic interactions with the database or any other storage device. These functionalities are often referred to as CRUD (Create, Retrieve, Update, and Delete).

• The data access layer need to be generic, simple, quick and efficient as much as possible. It should not include complex application/ business logics.

• I have seen systems with lengthy, complex store procedures (SP), which run through several cases before doing a simple retrieval. They contain not only most part of the business logic, but application logic and user interface logic as well. If SP is getting longer and complicated, then it is a good indication that you are burring your business logic inside the data access layer.

Refer Text Book - page no – 264 -268.

VIEW LAYER : DESIGNING INTERFACE OBJECTS

DESIGING VIEW LAYER CLASSES

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:

1. ***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. 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.

2. ***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.

3. ***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 can fying 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.

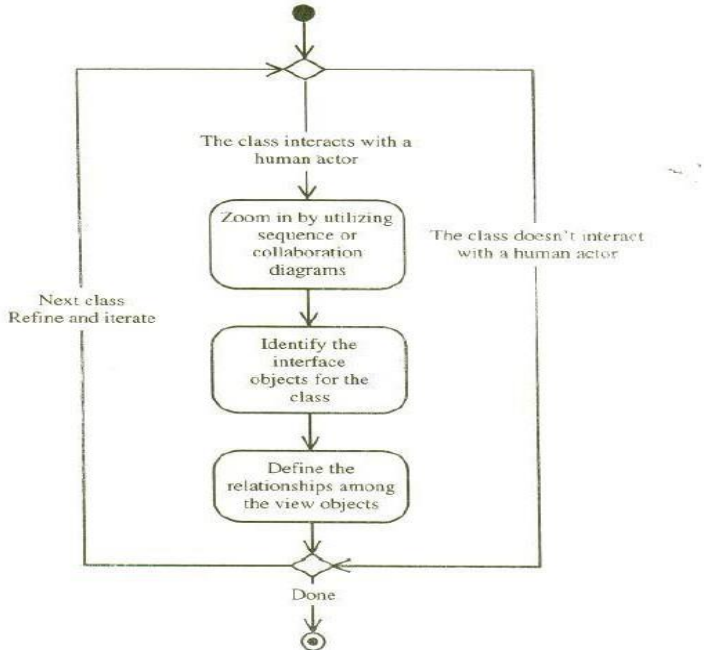


Fig 4.23 The macro level design process.

Questions

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Compare Coupling and Cohesion.	Analysis	BTL 4
2.	List out the types of Database models.	Remember	BTL 1
3.	Define Corollaries.	Remember	BTL 1
4.	List out the types of attributes.	Remember	BTL 1
5.	Define DDL and DML.	Remember	BTL 1
6.	Define Axiom.	Remember	BTL 1
7.	What is meant by Coupling?	Remember	BTL 1
8.	What is OCL?	Remember	BTL 1
9.	Analyze the purpose of DBMS.	Analysis	BTL 4
10.	List out the various Visibility modes.	Remember	BTL 1
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Elaborate Object Oriented Design process in detail.	Remember	BTL 1
2.	Explain the steps involved in designing the access layer classes.	Remember	BTL 1
3.	Explain the activities involved in the macro and micro level processes while designing the view layer classes.	Remember	BTL 1
4.	Explain about Corollaries in detail.	Remember	BTL 1
5.	Discuss in detail about (i)Client Server computing (ii)Distributed Database	Understand	BTL 2
6.	(i)Compare Traditional Database and Object Oriented Database. (ii)Analyze the Characteristics of OOD	Analysis	BTL 4
7.	Explain OODBMS in detail.	Remember	BTL 1

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – V - OBJECT ORIENTED ANALYSIS AND SYSTEM ENGINEERING - SCSA1401

UNIT 5

SOFTWARE QUALITY

Software Quality Assurance- Impact of Object Orientation on Testing - Develop Test Cases and Test Plans – System Usability and Measuring User Satisfaction: Usability Testing - User Satisfaction Testing.

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

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."

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 prograrnming-"debugging" still remains a most difficult, confused and unsatisfactory operation. The chief impact of this state of affairs is psychological.

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.

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 .

TESTING STRATEGIES

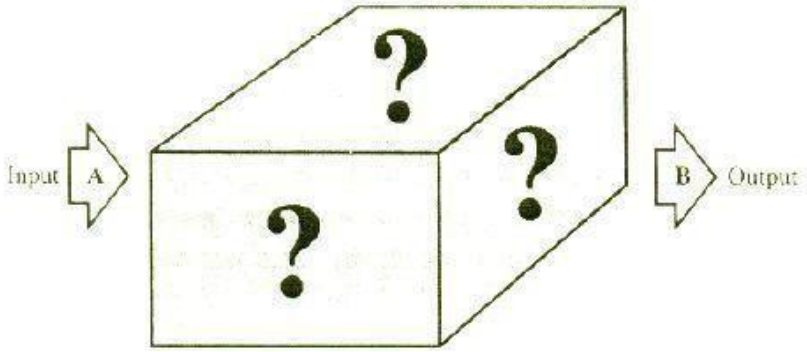
The extent of testing a system is controlled by many factors, such as the risks involved, limitations on resources, and deadlines. In light of these issues, we must deploy a testing strategy that does the "best" job of finding defects in a product within the given constraints. There are many testing 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).

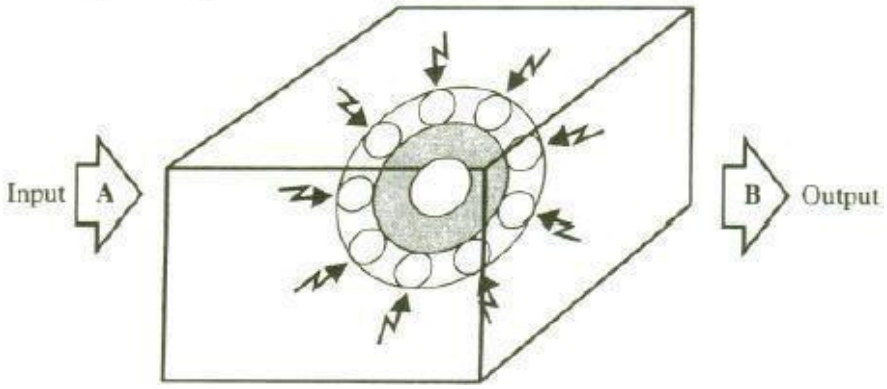
In white box testing, you are looking for bugs that have a low probability of execution, have been carelessly implemented, or were overlooked previously .

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 testingis 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 codebefore it is used by another.Bottom-up testing leads to integration testing, which leads to systems testing.

TEST CASES

To have a comprehensive testing scheme, the test must cover all methods or a goodmajority 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 theoutput 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 describethe usage test cases.

After all, tests always should be designed from specifications and not by looking at theproduct!

Myers describes the objective of testing as follows.

Testing is the process of executing a program with the intent of finding errors. A good testcase is the one that has a high probability of detecting an as-yet undiscovered error. A successfultest 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. .

.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 specifythe 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).

TEST PLANS

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 howto 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 realizeit.
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.

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 detennined what types of testing are necessary (such as black box, whitebox, 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 shouh l, De 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 outof- date that you need to rewrite the whole plan. Keep configuration infonnation on your plan, too. Notes about who made which updates and when can be very helpful down the road

MYERS'S DEBUGGING PRINCIPLES

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 tomodel 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.)

SYSTEM USABLILITY AND USER

SATISFACTION 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.

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

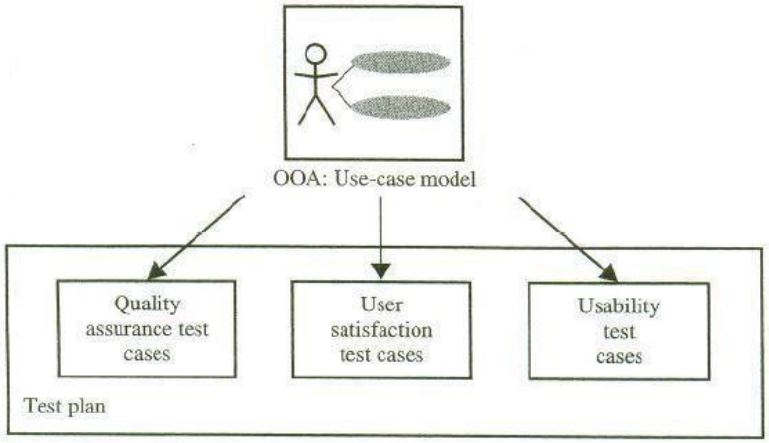
effectiveness, efficiency, and satisfaction. How do we measure usability?

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.

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 theirimpressions 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 usersif you target only a single skill level of users, such as novices or intermediate level users. .

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 toevaluate 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.

USER SATISFCATION TEST

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.

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 takeit.

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 playa 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 and10, 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."

A TOOL FOR ANALYZING USER SATISFACTION: THE USER SATISFACTION TESTTEMPLATE

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 mustallow 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
Ease of use	4	7							
Functionality	5	4							
Cost	1	6							
Realiablity	3	4							

Fig: User Satisfaction test for a Customer Tracing System

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, interviewsrecorded, 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.

CASE STUDY: DEVELOPING USABILITY TEST PLANS AND TEST CASES FOR THEVIANET 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.

DEVELOP TEST OBJECTIVES

The first step is to develop objectives for the test plan. Generally, test objectives are based on therequirements, 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 tooperate the ATM within 90 seconds.

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 usedthe 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 satisfactiontest to measure their level of satisfaction with the kiosk.

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 011 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?

		10	9	8	7	6	5	4	3	2	1	
Is easy to operate:	Very Easy											Very Hard
Buttons are right size and easily can be located:	Very Appropriate											Not Appropriate
Is efficient to use:	Very Efficient											Very Inefficient
Is fun to use:	Fun											No Fun
Is visually pleasing:	Very Pleasing											Not Pleasing
Provides easy recovery from errors:	Very Easy Recovery											Not at All

Comments:

☐ I have more to say; I would like to see you.

Questions

Part-A			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Compare Debugging and Testing.	Analysis	BTL 4
2.	Why quality assurance is needed?	Evaluate	BTL 5
3.	Define Blackbox Testing.	Remember	BTL 1
4.	List out the Testing strategies.	Remember	BTL 1
5.	Justify the importance of usability testing.	Evaluate	BTL 5
6.	Define Test case.	Remember	BTL 1
7.	Discuss scenario-based testing?	Evaluate	BTL 5
8.	Define Test Plan.	Remember	BTL 1
9.	What is meant by SQA?	Remember	BTL 1
10.	Compare Alpha and Beta testing.	Analysis	BTL 4
Part-B			
Q.No	**Questions**	**Competence**	**BT Level**
1.	Explain Myer's debugging principles	Analyze	BTL4
2.	Describe the different types of testing strategies	Understand	BTL 2
3.	Explain user satisfaction test with example.	Analyze	BTL4
4.	Explain in detail about (i)Blackbox Testing (ii)Whitebox Testing	Remember	BTL 1
5.	(i) Analyze the guidelines for developing quality assurance Test cases described by Freedman and Thomas. (ii)Explain about Software Quality Assurance.	Analyze	BTL4
6.	Describe the following (i)Debugging (ii)Guidelines for developing Test Plans	Understand	BTL 2
7.	Develop usability test plans and test cases for vianet bank ATM system.	Create	BTL 6

References

[1] Ali Bahrami, “Object oriented systems development using the unified modelling language”, McGraw- Hill.

[2] Grady Booch, James Rumbaugh, and Ivar Jacobson,“The Unified Modeling Language User Guide”, 3rd Edition Addison Wesley.

[3] John Deacon, “Object Oriented Analysis and Design”, 1st Edition, Addison Wesley.

[4] Bernd Oestereich, “Developing Software with UML, Object - Oriented Analysis and Design in Practice”, AddisonWesley