MC5305 - OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT I INTRODUCTION 9

An overview – Object basics – Object state and properties – Behavior – Methods – Messages – Information hiding – Class hierarchy – Relationships – Associations – Aggregations- Identity – Dynamic binding – Persistence – Meta classes – Object oriented system development life cycle.

UNIT II METHODOLOGY AND UML 9

Introduction – Survey – Rumbaugh, Booch, Jacobson methods – Unified modelling language – Static and Dynamic models – Rational Rose Suite - UML diagrams – Static diagram: Class diagram – Use case diagrams – Behaviour Diagram : Interaction diagram – State chart diagram – Activity diagram - Implementation diagram: Component diagram – Deployment diagram – example - Design of online railway reservation system using UML diagrams - Dynamic modelling – Model organization – Extensibility

UNIT III OBJECT ORIENTED ANALYSIS 9

Identifying Use case – Business object analysis – Use case driven object oriented analysis – Use case model – Documentation – Classification – Identifying object, relationships, attributes, methods – Super-sub class – A part of relationships Identifying attributes and methods – Object responsibility – construction of class diagram for generalization, aggregation – example – vehicle class.

UNIT IV OBJECT ORIENTED DESIGN 9

Design process and benchmarking – Axioms – Corollaries – Designing classes – Class visibility – Refining attributes – Methods and protocols – Object storage and object interoperability – Databases – Object relational systems – Designing interface objects – Macro and Micro level processes – The purpose of a view layer interface-OOUI - MVC Architectural Pattern and Design – Designing the system.

UNIT V CASE TOOLS 9

Railway domain : Platform assignment system for the trains in a railway station - Academic domain : Student Marks Analysing System - ATM system - Stock maintenance - Quiz System - E-mail Client system - Cryptanalysis – Health Care Systems. Use Open source CASE Tools: StarUML/ UML Graph for the above case studies.

REFERENCES:

1. Ali Bahrami, “Object Oriented System Development”, McGraw Hill International Edition, 2008
2. Brahma Dathan, Sarnath Ramnath, “Object-Oriented Analysis, Design and Implementation”, Universities Press, 2010
3. Bernd Bruegge, Allen H. Dutoit, Object Oriented Software Engineering using UML, Patterns and Java, Pearson 2004
4. Craig Larman, Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development” , 3rd Edition, Pearson Education, 2005
5. Grady Booch, James Rumbaugh, Ivar Jacobson, “The Unified Modeling Language User Guide”, Addison Wesley Long man, 1999
6. Martin Fowler, “UML Distilled A Brief Guide to Standard Object Modeling Language”, 3rd Edition, Addison Wesley, 2003
7. Russ Miles, Kim Hamilton, “Learning UML 2.0”, O‟Reilly, 2008
8. http://staruml.sourceforge.net/docs/StarUML_5.0_Developer_Guide.pdf
9. http://www.spinellis.gr/umlgraph/doc/index.html

QUESTION BANK

PART – A - UNIT – I

1. What is an Object, state of the object? Give an example. (Dec2016, May 2017)

Object is a real world entity. Software is a collection of discrete objects that encapsulate their data and logic / functionalities to model real world objects. Each object has attributes (data) and method (function). Example, Human being, table, chair, computer, book.

- The state is the set of values that describes an object at a specific point in time.
- It is represented by state symbols and the transitions are represented by a time
- Such transitions are represented by arrows connecting the state symbols.
- A state is represented by a circle.

2. What is inheritance?

- Inheritance is a relationship between classes where one class is the parent of another class (derived) class.
- Inheritance allows classes to share and reuse behaviours and attributes.
- Reuse what we already have.
- Inheritance is the property of object-oriented systems that allows objects to be built from other objects.
- Inheritance allows explicitly taking advantage of the commonality of objects when constructing new classes.
- The parent class is also known as the base class or super class.

3. What is data abstraction?

- Using this data abstraction mechanism, it is possible to create new, higher-level, and more specialized data abstraction.
- You can work directly in the language, manipulating the kind of "objects" required by you or your application, without having to constantly struggle to bridge the gap between how to conceive of these objects and how to write the code to represent them.

4. Why do you use object orientation? List out the benefits of object orientation.

It adapts to :Changing requirements, Easier to maintain, More robust, Promote greater design, Code reuse, Higher level of abstraction, Seamless transition among different phases of software development, Encouragement of good programming techniques, Promotion of reusability

Benefits of object orientation.

- Faster development, Reusability ,Increased quality, Object technology, Raising the level of abstraction

5. What is encapsulation and information hiding?

Information hiding is a principle of hiding internal data and procedures of an object. By providing an interface to each object in such a way as to reveal as little as possible about inner workings .Encapsulation protects the data from corruption .

6. What is the difference between an object's methods and an object's attribute?

Attribute / Property	**Method**
State of an object	Implementation behavior of an object
Description of an object represented in a programming language	It is a function or procedure for a class
Represented by data type	Access the internal state of an object of that class to perform operations.
Ex**: Student – Object,** Reg-No – Attribute, Compute-Grade( ) - Method	

7. What is an association? Differentiate between association and cardinality.(May 2016)

Association represents the relationship between objects and classes. For example, in the statement "a pilot can fly planes", is an association. Association is bi-directional; that means they can be reversed in both directions.

Association	**Cardinality**
Represents relationships among objects	Represents mapping among entities
Can be represented in the form of multiplicity as 1 ..1, 1..*. *..1, *..*	Establishes 1 ..1, 1 .. M, M ..1, M ..M relationship between entities of common interest
A null association can be defined as 0 .. 1	It is not possible.

8. What is the difference between a method and a message?

Message	**Method**
Objects perform operations in response to messages.	Behavior of an object
	It is a function or procedure for a class
	Access the internal state of an object of that class to perform operations.
Ex: Compute-Grade( ) - Method	Call Compute-Grade( ) - Message

9. What is Polymorphism? Why is polymorphism useful? Give an example.

Polymorphism allows us to write generic, reusable code. The same operation may behave differently on different classes. Polymorphism as the relationship of objects of many different classes by some common class; thus, any of the objects designated by this name is able to respond to some common set of operations in a different way. Since no assumption is made about the classes of an object that represents a message, fewer dependencies are needed in the code and, therefore, maintenance is easier. **Example** : Compute-Payroll is a function, it behave differently in office-worker, manager, production-worker classes

10. What is a formal class or abstract class, meta class? (Dec 2016, May2017)

Formal or abstract classes have no instances but define the common behaviors that can be inherited by more specific classes. For example, shape is an abstract class which can be inherited by classes such as circle, line, triangle, polygon, rectangle, square, hexagon and pentagon.

A **Meta-Class** is a class' class. If a class is an object, then that object must have a class. Compilers provide an easyway to picture Meta-Classes. All objects are instances of a class. All classes are instances of a meta class. The Meta-Class can also provide services to application programs, such as returning a set of all methods, instances or parents for review

11. How does object-oriented development eliminate duplication?

Duplication occurs when using a procedural language, since there is no concept of hierarchy and inheriting behaviour. An object-oriented system eliminates duplicated effort by allowing classes to share and reuse behaviors.

12. Differentiate object oriented and object based technology.

Object Based Technology	**Object Oriented Technology**
Object-Based technology does not support inheritance and polymorphism.	Object-oriented technology may include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance
Ex: Ada 83 and Modula-2	Ex: JAVA

13. List the object oriented system development activities, various types of prototypes.

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

The various **types of prototypes** are

- Horizontal prototype , Vertical prototype, Analysis prototype, Domain prototype

14. How are objects identified in an object-oriented system? How are the classes organized in an object-oriented environment?

An object's identity comes into being when the object is created and continues to represent that object from then on. This identity never is confused with another object, even if the original object has been deleted. The identity name never changes even if all the properties of object change—it is independent of the object's state.

Object identity often is implemented through some kinds of object identifier or unique identifier.

An object-oriented system organizes classes into a subclass-super class hierarchy.

Different properties and behaviours are used as the basis for making distinctions between classes and subclasses. At the top of the class hierarchy are the most general classes and at the bottom are the most specific. A subclass inherits all the properties and methods defined in the super class.

15. What is the lifetime of an object and how can you extend the lifetime of an object?

Objects have a lifetime. They are explicitly created and can exist for a period of time that, traditionally, has been the duration of the process in which they were created. From a language perspective, this characteristic is called object persistence. An object can persist beyond application session boundaries, during which the object is stored in a file or a database form.

17. What is object modeling? Differentiate correspondence and correctness measures

Object modeling is a process by which the logical objects in the real world are represented by actual objects in the program. UML class diagram also referred to as object modeling, which is static analysis diagram and it is used to model objects and their relationships.

The following points to be explained the difference between the correspondence and correctness measures

Correspondence	**Correctness**
Correspondence measures how well the delivered system corresponds to the needs of the operational environment.	Correctness measures the consistency of the product requirements with respect to the design specification.

18. Why is reusability important? How does object oriented software development promote reusability?

Reusability is important because it provides increased reliability, reduced time and cost for development. OOSD promotes reusability by

- Information hiding
- Conformance to naming standard
- 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 product to be reused.

19. How would you identify attributes, methods? Define meta class

- The following questions are used to identify system's responsibilities.
- What information about an object should be kept track of?
- What services must a class provide?
- Answering the first question will help us to identify the attribute of the class.

20. What is the importance of layered Architecture in object Oriented Development?(15)

An approach to software development that allows us to create objects that represent tangible elements of th business. Independent of how they are represented to the user through an interface or physically stored in th database. Consists of view or user interface, business and access layers.

21. Describe the role of object behavior in Object Oriented Development. (May 2015)

Object behavior is described in methods/procedures. Method is a function/procedure that is defined for a class(i.e.,) it accesses the internal state of a.n object Behavior is a collection of methods that describes what an

object is capable of doing. The object called as the receiver is the one on which method operates. Methods encapsulate the behavior of the object. Methods provide interfaces to the object and hide any of the internal structure and state maintained by the object.

22. What is object oriented analysis and design goals?(May 2016)

Object-oriented analysis and design (OOAD) is a popular technical approach for analyzing, designing application, system, or business by applying the object-oriented paradigm and visual modeling throughout the development life cycles to foster better stakeholder communication and product quality.

UNIT – II

1. Mention the parts of Object Modeling Techniques in Rumbaugh methodology and its role for describing the system. / What is an object model? What are the other OMT models?

- The object model is going to be a description of the objects and their structures. It includes identity of the objects, relationship to other objects, attributes and their operations.
- It is represented using an object diagram, which consists of classes interconnected by association lines that establish relationships among the classes.

A dynamic model : presented by the state diagrams and event flow diagrams
A functional model : presented by data flow and constraints.
It is a representation of flow of data between different processes in a business.
The process is any function being performed, data flow shows the direction of data element movement.

2. Define an Abstract class. (Dec 2016)

- Abstract classes allow you to re-use common implementation details across a set of classes that share a common ancestor. Abstract classes are much like Interfaces in that they both provide a template of what methods should be within the inheriting class, but there are big differences: - Interfaces only define names/types of methods that need to exist in an inheriting class, while abs-classes can have complete default code of method and just the details may need to be over-ridden. abstract classes are more common in that they can provide default implementation of code right away.

3. What are the phases of OMT?

- **Analysis**: This results in the object and dynamic and functional models. The object model describes the structure objects in a system and is represented by means of an object diagram.
- **System Design**: The results are a structure of the basic architecture of the system.
- The object model describes the structure objects in a system
- It is represented by means of an object diagram.
- **Object Design**:
- It produces a detailed design consisting of objects, dynamic, functional models.
- **Implementation**: This activity produces reusable, extendible and robust comprehensive code.

4. What do you mean by a method, methodology and process? List the diagrams in Booch method.

- A **method** is an implementation of an objects behavior.
- A model is an abstract of a system constructed to understand the system prior to building or modifying it.
- **Methodology** is a set of methods, models and rules for developing systems based on any set of standards.
- The **process** is defined as any operation being performed.
- **Diagrams in Booch method**

1. Class diagrams 2. Object diagrams 3. State transition diagrams 4. Module diagrams
5. Process diagrams 6. Interaction diagrams.

5. List the steps in Macro development process. State the reason for using it.

- ***Conceptualization*** where the core requirements of the system are outlined
- ***Analysis*** and the development model which focuses on the class diagrams,
  Design or creation of the computer architecture to establish relationships between the' classes
- ***Evolution or implementation*** to produce a code
- ***Maintenance*** to add new requirements and to eliminate the bugs.
- Each macro development process has its own micro development process which aims at
  - Identifying class and objects
  - Identifying class and object semantics.

- Identifying class and object relationships
- Identifying class and object interfaces and implementation

6. What is the strength of Jacobson Methodology?

- The strength of the Jacobson methodology is that it entire life cycle and stress trace ability between the different phases, both forward and backward.
- This enables the reuse of analysis and design work, reducing development time and memory significantly.

7. Define Pattern. Why do you use it?

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

Reasons

- It solves a problem. Patterns capture solutions, not just abstract principles or strategies.
- It is proven concept. Patterns capture solutions with a track record, not theories or speculations.
- 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 describes modules, but describe deeper system structures and mechanisms.
- The pattern has a significant human component. All software servers' human comfort or quality of life.

8. Define Framework. List the template of a pattern.

A framework is a way of presenting a generic solution to a problem that can be applied to all levels in a development.

Pattern Template

Name	Probe / Also known as	Context	Forces
Solution	Examples	Resulting context	Rationale
Related Pattern	Known uses	Proven solution	

.

9. Write the differences between design patterns and frameworks.

Design Patterns	Frameworks
is instructive information that captures the essential structure and insight of a successfully family of proven solutions to a recurring problem that arises within certain context and system offered	a way of presenting a generic solution to a problem that can be applied to all levels in a development.
Pattern solves a problem, is a proven concept, describes relationships,	It represents a set of classes that make up a reusable design for a specific class of software.
It has significant human component.	It partitions the design into abstract classes and also defines relationships between them.
Design patterns are less specialized than frameworks	They emphasize design reuse over code reuse.
Only examples of a pattern can be included in code	Frameworks can be included in code
Design patterns have to be implemented each time they are used.	Frameworks can be written down in programming languages and executed, reused

	directly.
Design patterns cannot have frameworks	Frameworks contain design patterns

10. Differentiate static and dynamic model.

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.	Dynamic modeling can be viewed as a collection of procedures or behaviors that, taken together, reflect the behavior of a system over time.
used to specify structure of the objects that exist in the problem domain	Refers representing the object interactions during runtime.
These are expressed using class, object and USECASE diagrams	represented by sequence, activity, collaboration and state chart diagrams
Time Independent view of the system. E.g., Class has same number of students in a year.	Time dependent view of the system E.g. ATM can accept card only when it is in ready state.
includes Class Diagram, Object diagram Component Diagram, Deployment Diagram	Includes Use-Case Diagram, Interaction Diagram, State Diagram, Activity diagram
The classes structure and their relationships to each other frozen in time are examples of static models	For example, an order interacts with inventory to determine product availability

11. List the nine graphical 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
   - 4.1 Component diagram
   - 4.2 Deployment diagram

12. What is a Qualifier? What is meant by Multiplicity?

- 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.
- A qualifier can be 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 rectangle usually is smaller than the attached class rectangle.

- Multiplicity specifies the range of allowable associated classes.
- It is given for roles within associations, parts within compositions, repetitions, and other purpose.
- A multiplicity specification is as a text string comprising a period-separated sequence of integer intervals, where an interval represents a range of integers in this format: " lower bound.. upper bound".

13. What is a class diagram?

- 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, their internal structures and their relationships to other classes can be represented using this diagram.

14. What is a factory method pattern?

- The factory method pattern is a creational pattern which uses factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created.
- This is done by **creating objects via calling a factory method**—either specified in an **interface** and **implemented by child classes**, or **implemented in a base class** and optionally overridden by derived classes—rather than by calling a constructor.

15. Compare Abstract factory and Factory method.(May 2017)

Factory Method

- It contains one method to produce one type of product related to its type.
- It is better than a Simple factory because the type is deferred to a sub-class.

Abstract Factory

- It produces a family of types that are related.
- It is noticeably different than a factory method as it has more than one method of types it produces.

16. What are the types of mediator **pattern?**

- **Concrete Mediator:** Implements the Mediator interface and coordinates communication between Colleague objects.
- It is aware of all the Colleagues and their purpose with regards to inter communication.

Concrete Colleague - Communicates with other Colleagues through its Mediator.

17. What is meant by memento pattern?

The memento pattern is a software design pattern that provides the ability to restore an object to its previous state (undo via rollback).

The memento pattern is implemented with three objects: the originator, a care taker and a memento.

The originator is some object that has an internal state. he care taker is going to do something to the originator, but wants to be able to undo the change.

The care taker first asks the originator for a memento object.

Then it does whatever operation (or sequence of operations) it was going to do.

18. What is an use case? What are some of the ways that use cases can be described? (Dec 2016)

- Use Case is a scenario depicting a user system interaction.
- It begins with the user of the system issuing a sequence of interrelated events. Use cases are described as:
  - Nonformal text with no clear flow of events.
  - Text, easy to read but with a clear flow of events.
  - Formal style using pseudo code.
- A use case must contain
  - How and when it begins and ends.
  - The interaction between the use case and its actors, including when it occurs and what is exchanged.
  - How and when it needs the data stored in the system arid when it will store data in the system.
  - Exceptions to the flow of data in the system.
  - How and when the concepts of problem domain are handled.

19. List the various steps in micro development processes of a Booch Methodology. (May 2015)

1) Identify classes and objects.
2) Identify classes and object semantics.
3) Identify class and object relationships.
4) Identify class and object interface and implementations.

20. What are the advantages of modeling? (May 2015)

- **Rumbaugh's OMT –** well suited for describing object model / the static structure of the system
- **Jacobson OOSE –** well suited for producing user-driven analysis
- **Booch OOAD –** produces detailed OO design modules

21. What is the primary goal in the design of UML? (May 2016)

- Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
- Provide extensibility and specialization mechanisms to extend the core concepts.
- Be independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the OO tools market.

Support higher-level development concepts such as collaborations, frameworks, patterns and components.

22. List the phases of Unified process.(May 2016)

Inception., Elaboration (milestone), Construction (release), Transition (final production release)

23. What is UML? (May 2017)

The Unified Modeling Language (UML) is a standardized way of specifying, visualizing, and documenting the artifacts of an object-oriented software under development (among other things).

UNIT – III

1. Mention the steps for finding use cases.(May 2015)

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. Name the use cases. Describe the use cases briefly by applying terms with which the user is familiar. This makes the description less ambiguous.

2. List any two use cases for a "Libraray Management System"? (May 2017)

1.Borrow books
2.Supplier Return books
3.Circulation clerk
3. Get an inter liability loan
4. Do research
5. Read books / news paper
6. Purchase supplier

4. What is aggregation? (Dec 2016, May 2017)

A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.

5. Define actor.

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 , First class / Business class passenger

The actor is a key to find correct use cases. Actor carries out the use cases and shows the relationship between users and actors.

6.When will Uses association, Extends association used?

The extends association is used when an use case that is similar to another use case but does a bit more specialized; in essence, it is like a subclass. It results in inheritance.

- E.g., Borrow book, get a interlibrary loan. Checking out a book is the basic use case. This is the case that will represent what happens when all goes smoothly. The uses association occurs when you are describing you use cases and notice that some of them have sub flows in common.
- The use association allows extracting the common sub flow and making it a use case of its own.
- For example, checking a library card is common among the borrow books and return books.

7. What are the guidelines for selecting candidate classes from the relevant & Fuzzy categories of classes?

- **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.

- **Adjective 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.
- **Attribute classes**
  Tentative objects that are used only as values should be defined or restated as attributes and not as a class.
- **Irrelevant classes:** Each class must have e a purpose and every class should be clearly defined and necessary.

8. Define 80-20 Rule. **List the guidelines for selecting classes in an application**

- 80 percent of the work can be done with 20 percent of the documentation
- The 20 percent is easily accessible and the rest (80 percent) is available to those (few) who need to know.

Guidelines for selecting classes in an application

- Look for nouns and noun phrases in the 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.

9. How would you name classes? List the approaches for identifying classes. Mention the types of object relationships

- The class name should be singular.
- Choose the class name from standard vocabulary for the subject matter with which the clients or users are comfortable.
- The class name should reflect its intrinsic nature.
- Use readable names.
- Capitalize class names.
- Code should be consistent and easy to read.

Approaches for identifying classes.

  - Noun phrase approach
  - Common class pattern approach
  - Use-case driven approach
  - Sequence / Collaboration Modeling
  - CRC approach

- **Types of object relationships**
- Association
- Super-sub structure (generalization hierarchy)
- Aggregation and a-part-of structure

10. How do you separate actors from users?

Each use case should have only one main actor
Isolate users from actors
Isolate actors from other actors (separate the responsibilities of each actor)
Isolate use cases that have different imitating actors and slightly different behaviors.

11. List the guidelines for identifying super -sub relationships.

- **Top-Down**: Look for noun phrases composed of various adjectives in a class name.
- Avoid Successive refinement. Specialize only when the subclasses have significant behavior.
- **Bottom-Up**: look for classes with similar attributes and methods, in most cases. You can Group them by moving the common attributes and methods to an abstract class
- **Reusability**: Move attributes and behavior as high as possible in the hierarchy.
- Do not create very specialized classes at the top of the hierarchy.
- **Multiple Inheritance**: Avoid excessive use of multiple inheritances.
- 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.

12. What is Generalization?Is association different from a part of a relationship?

- Superclass -subclass relationships, also known as Generalization hierarchy allow objects to be built

from other objects.

- Such relationships allow us to take advantage of the commonality of objects when constructing new classes.
- Association and part of relation are similar except for the fact that it depends on problem domain and a part of relation is a special case of association.

13. What are the guidelines for naming the classes?

- The class should describe a single object.
- It should be singular form of noun.
- Use names that the users are comfortable.
- The class name should reflect its intrinsic nature.
- Use readable names and Capitalize class names (Ex. Loan-Window).

14. List the guidelines for identifying super -sub relationships?

1. Top-Down 2. Bottom-Up 3. Reusability 4. Multiple Inheritance

15. What is super-sub class?

- It represents inheritance relationships between related classes, and the class hierarchy determines the lines o inheritance between classes
- Allow objects to be built form other objects
- Also known as generalization hierarchy
- Parent class – also known as base or super class or ancestor

16. Differentiate Actors and Use cases.

Actors	**Use cases**
Actor represents the role a user plays with respect to the system.	A use case defines the interactions between external actors and the system A sequence of transactions in a system
An external system that needs information from a current system.	Yields results of measurable values to an individual actor of the system.
Actors can be the ones that get value from the use case	Provides a special flow of events.

17. Give the guidelines for identifying the tentative associations.

The following are the guidelines for identifying the tentative association: A dependency between two or more classes may be an association, Association corresponds to a verb or prepositional phrase, such as part of, next to, works for, or contained in, A reference from class to another is an association. Some association is implicit or taken from general knowledge.

18. What are the various types of use cases?

Use cases could be viewed as concrete or abstract. In abstract use case is not complete and has no imitation acto but is used by a concrete use case, which does interact with actors.

19. State the three steps in CRC process.

- **Class Responsibility Collaboration (CRC) model** is a collection of standard index cards that have been divided into three sections.
- A class represents a collection of similar objects, a responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities.

20. How to eliminate unnecessary associations? Give examples. (May 2015)

- **Implementation association**
- Defer implementations-specific associations to the design phase.
- **Ternary associations**
- Ternary or n-ary associations complicate the representation. so when possible restate ternary associations as binary associations.
- **Direct actions (or derived) associations**

- Direct associations can be defined in terms of other associations
- Since they are redundant avoid n these types of association. they are discovered by testing access paths to objects.

21. Write the guidelines for finding the use cases. (May 2015)

- 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.
- Name the use cases.
- Describe the use cases briefly by applying terms with which the user is familiar. This makes the description less ambiguous.

22. List the types of use case relationships.(May 2016)

Association between actor and use case..Generalization of an actor..Extend between two use cases. Include between two use cases. Generalization of a use case.

23. Name any four attributes of bank class. (May 2016)

Account Number, Account Name, Amount, Types of Account

UNIT – IV

1. Differentiate good design and bad design.

Good design	Bad design
1. Uncoupled design with less information	Complex coupling with information
2. Each class must have a single purpose.	No clearly defined single purpose.
3. There is a strong association between physical system & logical system	No strong association

2. What are the advantages of ORDBMS?(Dec 2016)

The main advantage of the object relational database is its reusability and sharing. Reusability helps to extend the DBMS server to manage standard functionality centrally, rather than have it coded in each application. It ensures large storage capacity. Supports for composite data types. Supports scalability and improved concurrency.

3. List the object oriented corollaries and axioms? (May 2016)

Corollaries

1. Uncoupled design with less information content.
2. Single purpose.
3. Large number of simpler classes.
4. Strong mapping
5. Standardization
6. Design with Inheritance

Axioms

Independence axiom

- Maintain the independence of components. Each component must satisfy a requirement without affecting other components

Information axiom.

- Minimize the information content of the design. Least complex code to minimize complexity

4. Define axioms and corollaries

Axioms : is a fundamental truth that always is observed to valid and for which there is no counter example or exception

Corollaries : is a proposition that follows from an axiom or another proposition that has been proven

5. What do you mean by multiple inheritance? How can u achieve multiple inheritances with single inheritance? List out the advantages, challenges in design with inheritance?

- Some object oriented systems permit a class to inherit its state and behaviors from more than one super class.
- This kind of inheritance is referred to as multiple inheritance.
- It inherits the most appropriate class and add an object of another class as an attribute or aggregation.
- Inherit the most appropriate class and add an object of another class as an attribute or aggregation. Use the instance of the class as an attribute.

Single inheritance: When a single new class inherits the behavior of a base (parent class).

- **Advantages of inheritance**:

- It is an abstraction mechanism which may be used to classify entities.
- It is a reuse mechanism at both the design and the programming level.
- The inheritance graph is a source of organizational knowledge about domain and systems.
- **challenges in design with inheritance**
- Multiple inheritances introduce ambiguity and program is more difficult to understand.
- We need to determine which behavior to get from which class, particularly when several ancestors have same method.

6. What are Design Axioms?(May 2017)

Design Axioms are fundamental truth that is always observed to be valid

- **Axiom 1. The independence axiom.**
- **Axiom 2. The information axiom.**

7. Write down the ways to design UI.

- Make the interface forgiving.
- Make the interface visual.
- Provide immediate feedback.
- Avoid modes
- Make the interface consistent
- Make the interface Beautiful,
- Give the interface look and feel appearance
- Design interface with clarity.

8. List the types of cohesion.

- .A method can carry only one function
  - Coincidental cohesion (worst)
  - Procedural cohesion
  - Functional cohesion (best)
  - Logical cohesion
  - Communicational cohesion
  - Temporal cohesion
  - Sequential cohesion

9. What are the benefits of the Access Layer Class? List the access layer tasks

- These classes provide easy migration to emerging distributed object technology.
- These classes should be able to address the modest needs of two-tier client-server architectures as well as the difficult demands of fine-grained, peer-to-peer distributed architectures.
- **Access layer 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

10. What are the steps for the view layer macro process?

The view layer macro process consists of two steps: For every class identified determine if the class interacts with a human actor. If so, perform the following; otherwise, move to next the next class, Iterate and refine until the user satisfies.

11. What are applications responsible for the view layer?

The view layer objects are responsible for two major aspects of the applications:

Input -responding to user interaction: The user interface must be designed to translate an action by user, such as clicking on a button or selecting from a menu, into appropriate response,

Output displaying or printing business objects: This layer must paint the best picture possible of business objects for the user. In one interface, this may mean entry fields and list boxes to display an order and its items

12. How can you avoid a sub class inheriting inappropriate behavior?

Make some of the class members of the super class as protected.

- Use polymorphism instead of inheritance.
- Define an abstract class and include the required behavior of the objects in the subclasses.

13. Enlist the different stages of detailed design in OOD.

- Apply design axioms to design classes, their attributes, methods, association structures and protocols

- Design the Access Layer
- Design the view layer

14. What do you mean by life time, categories of life time and object persistence?

- Object lifetimes can be short for local objects (called transient objects)
- Long for objects stored indefinitely in a database (called persistent objects).
- Transient results to the evaluation of expressions,
- Variables involved in procedure activation,
- Global variables and variables that are dynamically allocated,
- Data that exist between the executions of a program,
- Data that exist between the versions of a program,
- Data that outlive a program.
- Persistence refers to the ability of some objects to outlive the programs that created them.

15. What are the major activities in the process of designing view layer classes?

The process of designing view layer classes is divided into four major activities:

The macro level UI design process – identifying view layer objects

- Micro level UI design activities,
- Designing the view layer objects by applying design axioms and corollaries
- Prototyping the view layer interface
- Testing usability and user satisfaction
- Refining and iterating the design.

16. What do you mean by encapsulation leakage? What are public, private and protected protocols?

- When details about a class's internal implementation are disclosed through the interface, it is known as encapsulation leakage
- **Private Protocol**
  - Accessible only to the operations of that class.
- **Public Protocol**
  - Stated behavior used by all the classes.
- **Protected Protocol**
  - Methods and attributes used by that class and subclasses

17. What is the purpose of view layer interface? (May 2017)

- User interface can employ one or more windows. Each window should serve a clear, specific purpose.
- Windows commonly used are Forms windows, data entry windows, Dialog boxes, Application windows, split window and main window.

18. What do you mean by class visibility? Give example.(Dec 2016)

+ public visibility	accessibility to all classes
# protected visibility	accessibility to sub classes and operations of the class
- private visibility	accessibility only to operations of the class

Bank client class	**Adding visibility, implementation**
Firstname	#firstname
Lastname	#lastname
Pinno	#pinno #card no #account. Account(instance collection)

Where	
+	Public visibility
-	Private visibility
#	Protected visibility

19. What is MVC?

- Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces.
- It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

20.What is meant by coupling? Describe the degree of coupling. (May 2015)

- Coupling denotes the measure of strength of association established by a connection from one object to another.
- Cohesion is the interaction between software components or objects.

Highly cohesive element can lower coupling because only minimal information is passed between components.

21. Give brief description about the basic types of attributes. (May 2015)

- Single valued attributes, Multi valued attributes, Reference to another object / instance connection.
- Visibility name : type-expression= initial-value
- + public visibility (accessibility to all classes),
- # protected visibility (accessibility to sub classes and operations of the class),
- - private visibility (accessibility only to operations of the class)

Bank client class	**Adding visibility, implementation**
Firstname	#firstname
Lastname	#lastname
Pinno	#pinno #card no #account. Account(instance collection)
Where + - #	Public visibility Private visibility Protected visibility

22. What is bench mark?(May 2016)

A benchmark is a point of reference by which something can be measured. In surveying, "bench mark" (two words) is a post or other permanent mark established at a known elevation that is used as the basis for measuring the elevation of other topographical points.

UNIT – V

1. Define language errors, run time errors and logical errors

Language Errors or syntax errors: It results from incorrectly constructed code, such as an incorrectly typed keyword or some necessary punctuation omitted.

Run time errors: They occur and are detected as the program is running, when a statement attempts an operation that is impossible to carry out.

Logic errors: When codes do not perform the way you intended. The code might be syntactically valid and run without performing any invalid operations and yet produce incorrect results.

2. What is Black box testing?(Dec 2016)

Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings. This method offtest can be applied virtually to every level of software testing: unit, integration, system and acceptance.

3. Define a test case. Give an example.(Dec 2016, May 2017)

A test case is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly. The process of developing test cases can also help find problems in the requirements or design of an application. A Test Case is a set of actions executed to verify a particular feature or functionality of our software application.

Example:

Let us say that we need to check an input field that can accept maximum of 10 characters.

While developing the test cases for the above scenario, the test cases are documented the following way. In the below example, the first case is a pass scenario while the second case is a FAIL.

Scenario	Test Step	Expected Result	Actual Outcome
Verify that the input field that can accept maximum of 10 characters	Login to application and key in 10 characters	Application should be able to accept all 10 characters.	Application accepts accepts all 10 10characters.
Verify that the input field that can accept maximum of 11 characters	Login to application and key in 11 characters	Application should NOT accept all 11 characters.	Application accepts accepts allall 10 characters.

4. Name some of the testing strategies.

- Black- Box Testing - White- Box Testing o Path Testing - Statement testing coverage - Branch testing coverage - Top-Down Testing - Bottom-Up Testing

5. Differentiate TQM and Six Sigma

TQM	Six Sigma
It also focuses on continuous quality improvements.	It focuses on continuous quality improvements for achieving near perfection
It views quality as conformance to internal requirements.	It views quality as conformance by restricting the number of possible defects.
It is not so	It is complementary to statistical process control
TQM initiatives focus on improving individual operations within unrelated business processes	It focuses on improving all the operations within a single business process.
TQM initiatives are usually a part-time activity that can be managed by non-dedicated managers	Six Sigma projects require the skills of professionals that are certified

6. Specify the issues in OO testing.

Composition Issues

- Objective of OO is to facilitate easy code reuse in the form of classes
- To allow this each class has to be rigorously unit tested.

Encapsulation issues

- Encapsulation requires that classes are only aware of their own properties, and are able to operate Independently
- If we do not have access to source code then structural testing can be impossible
- If we violate encapsulation for testing purposes, then the validity of test could be questionable

Inheritance - issues

- Inheritance is an important part of the object oriented paradigm .
- Unit testing a class with a super class can be impossible to do without the super classes methods/variables.

Polymorphism Issues

- Repeatedly testing same methods

7. Define unit, the purpose of unit testing.

- a single, cohesive function
- a function which, when coded, fits on one page
- the smallest separately compilable segment of code
- a task in a work breakdown structure
- code that is assigned to one person
- It ensures the independent, accurate execution of a function

8. Define MM- path, ASF, GUI testing

- A Method/Message Path (MM-Path) is a sequence of method executions linked by messages.
- An MM-Path starts with a method and ends when it reaches a method which does not issue any messages of its own.

ASF

- An Atomic System Function (ASF) is an input port event, followed by a set of MM-Paths, and terminated by an output port event.
- An atomic system function is an elemental function visible at the system level.

GUI testing

- It is the process of ensuring proper functionality of the graphical **user interface** (**GUI)** for agiven application and making sure it conforms to its written specifications.

9. What are the difficulties in GUI testing?

- GUI test automation is difficult
- Often GUI test automation is technology-dependent
- Observing and trace GUI states is difficult
- UI state explosion problem
- Controlling GUI events is difficult
- GUI test maintenance is hard and costly

10. Explain COTS and USTS?

- **Commercial off –the –shelf** (COTS) software tools are already written and a few are available for analyzing and conducting user satisfaction tests.
- **User satisfaction test spreadsheet** (USTS) automates many bookkeeping tasks and can assist in analyzing the user satisfaction test results.

11. Define Cryptanalysis / breaking the cipher/ crypto system.

- **Cryptanalysis** refers to the study of ciphers, cipher text, or cryptosystems with a view to finding weaknesses in them that will permit retrieval of the plaintext from the cipher text, without necessarily knowing the key or the algorithm.

12. What are the key functions of Inventory Tracking system?

• Tracking inventory as it enters the warehouse, shipped from a variety of suppliers.

- Tracking orders as they are received from a central but remote telemarketing organization; orders may also be received by mail, and are processed locally.
- Generating packing slips, used to direct warehouse personnel in assembling and then shipping an order.
- Generating invoices and tracking accounts receivable.
- Generating supply requests and tracking accounts payable.

13. What are the seven major functional activities in this business?

Major functional activities

• **Order entry** Responsible for taking customer orders and for responding to customer queries about the status of an order

Accounting Responsible for sending invoices and tracking customer payments (accounts receivable) as well as for paying suppliers for orders from purchasing (accounts payable)

• **Shipping** Responsible for assembling packages for shipment in support of filling customer orders

• **Stocking** Responsible for placing new inventory in stock as well as for retrieving inventory in support of filling customer orders

• **Purchasing** Responsible for ordering stock front suppliers and tracking supplier shipments

• **Receiving** Responsible for accepting stock from suppliers

Planning Responsible for generating reports to management and studying trends in inventory levels and customer activity

14. Define Client / server computing

- It is a decentralized architecture that enables end users to gain access to information transparently within a multivendor environment.
- Client –server applications couple a GUI to a server-based RDBMS.

15. State the components of client / server application.

- **Presentation logic**
- The part of an application that interacts with an end-user device such as a terminal, a bar code reader, or a handheld computer.
- Functions include "screen formatting, reading, and writing of the screen information, window management, keyboard, and mouse handling."
- **Business Logic**
- The part of an application that uses information from the user and from the database to carry out transactions as constrained by the rules of the business.
- Database Logic
- The part of an application that "manipulates data within the application.
- Data manipulation in relational DBMSs is done using some dialect of the Structured Query Language (SQL)."
- **Database processing**
- The "actual processing of the database data that is performed by the DBMS.
- The DBMS processing is transparent to the business logic of the application.

16. Why do you use blackboard framework? What are the components of blackboard framework?

- It holds the computational and solution-state data needed by and produced by the knowledge sources.
- The blackboard consists of objects from the solution space.
- The objects on the blackboard are hierarchically organized into levels of analysis.
- The objects and their properties define the vocabulary of the solution space

Components of blackboard framework

- A blackboard, multiple knowledge sources, and
- A controller that mediates among these knowledge sources

17. What is a knowledge source? State the reason for the use of knowledge source.

- The domain knowledge needed to solve a problem is partitioned into knowledge sources that are kept separate and independent.
- The objective of each knowledge source is to contribute information that will lead to a solution to the problem.

- A knowledge source takes a set of current information on the blackboard and updates it as encoded in its specialized knowledge.
- The knowledge sources are represented as procedures, sets of rules, or logic assertions

18. Why do you use UML?

- It allows communicating certain concepts more clearly than the alternatives.
- It is used to build a road map of a large system, use package diagrams to show the major parts of a system and their interdependencies.
- For each package draw a class diagram.
- Use patterns to describe the important ideas in the system that appears in multiple places.
- Patterns help you to explain why your design is the way it is.
- It is also useful to describe designs you have rejected and why you rejected them.

19. What are the two approaches for reasoning?

- Forward-chaining involves reasoning from specific assertions to a general assertion
- Backward-chaining starts with a hypothesis, then tries to verify the hypothesis from existing assertions.

20. Distinguish between runtime errors and logic errors. (May 2015)

Run time errors: They occur and are detected as the program is running, when a statement attempts an operation that is impossible to carry out.

Logic errors: When codes do not perform the way you intended. The code might be syntactically valid and run without performing any invalid operations and yet produce incorrect results.

21. What is the impact of testing on object orientation? (May 2015)

- some types of errors could become less plausible.(not worth testing for)
- some types of errors could become most(worth testing for now)
- Some new types of errors might appear.
- For example, when you invoke a method, it may be hard to tell exactly which method gets executed.
- The method may belong to one of many classes.
- It can be hard to tell the exact class of the method.
- When the code accesses it, it may get unexpected value.

22. Write the objectives of testing.(May 2016, May 2017)

- Finding defects which may get created by the programmer while developing the software.
- Gaining confidence in and providing information about the level of quality.
- To prevent defects.
- To make sure that the end result meets the business and user requirements.
- To ensure that it satisfies the BRS that is Business Requirement Specification and SRS that is System Requirement Specifications.
- To gain the confidence of the customers by providing them a quality product.

23. Name the two issues for software Quality. (May 2016)

1. Lack of domain knowledge:
2. Lack of technology knowledge:
3. Unrealistic schedules:
4. Badly engineered software:

24. Define debugging.

Debugging is the process of finding out where something went wrong and correcting the code to eliminate the errors or bugs that cause unexpected results.

Part – B - UNIT – I

1. Describe in detail about the object basics in object oriented environment.
   - Object is anything in the real world in an object
   - It has properties/attributes that describe the state of an object and methods/procedures which define its behaviour
   - Eg., Stacks, Bonds – objects for investment application
   - Parts, Assemblies - material management application
   - **Factors**

- What object does the application need?
- What functionality should these object have?
- Each object is responsible for itself.
- It is a combination of data and logic
- A window object is responsible for things like opening, sizing, and closing itself.
- A chart object is responsible for things like maintaining its data and labels, and even for drawing itself.
- It is represented by the class diagram.

Object	***Car***	***Window***	***chart***
Attributes	Colour	size	label
	Year	make	data
	model	price	
	Price		
Methods	Go () Stop () Turn_left () Turn_right ()	Open () Close ()	Draw ()

OBJECTS ARE GROUPED INTO CLASSES

- Class – set of objects that share a common structure and behaviour
- Single object- instance of a class
- Class- specification of structures (instance var), behaviour(methods) and inheritable for objects
- It is a generic notion of an object

E.g., employee class

x y z (objects/instance)

- Every object of a given class has same data format and respond to same set of instructions.
- The data associated with a particular object is managed by itself.

Eg., x, y

data- SSN, addr, salary

ATTRIBUTES: OBJECT STATE, PROPERTIES

Properties- state of an object

E.g., car(object)

cost, color, make , model…. (attributes)

- Color- sequence of character
- Cost- floating point/fixed point/integer

- An object abstract state can be independent of its physical representation

OBJECT BEHAVIOUR AND METHODS

- Object behavior is described in methods/procedures
- Method is a function/procedure that is defined for a class (i.e.,) it accesses the internal state of an object

- Behavior is a collection of methods that describes what an object is capable of doing
- The object called as the receiver is the one on which method operates
- Methods encapsulate the behavior of the object
- Methods provide interfaces to the object and hide any of the internal structure and state maintained by the object

- **Methods** provide a means to communicate with an object and access its properties

e.g., object- employee

message- compute payroll

employee

emp id, bp, hra, da, pf, ta, lic premium

cal netpay(), cal gross (), cal deduct()

```
class payroll
{
        int empid;
        float bp;
        float hra;
        float da;
        float pf;
        float ta;
        float lic;
void  cal netpay ()     //Method
{
  gross = bp + hra + da + ta;
  ded   =  lic premium + pf;
  netpay= gross - ded;
}
```

OBJECT RESPOND TO MESSAGES

- Object perform operations in response to messages
  E.g., car- stop message

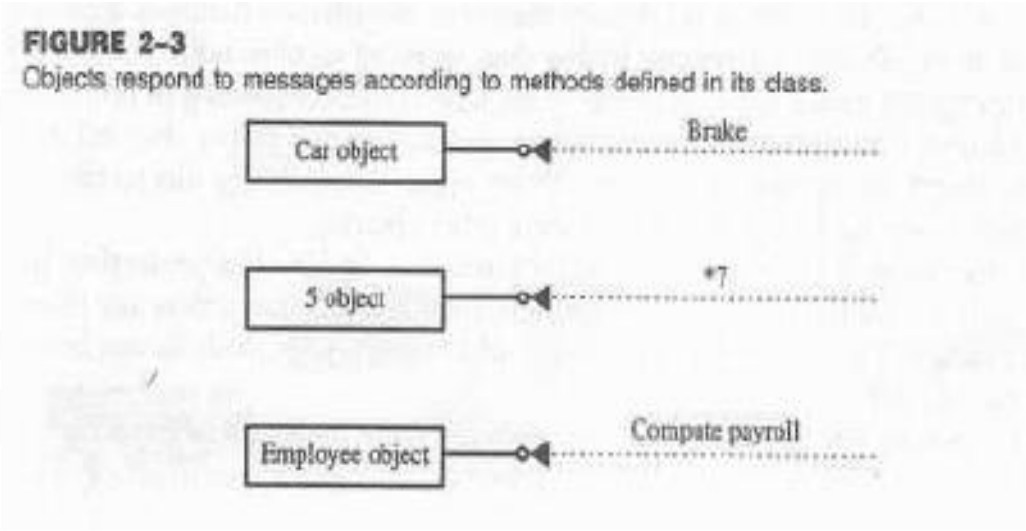
  - object responds to the message

- Messages are non-specific function calls

E.g., stop message – car, motorcycle, bicycle

- Object respond to messages according to methods defined in its class
  E.g.,

Object	Method
Car	Brake
5	* 7
employee	compute payroll



- Messages make no assumptions about the class of the receiver
- It is the receiver's responsibility to respond to message and results in flexibility

ENCAPSULATION AND INFORMATION HIDING

- Information hiding is the principle of concealing internal data and procedures of an object
- It provides an interface to each object in such a way to reveal its inner workings
- E.g., Simula – provides no protection /info hiding for object
- OO language provides a well-defined interface to their object thro' classes
- E.g., C++ - general encapsulation mechanisms with public, private and protected members
- Public members may be accessed from anywhere
- Private members are accessible only from within a class
- Protected members can be accessed only from subclasses

PER-OBJECT PROTECTION

- Methods can access only the receiver
- For achieving encapsulation, different classes of object use a common protocol or object's user interface

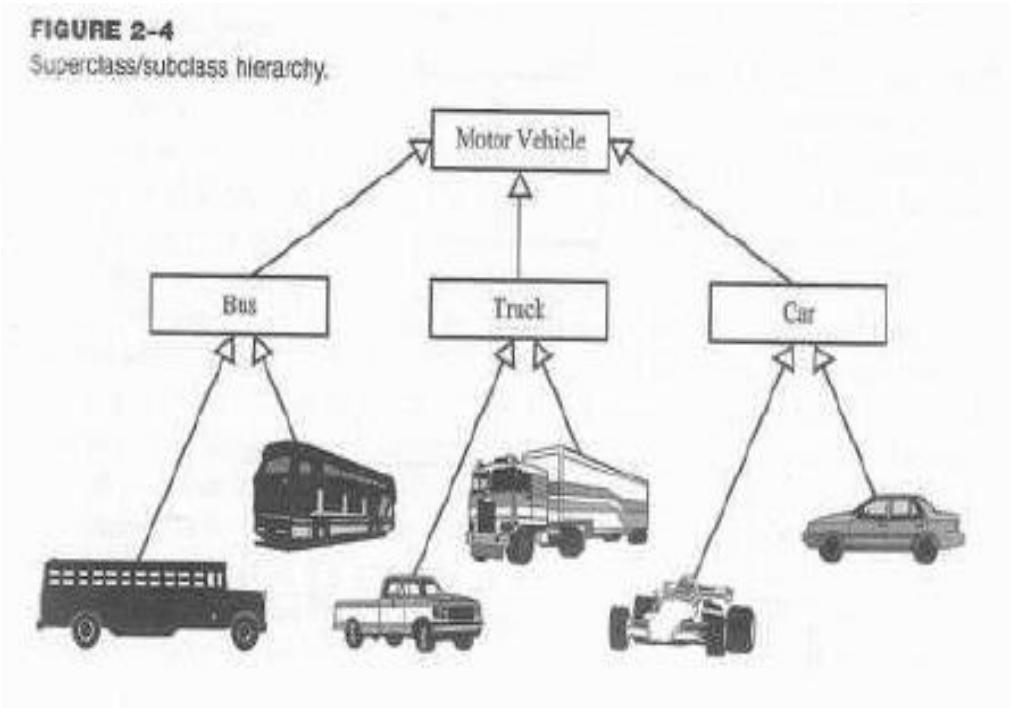
PER- CLASS PROTECTION

- Class methods can access any object of that class not only the receiver alone
  - E.g., engine- interface between the driver and car common protocol
- Data abstraction can be achieved , it incorporates encapsulation and polymorphism

CLASS HIERARCHY

- OO system organizes classes into subclass-super class hierarchy
- Top of the class hierarchy- most general classes/
- Bottom of the class hierarchy- most specifier
- Subclass – inherits all the properties and methods defined in the super class
- Subclasses and methods and properties specifier to that class

FIGURE 2-4
Superclass/subclass hierarchy.



- Super class generalizes the behavior
- A class may simultaneously be the subclass to some class and super class to another class
- Formal or abstract classes have no instances but define common behavior that can be inherited by more specifier classes
- Allows object to built from other object
- It is the relationship between classes, when one class is the parent classes of another class
- Parent class-base class/super class
- Allows classes to share and reuse behavior
- A class also inherits the behavior and attributes of all its super class
- Dynamic inheritance allows object to change and evolve over-time

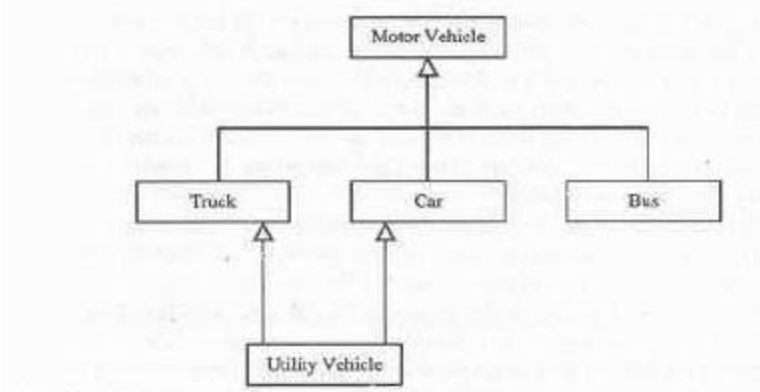


FIGURE 2-7
Utility vehicle inherits from both the Car and Truck classes.

- Dynamic inheritance - the ability to add, delete, change base classes from object at runtime

POL POLYMORPHISM

- Same operation behaving differently on different classes
- Polymorphism is the main difference when a message and sub-routine is called anywhere inside the program
- message refers to the instruction while methods define the implementation of the instructions.
- Object understands messages and messages has name like method
- An object first searches the methods defined by its class
  If found, that method is called up

  If not, it searches for the method in the super class

  If found, that method is called up

- Search is done upwards and a method specifies how to do
- Allow to write generic usable code
- No assumption is made about the class of an object

OBJECT RELATIONSHIP and ASSOCIATIONS

- Association – relationships between objects and classes
- Association is bi-directional

Eg., can fly – forward direction flown by – backward direction

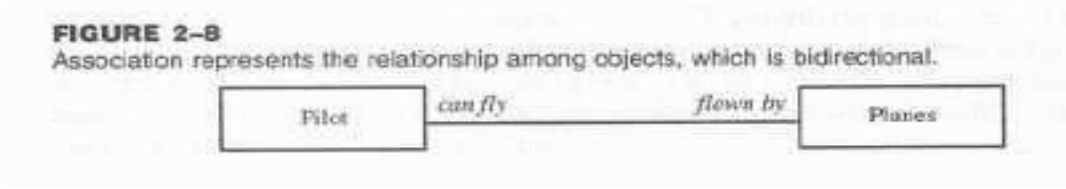


FIGURE 2–8
Association represents the relationship among objects, which is bidirectional.

- Cardinality represents how many instances of one class may relate to a single instance of an associated class

1-1 - 1-1

1-m - 1-*

m-1 - *-1

m-m - *-*

- 0……*

- 0……n

2. i)Discuss the object oriented system development life cycle in detail.(Dec2016, May2017)

- It consists of analysis, design, implementation, testing and refinement to transform users' needs into a software solution that satisfies those needs.
- The process can be divided in to several subtasks.
- Each subtask may have
- A description in terms of hoe does it work
- Specification of the input required for the process
- Specification of the output to be produced
- The software development process can be divided into smaller, interacting sub processes which may have further transformations

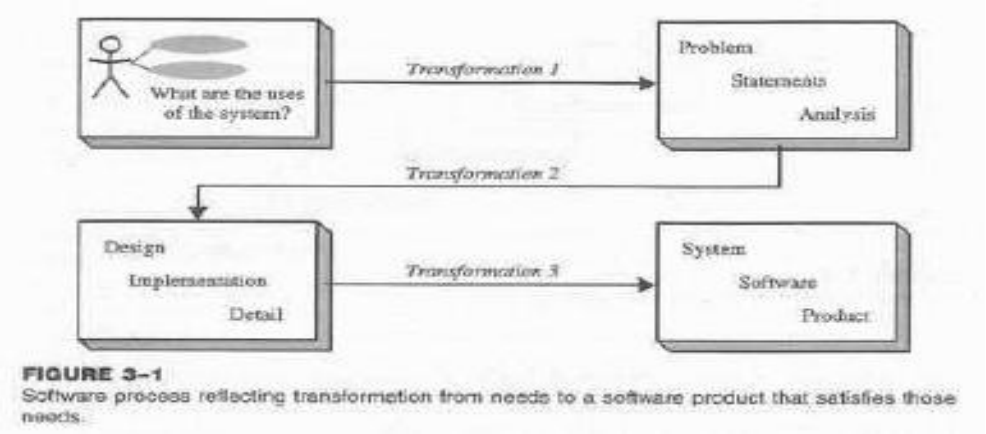


FIGURE 3–1
Software process reflecting transformation from needs to a software product that satisfies those needs.

- **Transformation 1 – Analysis**
- It translates the users' needs into system requirements and responsibilities
- **Transformation 2– Design**
- It begins with a problem statement and ends with the detailed design that can be transformed into an operational system
- **Transformation 3 – Implementation**
- It refines the detailed design into the system deployment that will satisfy the users' needs
- It includes the equipment, procedures, etc
- It represents embedding the software within its operational environment
- **E.g., Waterfall Model**
- This model is considered as the foundation for the rest of the models.
- It has analysis, design, coding, testing, implementation and deployment as the phases of software development.
- During the analysis phase of the development, the requirements from the customer is collected.
- A project plan is prepared to meet the client's requirements and a software requirement specification document is developed as the outcome of the project plan.
- The design phase of the project includes the preliminary design, architectural design and a detailed design of the entire project which is pictorially / diagrammatically represented either as a DFD or as a class diagram or as an ER diagram.
- The appropriate language or the environment is chosen for the development of the project during the coding phase of the application.
- The project that is developed will be tested with sample valid and invalid test cases to check the functionality of the entire application during the testing phase of the application.
- The application's functionality is completely tested at the developer's end during the implementation of the project.
- The actual system is installed at the clients' premises during the deployment phase of the project development.
- In each and every phase of the development, all the activities are to be documented.
- As a result, this model takes more time to develop and complete the project.
- This model is not suited for all types of real time application.

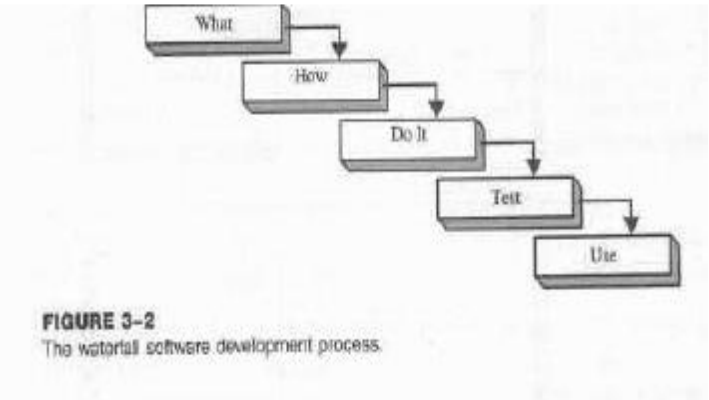


FIGURE 3–2
The waterfall software development process.

- **Building high quality software**
- The software process transforms the users' needs through the application domain to a software solution that satisfies those needs
- Once the system / program exists, test it to see if it is free of bugs.
- High quality products must meet users' needs and expectations
- The product should attain this high quality with minimal or no defects.
- The main focus is on improving the products or services prior to delivery of the product to the customer.
- The goal of building high quality software is user satisfaction
- How do we determine when the system is ready for delivery to the customer?
- Is it now an operational system that satisfies users' needs?
- Does it pass an evaluation process?
- **Basic approaches to system testing**
- **Correspondence**
- It measures how well the delivered system matches the needs of the operational environment
- **Validation** is the task of predicting correspondence
- The correspondence can not be determined until the system is in place.
- **Correctness**
- It measures the consistency of the product requirements with respect to the design specification
- **Verification** is the exercise of determining correctness
-

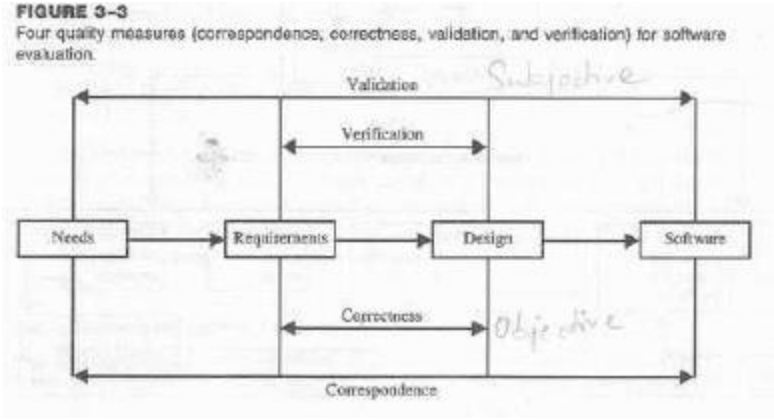


FIGURE 3-3
Four quality measures (correspondence, correctness, validation, and verification) for software evaluation.

ii) Differentiate Structured Approach from Object Oriented Approach.

SA/SD approach relies on modeling the processes that manipulate the given input data to produce the desired output.

- Involves creation of preliminary data flow diagrams and data modeling.
- Encourages top down design / top down decomposition /stepwise refinement.
- It works by continually refining a problem into simpler and simpler chunks.
- For every employee do

Begin

If employee = manager then

Call computemanagersalary

If employee = officeworker then

Call computeofficeworkersalary

If employee = production worker tehn

Call compute production worker salary

End

- **Object oriented approach**
- Object oriented analysis
- Object oriented information modeling
- Object oriented design
- Prototyping and implementation
- Testing, iteration and documentation
- Identification of classes
- Find the classes of objects that will compose the system i.e., Identify the entities
- These entities / objects can be individuals, organizations, machines, units of information, etc that makes the context of the real world system
- These entities help in developing a workable system
- Identify the hierarchical relation between super classes and sub classes
- Identify the attributes or properties of the objects and the behaviour / methods of the objects
- Assign each responsibility to the class to which it logically belongs
- Employee (name, address,salary.ss#)\
- Office worker (dataentry().Compute payroll(),Printreport())
- Manager (dataentry().Compute payroll(),Printreport())
- Production worker(dataentry().Compute payroll(),Printreport())
-

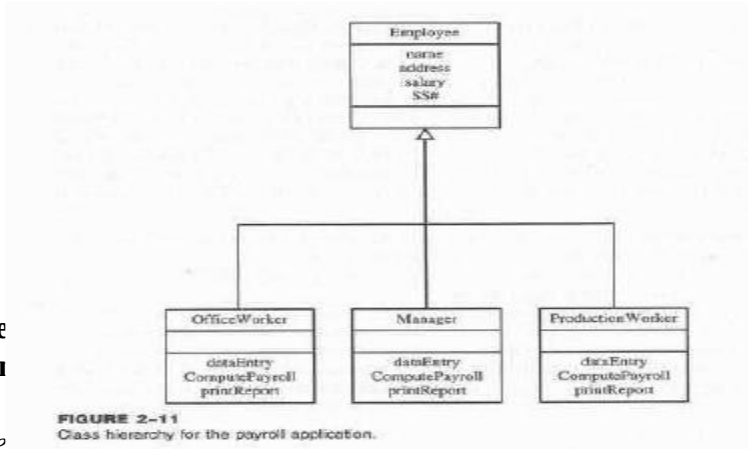


FIGURE 2–11
Class hierarchy for the payroll application.

- **For e**
  Begi

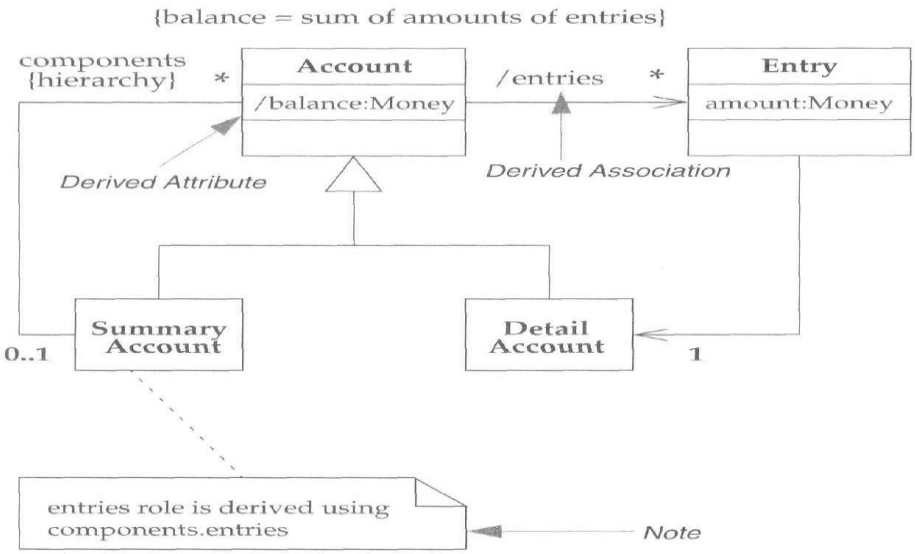
Employee computepayroll

End

3. Model a class diagram for a banking system. State the functional requirements you are considering.(Dec 2016)

- **Association**

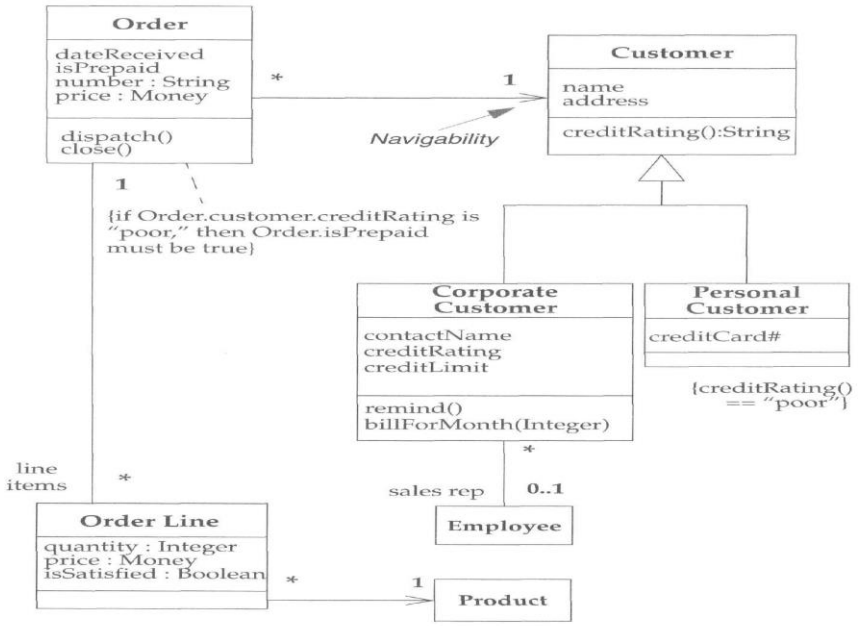
- A relationship can be defined by way of establishing associations among the objects.
- Such associations can be either unidirectional / bidirectional.
- Can use multiplicity to represent how many instances of one class may relate to a single instance of an associated class.
- Such association can be consumer – producer association, super – sub class association or a part of association.
- UML meta-model that shows the relationship among associations and generalization.
- associations represent conceptual relationships between classes.
- Derived associations and derived attributes can be calculated from other associations and attributes, respectively, on a class diagram.
- The purpose of identifying the relationships among classes and objects is to solidify the boundaries of and to recognize the collaborators with each abstraction identified earlier in the micro process.
- This activity formalizes the conceptual as well as physical separations of concern among abstractions begun in the previous step.
- Expressing the existence of an association identifies some semantic dependency between two abstractions, as well as some ability to navigate from one entity to another.
    - As part of design, we apply this step to specify the collaborations that: form the mechanisms of our architecture, as well as the higher-level clustering of classes into categories and modules into subsystems.
- Associations are used for reference objects. Frozen is a constraint that the UML defines as applicable to an attribute or an association end,



- Frozen indicates that the value of that attribute or association end may not change during the lifetime of the source object.
- A qualified association is known as associative arrays, maps, and dictionaries.

- class Order {
- public OrderLine getLineItem
- (Product aProduct);
- public void addLineItem
- (Number amount, Product forProduct);

- As implementation proceeds, we refine relationships such as associations into more
- Implementation-oriented relationships, including instantiation and use.
  - Each association has two association ends; each end is attached to one of the classes in the association.
  - An end can be explicitly named with a label.
  - This label is called a role name. (Association ends are often called roles.)
  - An association end also has multiplicity, which is an indication of how many objects may participate in the given relationship.
  - interface for an Order class:
    - class Order {
    - public Customer getCustomer();
    - public Set getOrderLines();
    - class Customer {
    - private Set _orders;
  - If navigability exists in only one direction, we call the association a **unidirectional association. A bidirectional association** contains navigability in both directions.
  - An association represents a permanent link between two objects



- **Aggregation** refers to the whole – part of relationships among objects.
- All objects are composed of and may contain other objects.
- An object can refer to other objects.
- An attribute can be an object t itself. .Parts with multiplicity > 1 may be created after the aggregate itself but, once created, they live and die with it.
- **Aggregation**
- The "is a" hierarchies denote generalization/specialization relationships, "part of, hierarchies describe aggregation relationships.
- For example, consider the following class:

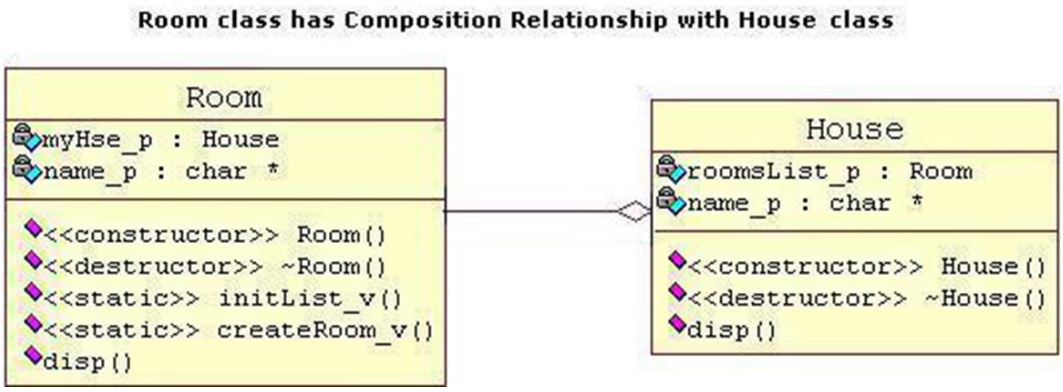
```
class Garden {
    public:
    Garden();
    virtual ~Garden();
    protected:
    Plant* repPlants[100];
    GrowingPlan repPlan;
    };
```

Aggregation permits the physical grouping of logically related structures, and inheritance allows these common groups to be easily reused one different abstractions

Composition

- **Composition** is a specialized form of Aggregation.
- It is a strong type of Aggregation.
- The Parent and Child objects have coincident lifetimes.
- Child object dose not have it's own lifecycle and if parent object gets deleted, then all of it's child objects will also be deleted.
- **Referential Integrity principle**
- Let’s take an example of a relationship between House and it's Rooms.

- House can contain multiple rooms there is no independent life for room and any room can not belong to two different house.
- If we delete the house room will also be automatically deleted.



4. Model a class diagram for a "Bus Ticket Booking System". State the functional requirements that you are considering. (May 2017)

Refer previous question

5. Write short notes on the following

i) Associations and Aggregations, composition

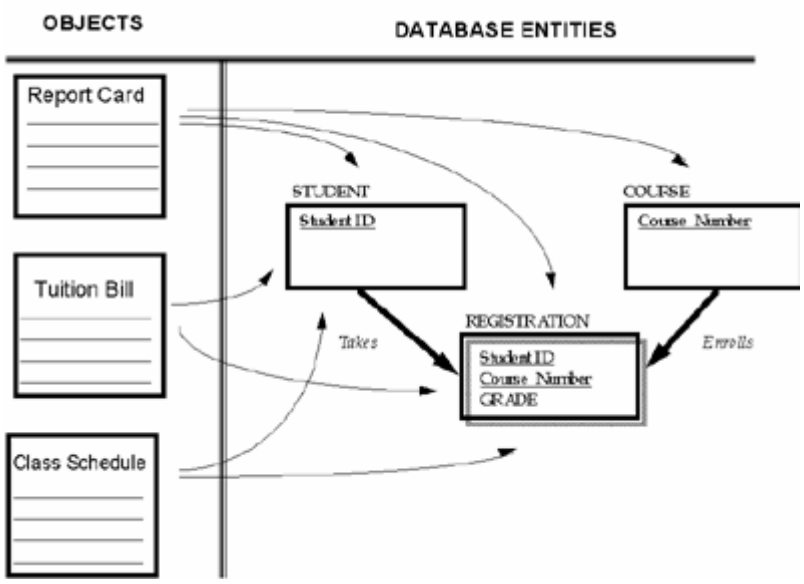
Refer previous question number 3

ii) Class Hierarchy and Inheritance

The IS-A relationship (pronounced "is a") is a data relationship that indicates a type/subtype data relationship.

While traditional Entity/Relation modeling deals only with single entities, the IS-A approach recognizes that many types or classes of an individual entity can exist.

In fact, the IS-A relationship is the foundation of object-oriented programming, which allows the designer to create hierarchies of related classes and then use inheritance and polymorphism to control which data items will participate in the low-level objects.



- After establishing a class hierarchy with the E/R model, the object-oriented principle of generalization is used to identify the class hierarchy and the level of abstraction associated with each class.
- Generalization implies a successive refinement of a class, allowing the super-classes of objects to inherit data attributes and behaviors that apply to the lower levels of a class.
- Generalization establishes taxonomy hierarchies.
- Taxonomy hierarchies organize classes according to their characteristics in increasing levels of detail.
- These hierarchies begin at a very general level and then proceed to a specific level, with each sublevel having its own unique data attributes and behaviors.
- In Figure 2.12, the IS-A relationship is used to create a hierarchy within the EMPLOYEE class.
- The base class for an employee has the basic data items such as name, address and phone number.
- There are sub-classes of employees, executives and hourly employees, each with their own special data items and methods.

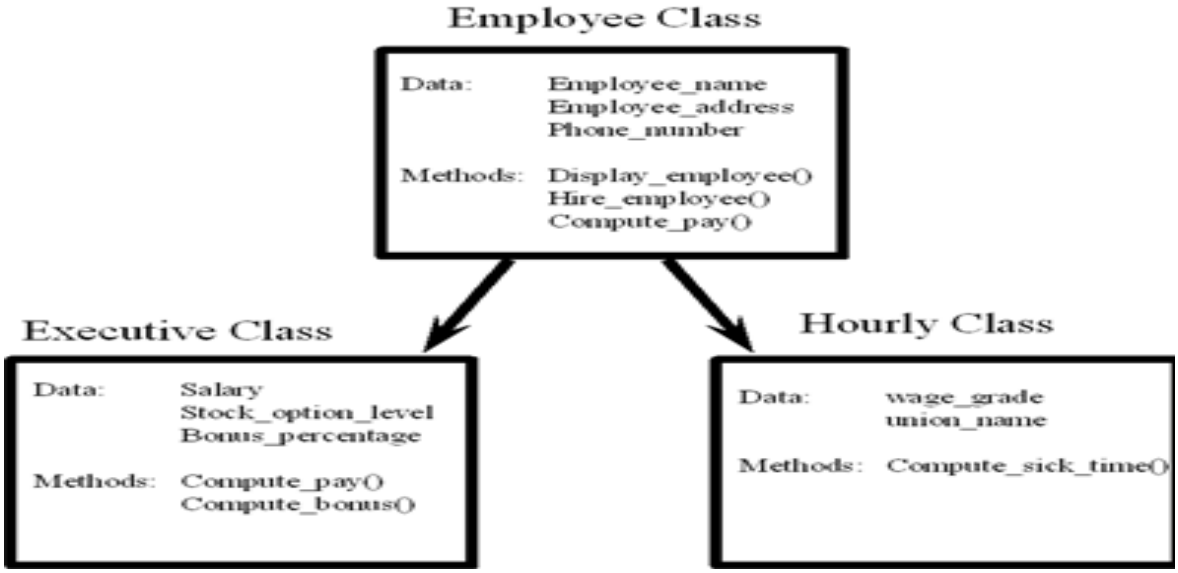


Figure 2.12 - An Entity/Relation model with added IS-A relationships.

- Consider the application of the IS-A relationship for a vehicle dealership, as shown in Figure 2.13. As you can see, the highest level in the hierarchy is VEHICLE.
- Beneath the vehicle class, you might find car and boat subclasses.
- Within the car class, the classes could be further partitioned into classes for TRUCK, VAN, and SEDAN.
- The VEHICLE class would contain the data items unique to vehicles, including the vehicle ID and the year of manufacture.
- The CAR class, because it IS-A VEHICLE, would inherit the data items of the VEHICLE class.
- The CAR class might contain data items such as the number of axles and the gross weight of the vehicle.
- Because the VAN class IS-A CAR, which in turn IS-A VEHICLE, objects of the VAN class inherit all data items and behaviors relating to the CAR and VEHICLE classes.

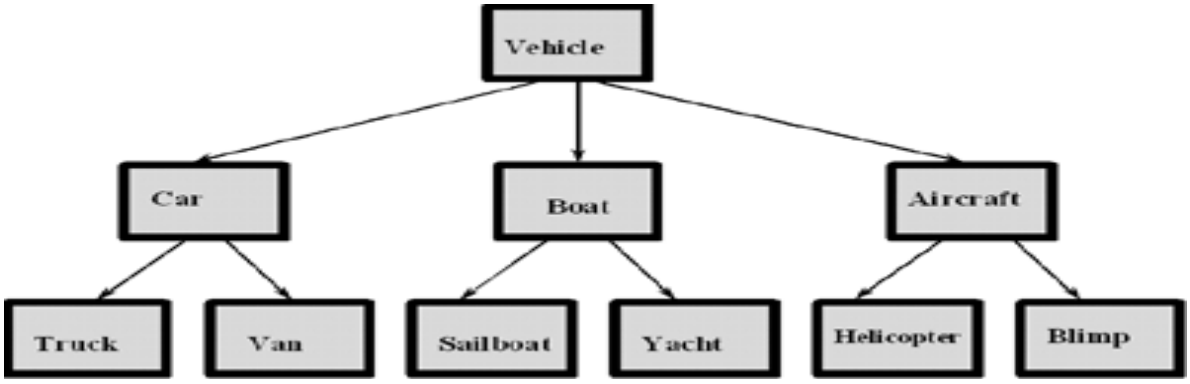


Figure 2.13 – A class hierarchy for a vehicle rental company.

Oracle has the ability to "create type" within type and model this relationship directly, but it is not a popular approach because the structures are hard to change.

The first technique (incorrect IMHO) is to create sub-tables for car, boat, sedan, and so on.

This encapsulates the data items within their respective tables, but it also creates the complication of doing unnecessary joins when retrieving a high-level item in the hierarchy.

For example, the following SQL would be required to retrieve all the data items for a luxury sedan:

```
select
   vehicle.vehicle_number,
   car.registration_number,
   sedan.number_of_doors,
   luxury.type_of_leather_upholstery
from
   vehicle,
   car,
   sedan,
   luxury
where
   vehicle.key = car.key
and
   car.key = sedan.key
and
   sedan.key = luxury.key;
```

- The IS-A relationship is best suited to the object-oriented data model, where each level in the hierarchy has associated data items and methods, and inheritance and polymorphism can be used to complete the picture.
- It is important to note that not all classes within a generalization hierarchy will be associated with objects.
- These non-instantiated classes only serve the purpose of passing data definitions to the lower-level classes.
- The object-oriented paradigm allows for abstraction, which means that a class can exist only for the purpose of passing inherited data and behaviors to the lower-level entities.
- The classes VEHICLE and CAR probably would not have any concrete objects, while objects within the VAN class would inherit from the abstract VEHICLE and CAR classes.

Inheritance

- A sub class inherits all of the properties and methods / procedures defined in its super class.
- Super classes generalizes behaviour
- A class may simultaneously be the sub class to some class and a super class to another class / classes
- The car class defines how a car behaves.
- E.g., the Ford class defines the behaviour of Ford cars.
- The Ford class inherits the general behaviour from the car class and adds specific behaviour to Ford.
- the redefinition of the behaviour of the car class is not needed.
- The stop method is defined in class Ford and not in Mustang class
- The Mustang class can inherit behaviour from the car and the vehicle classes
- The behaviours of any given class are behaviours of its super class or a collection of classes

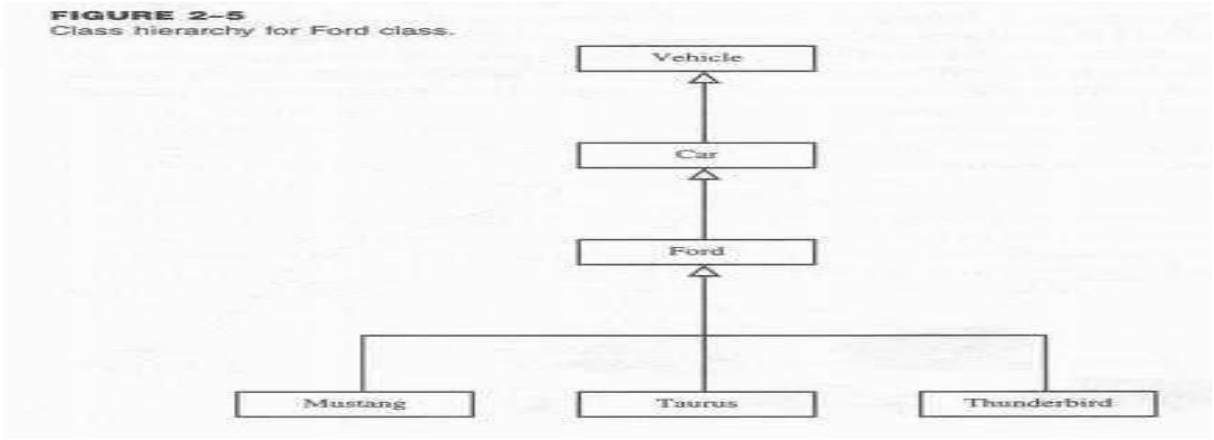


FIGURE 2–5
Class hierarchy for Ford class.

- Inheritance allows explicitly by considering the commonality of the objects when constructing new classes
- It is an association between classes where one class is the parent class of another derived class.
- The parent class is also known as the base class or super class
- Inheritance provides programming by extension.
- Reusability is achieved through inheritance
- It allows classes to share and reuse behaviours and attributes.
- The behaviour of a class instance is defined in that class's methods.
- A class also inherits the behaviour and attributes of all of its super classes.
- Dynamic inheritance / run time polymorphism allows objects to change and evolve over time.
- Base classes provide properties and attributes for objects.
- Changing base classes changes the properties and attributes of a class
- It refers to the ability to add, delete or change parents from objects or classes at run time.

OBJECT IDENTITY

- Every object has its own unique id

MC• An object id comes into being when it is created

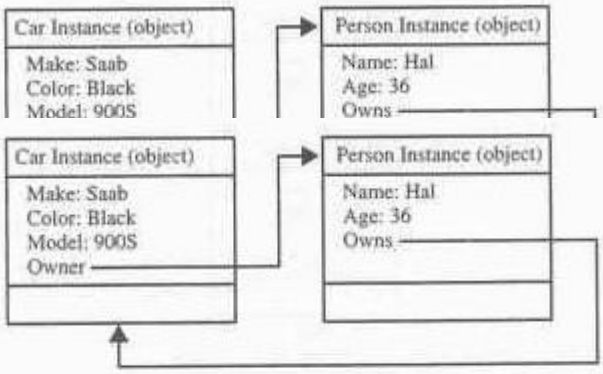


FIGURE 2-13
The owner property of a car contains a reference to the person instance named Hal.

directly denote the object to which

Static binding	dynamic binding
• Values are associated to the variables at the compile time	• Values are associated to the variable at runtime/execution time
• Optimizes the calls	• It occurs when polymorphic calls are issued
• It is not so	• Some method invocation decisions to be deferred until the information is known

OBJECT PERSISTENCE

- Lifetime of a particular object how long an object can exist for a period of time

META CLASS

- A class is an object.
- If it is an object, it must belong to a class.
- Such a class belong to a class is called as a meta class or a class of classes.
- Classes can be implemented with the set of methods, instances and parents / super class
- This can be defined as a class called meta class.
- This meta class can also provide services to application programs such as returning a set of all methods, instances.
- All objects are instances of a class and all classes are instances of a meta class

6. Explain Object Identity, Static and Dynamic Binding, Object Persistence and Meta classes.

7. Describe in detail about the following
   i) Prototyping and types of prototypes
   ii) Rapid Application Development
   iii)Component Based Development

Prototyping

- A prototype is a version of a s/w product developed in the early stage of the product life cycle for specific, experimental purpose.
- It is considered as the first model of the system.
- Since the customer requirements is not taken into account, the innovative ideas of the developer are applied for the development of the project.
- It does not have specification obtained from the client, no customer feedback and deployment.
- With lesser time, a prototype can be developed.
- Once the developer of the prototype is satisfied with its performance, the prototype can be converted into a commercial product to be sold to a client.
- At this stage, the needy client and the specifications are obtained from the actual client and the product is accordingly developed.
- The prototype that is kept at the experimental lab is now deleted. Hence it is known as the throw away prototype.
- It can further define the use case and makes use case modeling much easier.

- **Classification of prototype**

  1. Horizontal Prototype 3. Analysis Prototype
  2. Vertical Prototype 4. Domain Prototype

- **Horizontal Prototype**
- A horizontal prototype is a simulation of the interface but contains no functionality.
- This has the advantage of being very quick to implement, providing a good overall feel of the system.
- **Vertical Prototype**
- A vertical prototype is the subset of system features with complete functionality.
- A few implemented function can be tested in great depth.
- In practice prototype is hybrid between horizontal and vertical.
- **Analysis Prototype**
- 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 the concept.
- It can be discarded when it has served its purpose.
- **Domain prototype**
- A domain prototype is an aid for the incremental development of the ultimate software solution.
- The software is used as a tool for the staged deliver of subsystem to the user or other members of the development team.

- It demonstrates the feasibility of the implementation and eventually will evolve into a designable product.
- The typical time required to produce a prototype is from a few days to several weeks depending on the type and function of the prototype.

ii) Rapid Application Development (RAD) Model

- "Rapid Application Development (RAD) is an incremental software development process model that emphasises a very short development cycle [typically 60-90 days]."
- The RAD model, is a high-speed adaptation of the waterfall model, where the result of each cycle a fully functional system.
- RAD is used primarily for information systems applications
- The RAD approach encompasses the following phases:

Business modelling

- The information flow among business functions is modeled in the following manner:
- What information drives the business process?
- What information is generated?
- Who generates it?
- Where does the information go?
- Who processes it?

Data Modelling

- The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business.
- The characteristics (called attributes) of each object are identified and the relationships between these objects are defined.

Process Modelling

- The data objects defined in the data-modeling phase are transformed to achieve the information flow necessary to implement a business function.
- Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

Application Generation

- RAD assumes the use of the RAD fourth generation techniques and tools like VB, VC++, Delphi etc rather than creating software using conventional third generation programming languages.
- The RAD works to reuse existing program components (when possible) or create reusable components (when necessary).
- In all cases, automated tools are used to facilitate construction of the software.

Testing and turnover

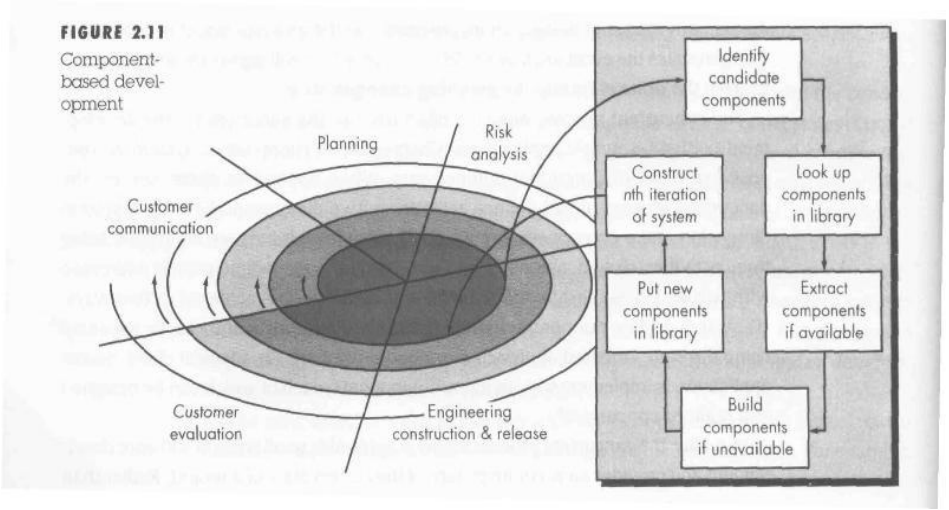
- Since the RAD process emphasizes reuse, many of the program components have already been tested.
- This minimizes the testing and development time.
- If a business application can be modularized so that each major function can be completed within the development cycle then it is a candidate for the RAD model.
- In this case, each team can be assigned a model, which is then integrated to form a whole.

Disadvantages

- For Large (but scalable) projects, RAD requires sufficient resources to create the right number of RAD teams.
- RAD projects will fail if there is no commitment by the developers or the clients to 'rapid-fire' activities necessary to get a system complete in a much abbreviated time frame.
- If a system cannot be properly modularized, building components for RAD will be problematic
- RAD is not appropriate when technical risks are high, e.g. this occurs when a new application makes heavy use of new technology.

Component – Based Development

- Component-Based Development (spiral model variation in which applications are built from prepackaged software components called classes) .
- It is an iterative approach to the creation of software.
- It composes applications from prepackaged software components.(Classes).
- Identify the candidate classes.
- Data & algorithms are packaged into a class.
- All the Classes are stored in a class library.
- To identify the classes , search it in the library , if it is there extract and reuse it.
- If it is not in the library .it is engineered using object oriented methods.
- **Steps**
- Available component – based products are searched and evaluated for the application domain in question.
- Component integration issues are considered
- Software architecture is designed to accommodate the components.
- Components are integrated into the architecture.
- Comprehensive testing is conducted to ensure proper functionality



Advantages

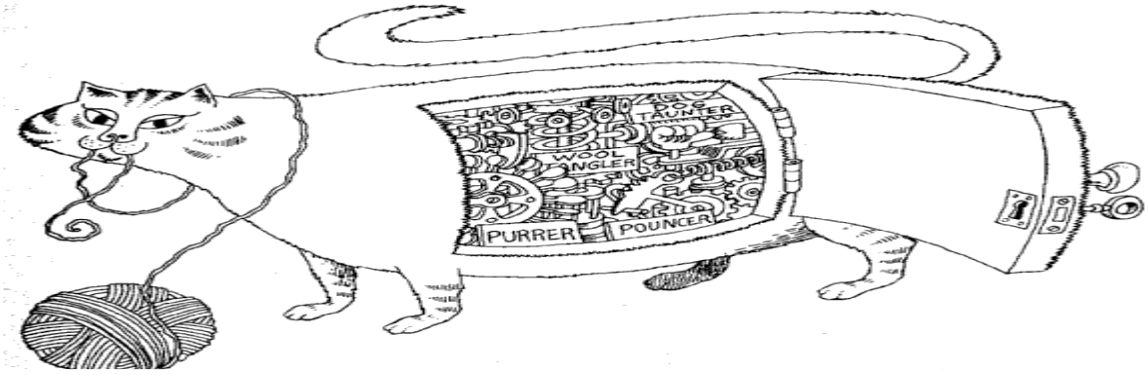
- It leads to software reusability.

- It reduces 70% reduction in development cycle time & 84% reduction in project cost.
- Its productivity index is 26.2%.

8. Write short notes on the following

Encapsulation and Abstraction with examples

- Encapsulation refers to the protection mechanism with public, private and protected members
- "No part of a complex System should depend on the internal details of any other part"
- An object encapsulates the data and the program.
- The user cannot see the contents inside the object but can use the object by calling the object's methods.
- Encapsulation helps an object to send a message to the target object requesting information.
- This ensures that no object can operate directly on another object's data.



Encapsulation hides the details of the implementation of an object.

- Encapsulation is most often achieved through *information biding*,
- It is the process of hiding all the secrets of an object that do not contribute to its essential characteristics; the structure of an object is hidden, as well as the implementation of its methods.
- Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

Examples of Encapsulation

Class *Heater* in C++

```
Class Heater {
    public:
      Heater (location);
     ~Heater ();
     Void turnOn();
    Void turnoff();
    Boolean isOn() const;
    private:
    …
```

```
};
```

This interface represents all that a client needs to know about the class **Heater**.

```
class Heater {
        public:
        …
        protected:
        const Location repLocation;
        Boolean repIs0n;
        SerialPort* repPort;
};
```

Implementation of each operation associated with this class

```
Heater::Heater (location 1) : repLocation (1), repIs0n (FALSE),
repPort(&SerialPort::ports[1]) {}
        Heater::~Heater() {}
        Void Heater::turnOn() {
        if (!repIs0n) {
        repPort->write(“*”);
        repPort->write(repLocation);
        repPort->write(1);
        repIs0n = TRUE;
        }
}
Void Heater::turnoff() {
        if (repIs0n) {
        repPort->write(“*”);
        repPort->write(replocation);
        repPort->write(O);
        repIs0n = FALSE;
}
}
Boolean Heater::isOn() const {
```

return repIs0n;

}

9. Explain in detail about Unified Approach.

- The idea behind the UA is to combine the best practices, processes, methodologies, and guidelines along with UML notations and diagrams.
- Unified approach is a methodology for software development.
- UML is a set of notations and conventions used to describe and model an application.
- UML does not specify a methodology.

- *The unified approach to software development revolves around (but is not limited to) the following processes and components.*

- *The processes are:*
  - Use-case driven development.
  - Object-oriented analysis.
  - Object-oriented design.
  - Incremental development and prototyping.
  - Continuous testing.

UA Methods and Technology

- *The methods and technology employed includes:*
  - Unified modelling language (UML) used for modelling.
  - Layered approach.
  - Repository for object-oriented system development patterns and frameworks.
  - Promoting Component-based development.

The unified approach comprises of analysis, design prototyping and testing.

- During analysis, identify the users / actors using the system requirements.
- Develop a simple business process model.
- Develop the use case.
- Develop the interaction, collaboration diagram.
- Apply classification.
- Design
- Apply design axioms to design classes, their attributes, methods, aggregation, structures and protocols.
- Design the access layer
- Design the view layer.
- Prototype and interface
- Usability and user satisfaction testing

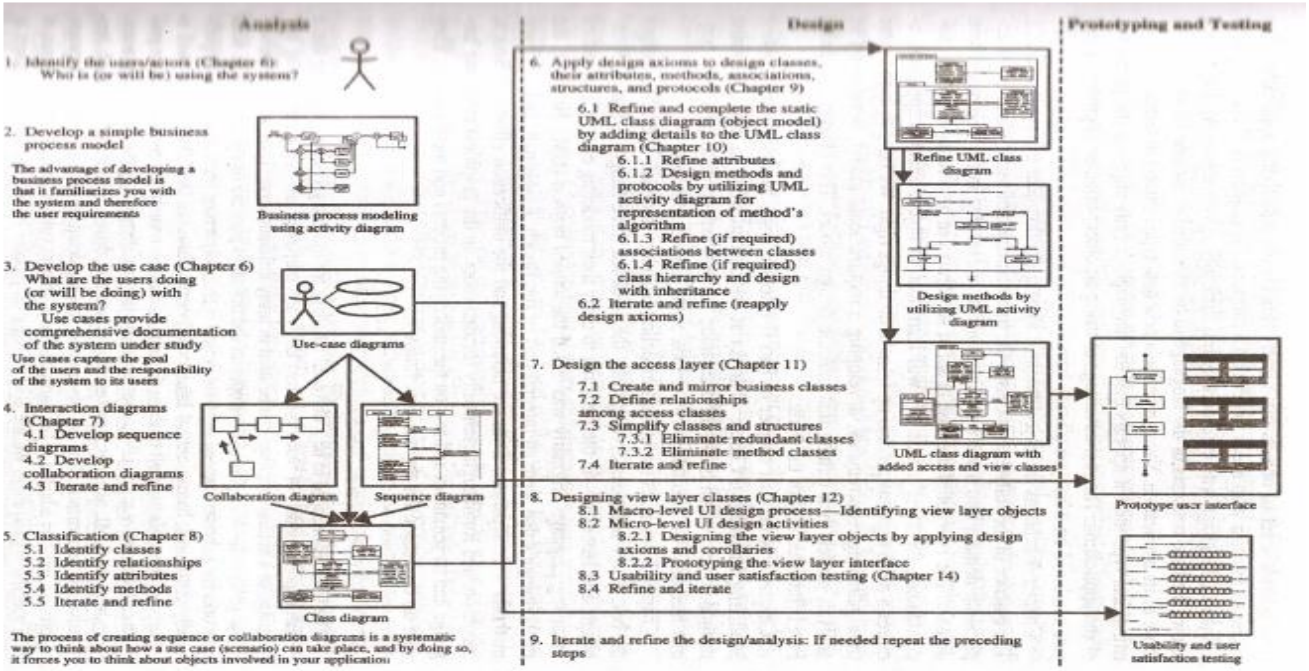
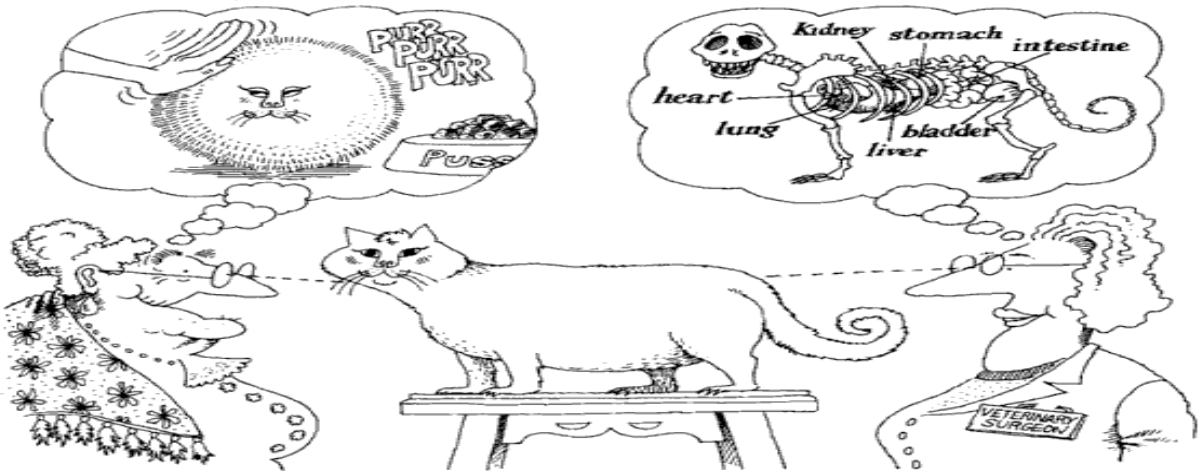


FIGURE 1–1
The unified approach road map.

10. What is meant by abstraction? What are the various kinds of abstraction?

- Abstraction is one of the fundamental ways that we as humans cope with complexity.
- A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary
- **Hoare** - "It arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences"
- **Shaw** - "a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others.
- **Grady Booch** - "A concept qualifies as an abstraction only if it can be described, understood, and analyzed independently of the mechanism that will eventually be used to realize it",
- *It denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.*
- An abstraction focuses on the outside view of an object
- It serves to separate an object's essential behavior from its implementation.



Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

Kinds of abstractions

- **Entity abstraction**
- An object that represents a useful model of a problem domain or solution-domain entity
- **Action abstraction**
- An object that provides a generalized set of operations, all of which perform the same kind of function
- **Virtual machine abstraction**
- An object that groups together operations that are all used by some superior level of control, or operations that all use some junior-level set of operations
- **Coincidental abstraction**
- An object that: packages a set of operations that have no relation to each other strive to build entity abstractions, because they directly parallel the vocabulary of a given problem domain.
- A ***client*** is any object that uses the resources of another object (known as the *server).*
- An ***invariant*** is some Boolean (true or false) condition whose truth must be preserved.
- For each operation associated with an object, we may define ***preconditions*** (invariants assumed by the operation) as well as ***post conditions*** (invariants satisfied by the operation).
- If a post condition is violated, this means that a server has not carried out its part of the contract, and so its clients can no longer trust the behavior of the server.
- An exception is an indication that some invariant has not been or cannot be satisfied.

Example

On a hydroponics farm, plants are grown in a nutrient solution, without sand, gravel, or other soils.

C++ code that capture our abstraction of a temperature sensor:

```
//Temperature in degrees Fahrenheit

typedef float Temperature;

// Number uniquely denoting the location of a sensor

typedef unsigned int Location;

class TemperatureSensor {

    Public:

    TemperatureSensor(Location);

    ~TemperatureSensor() ;

    void calibrate(Temperature actualTemperature);

    Temperature currentTemperature() const;

    private:

    …

};
```

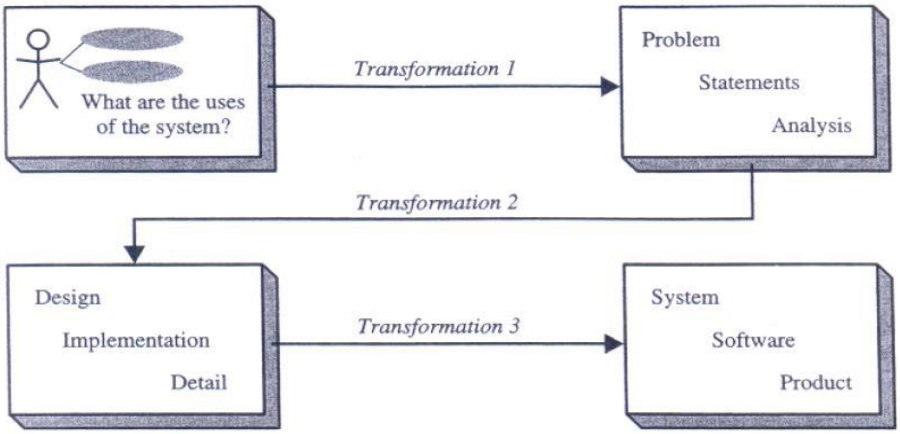
- The two typedefs, **Temperature** and **Location**, provide convenient aliases for more primitive types, thus letting us express our abstractions in the vocabulary of the problem domain5.
- **Temperature** is a floating-point type representing temperature in degrees Fahrenheit.
- The type **Location** denotes the places where temperature sensors may be deployed throughout the farm.
- The class **TemperatureSensor** captures our abstraction of a sensor itself; its representation isbhidden in the private part of the class.
- **TemperatureSensor** is defined as a class, not a concrete object, and therefore we must first create an *instance* so that we have something upon which to operate.
- For example, we might write:

```
Temperature temperature;

TemperatureSensor greenhouselSensor(l);

TemperatureSensor greenhouse2Sensor(2);

temperature = greenhouselSensor.currentTemperature();
```

i) Describe in detail about the software development process. (May 2015)

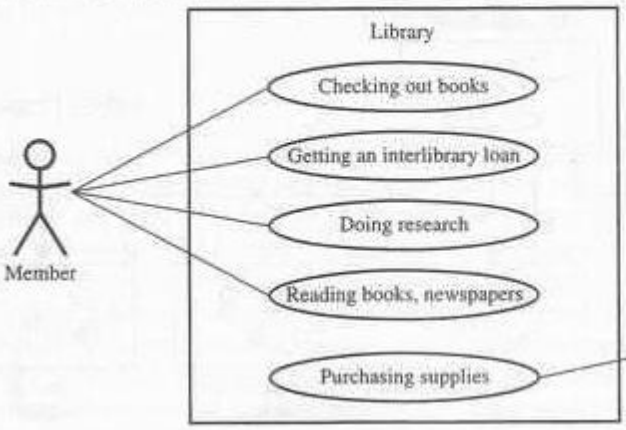
- The essence of the software development process that consists of analysis, design, implementation, testing, and refinement is to transform users' needs into a software solution that satisfies those needs.

- Develop a prototype to identify the features of the software.
- The prototype may help the users to comment on the usability of the product.
- The process can be divided into small, interacting sub processes.
- Each sub process must describe a specification of the inputs required, output to be generated.
    - **Transformation 1 – Analysis**
    - It translates the users' needs into system requirements and responsibilities
    - **Transformation 2– Design**
    - It begins with a problem statement and ends with the detailed design that can be transformed into an operational system
    - **Transformation 3 – Implementation**
    - It refines the detailed design into the system deployment that will satisfy the users' needs
    - It includes the equipment, procedures, etc
    - It represents embedding the software within its operational environment



- **Object oriented system development - Use case driven**
    - Use cases provide scenario for understanding the specific requirements.
    - Provides interaction between users and systems.
    - Captures the goal of the user and responsibility of the user to the system.
    - Shows various courses of the events to be performed.
    - **Use Case**
        - A sequence of transactions in a system.
        - Yields results of measurable values to an individual actor of the system.
        - Provides a special flow of events.
        - Group of various courses of events are represented as an use case class.
    - **E.g., Borrow book**
        - Whether a book is available in the library.
        - Whether the user is an member of the library**.**

FIGURE 4–6
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.



ii) List and explain the various system evaluation quality measures for software development.

- The software process transforms the users' needs through the application domain to a software solution that satisfies those needs
- Once the system / program exists, test it to see if it is free of bugs.
- High quality products must meet users' needs and expectations
- The product should attain this high quality with minimal or no defects.
- The main focus is on improving the products or services prior to delivery of the product to the customer.
- The goal of building high quality software is user satisfaction
- How do we determine when the system is ready for delivery to the customer?
- Is it now an operational system that satisfies users' needs?
- Does it pass an evaluation process?
- Basic approaches to system testing
- Correspondence
- It measures how well the delivered system matches the needs of the operational environment
- Validation is the task of predicting correspondence
- The correspondence can not be determined until the system is in place.
- Correctness
- It measures the consistency of the product requirements with respect to the design specification
- Verification is the exercise of determining correctness

11. i) Discuss in detail about different association relationship.(May 2016)
ii) Write about class hierarchy with suitable example.
Refer previous question answers.

UNIT – II

1. Explain in detail about Rumbaugh , Booch and Jacobson methods.(Dec 2016, May 2017)

OBJECT MODELING TECHNIQUE (OMT) Rumbaugh

- Describes a method for analysis, design and implementation of a system
- Fast intuitive approach
  - identify and model all the object
  - class , attributes , methods – defined easily
- Dynamic behavior of object is described – OMT dynamic model
  - Process description
  - Consumer – Producer association

PHASES – OMT

1. Analysis – outcomes- object , dynamic and functional model
2. System design – Basic architecture of the system
   - High level strategy decision

   Static
3. Object design – Design, Document Dynamic

   Functional
4. Implementation – Reusable, extendible code
   - OMT separates modeling into 3 phases viz.,
   - Object model – presented by object model & data dictionary
   - Dynamic model – state diagrams, event flow diagrams
   - Functional model – data flow & constraints

OBJECT MODEL

- Describes the structure of object
  - identity, relationships to other object
  - attributes, operations
- Uses object diagrams
  - object diagram – classes interconnected by associations
  - each class – set of object
  - association – establishes relationships among the classes

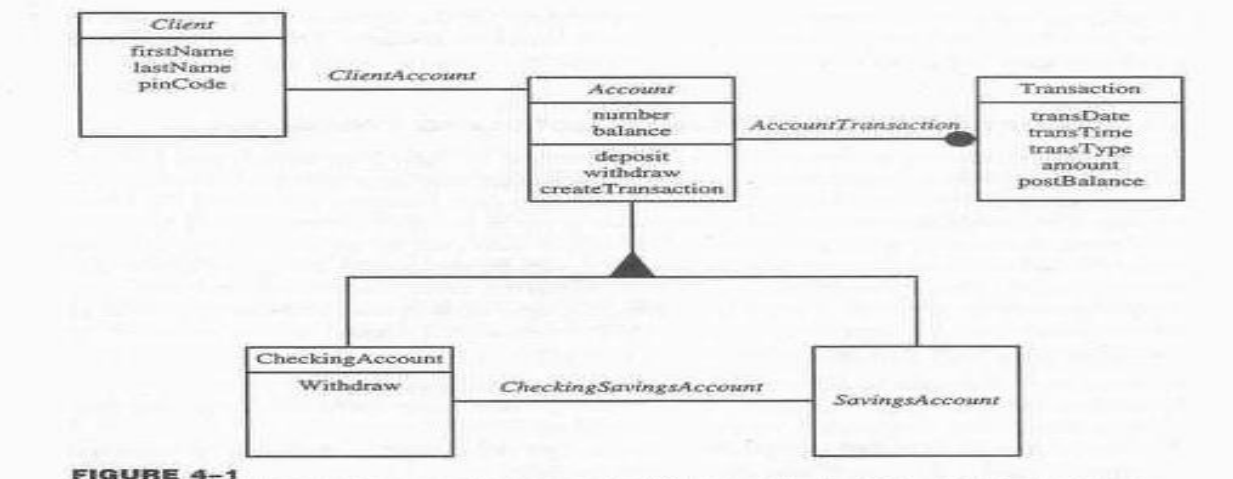


FIGURE 4–1

OMT – DYNAMIC MODEL

- Detailed and comprehensive model
- Depicts various states , transitions , events , actions
- State transition diagram – n/w of states , events
  - Each state receives one or more events at which time transition results in next state

Eg., dynamic model – Banking system

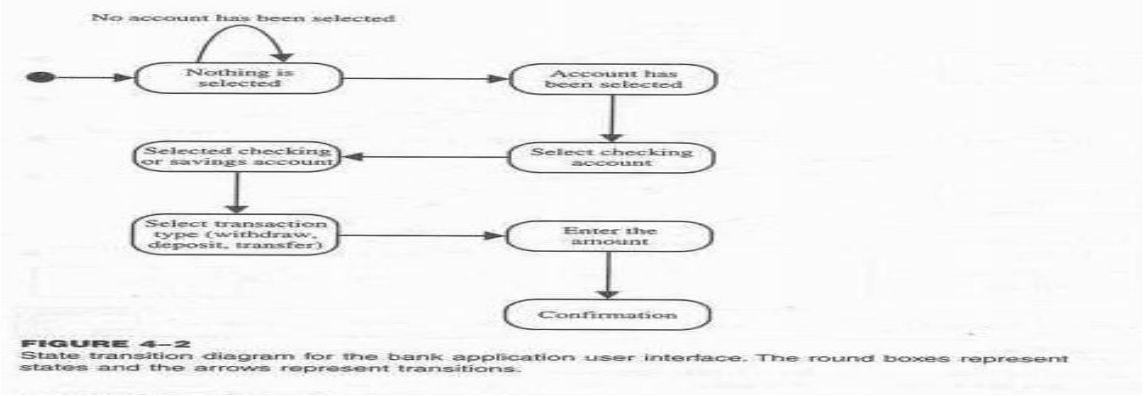


FIGURE 4–2
State transition diagram for the bank application user interface. The round boxes represent states and the arrows represent transitions.

OMT – FUNCTIONAL MODEL

- DFD shows flow of data between different processes
  - any function / process
  - data flow- direction of data element / movement
  - external entity-source / sink
  - data store-location where data are stored

Eg., ATM

OBJECT MODELING – BOOCH METHODOLOGY

State transition diagram – eg., alarm class

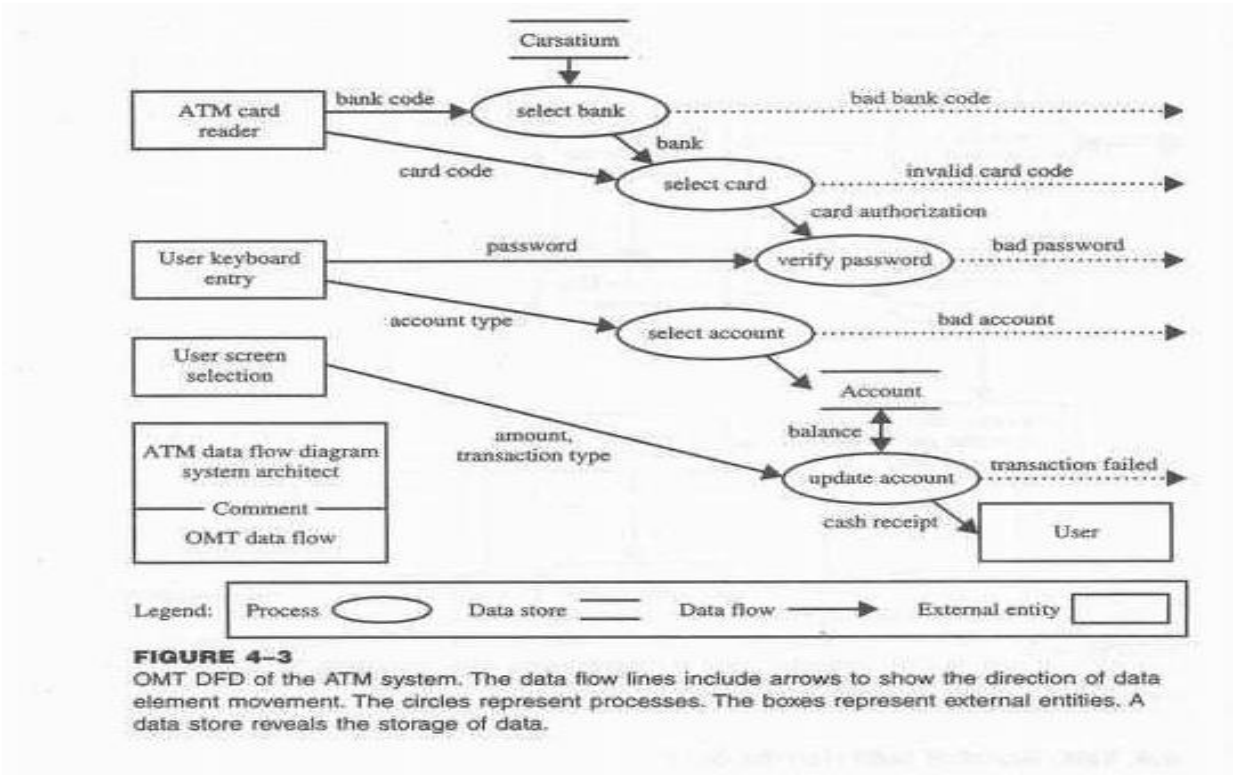
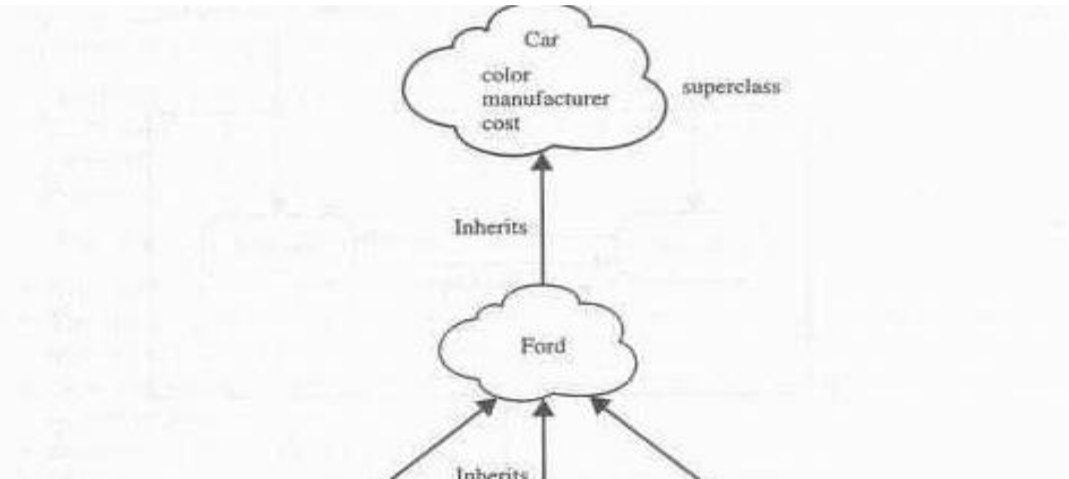


FIGURE 4–3
OMT DFD of the ATM system. The data flow lines include arrows to show the direction of data element movement. The circles represent processes. The boxes represent external entities. A data store reveals the storage of data.

MACRO DEVELOPMENT PROCESS

- Controlling framework for micro process
- Technical management of the system
- **Class diagram**



- **State Transition diagram**

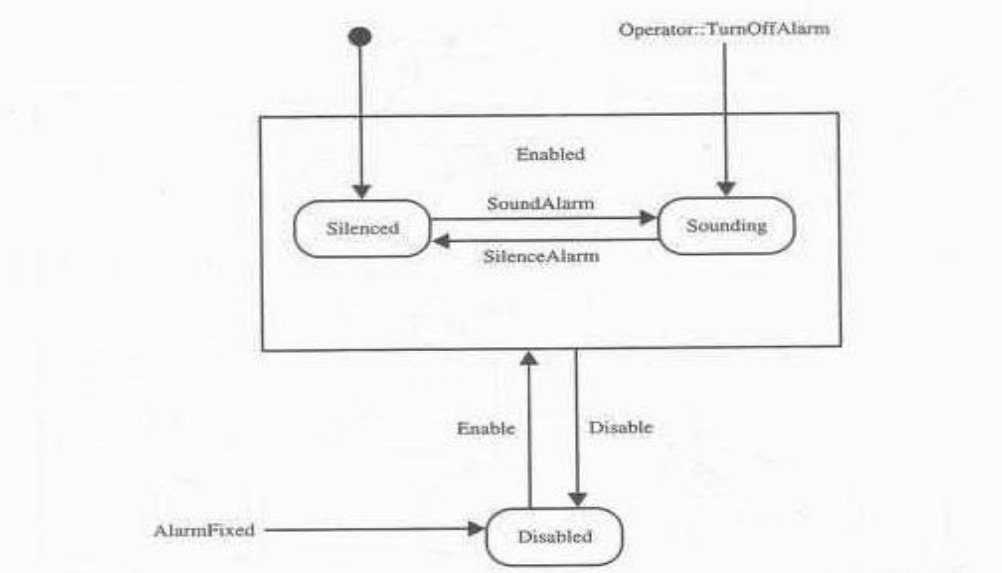


FIGURE 4–5
An alarm class state transition diagram with Booch notation. This diagram can capture the state of a class based on a stimulus. For example, a stimulus causes the class to perform some processing, followed by a transition to another state. In this case, the alarm silenced state can be changed to alarm sounding state and vice versa.

STEPS

1. **Conceptualization**
   - Establish core requirements of the system
   - Establish a set of goals
   - Develop a prototype
2. **Analysis & Development of the model**
   - user class diagrams – describe roles and relationships
   - user object diagram – describe designed behavior in terms of scenarios and interaction diagram
3. **Design / Create the system architecture**
   - use class diagram – what classes exist, their relationships
   - Object diagram – what mechanisms are used
   - Module diagram – map out where each class & object is declared
   - Process diagram – determines the allocation of process
4. **Evolution / Implementation**
   - Refine the system thru' many iterations
   - Produce a stream of s/w implementations
5. **Maintenance**
   Include localized changes to the system to add new requirements

MICRO DEVELOPMENT PROCESS

- Description of routine activities

STEPS

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

JACOBSON's METHODOLOGY

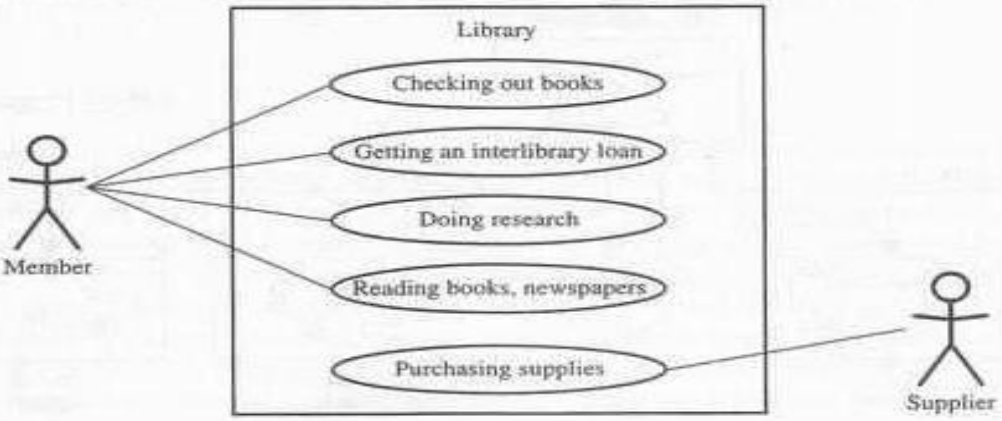
- OOSE - Object Oriented Software Engg
- Covers entire life cycle
- Traceability between different phases

USE CASES

- Scenarios for understanding system requirements
- An interaction between users and system
- Captures the goal of the user and responsibility of the system

Eg., Library system

FIGURE 4–6
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.



REQUIRED ANALYSIS

1. Non-format text with no clear flow of events
2. Text-easy to read with clear flow of events
3. Format style using pseudo code

- The use case description must contain –
    1. How and when use case begins and ends
    2. Interaction between use case and its actors
    3. When the interaction occurs and what is exchanged
    4. How and when the use case will need data stored in the system or will store the data in the system
    5. Exceptions – finding error during the run-time

2. Discuss generative, non generative patterns and anti patterns and list the guidelines on capturing patterns. Also discuss about abstract factory pattern.

Definition

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

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 captures solutions with a track record, not theories or speculation.

- **The solution is not obvious**
  The best patterns generate a solution to the 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 patterns

- They are patterns that not only describe a recurring problem, they tell us how to generate something and can be observed in the resulting system architectures.
- Alexander explains that the most useful patterns are generative: they are **dynamic**; they tell us what to do; they tell us how we shall, or may, generate them; they tell us too, that under certain circumstances, we must create them.

Non generative patterns

- They are **static** and passive: They describe recurring phenomena without necessarily saying how to reproduce them.

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/elements that should be recognizable on reading a pattern

1. 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.

2. Problem

A statement of the problem that describes its intent: the goals and objectives it wants to reach within the given context and forces.

3. Context

The preconditions under which the problem and its solution seem to recur and for which the solution is desirable.

4. Forces

A description of the relevant forces and constraints and how they interact or conflict with one another and with the goals we wish to achieve.

5. Solution

- Static relationships and dynamic rules describing how to realize the desired outcome.
- This often is equivalent to giving instructions that describe how to construct the necessary products.
- The description may encompass pictures, diagrams, and prose that identify the pattern's structure, its participants, and their collaborations, to show how the problem is solved.
- The solution should describe not only the static structure but also dynamic behavior.

6. 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.
- Examples help the reader understand the pattern's use and applicability.
- Visual examples and analogies often can be very useful.

7. Resulting context

- The state of configuration of the system after the pattern has been applied, including the consequences (both good and bad) of applying the pattern, other problems and patterns that may arise from the new context.
- It describes the post conditions 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.

8. 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.

9. 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.

10. 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.

Pattern Thumbnail

- Good pattern often begin with an abstract that provides a short summary or overview.
- This gives readers a clear picture of the pattern and quickly informs them of its relevance to any problems they may wish to solve.
- Sometimes such a description is called a **thumbnail sketch of the pattern** or a **pattern thumbnail**.

Anti patterns

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

1. Those describing a bad solution to a problem that resulted in a bad situation.
2. Those describing how to get out of a bad situation and how to proceed from there to a good solution.

Capturing Patterns

- 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 **reverses architecting**).

Guidelines for Capturing Patterns

1. Focus on practicability

Pattern should describe proven solutions to recurring problems rather than the latest scientific results.

2. Aggressive disregard of originality

Pattern writers do not need to be the original inventor or discoverer of the solutions that they document.

3. Nonananymous review

Pattern submissions are shepherded rather than reviewed. The pattern might be clarified or improved on.

4. Writers' workshops instead of presentations

Rather than being presented by the individual authors, the patterns are discussed in writers' workshops, open forums where all attending seek to improve the pattern presented by discussing what they like about them and the areas in which they are lacking.

5. 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 finishing form.

CREATIONAL PATTERNS - ABSTRACT FACTORY

INTENT

- Provide an interface for creating families of related/dependent objects without specifying their concrete classes

ALSO KNOWN AS

- Kit

MOTIVATION

- Uses interface toolkit that supports multiple look an feel standards such as MOTIF and presentation manager
- Different "look-and-feels" define different approaches and behaviors for user interface "widgets" like scroll bars, windows and buttons
- Objective ensure portability across look-and-feel standards

1. Define an abstract widget factory class that declares an interface for creating each basic kind of widget
2. Define an abstract class for each kind of widget

3. Concrete subclasses implement widgets for specific look-and-feel standards
4. Widget factory's interface has an operation that returns a new widget object for each abstract widget class

- Clients call these operations to obtain widget instances, but clients aren't aware of the concrete classes they are using
- Clients stay independent of the prevailing look and feel
- A widget factory enforces dependencies between the concrete widget classes

APPLICABILITY

- Use the abstract factory pattern when,
    1. A system should be independent of how its products are created, composed and represented
    2. A system should be configured with one of multiple families of products
    3. A family of related product objects is designed to be used together and constraints are enforced
    4. To provide a class library of products in which only the interfaces are to be revealed and their implementation

PARTICIPANTS

1. Abstract factory – widget factory declares an interface for operation that create abstract product objects
2. Concrete factory(MOTIF widget factory, OM widget factory) implements the operations to create concrete product objects
3. It promotes consistency among products – An application uses objects from only one family at a time
4. Supporting new kinds of products is difficult – abstract interface fixes the set of products that can be created
5. Supporting new kinds of products requires extending the factory interface, which involves changing the abstract class and all of its subclasses

IMPLEMENTATION

1. Factories as singletons – only one instance of concrete factory / product family is needed
2. Create the products
    - Concrete product subclasses actually create them

Eg., MOTIF window, MOTIF scroll bar

- Concrete family can be implemented using prototype pattern
- Prototype based approach eliminates the need for a new factory class

Eg., prototype based factory in small talk

dictionary : part catalog

Method :

make : part name

(part catalog at : part name)copy

Adding parts :

add part : part template named : part name

part catalog at : part name put:part template

3. **Define extendable factories**
    - Add a parameter to operations that create object

Eg., make- include kind of object to create

- Addition of parameter can be done with factory method pattern

SAMPLE CODE

```
Class maze factory {
    public :
        maze factory ()
        virtual maze * make maze const
            { return new maze () }
        virtual mall * make mall () const
            { return new mall () }
        virtual room * make room(int n) const
            { return new room (n) }
```

KNOWN USES

- Achieves portability across various window systems
  eg., x windows, seen view

RELATED PATTERNS

- Factory method
- Prototype
- Singleton

3. Prepare class diagram showing relationship among the classes. Include association, aggregation and generalization. Also add attributes and operations to the class diagram.

- The purpose of identifying the relationships among classes and objects is to solidify the boundaries of and to recognize the collaborators with each abstraction identified earlier in the micro process.
- This activity formalizes the conceptual as well as physical separations of concern among abstractions begun in the previous step.
- Expressing the existence of an association identifies some semantic dependency between two abstractions, as well as some ability to navigate from one entity to another.
    - As part of design, we apply this step to specify the collaborations that: form the mechanisms of our architecture, as well as the higher-level clustering of classes into categories and modules into subsystems.
- As implementation proceeds, we refine relationships such as associations into more
- Implementation-oriented relationships, including instantiation and use.
    - Each association has two association ends; each end is attached to one of the classes in the association.
    - An end can be explicitly named with a label.
    - This label is called a role name. (Association ends are often called roles.)
    - An association end also has multiplicity, which is an indication of how many objects may participate in the given relationship.
    - interface for an Order class:
        - class Order {
        - public Customer getCustomer();
        - public Set getOrderLines();
        - class Customer {
        - private Set _orders;
    - If navigability exists in only one direction, we call the association a **unidirectional association. A bidirectional association** contains navigability in both directions.
    - An association represents a permanent link between two objects

4. Develop the system for the inventory control management based on Booch Methodology with the following requirements:

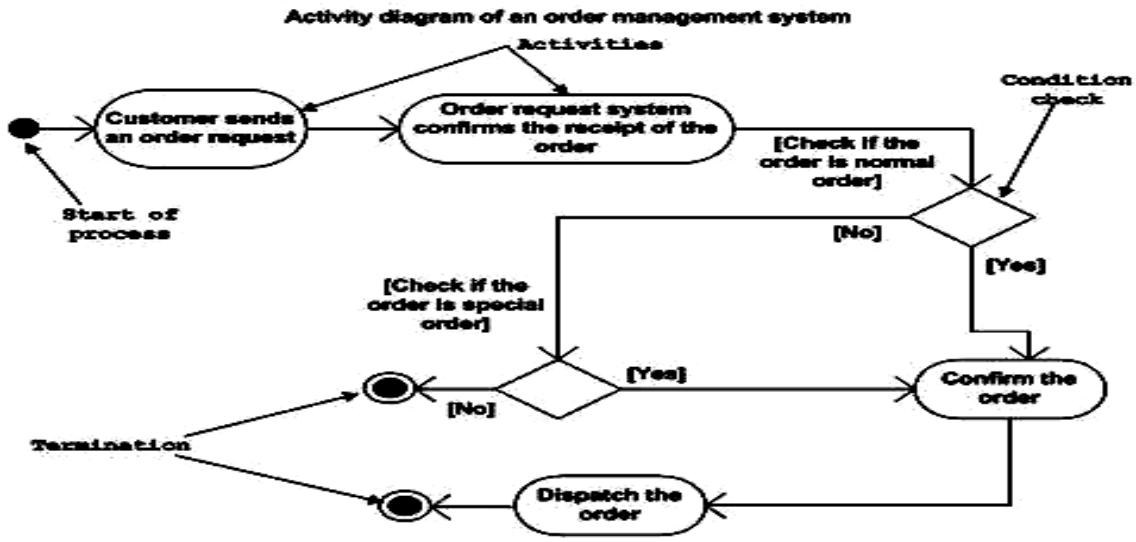
    i) Keep track of the stocks in the stores.

    ii)Tracking the orders as they are received.

- The activity diagram is made to understand the flow of activities and mainly used by the business users.

- The following diagram is drawn with the four main activities:
- Send order by the customer
- Receipt of the order
- Confirm order
- Dispatch order

- After receiving the order request condition checks are performed to check if it is normal or special order.
- After the type of order is identified dispatch activity is performed and that is marked as the termination of the process.



(i) Tracking the orders as they are received.

- one transition can be taken out of a given state, so we intend the guards to be mutually exclusive for any event.

1. If we have not checked all items, we get the next item and return to the Checking state to check it.
2. If we have checked all items and they were all in stock, we transition to the Dispatching state.
3. If we have checked all items but not all of them were in stock, we transition to the Waiting state.

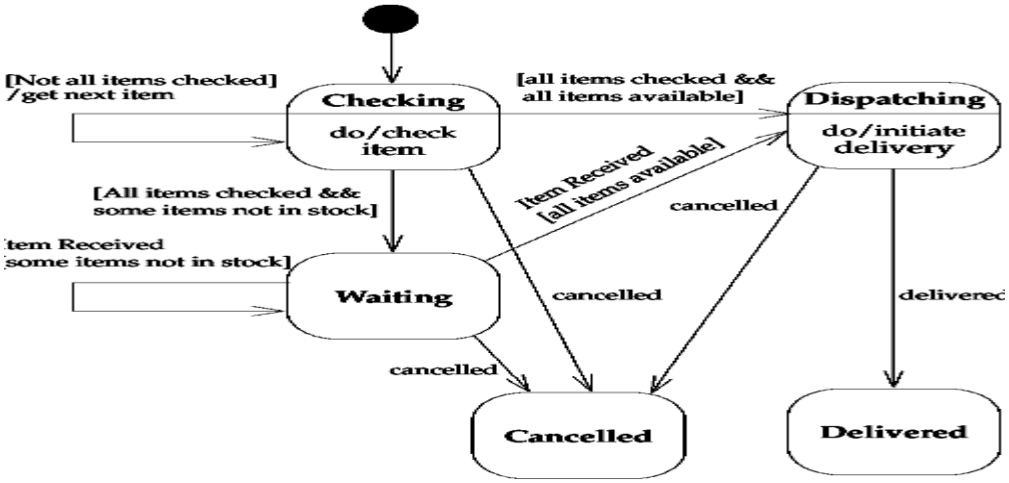


Figure (a) State Diagram without Super states

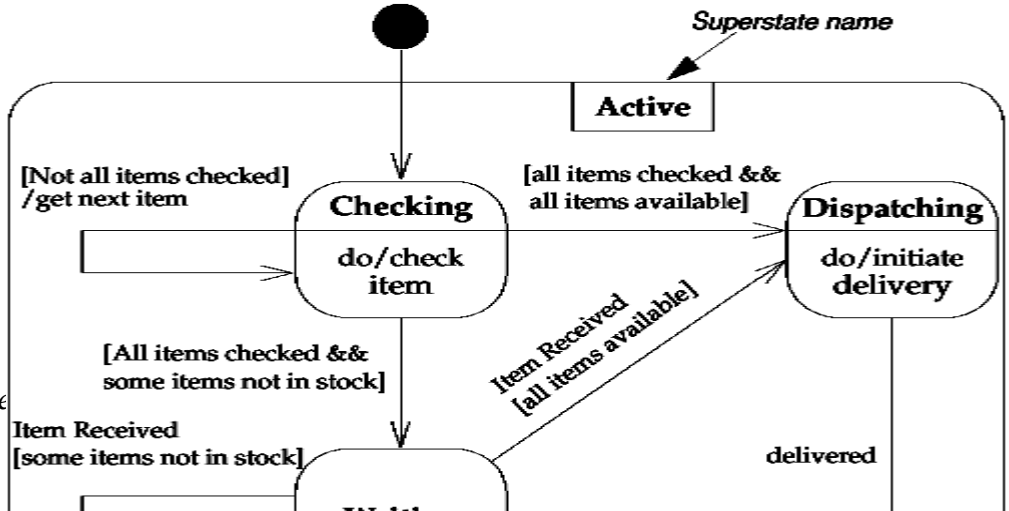
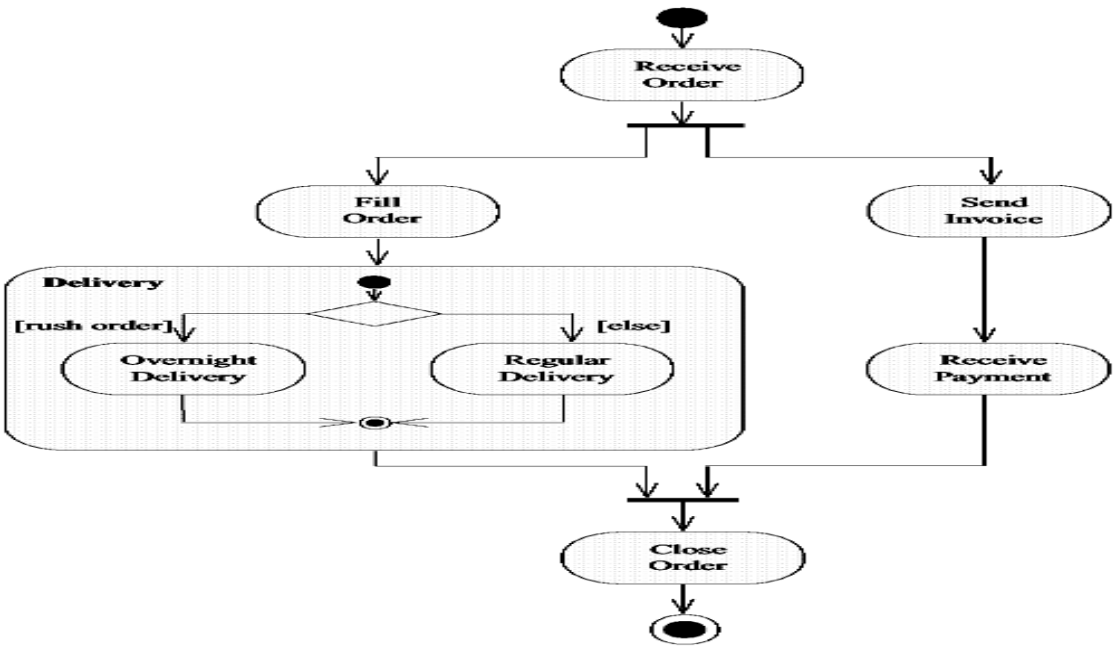


Figure (b) State Diagram with Super states

(ii) Deliver the orders.



4. Explain in detail about the unified approach in object oriented methodologies.

- The unified approach (UA) establishes a unifying and unitary framework for utilizing the UML to describe, model and document the software development process.
- It combines the best practices, processes, methodologies and guidelines along with UML notations for better understanding of OO concept and system development.
- It involves
- Use- case driven development
- Object-oriented analysis
- Object - oriented design
- Incremental development and prototyping
- Continuous testing
- The methods and technology employed include
- UML for modeling
- Layered approach
- Repository for object - oriented system development
- Patterns and frameworks
- Component - based development
- It allows iterative development by allowing to move between design and the modeling or analysis phases.
- It makes backtracking easy and departs from teh linear waterfall process.
- **Object - oriented Analysis**
- Analysis is the process of extracting the needs of a system to satisfy the users' requirements.
- The goal of OOA is to understand the domain of the problem and the system's responsibilities by understanding how the users use / will use the system.
- The models concentrate on describing what the system does rather than how it does it.
- Separating the behaviour of a system from the way it is implemented requires viewing the system from the user's perspective.
- **Steps in OOA process**
- Identify the actors
- Develop a simple business process model using UML activity diagram
- Develop the use case
- Develop interaction diagram
- Identify classes

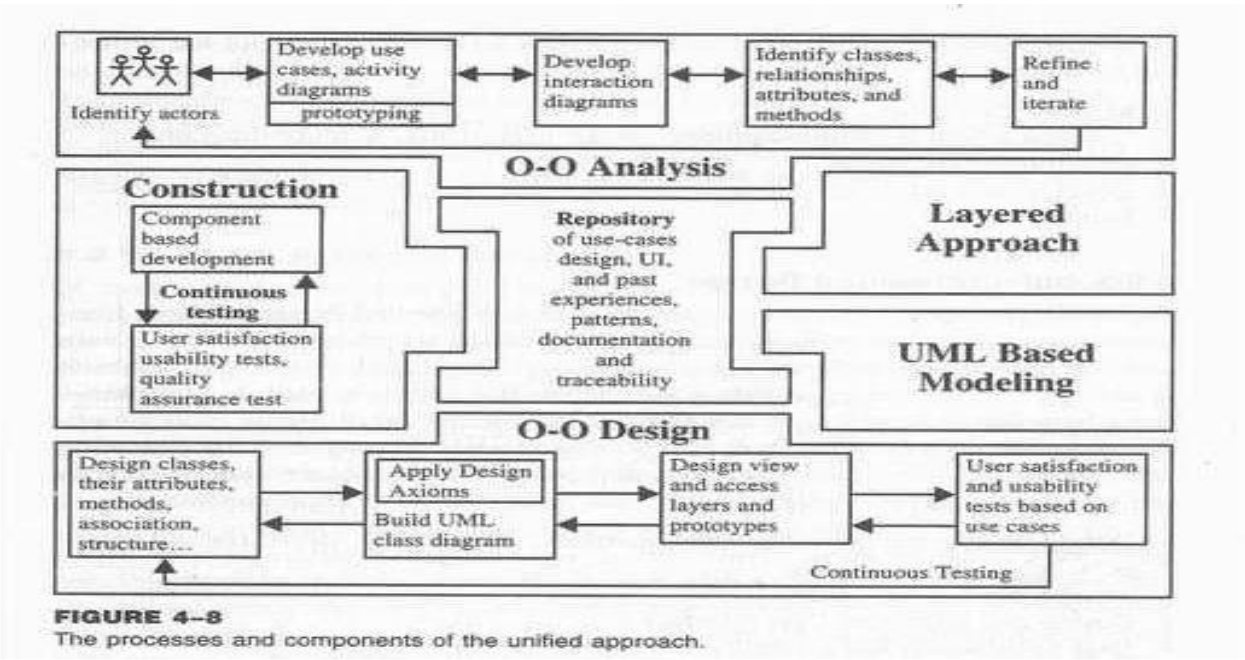


FIGURE 4–8
The processes and components of the unified approach.

- **Object - Oriented Design (OOD)**
- UA combines the analysis and interaction diagrams, object diagrams, domain models for the design of the application using object oriented design.
- Such designs can be developed in a way that they ae traceable across requirements, analysis, design, coding and testing.
- OOD process consists of
- Designing classes, their attributes, methods, associations, structures and protocols
- Apply design axioms
- Design the access layer
- User satisfaction and usability test based on the usage / use cases
- Iterate and refine the design
- Iterative development and continuous testing
- Iterate the development process until the user is satisfied with the system development.
- Apply testing effectively so that it uncovers the design weaknesses and suggests ways with which the system's quality is improved.
- Continue this refining cycle through the development process until the developer is satisfied with the results.
- During this iterative process, the prototypes will be incrementally transformed into the actual application.
- **Modeling based on the Unified Modeling Language**
- The UML merges the best of the notations used by the analysis and design methodologies
- The UA uses the UML to describe and model the analysis and design phases of system development.
- UA Proposed Repository
- Create a repository that allows the maximum reuse of previous experience and previously defined objects, patterns, frameworks and user interfaces.
- The UA's assumption is that creating additional applications will require no more than assembling components from the library.
- Microsoft repository, VisualAge, PowerBuilder, Visual C++ and Delphi provide such capability.
- These repositories contain all objects that have been previously defined and can be reused for newer applications.
- If a new object is designed, it has to be stored in the main repository for future use.
- **Layered Approach to Software Development**
- Systems can be developed with CASE tools or C/S architecture / Two - tier architecture

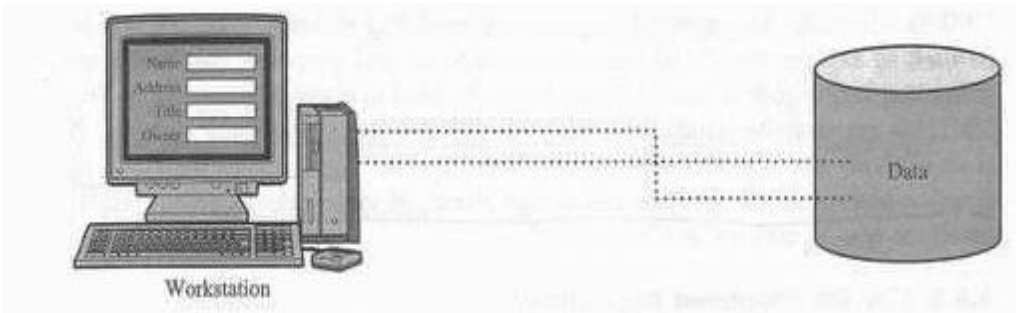


FIGURE 4–9
Two-layered architecture: interface and data.

- In a two - layered system, user interface screens are tied to the data through routines that sit directly behind the screen
- The routines required to access the data must exist within every screen.
- Any change to the business logic must be accomplished in every screen that deals with that portion of the business.

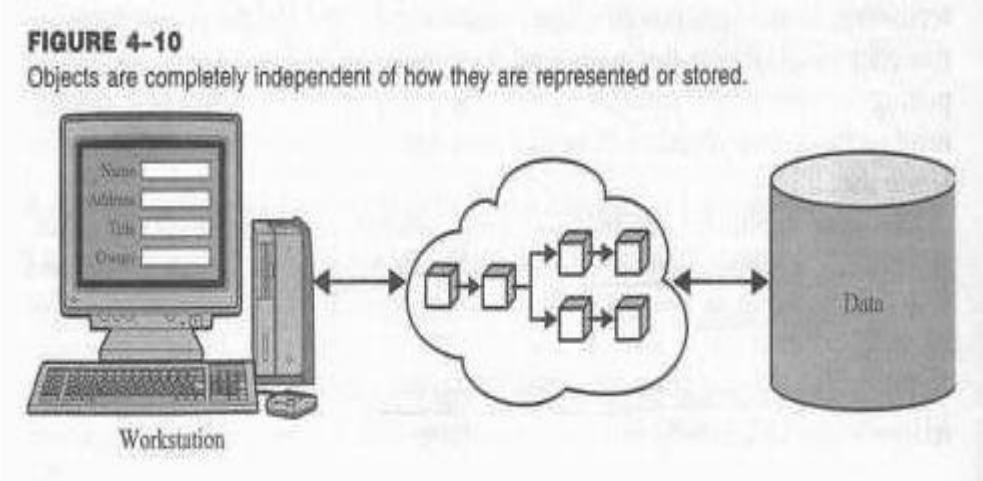
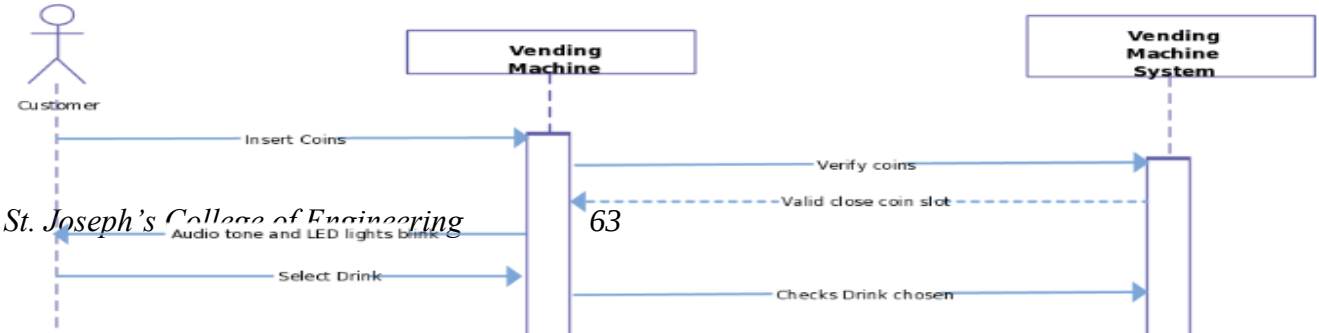


FIGURE 4-10
Objects are completely independent of how they are represented or stored.

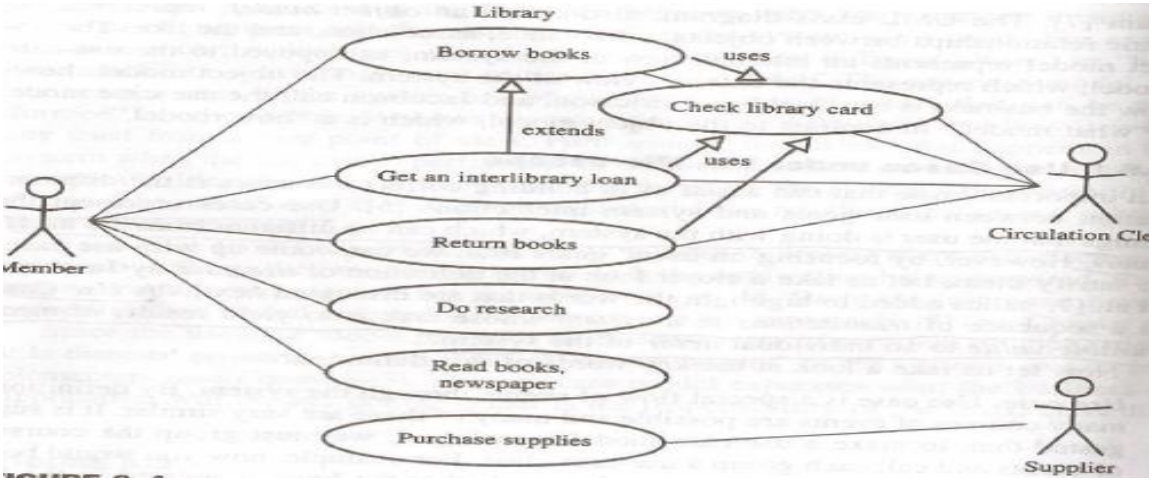
6. Draw the following diagrams and explain
   i) Sequence diagram for ticket vending machine in railway station.
   ii) Use case and activity diagrams for the library information system

- **Steps**
- The customer inserts coins in to the vending machine.
- The vending machine system verifies the coins.
- If the amount is correct and valid, it closes the coin slot
- The audio tone and LED light blinks so that the customer can select the place of the journey.
- The vending machine checks the availability of the ticket
- It confirms the chosen place.
- It dispenses the ticket.
- It turns LED lights off except for the chosen place.
- It also educes the ticket count by one.
- It checks the ticket count.



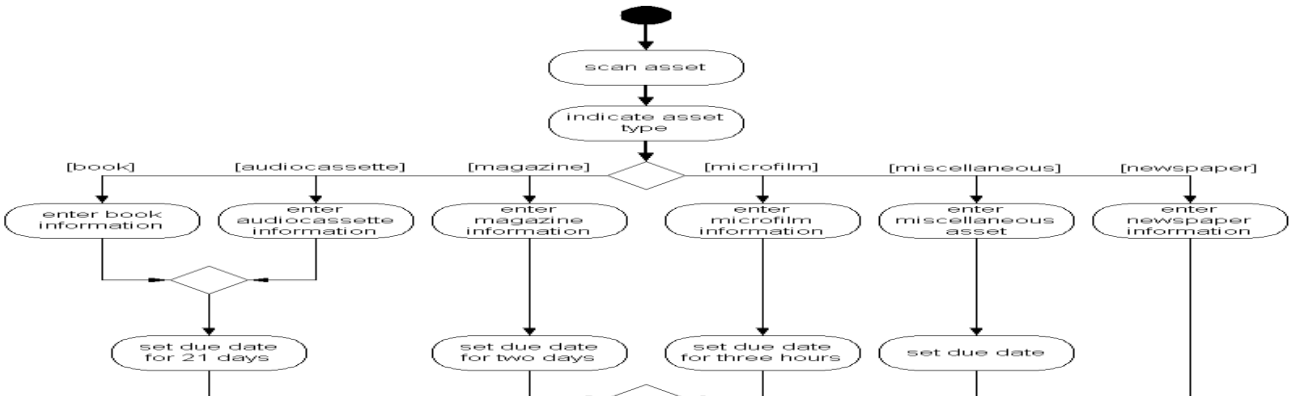
ii) Use case and activity diagrams for the library information system

- The primary actors are member, circulation clerk and the secondary actor is the supplier.
- The scenario are borrow books, get an interlibrary loan, return books, do research, read books, newspaper, check library card, purchase supplies
- The scenario get an interlibrary loan can have an additional extends association of another scenario borrow books.
- The " get an interlibrary loan", "borrow books" and " return books" have uses association " check library card"



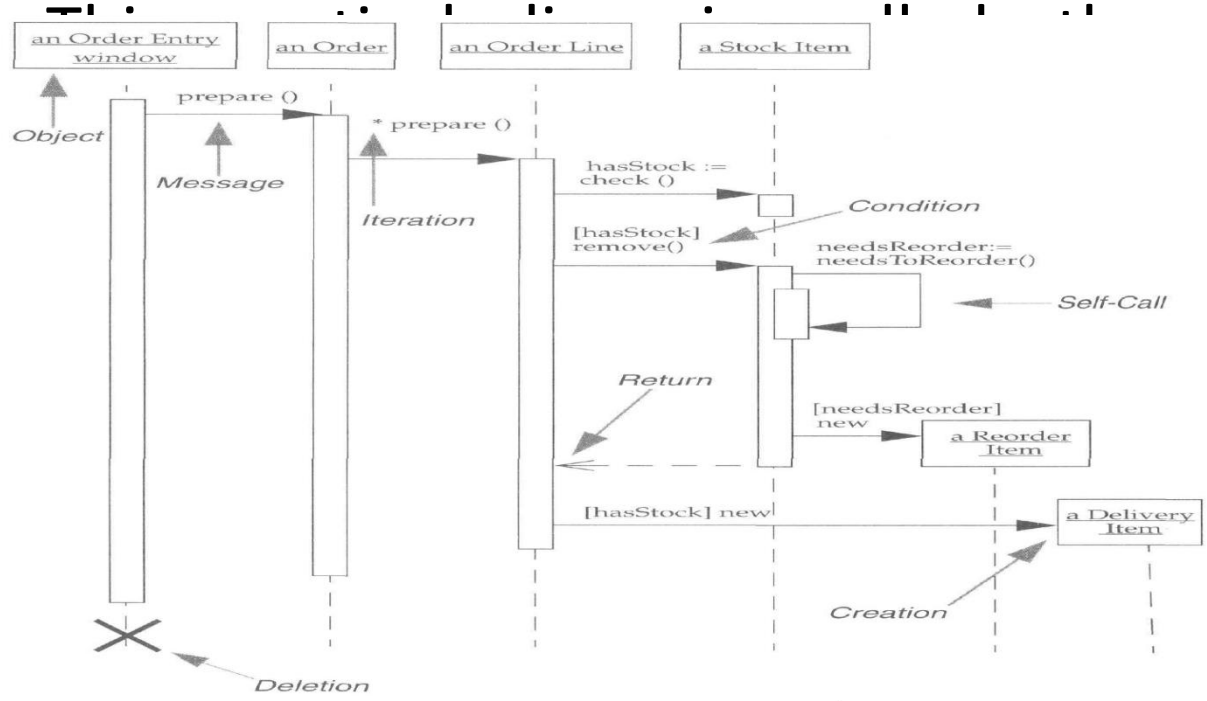
Activity diagram

- Start
- The assets available in the library are checked based on the asset type
- Check whether the asset type is "books" , If so, enter book information, set the due date for 21 days
- If the asset type is " audio cassettes", enter its relevant details, set the due date for 21 days
- If the asset type is "magazine", enter those details, set the due date for two days
- If the asset type is " microfilm:, enter those details, set the due date for three days
- Otherwise, enter those details and set the due date
- If the asset type is newspaper, enter the details
- Terminate and end



7. Describe in detail about the interaction diagrams with an example.

- **Interaction diagrams**
- Models that describe how groups of objects collaborate in some behavior.
- Captures the behavior of a single use case.
- The diagram shows a number of example objects and the messages that are passed between these objects within the use case.
- **Sequence diagrams**
- Within a sequence diagram, an object is shown as a box at the top of a dashed vertical line

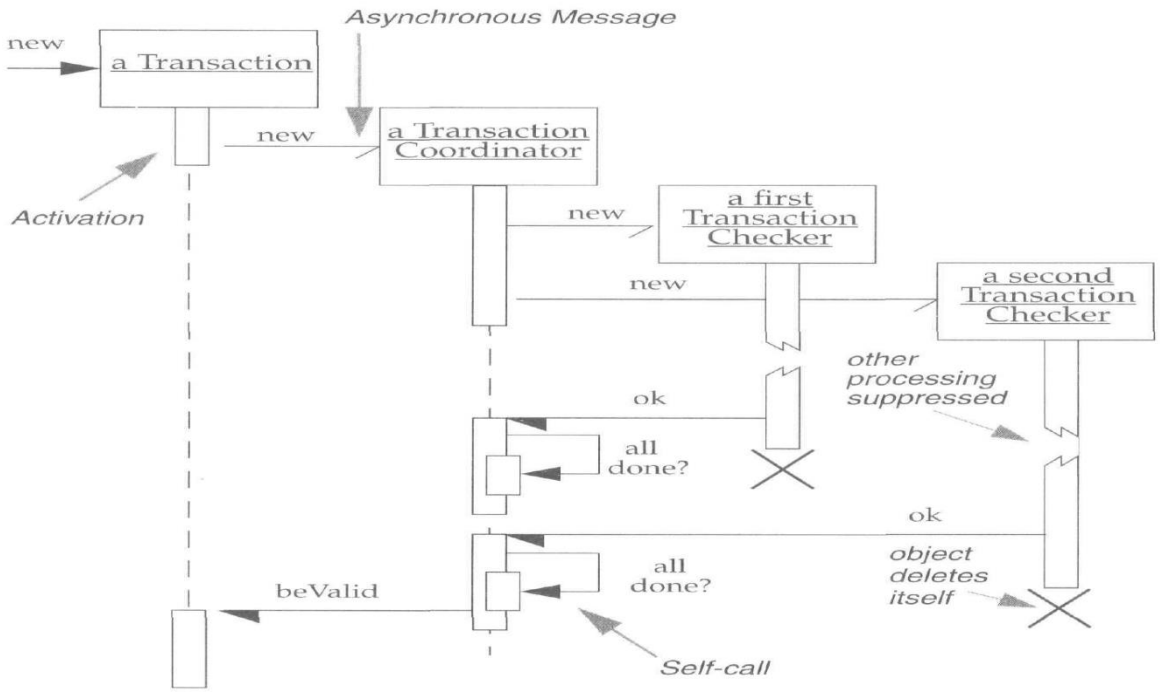


The order in which these messages

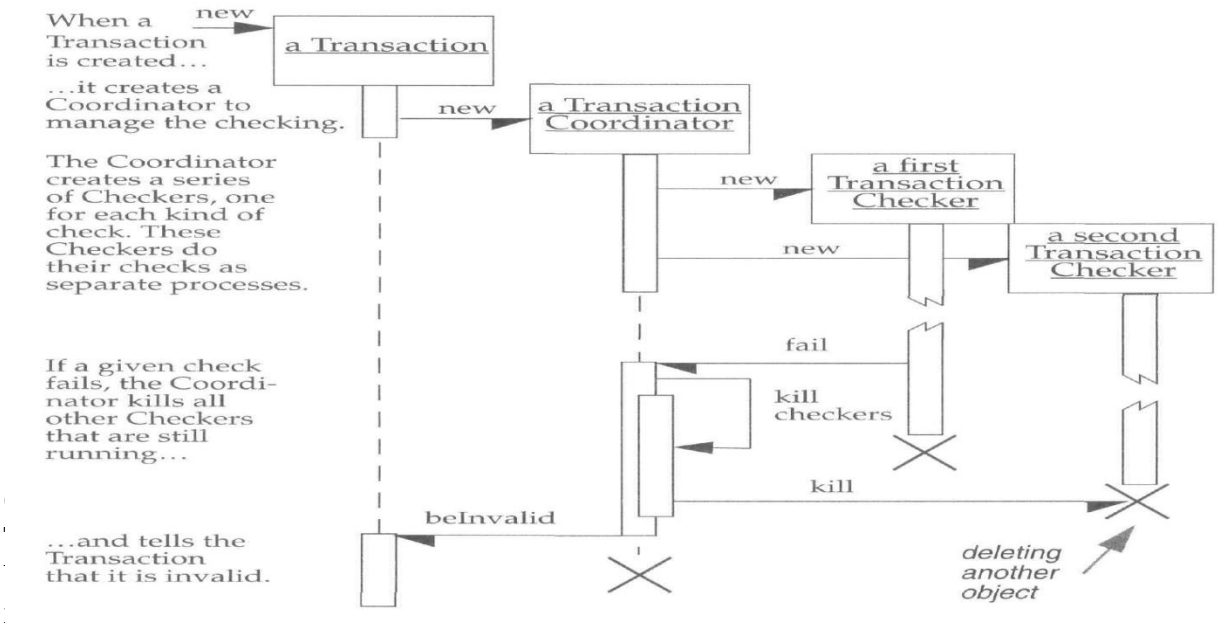
- This vertical line is called the object's **lifeline**
- Each message is represented by an arrow between the lifelines of two objects.
- The order in which these messages occur is shown top to bottom on the page.
- **Self-call, a message that an object sends** to itself, by sending the message arrow back to the same lifeline.
- There is a **condition indicates when a message is sent**
- The **iteration marker, s**hows that a message is sent many times to multiple receiver objects, as would happen when you are iterating over a collection.
- You can show the basis of the iteration within brackets, such as ***[for all order lines].**
- The above figure includes a **return, indicates a return from a message** not a new message
- Returns differ from the regular messages in that the line is dashed.
- Concurrent Processes and activation
- When a Transaction is created, it creates a Transaction Coordinator to coordinate the checking of the Transaction.
- This coordinator creates a number (in this case, two) of Transaction Checker objects, each of which is

responsible for a particular check.

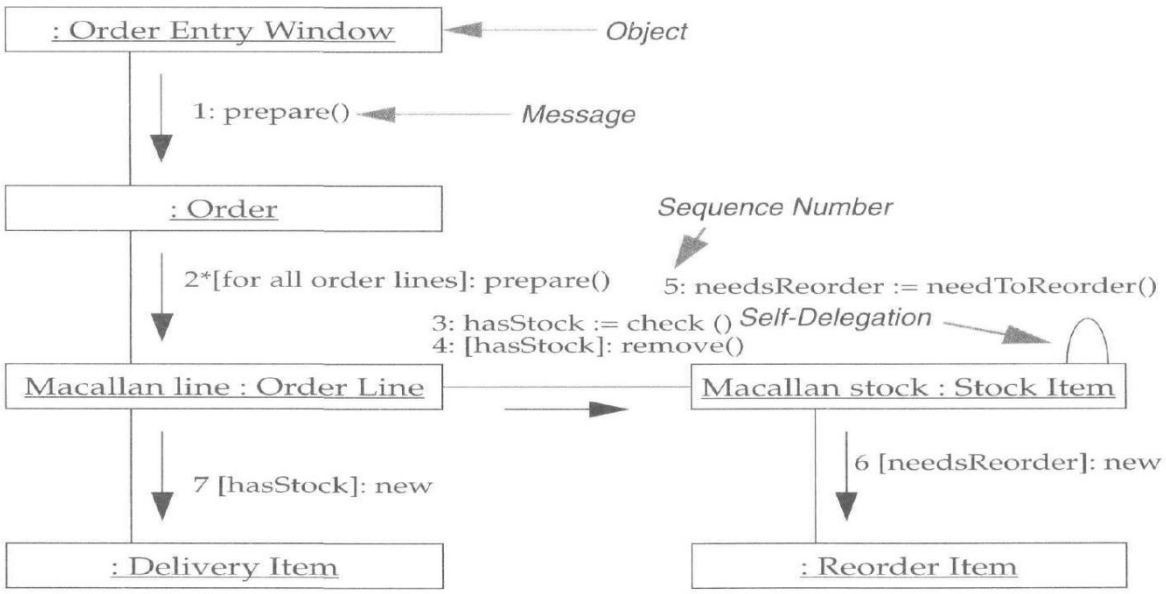
- When a Transaction is created, it creates a Transaction Coordinator to coordinate the checking of the Transaction.



- This coordinator creates a number (in this case, two) of Transaction Checker objects, each of which is responsible for a particular check.
- An asynchronous message can do one of three things:
- 1. Create a new thread, in which case it links to the top of an activation
- 2. Create a new object
- 3. Communicate with a thread that is already running
- Object deletion is shown with a large X. Objects can self-destruct
- They can be destroyed by another message

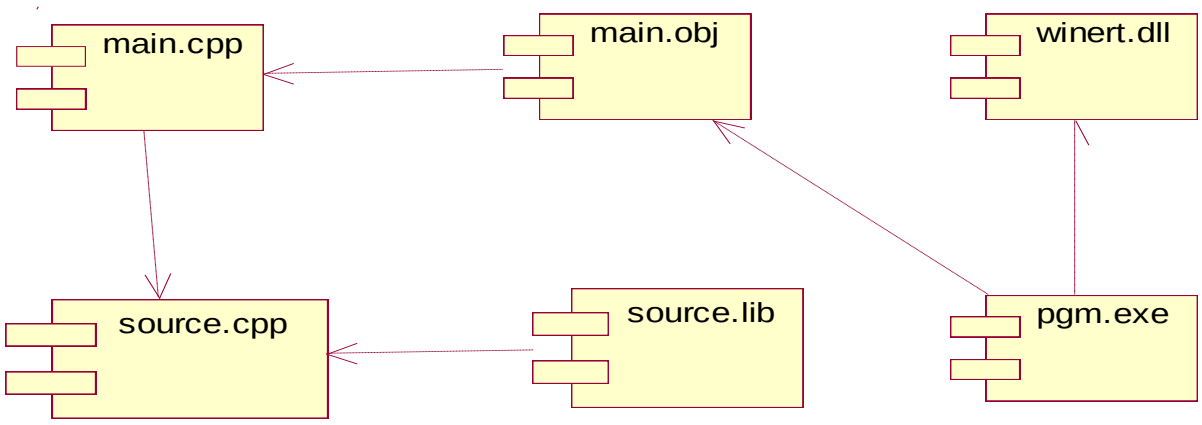


- r other



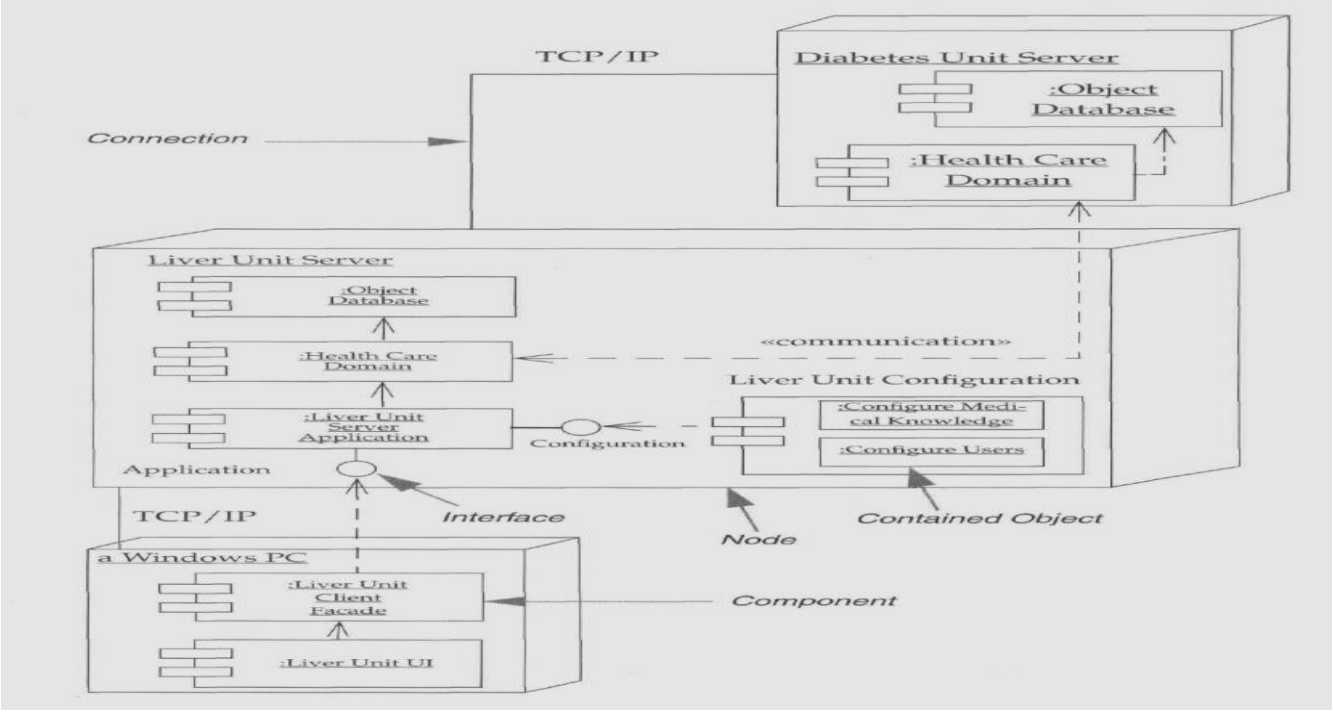
7. Describe in detail about the interaction diagrams with an example.

- **Component diagram**
- It shows the various components in a system and their dependencies**.**
- A component represents a physical module of code
- The dependencies among the components show how changes to one
- component may cause other components to change.
- Used to describe components within a software system.
- Components in UML are physical such as source code file, Libraries, dynamic components or executable programs.
- The six software components are
    - Main. Cpp Source. Lib
    - Main. Obj Source. Cpp
    - Wincrt. Dll Program.exe



- ***Deployment Diagrams***

- A deployment diagram shows the physical relationships among software and hardware components in the delivered system.
- Each node on a deployment diagram represents some kind of computational unit—in most cases, a piece of hardware.
- The hardware may be a simple device or sensor, or it could be a mainframe.

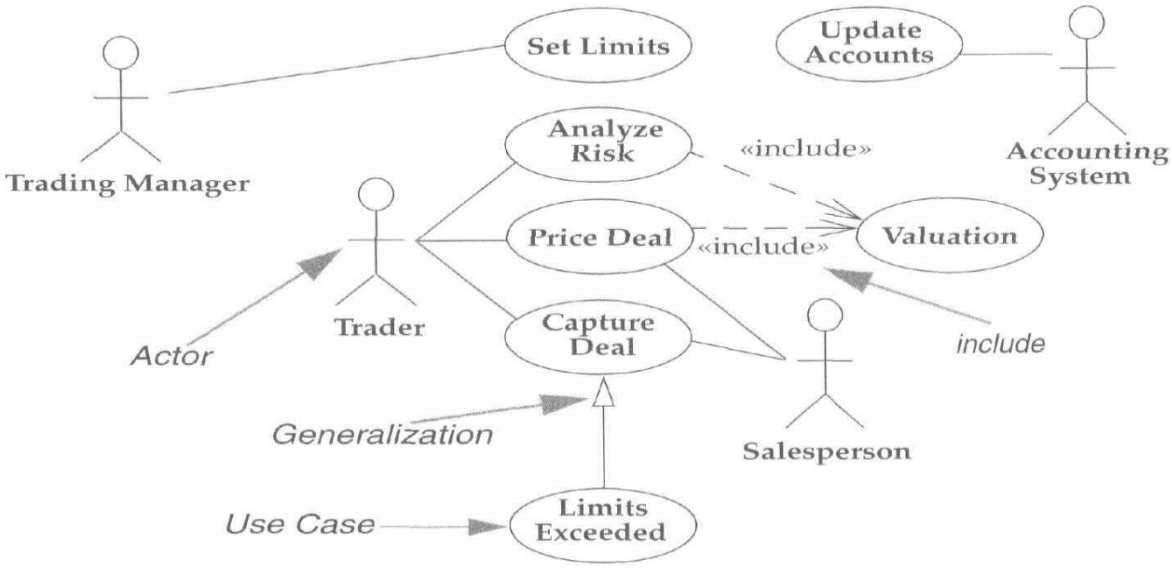


7. Explain in detail about the implementation diagrams/ physical diagrams with examples.
   Refer previous questions

9. Discuss in detail about the following with suitable examples.(MAY 2015)
   i) Use case diagram
   ii) Class diagram.

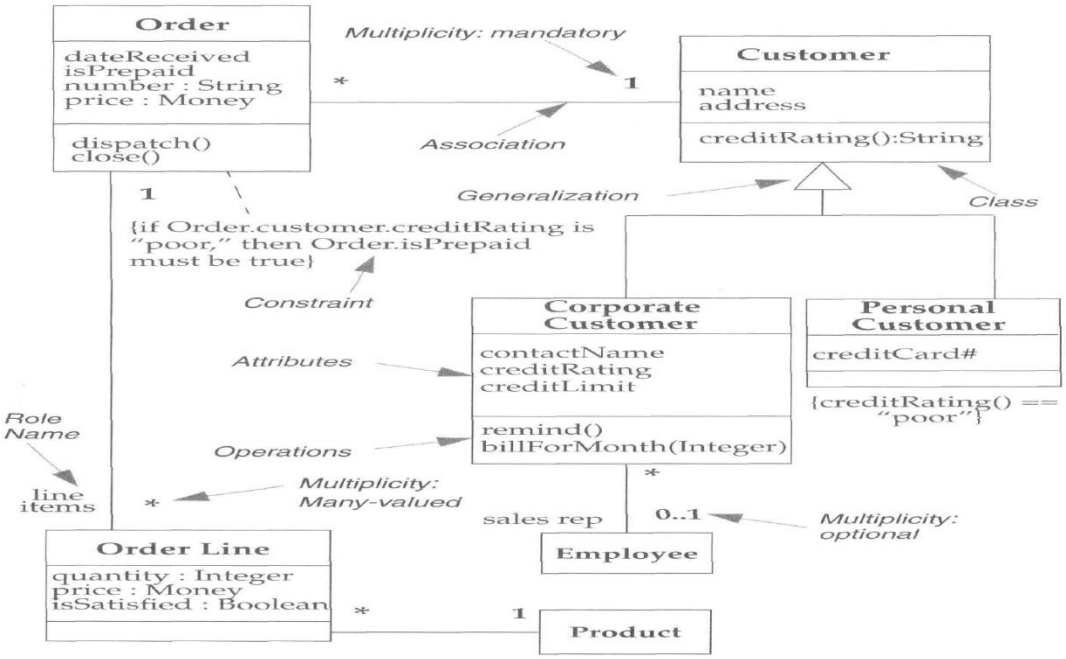
- A scenario is a sequence of steps describing an interaction between a user and a system.
- The customer browses the catalog and adds desired items to the shopping basket.
- When the customer wishes to pay, the customer describes the shipping and credit card information and confirms the sale.
- The system checks the authorization on the credit card and confirms the sale both immediately and with a follow-up email.
- Variations on the sequence
- **Buy a Product**
- 1. Customer browses through catalog, selects items to buy
- 2. Customer goes to check out
- 3. Customer fills in shipping information (address; next-day or 3-day delivery)
- 4. System presents full pricing information, including shipping
- Customer fills in credit card information
- 6. System authorizes purchase
- 7. System confirms sale immediately
- 8. System sends confirming email to customer
- **Alternative**: *Authorization Failure*
- At step 6, system fails to authorize credit purchase
- Allow customer to re-enter credit card information and re-try
- **Alternative**: *Regular Customer*
- 3a. System displays current shipping info, pricing information, last 4 digits of credit card information

- 3b. Customer may accept or override these defaults
- Return to primary scenario at step 6



(ii) Class diagram.

- A class diagram describes the types of objects in the system.
- Various kinds of static relationships that exist among them.
- principal kinds of static relationships:
- Associations
- for example, a customer may rent a number of videos
- subtypes (a nurse is a kind of person)



10. i) What are the different models that are present in OMT? Explain them in detail.(MAY 2015)

ii) What is pattern ? List and explain the essentials components of a pattern to recognize it properly.

OBJECT MODELING TECHNIQUE (OMT) Rum Baugh

- Describes a method for analysis, design and implementation of a system
- Fast intuitive approach
  - identify and model all the object
  - class , attributes , methods – defined easily
- Dynamic behavior of object is described – OMT dynamic model
  - Process description
  - Consumer – Producer association

PHASES – OMT

2. Analysis – outcomes- object , dynamic and functional model
4. System design – Basic architecture of the system
   - High level strategy decision

   Static

5. Object design – Design, Document Dynamic

   Functional

5. Implementation – Reusable, extendible code
   - OMT separates modeling into 3 phases viz.,
   - Object model – presented by object model & data dictionary
   - Dynamic model – state diagrams, event flow diagrams
   - Functional model – data flow & constraints

OBJECT MODEL

- Describes the structure of object
  - identity, relationships to other object
  - attributes, operations
- Uses object diagrams
  - obj diagram – classes interconnected by associations
  - each class – set of object
  - association – establishes relationships among the classes

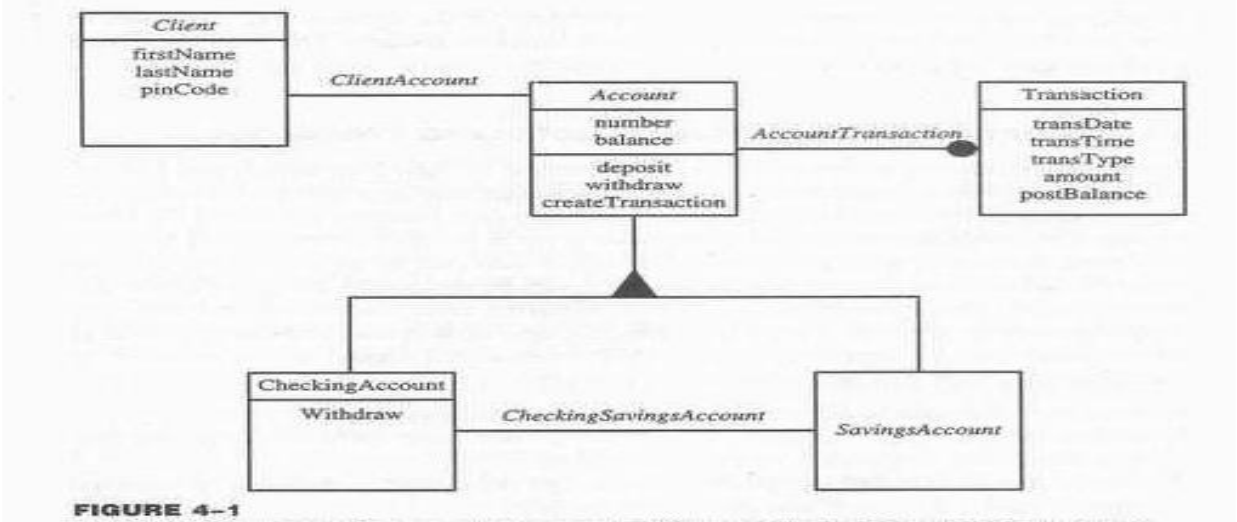


FIGURE 4–1
The OMT object model of a bank system. The boxes represent classes and the filled triangle represents specialization. Association between Account and transaction is one too many; since one account can have many transactions, the filled circle represents many (zero or more). The relationship betwen Client and Account classes is one to one: A client can have only one account and account can belong to only one person (in this model joint accounts are not allowed).

OMT – DYNAMIC MODEL

- Detailed and comprehensive model
- Depicts various states , transitions , events , actions
- State transition diagram – n/w of states , events
  - ➢ Each state receives one or more events at which time transition results in next state

Eg., dynamic model – Banking system

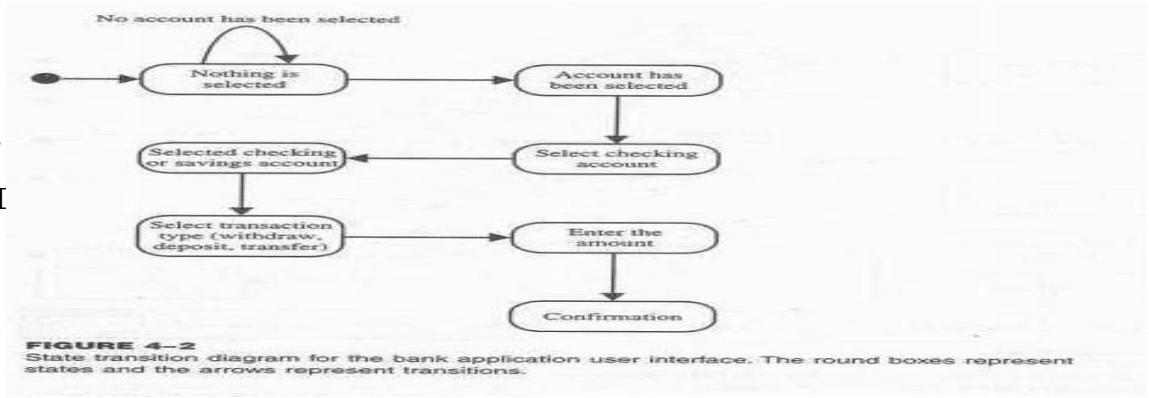


FIGURE 4–2
State transition diagram for the bank application user interface. The round boxes represent states and the arrows represent transitions.

OMT –

- I

- data store-location where data are stored

Eg., ATM

OBJECT MODELING – BOOCH METHODOLOGY

State transition diagram – eg., alarm class

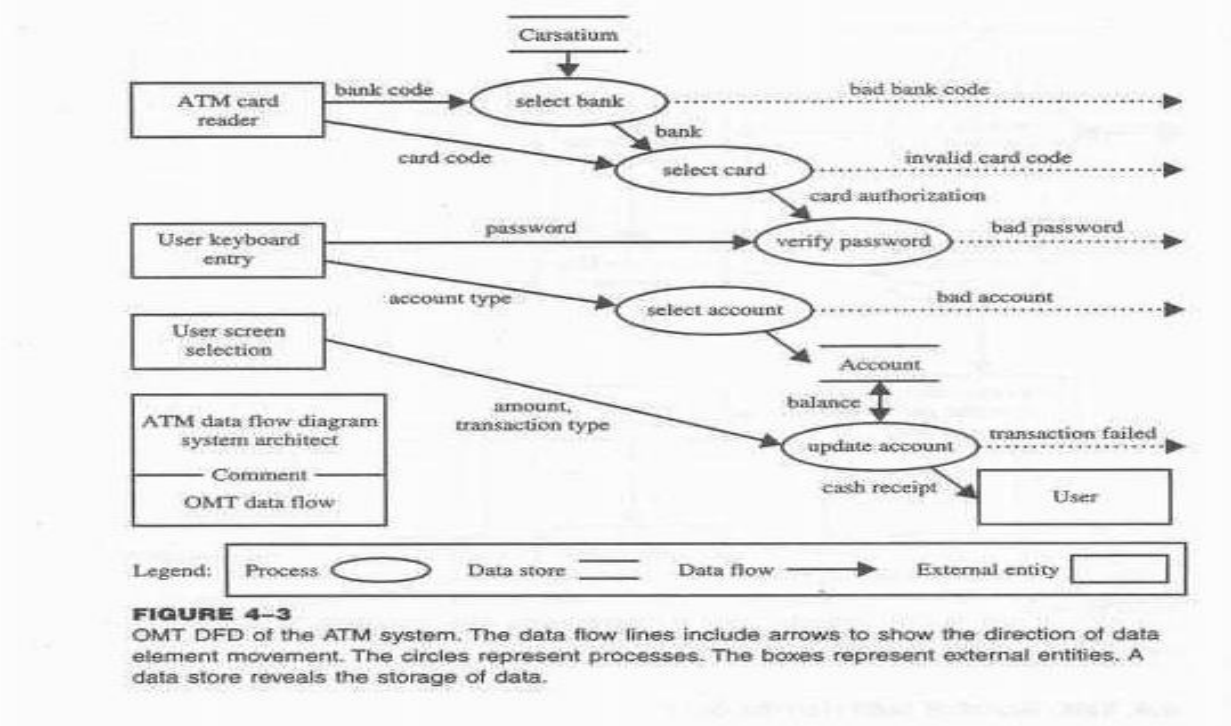


FIGURE 4–3
OMT DFD of the ATM system. The data flow lines include arrows to show the direction of data element movement. The circles represent processes. The boxes represent external entities. A data store reveals the storage of data.

(ii) What is pattern? List and explain the essentials components of a pattern to recognize it properly.

- Design patterns must have a predefined template.
- The template must specify the intent, the other name of the design pattern,
- Motivation, structure of the design pattern,
- Implementation, known uses of the design pattern
- Applicability, other uses, participants, related design patterns if any.
- Selection of the design pattern can be done based on either creational , structural or behavioural patterns.
- Eg., Abstract factory, decorator, Façade, proxy, state.
- **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:** A description of the goal behind the pattern and the reason for using it.
- **Also Known As:** Other names for the pattern.
- **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:** Situations in which this pattern is usable; the context for the pattern.
- **Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants:** A listing of the classes and objects used in the pattern and their roles in the design.
- **Collaboration:** A description of how classes and objects used in the pattern interact with each other.
- **Consequences:** A description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code:** An illustration of how the pattern can be used in a programming language.
- **Known Uses:** Examples of real usages of the pattern.
- **Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

11. i) Draw the class diagram for banking system.(May 2016)
    ii) Draw the sequence diagram for the amount withdraw from ATM.
    Refer Previous answers
12. What are patterns and frameworks? Discuss with examples. (Dec 2016)
    Refer Previous answers

UNIT – III

1. Use noun phrase approach to identify the objects from the grocery store problem.
   A store wants to automate its inventory. It has point-of-sale terminals that can record all of the items and quantities that a customer purchases. Another terminal is also available for the customer service desk to handle returns. It has a similar terminal in the loading deck to handle arriving shipments from suppliers. The meat department and produce department have terminals to enter losses/discounts due to spoilage.

A store wants to automate its inventory. It has point-of-sale terminals that can record all of the items and quantities that a customer purchases. Another terminal is also available for the customer service desk to handle returns. It has a similar terminal in the loading deck to handle arriving shipments from suppliers. The meat department and produce department have terminals to enter losses/discounts due to spoilage.

Step.1:	**Step.2:**	**Step.3**	**Step.4:**
Identify nouns Store Inventory Point-of-sale terminal Terminals Items Quantity Purchase Customer Customer service desk Handle returns Returns Loading dock Shipment Handle shipment Suppliers Meat department	**Eliminate irrelevant nouns** Store Point-of-sale terminal inventory Item Customer Customer service desk Handle returns Returns Handle shipment Shipment Meat department Produce department Department Enter losses Enter discount	**Eliminate redundancies** Store Point-of-sale terminal Item Customer service desk Handle returns Handle shipment Meat department Produce department Enter losses Enter discount	The final set of classes objects after the elimination process. Point-of-sale terminal Item Handling return Handling shipment Enter losses Enter discount Meat department Produce department Store

Produce department Department Enter losses Enter discount Spoilage			

2. Discuss with an example use case driven approach for object oriented analysis.(Dec 2016)
   Explain generalization and specialization with an example.(Dec 2016)
   Outline the activities carried out during object oriented analysis.
   - OOA Uses actors, use cases that describe the system.
   - Actors
   - External factors that interact with the system.
   - Use Cases
   - Scenario that describe how actors use the system
   - **Steps in identifying Use cases**
   - Identify the actors
   - Develop a simple business process model using UML activity Diagram.
   - Develop the use case.
   - What are the users doing with the system.
   - What will be the users doing with the system.
   - Comprehensive documentation of the system
   - Prepare interaction diagram.
   - Classification
   - Identify classes , relationships, attributes , methods.
   - Use cases provide scenario for understanding the specific requirements
   - It Provides interaction between users and system.

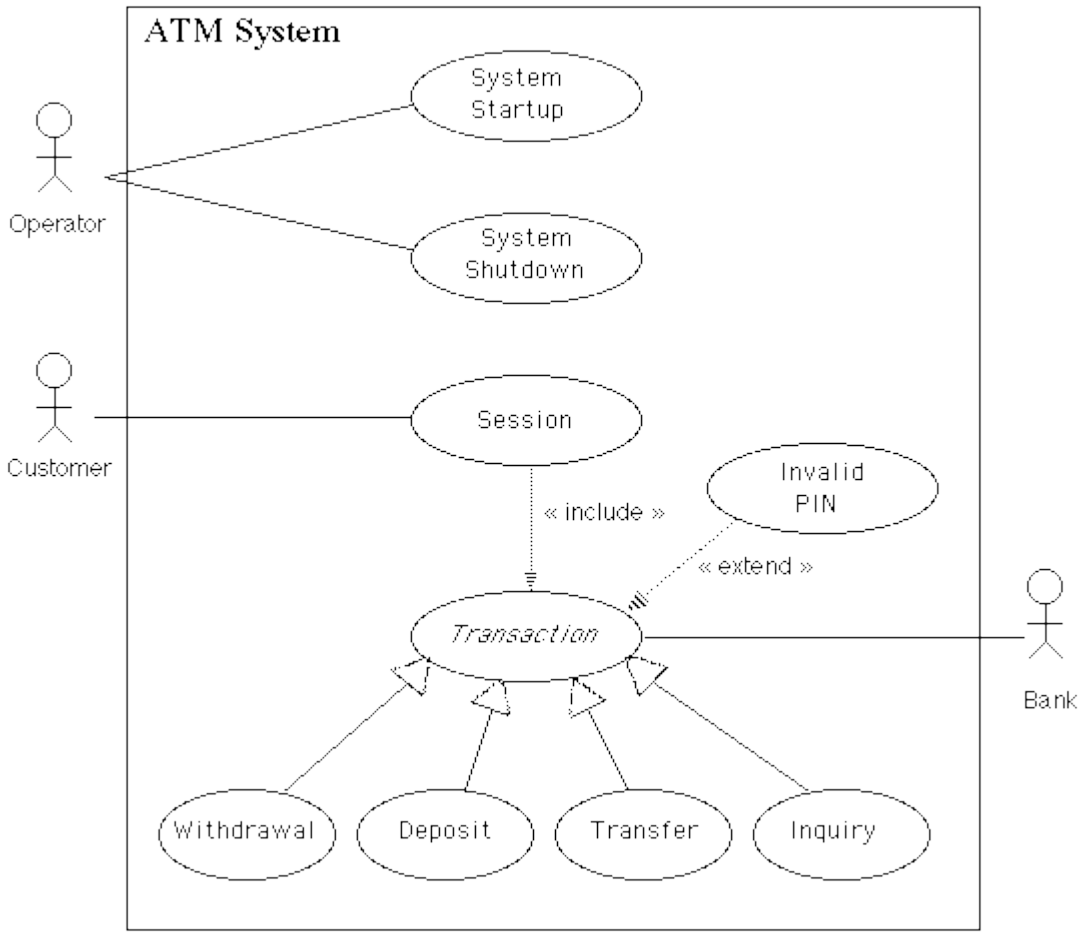
•Captures the goal of the user and responsibility of the user to the system.
•Shows various courses of the events to be performed.
•Defines what happens in the system when the use case is performed.
•Discovers classes and relationships among the subsystems of the system.
•Each use case must have a name and a brief technical description.
•Provides an external view of an application.

•**Use cases**

- A sequence of transactions in a system.
- Yields results of measurable values to an individual actor of the system.
- Provides a special flow of events.
- Group of various courses of events are represented as an use case class.
- Borrow book
- Whether the book is available in the library
- Whether the user is a member of the library.
- **Actors**
- User playing the role with respect to the system.
- Think about roles than people.
    - E.g., First class / Business class passenger.
- Actors are the key to find the use case.
- Single actor may perform several use cases.
- An use case may have several actors.
- An external system that needs information from a current system
- get a value from the use case or from the participant class.
- Actors communicate with the system's use cases.
- A measurable value.
- Evaluate the performance of the use cases in terms of price/ cost.
- E.g., Borrowing book – value for the member of the library.
- Atomic set of activities that are performed either fully or not at all.
- Triggered by a stimulus from an actor of the system.
- **Uses and extends association**
- Extends association is used when an use case similar to another use case does a bit more
- specific task.
- Extends results in inheritance.
- E.g., Borrow book, get a interlibrary loan.
- **Uses**
- Extracts common behaviour for sub flows.
- Creates an use case of its own.
- Abstract use case.
- Not complete, has no initiation actor.
- Concrete use case
- Interacts with others.
- Variations (Fowler & Scott)
- Explore simple and normal use case first.
- For every step in that use case, ask
    - what to go wrong.
    - How might their work be done differently.

3. i)For a Credit card system, every user has to be validated with a PIN number to make a transaction. A customer is allowed three times to validate card giving the correct pin number. Show the usecase representation for the same. (May 2017)

Use Cases for Example ATM System



System Startup Use Case
The system is started up when the operator turns the operator switch to the "on" position. The operator will be asked to enter the amount of money currently in the cash dispenser, and a connection to the bank will be established. Then the servicing of customers can begin.
[Interaction Diagram]

System Shutdown Use Case
The system is shut down when the operator makes sure that no customer is using the machine, and then turns the operator switch to the "off" position. The connection to the bank will be shut down. Then the operator is free to remove deposited envelopes, replenish cash and paper, etc.
[Interaction Diagram]

Session Use Case
A session is started when a customer inserts an ATM card into the card reader slot of the machine. The ATM pulls the card into the machine and reads it. (If the reader cannot read the card due to improper insertion or a damaged stripe, the card is ejected, an error screen is displayed, and the session is aborted.) The customer is asked to enter his/her PIN, and is then allowed to perform one or more transactions, choosing from a menu of possible types of transaction in each case. After each transaction, the customer is asked whether he/she would like to perform another. When the customer is through performing transactions, the card is ejected from the machine and the session ends. If a transaction is aborted due to too many invalid PIN entries, the session is also aborted, with the card being retained in the machine.
The customer may abort the session by pressing the Cancel key when entering a PIN or choosing a transaction type.

[Interaction Diagram]

Transaction Use Case

Note: Transaction is an abstract generalization. Each specific concrete type of transaction implements certain operations in the appropriate way. The flow of events given here describes the behavior common to all types of transaction. The flows of events for the individual types of transaction (withdrawal, deposit, transfer, inquiry) give the features that are specific to that type of transaction.

A transaction use case is started within a session when the customer chooses a transaction type from a menu of options. The customer will be asked to furnish appropriate details (e.g. account(s) involved, amount). The transaction will then be sent to the bank, along with information from the customer's card and the PIN the customer entered.

If the bank approves the transaction, any steps needed to complete the transaction (e.g. dispensing cash or accepting an envelope) will be performed, and then a receipt will be printed. Then the customer will be asked whether he/she wishes to do another transaction.

If the bank reports that the customer's PIN is invalid, the Invalid PIN extension will be performed and then an attempt will be made to continue the transaction. If the customer's card is retained due to too many invalid PINs, the transaction will be aborted, and the customer will not be offered the option of doing another.

If a transaction is cancelled by the customer, or fails for any reason other than repeated entries of an invalid PIN, a screen will be displayed informing the customer of the reason for the failure of the transaction, and then the customer will be offered the opportunity to do another.

The customer may cancel a transaction by pressing the Cancel key as described for each individual type of transaction below.

All messages to the bank and responses back are recorded in the ATM's log.

[Interaction Diagram]

Withdrawal Transaction Use Case

A withdrawal transaction asks the customer to choose a type of account to withdraw from (e.g. checking) from a menu of possible accounts, and to choose a dollar amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request before sending the transaction to the bank. (If not, the customer is informed and asked to enter a different amount.) If the transaction is approved by the bank, the appropriate amount of cash is dispensed by the machine before it issues a receipt. (The dispensing of cash is also recorded in the ATM's log.)

A withdrawal transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the dollar amount.

[Interaction Diagram]

Deposit Transaction Use Case

A deposit transaction asks the customer to choose a type of account to deposit to (e.g. checking) from a menu of possible accounts, and to type in a dollar amount on the keyboard. The transaction is initially sent to the bank to verify that the ATM can accept a deposit from this customer to this account. If the transaction is approved, the machine accepts an envelope from the customer containing cash and/or checks before it issues a receipt. Once the envelope has been received, a second message is sent to the bank, to confirm that the bank can credit the customer's account - contingent on manual verification of the deposit envelope contents by an operator later. (The receipt of an envelope is also recorded in the ATM's log.)

A deposit transaction can be cancelled by the customer pressing the Cancel key any time prior to inserting the envelope containing the deposit. The transaction is automatically cancelled if the customer fails to insert the envelope containing the deposit within a reasonable period of time after being asked to do so.

[Interaction Diagram]

Transfer Transaction Use Case

A transfer transaction asks the customer to choose a type of account to transfer from (e.g. checking) from a menu of possible accounts, to choose a different account to transfer to, and to type in a dollar amount on the keyboard. No further action is required once the transaction is approved by the bank before printing the receipt.
A transfer transaction can be cancelled by the customer pressing the Cancel key any time prior to entering a dollar amount.
[Interaction Diagram]

Inquiry Transaction Use Case
An inquiry transaction asks the customer to choose a type of account to inquire about from a menu of possible accounts. No further action is required once the transaction is approved by the bank before printing the receipt.
An inquiry transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the account to inquire about.
[Interaction Diagram]

Invalid PIN Extension
An invalid PIN extension is started from within a transaction when the bank reports that the customer's transaction is disapproved due to an invalid PIN. The customer is required to re-enter the PIN and the original request is sent to the bank again. If the bank now approves the transaction, or disapproves it for some other reason, the original use case is continued; otherwise the process of re-entering the PIN is repeated. Once the PIN is successfully re-entered, it is used for both the current transaction and all subsequent transactions in the session. If the customer fails three times to enter the correct PIN, the card is permanently retained, a screen is displayed informing the customer of this and suggesting he/she contact the bank, and the entire customer session is aborted.
If the customer presses Cancel instead of re-entering a PIN, the original transaction is cancelled.

iii) Write short notes on the following CRC approach for identifying classes.
A Class Responsibility Collaborator (CRC) model (Beck & Cunningham 1989; Wilkinson 1995; Ambler 1995) is a collection of standard index cards that have been divided into three sections, as depicted in Figure 1. A class represents a collection of similar objects, a responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities. Figure 2 presents an example of two hand-drawn CRC cards.
Figure 1. CRC Card Layout.

Figure 2. Hand-drawn CRC Cards.

Although CRC cards were originally introduced as a technique for teaching object-oriented concepts, they have also been successfully used as a full-fledged modeling technique. My experience is that CRC models are an incredibly effective tool for conceptual modeling as well as for detailed design. CRC cards feature prominently in eXtreme Programming (XP) (Beck 2000) as a design technique. My focus here is on applying CRC cards for conceptual modeling with your stakeholders.

A class represents a collection of similar objects. An object is a person, place, thing, event, or concept that is relevant to the system at hand. For example, in a university system, classes would represent students, tenured professors, and seminars. The name of the class appears across the top of a CRC card and is typically a singular noun or singular noun phrase, such as Student, Professor, and Seminar. You use singular names because each class represents a generalized version of a singular object. Although there may be the student John O'Brien, you would model the class Student. The

information about a student describes a single person, not a group of people. Therefore, it makes sense to use the name Student and not Students. Class names should also be simple. For example, which name is better: Student or Person who takes seminars?

A responsibility is anything that a class knows or does. For example, students have names, addresses, and phone numbers. These are the things a student knows. Students also enroll in seminars, drop seminars, and request transcripts. These are the things a student does. The things a class knows and does constitute its responsibilities. Important: A class is able to change the values of the things it knows, but it is unable to change the values of what other classes know.

Sometimes a class has a responsibility to fulfill, but not have enough information to do it. For example, as you see in Figure 3 students enroll in seminars. To do this, a student needs to know if a spot is available in the seminar and, if so, he then needs to be added to the seminar. However, students only have information about themselves (their names and so forth), and not about seminars. What the student needs to do is collaborate/interact with the card labeled Seminar to sign up for a seminar. Therefore, Seminar is included in the list of collaborators of Student.

Figure 3. Student CRC card.

Collaboration takes one of two forms: A request for information or a request to do something. For example, the card Student requests an indication from the card Seminar whether a space is available, a request for information. Student then requests to be added to the Seminar, a request to do something. Another way to perform this logic, however, would have been to have Student simply request Seminar to enroll himself into itself. Then have Seminar do the work of determining if a seat is available and, if so, then enrolling the student and, if not, then informing the student that he was not enrolled.

So how do you create CRC models? Iteratively perform the following steps:

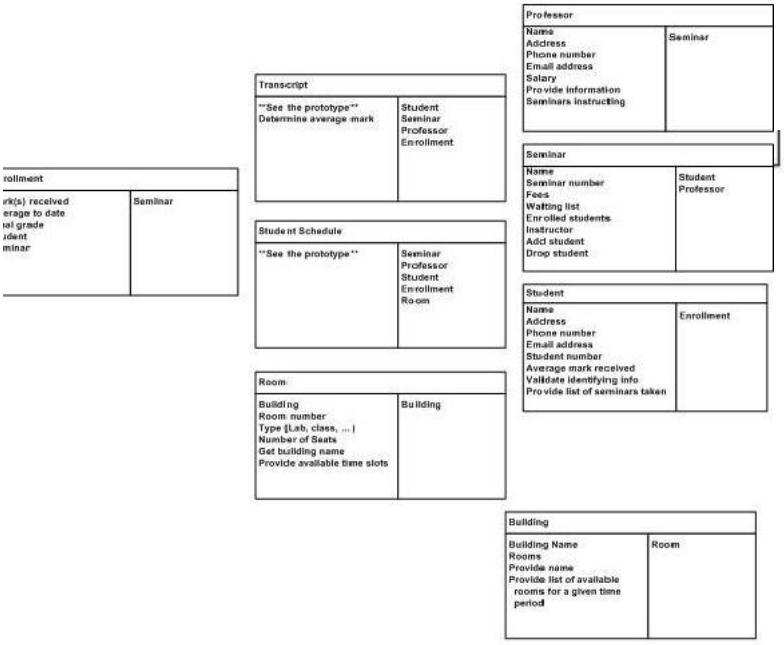
Find classes. Finding classes is fundamentally an analysis task because it deals with identifying the building blocks for your application. A good rule of thumb is that you should look for the three-to-five main classes right away, such as Student, Seminar, and Professor in Figure 4. I will sometimes include UI classes such as Transcript and Student Schedule, both are reports, although others will stick to just entity classes. Also, I'll sometimes include cards representing actors when my stakeholders are struggling with the concept of a student in the real world (the actor) versus the student in the system (the entity).

Find responsibilities. You should ask yourself what a class does as well as what information you wish to maintain about it. You will often identify a responsibility for a class to fulfill a collaboration with another class.

Define collaborators. A class often does not have sufficient information to fulfill its responsibilities. Therefore, it must collaborate (work) with other classes to get the job done. Collaboration will be in one of two forms: a request for information or a request to perform a task. To identify the collaborators of a class for each responsibility ask yourself "does the class have the ability to fulfill this responsibility?". If not then look for a class that either has the ability to fulfill the missing functionality or the class which should fulfill it. In doing so you'll often discover the need for new responsibilities in other classes and maybe even the need for a new class or two.

Move the cards around. To improve everyone's understanding of the system, the cards should be placed on the table in an intelligent manner. Two cards that collaborate with one another should be placed close together on the table, whereas two cards that don't collaborate should be placed far apart. Furthermore, the more two cards collaborate, the closer they should be on the desk. By having cards that collaborate with one another close together, it's easier to understand the relationships between classes.

Figure 4. CRC Model.



How do you keep your CRC modeling efforts agile? By following the AM practice Model in Small Increments. The best way to do this is to create a CRC model for a single requirement, such as a user story, business rule, or system use case, instead of the entire collection of requirements for your system. Because CRC cards are very simple tools they are inclusive, enabling you to follow AM's Active Stakeholder Participation practice.

It's important to recognize that a CRC model isn't carved in stone. When you evolve it into a UML class diagram, or perhaps straight into code, you'll change the schema over time. Responsibilities will be reorganized, new classes will be introduced, existing classes will disappear, and so on. This is what happens when you take an evolutionary approach to development.

3. Appraise the steps in modelling a usecase diagram with an example.(May 2017)
   Refer previous questions
4. Explain in detail about how to identify object relationships, attributes and methods with an example.

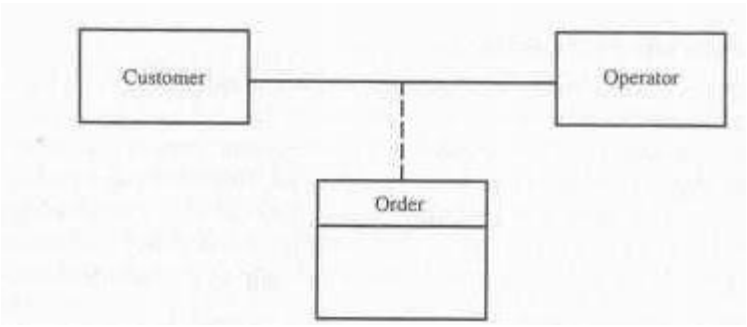
Types of relationships among objects

1. Association
2. Super-sub structure
3. Aggregation and a-part-of structure

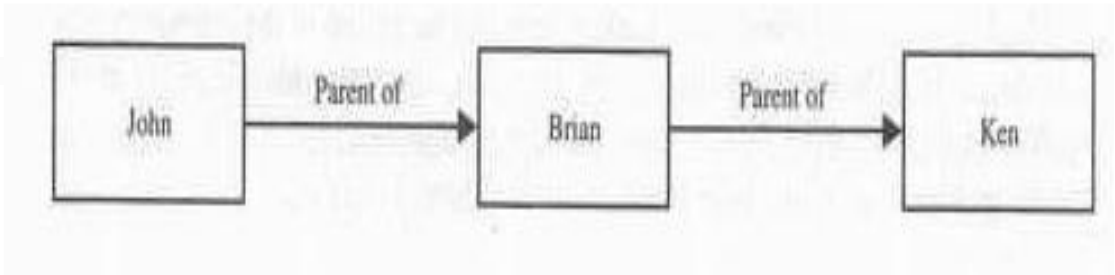
Association

- Physical or conceptual connection b/w two or more objects
- Identifying associations
  - Begins by analyzing the interactions b/w classes
  - Following questions help us to identify associations
    - Is the class capable of fulfilling required task by itself?
    - If not, what does it need?

    - From what other class can it acquire what it needs?
  - Extract all associations from problem statement and get them down on paper
  - Refine them later
- **Guidelines for identifying association**
  - A Dependency between 2 or more classes may be an association.
  - Association often corresponds to a verb or prepositional phrase.
    - Eg. Next to, works for
  - A Reference from one class to another is an association
    - Taken from general knowledge
- **Common Association Patterns**
  - Location Association
    - next to, part of, contained in
    - Eg.
  - Communication Association
    - talk to, order to
    - Eg.

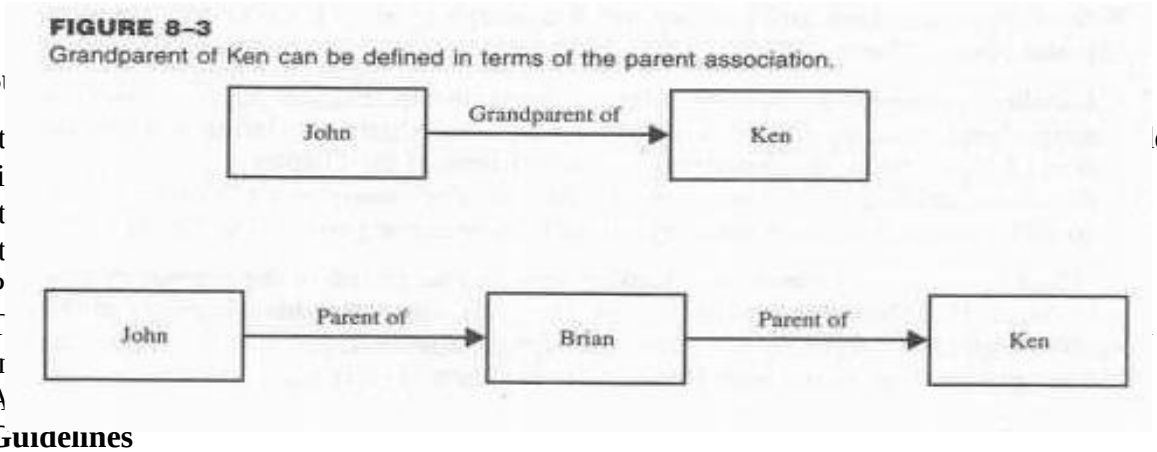


- **Eliminate Unnecessary Associations**
- **Implementation Association**
- Defer implementation - specific associations to the design phase.
- It is concerned with the implementation or design of the class within certain programming or development environment.
- **Ternary Associations**
- Ternary or n-ary association is an association among more than two classes.
- It complicates the representations.
- When possible, such ternary associations must be reduced to binary associations.

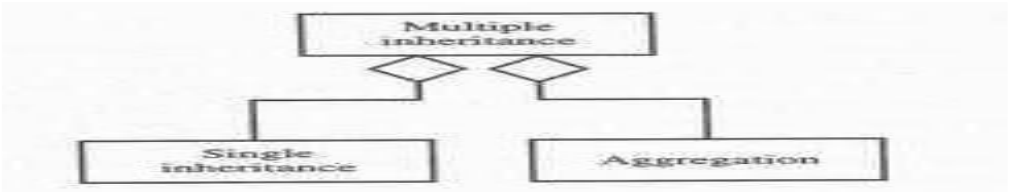


- **Directed actions / derived associations**
- It is defined in terms of other associations.
- Since they are redundant, these associations must be avoided.

Super-S

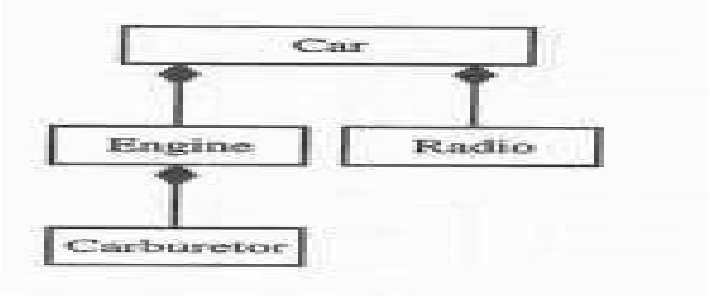


- It etermines the li
- It
- It
- P
- T
- Ir
- A
- **Guidelines**
  - Top-down
    - Look for noun phrases composed of various adjectives in the class name
    - Specialize only when subclasses have significant behavior
    - Avoid excessive refinement
  - Bottom-up
    - Look for classes with similar attributes or methods.
    - Group them by moving the common attributes and methods to an abstract class
    - Don't force classes to fit a preconceived generalization structure
  - Reusability
    - Move attributes and methods as high as possible in the hierarchy
    - Don't create very specialized classes at the top of the hierarchy.
  - Multiple Inheritance
    - Avoid excessive use of it
    - It is highly complicated structure when several ancestors define the method.
    - It is more difficult to understand the program written in multiple inheritance.
    - Inherit from the most appropriate class and add an object of another class as an attribute.
    - It can be represented as an aggregation of a single inheritance and aggregation.

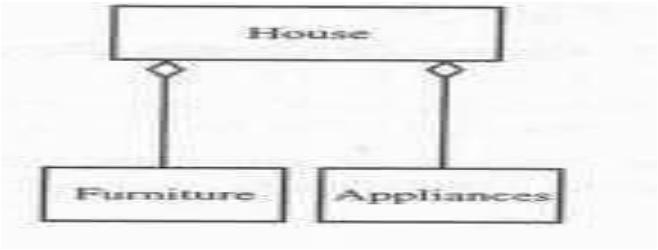


A-part-of relationships – aggregation

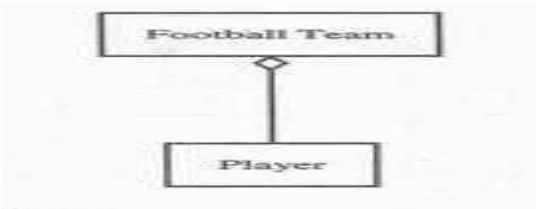
- Represents a situation where a class consists of several component classes
- A class that is composed of other classes does not behave like its parts; it behaves differently.



- Properties
  - Transitivity
    - A is part of B and B is part of C, then A is part of C
    - Eg. Carburetor part of engine, engine is part of car; carburetor part of car
  - Antisymmetry
    - A is part of B, B is not part of A
    - Eg. Engine part of car, car not part of engine
- 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, include it as an attribute with the whole class
- Does it provide a useful abstraction in dealing with the problem domain?
- **Guidelines**
  - Assembly
    - Constructed from its parts and an assembly-part situation physically exists
    - Eg. French onion soup is an assembly of onion, butter, cheese
  - Container
    - A physical whole encompasses but is not constructed from physical parts
    - -Eg. House container for furniture and appliances



  - Collection-member
    - A conceptual whole encompasses parts that may be physical or conceptual
    - Eg. Football team is a collection of players



Identifying attributes & methods

- Identifying attributes of a system's classes starts with understanding the system's responsibilities.
- This is done by developing use cases and the desired characteristics of the applications such as determining what information users need from the system.
- What information about an object should we keep track of?
  - Attributes
- What services must a class provide?
  - methods

6. Discuss the difficulty of classification. Briefly explain the classical and modern approaches used for identifying classes and objects.(May 2016)

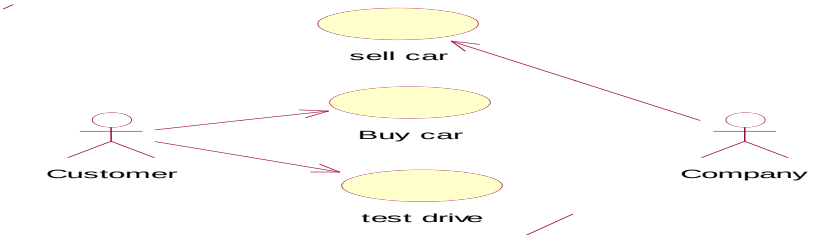
Guidelines

- Nouns followed by prepositional phrases (Eg. Cost of), adjectives or adverbs
- Keep class simple; state only enough attributes
- use your knowledge of application domain and real world to find them
- Omit derived attributes
- Do not carry discovery of attributes to excess. Add more attributes in subsequent iterations

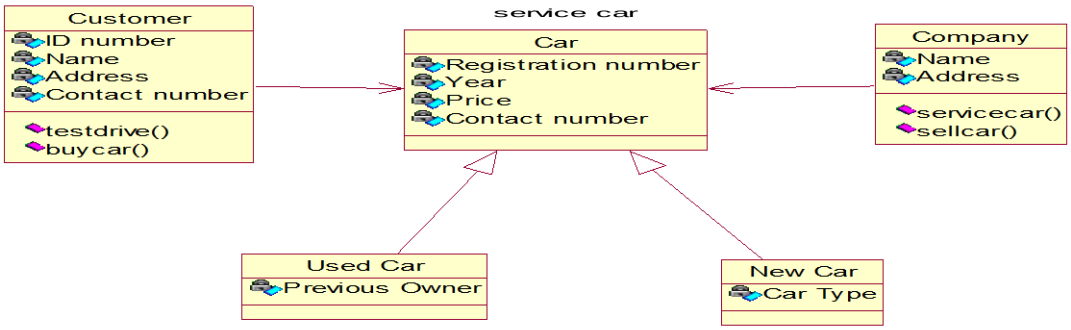
- **Methods & messages**
- **Every piece of data, or object, is surrounded by a rich set of routines called methods**
- Define methods by analyzing UML diagrams and use cases

7. Explain the following: (i) Give a detailed note on super-sub class relationship and a-part-of relationship. Ii) Guidelines for identifying super-sub relationships(May 2016)
   Refer the previous answers
8. Consider a Car Company which sell both the used cars and new cars of various model, design the system and give the Use case, Class diagrams with the following requirements.
   i) Customers are allowed to perform test drive before purchasing the car.
   ii)The Company also sell car parts and do servicing.
   iii)The customer either purchases the parts available in the part department directly or the service department can replace the parts of the car during servicing the car.

Customers are allowed to perform test drive before purchasing the car.

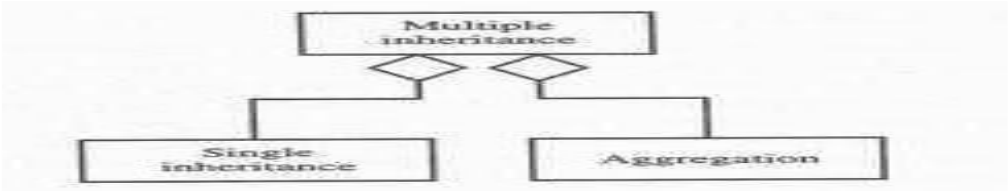


Class Diagram



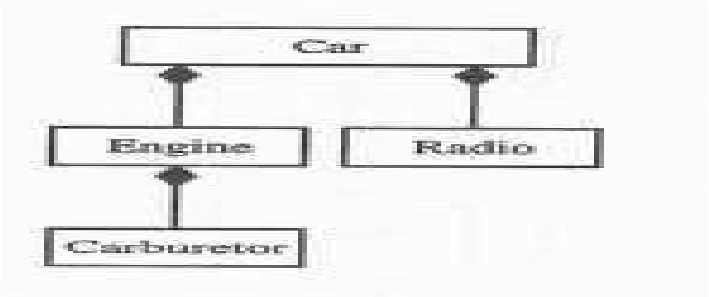
9. Which relation is called as part of relation? What are the major properties of it and how to identify it? explain with suitable examples.
   - **Super-sub class relationship and a-part-of relationship**
   - It represents inheritance relationships between related classes, and the class hierarchy determines the lines of inheritance between classes
   - It allows objects to be built form other objects
   - It helps to identify commonality of objects when constructing new classes
   - Parent class – also known as base or super class or ancestor
   - The super - sub class hierarchy is based on inheritance which is programming by extension

- Inheritance allows classes to share and reuse behaviour and attributes
- A class inherits the behaviours and attributes of all of its super classes.
- **Guidelines**
  - Top-down
    - Look for noun phrases composed of various adjectives in the class name
    - Specialize only when subclasses have significant behavior
    - Avoid excessive refinement
  - Bottom-up
    - Look for classes with similar attributes or methods.
    - Group them by moving the common attributes and methods to an abstract class
    - Don't force classes to fit a preconceived generalization structure
  - Reusability
    - Move attributes and methods as high as possible in the hierarchy
    - Don't create very specialized classes at the top of the hierarchy.
  - Multiple Inheritance
    - Avoid excessive use of it
    - It is highly complicated structure when several ancestors define the method.
    - It is more difficult to understand the program written in multiple inheritance.
    - Inherit from the most appropriate class and add an object of another class as an attribute.
    - It can be represented as an aggregation of a single inheritance and aggregation.



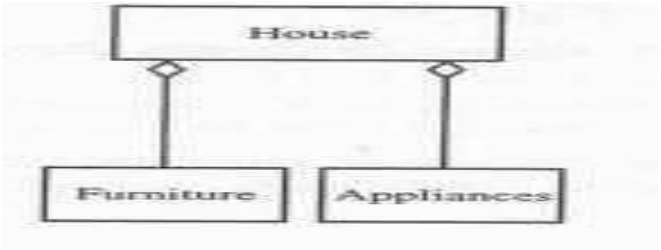
A-part-of relationships – aggregation

- Represents a situation where a class consists of several component classes
- A class that is composed of other classes does not behave like its parts; it behaves differently.

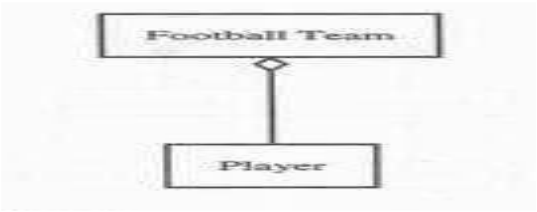


- Properties
  - Transitivity
    - A is part of B and B is part of C, then A is part of C
    - Eg. Carburetor part of engine, engine is part of car; carburetor part of car
  - Antisymmetry
    - A is part of B, B is not part of A
    - Eg. Engine part of car, car not part of engine
- Does the part 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, include it as an attribute with the whole class
- Does it provide a useful abstraction in dealing with the problem domain?
- **Guidelines**
  - Assembly
    - Constructed from its parts and an assembly-part situation physically exists
    - Eg. French onion soup is an assembly of onion, butter, cheese
  - Container
    - A physical whole encompasses but is not constructed from physical parts
    - -Eg. House container for furniture and appliances



  - Collection-member
    - A conceptual whole encompasses parts that may be physical or conceptual
    - Eg. Football team is a collection of players



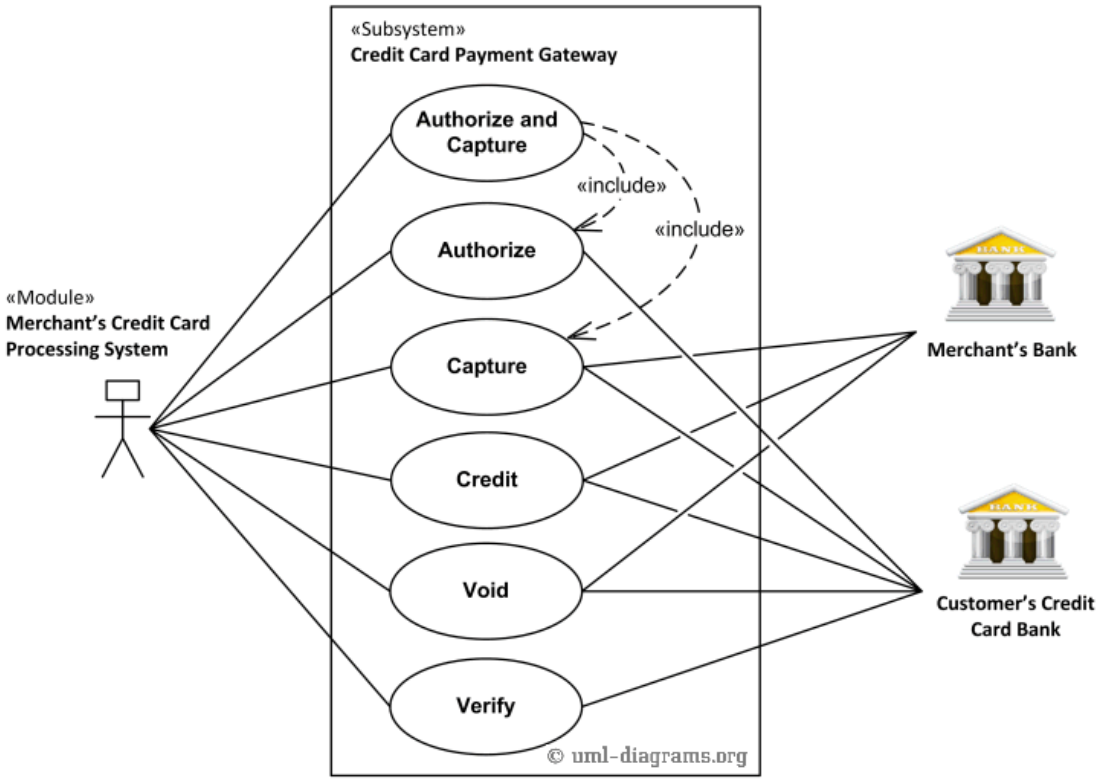
10. i) Explain the noun phrase approach for classification and identification of objects (May 2015)
    ii)Write about guidelines for developing effective documentation

    Refer the previous answers

11. **For a Credit card system, every user has to be validated with a PIN number to make a transaction. A customer is allowed three times to validate card giving the correct pin number. Show the usecase representation for the same. (May 2017)**

This UML use case diagram example shows some use cases for a system which processes credit cards.
Credit Card Processing System (aka Credit Card Payment Gateway) is a **subject**, i.e. system under design or consideration. Primary **actor** for the system is a **Merchant's Credit Card Processing System**. The merchant submits some credit card transaction request to the credit card payment gateway on behalf of a customer. Bank which issued customer's credit card is actor which could approve or reject the transaction. If transaction is approved, funds will be transferred to merchant's bank account.
Authorize and Capture use case is the most common type of credit card transaction. The requested amount of money should be first authorized by **Customer's Credit Card Bank**, and if approved, is further submitted for

settlement. During the settlement funds approved for the credit card transaction are deposited into the **Merchant's Bank** account.
In some cases, only **authorization** is requested and the transaction will not be sent for settlement. In this case, usually if no further action is taken within some number of days, the authorization expires. Merchants can submit this request if they want to verify the availability of funds on the customer's credit card, if item is not currently in stock, or if merchant wants to review orders before shipping.
Capture (request to capture funds that were previously authorized) use case describes several scenarios when merchant needs to complete some previously authorized transaction - either submitted through the payment gateway or requested without using the system, e.g. using voice authorization.

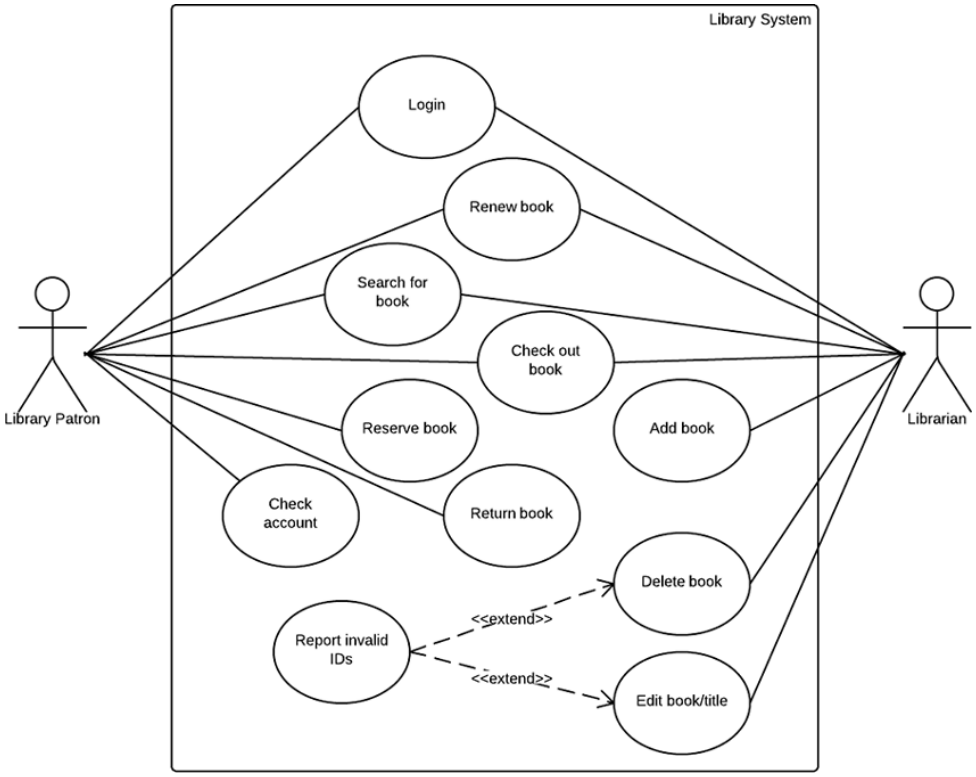


UML use case diagram example for a credit cards processing system.
Credit use case describes situations when customer should receive a refund for a transaction that was either successfully processed and settled through the system or for some transaction that was not originally submitted through the payment gateway.
Void use case describes cases when it is needed to cancel one or several related transactions that were not yet settled. If possible, the transactions will not be sent for settlement. If the Void transaction fails, the original transaction is likely already settled.
Verify use case describes zero or small amount verification transactions which could also include verification of some client's data such as address.
Briefly explain about use case model for library mgt. system. (May2016)

UML Use Case Diagram for Library Management System

Even in this age of high-powered computers, an old-fashioned library has its place. To find information, a patron and librarian must work together to narrow search parameters and identity relevant resources. In UML, the process of checking out a book can be represented as a use case, with symbols that represent actors and other essential entities. To create a use case diagram of your own, just open up a document and start dragging shapes onto the page. If you're not sure where to start, the examples below can help.

Library Management System Use Case Diagram Template



UNIT – IV

1. What is coupling? Tabulate and explain in detail about the types of coupling among objects or components.
   - Coupling – measure of strength of association of objects.
   - Helps to focus on design issues such as
     - The degree of coupling is a function of
     - How complicated the connection is
     - Whether the connection refer to the object itself / some data structures within it.
     - What is being sent / received?

Types of Coupling

- ***Interaction coupling***
  - The amount and complexity of messages between components.
  - Has little interaction.
- ***Inheritance coupling***
- Form of coupling between super and sub classes

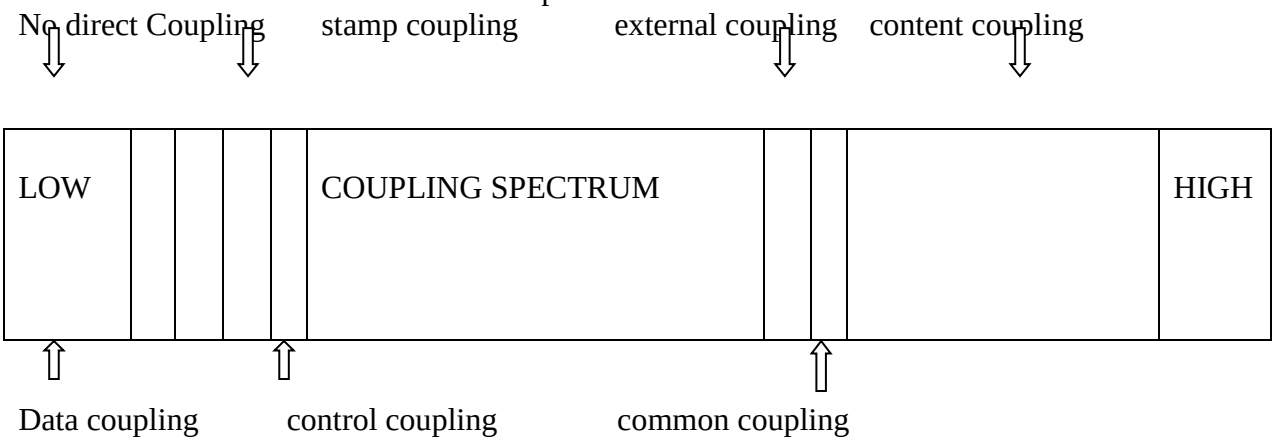
Degree of coupling	Name	Description
1. Very High	Content Coupling	Connection involves direct reference to attributes / methods of another objects
2. High	Common coupling	Two objects accessing common global data space.

3.Medium	Control coupling	Explicit control of the processing objects
4. Low	Stamp coupling	Passing an aggregate data structure to another objects.
5. Very low	Data coupling	Simple data items /aggregate structures.

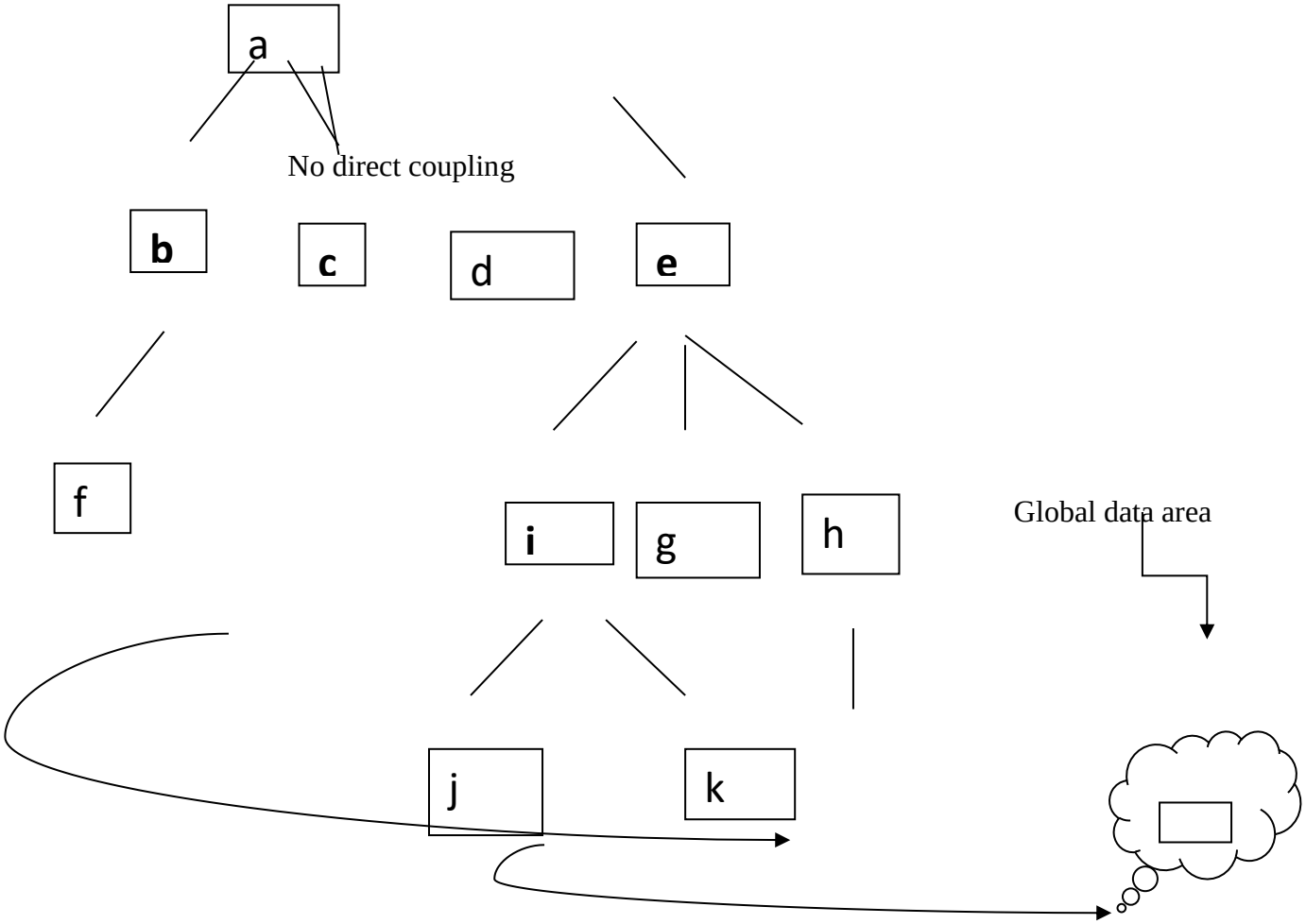
- It is the measure of interconnection among module.
- It depends upon the interface complexity between module

(i.e)

- Entry point
- Reference point to the module
- What data pass across the module



- No Direct Coupling:
- In the example a, d are from different modules, so no direct coupling occurs.
- Data Coupling:
- Occurs when long strings of argument are passed between components.
- E.g., c is subordinate to a.
  - Data are passed from a to c is data coupling.
- Stamp Coupling:
- Variation of data coupling occurs when parts of larger data structures are passed between components.
- When a portion of data structure passed via a module interface (b and a).
- Control Coupling:
- Occurs when one components passes control flags as argument to another.
  - Module e pass control to f,g,h
- External Coupling:
- Occurs when a component communicate or collaborates with in infrastructure components.
- Eg) database
- Common Coupling:
- Occurswhen several components make use of a global variable.
- c,g,k are data item using the global area.



- Content Coupling:
- Occurs when one component modifies internal data in another component
- Routine Call Coupling
- Occurs when one operator invokes another
- Type Use Coupling
- Occurs when one component uses a data type defined in another
- Inclusion or Import Coupling
- Occurs when one component imports a package or uses the content of another
- OO design has 2 types of coupling: Interaction coupling and Inheritance coupling
- Interaction coupling
- The amount & complexity of messages between components.
- Desirable to have a little interaction.
- Minimize the number of messages sent & received by an object
- Inheritance coupling
- Coupling between super-and subclasses
- A subclass is coupled to its super class in terms of attributes & methods
- High inheritance coupling is desirable
- Each specialization class should not inherit lots of unrelated & unneeded methods & attributes

3. Explain in detail about the designing user interface for bank ATM.

- **Macro Process**
  Determine if the class interacts with a human actor

- The only class that interacts with a human actor is ATM Machine

- Identify view (interface) objects for the class
  - Use cases interact directly with actors are:
    - Bank transaction
    - Checking transaction history
    - Deposit checking
    - Deposit savings
    - Savings transaction history
    - Withdraw checking
    - Withdraw savings
    - Valid/invalid pin
  - Based on these use cases, we have identified 8 view objects

- Define the relationships among the view objects
  - AccountTransactionUI (for bank transaction)
  - CheckingTransactionHistoryUI
  - SavingsTransactionHistoryUI
  - BankClientAccessUI (for validating a PIN code)
  - DepositCheckingUI
  - DepositSavingsUI
  - WithdrawCheckingUI
  - WithdrawSavingsUI

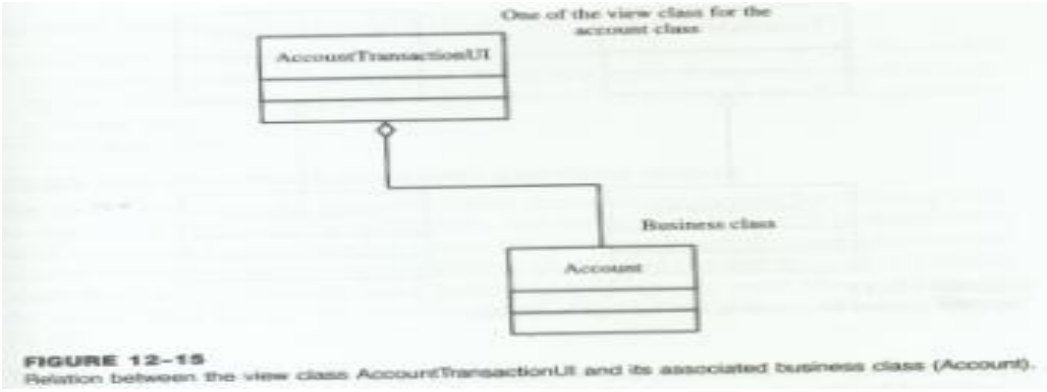
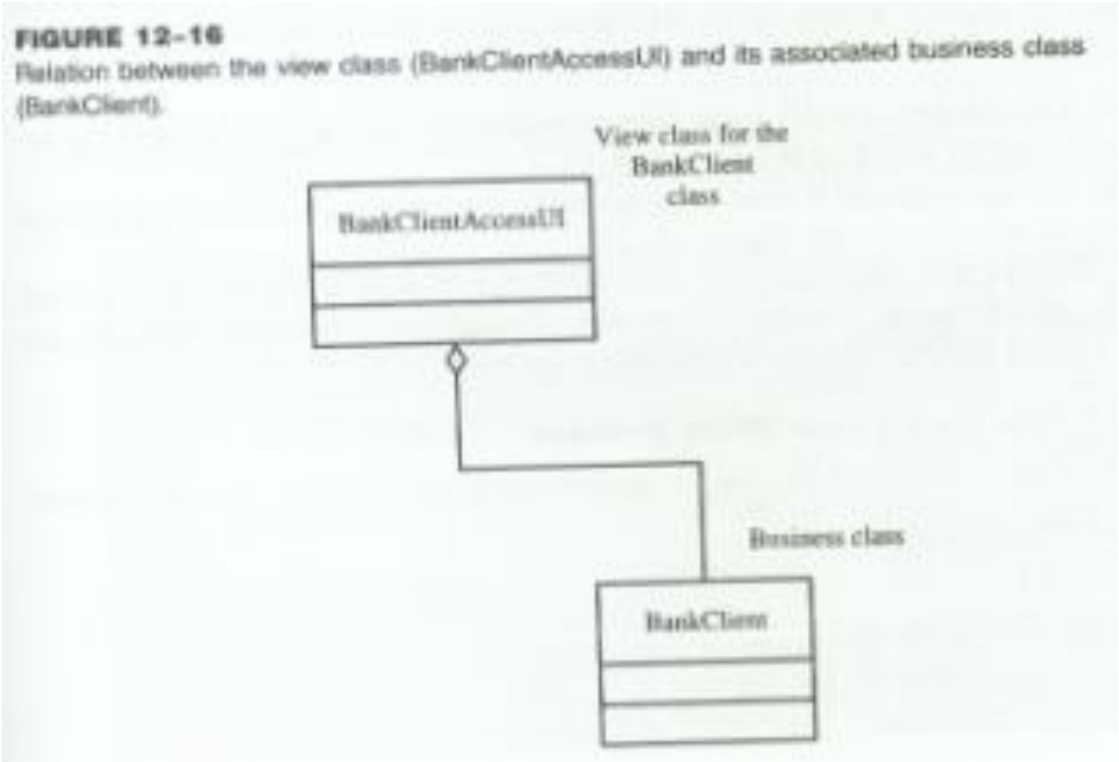


FIGURE 12–15
Relation between the view class AccountTransactionUI and its associated business class (Account).

FIGURE 12–16
Relation between the view class (BankClientAccessUI) and its associated business class (BankClient).



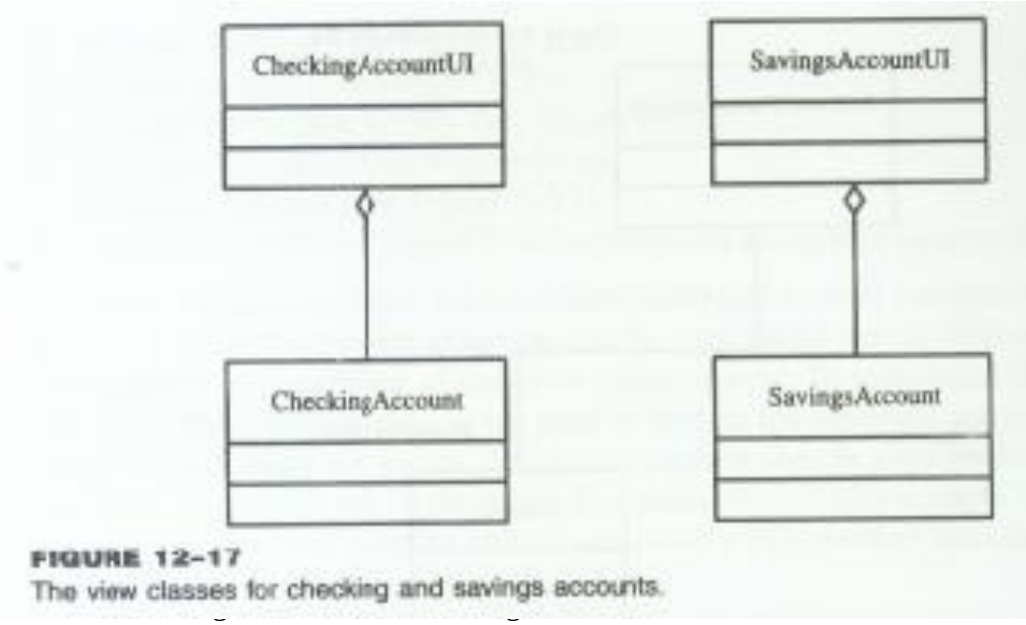


FIGURE 12–17
The view classes for checking and savings accounts.

- WithdrawCheckingUI,
  - Finally, we need one more view class that provides main control or main UI

12.8.2 The View Layer Micro Process

Based on the outcome of the macro process, we have the following view classes:

BankClientAccessUI
MainUI
AccountTransactionUI
CheckingAccountUI
SavingsAccountUI

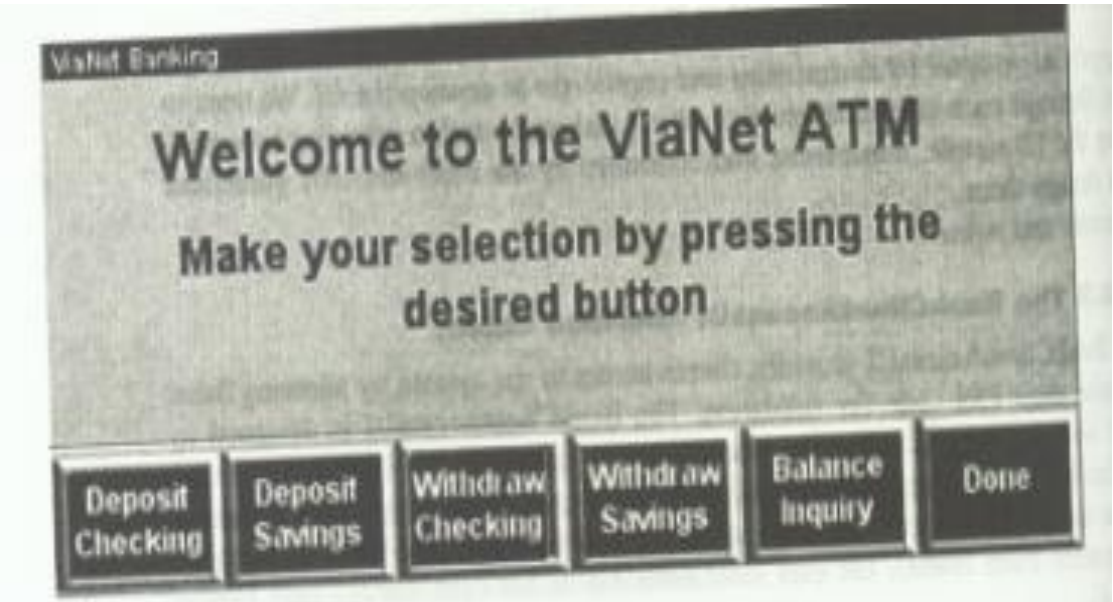
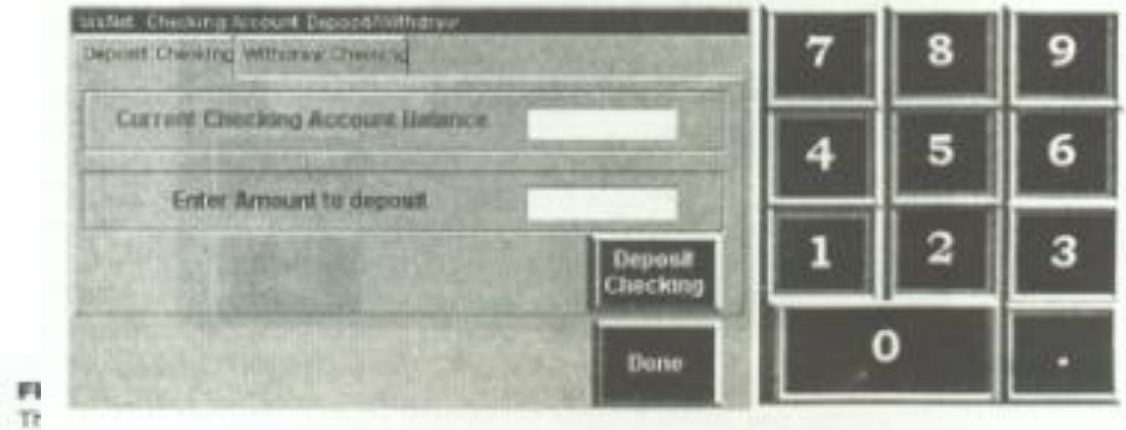
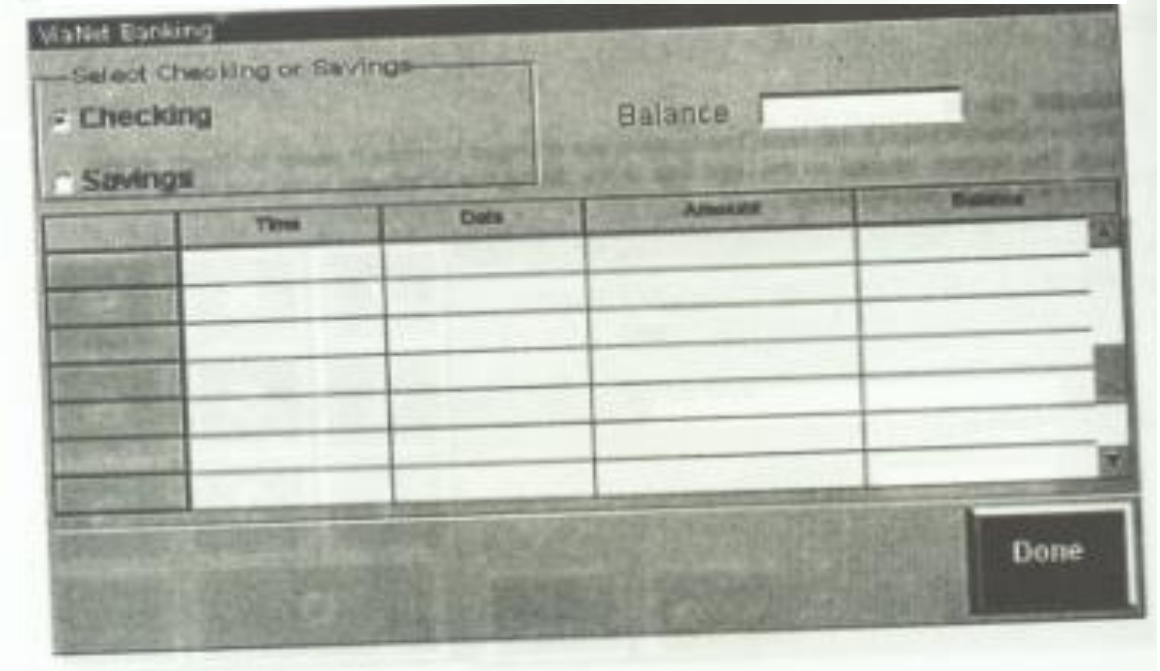


FIGURE 12-19
The MainUI interface

FIGURE 12-21
The CheckingAccountUI and SavingsAccountUI interface objects.





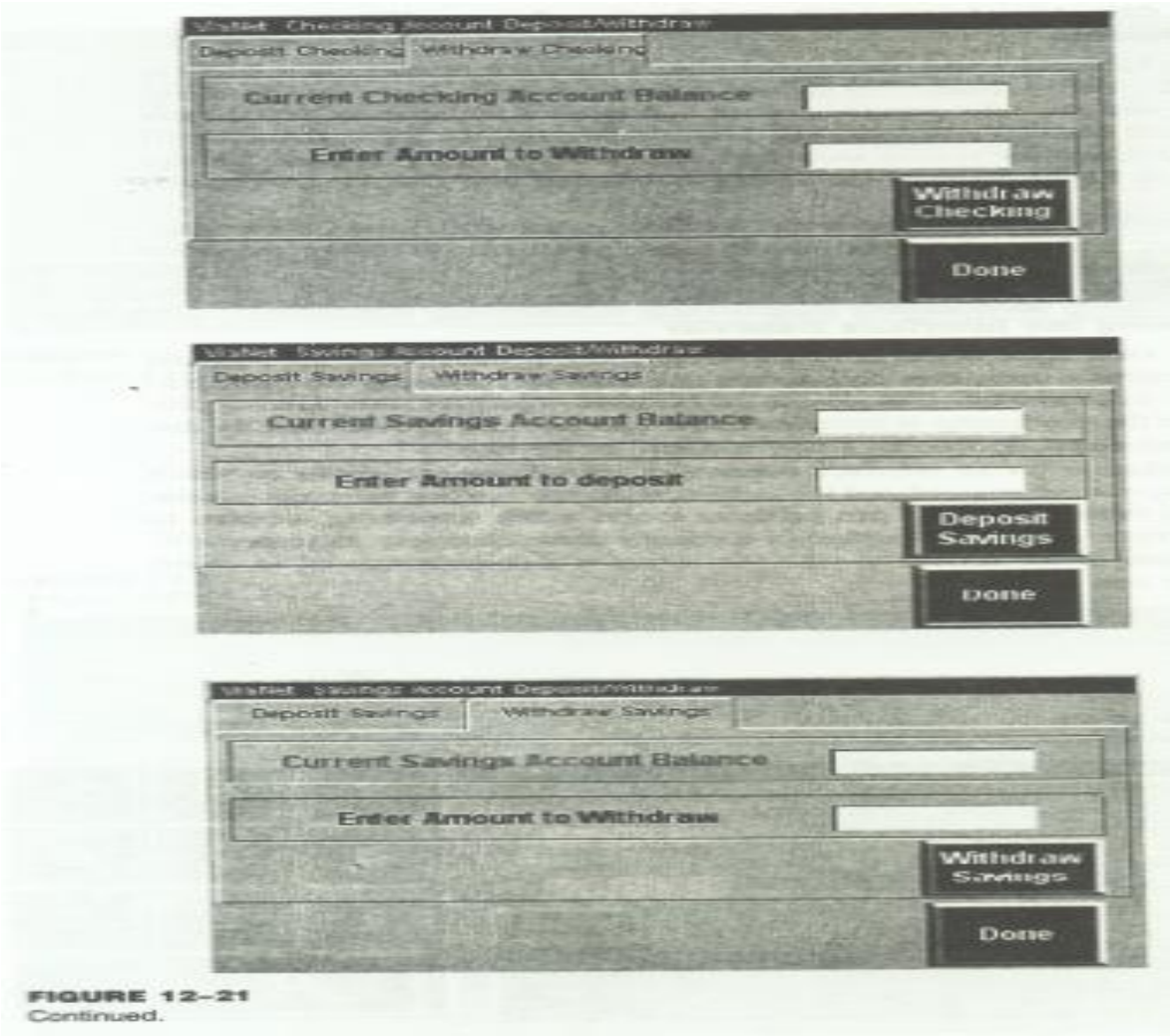


FIGURE 12–21
Continued.

Identifying Events and Actions for BankClientAccessUI Interface Object

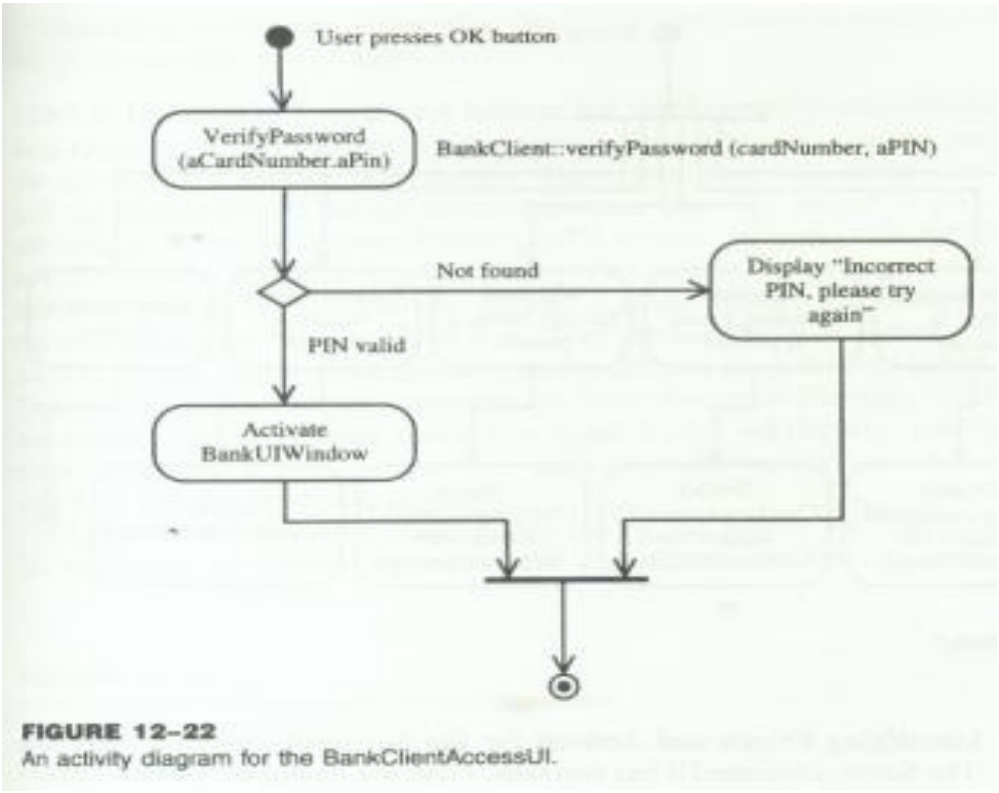


FIGURE 12–22
An activity diagram for the BankClientAccessUI.

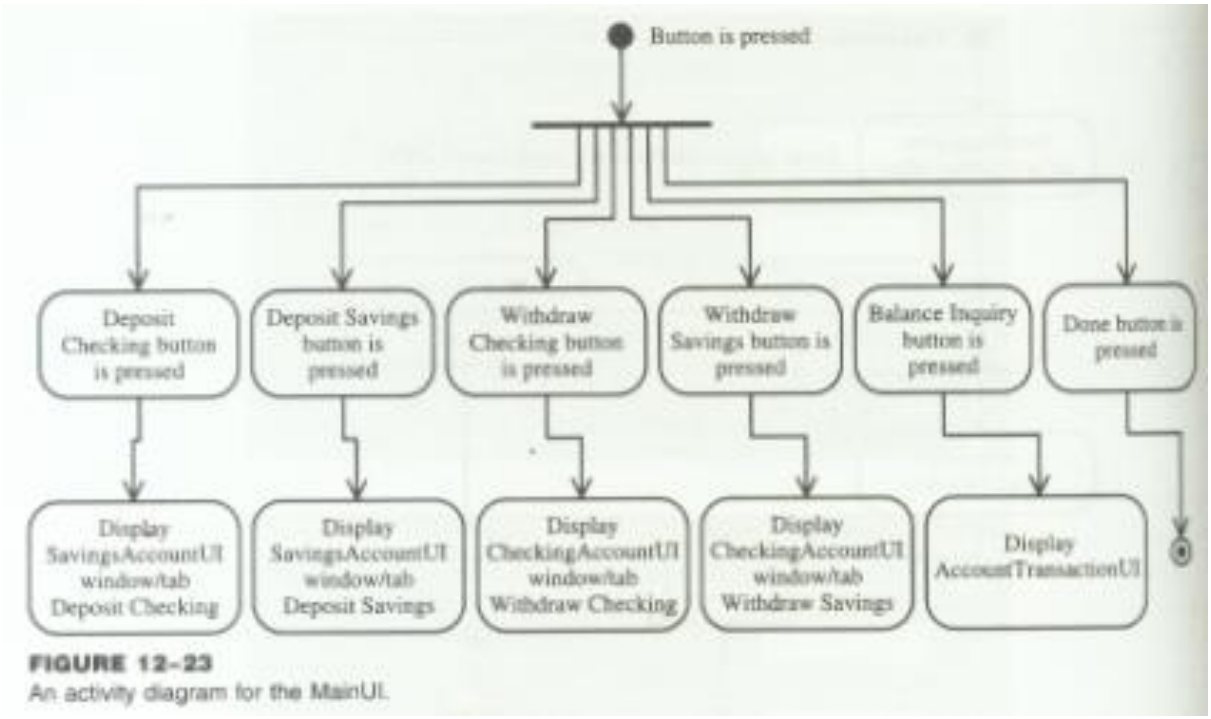


FIGURE 12-23
An activity diagram for the MainUI.

4. Describe in detail about designing interface objects / view layer objects .

- The creative process involves a curious and imaginative mind
- A broad background and fundamental knowledge of existing tools and methods
- A desire to complete and thorough job discovering solutions once a problem has been defined.
- To deal with uncertainty and ambiguity and to defer premature closure
- When designing the UI objects, a decision is taken as to how the users can interact with the system
- **View layer classes / interface objects**
- Objects that represent the set of operations in the business that users must perform to complete their tasks.
- Any objects that have direct contact with the outside world are visible in interface objects whereas business or access objects are more independent of their environment.
- **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.
- The knowledge of which message to and to which business object.
- Output - displaying or printing business objects
- This layer must give the pictorial representation of the business objects for the user.
- **Activities of view layer classes**
- The **macro level UI design process** - identifying view layer objects
- This activity takes place during the analysis phase of system development.
- Identify classes that interact with human actors by analyzing the use cases developed in the analysis phase.
- These use cases capture a complete, unambiguous and consistent picture of the interface requirements of the system.
- Sequence and collaboration diagrams allow the view of actor - system interaction and extrapolate interface classes that interact with human actors.
- **Macro level UI design activities**
- Designing the view layer objects by applying design axioms and corollaries
- Decide how to use and extend the components so that they support application - specific functions and provide the most usable interface
- **Prototyping the view layer interface**
- Prepare a prototype of some of the basic aspects of the design
- Testing Usability and user satisfaction
- Measure user satisfaction and its usability, focusing primarily on functionality
- Adoption of usability in the later stages of teh life cycle will not produce sufficient improvement of overall quality.

5. Describe the processes of creating the access layer classes.

- The access layer design process consists of the activities such as mirror business class package, mirror super - sub relationships and normalize classes and relationships
- If a class interacts with a **nonhuman** actor, such as another system, database or the web, then the class automatically becomes the access class.
- Process
- For every business class identified, mirror the business class package.
- For every business class that has been identified and created, create one access class in the access layer package.
- For example, if there are three business classes (Class1, Class2 and Class3), create three access layer classes (class1DB, class2DB and Class3DB)
- **Define relationships**

- The same rule as applies among business class objects also applies among access classes.
- **Simplify classes and relationships**
- Eliminate redundant or unnecessary classes or structures
- Combine simple access classes and simplify the super - sub classes
- **Redundant classes**
- If there is more than one class that provides similar services, e.g., translate request and translate results, select one and eliminate the other
- Method classes
- Revisit the classes that consists only one or two methods to find if they can be eliminated or combined with existing classes.
- If no such class can be found from the access layer package, select its associated class from the business package and add the methods as private. Now, an access method is created.
- Iterate and refine
- The access layer class not only stores the attributes but also the methods
- This is done by using OODBMS or RDBMS

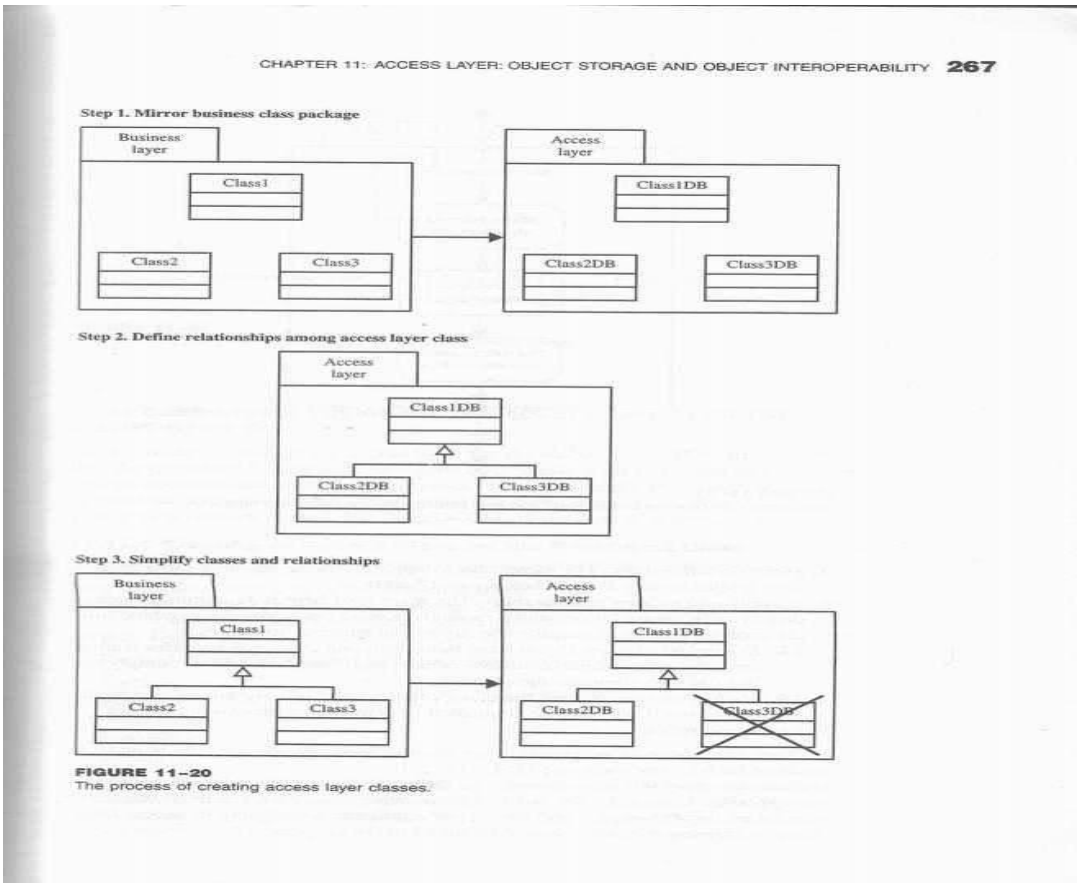


FIGURE 11–20
The process of creating access layer classes.

- For every business class identified, determine if the class has persistent data.
- an attribute can be either transient or persistent/ non transient
- An attribute is transient if the following condition exists
- Temporary storage for an expression evaluation or its value can be dynamically allocated.
- An attribute is persistent if the data exist between executions of a program or outlive the program.
- If the method has any persistent attributes, go to the next step - mirror the business class package.
- Otherwise, the class needs no associated access layer class.
- Mirror the business class package

- For every business class identified and created, create one access class in the access layer package.
- For example, if there are three business classes (Class1, Class2 and Class3), create three access layer classes (class1DB, class2DB and Class3DB)
- **Define relationships**
- The same rule as applies among business class objects also applies among access classes.
- **Simplify classes and relationships**
- Eliminate redundant or unnecessary classes or structures
- Combine simple access classes and simplify the super - sub classes
- **Redundant classes**
- If there is more than one class that provides similar services, e.g., translate request and translate results, select one and eliminate the other
- Method classes
- Revisit the classes that consists only one or two methods to find if they can be eliminated or combined with existing classes.
- Iterate and refine
- The access class that has been defined must be made as a-part of its business class.

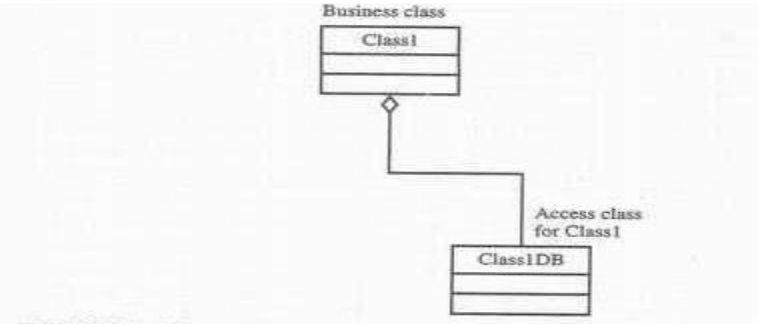


FIGURE 11–22
The relation between a business class and its associated access class.

6. Describe in detail about Model View Controller architectural pattern with an example.(Dec 16,17)

- A pattern is a proven solution to a problem in a context.
- Each pattern is a three-part rule which expresses a relation between a certain context, a problem, and a solution.
- Design patterns represent a solution to problems that arise when developing software within a particular context.

Categories are Design ,Architectural, Analysis ,Creational , Structural and Behavioral

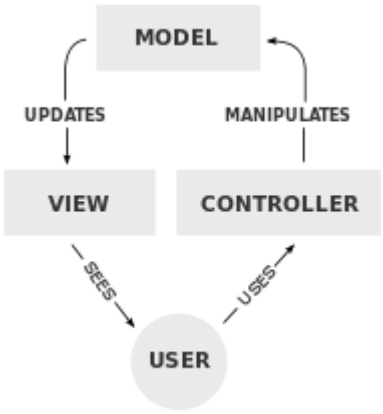
- **Model–view–controller** (**MVC**) is a software architectural pattern for implementing user interfaces.
- It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.
- MVC expresses the "core of the solution" to a problem while allowing it to be adapted for each system.

MVC Architecture

- The Model represents the structure of the data in the application, as well as application-specific operations on those data.
- A View (of which there may be many) presents data in some form to a user, in the context of some application function.

- A Controller translates user actions (mouse motions, keystrokes, words spoken, etc.) and user input into application function calls on the model.
- It selects the appropriate View based on user preferences and Model state.

Components



A **typical collaboration of the MVC components**

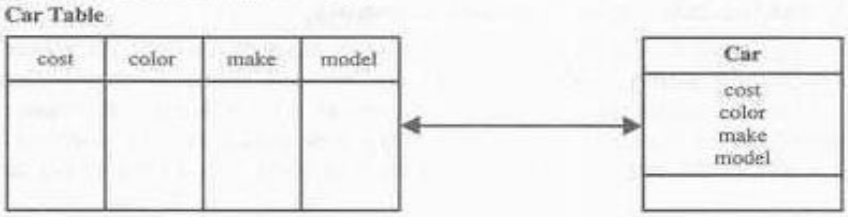
- The central component of MVC, the model, captures the behavior of the application in terms of its problem domain, independent of the user interface.
- The model directly manages the data, logic and rules of the application.
- A view can be any output representation of information, such as a chart or a diagram; multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
- The controller, accepts input and converts it to commands for the model or view.

- **Interactions**
- A **controller** can send commands to the model to update the model's state (e.g., editing a document).
- It can also send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document).
- A **model** notifies its associated views and controllers when there has been a change in its state.
- This notification allows the views to produce updated output, and the controllers to change the available set of commands.
- In some cases an MVC implementation might instead be "passive," so that other components must poll the model for updates rather than being notified.
- A **view** requests information from the model that it uses to generate an output representation to the user.
- *Use in web applications*
    - Model–view–controller has been widely adopted as an architecture for World Wide Web applications in major programming languages.
    - Several commercial and noncommercial web application frameworks have been created that enforce the pattern.
    - These frameworks vary in their interpretations, mainly in the way that the MVC responsibilities are divided between the client and server.
- Early web MVC frameworks took a thin client approach that placed almost the entire model, view and controller logic on the server.
- In this approach, the client sends either hyperlink requests or form input to the controller and then receives a complete and updated web page (or other document) from the view; the model exists entirely on the server.
- As client technologies have matured, frameworks such as AngularJS, Ember.js, JavaScriptMVC and Backbone have been created that allow the MVC components to execute partly on the client.

- **Details of MVC Design Pattern**
- Name (essence of the pattern)
  - Model View Controller MVC
- Context (where does this problem occur)
  - MVC is an architectural pattern that is used when developing interactive application such as a shopping cart on the Internet.
- Problem (definition of the reoccurring difficulty)
  - User interfaces change often, especially on the internet where look-and-feel is a competitive issue. Also, the same information is presented in different ways. The core business logic and data is stable.
- **Solution** (how do you solve the problem)
  - Use the software engineering principle of "separation of concerns" to divide the application into three areas:
    - **Model** encapsulates the core data and functionality
    - **View** encapsulates the presentation of the data there can be many views of the common data
    - **Controller** accepts input from the user and makes request from the model for the data to produce a new view.

7. Appraise with an example the process of designing classes. (May 2017)
   Refer previous answers
8. Write short notes on the following: i) Object-Relation mapping. ii) Object Interoperability with a suitable example.
   - **Object-Relation mapping**
   - A class has a set of attributes (properties / data members).
   - Object classes describe behaviour with methods
   - A tuple of a table can be correlated to an instance of a class contains the data for a single object.
   - 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.
   - 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***.
   - The method defines the behaviour of the object
   - **Mapping Capabilities**
   - They are two - way mappings - they map from the relational system to the object and from teh object to the relational system
   - **Table - class mapping**
   - **Table - multiple classes mapping**
   - **Table - inherited classes mapping**
   - **Tables - inherited classes mapping**
   - **Table - class mapping**
   - It is a one-to-one mapping of a table to a class and the mapping of columns in a table to properties in a class.
   - A single table is mapped to a single class.
   - All the columns may be mapped to properties.
   - It is sufficient to map only those columns for which an object model is required by the applications.
   - Each row in the table represents an object instance and each column represents an object attribute.
   - This one-to-one mapping of the table - class approach provides the transition between a relational data representation and an application object.
   - It is simple but offers little flexibility

FIGURE 11–11
Table-class mapping. Each row in the table represents an object instance and each column in the table corresponds to an object attribute.



- **Table -**
- A single
- Two or ıs in a single table.
- At run on a column value in the table.

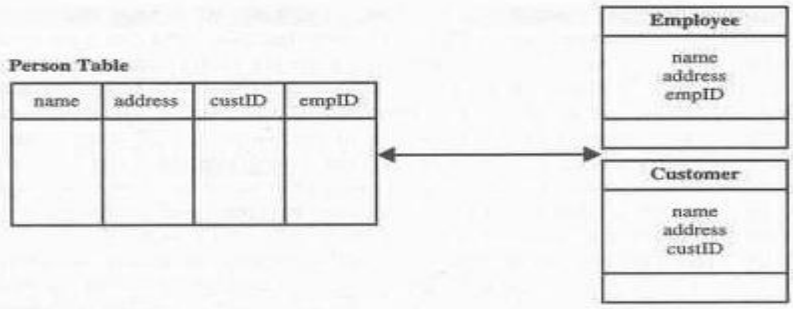


FIGURE 11–12
Table-multiple classes mapping. The custID column provides the discriminant. If the value for custID is null, an Employee instance is created at run time; otherwise, a Customer instance is created.

- 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.
- **Table - inherited classes mapping**
- A single table maps to many classes that have a common super class.
- The mapping allows the user to specify the columns to be shared among the related classes.
- The super class may either be an abstract or instantiated

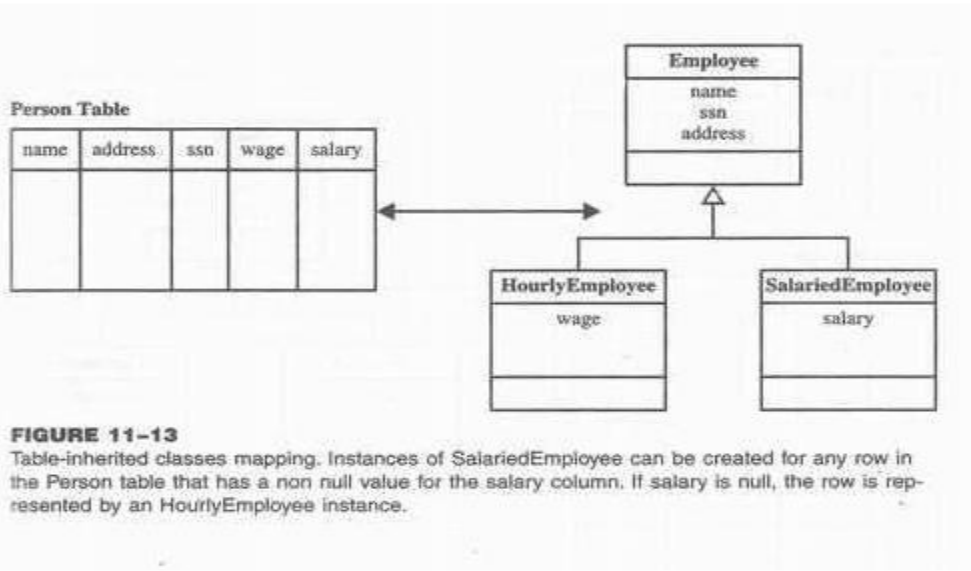


FIGURE 11–13
Table-inherited classes mapping. Instances of SalariedEmployee can be created for any row in the Person table that has a non null value for the salary column. If salary is null, the row is represented by an HourlyEmployee instance.

- The instances of salariedEmployee can be created for any row in the Person table that has a non null value for the Salary column.
- If Salary is null, the row is represented by an hourlyEmployee instance.
- **Tables - inherited classes mapping**
- It 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 is modeled by a primary key that acts as a foreign key to another table.
- In the object model, ***is - a*** is for an inheritance relationships that expresses clearer definition.

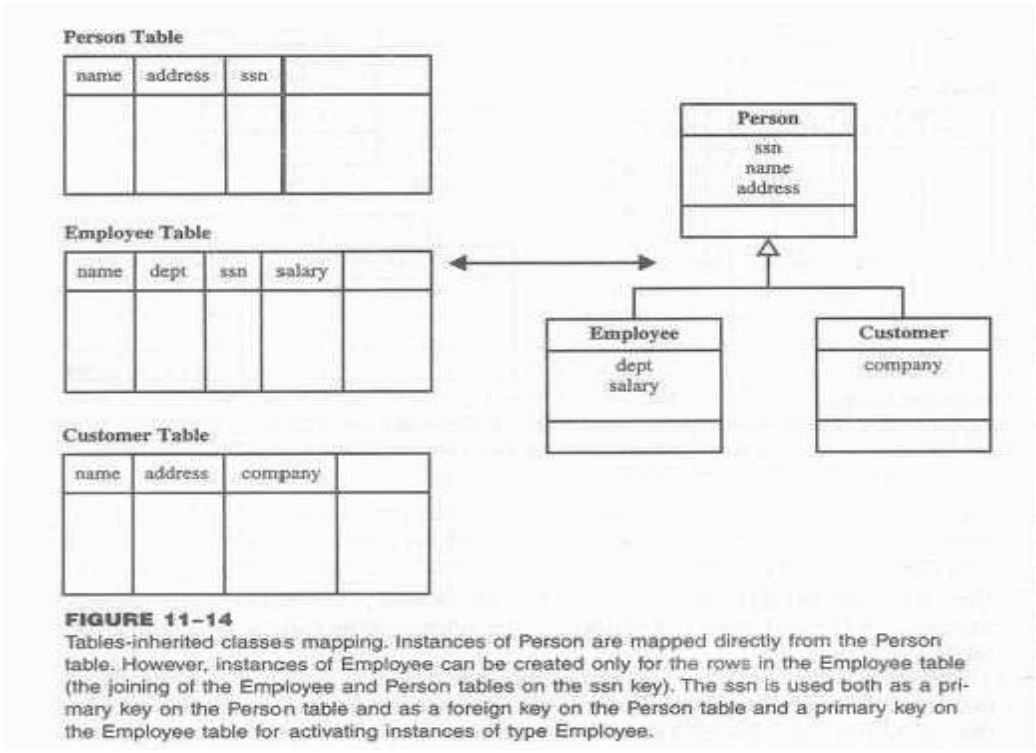


FIGURE 11–14
Tables-inherited classes mapping. Instances of Person are mapped directly from the Person table. However, instances of Employee can be created only for the rows in the Employee table (the joining of the Employee and Person tables on the ssn key). The ssn is used both as a primary key on 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.

- The departmentID of the Employee uses teh foreign key in column Employee-departmentID.
- Each employee instance has a direct reference of class Department(association) to teh department object to which it belongs.

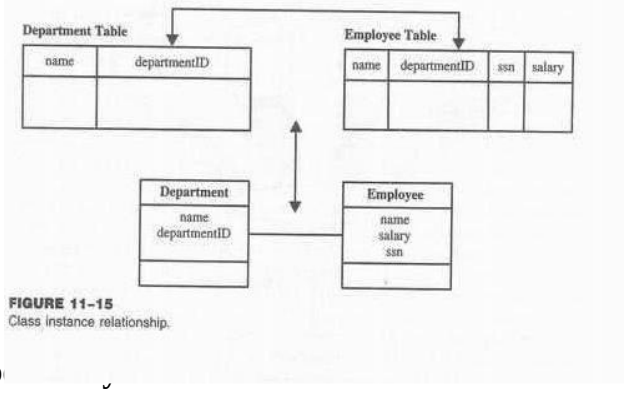


FIGURE 11–15
Class instance relationship.

Object Interop

- Distributed objects computing(DOC) utilizes reusable s/w components that can roam anywhere on network run on different platforms, communicate with legacy applications by means of object wrappers, manage themselves & resources they control.
- It promises the most flexible client-server systems
- They are reusable software components that can be distributed and accessed by users across the network.
- Applications no longer consist of clients and servers but users, objects and methods
- The user no longer needs 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.
- It can be used to integrate mission - critical applications and data residing on systems that are geographically remote.
- OMG specifies 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 will become the universal client-server middleware.
- The distributed component object model is an Internet and component strategy where ActiveX (OLE) plays the role of DCOM object.
- DOC standards : Object Management Group's (OMG) CORBA, Microsoft's Activex/DCOM (Both provides object interoperability), OpenDoc.

- **Common Object request Broker Architecture (CORBA)**
  - A standard proposed to integrate distributed, heterogeneous business applications & data
  - CORBA interface definition language (IDL) allows developers to specify language-neutral, object-oriented interfaces for application & system components
  - CORBA **Object Request Brokers (ORB)** implement a communication channel through which applications can access object interfaces & request data and services

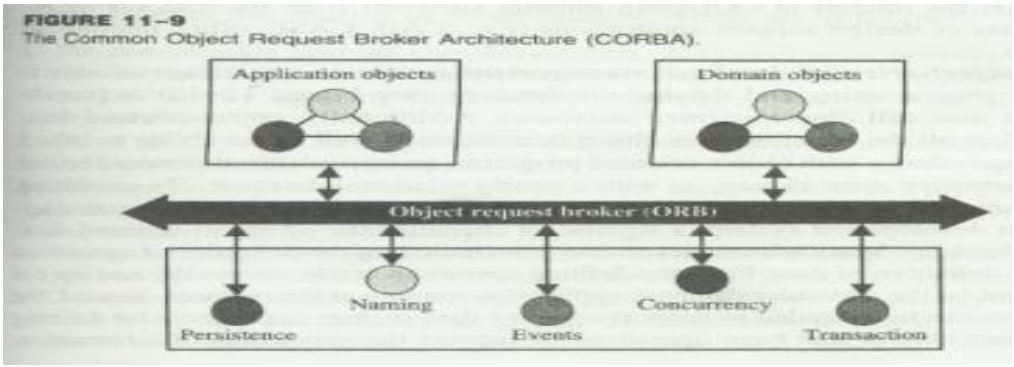
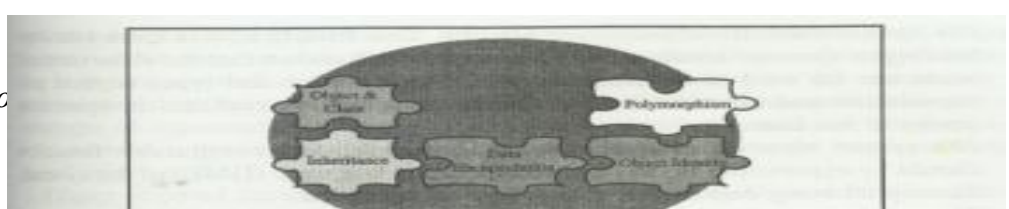


FIGURE 11–9
The Common Object Request Broker Architecture (CORBA).

- 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**
  - COM & DCOM are Microsoft's alternative to OMG's CORBA,ActiveX – formerly known as OLE

9. Explain the rules of OODBMS and compare O-O Databases with traditional databases.
   - ***An object-oriented database management system (OODBMS) is a database management system (DBMS) that supports the modeling and creation of data as objects.***

- It is a combination of object oriented programming and database technology that provide an integrated application development system



This includes some kind of support for classes of objects and the inheritance of class properties and methods by subclasses and their objects.

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

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.

Characteristics of OODBMS

- It includes Object oriented language properties and database requirements

Rules for object oriented system

The system must support complex objects.

- It 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

Object identity must be supported

- A data object must have an identity and existence independent of its values
- 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

The system must support inheritance

- Classes and types can participate in a class hierarchy
- It factors out shared code and interfaces

The system must support types (as defined in C++) and classes (defined in SmallTalk)

The system must avoid premature binding / late binding/ dynamic binding

- System must resolve conflicts in operation names at run time

The system must be computationally complete

- Any computable function should be expressible in the DML of the system.

The system must be extensible

- The user of the system should be able to create new types that have equal status to teh system's predefined types.
- It must be persistent, able to remember on object state
- It must be able to manage very large databases
- It must accept concurrent users
- It must be able to recover from hardware and software failures
- Data query must be simple

OODBMS	Traditional DBMS
Objects are an active component	Records play a passive role
Provide inheritance	No inheritance
Maintain data integrity regardless of system, network or media failure	It also does the same
Represent relationships explicitly	It is not so
Ability to interact with other objects and with itself.	It is the same
Allow representation and storage of data in the form of objects	The information is
Improved Access performance over relational value-based relationships	
Each object has its own identity	
The object identity is independent of the state of the object	
Object identity allows objects to be related and shared within a distributed computing network	

10. Explain in detail about the macro and micro level process of designing view layer classes.(May16)

Macro process consists of 2 steps

- 1. for every class identified, determine if the class interacts with a human actor. If so perform the following; otherwise move to the next class
  - 1.1 Identify view (interface) objects for the class

- 1.2 define the relationships among the view objects

– 2. Iterate and refine

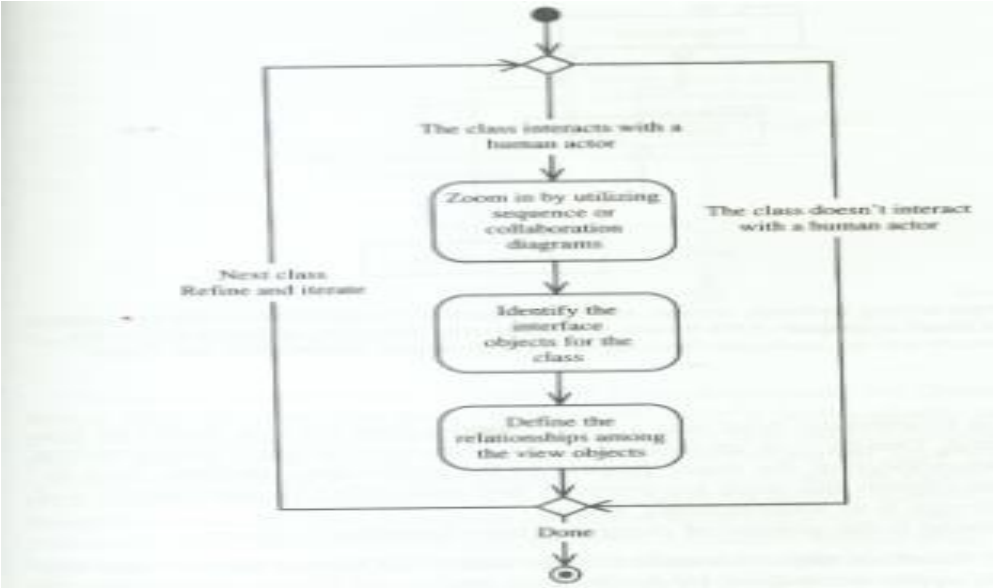


FIGURE 12–1
The macro-level design process.

Micro-level process

– 1. For every interface object identified in the macro design process, apply micro-level UI design rules and corollaries to develop the UI.

– 2. Iterate and refine

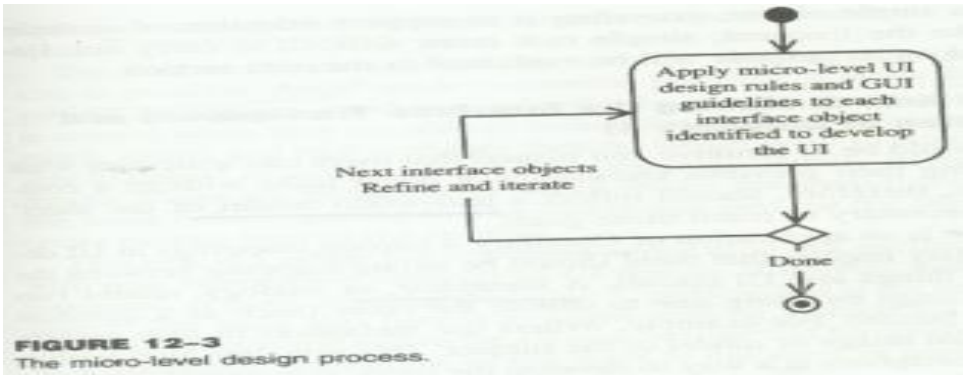


FIGURE 12–3
The micro-level design process.

11. Explain the design axioms and corollaries derived as a consequence of the axioms. (Dec 2016)
Axioms

- Fundamental truth
- Always valid for which there is no counter example / exception.
- Can't be proven / derived.

Theorem

- Proposition that may not be self evident.
- Can be proved from axioms
- Equivalent to law/principle
- Corollary
  - Proposition that follows from an axiom / another proposition that has been proven.
  - Valid / invalid
  - Similar to theorem.

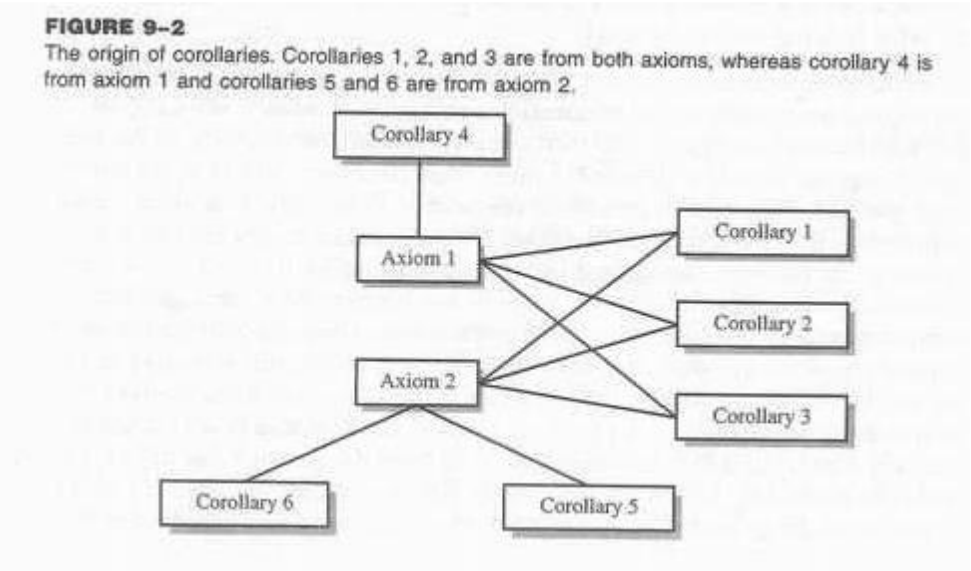
Axiom 1 Maintain independence among the components

Axiom 2 Minimize the information content of the design.

Corollaries - Design Rules

- Useful in making specific design decisions.
- Applied to actual situations more easily than axioms.
- Derived from two basic axioms.

FIGURE 9–2
The origin of corollaries. Corollaries 1, 2, and 3 are from both axioms, whereas corollary 4 is from axiom 1 and corollaries 5 and 6 are from axiom 2.



- **Corollary 1 - Uncoupled design with less information content**
- Single purpose.
- Large number of simple classes allow reusability.
- Strong mapping between analysis objects and design objects.
- Standardization.
- Design with inheritance.
- **Corollary 2 – single purpose**
- Every class should be Clearly defined classes
- The documentation of a class should be able to easily explain its purpose .
- It must be simple, more precise

- Each method is to provide only one service.
- **Corollary 3 – Large number of simpler classes reusability**
- Smaller the classes, the better you reuse.
- Large and complex classes are too sophisticated to be reused.

- OOD puts encapsulation, modularization and polymorphism for reuse.
  - E.g., description of software IC library framework between OOD and building network using IC chips.
- Rarely practiced effectively (C & Y)
- Organizations service which have achieved high levels of reusability.
- Institutionalized approach software created intentionally to be reused.
  - ***Corollary 4 – strong mapping***
  - OOA & OOD based on same models.
  - A strong mappings links classes during design phase.
  - ***Corollary 5 – Standardization***
  - It requires good understanding of the classes in the OOPs environment.
  - Small talk, Java, C++ , PowerBuilder has several built-in class libraries.
  - Class libraries must easily be searched, based on users' criteria
  - Design patterns 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***
  - Inheritance minimizes the amount of program instructions.
  - Determine the ancestor, attributes, messages of a class.
  - Construct the methods and protocols of a class.
  - **Single inheritance**

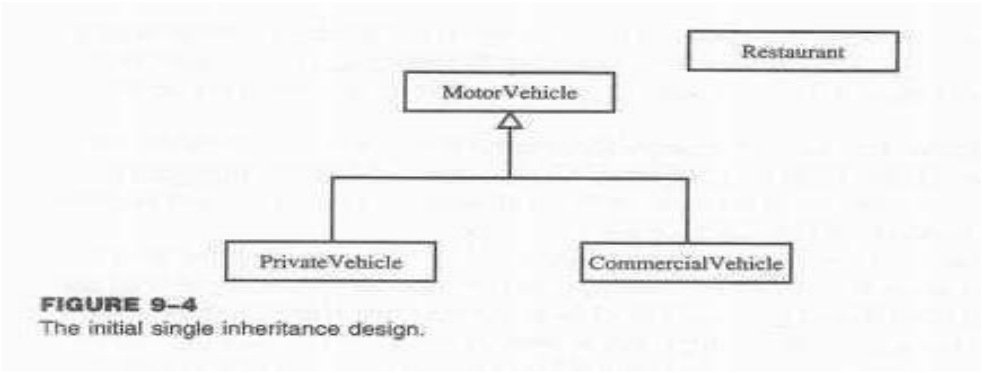


FIGURE 9–4
The initial single inheritance design.

- Redesign the application by using the inheritance mechanisms supported by the system's target language.
- If the language supports single inheritance exclusively, select a formal super class.
- Identify other related sub classes and relevant methods.
- **Advantages of single inheritance**
- It avoids ambiguity in the selection of methods by a class.
- It concentrates only on the specific behaviour of an object.

FIGURE 9–5
The single inheritance design modified to allow licensing food trucks.

- **Multiple Inheritance in a single inheritance system**
- LISP and C++ support multiple inheritance where objects can inherit behaviour from unrelated areas of the class tree.
- Disadvantages
- How to determine which behaviour to get from which class when several ancestors define the same method.
- It is more difficult to understand programs written in a multiple inheritance system
- To achieve 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.
-

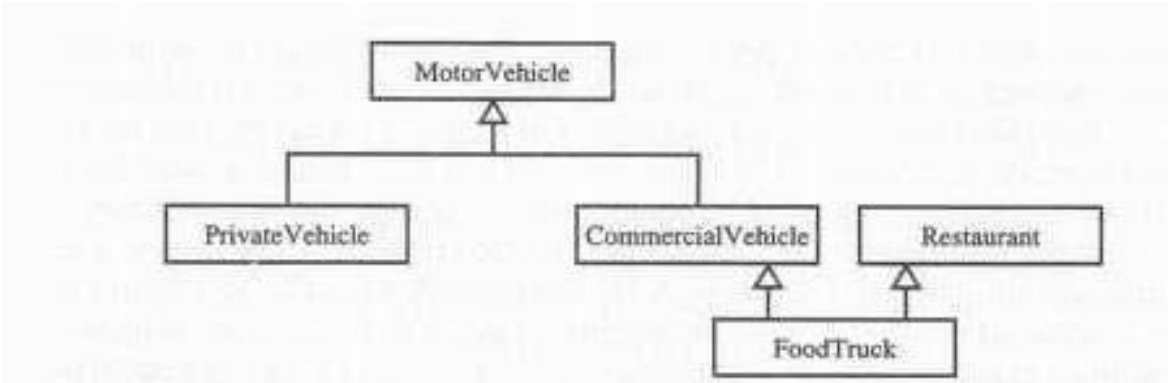


FIGURE 9–7
Multiple inheritance design of the system structure.

- **Avoiding inheriting inappropriate behaviour**
- Before a class inherits, the following questions can be considered
- Is the sub class fundamentally similar to its super class? (high inheritance coupling)
- Is it an entirely new information that provide some expertise from its super class (low inheritance coupling)
- If low inheritance coupling is preferred, add an attribute that incorporates the proposed super class's behaviour rather than an inheritance from the super class.

12. Write short notes on the following i)Explain about object relational systems (May 2015)
ii) Discuss the various design corollaries (May 2015/2016)

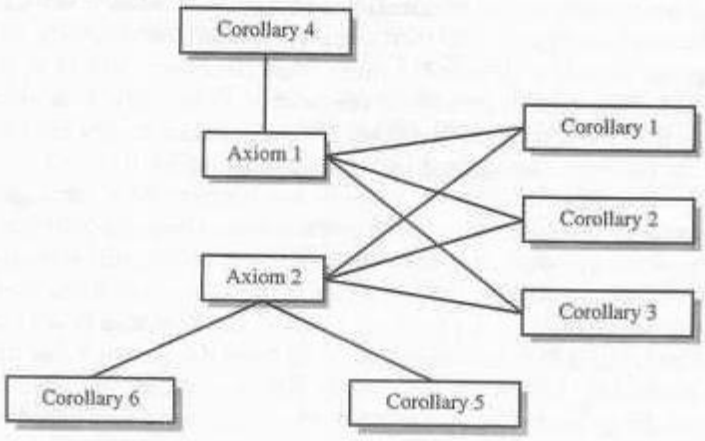
object relational systems

design corollaries

Corollaries - Design Rules

- Useful in making specific design decisions.
- Applied to actual situations more easily than axioms.
- Derived from two basic axioms.

FIGURE 9–2
The origin of corollaries. Corollaries 1, 2, and 3 are from both axioms, whereas corollary 4 is from axiom 1 and corollaries 5 and 6 are from axiom 2.



- **Corollary 1 - Uncoupled design with less information content**
- Single purpose.
- Large number of simple classes allow reusability.
- Strong mapping between analysis objects and design objects.
- Standardization.
- Design with inheritance.
- **Corollary 2 – single purpose**
- Every class should be Clearly defined classes
- The documentation of a class should be able to easily explain its purpose .
- It must be simple, more precise
- Each method is to provide only one service.
- **Corollary 3 – Large number of simpler classes reusability**
- Smaller the classes, the better you reuse.
- Large and complex classes are too sophisticated to be reused.

- OOD puts encapsulation, modularization and polymorphism for reuse.
  - E.g., description of software IC library framework between OOD and building network using IC chips.
- Rarely practiced effectively (C & Y)
- Organizations service which have achieved high levels of reusability.
- Institutionalized approach software created intentionally to be reused.
  - ***Corollary 4 – strong mapping***
  - OOA & OOD based on same models.
  - A strong mappings links classes during design phase.
  - ***Corollary 5 – Standardization***
  - It requires good understanding of the classes in the OOPs environment.
  - Small talk, Java, C++ , PowerBuilder has several built-in class libraries.

- Class libraries must easily be searched, based on users' criteria
- Design patterns 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***
- Inheritance minimizes the amount of program instructions.
- Determine the ancestor, attributes, messages of a class.
- Construct the methods and protocols of a class.
- **Single inheritance**

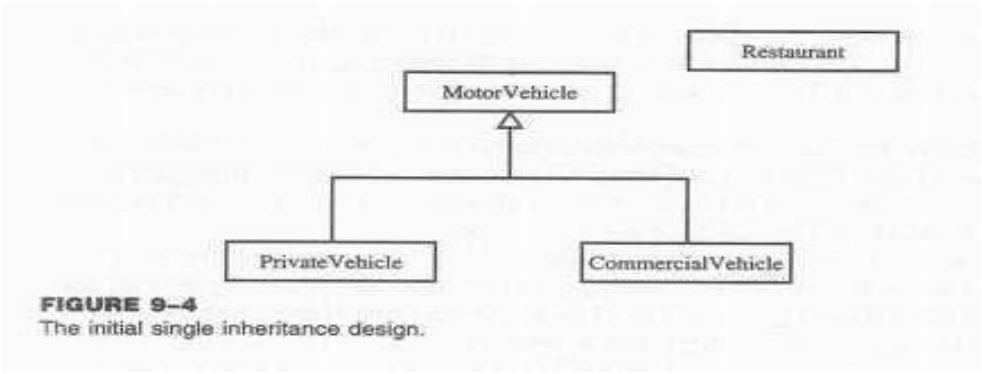
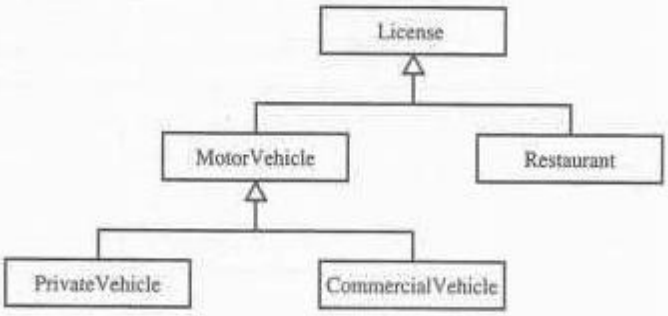


FIGURE 9–4
The initial single inheritance design.

- Redesign the application by using the inheritance mechanisms supported by the system's target language.
- If the language supports single inheritance exclusively, select a formal super class.
- Identify other related sub classes and relevant methods.
- **Advantages of single inheritance**
- It avoids ambiguity in the selection of methods by a class.
- It concentrates only on the specific behaviour of an object.

FIGURE 9–5
The single inheritance design modified to allow licensing food trucks.



- **Multiple Inheritance in a single inheritance system**
- LISP and C++ support multiple inheritance where objects can inherit behaviour from unrelated areas of the class tree.
- Disadvantages
- How to determine which behaviour to get from which class when several ancestors define the same method.
- It is more difficult to understand programs written in a multiple inheritance system

- To achieve 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.
-

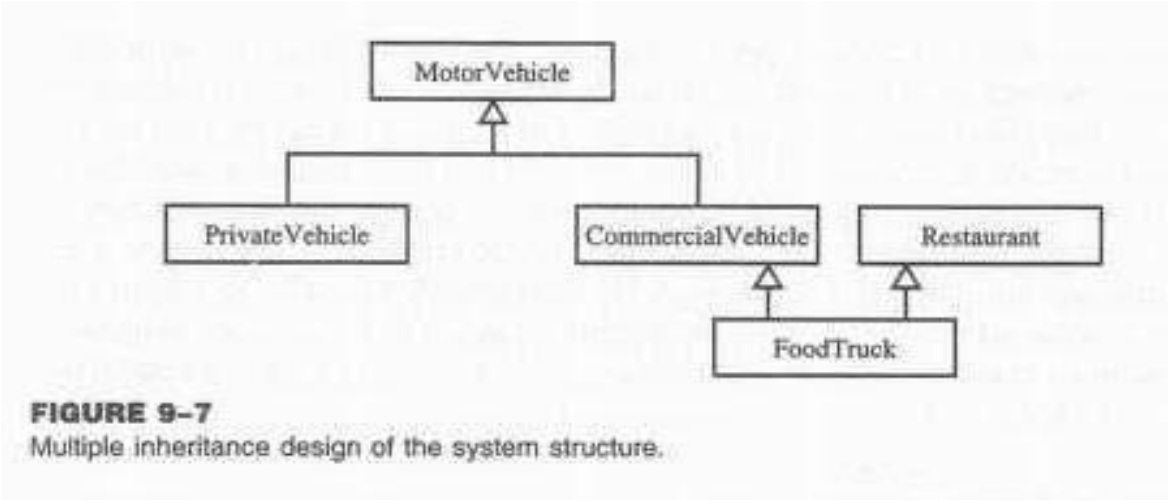


FIGURE 9–7
Multiple inheritance design of the system structure.

- **Avoiding inheriting inappropriate behaviour**
- Before a class inherits, the following questions can be considered
- Is the sub class fundamentally similar to its super class? (high inheritance coupling)
- Is it an entirely new information that provide some expertise from its super class (low inheritance coupling)
- If low inheritance coupling is preferred, add an attribute that incorporates the proposed superclass's behaviour rather than an inheritance from the super class.

UNIT – V

1. i)Explain in detail about the quality assurance testing.

iii) Sketch the various guidelines for developing quality assurance test cases and test plans.

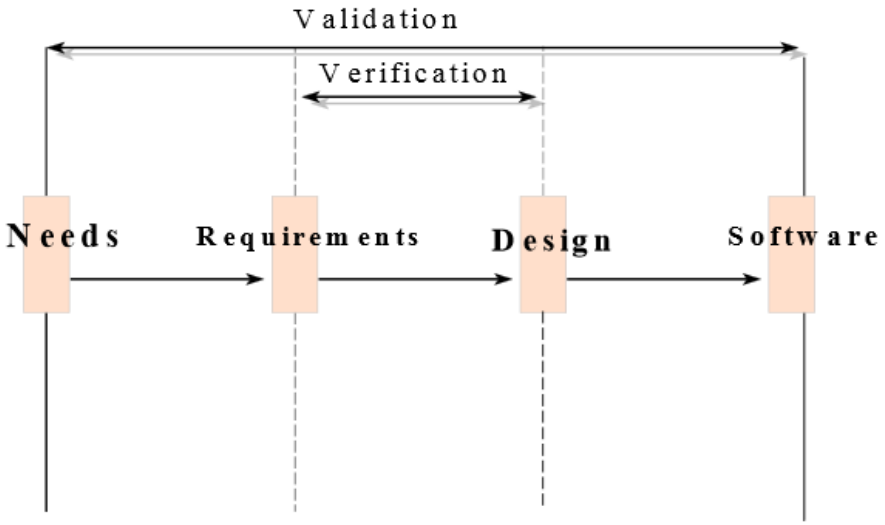
Quality Measures

Verification

- "Am I building the product right?"

Validation

- "Am I building the right product?"



Quality Assurance testing can be divided into 2 major categories.

i) Error Based Testing techniques search a given class's methods for particular clues of interest, then describe how these clues should be tested.

Eg: compute payroll method of employee class need to be tested.

Employee.computepayroll(hours)

To test this method we must try different values for hours (say 40,0,-10,100) to see of the program can handle them. (also known as testing the Boundary condition). This method should be able to handle any value; if not, error must be recorded and reported.

ii) Scenario based Testing (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. It can also identify interaction bugs.

- they often are more complex and realistic than error-based tests.
- they tend to exercise multiple subsystems in a single test.

Testing Objectives

- Testing is a process of executing a program with the intent of finding an error.
- A good test case is one that has a high probability of finding an as-yet undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

Testing Principles

1. All tests should be traceable to customer requirements.
2. Tests should be planned long before testing begins.
3. The pareto principle applies to software testing.
4. Test should begin " in the small" and progress toward testing "in the large"
5. Exhaustive testing is not possible.

White-box testing

White box testing is also called structural testing or glass box testing.

- The internal program logic is checked
- logical paths thro the s/w are tested
- status of program is examined

- all independent paths within a module is tested
- It tests the code
- It is applied in the early stage of testing process. It uses the control structure of the procedural design to test cases.

Sample Application

In-circuit testing is a good example of a white-box system testing where the tester is looking at the interconne between different components of the application and verifying the proper functioning of each internal conne We can also consider the example of an auto-mechanic who takes care of the inner workings of a vehicle to that all the components are working correctly to ensure the proper functioning

of the vehicle.

Basic Path Testing

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. Cyclomatic complexity has a foundation in graph theory and is computed in one of three ways:

1. The number of regions corresponds to the cyclomatic complexity.
2. Cyclomatic complexity, V(G), for a flow graph, G, is defined as
   V(G)=E-N+2 where E is the number of flow graph edges, and N is the number of flow graph nodes.
3. Cyclomatic complexity, V(G), for a flow graph, G is also defined as
   V(G)=P=1

   where P is the number of predicate nodes contained in the flow graph G.

Black box testing/behavioral testing

- It exercises all the functional requirements of the software.
- It does not know the internal working of the product.
- It specifies the functions of the software.
- It demonstrates that software functions are operational, ie whether the i/p is accepted and output is co produced, and that the integrity of external information (database) is maintained.
- It does not check the logical errors present in the application.

Sample Application

Search engine is a very good example of a black box system. We enter the text that we want to search, by pressing “search” we get the results. Here we are not aware of the actual process that has been implemented to get the results. We simply provide the input and get the results

System testing techniques

Recovery Testing

Security Testing

Stress Testing

Performance Testing

Regression Testing

Unit Testing

- focuses verification effort on the smallest unit of s/w design.
- control paths are tested to uncover errors within the boundary of the module.
- focuses on the internal processing logic and data structures within the boundaries of the component.
- can be conducted in parallel for multiple components.

Integration Testing

Integration testing is a systematic technique for constructing the s/w architecture while at the same time " conducting tests to uncover errors associated with interfacing". The objective is to take unit tested components and build a program structure that has been dictated by design.'

There are two types of integration

1. non incremented integration
2. incremental integration

Software reliability

Software reliability is the probability of failure-free operation of a computer program in a specified environment for a specified time.

A simple measure of reliability is mean-time –between –failures.(MTBF)

MTBF = MTTF + MTTR

Availability is the probability that a program is operating according to requirement at a given point in tim is defined as

Availability = [MTTF / (MTTF + MTTR)]*100%

Defect removal efficiency.

To assess the real time quality of software, engineers use certain technical measures. Defect removal effi (DRE) is one such quality metric that provides benefit at the project level. It is actually a measure of filtering abi quality assurance and control activities as they are applied throughout all process framework activities.

DRE = E / E+D

ii) Sketch the various guidelines for developing quality assurance test cases and test plans

SAMPLE BUG REPORT:

Bug Name: Application crash on clicking the SAVE button while creating a new user.
Bug ID: (It will be automatically created by the BUG Tracking tool once you save this bug)
Area Path: USERS menu > New Users
Build Number: Version Number 5.0.1
Severity: HIGH (High/Medium/Low) or 1
Priority: HIGH (High/Medium/Low) or 1
Assigned to: Developer-X
Reported By: Your Name
Reported On: Date
Reason: Defect
Status: New/Open/Active (Depends on the Tool you are using)
Environment: Windows 2003/SQL Server 2005

Description:
Application crash on clicking the SAVE button while creating a new user, hence unable to create a new user in the application.

Steps To Reproduce:
1) Logon into the application
2) Navigate to the Users Menu > New User
3) Filled all the user information fields
4) Clicked on 'Save' button
5) Seen an error page "ORA1090 Exception: Insert values Error…"
6) See the attached logs for more information (Attach more logs related to bug..IF any)
7) And also see the attached screenshot of the error page.

Expected result: On clicking SAVE button, should be prompted to a success message "New User has been created successfully".

2. What are the steps involved in Cryptanalysis explain with example.(May 2016)

Cryptanalysis refers to the study of ciphers, cipher text, or cryptosystems (that is, to secret code systems) with a view to finding weaknesses in them that will permit retrieval of the plaintext from the ciphertext, without necessarily knowing the key or the algorithm. This is known as breaking the cipher, ciphertext, or cryptosystem.

1) Known-plaintext analysis: With this procedure, the cryptanalyst has knowledge of a portion of the plaintext from the ciphertext. Using this information, the cryptanalyst attempts to deduce the key used to produce the ciphertext.

2) Chosen-plaintext analysis (also known as differential cryptanalysis): The cryptanalyst is able to have any plaintext encrypted with a key and obtain the resulting ciphertext, but the key itself cannot be analyzed. The cryptanalyst attempts to deduce the key by comparing the entire ciphertext with the original plaintext. The Rivest-Shamir-Adlemanencryption technique has been shown to be somewhat vulnerable to this type of analysis.

3) Ciphertext-only analysis: The cryptanalyst has no knowledge of the plaintext and must work only from the ciphertext. This requires accurate guesswork as to how a message could be worded. It helps to have some knowledge of the literary style of the ciphertext writer and/or the general subject matter.

4) Man-in-the-middle attack: This differs from the above in that it involves tricking individuals into surrendering their keys. The cryptanalyst/attacker places him or herself in the communication channel

between two parties who wish to exchange their keys for secure communication (via asymmetric or public key infrastructure cryptography). The cryptanalyst/attacker then performs a key exchange with each party, with the original parties believing they are exchanging keys with each other. The two parties then end up using keys that are known to the cryptanalyst/attacker. This type of attack can be defeated by the use of a hash function.

5) Timing/differential power analysis: This is a new technique made public in June 1998, particularly useful against the smart card, that measures differences in electrical consumption over a period of time when a microchip performs a function to secure information. This technique can be used to gain information about key computations used in the encryption algorithm and other functions pertaining to security. The technique can be rendered less effective by introducing random noise into the computations, or altering the sequence of the executables to make it harder to monitor the power fluctuations. This type of analysis was first developed by Paul Kocher of Cryptography Research, though Bull Systems claims it knew about this type of attack over four years before.

In addition to the above, other techniques are available, such as convincing individuals to reveal passwords/keys, developing Trojan horse programs that steal a victim's secret key from their computer and send it back to the cryptanalyst, or tricking a victim into using a weakened cryptosystem.

3. What is a test plan? Develop a test plan for Internet Banking Application. What do you mean by usability and user interface testing? List the guidelines for developing usability testing.

A test plan is developed to detect and identify potential problems before delivering the software to its users. The test plan need not be very large; in fact, devoting too much time to the plan can be counterproductive. Test plan must include the type of testing to be done.

Test plan for Internet Banking Application

1) Test Plan id: IBS-ST-TP_001

2) Introduction

It is system test plan for Internet banking System, internet web application, provides access to Account holders and guest users from any ware in the world. It has two interfaces one is Admin interface another is user interface. Admin can be accessed by Bank Authorized users, user interface can be accessed by bank account holders and guest users.

The purpose of the system(Application) is to provide bank information and services online, Bank account get banking services from any ware, without visiting the bank branches.

3) Test Items:

- Admin Interface
- Master Data
- User Management
- Reports
- User Interface
- Information
- Personal Banking
- Corporate banking
- Business Etc.,

4) References

- Requirements
- Project Plan
- Test Strategy
- Use cases(if available)
- High Level Design doc
- Low Level design docs
- Process guide line doc

- Prototypes

5) Features to be tested:

a) Admin Interface:

i) Master Data

1) Add new branch, edit branch/ delete branch
2) Add new ATM
3) Add new loan type
4) Add new account type
5) Add new deposit type

ii) User Management

1) Create new user
2) Edit user
3) Delete user

iii) Reports

1) Branch wise report
2) User wise report
3) Day, month yearly reports
4) Service wise report(only loans, only new account, fixed deposits)

b) User Interface:

i)Information

1) Branch locators
2) ATM locators
3) Loans information
4) Bank history
5) Bank financial details
6) Fixed deposits information
7) Calculators

ii) Personal Banking

1) Login
2) Balance enquiry
3) Bill payment(utilities, Subscriptions)
4) Fund transfer(transfer to same bank, others bank)
5) Statement generation (mini stmt, detailed report)

iii) Corporate Banking

1) Add user, Edit user, Delete user
2) Balance enquiry
3) Money transfer
4) Payroll
5) Reports
6) Features not to be tested:

6) NA

7) Entry Criteria

a) Test Design

Team formation, responsibilities, Schedule, Requirements, Test Case Template
Training on domain, on Automation tools

b) Test Execution

- Readiness of test lab
- Readiness of AUT
- Requirements
- Test Case docs.
- Test Data
- Defect Report Template

8) Exit Criteria

- All possible test cases executed.
- Maximum defects fixed, final Regression performed successfully
- Confidence on test process
- Time Limitations
- Budget Limitations

9) Suspension Criteria

- Show-stopper bug found
- Supplier issues
- Vast change in requirements
- If resolving defects are more

10) Roles and responsibilities

S.No.	Name	Role	Responsibilities	Remarks
1	Kareemulla SK	Test Lead	Test Planning, guidance, Monitoring and Test control	
2	Venkat Rao. P	Sr.Tester	Test Data Documentation, Generating Test Scenarios.	
3	Swapna. Dk	Tester	Test Case Documentation, test execution. Defect reporting and tracking for personal banking module.	
4	Srinivas V	Tester	Test Case Documentation, Test execution, defect reporting and tracking for Personal banking module.	
5	Suneetha B	Tester	Test Case Documentation, Test execution , defect reporting and tracking for corporate banking module	

11) Schedule:

S.No.	Task	Days	Duration	Remarks
1	Understanding & Analyzing Requirements	5	2nd july to 6th July	
2	Review meeting	01	9th July	
3	Generating Test Scenarios	10	11th July to 26th July	
4	Reviews	02	25th July to 12th Aug	
5	Test Case Documentation	40	20th July to 12th Aug	
6	Reviews	04	14th Aug to 18th Aug	
7	Test Data Collection	06	20th Aug to 26th Aug	
8	Test data collection	06	20th Aug to 26th Aug	

9	Reviews	01	28th Aug	
10	Verifying Test Environment Setup	01	20th AUG	
11	Create Test Batches	02	30th ,31st Aug	
12	Sanity Testing	01	3rd Sep	
13	Comprehensive Testing	25	4th SEP to 2nd OCT	
14	Sanity Testing	01	3rd Oct	
15	Selecting Test Cases	02	4th and 5th OCT	
16	Regression Testing	05	8th OCT to 12th Oct	
17	Sanity Testing	01	15th Oct	
18	Selecting Test cases	01	16th Oct	
19	Regression Testing Cycle 2	04	17th Oct to 22nd Oct	
20	…			
21	…			
22	Final Regression	08	19th Nov to 28th Nov	
23	Evaluating exit criteria	01 OR 02	29TH, 30TH Nov	
24	Collcting all artifacts	02	3rd, 4th Dec	
25	Test Summary Report	01	5th Dec	

Note: Regression testing depends on application and strength of development team.

12) Training
- Training program on banking domain
- Test Automation Training using QTP tool

13) Risks & Mitigations
- Team member's issues
- Vendor issues
- Time
- Budget

14) Test Environment/ Lab

Application type, Web Application, Internet and public

Server side:
- Windows 2003 server
- Unix server

Ms Exchange server

a) Web server b) EDP c) Data storage
   - Bugzilla Tool
   - VSS
   - MS office
   - QTP tool
   - Browser IE 7.0

Client side
- Windows XP +SP2
- VSS
- MS office
- QTP

15) Test Deliverables
- Test Plan
- Review reports
- RTM

- Test Scenario docs
- Test data
- Opened, closed defect reports
- Test summary report

16) Approvals

S.No.	Tasks	Author/Role	Date & Signature
1	Test Plan Documentation	Kareemulla Sk(test Lead	
2	Review	Hari Prasad(QA Analist)	
3	Approval	Vinod Rao(project Manager)	

17) Glossary

- AUT-Application Under test
- PIN-Project Initiation Note
- SRS- Software Requirements Specification

4. What do you mean by usability and user interface testing? List the guidelines for developing usability testing.

System Usability & Measuring User Satisfaction

- Verification - "Am I building the product right?"
- Validation - "Am I building the right product?"
- Two main issues in software quality are validation or user satisfaction and verification or quality assurance.
- The process of designing view layer classes consists of the following steps:

1. In the macro-level user interface (UI) design process, identify view layer objects.
2. In the micro-level UI, apply design rules and GUI guidelines.
3. Test usability and user satisfaction.
4. Refine and iterate the design.

Usability and User Satisfaction Testing

Two issues will be discussed:

1. Usability Testing and how to develop a plan for usability testing.
2. User Satisfaction Test and guidelines for developing a plan for user satisfaction testing.

- The International Organization for Standardization (ISO) defines usability as the effectiveness, efficiency, and satisfaction with which a

  specified set of users can achieve a specified set of tasks in particular environments.

Defining tasks. What are the tasks?

Defining users. Who are the users?

A means for measuring effectiveness, efficiency, and satisfaction.

The phrase two sides of the same coin is helpful for describing the relationship between the usability and functionality of a system.

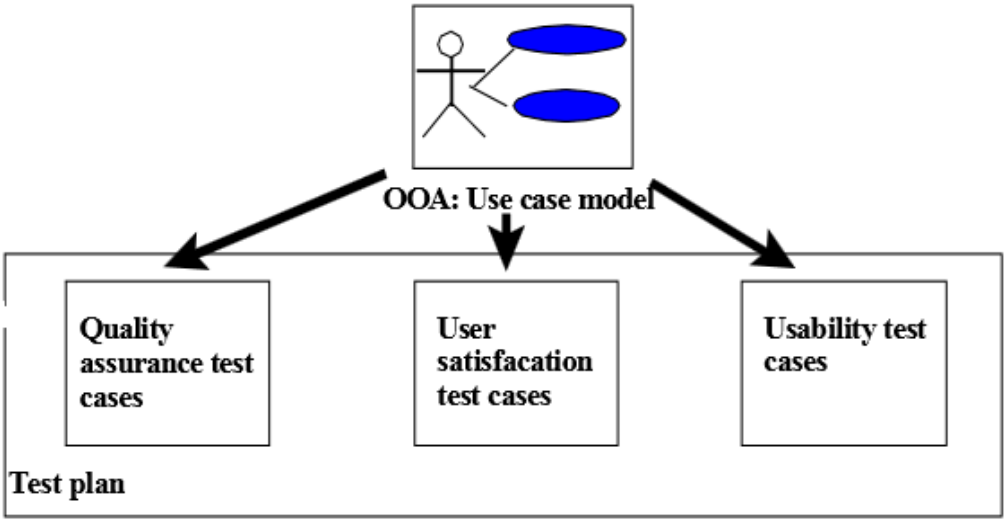
Usability testing measures the ease of use as well as the degree of comfort and satisfaction users have with the software.

Usability testing must begin with defining the target audience and test goals.

Run a pilot test to work out the bugs of the tasks to be tested.

Make certain the task scenarios, prototype, and test equipment work smoothly.

- Usability testing should begin in the early stages of product development, when developing use cases.
- The findings from usability testing can be incorporated into the usability test plan and test cases.
- Usability testing must be a key part of the UI design process



Guidelines For Developing Usability Testing

"Focus groups" are helpful for generating initial ideas or trying out new ideas.

It requires a moderator who directs the discussion about aspects of a task or design

- Apply usability testing early and often.
- Include all of a software's components in the test.
- The testing doesn't need to be very expensive, a tape recorder, stopwatch, notepad and an office can produce excellent results
- 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.
- Focus on tasks, not features.
- Remember that your customers will use features within the context of particular tasks

Recording the Usability Test

- Do not interrupt participants during a test.
- If they need help, begin with general hints before moving to specific advice.
- Keep in mind that less intervention usually yields better results.
- Ask subjects to think aloud, so you can hear what assumptions and inferences they are making.
- Record how long they take to perform a task as well as any problems they encounter.
- Record the test results using a portable tape recorder, or better, a video camera.

- ❖ You may also want to follow up the session with the user satisfaction test

5. Explain in detail about automated testing tools. Write the advantages.(May 2016, Dec 2016)
   The use of software testing tools can dramatically simplify testing, increase the defect find rate and ultimately achieve a higher release quality. A testing tool is needed to ensure that your system or your application is in a good condition. Testing tools are often helpful for those who wish to automate tests on he applications that they are developing.
   A testing tool can also be used by the users. There are a wide variety of software test tools that address many aspects of the testing process; they can be applied to different types of software, different programming languages, and address different types of testing. The scope and quality of such tools vary widely and provide varying degrees of assistance.
   Types of Software Testing Tools can be categorized by the testing activity or the process they are utilized.
   1. There are drive testing tools for your hard drives so that you will know how fast it can write, read and access the data.
   2. There are also tools for testing the CD or DVD drives
   3. The optical drive tests are very useful in knowing how quick the CD or DVD can burn files or folders,
   4. There are also tools that will enable you to see the health of your system as a whole. This is because it can give you the current health of your hard drive as well as outcomes about the speed tests on broadband, DSL, VoIP and many more.
   5. There are web service testing tools that can help you in checking the efficiency of your site when developing a website to verify that your site works perfectly whenever a visitor logs in and out.
   6. Integration testing is generally managed by the data architect or software designer in an integration test environment which "mirrors" the intended production environment.
   7. Functional testing is done to ensure that each separate function of the software works independently as well as a group.
   8. Performance testing is done to ensure your software performs in the manner in which it was designed to perform.
   9.Unit testing required for testing of the individual software units
   to make sure they continue to operate and function as a whole and individually.
   10. Compatibility and usability should also be tested before release of the software. It is important that the software be tested fully for loadability and traffic flow.
   By employing these tools means the more improvements you will see with productivity increases.
   Create Test Cases
   Identifying workload profiles and key scenarios generally does not provide all of the information necessary to implement and execute test cases. Additional inputs for completely designing a stress test include performance objectives, workload characteristics, test data, test environments, and identified metrics. Each test design should mention the expected results and/or the key data of interest to be collected, in such a way that each test case can be marked as a "pass," "fail," or "inconclusive" after execution.

   The following is an example of a test case based on the order-placement scenario.
   Test 1 – Place Order Scenario
   Workload: 1,000 simultaneous users.
   Think time: Use a random think time between 1 and 10 seconds in the test script after each operation.
   Test Duration: Run the test for two days.
   Expected results:
   Application hosting process should not recycle because of deadlock or memory consumption.
   Throughput should not fall below 35 requests per second.
   Response time should not be greater than 7 seconds for 95 percent of total transactions completed.

"Server busy" errors should not be more than 10 percent of the total response because of contention-related issues.

Order transactions should not fail during test execution. Database entries should match the "Transactions succeeded" count.

5. Propose an object oriented design for an inventory control system. State the functional requirements you are considering.(Dec 2016)

i)Explain in detail about user satisfaction test

ii) Explain about the test cases and User satisfaction test for Bank ATM System.

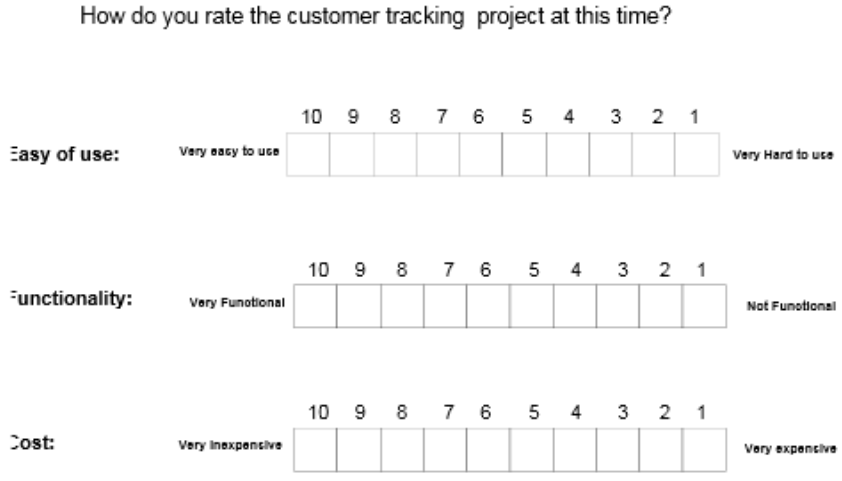
User Satisfaction Test

- User satisfaction test is the process of quantifying the usability test with some measurable attributes of the test, such as functionality, cost, or ease of use.
- The best measure of user satisfaction is the product itself, since you can observe how users are using it, or avoiding it.
- As a communication vehicle between designers, as well as among users and designers.
- To detect and evaluate changes during the design process.
- To provide us with 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.
- The test is inexpensive, easy to use and it is educational to those who administer it and those who fill it out.
- Even if the results may never be summarized, or filled out, the process of creating the test itself will provide us with useful information

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.
- Use cases and users can provide us with the attributes that 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.
- Once these attributes have been identified, they can play a crucial role in the evaluation of the final product.

Custom Form for User Satisfaction Test



Gause and Weinberg raise the following important point in conducting 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 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 user satisfaction test for a customer tracking system.

Measuring User Satisfaction

Project Name: **Customer Tracking System**

User	1	2	3	4	5	6	7	8	9	##	1	2	3	4	5	6	7	8	9	##	1	2	3	4	5	6	7	8
Ease of Use	1	7									4	2									4	2						
Functionality	4	4									6	8									8	7						
Intuitiveness of I	4	6									5	8									8	8						
Cost	1	1									5	5									5	5						
Reliability	3	4									6	8									8	7						
Comments																												

A shift in the user satisfaction rating indicates that something is happening.

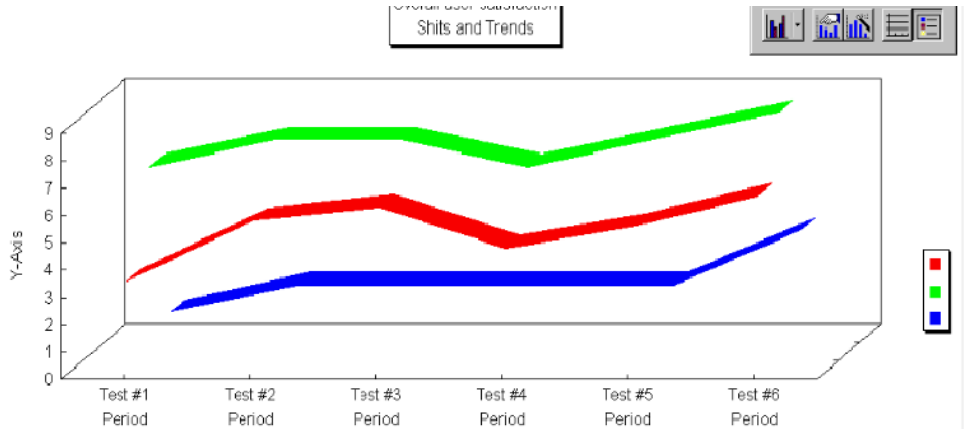
Measuring User Satisfaction

Project Name: Customer Tracking System

St. ,

Test #1	Test #2	Test #3	Test #4	Test #5	Test #6
Period 1	Period 2	Period 3	Period 4	Period 5	Period 6

Plotting the high and low responses indicates where to go for maximum information



User Satisfaction Cycle

1. Create a user satisfaction test with the users.
2. Conduct the test regularly and frequently.

3. Read comments very carefully, especially if they express strong feeling.

4. Use the information.

ii) Explain about the test cases and User satisfaction test for Bank ATM System.(Dec 2016)

Test cases for ATM Machine

1. Machine is accepting ATM card

2. Machine is rejecting expired card

3. Successful entry of PIN number

4. Unsuccessful operation due to enter wrong PIN number 3 times

5. Successful selection of language

6. Successful selection of account type

7. Unsuccessful operation due to invalid account type

8. Successful selection of amount to be withdraw

9. Successful withdrawal.

10. Expected message due to amount is greater than day limit

11. Unsuccessful withdraw operation due to lack of money in ATM

12. Expected message due to amount to withdraw is greater than possible balance.

13. Unsuccessful withdraw operation due to click cancel after insert card

User Satisfaction test for Bank ATM

Steps:

1. Develop test objectives.
2. Develop test cases.
3. Analyze the tests.

1. Develop test objectives –

1. Test objectives are based on the requirements, use cases, or convert or desired system usage.
2. "Ease of use" is the most important requirement.

Objectives to test usability of Via. Net bank ATM kiosk and it's Via:

95% of users should be able to find out how to withdraw money from the ATM machine without error (or) any format training.

90% of consumers should be able to operate the ATM within 90 seconds.

2. Develop Test Cases:

Test cases for usability testing are slightly different from test cases for quality assurance. We are not testing the input and expected output. But how users interact in value system. The usability test scenarios are based on the following use cases.

1. Deposit chewing
2. Withdraw chewing
3. Deposit chewing
4. Withdraw savings
5. Saving transaction himself
6. Checking transaction himself

Start by explaining the testing process and equipment to the participants to ease the pressure. As the participants work, second the time they take to perform a task as well as any problems they encounter. Once the test subjects complete their tasks, conduct a user satisfaction test to measure their level of satisfaction with the kiosk. The users use cases and test object should provide the attributes to be included in the test. The following attributes can play a certain role because ease of use is the main issue of user interface.

1. Easy to operate
2. Burrows are easily located.
3. It is efficient
4. It is visually pleasing
5. Provides easy recovery from error.
6. Analyze the Tests:

The final step is to analyze the tasks and document the test results. We also need to analyze the results of user satisfaction tasks. The user satisfaction test can be used as tool for finding out attributes are important or unimportant.

An Example:

In withdrawal checking, in adding to entering the amount for withdrawal, we are able to select from the list of predefined values say $20, $40, etc.

6. Write short notes on the following:
   i) Impact of Object Orientation on Testing
   ii) Impact of Inheritance in Testing

The impact of an object orientation on testing is summarized by the following .

- some types of error could become less plausible (not worth testing for).
- some types of error could become more plausible (worth testing for now).
- Some new types of errors might appear.

The testing approach is essentially the same in both the environments

(i.e)

- Object oriented environment and
- Non object oriented environment

- The problem of testing messages in an object orientation is the same as testing code that takes a function as a parameter and then invokes it.

MARICK argues that the process of testing variable uses in OOD essentially does not change, but you have to look in more places to decide what needs testing.

- Has the plausibility of faults changed?
- Are some types of fault now more plausible or less plausible?
- Since object oriented methods generally are smaller, these are easier to test.
- At the same time there are more opportunities for integration faults. They become more likely, more plausible.

❖ Impact of inheritance in testing
❖ Reusability of test

ii) Impact of Inheritance in Testing

Suppose you have the situation: base class contains method inherited( ) and redefined( ) and the derived class redefines the redefined( ) method.

- The derived::redefined has to be tested afresh since it is a new code.
- Does derived::inherited( ) have to be retested? If it uses redefined( ) and the redefined( ) has changed, the derived::inherited( ) may mishandle the new behavior. So, it needs new tests even though the derived::inherited( ) itself has not changed.
- If the base::inherited( ) has been changed, the derived::inherited( ) may not have to be completely tested. Whether it does depends on the base methods; otherwise it must be tested again.

7. i)Describe in detail about the different types of testing strategies.
ii) Write and explain the guidelines for developing quality assurance test cases in inventory control systems.(April/May2015)

White-box testing

White box testing is also called structural testing or glass box testing.

- The internal program logic is checked
- logical paths thro the s/w are tested
- status of program is examined
- all independent paths within a module is tested
- It tests the code
- It is applied in the early stage of testing process. It uses the control structure of the procedural design to test cases.

Sample Application

In-circuit testing is a good example of a white-box system testing where the tester is looking at the interconne between different components of the application and verifying the proper functioning of each internal conne We can also consider the example of an auto-mechanic who takes care of the inner workings of a vehicle to that all the components are working correctly to ensure the proper functioning

of the vehicle.

Basic Path Testing

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. Cyclomatic complexity has a foundation in graph theory and is computed in one of three ways:

4. The number of regions corresponds to the cyclomatic complexity.
5. Cyclomatic complexity, V(G), for a flow graph, G, is defined as
   V(G)=E-N+2 where E is the number of flow graph edges, and N is the number of flow graph nodes.
6. Cyclomatic complexity, V(G), for a flow graph, G is also defined as
   V(G)=P=1

   where P is the number of predicate nodes contained in the flow graph G.

Black box testing/behavioral testing (May 2016)

- It exercises all the functional requirements of the software.
- It does not know the internal working of the product.
- It specifies the functions of the software.
- It demonstrates that software functions are operational, ie whether the i/p is accepted and output is co produced, and that the integrity of external information (database) is maintained.
- It does not check the logical errors present in the application.

Sample Application

Search engine is a very good example of a black box system. We enter the text that we want to search, by pressing "search" we get the results. Here we are not aware of the actual process that has been implemented to get the results. We simply provide the input and get the results

System testing techniques

Recovery Testing

Security Testing

Stress Testing

Performance Testing

Regression Testing

Unit Testing

- ❖ focuses verification effort on the smallest unit of s/w design.
- ❖ control paths are tested to uncover errors within the boundary of the module.
- ❖ focuses on the internal processing logic and data structures within the boundaries of the component.
- ❖ can be conducted in parallel for multiple components.

Integration Testing

Integration testing is a systematic technique for constructing the s/w architecture while at the same time " conducting tests to uncover errors associated with interfacing". The objective is to take unit tested components and build a program structure that has been dictated by design.'

There are two types of integration

3. non incremented integration
4. incremental integration

ii) Write and explain the guidelines for developing quality assurance test cases in inventory control systems.(May 2015)

```
include<iostream.h>
#include<stdio.h>
#include<conio.h>
class invent
{
        public:
                int icode,iqty,iprice,q;
                char iname[20];
                void getdata(void);
                void display(void);

};
void invent::getdata(void)
{
        cout<<"\nEnter the Product Details:\n";
        cout<<"Enter the Item Name:\t";
        cin>>iname;
        cout<<"\nEnter the Item Code:\t";
        cin>>icode;
        cout<<"\nEnter the Item Quantity:\t";
        cin>>iqty;
        cout<<"\nEnter the Item Price:\t";
```

```
        cin>>iprice;
}
void invent::display(void)
{
        cout<<"\nItem Name:\t"<<iname;
        cout<<"\nItem Code:\t"<<icode;
        cout<<"\nStock Avilable:\t"<<iqty;
        cout<<"\nItem Price:\t"<<iprice;
        cout<<"\n.....";
}
void main()
{
        clrscr();
        invent inv[20];
        int opt,q,j,out=0;
        do
        {
                cout<<"\n1.Store Item Details\n2.Display All\n3.Exit\n";
                cout<<"Enter the option:\t";
                cin>>opt;
                switch(opt)
                {
                        case 1:
                        cout<<"\nEnter the Number of items to add\t";
                                cin>>q;
                                for(j=1;j<=q;j++)
                                {
                                        inv[j].getdata();
                                }
                                break;
                        case 2:
                                for(j=1;j<=q;j++)
```

```
                                        {
                                                inv[j].display();
                                        }
                                        break;
                                case 3:
                                        out=3;
                                        break;
                        }
                        if(out==3)
                                break;
                }while(opt<=3);
                getch();
        }
```

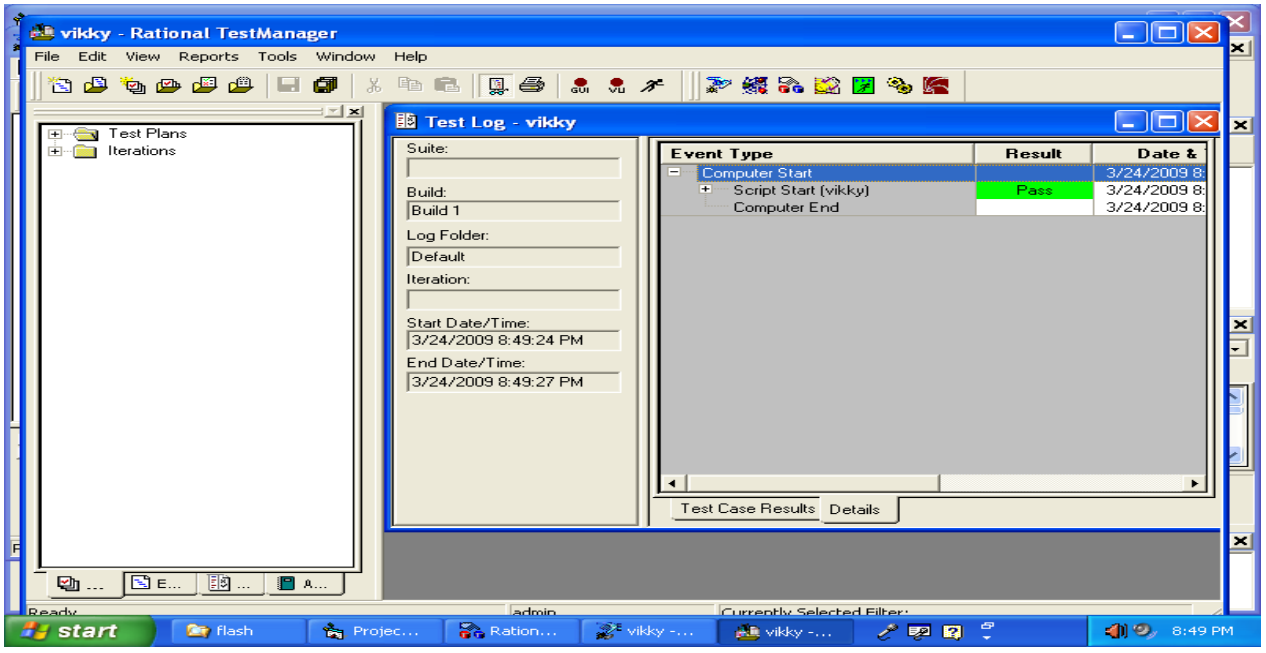
Output

```
Enter the option
1.Store Item Details
2.Display All
3.Exit
1
Enter the Number of items to add 1
Enter the Product Details
Enter the Item Name Pen
Enter the Item Code:1001
Enter the Item Quantity10
Enter the Item Price:100
Enter the option
1.Store Item Details
2.Display All
3.Exit
2
Item Name:: Pen
```

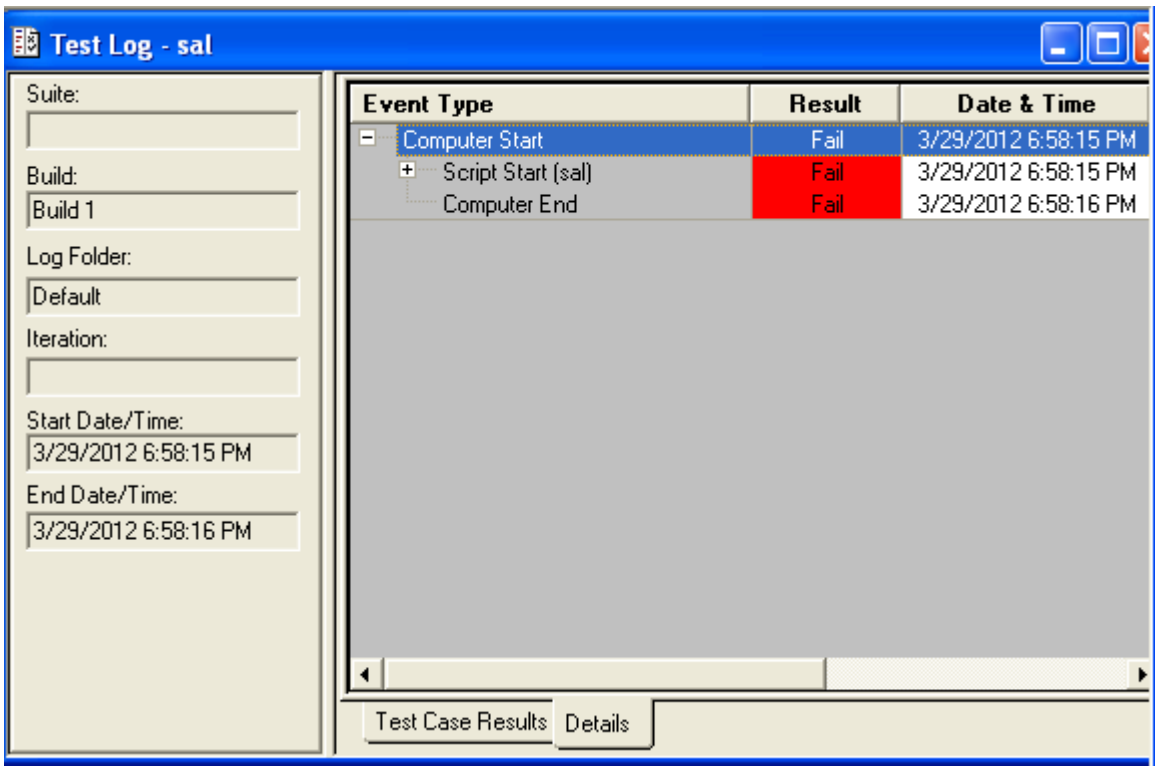
Item Code:1001

Stock Avilable: 10

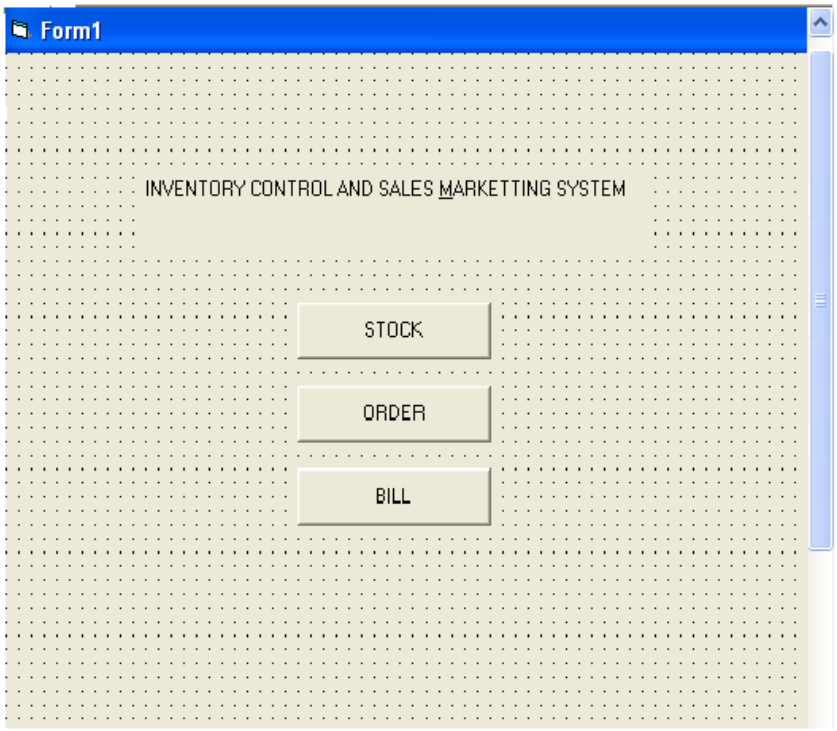
Item Price:100

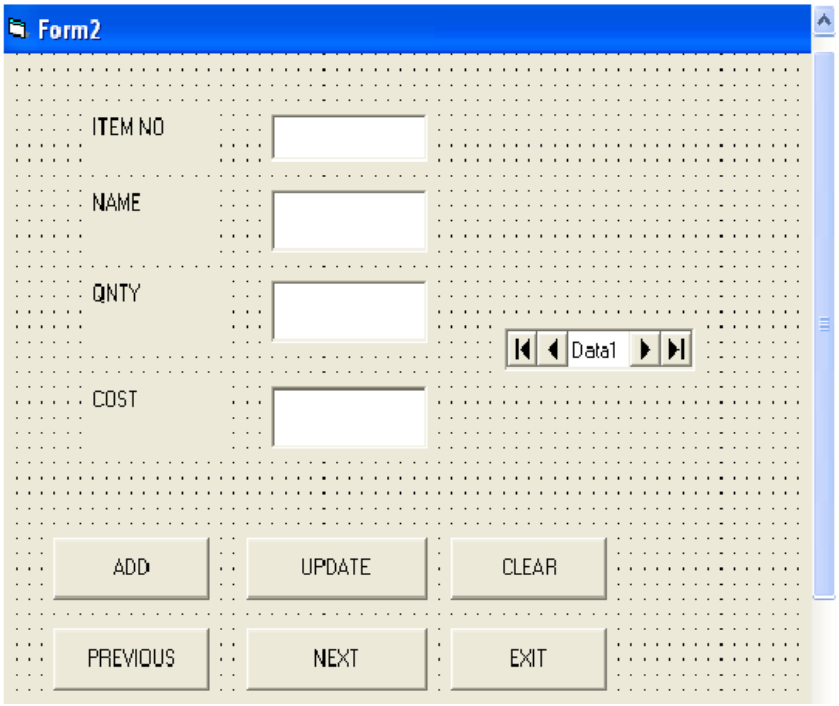


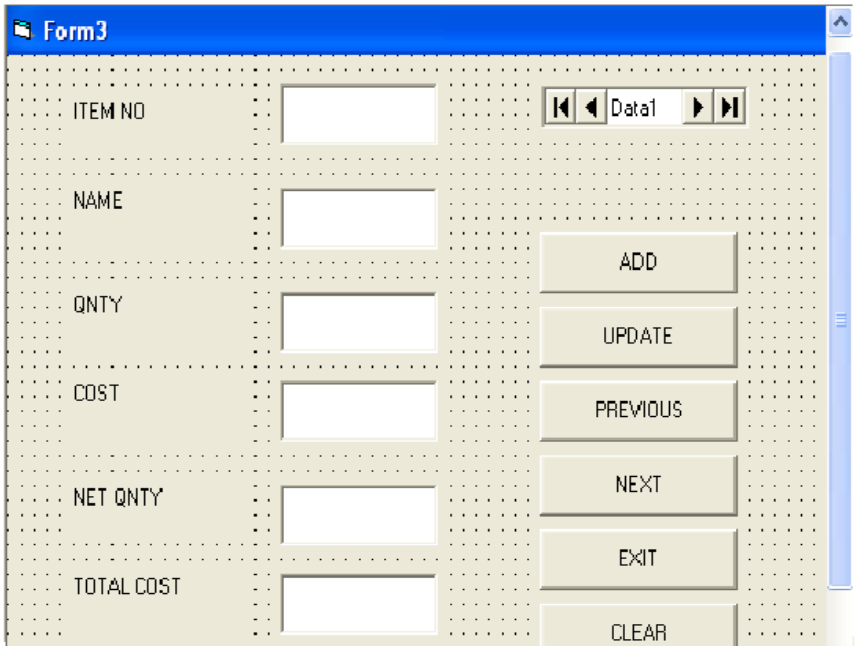
Output –invalid input

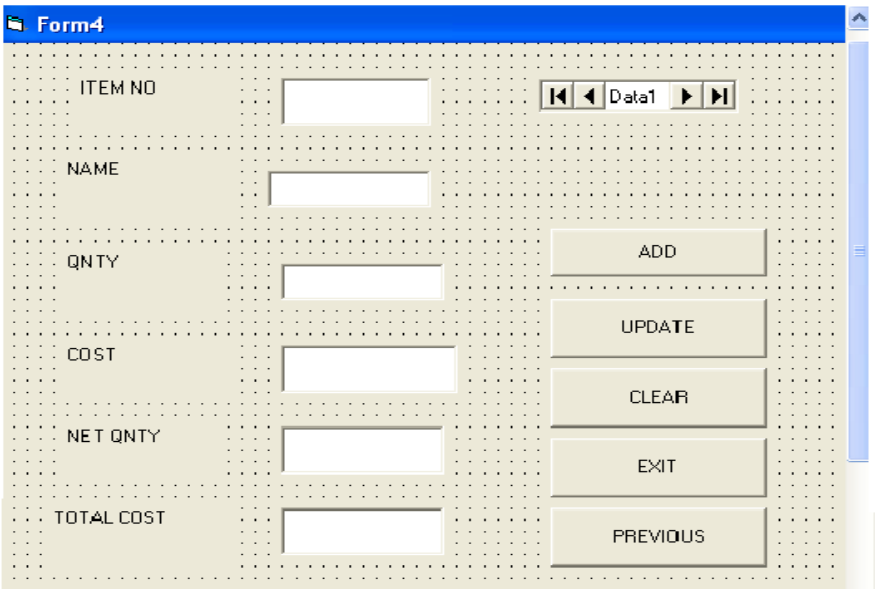


Design


Form1
INVENTORY CONTROL AND SALES MARKETTING SYSTEM
STOCK
ORDER
BILL


Form2
ITEM NO
NAME
QNTY
COST
Data1
ADD
UPDATE
CLEAR
PREVIOUS
NEXT
EXIT





FORM 1:

Private Sub Command1_Click()

Form2.Show

End Sub

Private Sub Command2_Click()

```
Form3.Show
End Sub

Private Sub Command3_Click()
Form4.Show
End Sub
```

FORM 2:

```
Dim db As New ADODB.Connection
Dim rs As New ADODB.Recordset
Private Sub Command1_Click()
rs.Close
rs.LockType = adLockOptimistic
rs.Open , , adOpenKeyset
rs.AddNew
rs.Fields(0) = Text1.Text
rs.Fields(1) = Text2.Text
rs.Fields(2) = Text3.Text
rs.Fields(3) = Text4.Text
rs.Update
MsgBox "the data has been added", vbOKCancel
End Sub

Private Sub Command3_Click()
rs.Close
rs.LockType = adLockOptimistic
rs.Open , , adOpenKeyset
rs.Update
rs.Fields(0) = Text1.Text
rs.Fields(1) = Text2.Text
rs.Fields(2) = Text3.Text
rs.Fields(3) = Text4.Text
```

```
rs.Update
MsgBox "the data has been updated", vbOKCancel
End Sub

Private Sub Command4_Click()
Form1.Show
End Sub

Private Sub Command5_Click()
Form3.Show
End Sub

Private Sub Command6_Click()
End
End Sub

Private Sub Command7_Click()
MsgBox "do you want to clear the content", vbOKCancel
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
MsgBox "text cleared", vbOKOnly
End Sub

Private Sub Form_Load()
db.ConnectionString = "Provider=microsoft.jet.oledb.4.0;Data Source=Y:\pen\vb\stock.mdb;Persist
Security Info=false"
db.Open
Set rs = db.Execute("select * from stck")
rs.Close
rs.LockType = adLockOptimistic
```

```
rs.Open , , adOpenKeyset
dis
End Sub
Sub dis()
If (rs.EOF) Then
MsgBox "there is no data"
Else
Text1.Text = rs(0)
Text2.Text = rs(1)
Text3.Text = rs(2)
Text4.Text = rs(3)
End If
End Sub
```

FORM 3:

```
Dim db As New ADODB.Connection
Dim rs As New ADODB.Recordset

Private Sub Command1_Click()
rs.Close
rs.LockType = adLockOptimistic
rs.Open , , adOpenKeyset
rs.AddNew
rs.Fields(0) = Text1.Text
rs.Fields(1) = Text2.Text
rs.Fields(2) = Text3.Text
rs.Fields(3) = Text4.Text
rs.Fields(4) = Text5.Text
rs.Fields(5) = Text6.Text
rs.Update
MsgBox "the data has been added", vbOKCancel
```

```
End Sub


Private Sub Command1_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As
Single)
Text6.Text = Val(Text4.Text) * Val(Text5.Text)
End Sub



Private Sub Command3_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As
Single)
Text6.Text = Val(Text4.Text) * Val(Text5.Text)
End Sub


Private Sub Command3_Click()
rs.Close
rs.LockType = adLockOptimistic
rs.Open , , adOpenKeyset
rs.Update
rs.Fields(0) = Text1.Text
rs.Fields(1) = Text2.Text
rs.Fields(2) = Text3.Text
rs.Fields(3) = Text4.Text
rs.Fields(4) = Text5.Text
rs.Fields(5) = Text6.Text
rs.Update
MsgBox "the data has been updated", vbOKCancel
End Sub


Private Sub Command4_Click()
Form1.Show
End Sub
```

```
Private Sub Command5_Click()
Form4.Show
End Sub

Private Sub Command6_Click()
End
End Sub

Private Sub Command7_Click()
MsgBox "do you want to clear the content", vbOKCancel
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Text5.Text = ""
Text6.Text = ""
MsgBox "text cleared", vbOKOnly
End Sub

Private Sub Form_Load()
db.ConnectionString = "Provider=microsoft.jet.oledb.4.0;Data Source=Y:\pen\vb\stock.mdb;Persist Security Info=false"
db.Open
Set rs = db.Execute("select * from ord")
rs.Close
rs.LockType = adLockOptimistic
rs.Open , , adOpenKeyset
dis
End Sub
Sub dis()
If (rs.EOF) Then
MsgBox "there is no data"
```

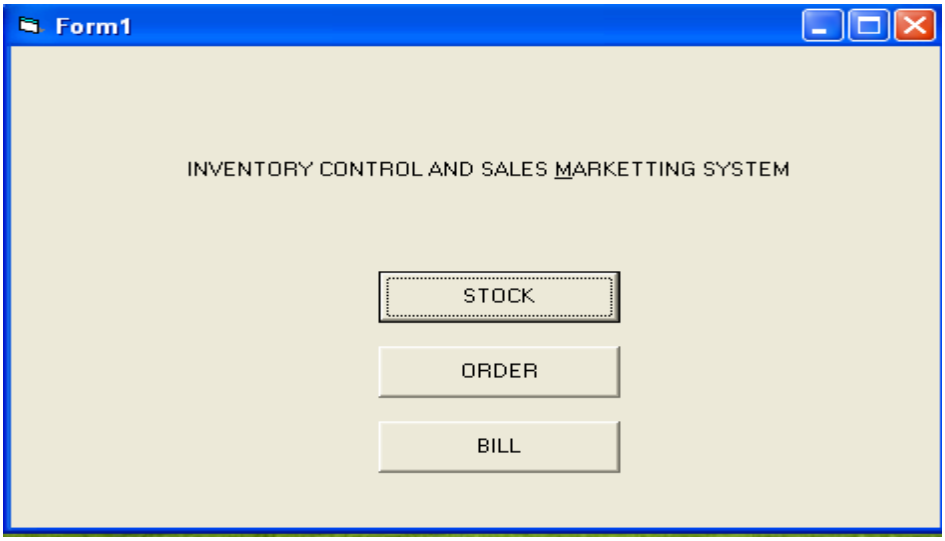
```
Else
Text1.Text = rs(0)
Text2.Text = rs(1)
Text3.Text = rs(2)
Text4.Text = rs(3)
Text5.Text = rs(4)
End If
End Sub
```

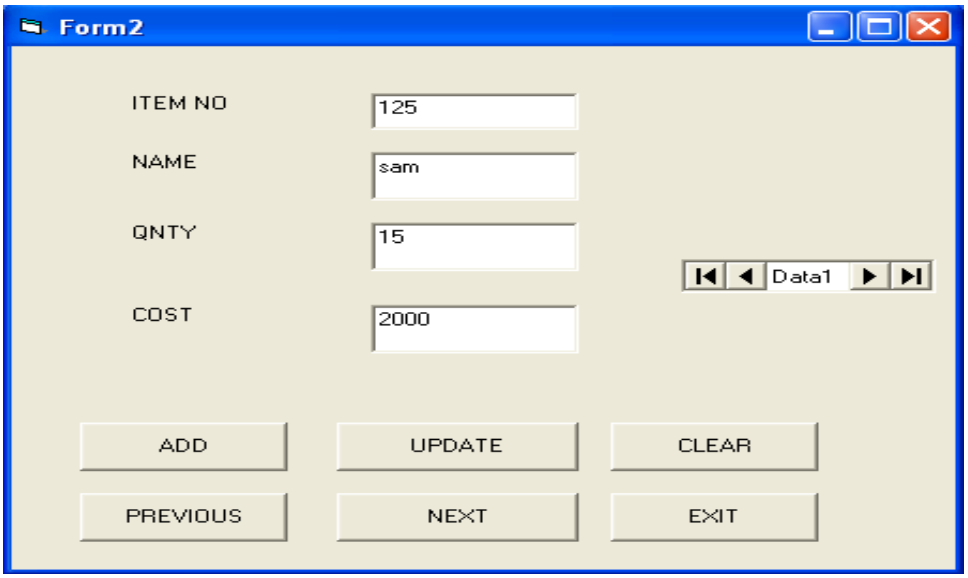
FORM 4:

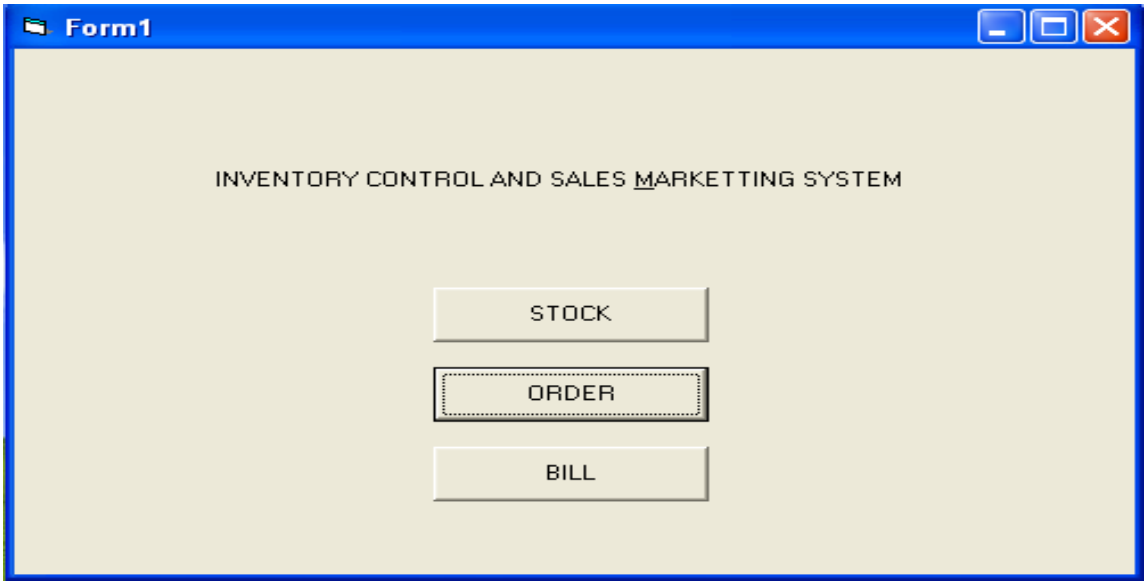
```
Dim db As New ADODB.Connection
Dim rs As New ADODB.Recordset

Private Sub Command1_Click()
rs.Close
rs.LockType = adLockOptimistic
rs.Open , , adOpenKeyset
rs.AddNew
rs.Fields(0) = Text1.Text
    rs.Fields(1) = Text2.Text
    rs.Fields(2) = Text3.Text
    rs.Fields(3) = Text4.Text
    rs.Fields(4) = Text5.Text
    rs.Fields(5) = Text6.Text
    rs.Update
    MsgBox "the data has been added", vbOKCancel
    End Sub

    Private Sub Command1_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As
    Single)
    Text6.Text = Val(Text4.Text) * Val(Text5.Text)
```

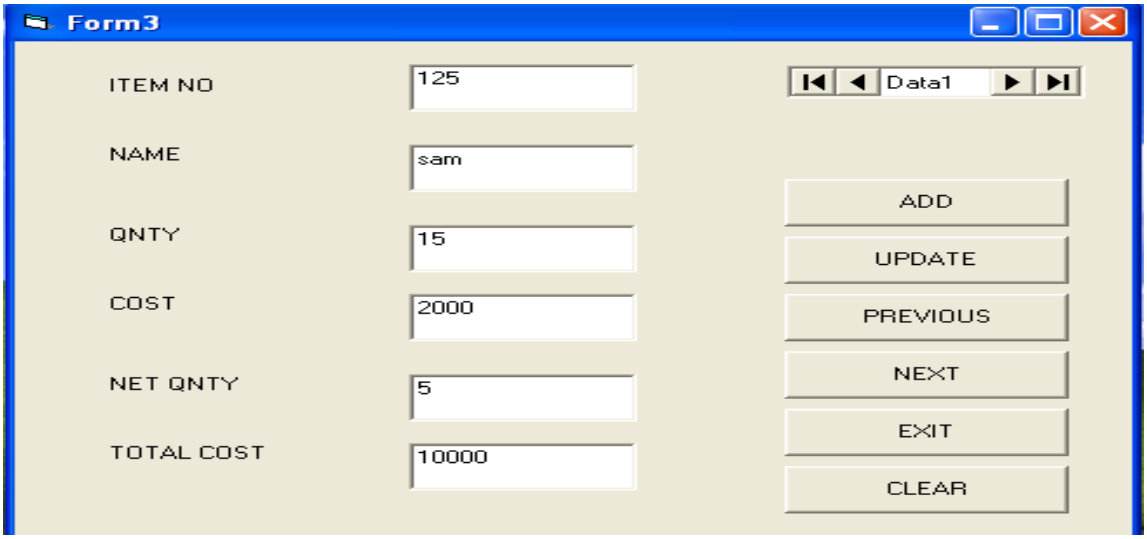
```
End Sub

Private Sub Command3_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As
Single)
Text6.Text = Val(Text4.Text) * Val(Text5.Text)
End Sub

Private Sub Command3_Click()
rs.Close
rs.LockType = adLockOptimistic
rs.Open , , adOpenKeyset
rs.Update
rs.Fields(0) = Text1.Text
rs.Fields(1) = Text2.Text
rs.Fields(2) = Text3.Text
rs.Fields(3) = Text4.Text
rs.Fields(4) = Text5.Text
rs.Fields(5) = Text6.Text
rs.Update
MsgBox "the data has been added", vbOKCancel
End Sub

Private Sub Command4_Click()
Form1.Show
End Sub

Private Sub Command5_Click()
End
End Sub

Private Sub Command6_Click()
MsgBox "do you want to clear the content", vbOKCancel
```

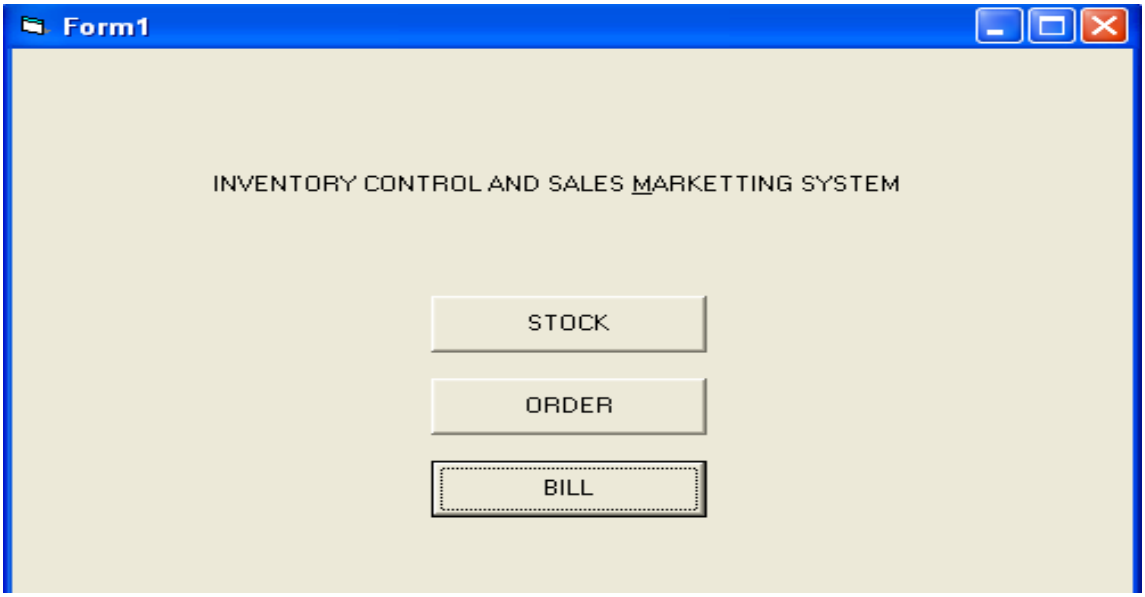
```
Text1.Text = ""
Text2.Text = ""
Text3.Text = ""
Text4.Text = ""
Text5.Text = ""
Text6.Text = ""
MsgBox "text cleared", vbOKOnly
End Sub

Private Sub Form_Load()
db.ConnectionString = "Provider=microsoft.jet.oledb.4.0;Data
Source=Y:\pen\vb\stock.mdb;Persist Security Info=false"
db.Open
Set rs = db.Execute("select * from bill")
rs.Close
rs.LockType = adLockOptimistic
rs.Open , , adOpenKeyset
dis
End Sub
Sub dis()
If (rs.EOF) Then
MsgBox "there is no data"
Else
Text1.Text = rs(0)
Text2.Text = rs(1)
Text3.Text = rs(2)
Text4.Text = rs(3)
Text5.Text = rs(4)
End If
End Sub
```

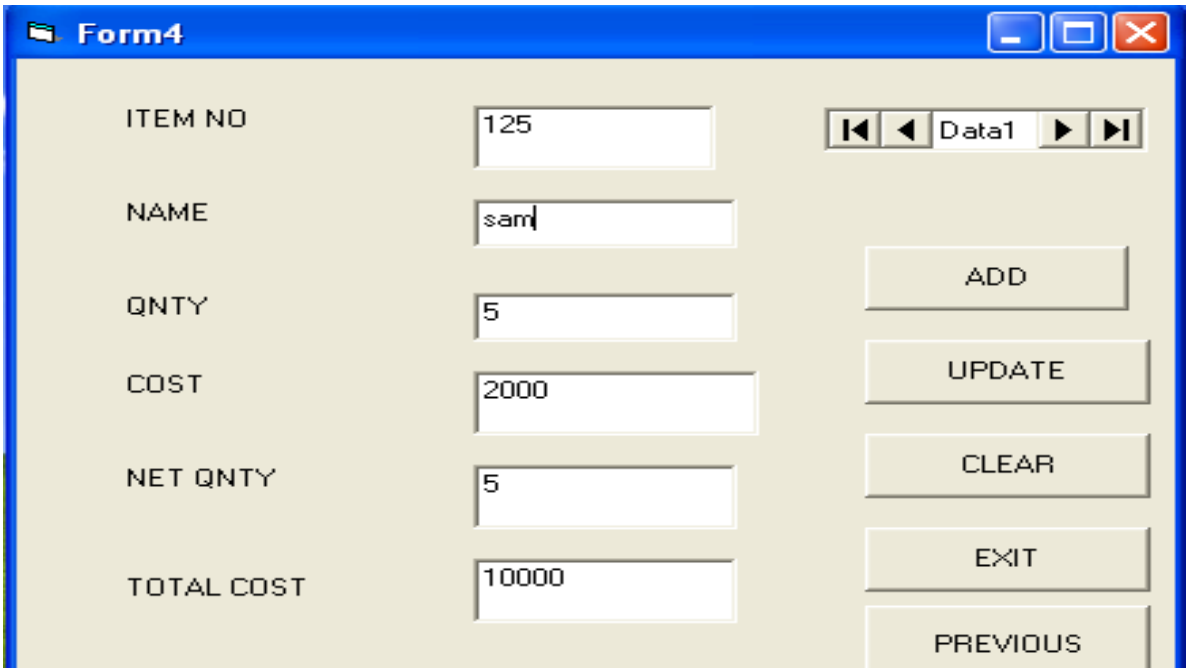

Form1
INVENTORY CONTROL AND SALES MARKETTING SYSTEM
STOCK
ORDER
BILL


Form2
ITEM NO
125
NAME
sam
QNTY
15
COST
2000
Data1
ADD
UPDATE
CLEAR
PREVIOUS
NEXT
EXIT

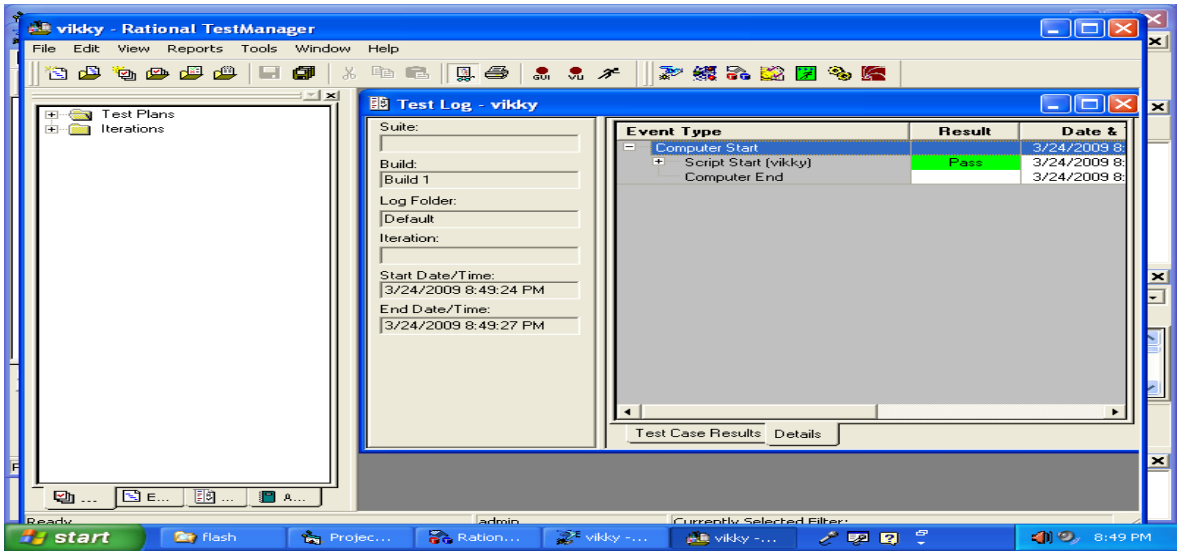

Form1
INVENTORY CONTROL AND SALES MARKETTING SYSTEM
STOCK
ORDER
BILL


Form3
ITEM NO
125
Data1
NAME
sam
QNTY
15
COST
2000
NET QNTY
5
TOTAL COST
10000
ADD
UPDATE
PREVIOUS
NEXT
EXIT
CLEAR

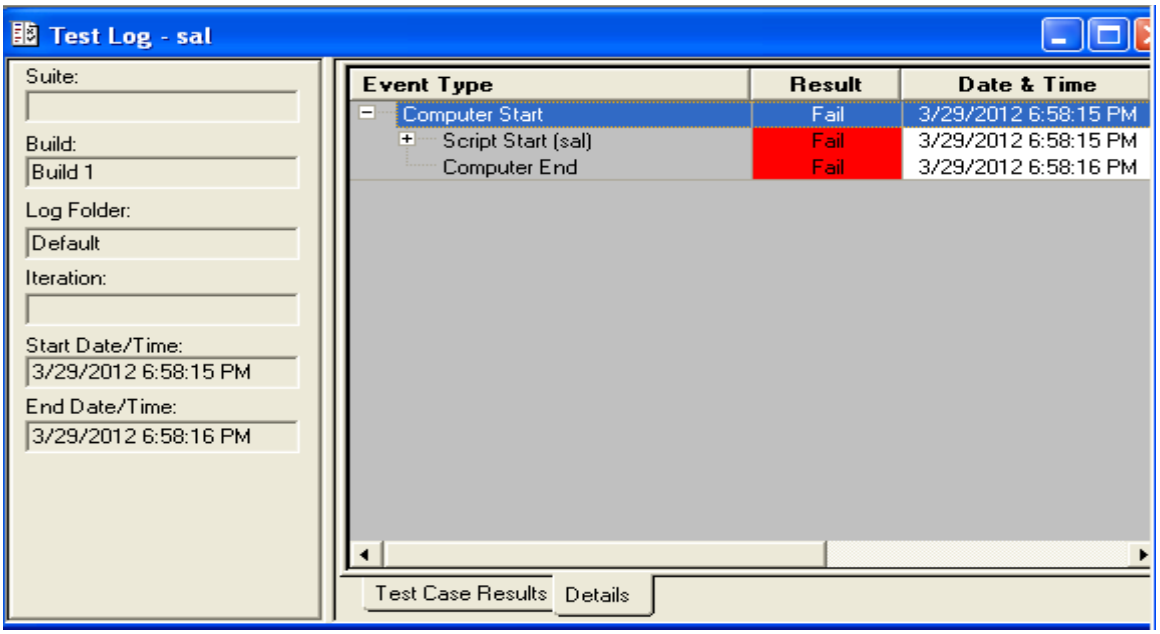




Output after testing with Rational Robot



OUTPUT-INVALID INPUT



8. List and explain the challenges in testing the health care systems. (April/May2015)

)Explain in detail about the quality assurance testing.

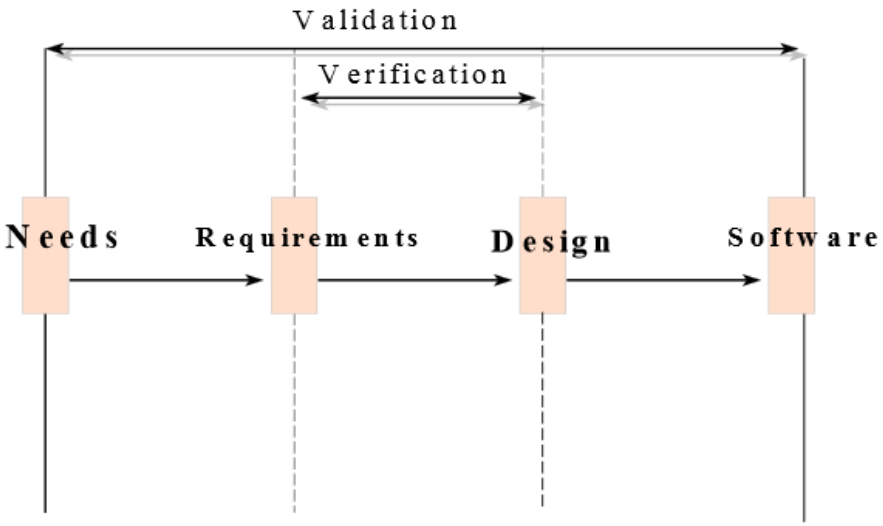
Quality Measures

rification

- "Am I building the product right?"

lidation

- "Am I building the right product?"



ality Assurance testing can be divided into 2 major categories.

Error Based Testing techniques search a given class's methods for particular clues of interest, then describe how these clue
ted.

: compute payroll method of employee class need to be tested.

Employee.computepayroll(hours)

test this method we must try different values for hours (say 40,0,-10,100) to see of the program can handle them. (als
ting the Boundary condition). This method should be able to handle any value; if not, error must be recorded and reported.

Scenario based Testing (usage based Testing) concentrates on what the user does, not what the product does.

is means capturing use cases and the tasks users perform, then performing them and their variants as tests. It can a
eraction bugs.

- they often are more complex and realistic than error-based tests.
- they tend to exercise multiple subsystems in a single test.

sting Objectives

- Testing is a process of executing a program with the intent of finding an error.