Table of Contents

CHAPTER 1: Object Oriented Fundamentals 1-52

1.1 Introduction 1
Different Types of Programming Paradigm 1
Object-oriented Programming Paradigm 2
Functional Programming vs Object Oriented Programming 3
What Is Analysis? 3
What Is Design? 3
1.2 Object Oriented Analysis and Design 4
What Is Object-Oriented Analysis and Design? 4
1.3 Defining Models 4
Key Steps of OOAD 5
1.4 Requirement Process/Requirement Analysis 7
1.5 Case Study 9
1.6 Use Cases 9
What are Actors, Scenarios, and Use Cases? 11
Three Common Use Case Formats 12
Example Use Case 12
Use Case Diagrams 14
Forward and Reverse Engineering 19
1.7 Object Oriented Development Cycle 19
Agile Development 21
1.8 Overview of the Unified Modeling Language: UML Fundamentals and Notations 22
A Conceptual Model of the UML 24
Building Blocks of the UML 24
Structural Things 25
Behavioral Things 26
Grouping Things 27
Annotational Things 27
Relationships in the UML 28
Diagrams in the UML 29
Common Mechanisms in the UML 30
Extensibility Mechanisms 32
Views of UML 32
Additional 34
Algorithmic Decomposition 34
Algorithmic vs. Object-Oriented Decomposition 35
Object Oriented Metrices 35

CHAPTER 2: Object Oriented Analysis 53-70

2.1 Building Domain/Conceptual Models 53
Domain Analysis 53

2.2 **Adding Associations and Attributes** **57**
2.3 **Representation of System Behavior** **62**

CHAPTER 3: Object Oriented Design **71-138**

3.1 **Object Oriented Design** **71**
3.1.1 Input for OOD (Object-Oriented Design) 71
3.1.2 Object-Oriented concepts: 72
3.1.3 Designing Concepts 73
3.1.4 Output obtained from OOD 73
3.1.5 How do developers design objects? 73
3.1.5.1 Designing Objects 74
3.1.5.2 Object Design Techniques 75
3.2 **Use case and Use case Diagrams** **76**
3.2.1 Terms and Concepts 76
3.2.2 Relationships between use Cases 77
3.2.3 Realization of Use case 79
3.2.3.1 Class Diagrams Owned by a Use-Case Realization . 79
3.2.3.2 Communication and Sequence Diagrams Owned by a Use-Case Realization 79
3.3 **UML Interaction Diagrams** **80**
3.3.1 Sequence Diagrams 80
3.3.1.1 Terms and notations 81
3.3.1.2 Example 86
3.4 **Communication/Collaboration Diagram** **89**
3.4.1 Comparison of Sequence and Collaboration Diagrams 89
3.4.2 Basic Collaboration Diagram Notation 90
3.4.3 Modeling the Realization of Operation 92
3.5 **Class Diagrams** **92**
3.5.1 Design Class Diagram (DCD) 93
3.5.2 Classifier 93
3.5.3 Types of Relationships in Class Diagrams 94
3.5.4 Comparison between Association, Aggregation and Composition 95
3.5.5 Attribute Text and Association Lines 96
3.5.6 Association Class 97
Singleton Classes 98
3.6 **Deployment Diagrams** **99**
3.6.1 Some examples of 99
3.7 **Activity Diagram** **100**
3.7.1 Terms and Concepts 101
3.7.2 Some Examples 103
3.7.3 Swimlanes 104
3.7.4 Activity Diagram Notations 105

3.8 **State Machine Diagram** 107
3.9 **Object Diagrams** 110
3.9.1 Basic Object Diagram Symbols and Notations 111
3.9.2 Class Diagram vs. Object Diagram 112
3.9.3 Example 112
3.10 **Visibility** 114
3.11 **What are Patterns?** 115
3.11.1 Characteristics of Good patterns 115
3.11.2 Types of Design Pattern 116
3.11.3 How to Describe Patterns? 116
3.12 **GRASP Patterns** 117

CHAPTER 4: Implementation 139-173

4.1 **Introduction** 139
4.2 **Programming and Development Process** 139
4.2.1 Software Engineering 140
4.2.2 Unified Software Development Process 141
4.3 **Mapping Design to Code** 142
4.4 **Creating Class Definitions from Design Class Diagrams** 142
4.4.1 Classes in Object Oriented Programming 142
4.4.1.1 Defining Classes 143
4.4.1.2 Encapsulation 144
4.4.1.3 Implementing Classes 144
4.4.1.4 Constructors 145
4.4.1.5 Static Members 145
4.4.1.6 Abstraction and Polymorphism 146
4.4.1.7 Interface 146
4.4.2 UML to Class definition 147
4.5 **Creating Methods from Collaboration Diagram** 151
4.6 **Container/Collection Classes in Code** 152
4.6.1 List 153
4.6.2 Set 153
4.6.3 Vector 154
4.7 **Exception and Error Handling** 155
4.7.1 Testing and Quality Assurance 156
4.7.1.1 Object Oriented Metrics 158
SAMPLE PROBLEM 159

Bibliography 174

Chapter 1

Object Oriented Fundamentals

1.1 Introduction

Different Types of Programming Paradigm

There are different types of programming paradigm. The most major programming paradigm are as follows:

1. Imperative Programming Paradigm
2. Logical Programming Paradigm
3. Functional Programming Paradigm
4. Object-oriented Programming Paradigm

Imperative Programming Paradigm

It is one of the oldest programming paradigms. It features close relation to machine architecture. It is based on Von Neumann architecture. It works by changing the program state through assignment statements. It performs step by step task by changing state. The focus is on how to achieve the goal. The paradigm consists of several statements and after execution of all the result is stored.

Logical Programming Paradigm

The Logical Paradigm takes a declarative approach to problem-solving. Various logical assertions about a situation are made, establishing all known facts. Then queries are made. The role of the computer becomes maintaining data and logical deduction.

Functional Programming Paradigm

The Functional Programming paradigm views all subprograms as functions in the mathematical sense-informally, they take in arguments and return a single solution. The solution returned is based entirely on the input, and the time at which a function is called has no relevance. The computational model is therefore one of function application and reduction. It is a declarative type of programming style.

Object-oriented Programming Paradigm

Object-oriented programming (OOP) is a programming technique distinguished by the identification of classes of objects that are strongly linked to the methods (functions) with which they are related.

Some Terminologies related to Object-oriented Programming Paradigm:

1. Class
2. Object
3. Method
4. Encapsulation
5. Abstraction
6. Inheritance

Class: Class is a template or the blueprint from which the objects are created. Class does not reserve the memory. Class is the description of the collection of objects that has the common operations and common attributes. Classes are the most important building block of any object-oriented system. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces.

(Attributes are the data values that define the object, and the operations describe behavior of objects)

Object: Objects are the instance of the class (i.e., Objects are created from class). Each object that is created is unique. Each object has its own identity, behavior, and state.

(Identity separates each object and makes each object unique from other objects, behavior is the method that object performs and state is the value of an attribute (characteristic property of an object).

Method: Methods are the those through which objects can manipulate the data.

Encapsulation: Encapsulation refers to the concept of combining data and processes that operate on that data into a single entity. This idea is also frequently used to hide an object's internal representation or state from the outside world. This is also known as called data hiding.

Abstraction: One of the fundamental ideas of object-oriented programming (OOP) languages is abstraction. Its primary purpose is to deal with complexity by hiding unwanted details from the user. This allows the user to build more complicated logic on top of the offered abstraction without having to understand or even consider all of the underlying complexity.

Inheritance: Inheritance is a process through which one object acquires all of the attributes and behaviors of its parent.

Functional Programming vs Object Oriented Programming

Functional Programming	Object Oriented Programming
Programs are divided into functions	Programs are divided into objects.
Importance is given to the functions not to the data.	Importance is given to the data rather than the functions.
Encapsulation is not possible.	Encapsulation is possible.
It generally follows "Top-Down" Approach	It generally follows "Bottom- Up" Approach
It provides less flexibility and abstraction as compared to object-oriented programming.	It is less difficult to modify object-oriented programs and reuse code as compared to structured programs.

What Is Analysis?

Analysis emphasizes an investigation of the problem and requirements, rather than a solution.

For example, if a new computerized library information system is desired, how will it be used? "Analysis" is a broad term, best qualified, as in requirements analysis (an investigation of the requirements) or object analysis (an investigation of the domain objects).

The task involved during analysis process are as follows:

1. Identify the objects
2. Organize the objects by creating object model diagram
3. Define object attributes
4. Define the behavior of an object (object actions)
5. Describe interaction between objects

What Is Design?

Design emphasizes a conceptual solution that fulfills the requirements, rather than its implementation.

For example, a description of a database schema and software objects. Ultimately, designs can be implemented.

The task involved during design process are as follows:

1. Implementation of logic (algorithms)
2. Implementation of methods
3. Implementation of association.

1.2 Object Oriented Analysis and Design

What Is Object-Oriented Analysis and Design?

During object-oriented analysis, there is an emphasis on finding and describing the objects—or concepts—in the problem domain.

For example, in the case of the library information system, some of the concepts include Book, Library, and Patron.

During object-oriented design, there is an emphasis on defining software objects and how they collaborate to fulfill the requirements.

For example, in the library system, a Book software object may have a title attribute and a getChap-ter method (see Figure 1). Finally, during implementation or object-oriented programming, design objects are implemented, such as a Book class in Java.

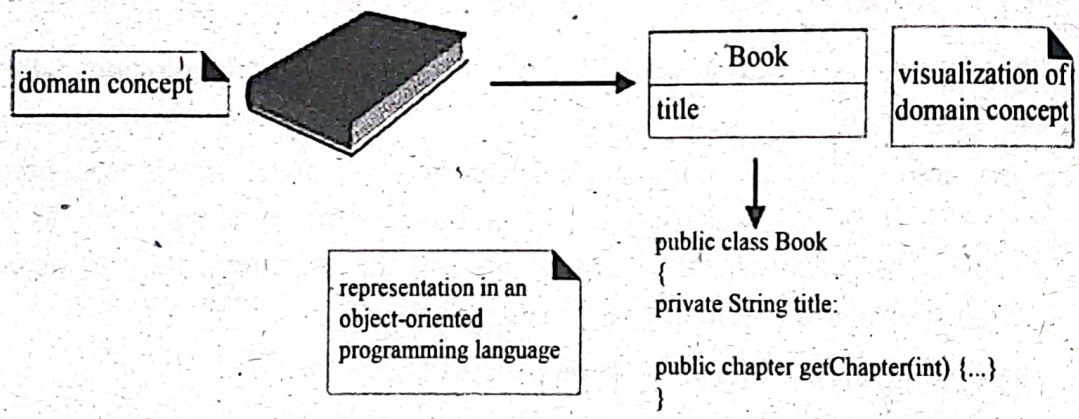


Figure 1: Object orientation emphasizes representation of objects

1.3 Defining Models

The goal of object-oriented modeling and design is to teach how to use object-oriented principles across the software development life cycle. Object-oriented modeling and design are a style of thinking about issues that involves the use of models that are arranged around real-world notions.

There are 5 types of models in the object-oriented modeling and design.

1. Conceptual Model
2. Structural Model
3. Behavioral Model
4. Specification Model
5. Implementation Model

Conceptual Model: Conceptual model represents the concepts of the problem domain. The main use of Conceptual model is that it makes the problem clearer and makes understand of the problem requirement much easier.

Structural Model: Conceptual model represents the structure of the problem domain.

Behavioral Model: Behavior model represents the behavior of the system. It describes the interaction between the system.

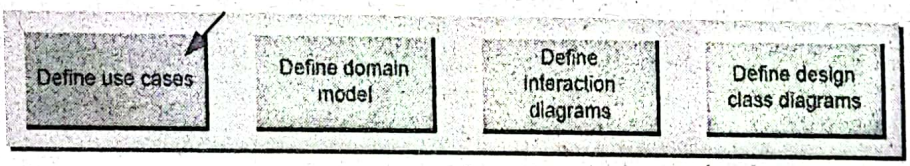
Specification Model: Specification Model describes the software abstractions with the specifications and interface.

Implementation Model: Implementation Model describe the software with implementations in a particular technology.

Key Steps of OOAD

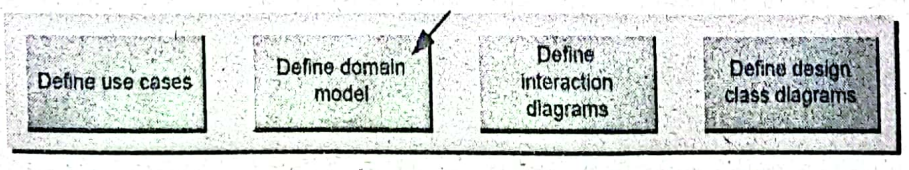
Analyzing the Key steps of OOAD with the example of the "dice game" in which a player rolls two dice. If the total is seven, they win; otherwise, they lose.

1. **Define Use Case:** Requirement's analysis may include a description of related domain processes these can be written as **use cases.** Use cases are not an object-oriented artifact—they are simply written stories. However, they are a popular tool in requirements analysis and are an important part of the Unified Process

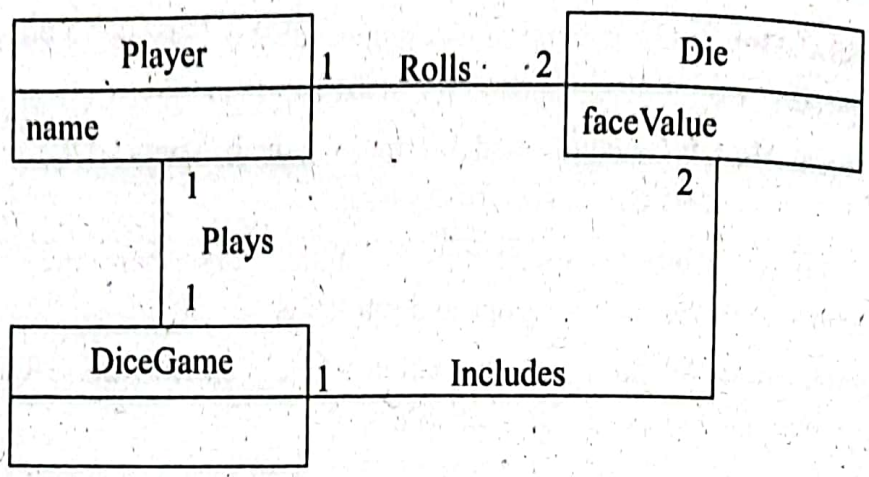


For Example: Here is a brief version of the *Play a Dice Game* use case: **Play a Dice Game:** A player picks up and rolls the dice. If the dice face value total seven, they win; otherwise, they lose.

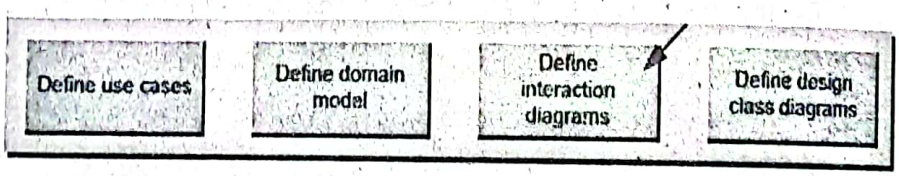
2. **Define a Domain Model:** Object-oriented analysis is concerned with creating a description of the domain from the perspective of classification by objects. A decomposition of the domain involves an identification of the concepts, attributes, and associations that are considered noteworthy. The result can be expressed in a **domain model.**



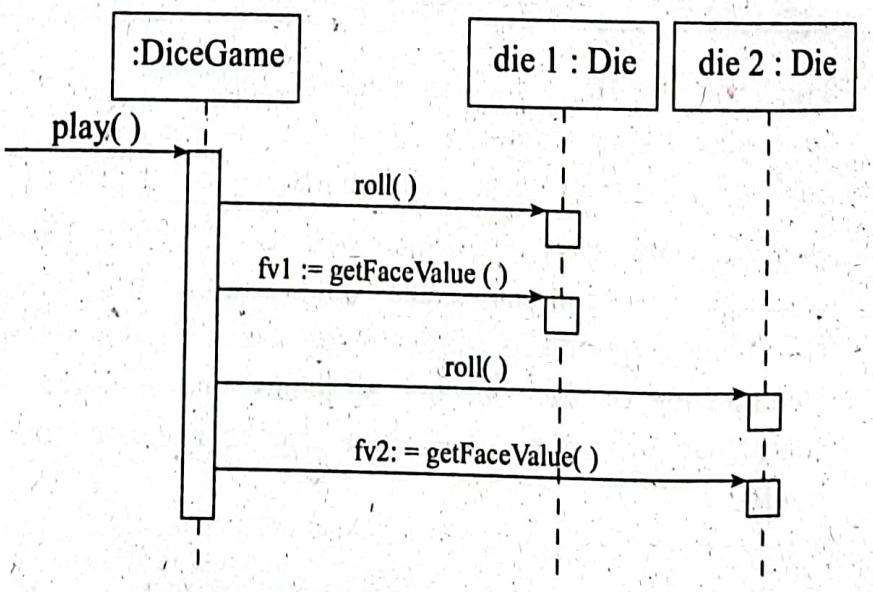
For Example: Here the partial domain model of the dice game is show below



3. **Define Interaction Diagrams:** Object-oriented design is concerned with defining software objects and their collaborations. A common notation to illustrate these collaborations is the **interaction diagram.** It shows the flow of messages between software objects, and thus the invocation of methods.

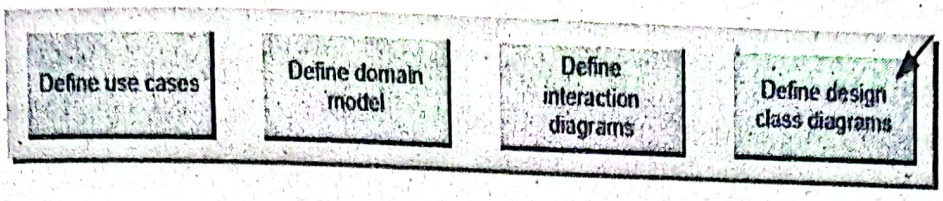


For example, assume that a software implementation of the dice game is desired. The interaction diagram below illustrates the essential step of playing, by sending messages to instances of the *Dice Game* and *Die* classes.

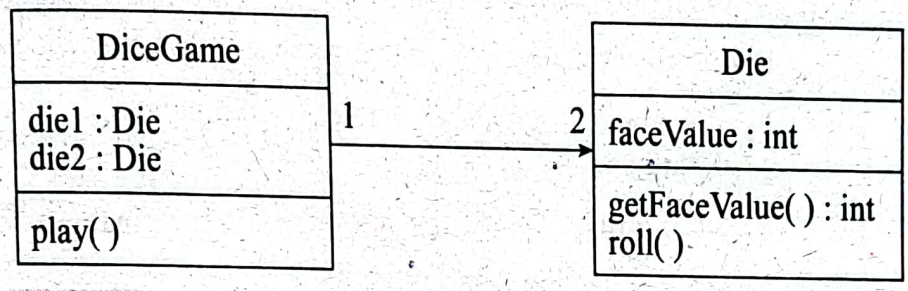


4. **Define Design Class Diagrams**

In addition to a *dynamic* view of collaborating objects shown in interaction diagrams, it is useful to create a *static* view of the class definitions with a **design class diagram.** This illustrates the attributes and methods of the classes.



For example, in the dice game, an inspection of the interaction diagram leads to the partial design class diagram shown in Figure 1.5. Since a play message is sent to a Dice Game object, the Dice Game class requires a play method, while class Die requires a roll and getFaceValue method. In contrast to the domain model, this diagram does not illustrate real-world concepts; rather, it shows software classes.



Note: This is just the summary. We will learn the concepts and methods to draw each diagram above in the further lessons.

1.4 Requirement Analysis

Process/Requirement

Requirements are capabilities and conditions to which the system and more broadly, the project must conform. A prime challenge of requirements analysis is to find, communicate, and remember (that usually means write down) what is really needed, in a form that clearly speaks to the client and development team members.

A systematic approach to finding, documenting, organizing, and tracking the changing requirements of a system is called Requirement Process.

It is the sequence of the activities that should be performed in the requirement phase to produce the high-quality SRS document.

Functional Requirement: A system or one of its components is defined by a functional requirement. It outlines the functions that software must carry out. The inputs, its behavior, and outputs make up a function. A system's likely function can be determined by a computation, data manipulation, business process, user interaction, or any other specialized feature. In software engineering, functional requirements assist you in capturing the intended behavior of the system. This behavior may be represented in terms of actions, processes, or tasks that a system must carry out.

Non Functional Requirement: A software system's quality attribute is defined by a non-functional need. They stand for a set of criteria used to assess a system's particular performance. For example, how quickly does the website load? To guarantee the overall software system's usability and efficiency, a non-functional requirement is necessary. Systems that don't meet non-functional requirements may not be able to meet user needs.

Requirement Types:

1. **Functional** features, capabilities, security.
2. **Non Functional:**
   - **Usability** human factors, help, documentation.
   - **Reliability** frequency of failure, recoverability, predictability.
   - **Performance** response times, throughput, accuracy, availability, resource usage.
   - **Supportability** adaptability, maintainability, internationalization, configurability.

Requirement Process Consists of the following activities:

1. Requirements elicitation
2. Requirements specification
3. Requirements verification and validation
4. Requirements management

Requirement Elicitation:

It relates to the many methods used to learn about the project topic and needs. Customers, business manuals, existing software of the same sort, standards, and other project stakeholders are among the different sources of domain knowledge. Interviews, brainstorming, task analysis, the Delphi approach, prototyping, and other techniques are used to elicit requirements.

The UP (UML Patterns) welcomes any requirements elicitation method that can add value and that increases user participation.

Requirements specification

This activity is used to create formal software requirement models. These models provide all needs, including functional and non-functional requirements, as well as limitations. More knowledge about the problem may be necessary during specification, which may trigger the elicitation process.

ER diagrams, data flow diagrams (DFDs), and other models are implemented at this phase.

Requirements verification and validation

Verification: It refers to the group of tasks that guarantee the software carries out a particular function appropriately.

Validation: It refers to a different set of duties that guarantee the software developed may be linked back to client specifications.

Requirements management

The process of evaluating, recording, tracking, prioritizing, and deciding on requirements, as well as regulating communication to relevant stakeholders, is known as requirement management. This stage deals with the changing nature of requirements. It should be assured that the SRS is as changeable as feasible in order to integrate changes in end-user needs at subsequent stages.

1.5 Case Study

Case study on The NextGen POS System

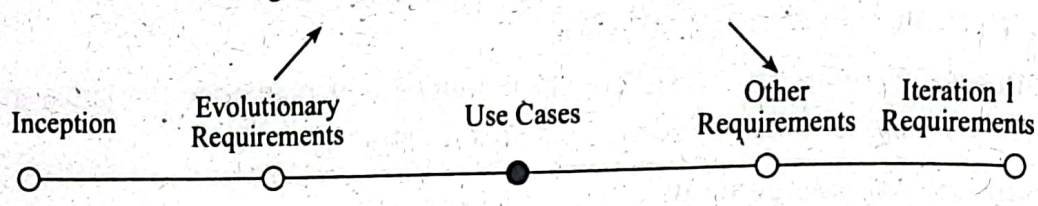
A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.

Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing. Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added. Therefore, we will need a mechanism to provide this flexibility and customization.

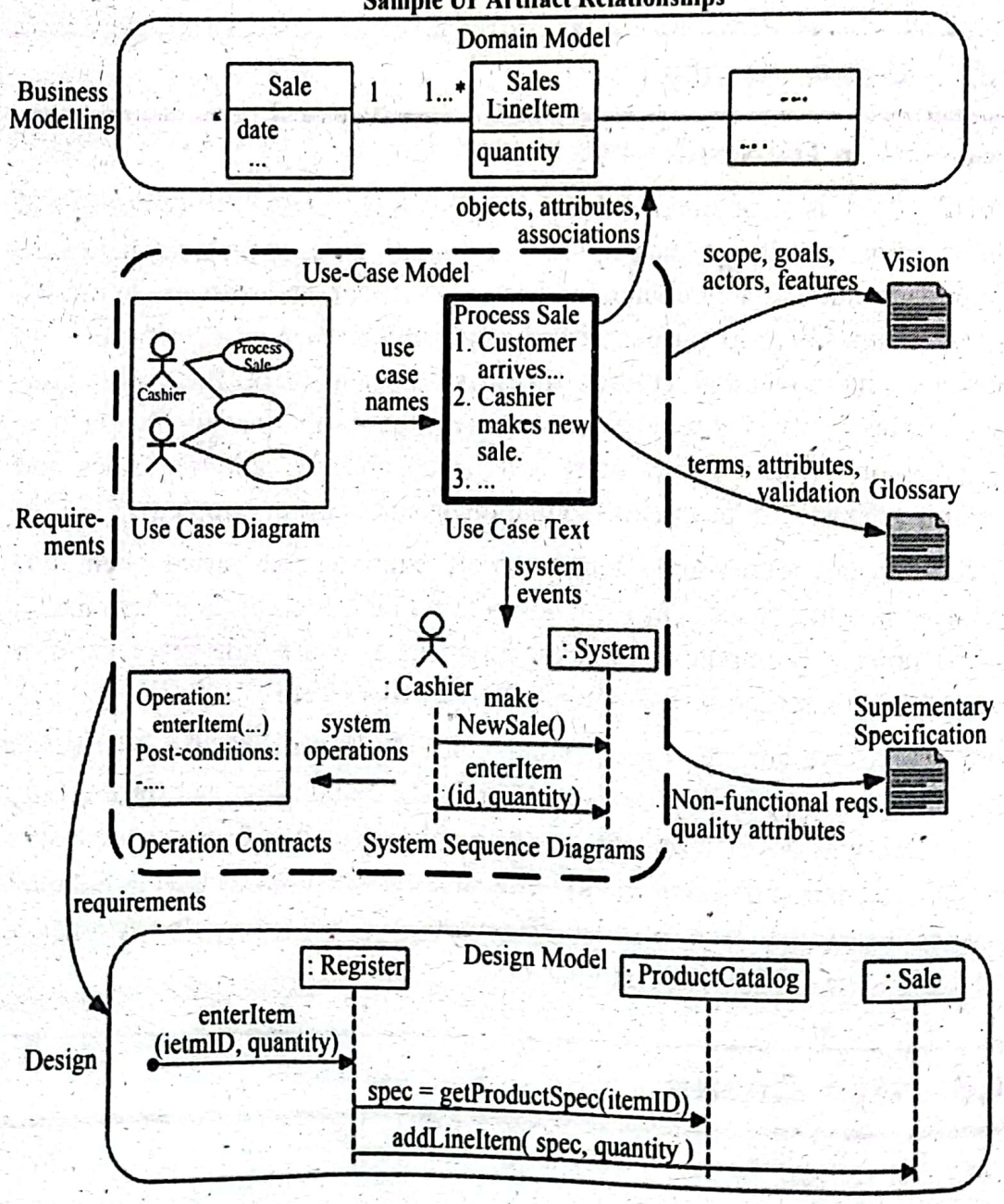
1.6 Use Cases

Use cases are text stories, widely used to discover and record requirements. They influence many aspects of a project including OOA/D and they are the input to many subsequent artifacts in the case studies.



The influence of UP artifacts, with an emphasis on text use cases, is shown in Figure below. High-level goals and use-case diagrams are input to the creation of the use case text. The use cases can in turn influence many other analysis, design, implementation, project management, and test artifacts.

Sample UP Artifact Relationships



Informally, use cases are text stories of some actor using a system to meet goals. Here is an example brief format use case:

Process Sale: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

Notice **that use cases are not diagrams, they are text**. Focusing on secondary-value UML use case diagrams rather than the important use case text is a common mistake for use case novices.

Use cases often need to be more detailed or structured than this example, but the essence is discovering and recording functional requirements by writing stories of using a system to fulfill user goals; that is, cases of use. It isn't supposed to be a difficult idea, although it's often difficult to discover what's needed and write it well.

Use cases are text documents, not diagrams, and use-case modeling is primarily an act of writing text, not drawing diagrams.

What are Actors, Scenarios, and Use Cases?

An **actor** is something with behavior, such as a person (identified by role), computer system, or organization, for example, a cashier.

A **scenario** is a specific sequence of actions and interactions between actors and the system; it is also called a **use case instance**. It is one particular story of using a system, or one path through the use case; for example, the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit payment denial.

A **use case** is a collection of related success and failure scenarios that describe an actor using a system to support a goal.

Why Use Cases?

Use cases emphasize the user goals and perspective; we ask the question "Who is using the system, what are their typical scenarios of use, and what are their goals?" This is a more user-centric emphasis compared to simply asking for a list of system features. Much has been written about use cases, and though worthwhile, creative people often obscure a simple idea with layers of sophistication or over-complication. It is usually possible to spot a novice use-case modeler (or a serious Type-A analyst!) by an over-concern

with secondary issues such as use case diagrams, use case relationships, use case packages, and so forth, rather than a focus on the hard work of simply writing the text stories. That said, a strength of use cases is the ability to scale both up and down in terms of sophistication and formality.

What are Three Kinds of Actors in Use Case?

An **actor** is anything with behavior, including the system under discussion (SuD) itself when it calls upon the services of other systems.

1. **Primary actor** has user goals fulfilled through using services of the SuD. For example, the cashier.

   Why identify? To find user goals, which drive the use cases.

2. **Supporting actor** provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.

   Why identify? To clarify external interfaces and protocols.

3. **Offstage actor** has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.

   Why identify? To ensure that *all* necessary interests are identified and satisfied. Offstage actor interests are sometimes subtle or easy to miss unless these actors are explicitly named.

Three Common Use Case Formats

brief Terse one-paragraph summary, usually of the main success scenario.

casual Informal paragraph format. Multiple paragraphs that cover various scenarios.

fully dressed All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.

Example Use Case

Next Gen Pos application Use case for Success Scenario

Scope: NextGen POS application

Level: user goal

Primary Actor: Cashier

Stakeholders and Interests:

- **Cashier:** Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
- **Salesperson:** Wants sales commissions updated.

- **Customer:** Wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns.
- **Company:** Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
- **Manager:** Wants to be able to quickly perform override operations, and easily debug Cashier problems.
- **Government Tax Agencies:** Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
- **Payment Authorization Service:** Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.
- **Preconditions:** Cashier is identified and authenticated.
- **Success Guarantee (or Postconditions):** Sale is saved. Tax is correctly calculated. Accounting and
- Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

Main Success Scenario:

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Use Case Diagrams

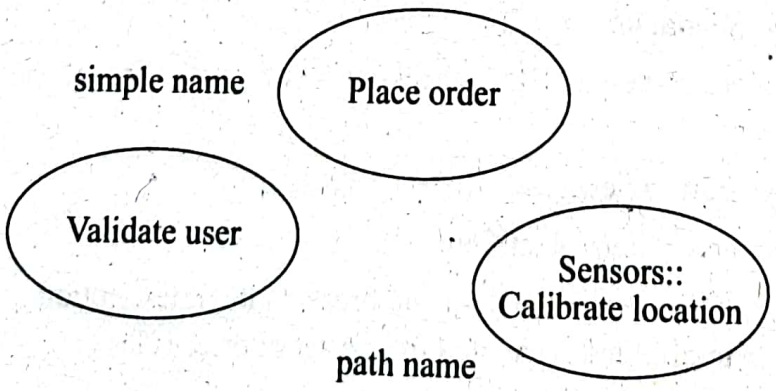
The UML provides use case diagram notation to illustrate the names of use cases can actors, and the relationships between them A use case diagram is an excellent picture of the system context; it makes a good **context diagram**, that is, showing the boundary of a system, what lies outside of it, and how it gets used. It serves as a communication tool that summarizes the behavior of a system and its actors.

A *use case* is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor. Graphically, a use case is rendered as an ellipse.

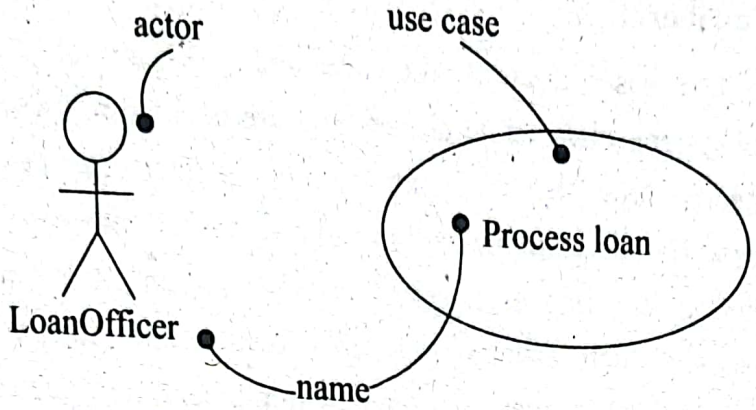
Use case diagrams are important for visualizing, specifying, and documenting the behavior of an element. They make systems, subsystems, and classes approachable and understandable by presenting an outside view of how those elements may be used in context. Use case diagrams are also important for testing executable systems through forward engineering and for comprehending executable systems through reverse engineering.

Terminologies

1. **Name:** A use case name must be unique within its enclosing package. Every use case must have a name that distinguishes it from other use cases. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the use case name prefixed by the name of the package in which that use case lives. A use case is typically drawn showing only its name.



2. **Actors:** An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. Typically, an actor represents a role that a human, a hardware device, or even another system plays with a system. For example, if you work for a bank, you might be a **LoanOfficer**. If you do your personal banking there, as well, you'll also play the role of **Customer**. An instance of an actor, therefore, represents an individual interacting with the system in a specific way. Although you'll use actors in your models, actors are not actually part of the system. They live outside the system.



3. **Flow of Events:** A use case describes *what* a system (or a subsystem, class, or interface) does but it does not specify *how* it does it. When you model, it's important that you keep clear the separation of concerns between this outside and inside view. You can specify the behavior of a use case by describing a flow of events in text clearly enough for an outsider to understand it easily. When you write this flow of events, you should include how and when the use case starts and ends, when the use case interacts with the actors and what objects are exchanged, and the basic flow and alternative flows of the behavior.

For example, in the context of an ATM system, you might describe the use case **ValidateUser** in the following way:

Main flow of events:

The use case starts when the system prompts the *Customer* for a PIN number. The *Customer* can now enter a PIN number via the keypad. The *Customer* commits the entry by pressing the Enter button. The system then checks this PIN number to see if it is valid. If the PIN number is valid, the system acknowledges the entry, thus ending the use case.

Exceptional flow of events:

The *Customer* can cancel a transaction at any time by pressing the Cancel button, thus restarting the use case. No changes are made to the *Customer*'s account.

Exceptional flow of events:

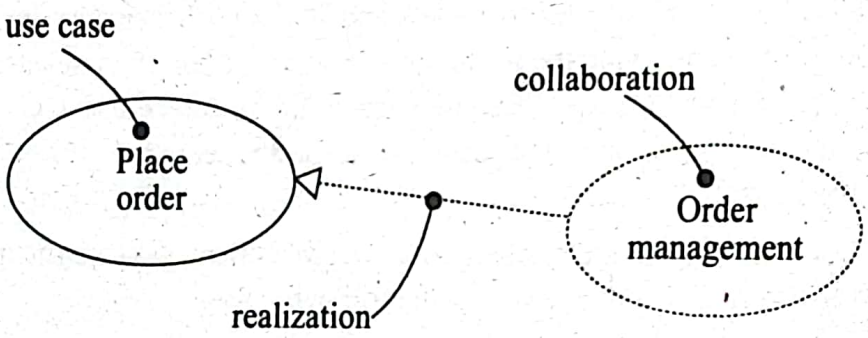
The *Customer* can clear a PIN number anytime before committing it and reenter a new PIN number.

Exceptional flow of events:

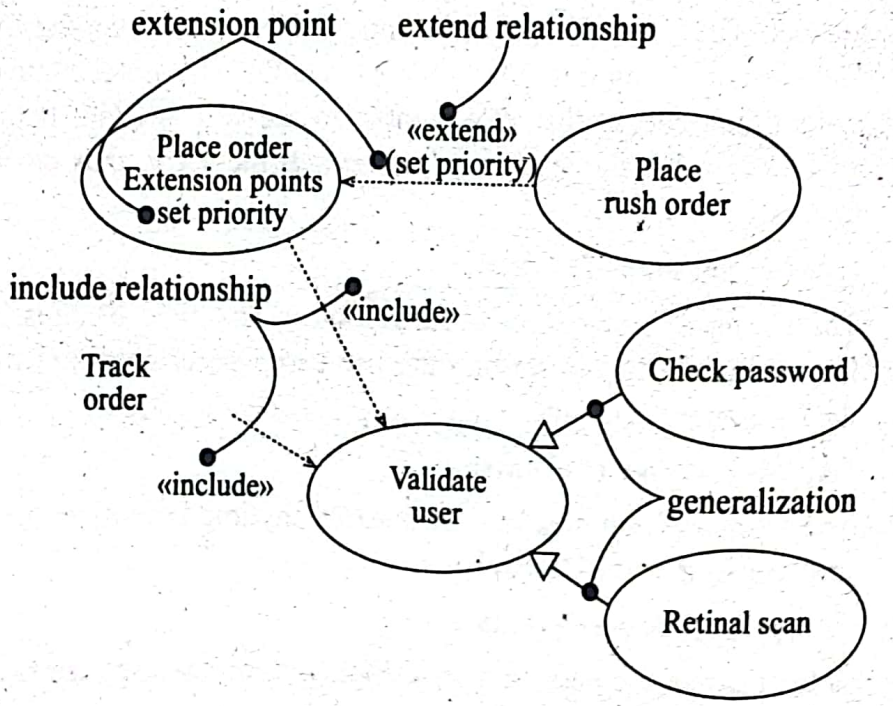
If the *Customer* enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the *Customer* from interacting with the ATM for 60 seconds.

4. Collaborations

A use case captures the intended behavior of the system (or subsystem, class, or interface) you are developing, without having to specify how that behavior is implemented. That's an important separation because the analysis of a system (which specifies behavior) should, as much as possible, not be influenced by implementation issues (which specify how that behavior is to be carried out). Ultimately, however, you have to implement your use cases; and you do so by creating a society of classes and other elements that work together to implement the behavior of this use case. This society of elements, including both its static and dynamic structure, is modeled in the UML as a collaboration.



5. Relationship



Generalization:

Generalization among use cases is just like generalization among classes. Here it means that the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears (both the parent and the child may have concrete instances).

Include:

An include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base. The included use case never stands alone, but is only instantiated as part of some larger base that includes it. You can think of include as the base use case pulling behavior from the supplier use case. You use an include relationship to avoid describing the same flow of events several times, by putting the common behavior in a use case of its own (the use case that is included by a base use case). The include relationship is essentially an example of delegation• you take a set of responsibilities of the system and capture it in one place (the included use case), then let all other parts of the system (other use cases) include the new aggregation of responsibilities whenever they need to use that functionality.

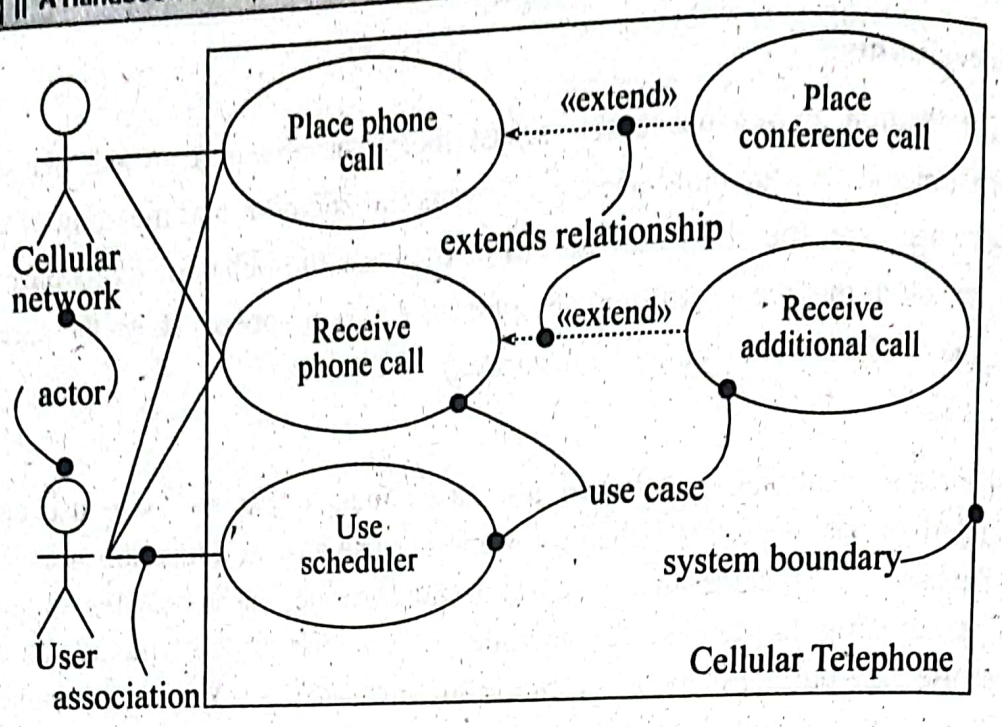
Extend:

An extend relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case. The base use case may stand alone, but under certain conditions, its behavior may be extended by the behavior of another use case. This base use case may be extended only at certain points called, not surprisingly, its extension points. You can think of extend as the extension use case pushing behavior to the base use case. You use an extend relationship to model the part of a use case the user may see as optional system behavior. In this way, you separate optional behavior from mandatory behavior. You may also use an extend relationship to model a separate sub flow that is executed only under given conditions. Finally, you may use an extend relationship to model several flows that may be inserted at a certain point, governed by explicit interaction with an actor.

Association:

An association is a structural relationship that describes a set of links; a link is a connection among objects.

Example: Use case diagram for cellular Telephone



Concepts of use Case

1. **Common Properties:** A use case diagram is just a special kind of diagram and shares the same common properties as do all other diagrams a name and graphical contents that are a projection into a model. What distinguishes a use case diagram from all other kinds of diagrams is its particular content.

2. **Contents:**

   Use case diagrams commonly contain

   - Use cases
   - Actors
   - Dependency, generalization, and association relationships

   Like all other diagrams, use case diagrams may contain notes and constraints. Use case diagrams may also contain packages, which are used to group elements of your model into larger chunks. Occasionally, you'll want to place instances of use cases in your diagrams, as well, especially when you want to visualize a specific executing system.

3. **Uses of Use Case**

   a. **To model the context of a system:** Modeling the context of a system involves drawing a line around the whole system and asserting which actors lie outside the system and interact with it. Here, you'll apply use case diagrams to specify the actors and the meaning of their roles

b. **To model the requirements of a system:** Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do it. Here, you'll apply use case diagrams to specify the desired behavior of the system. In this manner, a use case diagram lets you view the whole system as a black box; you can see what's outside the system and you can see how that system reacts to the things outside, but you can't see how that system works on the inside.

Forward and Reverse Engineering

Forward engineering is the process of transforming a model into code through a mapping to an implementation language. A use case diagram can be forward engineered to form tests for the element to which it applies. Each use case in a use case diagram specifies a flow of events (and variants of those flows), and these flows specify how the element is expected to behave that's something worthy of testing. A well-structured use case will even specify pre- and postconditions that can be used to define a test's initial state and its success criteria. For each use case in a use case diagram, you can create a test case that you can run every time you release a new version of that element, thereby confirming that it works as required before other elements rely on it.

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language. Automatically reverse engineering a use case diagram is pretty much beyond the state of the art, simply because there is a loss of information when moving from a specification of how an element behaves to how it is implemented. However, you can study an existing system and discern its intended behavior by hand, which you can then put in the form of a use case diagram. Indeed, this is pretty much what you have to do anytime you are handed an undocumented body of software. The UML's use case diagrams simply give you a standard and expressive language in which to state what you discover.

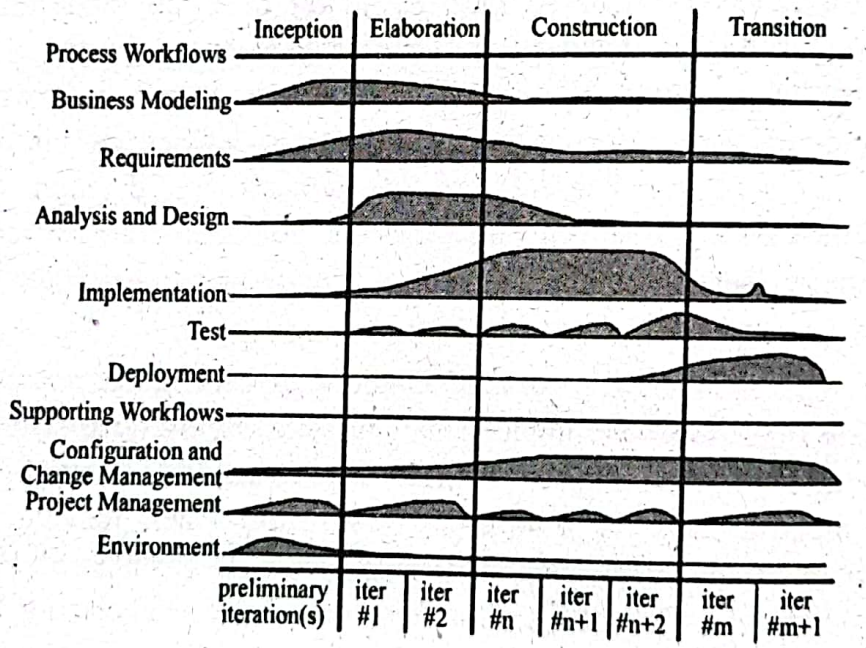
1.7 Object Oriented Development Cycle

For the software development through object-oriented approach one of the most used approach is Rational Unified Process (RUP). The Rational Unified Process is one such life cycle approach that is especially well-suited to the UML. The goal of the Rational Unified Process is to enable the production of highest quality software that meets end-user needs within predictable schedules and budgets. The Rational Unified Process captures some of the

best current software development practices in a form that is tailorable for a wide range of projects and organizations

Characteristics of RUP

1. The Rational Unified Process is an iterative and incremental process.
2. The Rational Unified Process's activities emphasize the creation and maintenance of models rather than paper documents.
3. Development under the Rational Unified Process is architecture-centric.
4. Development activities under the Rational Unified Process are use case- driven.
5. The Rational Unified Process is a configurable process.
6. The Rational Unified Process encourages objective ongoing quality control and risk management.



Phases of RUP:

1. Inception
2. Elaboration
3. Construction
4. Transition

Inception is the first phase of the process, when the seed idea for the development is brought up to the point of being at least internally sufficiently well-founded to warrant entering into the elaboration phase.

Elaboration is the second phase of the process, when the product vision and its architecture are defined. In this phase, the system's requirements are articulated, prioritized, and baselined. A system's requirements may range from general vision statements to precise evaluation criteria, each specifying particular functional or nonfunctional behavior and each providing a basis for testing.

Construction is the third phase of the process, when the software is brought from an executable architectural baseline to being ready to be transitioned to the user community. Here also, the system's requirements and especially its evaluation criteria are constantly reexamined against the business needs of the project, and resources are allocated as appropriate to actively attack risks to the project.

Transition is the fourth phase of the process, when the software is turned into the hands of the user community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated, and features that didn't make an earlier release are added.

Agile Development

Agile development methods usually apply timeboxed iterative and evolutionary development, employ adaptive planning, promote incremental delivery, and include other values and practices that encourage *agility* rapid and flexible response to change. It is not possible to exactly define **agile methods**, as specific practices vary widely. However, short timeboxed iterations with evolutionary refinement of plans, requirements, and design is a basic practice the methods share. In addition, they promote practices and principles that reflect an agile sensibility of simplicity, lightness, communication, self-organizing teams, and more. Example practices from the Scrum agile method include a *common project workroom* and *self-organizing teams* that coordinate through a daily stand-up meeting with four special questions each member answers.

Agile Principles and Strengths

- Agile highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Agile processes promote sustainable development.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.
- Continuous attention to technical excellence and good design enhances agility.

- Business people and developers must work together daily throughout the project.
- Simplicity the art of maximizing the amount of work not doneis essential.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
- Working software is the primary measure of progress.

1.8 Overview of the Unified Modeling Language: UML Fundamentals and Notations

UML (Unified Modeling Language)

The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML is used to visualize, specify, construct, and document the artifacts of a software intensive system. The UML is only a language and so is just one part of a software development method. The UML is process independent, although optimally it should be used in a process that is use case driven, architecture-centric, iterative, and incremental.

An Overview of the UML

The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

the artifacts of a software-intensive system.

The UML Is a Language

A language provides a vocabulary and the rules for combining words in that vocabulary for the purpose of communication. A modeling language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system. A modeling language such as the UML is thus a standard language for software blueprints.

The UML Is a Language for Visualizing

For many programmers, the distance between thinking of an implementation and then pounding it out in code is close to zero. You think it, you code it. In fact, some things are best cast directly in code. Text is a wonderfully minimal and direct way to write expressions and algorithms. In such cases, the programmer is still doing some modeling, albeit entirely mentally. He or she may even sketch out a few ideas on a white board or on a napkin. However, there are several problems with this. First, communicating those conceptual models to others is error-prone unless everyone involved speaks the same language. Typically, projects and organizations develop their own language, and it is difficult to understand what's going on if you are an outsider or new to the group. Second, there are some things about a software system you can't understand unless you build models that transcend the textual programming language. For example, the meaning of a class hierarchy can be inferred, but not directly grasped, by staring at the code for all the classes in the hierarchy. Similarly, the physical distribution and possible migration of the objects in a Webbased system can be inferred, but not directly grasped, by studying the system's code. Third, if the developer who cut the code never wrote down the models that are in his or her head, that information would be lost forever or, at best, only partially recreatable from the implementation, once that developer moved on. Writing models in the UML addresses the third issue: An explicit model facilitates communication. Some things are best modeled textually; others are best modeled graphically. Indeed, in all interesting systems, there are structures that transcend what can be represented in a programming language. The UML is such a graphical language.

The UML Is a Language for Specifying

In this context, *specifying* means building models that are precise, unambiguous, and complete. In particular, the UML addresses the specification of all the important analysis, design, and implementation decisions that must be made in developing and deploying a software-intensive system

The UML Is a Language for Constructing

The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in the UML to a programming language such as Java, C++, or Visual Basic, or even to tables in a relational database or the persistent store of an object-oriented database. Things that are best expressed graphically are done so graphically in the UML, whereas things that are best expressed textually are done so in the programming language.

The UML Is a Language for Documenting

A healthy software organization produces all sorts of artifacts in addition to raw executable code.

These artifacts include (but are not limited to)

- Requirements
- Design
- Project plans
- Prototypes
- Architecture
- Source code
- Tests
- Releases

Depending on the development culture, some of these artifacts are treated more or less formally than others. Such artifacts are not only the deliverables of a project, they are also critical in controlling, measuring, and communicating about a system during its development and after its deployment. The UML addresses the documentation of a system's architecture and all of its details.

A Conceptual Model of the UML

To understand the UML, you need to form a conceptual model of the language, and this requires learning three major elements: the UML's basic building blocks, the rules that dictate how those building blocks may be put together, and some common mechanisms that apply throughout the UML

Building Blocks of the UML

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things 2. Relationships 3. Diagrams

Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

Things (Models) in the UML

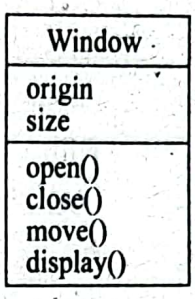
There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

Structural Things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things. They are:

1. **Class:** A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations.

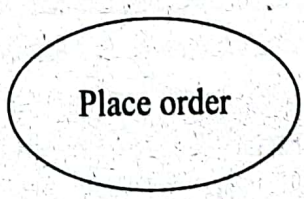


2. **Interface:** An interface is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior. An interface defines a set of operation specifications (that is, their signatures) but never a set of operation implementations. Graphically, an interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface,

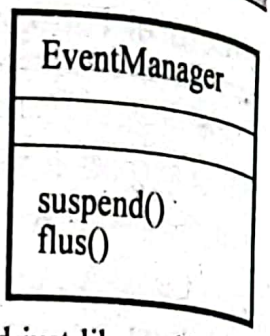
ISpelling

3. **Collaborations:** *A collaboration* defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several collaborations. These collaborations therefore represent the implementation of patterns that make up a system. Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name

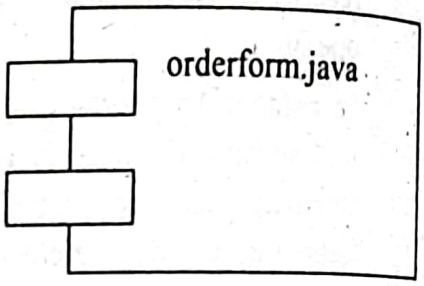
4. **Use Case:** A *use case* is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name.



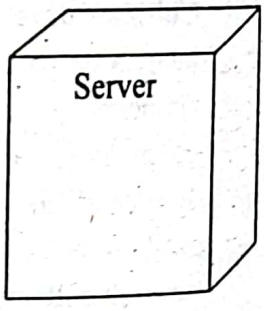
5. **Active Classes:** An *active class* is a class whose objects own one or more processes or threads and therefore can initiate control activity. An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations.



6. **Components:** A *component* is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. In a system, you'll encounter different kinds of deployment components, such as COM+ components or Java Beans, as well as components that are artifacts of the development process, such as source code files. A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations. Graphically, a component is rendered as a rectangle with tabs, usually including only its name.



7. **Nodes:** A *node* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. A set of components may reside on a node and may also migrate from node to node. Graphically, a node is rendered as a cube, usually including only its name,

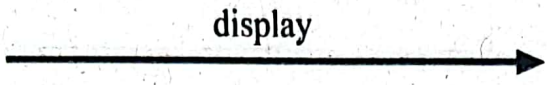


Behavioral Things

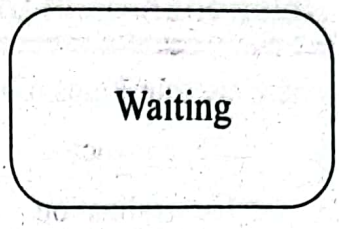
Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things. They are:

1. **Interaction:** An *interaction* is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose. The behavior of a society of objects or of an individual operation may be specified with an interaction. An

interaction involves a number of other elements, including messages, action sequences (the behavior invoked by a message), and links (the connection between objects). Graphically, a message is rendered as a directed line, almost always including the name of its operation



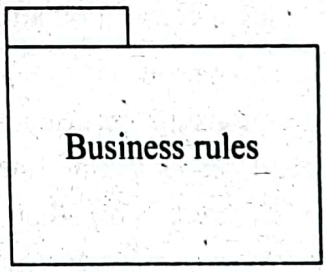
2. **State Machines:** A *state machine* is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. The behavior of an individual class or a collaboration of classes may be specified with a state machine. A state machine involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates,



Grouping Things

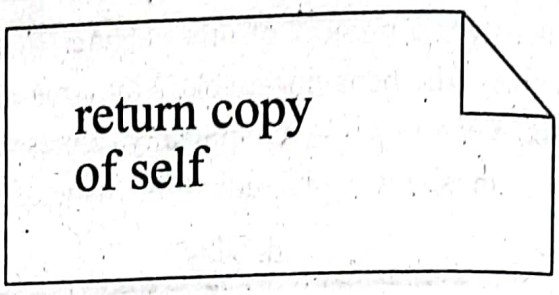
Grouping things are the organizational parts of UML models. These are the boxes into which a model can be decomposed. In all, there is one primary kind of grouping thing, namely, packages

A *package* is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package. Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time). Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents



Annotational Things

Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note. A *note* is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.



Relationships in the UML

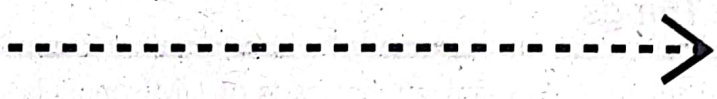
There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization

These relationships are the basic relational building blocks of the UML

Dependency

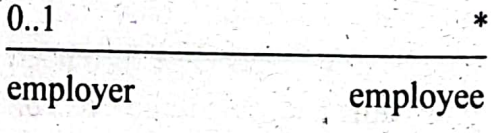
A *dependency* is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label



Association

An *association* is a structural relationship that describes a set of links, a link being a connection among objects.

Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names



Generalization

A *generalization* is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent

Realizations

A *realization* is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. You'll encounter realization relationships in two places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them. Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship.

Diagrams in the UML

A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). The UML includes nine such diagrams.

1. **Class diagram:** A class diagram shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.
2. **Object diagram:** An object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.
3. **Use case diagram:** A use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system
4. **Sequence diagram:** A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages.
5. **Communication/Collaboration diagram:** A collaboration diagram is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

6. **State chart diagram:** A statechart diagram shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic view of a system. They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems

7. **Activity diagram:** An activity diagram is a special kind of a statechart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.

8. **Component diagram:** A component diagram shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

9. **Deployment diagram:** A deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of an architecture. They are related to component diagrams in that a node typically encloses one or more components.

Common Mechanisms in the UML

A building is made simpler and more harmonious by the conformance to a pattern of common features. A house may be built in the Victorian or French country style largely by using certain architectural patterns that define those styles. The same is true of the UML. It is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

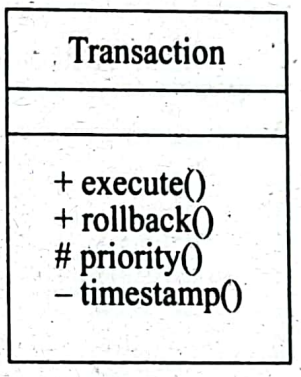
Specifications

The UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block. For example, behind a class icon is a specification that provides the full set of attributes, operations (including their full signatures), and behaviors that the class embodies; visually, that class icon might only show a small part of this specification. Furthermore, there might be another view of that class that presents a completely different set of parts yet is still consistent with the class's

underlying specification. You use the UML's graphical notation to visualize a system; you use the UML's specification to state the system's details. Given this split, it's possible to build up a model incrementally by drawing diagrams and then adding semantics to the model's specifications, or directly by creating a specification, perhaps by reverse engineering an existing system, and then creating diagrams that are projections into those specifications. The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion. The UML's diagrams are thus simply visual projections into that backplane, each diagram revealing a specific interesting aspect of the system.

Adornments

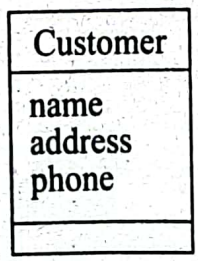
Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented systems. The class notation also exposes the most important aspects of a class, namely its name, attributes, and operations. A class's specification may include other details, such as whether it is abstract or the visibility of its attributes and operations. Many of these details can be rendered as graphical or textual adornments to the class's basic rectangular notation. For example, figure below shows a class, adorned to indicate that it is an abstract class with two public, one protected, and one private operation.

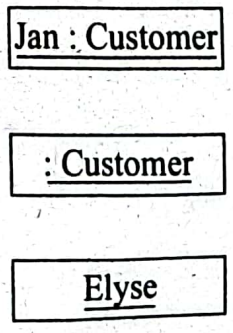


Common Divisions

In modeling object-oriented systems, the world often gets divided in at least a couple of ways.

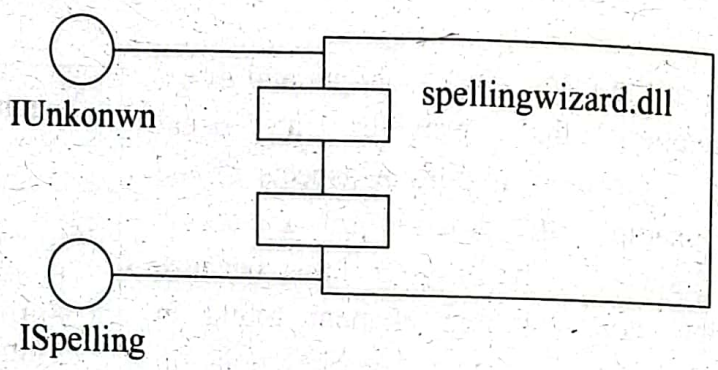
First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown below





In this figure, there is one class, named **Customer**, together with three objects: **Jan** (which is marked explicitly as being a **Customer** object), : **Customer** (an anonymous **Customer** object), and **Elyse** (which in its specification is marked as being a kind of **Customer** object, although it's not shown explicitly here).

Second, there is the separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown below.

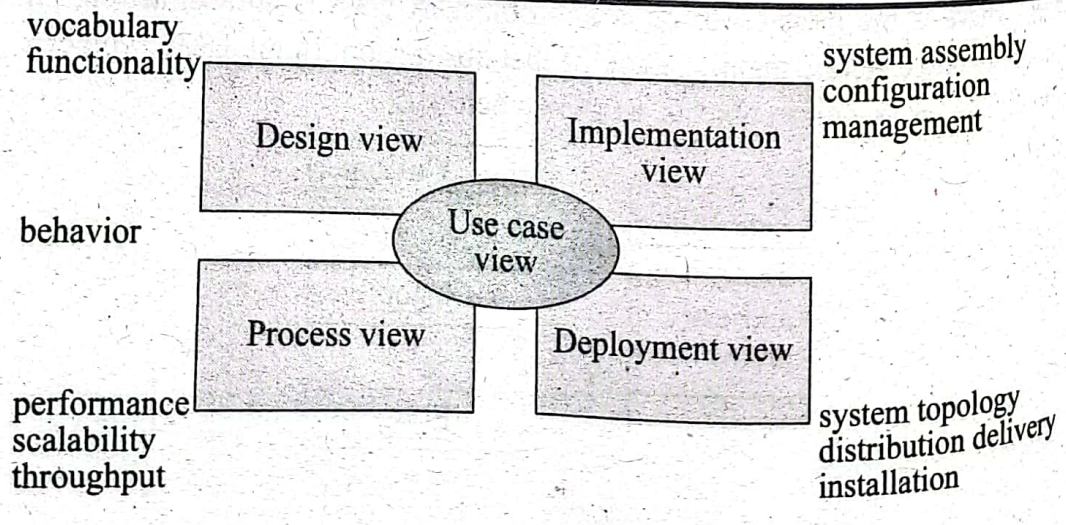


In this figure, there is one component named **spellingwizard.dll** that implements two interfaces, **IUnknown** and **ISpelling.**

Extensibility Mechanisms

The UML provides a standard language for writing software blueprints, but it is not possible for one closed language to ever be sufficient to express all possible nuances of all models across all domains across all time. For this reason, the UML is opened-ended, making it possible for you to extend the language in controlled ways.

Views of UML



The use case view of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers. This view doesn't really specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture. With the UML, the static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

The design view of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that represent these threads and processes.

The implementation view of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system. With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

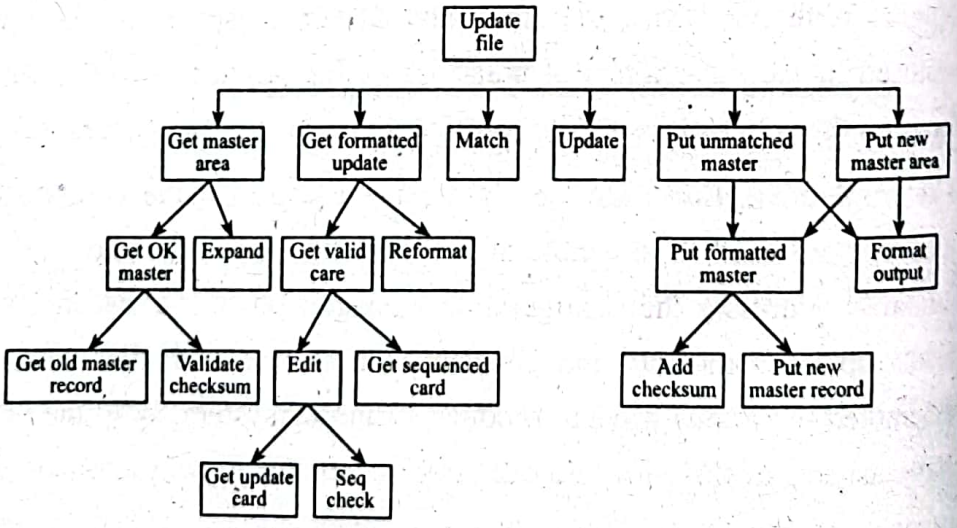
The deployment view of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture that most concern them. These five views also interact with one another nodes in the deployment view hold components in the implementation view that, in turn, represent the physical realization of classes, interfaces, collaborations, and active classes from the design and process views. The UML permits you to express every one of these five views and their interactions.

Additional

Algorithmic Decomposition

- This is the classical top-down structured design approach.
- In this approach, the system is decomposed using a state-machine approach. We decompose the large mapping from initial to final state by subdividing it into subcomponent state mappings, then further refining these state mappings, until we have gotten down to primitive instructions (which provide the finest granularity of state mapping specifiable on our computer).



Object-Oriented Decomposition

- In this alternative approach, we think of the system in terms of objects.
- We consider the whole system a single object, and decompose it into its relevant object parts. These objects have specific responsibilities with respect to the system as a whole, respond to specific stimuli, and interact in well-defined ways.

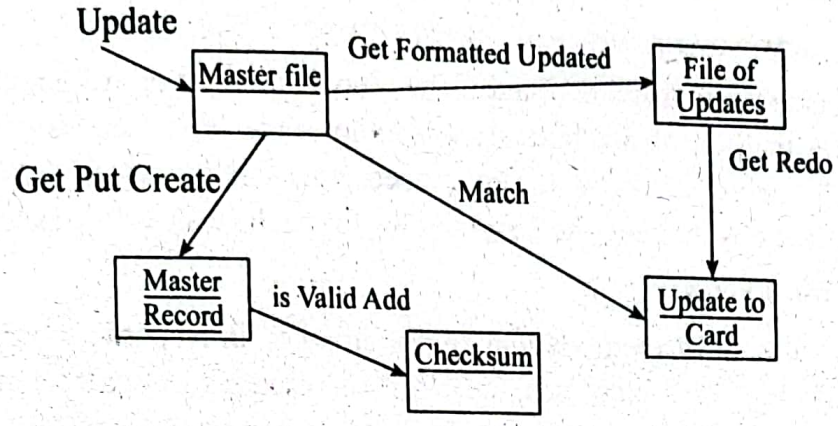


Fig.: Object-Oriented Decomposition

Algorithmic vs. Object-Oriented Decomposition

- Algorithmic view emphasizes the sequencing of events
- Object-oriented view emphasizes the agents who are partaking in these events
- Neither view is the *one true way*. We must employ multiple views if we want to solve complex problems with elegant, simple solutions.
- Experience indicates it is better to take the object-oriented view first when designing

Object Oriented Metrices

Weighted Method per Class

WMC measures the complexity of a class. Complexity of a class can for example be calculated by the cyclomatic complexities of its methods. High value of WMC indicates the class is more complex than that of low values

Depth of Inheritance Tree

The DIT metric is the maximum path length from a node to the tree's root. As a result, this metric computes how far down the inheritance hierarchy a class is declared. DIT represents the complexity of a class's behavior, the complexity of a class's design, and potential reuse. As a result, understanding a system with many inheritance layers can be difficult. A high DIT value indicates that many methods may be reused. (DIT) determine the depth of a class in a class hierarchy. This metric evaluates a class's reuse potential as well as its likely ease of maintenance. A class with a small DIT has a lot of reusability. (That is, it is more likely to be a broad abstract class.)

Number of Children

This metric indicates how many sub-classes will inherit the parent class's methods. Because more children in a class indicate more responsibility, the amount of testing will increase as NOC increases. As a result, NOC represents the effort required to test and reuse the class.

Coupling between Objects

Two objects are said to be coupled if they have an effect on one another. If a class uses a method or attribute from another class, that class is said to be coupled. A class's capacity to be reused will decline when CBO rises. Therefore, it is best to keep the CBO values for each class as low as feasible.

Response for a class

RFC is the number of methods that can be invoked in response to a message in a class. As the RFC increases, the effort required for testing also increases because the test sequence grows. If RFC increases, the overall design complexity of the class increases and becomes hard to understand. On the other hand lower values indicate greater polymorphism.

Old Exam Question-Answer

1. **Describe about Unified Modeling Language and list out to which models the following models belongs to usecase diagram, activity diagram, conceptual model, sequence diagram, collaboration diagram, state diagram and class diagram.** **[2079 Jestha Back]**

⇒ **Part I:** The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML is used to visualize, specify, construct, and document the artifacts of a software intensive system. The UML is process independent, although optimally it should be used in a process that is use case driven, architecture-centric, iterative, and incremental. The UML encompasses three kinds of building blocks:

1. Things 2. Relationships 3. Diagrams

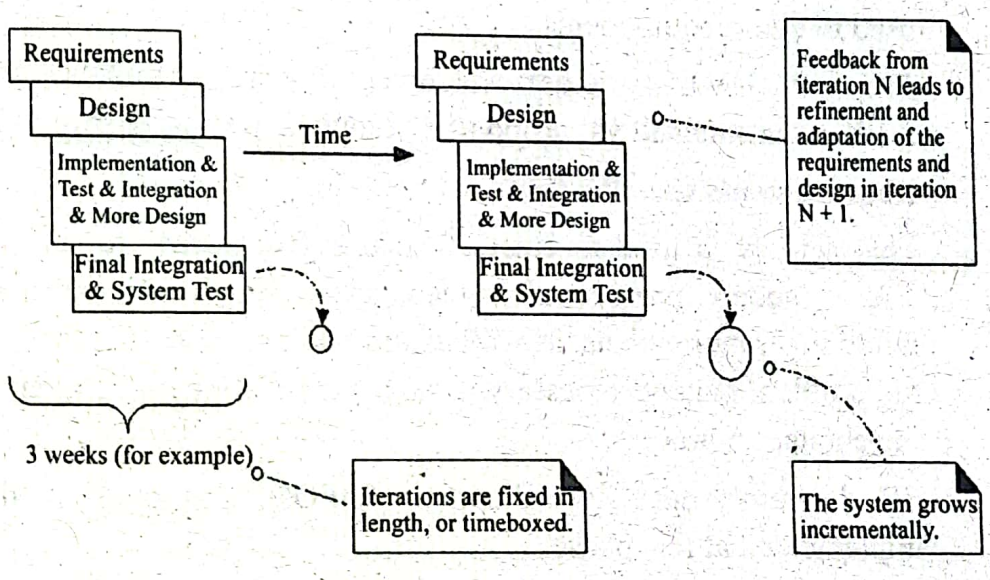
Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

Part II

Diagram	Model
UseCase Diagram	Structural Model
Activity Diagram	Behavioral Model
Conceptual Model	Conceptual Model
Sequence Diagram	Behavioral Model
Structural Model	Structural Model
State Diagram	Behavioral Model
Class Diagram	Structural Model

2. **Describe incremental and iterative development process with suitable example.** **[2079 Jestha Back]**

⇒ In Iterative development lifecycle approach development is organized into a series of short, fixed-length (for example, three-week) mini-projects called **iterations**; the outcome of each is a tested, integrated, and executable *partial* system. Each iteration includes its own requirements analysis, design, implementation, and testing activities. The iterative lifecycle is based on the successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation as core drivers to converge upon a suitable system. The system grows incrementally over time, iteration by iteration, and thus this approach is also known as **iterative and incremental development** (see Figure Below). Because feedback and adaptation evolve the specifications and design, it is also known as **iterative and evolutionary development**.



Example

As an example (not a recipe), in a three-week iteration early in the project, perhaps one hour Monday morning is spent in a kickoff meeting with the team clarifying the tasks and goals of the iteration. Meanwhile, one person reverse-engineers the last iteration's code into UML diagrams (via a CASE tool), and prints and displays noteworthy diagrams. The team spends the remainder of Monday at whiteboards, working in pairs while agile modeling, sketching rough UML diagrams captured on digital cameras, and writing some pseudocode and design notes. The remaining days are spent on implementation, testing (unit, acceptance, usability, ...), further design, integration, and daily builds of the partial system. Other activities include demonstrations and evaluations with takeholders, and planning for the next iteration.

3. Describe the major components of requirement analysis with suitable example. **[2079 Jestha Back]**

⇒ Requirement Process Consists of the following activities:

1. Requirements elicitation
2. Requirements specification
3. Requirements verification and validation
4. Requirements management

Requirement Elicitation:

It relates to the many methods used to learn about the project topic and needs. Customers, business manuals, existing software of the same sort, standards, and other project stakeholders are among the different sources of domain knowledge. Interviews, brainstorming, task analysis, the Delphi approach, prototyping, and other techniques are used to elicit requirements.

The UP (UML Patterns) welcomes any requirements elicitation method that can add value and that increases user participation.

Requirements specification

This activity is used to create formal software requirement models. These models provide all needs, including functional and non-functional requirements, as well as limitations. More knowledge about the problem may be necessary during specification, which may trigger the elicitation process.

ER diagrams, data flow diagrams (DFDs), and other models are implemented at this phase.

Requirements verification and validation

Verification: It refers to the group of tasks that guarantee the software carries out a particular function appropriately.

Validation: It refers to a different set of duties that guarantee the software developed may be linked back to client specifications.

Requirements management

The process of evaluating, recording, tracking, prioritizing, and deciding on requirements, as well as regulating communication to relevant stakeholders, is known as requirement management. This stage deals with the changing nature of requirements. It should be assured that the SRS is as changeable as feasible in order to integrate changes in end-user needs at subsequent stages.

4. **What are the differences of OO based system to other conventional design in context with encapsulation and inheritance?**

⇒ See in book

5. **Compare between Function oriented and Object oriented system. What are the advanced relationships on Use case diagram? Explain <<include>> and <<extend>> relationship with suitable diagram.**

[2078 Chaitra Regular, 2075 Bhadra regular, 2075 Baishak Back]

⇒ **Part I**

Functional Programming	Object Oriented Programming
Programs are divided into functions	Programs are divided into objects.
Importance is given to the functions not to the data.	Importance is given to the data rather than the functions.
Encapsulation is not possible.	Encapsulation is possible.
It generally follows "Top-Down" Approach	It generally follows "Bottom-Up" Approach
It provides less flexibility and abstraction as compared to object-oriented programming.	It is less difficult to modify object-oriented programs and reuse code as compared to structured programs.

Part II

Relationship in use case diagrams:

1. **Generalization**

   Generalization among use cases is just like generalization among classes. Here it means that the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears (both the parent and the child may have concrete instances).

2. **Association**

   An association is a structural relationship that describes a set of links; a link is a connection among objects.

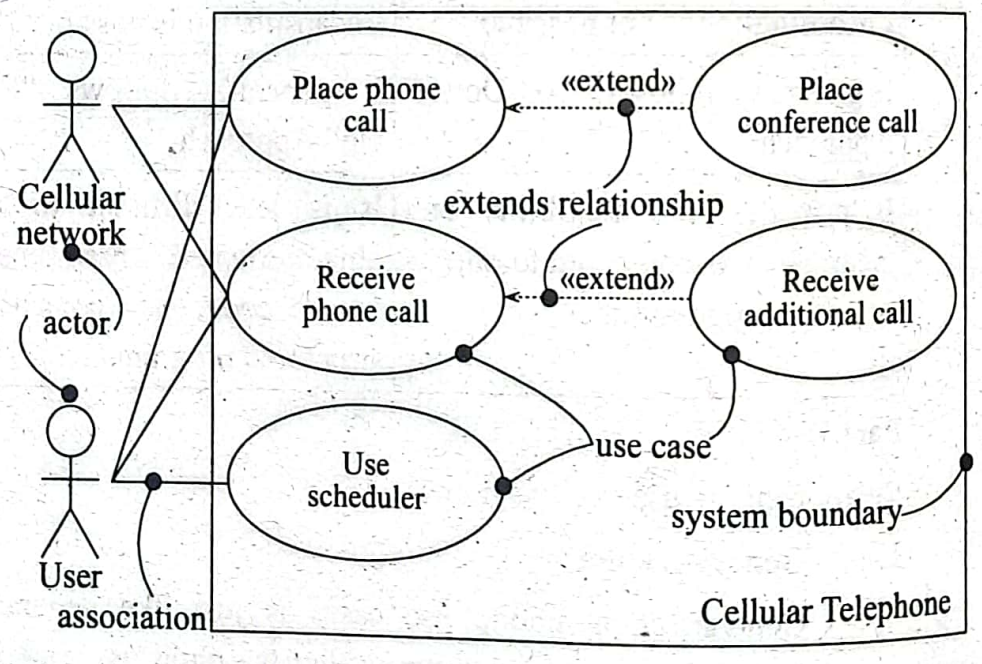
3. **Include**

An include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base. The include relationship is essentially an example of delegation you take a set of responsibilities of the system and capture it in one place (the included use case), then let all other parts of the system (other use cases) include the new aggregation of responsibilities whenever they need to use that functionality

4. **Extend**

An extend relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case. The base use case may stand alone, but under certain conditions, its behavior may be extended by the behavior of another use case.

Example: Use case diagram for cellular Telephone



6. **How does requirement elicitation process happen in Object Oriented Analysis? List the different types of requirement used to describe the object oriented system.** **[2078 Chaitra Regular]**

⇒ **Part I**

Requirement Elicitation: It relates to the many methods used to learn about the project topic and needs. Customers, business manuals, existing software of the same sort, standards, and other project stakeholders are among the different sources of domain knowledge.

Interviews, brainstorming, task analysis, the Delphi approach, prototyping, and other techniques are used to elicit requirements.

The UP (UML Patterns) welcomes any requirements elicitation method that can add value and that increases user participation.

Part II

Functional Requirement: A system or one of its components is defined by a functional requirement. It outlines the functions that software must carry out. The inputs, its behavior, and outputs make up a function. A system's likely function can be determined by a computation, data manipulation, business process, user interaction, or any other specialized feature. In software engineering, functional requirements assist you in capturing the intended behavior of the system. This behavior may be represented in terms of actions, processes, or tasks that a system must carry out.

Non Functional Requirement: A software system's quality attribute is defined by a non-functional need. They stand for a set of criteria used to assess a system's particular performance. For example, how quickly does the website load? To guarantee the overall software system's usability and efficiency, a non-functional requirement is necessary. Systems that don't meet non-functional requirements may not be able to meet user needs.

Requirement Types:

- **Functional** features, capabilities, security
- **Usability, human** factors, help, documentation.
- **Reliability, frequency** of failure, recoverability, predictability.
  - **Performance** response times, throughput, accuracy, availability, resource usage.
  - **Supportability adaptability,** maintainability, internationalization, configurability.

7. **How can you express exceptions in UML diagrams? Illustrate with possible exceptions in any suitable scenario.**

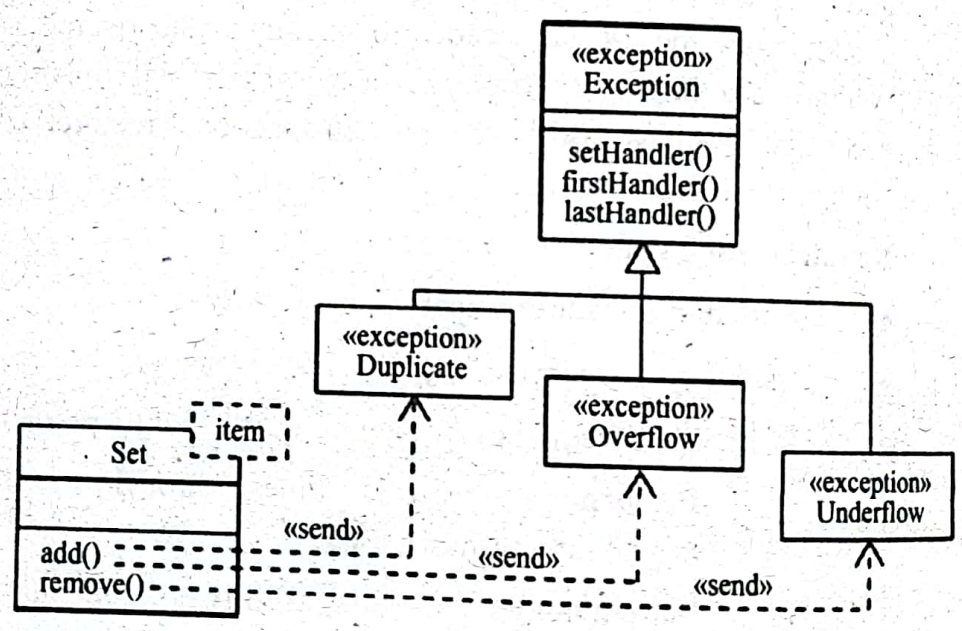
[2075 Bhadra Regular]

⇒ To model exceptions,

- For each class and interface, and for each operation of such elements, consider the exceptional conditions that may be raised.

- Arrange these exceptions in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions, as necessary.
- For each operation, specify the exceptions that it may raise. You can do so explicitly (by showing send dependencies from an operation to its exceptions) or you can put this in the operation's specification.

Models a hierarchy of exceptions that may be raised by a standard library of container classes, such as the template class Set. This hierarchy is headed by the abstract signal Exception and includes three specialized exceptions: Duplicate, Overflow, and Underflow. As shown, the add operation raises Duplicate and Overflow exceptions, and the remove operation raises only the Underflow exception. Alternatively, you could have put these dependencies in the background by naming them in each operation's specification. Either way, by knowing which exceptions each operation may send, you can create clients that use the Set class correctly.



8. Describe functional and nonfunctional requirements with suitable examples. **[2075 Baishak Back]**

⇒ **Functional Requirement:**

A system or one of its components is defined by a functional requirement. It outlines the functions that software must carry out. The inputs, its behavior, and outputs make up a function. A system's likely function can be determined by a computation, data manipulation, business process, user interaction, or any other specialized feature. In

software engineering, functional requirements assist you in capturing the intended behavior of the system. This behavior may be represented in terms of actions, processes, or tasks that a system must carry out.

Non Functional Requirement:

A software system's quality attribute is defined by a non-functional need. They stand for a set of criteria used to assess a system's particular performance. For example, how quickly does the website load? To guarantee the overall software system's usability and efficiency, a non-functional requirement is necessary. Systems that don't meet non-functional requirements may not be able to meet user needs.

Example:

Requirement Types:

- **Functional** features, capabilities, security
- **Usability, human** factors, help, documentation.
- **Reliability, frequency** of failure, recoverability, predictability.
  - **Performance** response times, throughput, accuracy, availability, resource usage.
  - **Supportability adaptability,** maintainability, internationalization, configurability.

9. What are the strengths of the agile development method? Explain how the requirement elicitation process happens in OOA.

⇒ **Part I**

Agile Principles and Strengths

- Agile highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Agile processes promote sustainable development.
- Welcome changing requirements, even late in development.
- Agile processes harness change for the customer's competitive advantage.
- The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.

- Continuous attention to technical excellence and good design enhances agility.
- Business people and developers must work together daily throughout the project.
- Simplicity the art of maximizing the amount of work not done is essential.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
- Working software is the primary measure of progress.

Part II:

Requirement Elicitation:

It relates to the many methods used to learn about the project topic and needs. Customers, business manuals, existing software of the same sort, standards, and other project stakeholders are among the different sources of domain knowledge. Interviews, brainstorming, task analysis, the Delphi approach, prototyping, and other techniques are used to elicit requirements.

The UP (UML Patterns) welcomes any requirements elicitation method that can add value and that increases user participation.

10. What are the meaning of Object and Encapsulation in terms of OOAD? Explain briefly with example. [2074 Regular Bhadra]

⇒ **Object:** Objects are the instance of the class (i.e., Objects are created from class). Each object that are created are unique. Each object has its own identity, behavior, and state.

(Identity separates each object and makes each object unique from other objects, behavior is the method that object performs and state is the value of an attribute (characteristic property of an object).

Encapsulation: Encapsulation refers to the concept of combining data and processes that operate on that data into a single entity. This idea is also frequently used to hide an object's internal representation or state from the outside world. This is also known as called data hiding.

11. In OOAD, there are various types of models, like conceptual, structural, behavioral, etc. What is the significance of these many different types of model? Explain with illustrative examples.

[2074 Regular Bhadra]

⇒ **Conceptual Model:** Conceptual model represents the concepts of the problem domain. The main use of Conceptual model is that it makes the problem clearer and makes understand of the problem requirement much easier.

Structural Model: Conceptual model represents the structure of the problem domain.

Behavioral Model: Behavior model represents the behavior of the system. It describes the interaction between the system.

Specification Model: Specification Model describes the software abstractions with the specifications and interface.

Implementation Model: Implementation Model describe the software with implementations in a particular technology.

12. What do you mean by Unified process (UP) in OOAD? Explain the phases with suitable diagrams.

⇒ **Part I:** The Unified Software Development Process or Unified Process is an iterative and incremental software development process framework. The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process (RUP). Other examples are OpenUP and Agile Unified Process.

Part II:

Phases of RUP:

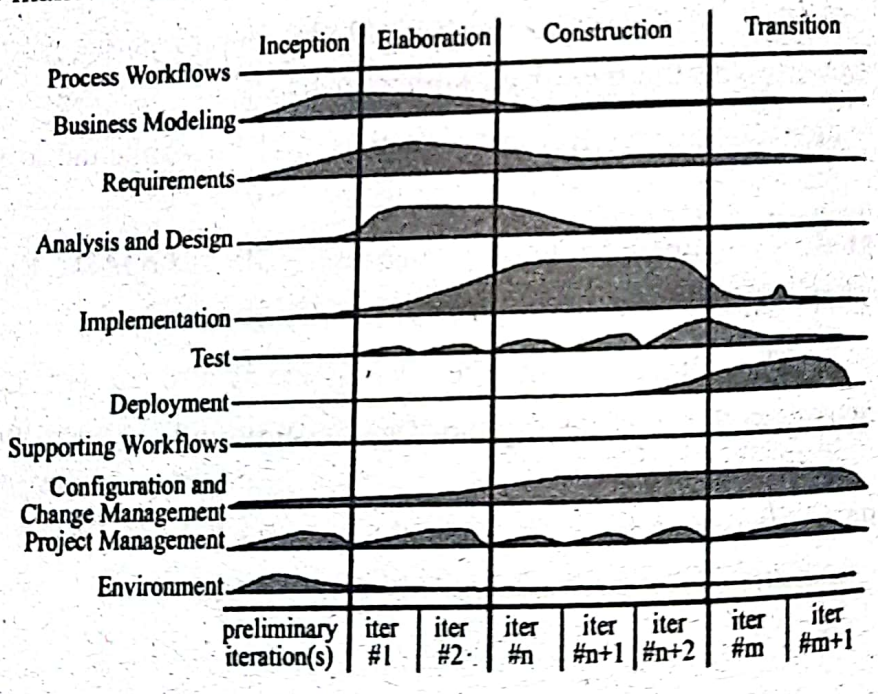
1. Inception
2. Elaboration
3. Construction
4. Transition

Inception is the first phase of the process, when the seed idea for the development is brought up to the point of being at least internally sufficiently well-founded to warrant entering into the elaboration phase.

Elaboration is the second phase of the process, when the product vision and its architecture are defined. In this phase, the system's requirements are articulated, prioritized, and baselined. A system's requirements may range from general vision statements to precise evaluation criteria, each specifying particular functional or nonfunctional behavior and each providing a basis for testing.

Construction is the third phase of the process, when the software is brought from an executable architectural baseline to being ready to be transitioned to the user community. Here also, the system's requirements and especially its evaluation criteria are constantly reexamined against the business needs of the project, and resources are allocated as appropriate to actively attack risks to the project.

Transition is the fourth phase of the process, when the software is turned into the hands of the user community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated, and features that didn't make an earlier release are added.



13. What do you mean by Unified process (UP) in OOAD? Explain the phases with suitable diagrams.

⇒ The Unified Software Development Process or Unified Process is an iterative and incremental software development process framework. The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process (RUP). Other examples are OpenUP and Agile Unified Process.

Phases are described in theory section.

14. What makes the use-case diagram as an important in UML diagrams?
[2073 Bhadra Regular]

⇒ **1. To model the context of a system**

Modeling the context of a system involves drawing a line around the whole system and asserting which actors lie outside the system and interact with it. Here, you'll apply use case diagrams to specify the actors and the meaning of their roles

2. To model the requirements of a system

Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do it. Here, you'll apply use case diagrams to specify the desired behavior of the system. In this manner, a use case diagram lets you view the whole system as a black box; you can see what's outside the system and you can see how that system reacts to the things outside, but you can't see how that system works on the inside.

15. What are requirements process alternatives before having OO Analysis?
[2073 Bhadra Regular]

⇒ Before OOA there were different methods for the requirement process. Some of the are as follows:

1. Business process modeling notation (BPMN)

BPMN has its own symbols and elements, this method is comparable to making process flowcharts. Graphs representing the business process are produced using business process modeling and notation. Understanding the business process is made simpler by these graphs. The process improvement methodology known as BPMN is quite well-liked.

2. Flowchart technique

A flowchart illustrates the sequential flow and control logic of a collection of connected actions. There are several types of flowcharts, including top-down, cross-functional, and linear. System interactions, data flows, and other things can be depicted in a flowchart. Flow charts are simple to comprehend and may be used by team members on both the technical and non-technical sides. The use of flowcharts helps in highlighting a process' essential characteristics.

3. **Data flow diagram**

This method is used to illustrate systems and processes visually that are complicated and challenging to describe in text. Data flow diagrams show how information moves via a system or a process. Additionally, it comprises the many subprocesses, data stores, and data inputs and outputs. DFD uses standardized notations and symbols to define multiple entities and their relationships. It is simpler to see any flaws when all the system's components are visualized. Then, these flaws are fixed in an effort to develop a reliable solution.

4. **Gantt Charts**

Gantt charts are implemented in project planning because they give a visual depiction of the tasks that are planned out together with the timelines. The Gantt charts make it easier to understand what is expected to be finished by when. All of the project's tasks' start and end dates can be displayed in a single view.

16. Differentiate between functional and non-functional requirement. What are the relationships in Use Case Diagrams and explain <<include>> and <<extend>> relationships with diagram?

[2072 Ashwin Regular]

⇒

Functional Requirements	Non Functional Requirements
A system or one of its components is defined by a functional requirement.	A software system's quality characteristic is defined by a non-functional requirement.
It deals with "What should the software system do?"	It places constraints on "How should the software system fulfill the functional requirements?"
Functional requirement is specified by User.	Non-functional requirement is specified by technical peoples e.g. Architect, Technical leaders and software developers.
It is mandatory.	It is not mandatory.
It is captured in use case.	It is captured as a quality attribute.

Defined at a component level.	Applied to a system as a whole.
Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
Functional Testing like System, Integration, End to End, API testing, etc are done.	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc are done.
Usually easy to define.	Usually more difficult to define.
Example 1) Authentication of user whenever he/she logs into the system. 2) System shutdown in case of a cyber attack. 3) A Verification email is sent to user whenever he/she registers for the first time on some software system.	**Example** 1) Emails should be sent with a latency of no greater than 12 hours from such an activity. 2) The processing of each request should be done within 10 seconds 3) The site should load in 3 seconds when the number of simultaneous users are > 10000

Part II

Relationship in use case diagrams:

1. **Generalization**

   Generalization among use cases is just like generalization among classes. Here it means that the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears (both the parent and the child may have concrete instances).

2. **Association**

   An association is a structural relationship that describes a set of links; a link is a connection among objects.

3. **Include**

   An include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base. The include relationship is essentially an example of delegation you take a set of responsibilities of the system and capture it in one place (the

included use case), then let all other parts of the system (other use cases) include the new aggregation of responsibilities whenever they need to use that functionality

4. **Extend**

An extend relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case. The base use case may stand alone, but under certain conditions, its behavior may be extended by the behavior of another use case.

17. Explain all types of external actors in relation to System under Discussion (SuD). **[2072 Ashwin regular]**

⇒ An **actor** is anything with behavior, including the system under discussion (SuD) itself when it calls upon the services of other systems.

1. **Primary actor** has user goals fulfilled through using services of the SuD. For example, the cashier.

   Why identify? To find user goals, which drive the use cases.

2. **Supporting actor** provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.

   Why identify? To clarify external interfaces and protocols.

3. **Offstage actor** has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.

   Why identify? To ensure that *all* necessary interests are identified and satisfied. Offstage actor interests are sometimes subtle or easy to miss unless these actors are explicitly named.

Practice Questions

1. What is Object-Oriented Analysis?
2. What are different steps involved during analysis?
3. What are different steps involved during designing?
4. What do you mean by static and dynamic modeling?

5. What is Object-Oriented Design?
6. What are the key steps of OOAD?
7. What is Object-Oriented Analysis and Design?
8. What is the UML? List out its Perspectives.
9. Write down the benefits of the Agile UP.
10. List the four phases of UP.
11. Define Requirements and mention its types.
12. Define Actor. What are different kinds of Actors? Explain briefly.
13. What are Use Case Diagrams?
14. Define Inception. Explain about artifacts of Inception.
15. Explain about Unified process phases.(Hint: Iterative Development, Additional UP Best Practices and Concepts, The UP Phases and Schedule, The UP Disciplines , Process Customization and the Development Case, The Agile UP, The Sequential "Waterfall
16. List out the components of Object-Oriented Analysis and Design.
17. Define and Give examples for each: 1. Milestone 2. Object 3. Constraints 4. Encapsulation 5.Abstraction 6. Class 7. Scenarios 8. Unified Software Development Process
18. What are different kinds of diagrams in UML?
19. Describe different things in UML.
20. Describe different relationship in UML
21. Describe different common mechanisms of UML.
22. Examine the purpose of using use cases to describe requirements.
23. Deliberate different notations of UML diagrams in detail.
24. List any two common ways to relate use cases and give suitable examples.
25. Explain in detail about unified processes in object oriented analysis and design? Explain phases with a neat diagram.
26. Describe the POS system? List the components of POS system.
27. List the various UML diagrams and explain the purpose of each diagram.
28. Difference between Reverse Engineering and Forward Engineering.
29. Describe Requirement Analysis Process in OOAD.

30. Difference between Algorithmic Decomposition and Object oriented decomposition.

31. Briefly describe object oriented metrics.

32. Let's say you own a small baking company, where you make and design custom cakes for different occasions. You now wish to take your business online, so that you could cater to a large customer bse. You hire a web development company to build an online cake store for you. This software product is build on the basis of the Unified Process Model (UPM).

    Define and explain UPM with its phases for developing the above online baking company.

33. What is the purpose of the use case model? Identify the actors, scenarios and uses cases for library management system.

❑❑❑

Chapter 2

Object Oriented Analysis

2.1 Building Domain/Conceptual Models

Domain Model is the outcome of the object oriented analysis performed in the middle level of abstraction i.e. Business Area Level among the three levels listed as:

1. Enterprise Level
2. Business Area Level
3. Application Level

This analysis can also be known as domain analysis and helps an organization to create a library of reusable classes that will be widely applicable to its entire category of applications internally.

Domain Analysis

Domain Model

A domain model is the visual representation of conceptual classes or real world objects in a domain of interest. It is not a set of diagrams describing software classes. Domain models are also known as conceptual models or domain object models.

A domain model using UML notation can be represented with a set of class diagrams in which no operations are defined. It may show:

- Domain models or conceptual classes
- Associations between conceptual classes
- Attributes of conceptual classes

Below is an example of a partial domain model. It illustrates that the conceptual class of *Payment* and *Sale* are significant in this domain.

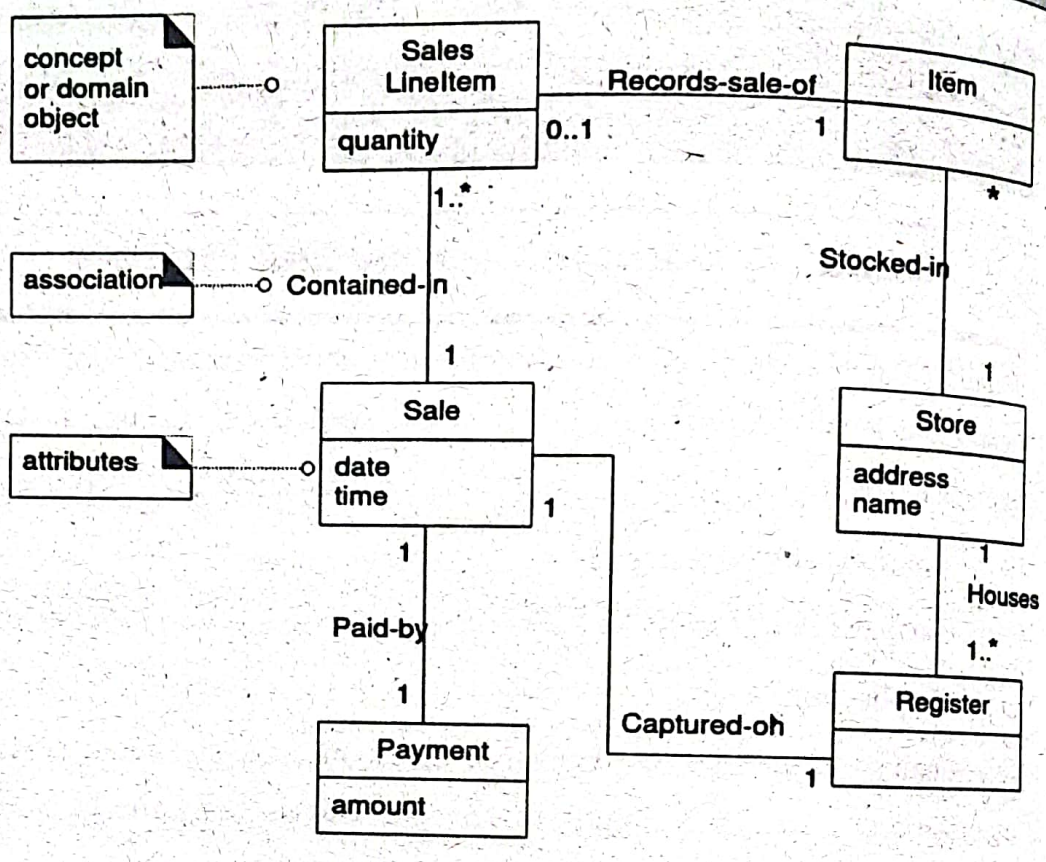


Fig. 2.1 : Partial Domain Model – a visual dictionary

Upon reflecting on Fig 2.1 for a moment, it visualizes conceptual classes in the domain which can be broadly applicable. It also depicts an abstraction of the conceptual classes, because there are many things one can communicate about the objects. It is easy to comprehend the discrete elements and their relationships in this visual language, since it's easier for the human brain to understand things visually rather than in sentences. Thus, the domain model may be considered a visual dictionary of the important abstractions, domain vocabulary, and information content of the domain.

Conceptual Classes

As discussed above, we now know that the domain model illustrates conceptual classes or vocabulary in the domain. A conceptual class is an idea, thing, or object. More formally, a conceptual class may be considered in terms of its symbol, intension, and extension.

- **Symbol :** words or images representing a conceptual class
- **Intension :** the definition of a conceptual class
- **Extension :** the set of examples to which the conceptual class applies

To better understand conceptual classes, let us take the same example of the purchase transaction as above.

We may choose to name it by the symbol *Sale*. The intension of a *Sale* may state that it "represents the event of a purchase transaction, and has a date and time." The extension of *Sale* is all the examples of *Sales*.

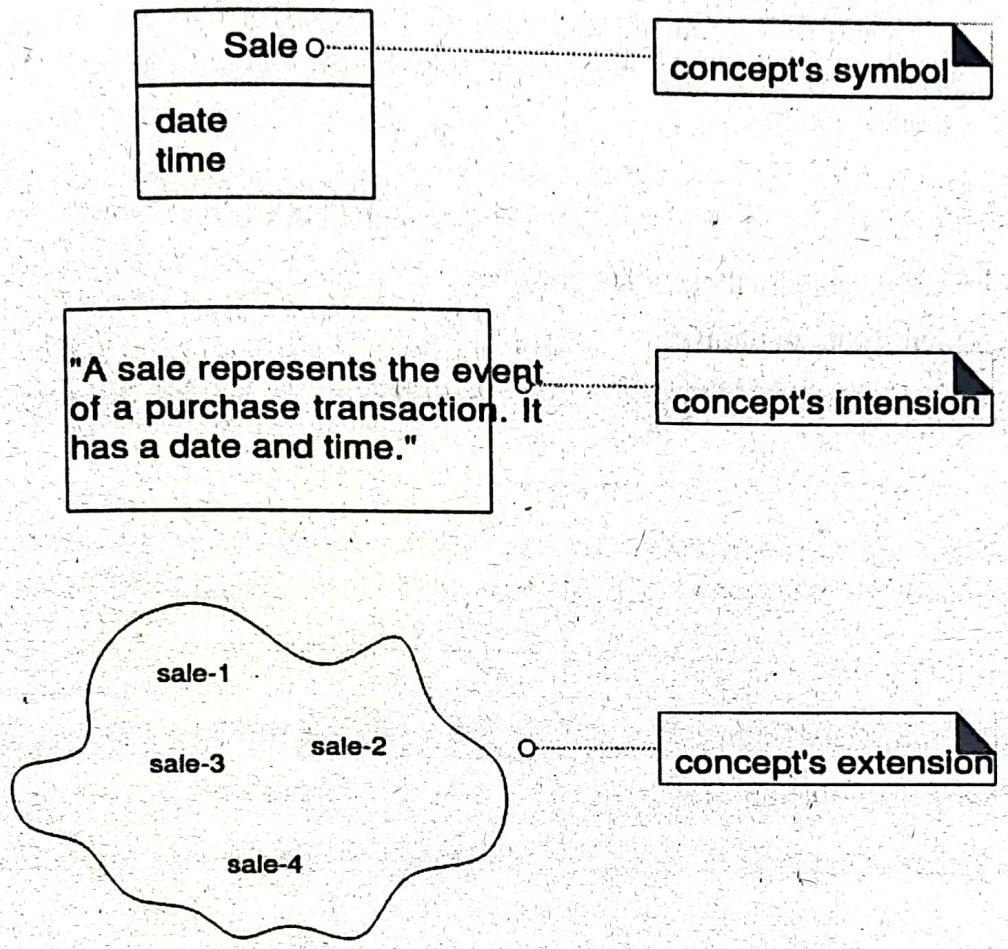


Fig. 2.2 : A conceptual class with symbol, intension and extension.

Creating a Domain Model

Creating a domain model is the domain analysis phase. This can be done as:

1. Conceptual classes identification
2. Draw them as classes in UML class diagram
3. Add associations and attributes

Conceptual Class Identification

The following is a useful guideline in identifying conceptual classes:

- It is better to over specify a domain model with lots of fine grained conceptual classes than to underspecify it.
- Do not think that a domain model is better if it has fewer conceptual classes; quite the opposite tends to be true.
- It is common to miss conceptual classes during the initial identification step, and to discover them later during the consideration of attributes or associations, or during design work. When found, they may be added to the domain model.

- Do not exclude a conceptual class simply because the requirements do not indicate any obvious need to remember information about it, or because the conceptual class has no attributes.
- It is valid to have attributes conceptual classes, or conceptual classes which have a purely behavioral role in the domain instead of an information role.

Strategies to identify Conceptual Classes

These two techniques are used to identify conceptual classes:

1. Use a conceptual class category list
2. Identify noun phrases

Another excellent technique for domain modeling is the use of analysis patterns, which are existing partial domain models created by experts.

Using a Conceptual Class Category List

In this technique, we start with the creation of a domain model by making a list of candidate conceptual classes. Below table may contain many common categories that are usually worth considering, though not in any particular order of importance.

Conceptual Class Category	Examples
Physical or tangible objects	Register Airplane
Specifications, designs, or descriptions of things	Product Specification Flight Description
Places	Store Airport
Transactions	Sale Payment Reservation
Roles of people	Cashier Pilot
Containers of other things	Store Bin
Things in a container	Item Passenger
Abstract noun concepts	Hanger
Organizations	Sales Department Object Airline

Events	Sale Payment Meeting Flight

Finding Conceptual Classes with Noun Phrase Identification

Another useful technique in identifying conceptual classes for domain modeling is by identifying nouns and noun phrases in textual descriptions of a domain, and consider them as candidate conceptual classes or attributes.

The Basic flow for an example is below:

1. **Customer** arrives at a **POS checkout** with **goods** and/or **services** to purchase.
2. **Cashier** starts a new **sale**.
3. Cashier enters the **item identifier**.
4. System records **sale line items** and presents **item description, price,** and running total.
5. System presents the total with **taxes** calculated.
6. Cashier tells the customer the total, and asks for **payment**.
7. Customer pays and the system handles the payment.

2.2 Adding Associations and Attributes

1. **Adding Associations:** An association is a relationship between classes (more precisely, instances of those classes) that indicates some meaningful and interesting connection. It is represented as a line between classes with an association name. The association is inherently bidirectional, meaning that from instances of either class, logical traversal to the other is possible.

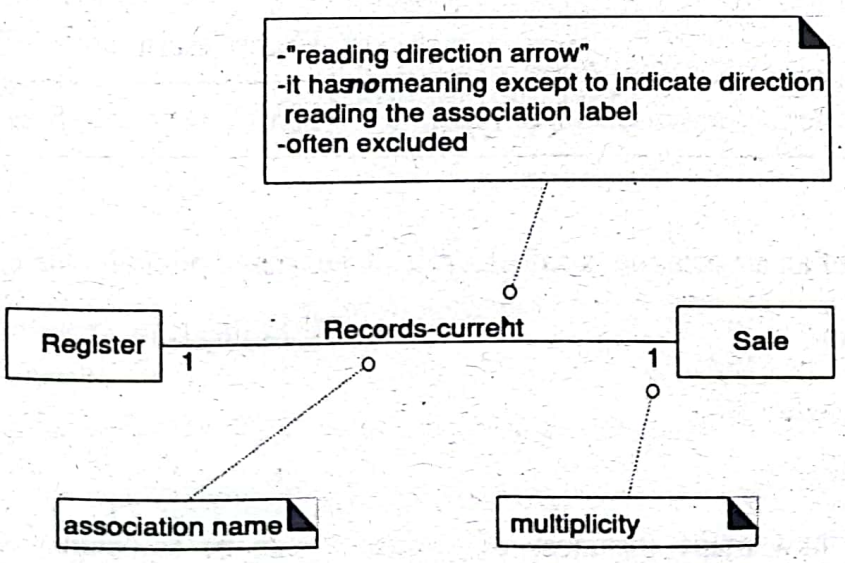


Fig. 2.3 : UML notation for associations.

Criteria for useful associations

- Associations worth noting usually imply knowledge of a relationship that needs to be preserved for some duration.
- Including the following associations in a domain model:
  - Associations for which knowledge of the relationship needs to be preserved for some duration ("need to know" associations)
  - Associations derived from the common associations lists.
- A domain model with *n* different conceptual classes - $n * (n-1)$ associations to other conceptual classes.

Finding Associations

Associations can be found using a common associations list as shown below. For this, start the addition of associations by using the list in a table.

Category	Examples
A is a physical part of B	Drawer — Register Wing — Airplane
A is a logical part of B	SalesLineItem — Sale FlightLeg — FlightRoute
A is physically contained in/on B	ItemDescription — Catalog Flight — FlightSchedule
A is a description for B	ItemDescription — Item FlightDescription — Flight
A is a line item of a transaction or report B	SalesLineItem — Sale

Roles

Each end of an association is called a role. Roles may optionally have:

- Name
- Multiplicity expression
- Navigability

Multiplicity

It defines how many instances of a class A can be associated with one instance of a class B.

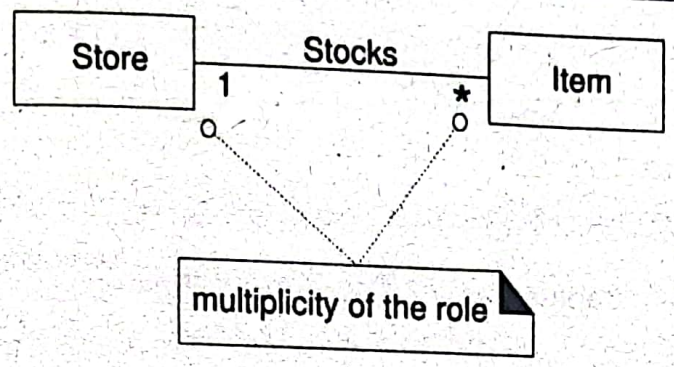


Fig. 2.4 : Multiplicity on an association.

For example, a single instance of a *Store* can be associated with "many" (zero or more, indicated by the *) *Item* instances.

Some examples of multiplicity expressions are shown as in figure below.

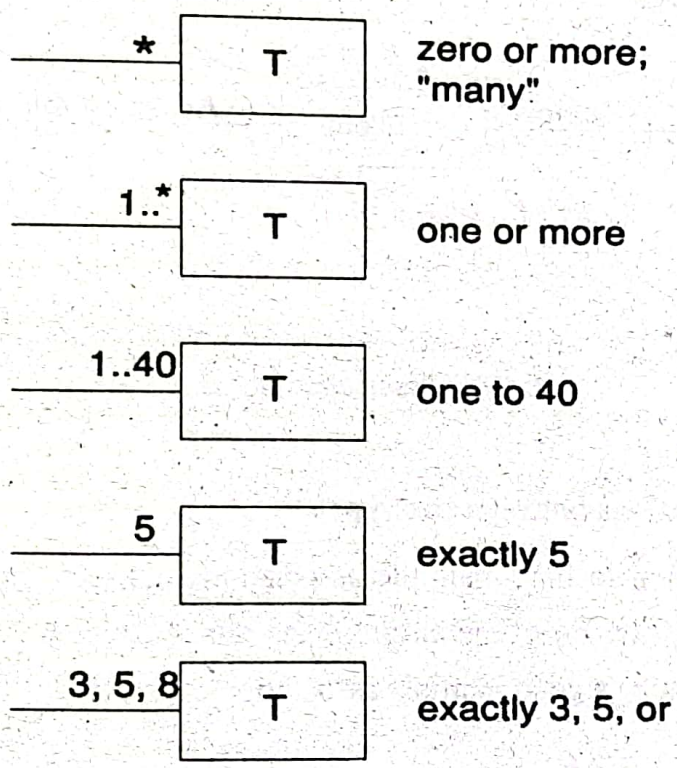


Fig. 2.5 : Multiplicity values.

Naming Associations

Name an association based on a format of TypeName-VerbPhrase-Typename format where the verb phrase creates a sequence that is readable and meaningful in the model context. Association names should start with a capital letter, since an association represents a classifier of links between instances.

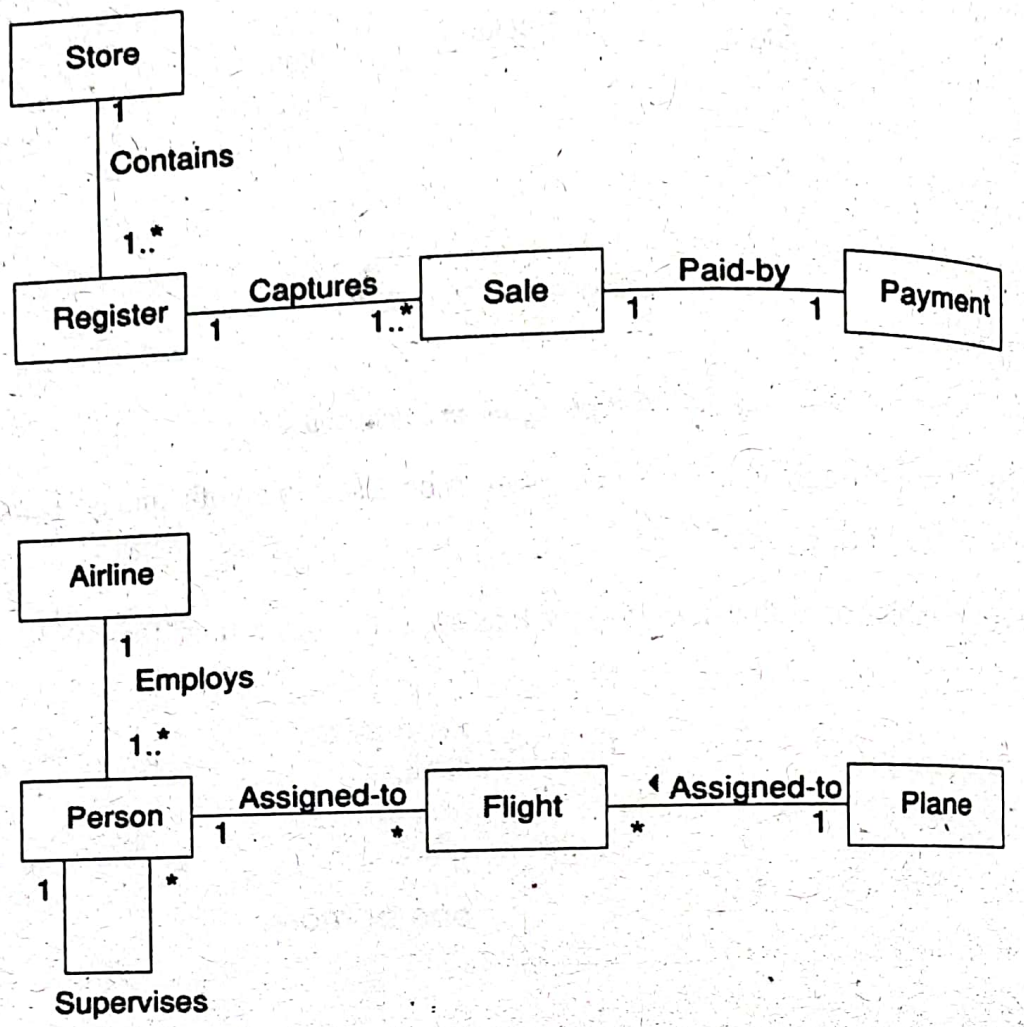


Fig. 2.6 : Association names.

Multiple Associations between two types

It is not uncommon to have multiple associations between the objects. An example from the domain of the airline is the relationships between a *Flight* and an *Airport* can be a good multiple association example.

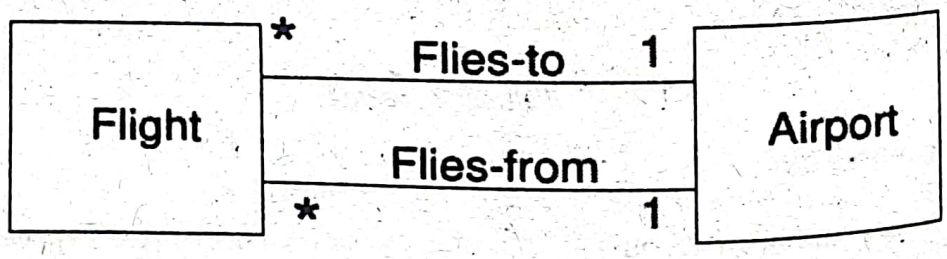
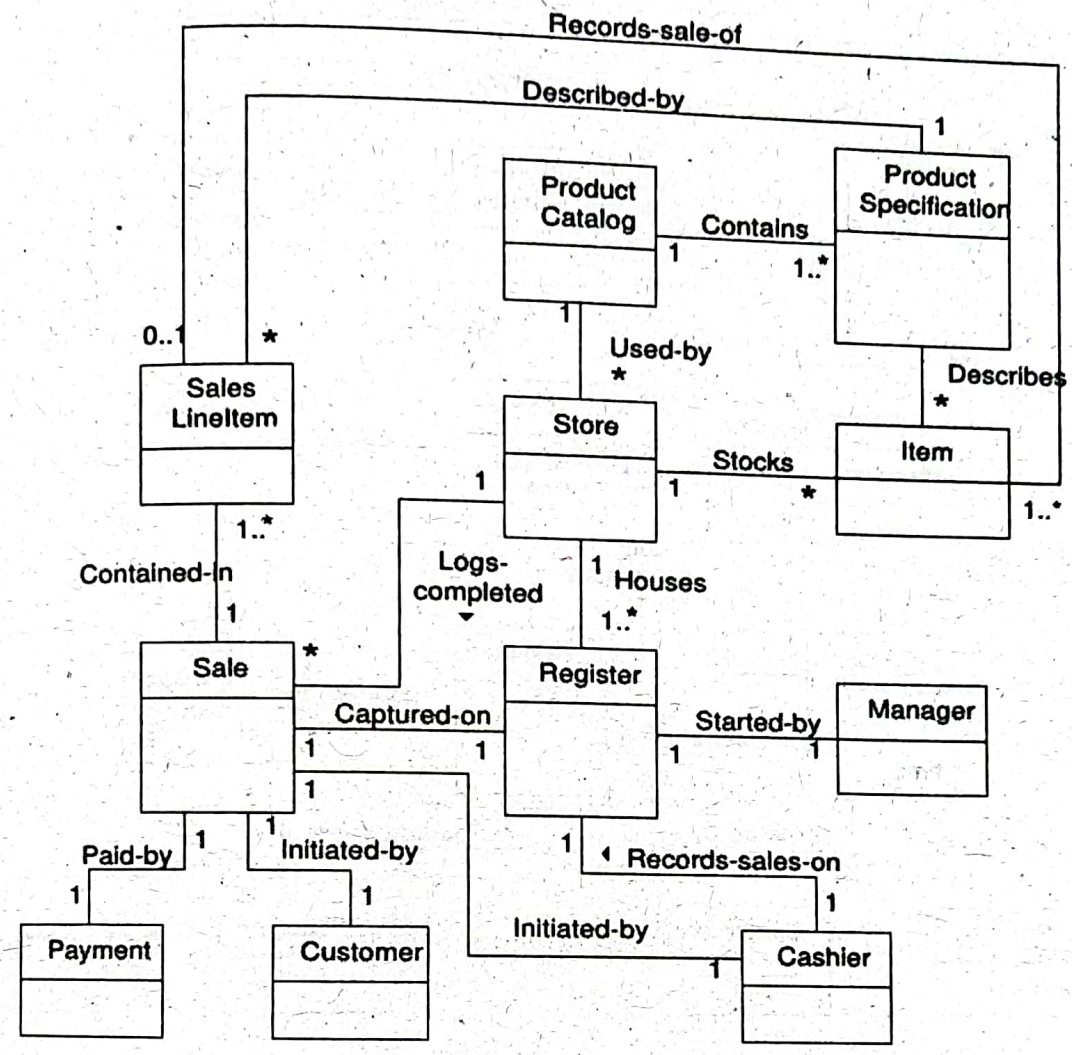


Fig. 2.7 : Multiple associations.

Example: Associations in the NextGen POS Domain Model



2. **Adding Attributes:** An attribute is a logical data value of an object. It is very useful to identify those attributes of conceptual classes that are needed to satisfy the information requirements of the current scenarios under development.

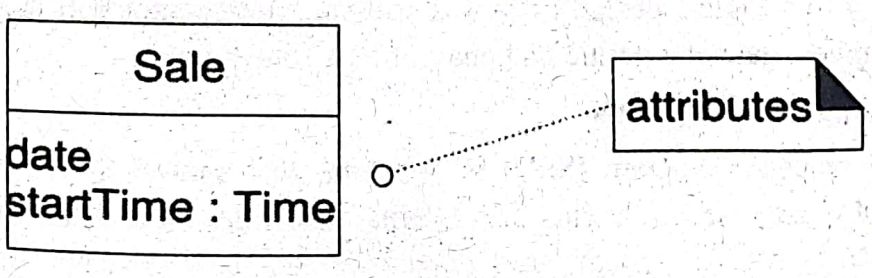
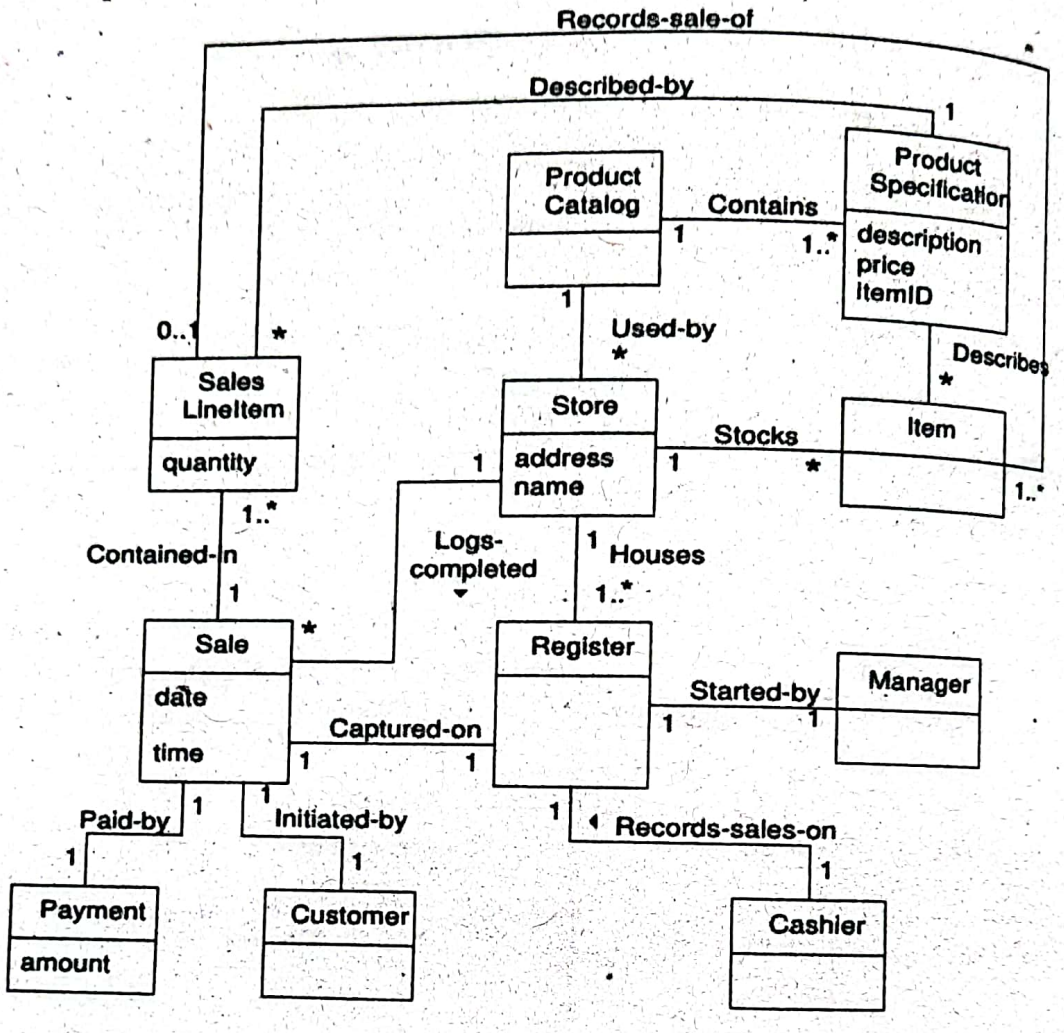


Fig. 2.8 : Class and attributes UML notation.

The full syntax for an attribute in the UML is :

visibility name : type multiplicity = default (property-string)

Example: Attributes in the NextGen POS Domain Model



2.3 Representation of System Behavior

System behavior may be understood as the description of what a system does, without need to explain how it does it. System behavior includes sequence diagrams, use cases and system operation contracts. It is needed before proceeding to a logical design of how a software application will work, it is useful to investigate and define its behavior as a "black box".

System Sequence Diagrams

A system sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and inter-system events. All systems are treated as a black box; the emphasis of the diagram is events that cross the system boundary from actors to systems.

An SSD should be done for the main success scenario of the use case, and frequent or complex alternative scenarios.

This example is for the main success scenario of the *Process Sale* use case. It indicates that the cashier generates *makeNewSale, enterItem, endSalem,* and *makePayment* system events.

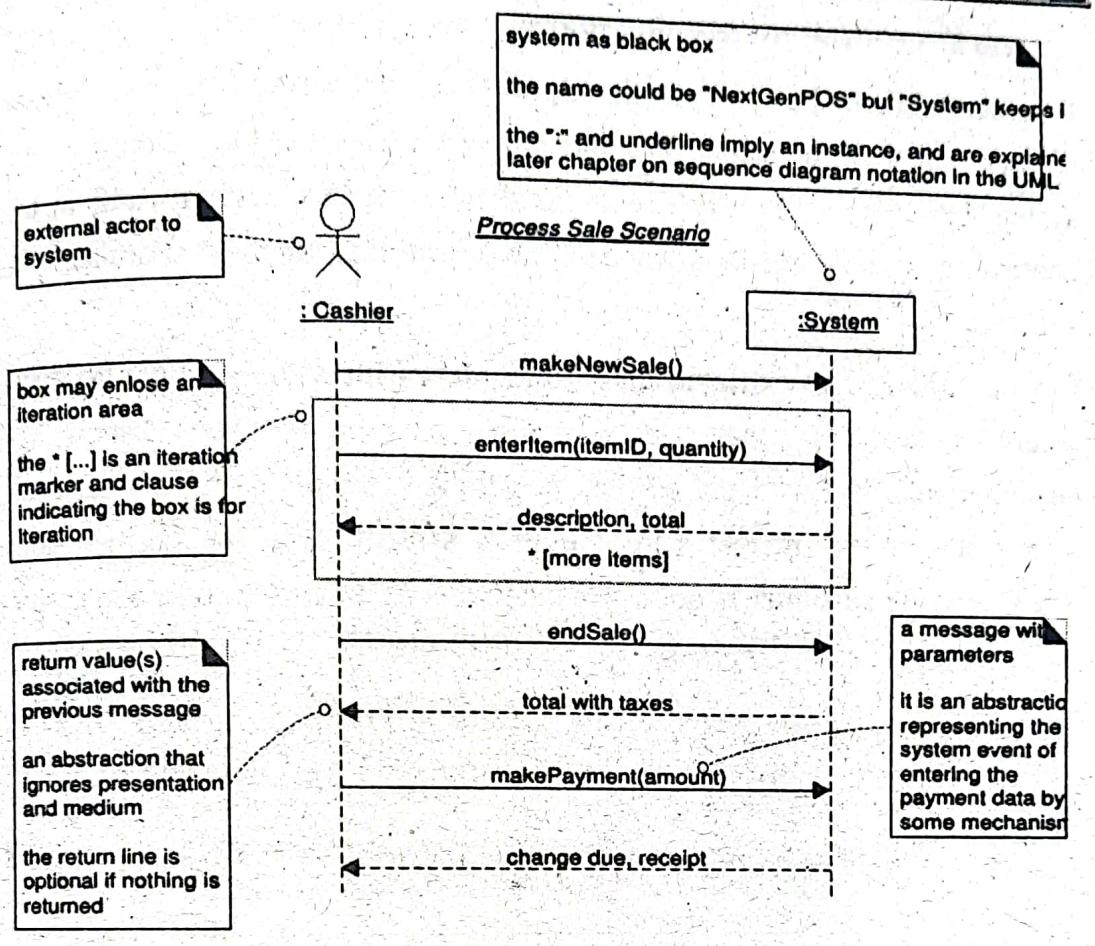


Fig. 2.9 : SSD for a Process Sale scenario.

SSD and Use Cases

An SSD shows system events for a scenario of a use case, therefore it is generated from inspection of a use case.

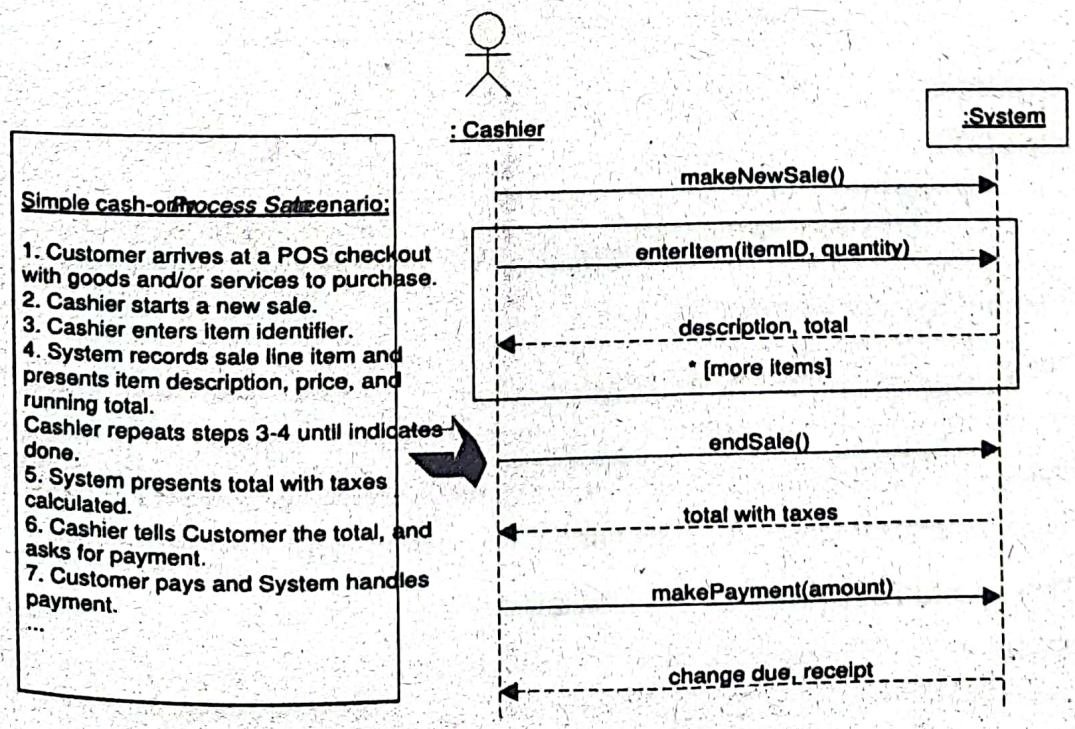


Fig. 2.10 : SSDs are derived from use cases.

System Events and System Boundary

It is necessary to be clear on the choice of a system boundary in order to identify the system events. For the purpose of software development, the system boundary is usually chosen to be the software system itself; in this context, a system event is an external event that directly stimulates the software.

Consider the *Process Sale* use case to identify system events. First, we must determine the actors that directly interact with the software system. The customer interacts with the cashier, but for this simple cash only scenario, does not directly interact with the POS system, only the cashier does. Therefore, the customer is not a generator of system events; only the cashier is.

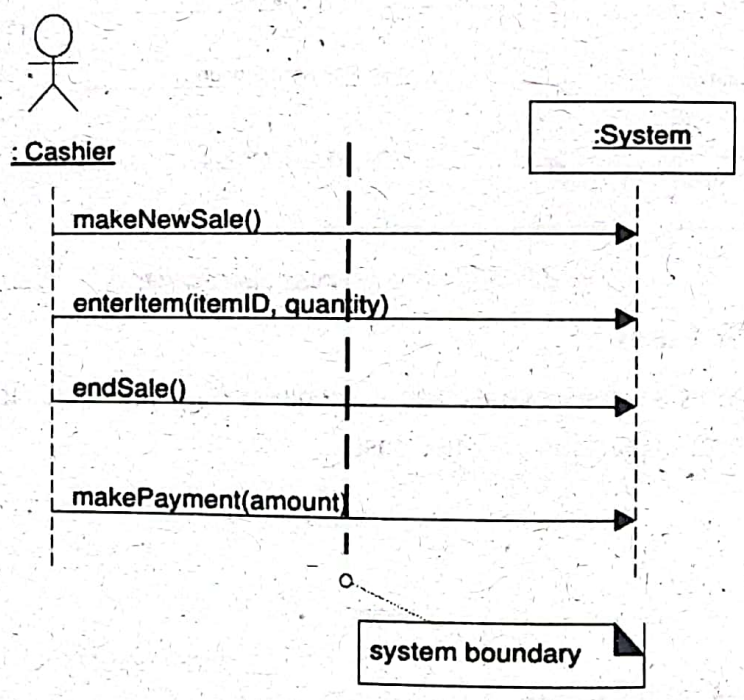


Fig. 2.11: Defining the system boundary

Naming System Events and Operations

System events should be expressed at the level of intent rather than in terms of the physical input medium or interface widget level. It also improves the clarity to start the name of a system event with a verb (add..., enter..., end..., make...) since it emphasizes the command orientation of these events.

Thus, "enterItem" is better than "scan" because it captures the intent of the operation while remaining abstract with respect to the design choices and about the interface used to capture the information.

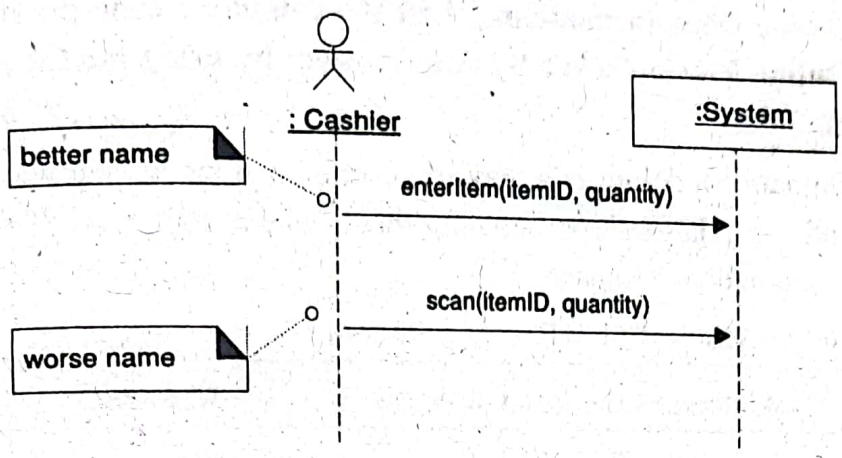


Fig. 2.12 : Naming the system events and operations

Operation Contracts

Operation contract describes the outcome of executing system operation in terms of state change to domain objects. It gives detailed representation of system behavior. It is a document containing:

1. Operation
2. Cross reference
3. Pre conditions
4. Post conditions

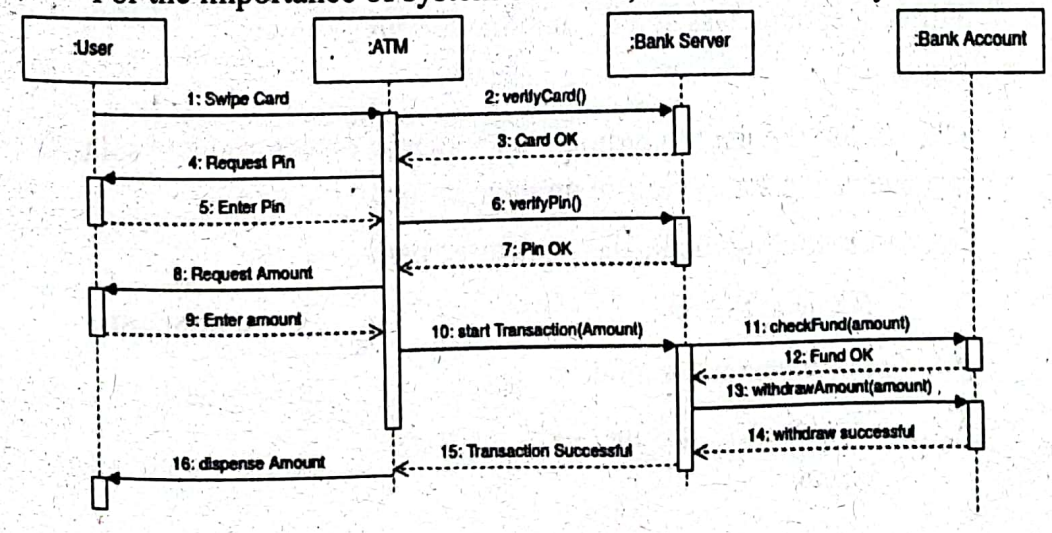
Old Exam Question-Answer

1. List out the possible ways to find out concept name and relationships of domain model and construct domain model of Point of Sale Terminal of retail store that should include the concept of user management, inventory management, sale and payment. [2079 Jestha]

⇒ Refer to the theory section.

2. Describe the importance of system behavior. Provide UML notation of sequence diagram and draw sequence diagram for cash withdrawal from Automated Teller Machine with proper guard condition. [2079 Jestha]

⇒ For the importance of system behavior, refer to the theory section.



3. Define Domain modeling. List the candidate conceptual class for Online Movie Ticket Booking System by using the Category List method. **[2078 Chaitra]**

⇒ Domain Modeling is a way to describe and model real world entities and the relationships between them, which collectively describe the problem domain space.

Online Movie Ticket Booking System

Conceptual Class Category	Examples
Physical or tangible objects	Counter
Places	Hall
Transactions	Booking Payment Reservation
Roles of people	Cashier
Things in a container	Viewer
Abstract noun concepts	Movie
Events	Booking Payment Viewing Show

4. Define conceptual class and domain model. Explain primary relationship between class, dependency, association and composition with their corresponding notation.

[2075 Baisakh, 2070 Bhadra]

⇒ For conceptual class and domain model refer above.

Class:

Classes are the main building blocks of any object oriented solution. It represents the static view of an application.

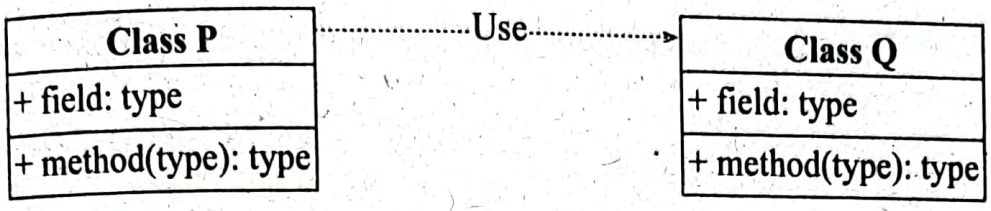
In most modeling tools, class has three parts,

- Name at the top,
- Attributes in the middle,
- Operations or methods at the bottom

Classname
+ field: type
+ method(type): type

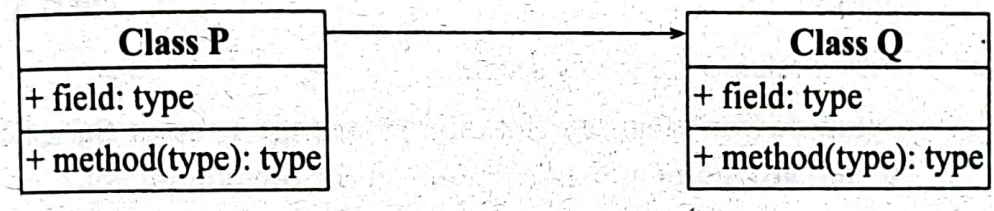
Dependency:

A dependency is a relationship in which one element e.g. client, uses or depends on another element e.g. supplier. So, changes made in one element may affect the element which is dependent on the other.



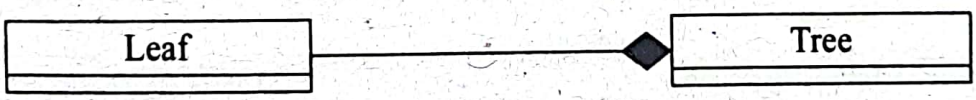
Association:

An association relationship is a structural relationship between two model elements that shows that objects of one class connect and can navigate to objects for another class.



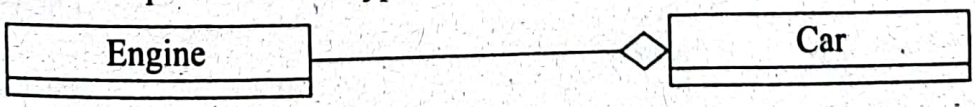
Composition:

It is a type of association relationship where a child cannot exist independently of the parent. It is described as "part of" type of relationship. It is a strong type of association.



Aggregation:

It is a type of association relationship where a child can exist independently of the parent class. It is described as "has a" type of relationship. It is a weak type of association.



5. **What are the perceptions of applying UML? Consider the Library System. Each book in a library contains bibliography, each bibliography consists of a number of references to other books. A book will be referred to in many cases and therefore a reference can appear in more than one bibliography. Use noun phrase identification strategies to find the conceptual classes of above cases.** **[2074 Bhadra]**

⇒ There are three perceptions of applying UML: Conceptual perceptions, Specification perceptions, Implementation perceptions.

Library System:

- Each Book in a Library contains bibliography.
- Each bibliography consists of a number of references to other books.

Conceptual classes are: Book, Library, Bibliography, References

Relations:

Library (1…*) Book

Book (1…1) Bibliography

Bibliography (1…*) Reference

6. **Define the strategies to identify conceptual classes. Describe the steps to create a Domain Model used for representing conceptual classes.** **[2073 Bhadra]**

⇒ Please refer to the theory section.

7. **What do you mean by Domain Modeling? Present the guidelines to add attributes and associations in the domain model.**

[2072 Ashwin]

⇒ Please refer to the theory section.

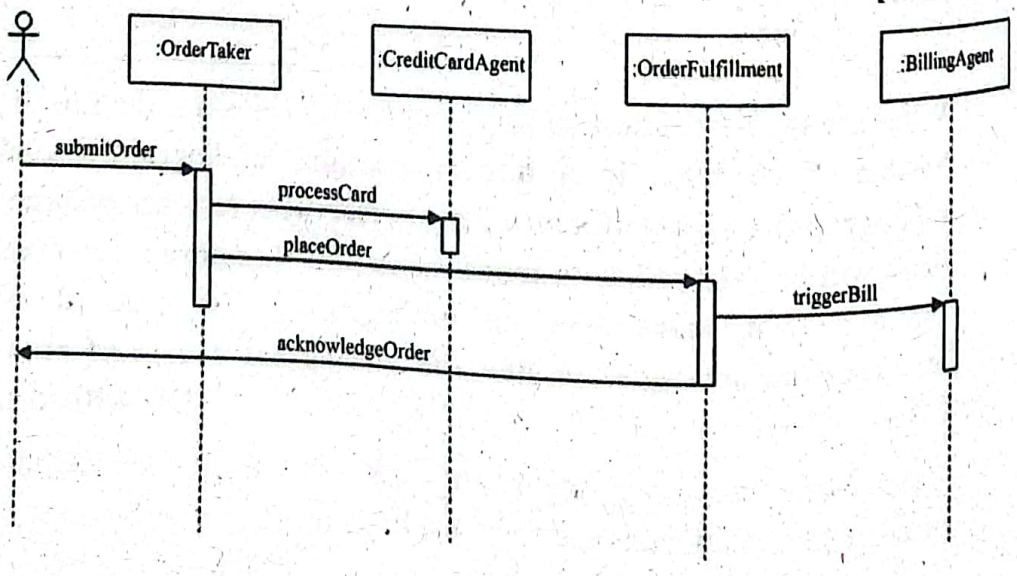
8. **How can you represent the dynamic behavior of the system in Object Oriented Analysis (OOA)? Explain with an example.**

[2072 Ashwin]

⇒ Please refer to the theory section.

9. **The model depicts the online order processing system as illustrated in figure below. Explain in detail of the diagram type with every symbol being used and semantics of the diagram.**

[2070 Magh]



Actor: A lifeline notation with an actor element symbol is used when the particular sequence diagram is owned by a use case.

Here, the Actor submits the order to the ordertaker and all the processing is done afterwards.

Object Instances: As shown in figure above, Order Taker, Credit Card Agent, Order Fulfillment, Billing Agent are all the instances of their classes. These are denoted with the lifeline notations. The lifeline is the dotted vertical line under the instances.

Activation Bars: The activation bar is the box placed on the lifeline. It is used to indicate that an object is active (or instantiated) during an interaction between two objects. The length of the rectangle indicates the duration of the objects staying active.

In a sequence diagram, an interaction between two objects occurs when one object sends a message to another. The use of the activation bar on the lifelines of the Message caller and the Message receiver indicates that both are active during the exchange of the message.

Message Arrows: An arrow from the Message caller to the receiver specifies a message in a sequence diagram. A message can flow in any direction, from left to right, right to left or back to the message caller itself. While you can describe the message being sent from one object to another on the arrow, with different arrowheads you can indicate the type of message being sent or received.

A message has an attribute to firmly denote the description of the message and is written along the message line as shown in the figure above. All above messages are synchronous messages.

Practice Questions

1. What is a Domain Model?
2. How is the domain model illustrated?
3. Why Call a Domain Model a "Visual Dictionary"?
4. What are the elements not suitable in a domain model?
5. What are Conceptual Classes?
6. How to Create a Domain Model?
7. How to Find Conceptual Classes?

8. Why Should We Avoid Adding Many Associations?
9. What is Aggregation?
10. What is Composition?
11. What do you mean by System Sequence Diagrams?
12. Define System Events and the System Boundary.
13. How to Naming System Events and Operations?
14. What is meant by Low Coupling and High cohesion?

❑❑❑

Chapter 3

Object Oriented Design

3.1 Object Oriented Design

An object consists of data and methods that are grouped together to represent an entity. The object-interface defines how an object can interact with other objects. **Object-oriented design (OOD)** is the process of planning a system of interacting objects for the purpose of solving a software problem. It is one approach to software design. It follows the defining objects and their interactions as documented during object-oriented analysis. In other words, it is a method of design encompassing the process of object-oriented decomposition.

Object-oriented design is the discipline of defining the objects and their interactions to solve a problem that was identified and documented during object-oriented analysis.

Object-oriented design is the process of planning a system of interacting objects for the purpose of solving a software problem. It is one approach to software design

3.1.1 Input for OOD (Object-Oriented Design)

The input for object-oriented design is provided by the output of object-oriented analysis. Realize that an output artifact does not need to be completely developed to serve as input of object-oriented design; analysis and design may occur in parallel, and in practice the results of one activity can feed the other in a short feedback cycle through an iterative process. Both analysis and design can be performed incrementally.

Some typical input artifacts for OOD are:

- **Conceptual model:** The result of object-oriented analysis, it captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage.
- **Use case:** A description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function. Use case actors may be end users or other systems. In many circumstances use cases are further elaborated into use case diagrams. Use case diagrams are used to identify the actor (users or other systems) and the processes they perform.

- **System sequence diagram:** A system sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.
- **User interface documentations (if applicable):** A Document that shows and describes the look and feel of the end product's user interface. It is not mandatory to have this, but it helps to visualize the end-product and therefore helps the designer.
- **Relational data model (if applicable):** A data model is an abstract model that describes how data is represented and used. If an object database is not used, the relational data model should usually be created before the design, since the strategy chosen for object-relational mapping is an output of the OO design process. However, it is possible to develop the relational data model and the object-oriented design artifacts in parallel, and the growth of an artifact can stimulate the refinement of other artifacts.

3.1.2 Object-Oriented concepts:

The object oriented design is composed of five basic concepts which are the implementation level features that are built into the programming language.

- ***Object/Class*:** An association of data structures with the methods or functions that act on the data is called a class, or object (an object is created based on a class). Each object serves a separate function. It is defined by its properties, what it is and what it can do. An object can be part of a class, which is a set of objects that are similar.
- **Information hiding:** The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as private or protected to the owning class.
- **Inheritance:** The ability for a class to extend or override functionality of another class. The so-called subclass has a whole section that is derived (inherited) from the super-class and then it has its own set of functions and data.
- **Interface:** The ability to defer the implementation of a method. The ability to define the functions or methods signatures without implementing them.
- **Polymorphism:** The ability to replace an object with its sub-objects.

3.1.3 Designing Concepts

- Defining objects, *class diagrams,*
- Identifying attributes.
- Using design patterns(*if applicable*): A design pattern is not a finished product but a description of a solution to a common problem. It is advantageous since it can be used in multiple applications. It can also be thought of as a template to solve problems in different situations.
- Identify objects/data(if applicable): Identify objects that have to last longer than a single runtime of an application. If a *relational-database* is used then the design is object relational mapping.
- Identify and define remote objects (if applicable).

3.1.4 Output obtained from OOD

- ***Sequence Diagram:*** A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur.
- ***Class Diagram:*** A class diagram is a type of static structure UML diagram that describes the structure of a system by showing the system's classes, its attributes, and the relationships between the classes. The messages and classes identified through the development of the sequence diagrams can serve as input to the automatic generation of the global class diagram of the system.

3.1.5 How do developers design objects?

Developers design objects in three ways:

1. **Code:** Design while code in Java, C++, C#. From mental model to code.
2. **Draw, then code: Drawing some UML then switching to no. 1 with a strong IDE.**
3. **Only Draw:** This tool generates everything from the diagrams. This might still require a programming language attached to UML graphic elements.

3.1.5.1 Designing Objects

What are Static and Dynamic Modeling?

In the system analysis or object-oriented analysis phase of software development, the system requirements are determined, the classes are identified and the relationships among classes are identified.

The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modeling, dynamic modeling, and functional modeling.

Object Modeling (Static Modeling)

Object modeling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.

The process of object modeling can be illustrated in the following steps:

- Identify objects and group into classes
- Identify the relationships among classes
- Create user object model diagram
- Define user object attributes
- Define the operations that should be performed on the classes
- Review glossary

Dynamic Modeling

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modeling.

Dynamic Modelling can be defined as "a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world".

The process of dynamic modeling can be visualized in the following steps –

- Identify states of each object
- Identify events and analyze the applicability of actions
- Construct dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

3.1.5.2 Object Design Techniques

CRC Cards(Class-Responsibility-Collaboration card):

Class-Responsibility-Collaboration cards are brainstorming tool used in the design of object-oriented software. They are popular among expert designers and recommended by adept programmers. They may be a great deal of capturing interactions and relationships between the objects.

CRC cards are usually created from index cards. Members of a brainstorming session will write one CRC card for each relevant class/object of their design. The card is partitioned into three areas:

1. On the top of the card , class name
2. On the left , the responsibilities of the class
3. On the right , the collaborators (other classes) with which the class interacts to fulfill its responsibilities.

Student	
Student number	Seminar
Name	
Address	
Phone number	
Enroll in a seminar	
Drop a seminar	
Request transcripts	

Responsibility

A Responsibility is anything that the class knows or does. These responsibilities are things that the class has knowledge about itself, or things the class can do with the knowledge it has.

For example, a person class might have knowledge (and responsibility) for its name, address, and phone number. In another example an automobile class might have knowledge of its size, its number of doors, or it might be able to do things like stop and go. The Responsibilities of a class appear along the left side of the CRC card.

Collaborator

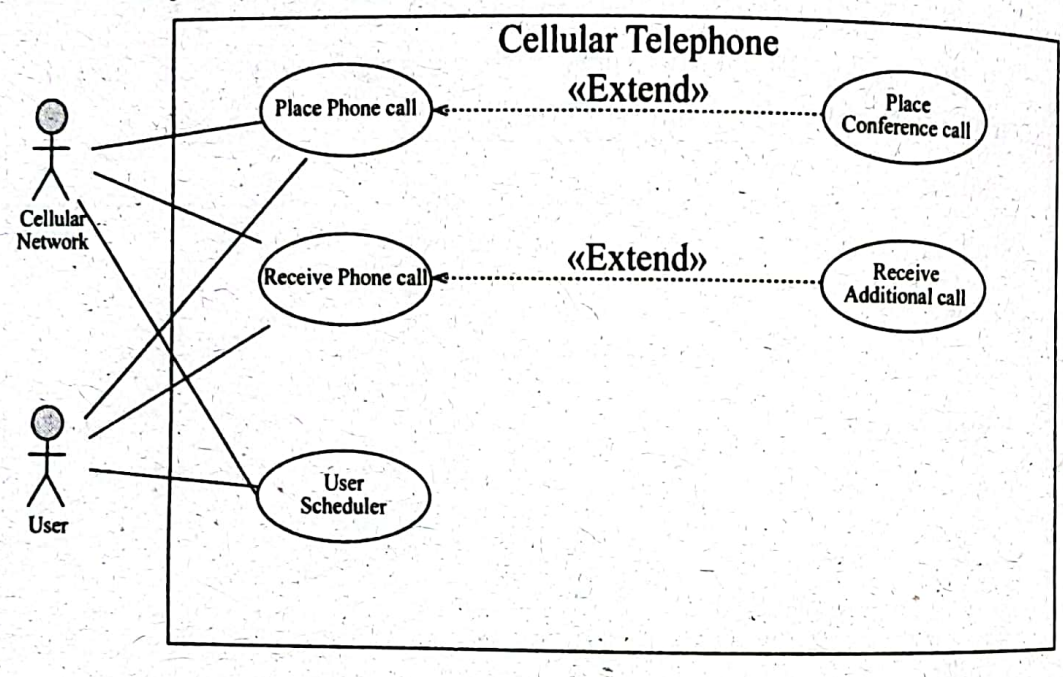
A Collaborator is another class that is used to get information for, or perform actions for the class at hand. It often works with a particular class to complete a step (or steps) in a scenario. The Collaborators of a class appear along the right side of the CRC card.

3.2 Use case and Use case Diagrams

A use case describes a certain piece of desired functionality of an application system.

It is constructed during the analysis stage. It shows the interaction between an actor, which could be a human or a piece of software or hardware and the system. It does not specify how the system carries out the task.

You apply use case diagrams to model the use case view of a system .Use case diagrams are important for visualizing, specifying, and documenting the behavior of an element. They make systems, subsystems, and classes approachable and understandable by presenting an outside view of how those elements may be used in context.



3.2.1 Terms and Concepts

A use case diagram is a diagram that shows a set of use cases and actors and their relationships.

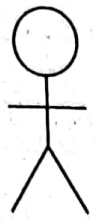
Use case diagrams commonly contain

- Use cases
- Actors
- Dependency, generalization, and association relationships

Use cases: It represents the functionality of the system. It is represented by an oval figure along with the functionality.

Actors: Actors are the entities which lie outside of the system and interact with it.

They are represented by stick-man.

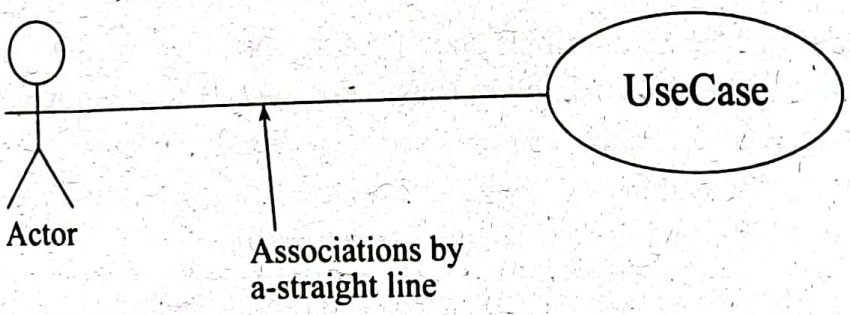


Actor

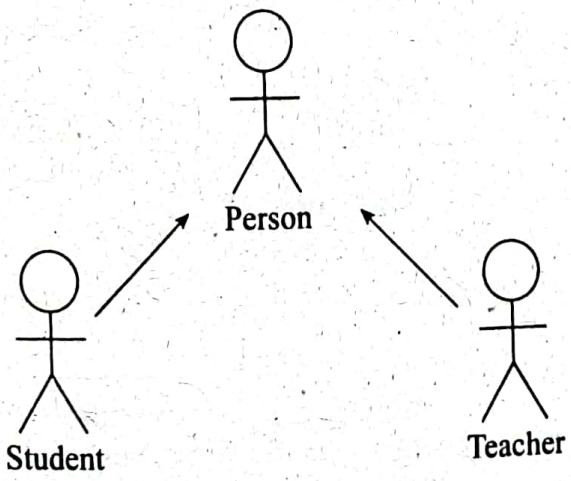
3.2.2 Relationships between use Cases

In use case there are mainly three types of the relationships between the use cases and as well as between the actors.

1. **Association:** This one is straightforward and present in every use case diagram. Few things to note.
   - An actor must be associated with at least one use case.
   - An actor can be associated with multiple use cases.
   - Multiple actors can be associated with a single use case.



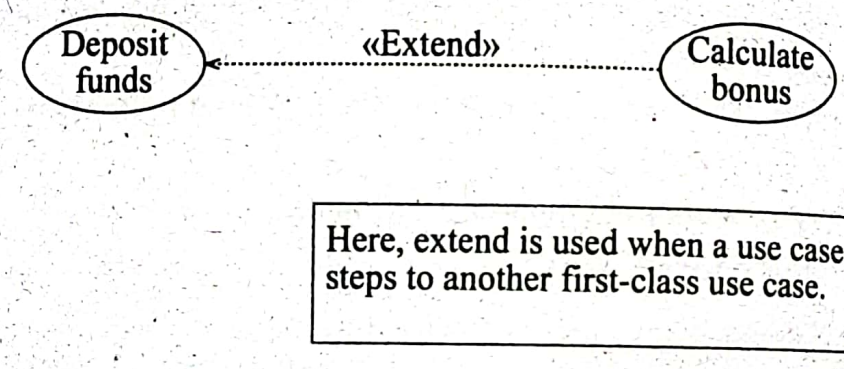
2. **Generalization:** Generalization of an actor means that one actor can inherit the role of the other actor. The descendant inherits all the use cases of the ancestor. The descendant has one or more use cases that are specific to that role.



A teacher and a student can be generalized to person

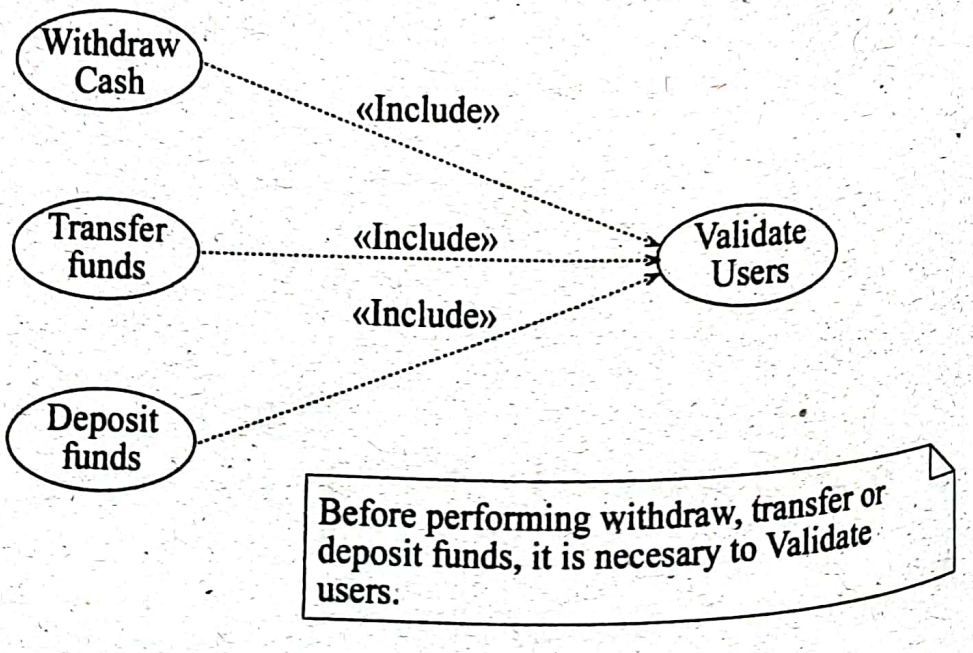
3. **Extends:** As the name implies it extends the base use case and adds more functionality to the system. Here are a few things to consider when using the **<<extend>>** relationship.

- The extending use case is dependent on the extended (base) use case.
- The extending use case is usually optional and can be triggered conditionally.
- The extended (base) use case must be meaningful on its own. This means it should be independent and must not rely on the behavior of the extending use case.



4. **Include:** Include relationships show that the behavior of the included use case is part of the including (base) use case. The main reason for this is to reuse common actions across multiple use cases. In some situations, this is done to simplify complex behaviors. Few things to consider when using the <<include>> relationship.

- The base use case is incomplete without the included use case.
- The included use case is mandatory and not optional.



You apply a use case to a system to model out the static use case view of the system. This mainly supports the behavior of a system: the outward visible services that the system provides to the users(Actors/ external entities).

The use case can be applied in one of the two ways:

1. To model the context of a system.
2. To model the requirements of the system.

3.2.3 Realization of Use case

A use-case realization represents how a use case will be implemented in terms of collaborating objects. It can include, for example, a textual description (a document), class diagrams of participating classes and subsystems, and interaction diagrams (communication and sequence diagrams) that illustrate the flow of interactions between class and subsystem instances.

In a model, a use-case realization is represented as a UML collaboration that groups the diagrams and other information (such as textual descriptions) that form part of the use-case realization.

UML diagrams that support use-case realizations can be produced in an analysis context, a design context, or both, depending on the needs of the project. For each use case in the use-case model, there can be a use-case realization in the analysis/design model with a realization relationship to the use case. In UML this is shown as a dashed arrow, with an arrowhead like a generalization relationship, indicating that a realization is a kind of inheritance, as well as a dependency.

3.2.3.1 Class Diagrams Owned by a Use-Case Realization

For each use-case realization there can be one or more class diagrams depicting its participating classes. A class and its objects often participate in several use-case realizations. It is important while designing to coordinate all the requirements on a class and its objects that different use-case realizations can have. The figure below shows an analysis class diagram for the realization of the Withdraw Cash Item use case.

3.2.3.2 Communication and Sequence Diagrams Owned by a Use-Case Realization

For each use-case realization there can be one or more interaction diagrams depicting its participating objects and their interactions. There are two types of interaction diagrams: sequence diagrams and communication diagrams.

They express similar information, but show it in different ways. Sequence diagrams show the explicit sequence of messages and are better when it is important to visualize the time ordering of messages, whereas communication diagrams show the communication links between objects and are better for understanding all of the effects on a given object and for algorithm design.

Realizing use cases through interaction diagrams helps to keep the design simple and cohesive. Assigning responsibilities to classes on the basis of what the use-case scenario explicitly requires encourages the design to contain the following:

- Only the functionality actually used in support of a use case scenario,
- Functionality that can be tested through an associated test case,
- Functionality that is more easily traceable to requirements and changes,
- Explicitly declared class dependencies that are easier to manage.

3.3 UML Interaction Diagrams

The term interaction diagram is a generalization of two more specialized UML diagram types:

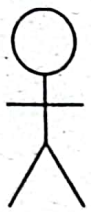
- Sequence diagrams
- communication diagrams

3.3.1 Sequence Diagrams

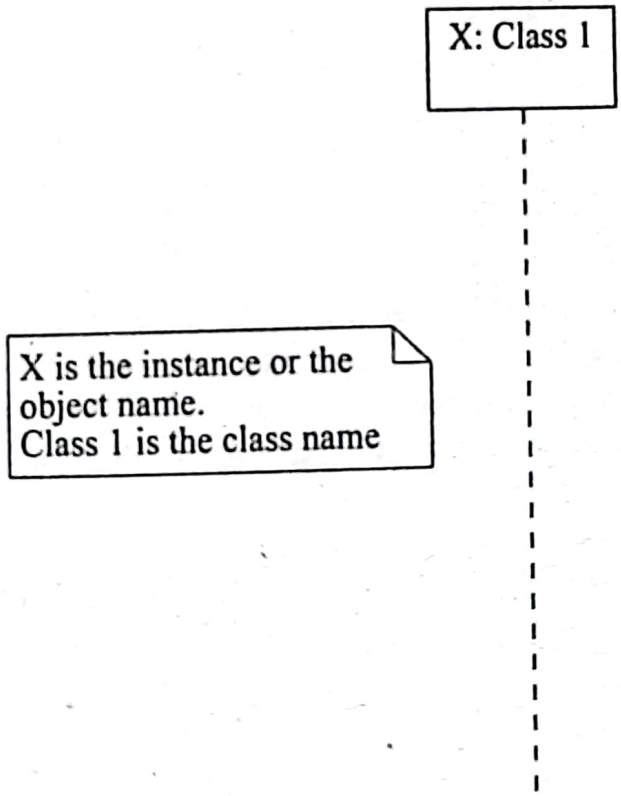
A sequence diagram is the most commonly used interaction diagram. Interaction diagram – An interaction diagram is used to show the interactive behavior of a system. Since visualizing the interactions in a system can be a cumbersome task, we use different types of interaction diagrams to capture various features and aspects of interaction in a system. Sequence Diagrams – A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place. We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

3.3.1.1 Terms and notations

1. **Actors:** An actor in a UML diagram represents a type of role where it interacts with the system and its objects. An actor is always outside the scope of the system we aim to model using the UML diagram. We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram.

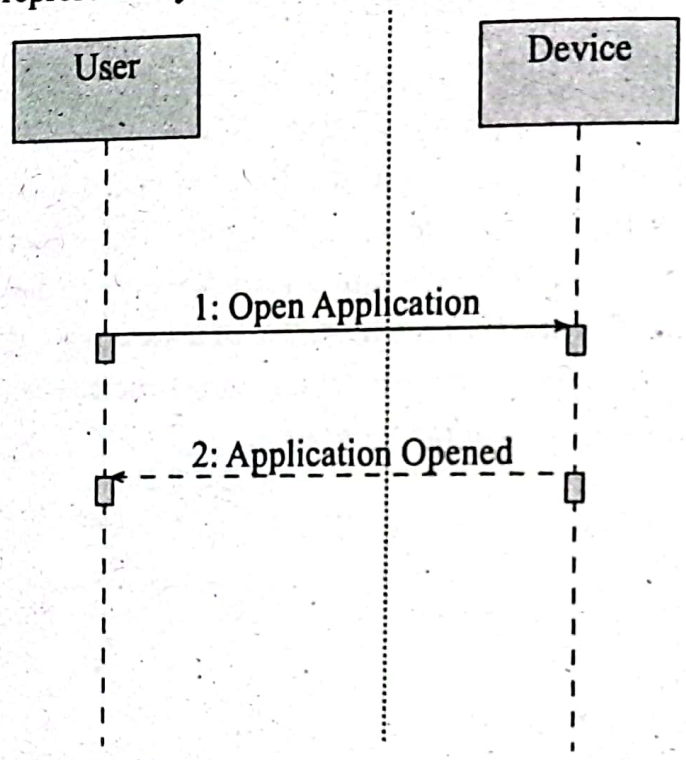


2. **Lifelines:** A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline. Lifeline elements are located at the top in a sequence diagram. The standard in UML for naming a lifeline follows the following format – Instance Name: Class Name. ***Difference between a lifeline and an actor*** – A lifeline always portrays an object internal to the system whereas actors are used to depict objects external to the system.

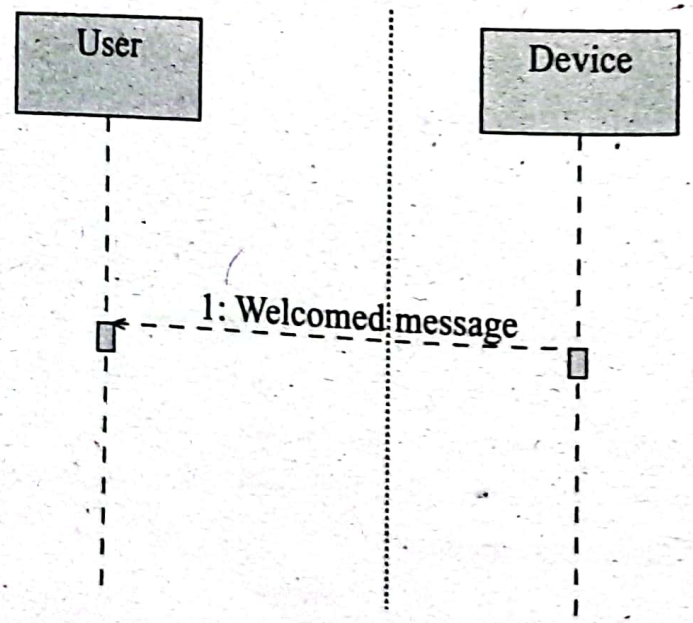


3. **Messages:** Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline. We represent messages using arrows. Lifelines and messages make the core of a sequence diagram. Messages can be broadly classified into the following categories:

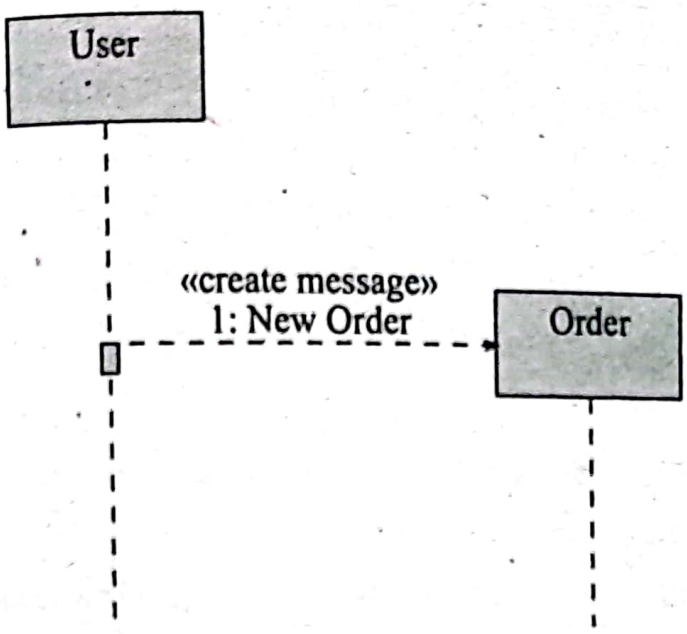
a. **Synchronous messages:** A synchronous message waits for a reply before the interaction can move forward. The sender waits until the receiver has completed the processing of the message. The caller continues only when it knows that the receiver has processed the previous message i.e. it receives a reply message. A large number of calls in object oriented programming are synchronous. We use a solid arrow head to represent a synchronous message.



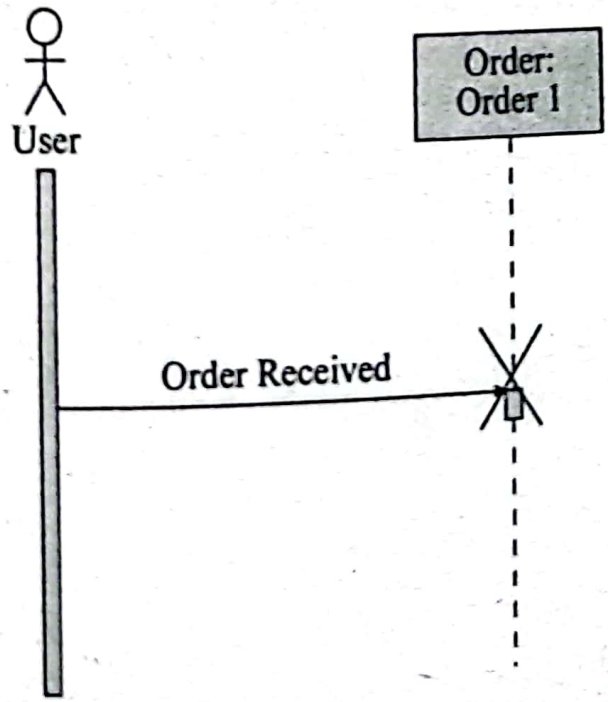
b. **Asynchronous messages:** An asynchronous message does not wait for a reply from the receiver. The interaction moves forward irrespective of the receiver processing the previous message or not. We use a lined arrow head to represent an asynchronous message.



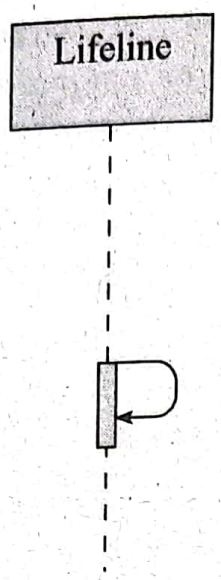
c. **Create message:** We use a Create message to instantiate a new object in the sequence diagram. There are situations when a particular message call requires the creation of an object. It is represented with a dotted arrow and a word labeled on it to specify that it is the created Message symbol. For example – The creation of a new order on an e-commerce website would require a new object of Order class to be created.



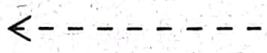
d. **Delete Message:** We use a Delete Message to delete an object. When an object is deallocated memory or is destroyed within the system we use the Delete Message symbol. It destroys the occurrence of the object in the system. It is represented by an arrow terminating with a x. **For example:** In the scenario below when the order is received by the user, the object of order class can be destroyed.



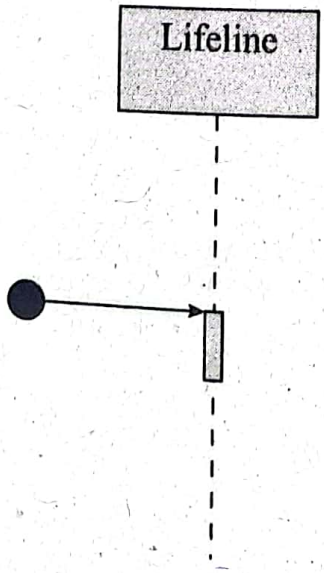
e. **Self Message:** Certain scenarios might arise where the object needs to send a message to itself. Such messages are called Self Messages and are represented



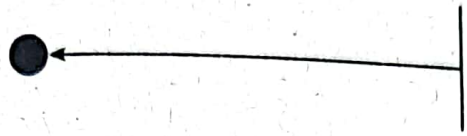
f. **Reply Message:** Reply messages are used to show the message being sent from the receiver to the sender. We represent a return/reply message using an open arrowhead with a dotted line. The interaction moves forward only when a reply message is sent by the receiver.



g. **Found Message:** A Found message is used to represent a scenario where an unknown source sends the message. It is represented using an arrow directed towards a lifeline from an endpoint.



h. **Lost Message:** A Lost message is used to represent a scenario where the recipient is not known to the system. It is represented using an arrow directed towards an end point from a lifeline.

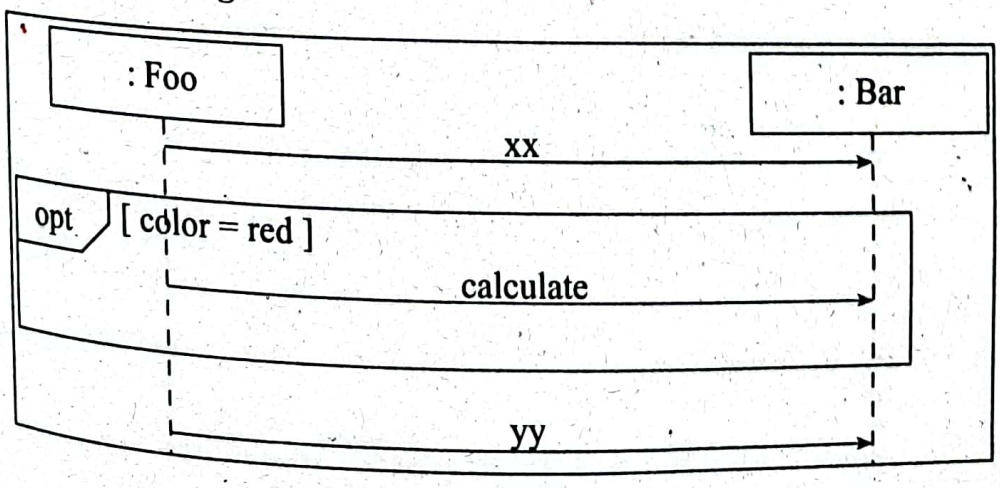


4. **Guards:** To model conditions we use guards in UML. They are used when we need to restrict the flow of messages on the pretext of a condition being met. Guards play an important role in letting software developers know the constraints attached to a system or a particular process.

The following table summarizes some of the common frame operators:

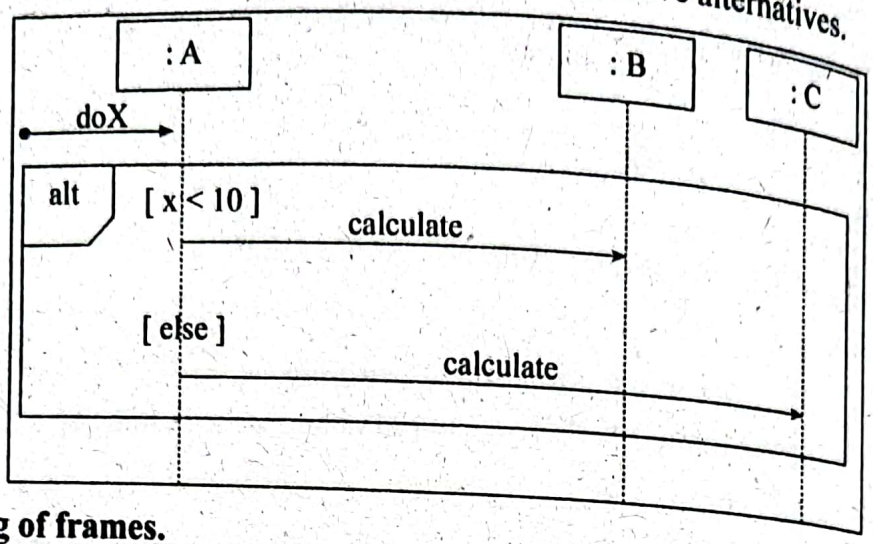
Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write loop(n) to indicate looping n times. There is discussion that the specification will be enhanced to define a FOR loop, such as loop(i, 1, 10)
opt	Optional fragment that executes if guard is true.
par	Parallel fragments that execute in parallel
region	Critical region within which only thread can run

Conditional messages

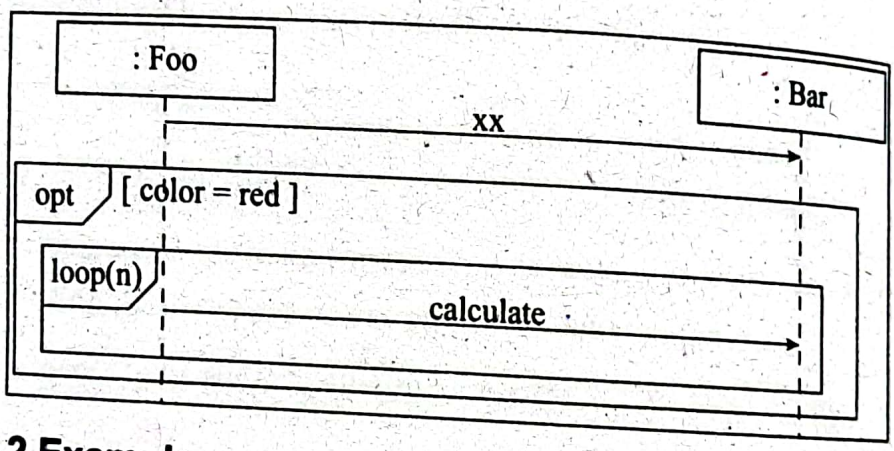


Mutually Exclusive Conditional Messages

An ALT frame is placed around the mutually exclusive alternatives.

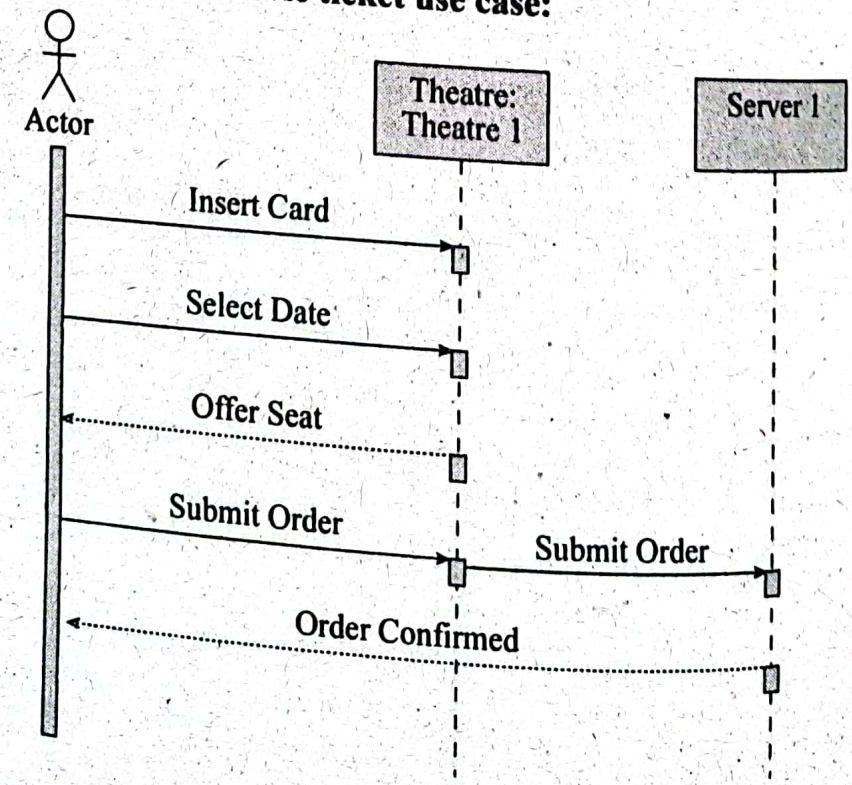


Nesting of frames.

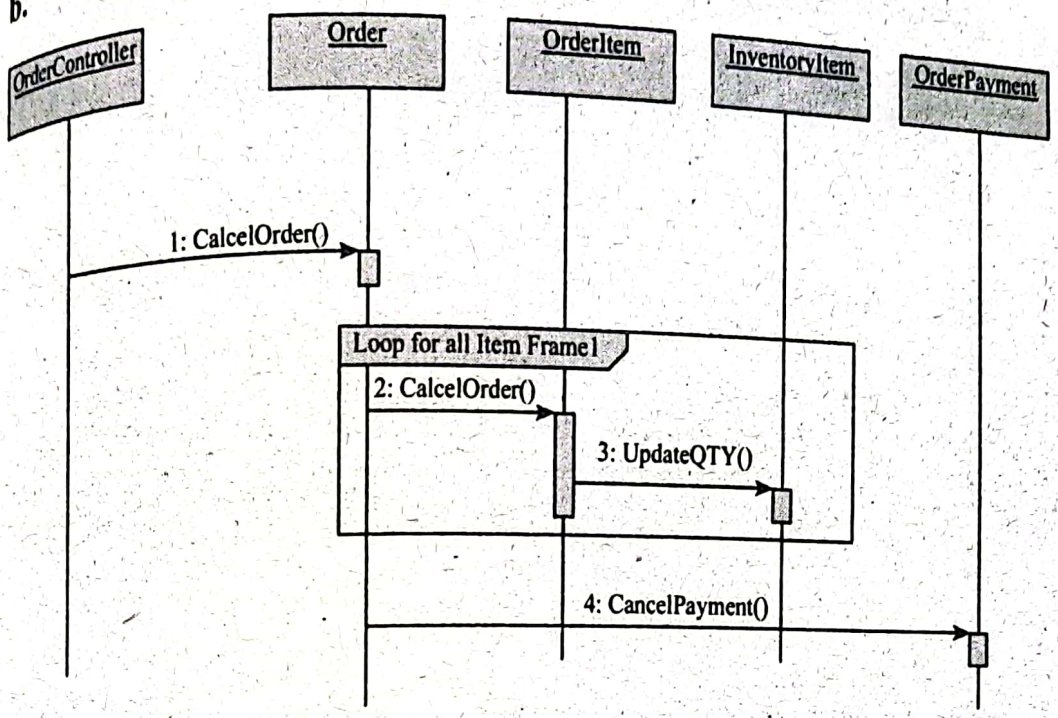


3.3.1.2 Example

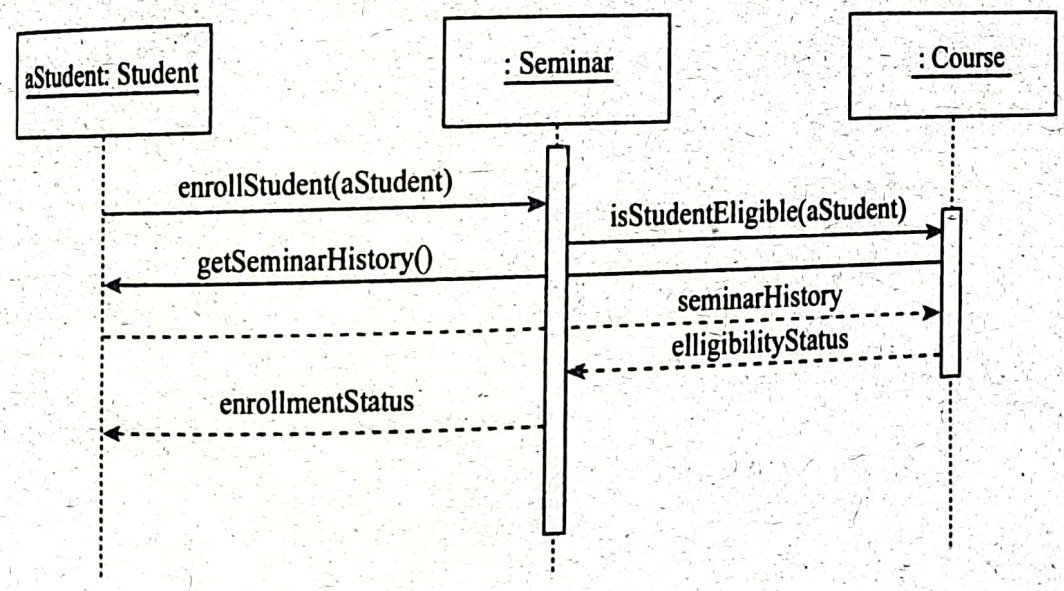
a. **Order online movie ticket use case:**



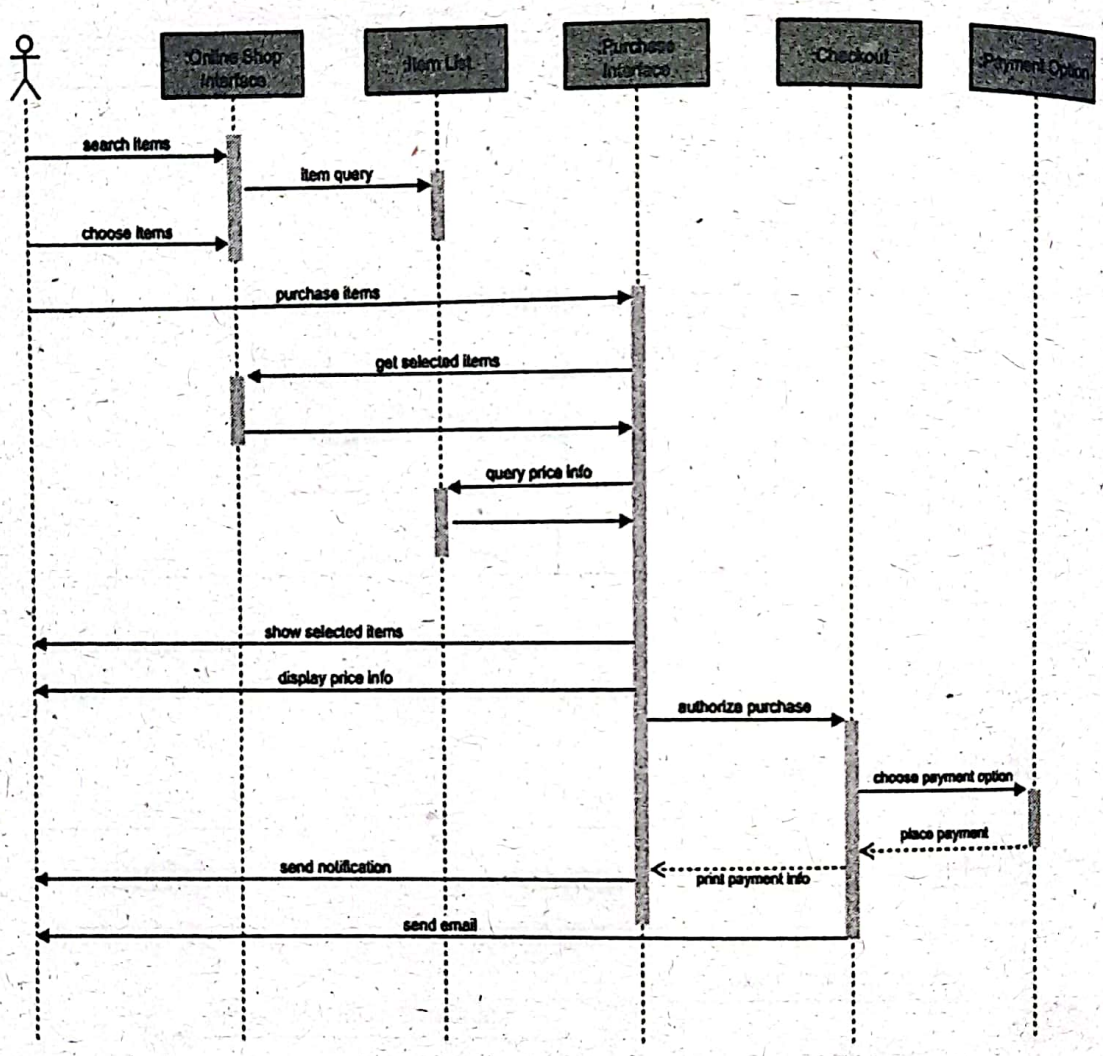
b. **Sequence Diagram for Cancel Order**



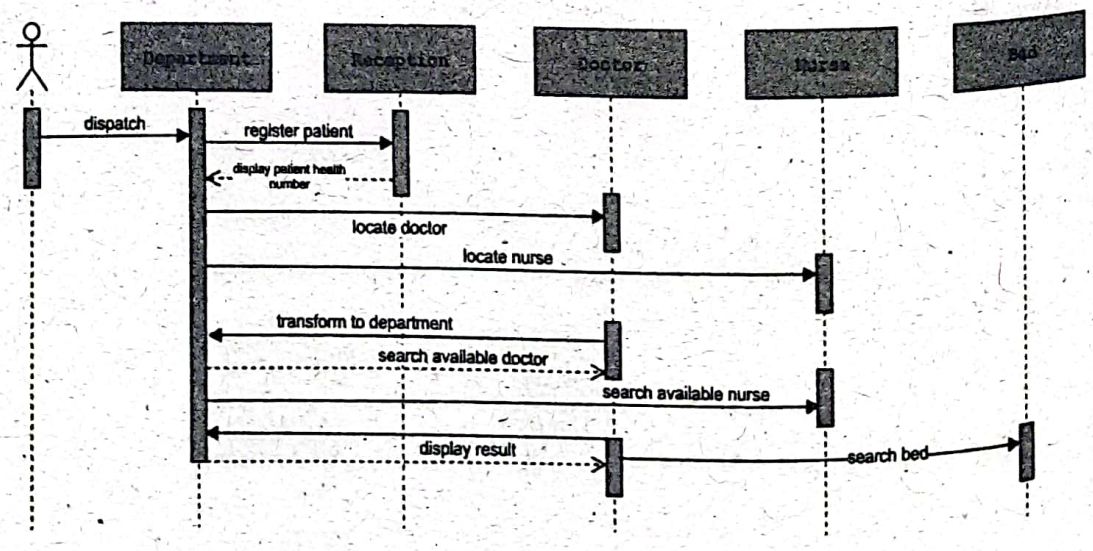
c. **Enroll in seminar use case**



d. Online shopping cart



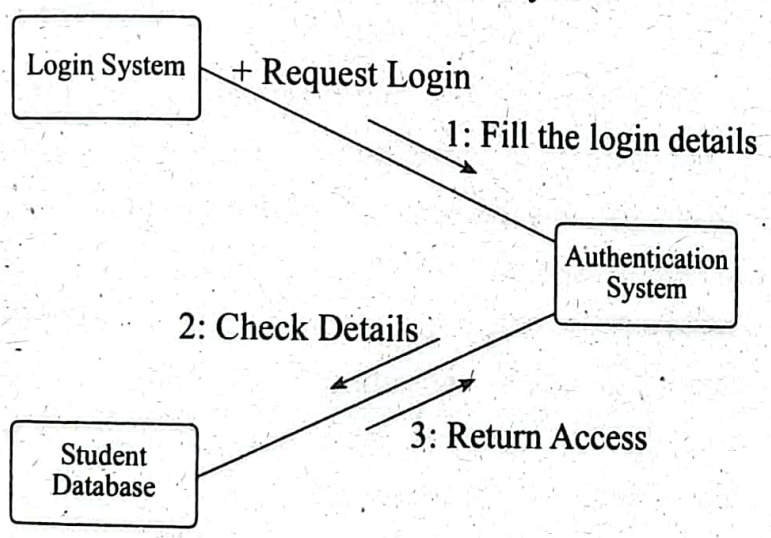
e. Hospital management system



3.4 Communication/Collaboration Diagram

A collaboration diagram shows both structural and behavioral aspects explicitly. This is unlike a sequence diagram which shows only the behavioral aspects. The structural aspect of a collaboration diagram consists of objects and the links existing between them. In this diagram, an object is also called a collaborator. The behavioral aspect is described by the set of messages exchanged among the different collaborators. The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labeled arrow placed near the link. Messages are prefixed with sequence numbers because they are only way to describe the relative sequencing of the messages in this diagram. The use of the collaboration diagrams in our development process would be help us to determine which classes are associated with the other classes.

Collaboration diagram for student management system



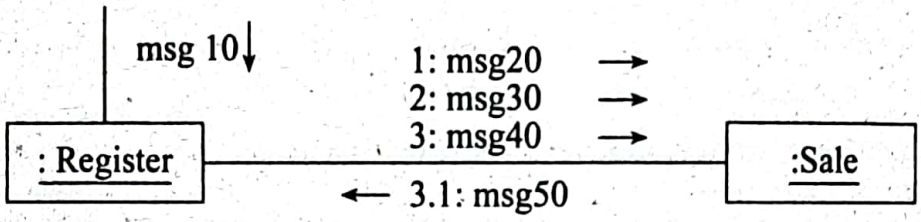
3.4.1 Comparison of Sequence and Collaboration Diagrams

Type	Strengths	Weakness
Sequence	• clearly shows the sequence or ordering of messages • large set of detailed notation options	• consumes horizontal space
Communication	• space can be effectively utilized	• more difficult to see the ordering of messages • fewer notation options

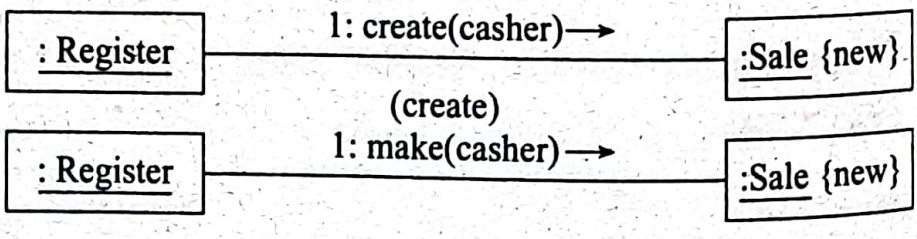
3.4.2 Basic Collaboration Diagram Notation

- **Links:** A link is a connection path between two objects; it indicates some form of navigation and visibility between the objects is possible. More formally, a link is an instance of an association.

- **Messages:** Each message between objects is represented with a message expression and small arrow indicating the direction of the message. Many messages may flow along this link. A sequence number is added to show the sequential order of messages in the current thread of control.



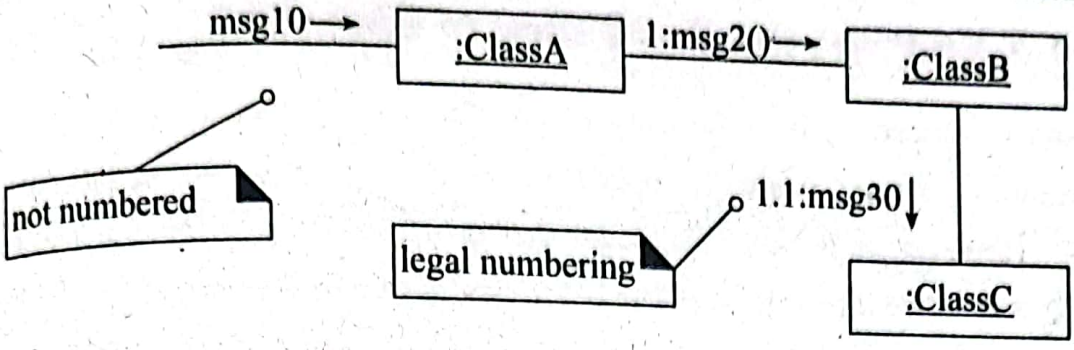
- **Creation of instances:** Any message can be used to create an instance, but there is a convention in the UML to use a message named create for this purpose. If another (perhaps less obvious) message name is used, the message may be annotated with a special feature called a UML stereotype, like so: «create». The create message may include parameters, indicating the passing of initial values. Furthermore, the UML property {new} may optionally be added to the instance box to highlight the creation.



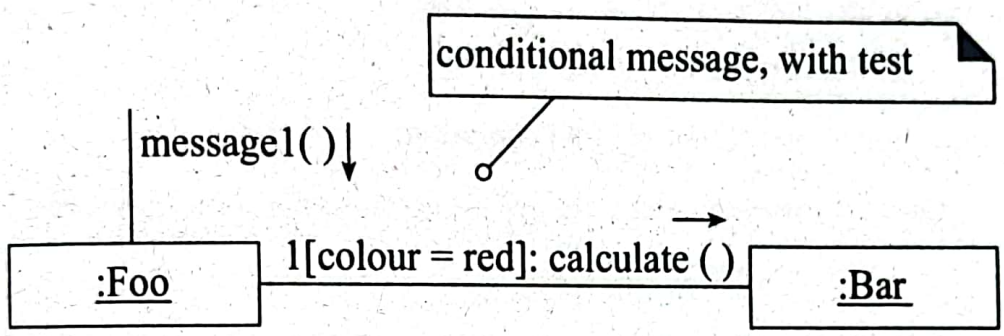
- **Message number sequencing:** The order of messages is illustrated with sequence numbers

  1. The first message is not numbered. Thus,msg1() is unnumbered.

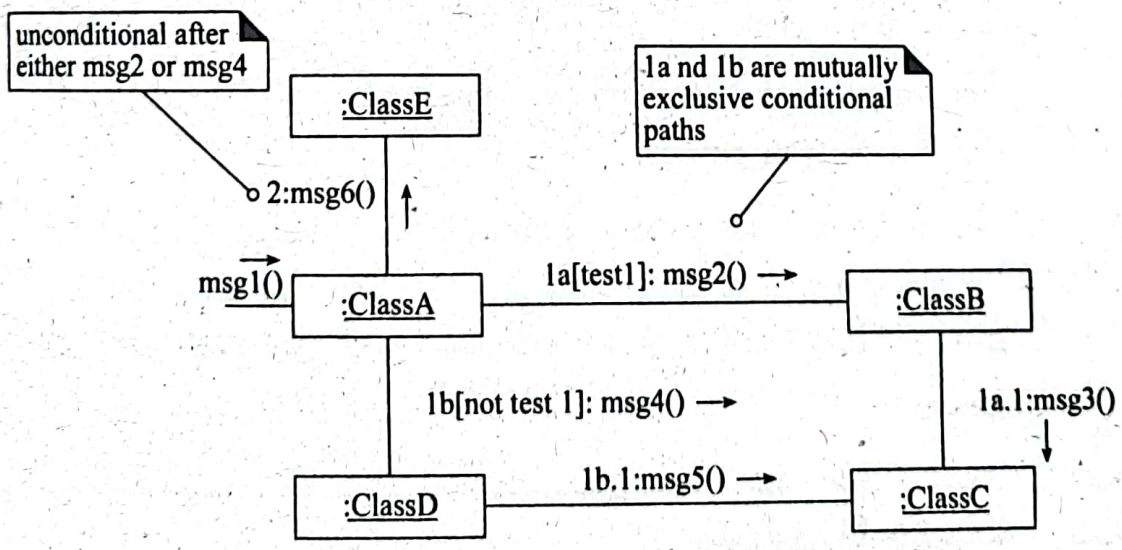
  2. The order and nesting of subsequent messages is shown with a legal numbering scheme in which nested messages have a number appended to them.



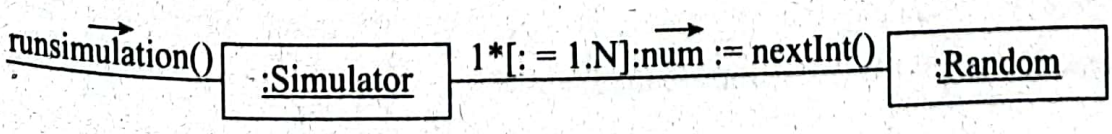
- **Conditional message:** A conditional message is shown by following a sequence number with a conditional clause in square brackets, similar to an iteration clause. The message is only sent if the clause evaluates to true.



- **Mutually exclusive conditional paths:**



- **Iterations or Looping:** Iteration notation is shown in Figure below. If the details of the iteration clause are not important to the modeler, a simple '*' can be used.



3.4.3 Modeling the Realization of Operation

Another purpose of the collaboration diagrams is to model the purpose of operations. In many cases, the operation can be realized by diving into the code. However, it is better to model the operations in collaboration for a better understanding of the scenario before you dive into the code.

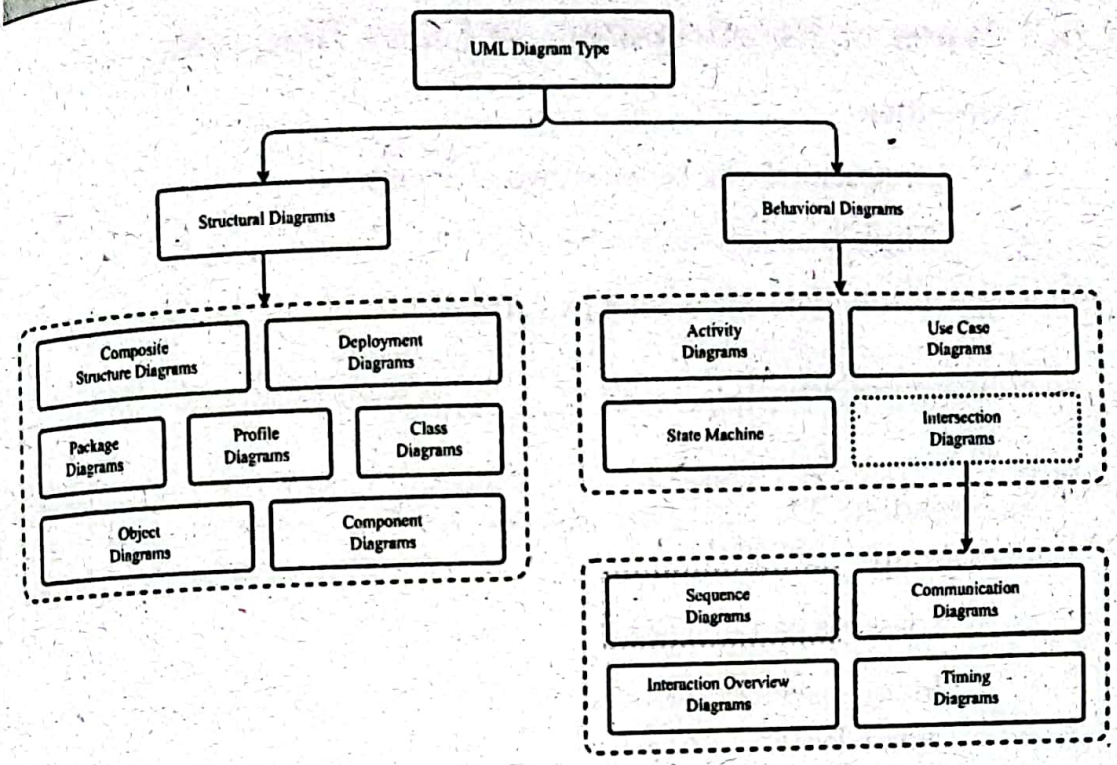
The parameters return value, and objects local to an operation provide the context for its realization. Therefore, these elements are visible to the structural aspect of the collaboration that realizes the operation, just as actors are visible to the structural aspect of a collaboration that realizes a use case. You can model the relationship among these parts using class diagrams that specify the structural part of collaboration.

To model the implementation of an operation:

- Identify the parameters, return value, and other objects visible to the operation.
- If the operation is trivial, represent its implementation directly in code, which you can keep in the backplane of your model, or explicitly visualize it in a note.
- If the operation is algorithmically intensive, model its realization using an activity diagram.
- If the operation is complex or requires detailed design then represent its implementation as collaboration. The structural and behavioral aspects of the design can be expanded via class and interaction diagrams.

3.5 Class Diagrams

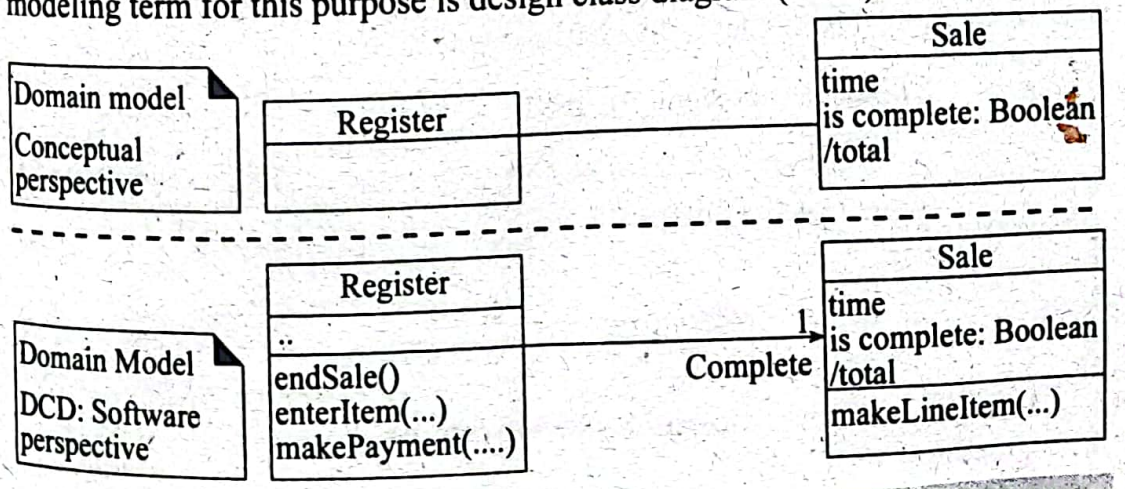
In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.



A class diagram depicts the static structure of a class. It is made up of the set of classes and their relationships between classes. It provides a basic notation of the static structure of the system. It is extremely useful for developers as well as the system analyst to study the system under consideration.

3.5.1 Design Class Diagram (DCD)

In a conceptual perspective the class diagram can be used to visualize a domain model. For discussion, we also need a unique term to clarify when the class diagram is used in a software or design perspective. A common modeling term for this purpose is design class diagram (DCD).



3.5.2 Classifier

A UML classifier is "a model element that describes behavioral and structure features". Classifiers can also be specialized. They are a generalization of many of the elements of the UML, including classes, interfaces, use cases, and actors. In class diagrams, the two most common classifiers are regular classes and interfaces.

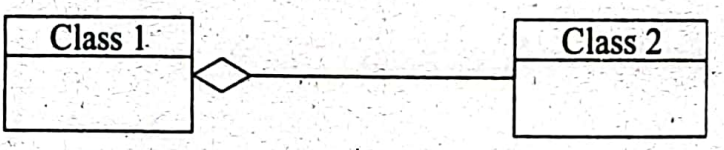
3.5.3 Types of Relationships in Class Diagrams

1. **Association**

- A structural link between two peer classes.
- There is an association between Class1 and Class2
- A solid line connecting two classes

2. **Aggregation:** A special type of association. It represents a "part of" relationship.

- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.
- A solid line with an unfilled diamond at the association end connected to the class of composite

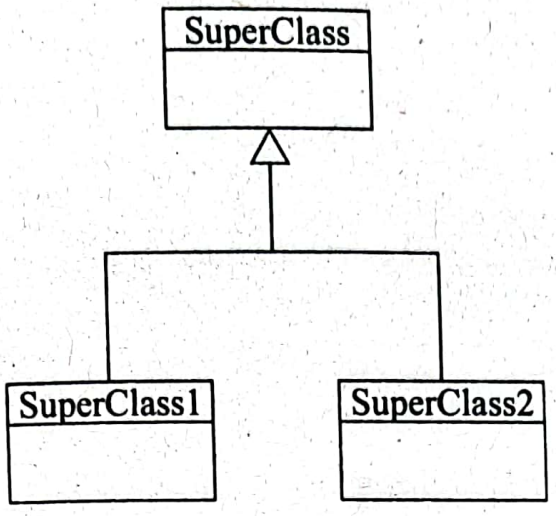


3. **Composition:** A special type of aggregation where parts are destroyed when the whole is destroyed.

- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.
- A solid line with a filled diamond at the association connected to the class of composite

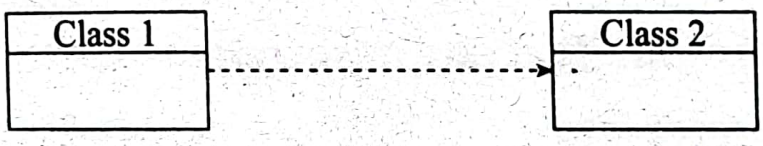
4. **Inheritance(Generalization/Specialization)**

- Represents an "is-a" relationship.
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of Super Class.
- A solid line with a hollow arrowhead that point from the child to the parent class



5. **Dependency**

- Exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2
- A dashed line with an open arrow



3.5.4 Comparison between Association, Aggregation and Composition

Association	Aggregation	Composition
Association relationship is represented using an arrow.	Aggregation relationship is represented by a straight line with an empty diamond at one end.	The composition relationship is represented by a straight line with a black diamond at one end.
In UML, it can exist between two or more classes.	It is a part of the association relationship.	It is a part of the aggregation relationship.
It incorporates one-to-one, one-to-many, many-to-one, and many-to-many association between the classes.	It exhibits a kind of weak relationship.	It exhibits a strong type of relationship.
It can associate one more objects together.	In an aggregation relationship, the associated objects exist independently within the scope of the system.	In a composition relationship, the associated objects cannot exist independently within the scope of the system.

In this, objects are linked together.	In this, the linked objects are independent of each other.	Here the linked objects are dependent on each other.
It may or may not affect the other associated element if one element is deleted.	Deleting one element in the aggregation relationship does not affect other associated elements.	It affects the other element if one of its associated element is deleted.
Example: A tutor can associate with multiple students, or one student can associate with multiple teachers.	Example: A car needs a wheel for its proper functioning, but it may not require the same wheel. It may function with another wheel as well.	Example: If a file is placed in a folder and that is folder is deleted. The file residing inside that folder will also get deleted at the time of folder deletion.

3.5.5 Attribute Text and Association Lines

Attributes of a classifier (also called structural properties in the UML) are shown several ways:

- Attribute text notation, such as current Sale : Sale.
- Association line notation
- Both together

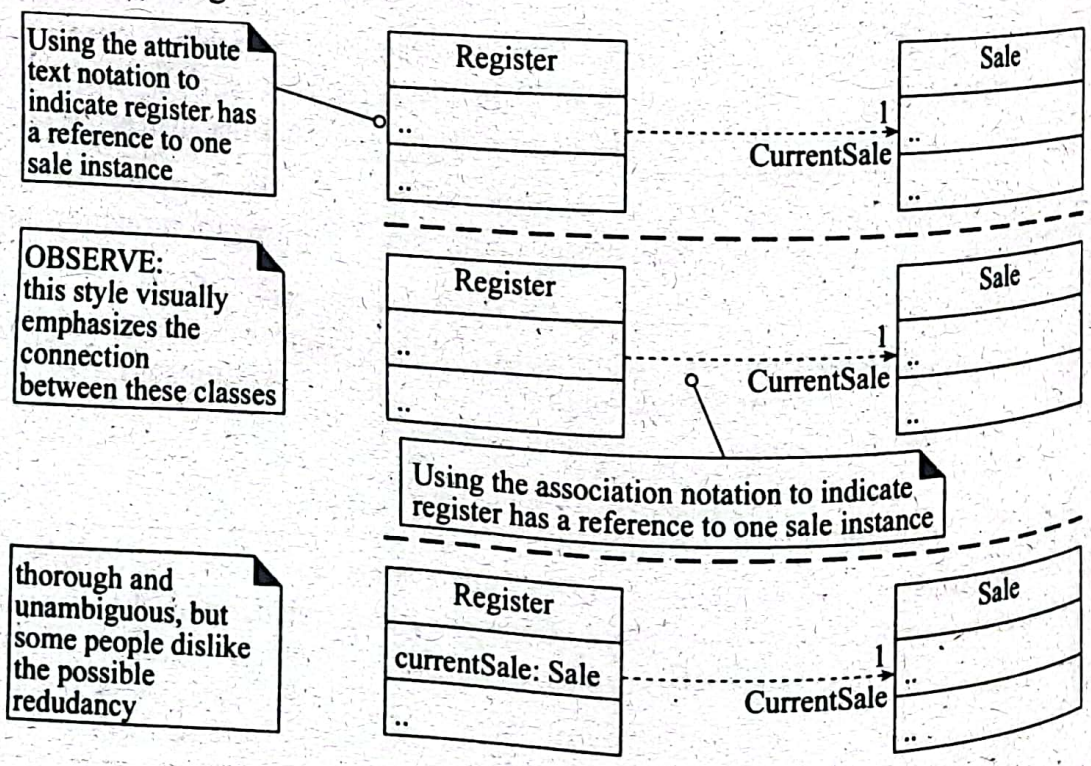


Figure shows these notations being used to indicate that a Register object has an attribute (a reference to) one Sale object.

The full format of the attribute text notation is:

visibility name : type multiplicity = default {property-string}

Also, the UML allows any other programming language syntax to be used for the attribute declaration, as long as the reader or tools are notified.

visibility marks include + (public), – (private), and so forth.

Guideline: Attributes are usually assumed private if no visibility is given.

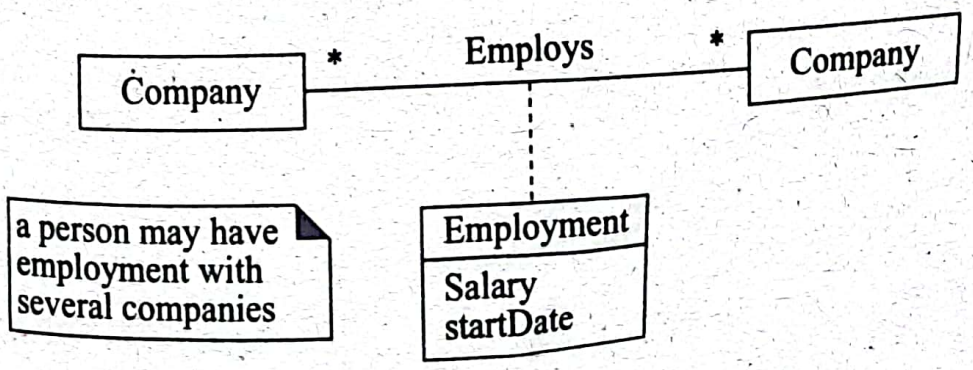
Notes, Comments, Constraints, and Method Bodies

Note symbols can be used on any UML diagram, but are especially common on class diagrams. A UML note symbol is displayed as a dog-eared rectangle with a dashed line to the annotated element; A note symbol may represent several things, such as:

- a UML note or comment, which by definition have no semantic impact
- a UML constraint, in which case it must be encased in braces '{...}'
- a method body the implementation of a UML operation.

3.5.6 Association Class

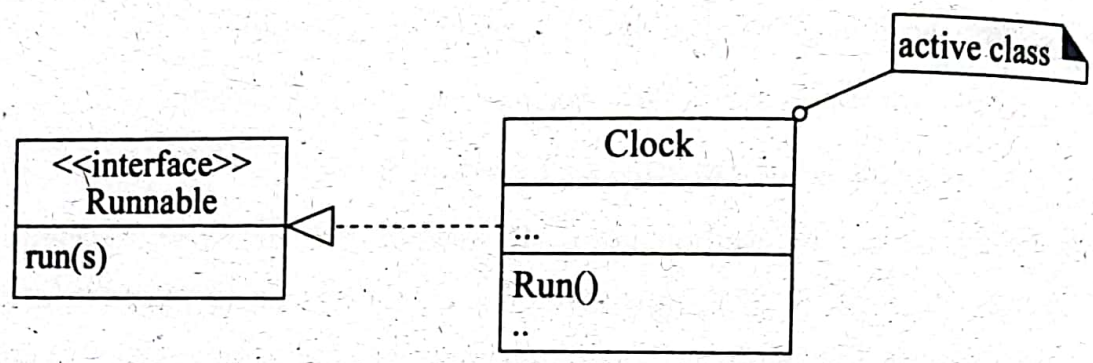
An association class allows you to treat an association itself as a class, and model it with attributes, operations, and other features. For example, if a Company employs many Persons, modeled with an Employs association, you can model the association itself as the Employment class, with attributes such as startDate. In the UML, it is illustrated with a dashed line from the association to the association class.



Singleton Classes

Active class

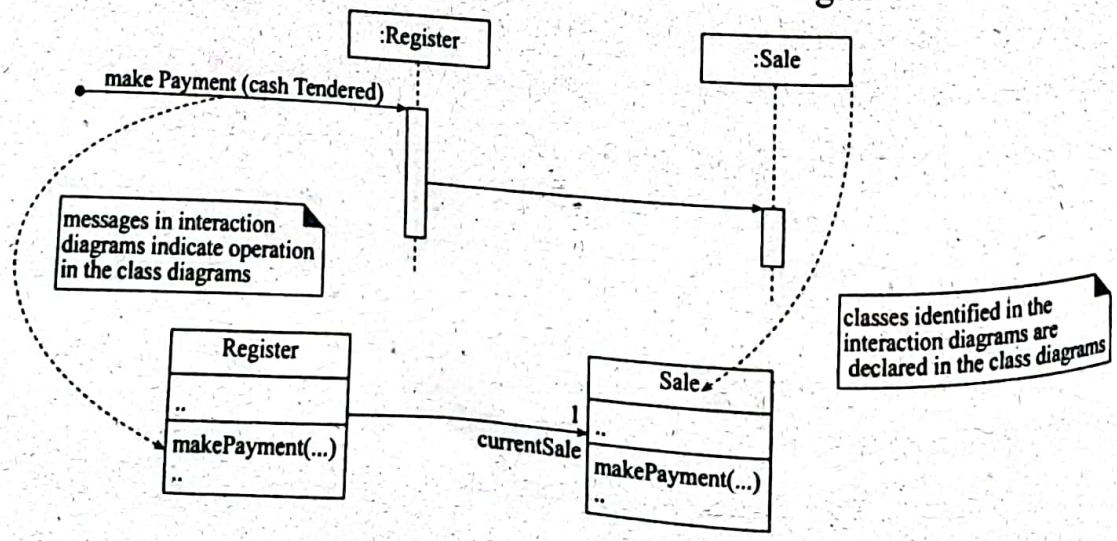
An active object runs on and controls its own thread of execution. Not surprisingly, the class of an active object is an active class. In the UML, it may be shown with double vertical lines on the left and right sides of the class box.



Relationship between Interaction and Class Diagrams?

When we draw interaction diagrams, a set of classes and their methods emerge from the creative design process of dynamic object modeling. Thus, from interaction diagrams the definitions of class diagrams can be generated. This suggests a linear ordering of drawing interaction diagrams before class diagrams, but in practice, especially when following the agile modeling practice of models in parallel, these complementary dynamic and static views are drawn concurrently.

Guideline: A good UML tool should automatically support changes in one diagram being reflected in the other. If wall sketching, use one wall for interaction diagrams, and an adjacent wall for class diagrams.



3.6 Deployment Diagrams

Deployment diagrams are one of the two kinds of diagrams used in modeling the physical aspects of an object-oriented system. A deployment diagram shows the configuration of run time processing nodes and the components that live on them.

You use deployment diagrams to model the static deployment view of a system. For the most part, this involves modeling the topology of the hardware on which your system executes. Deployment diagrams are essentially class diagrams that focus on a system's nodes.

A deployment diagram is a diagram that shows the configuration of run time processing nodes and the components that live on them. Graphically, a deployment diagram is a collection of vertices and arcs.

Deployment diagrams commonly contain

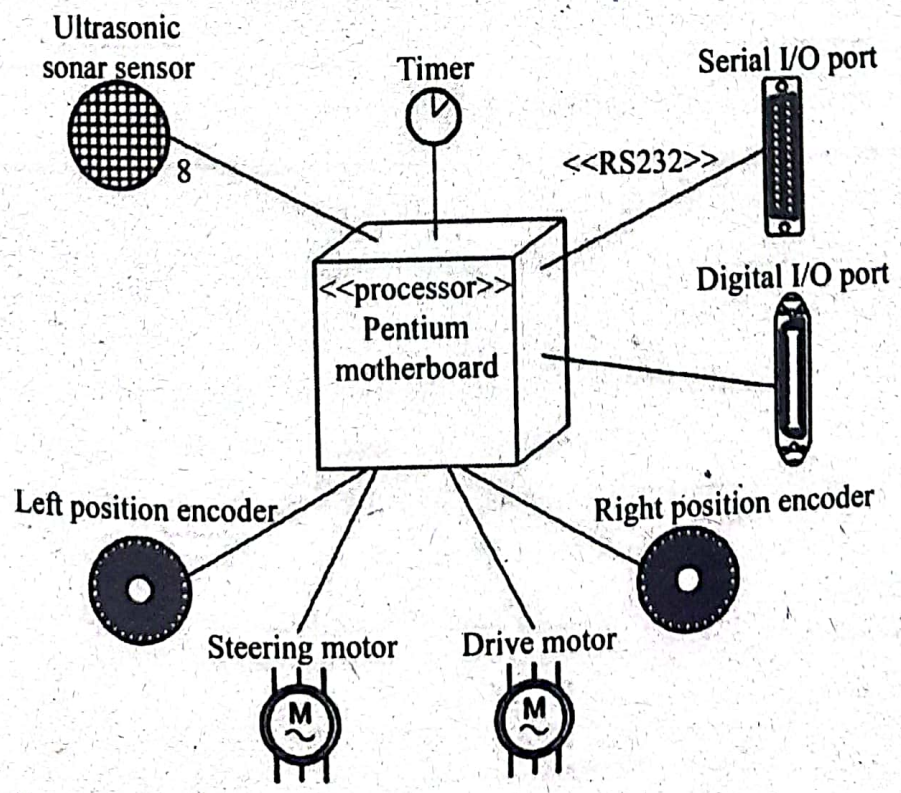
- Nodes
- Dependency and association relationships

When you model the static deployment view of a system, you'll typically use deployment diagrams in one of three ways.

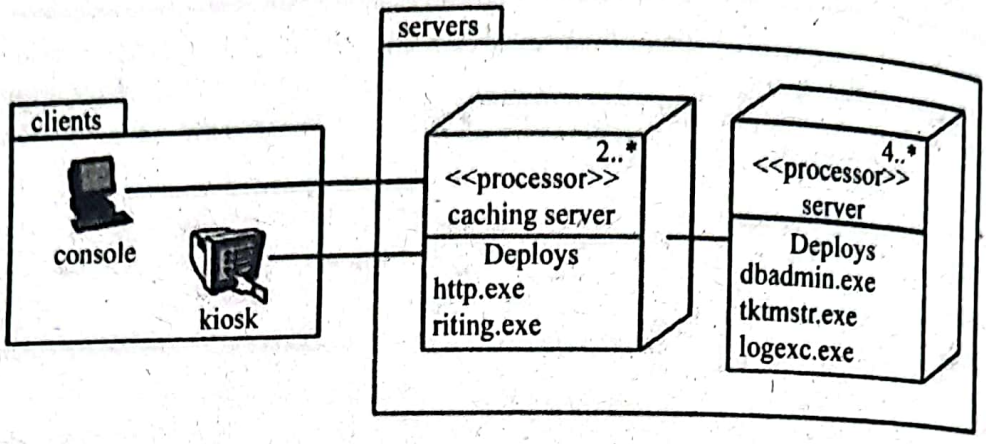
1. To model embedded systems
2. To model client/server systems
3. To model fully distributed systems

3.6.1 Some examples of

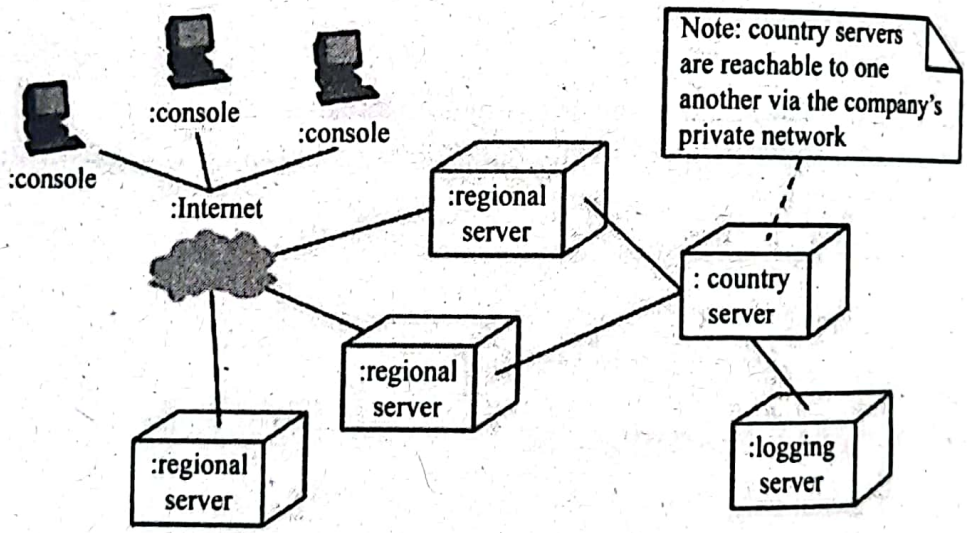
a. Modeling a Embedded system



b. Modeling a Client/Server system



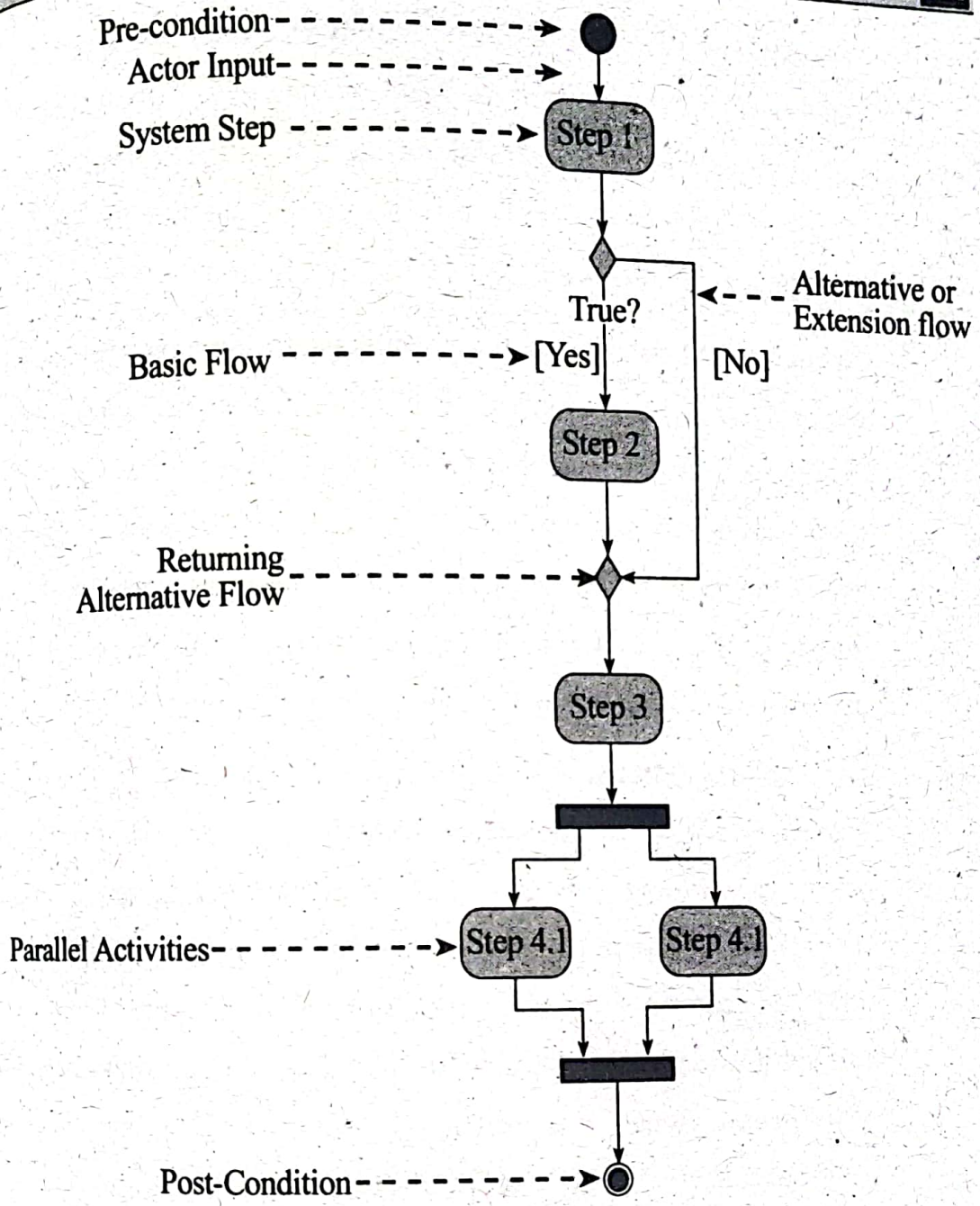
c. Modeling a Distributed system



3.7 Activity Diagram

Activity diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. An activity diagram is essentially a flowchart, showing flow of control from activity to activity.

You use activity diagrams to model the dynamic aspects of a system. For the most part, this involves modeling the sequential (and possibly concurrent) steps in a computational process. With an activity diagram, you can also model the flow of an object as it moves from state to state at different points in the flow of control. Activity diagrams may be used to model the flow of control of an operation



3.7.1 Terms and Concepts

Activity diagrams commonly contain

- **Activity states and action states:** In the flow of control modeled by an activity diagram, things happen. You might evaluate some expression that sets the value of an attribute or that returns some value. Alternatively, you might call an operation on an object, send a signal to an object, or even create or destroy an object. These executable, atomic computations are called action states because they are states of the system, each representing the execution of an action.

- **Transitions:** When the action or activity of a state completes, flow of control passes immediately to the next action or activity state. You specify this flow by using transitions to show the path from one action or activity state to the next action or activity state. In the UML, you represent a transition as a simple directed line,

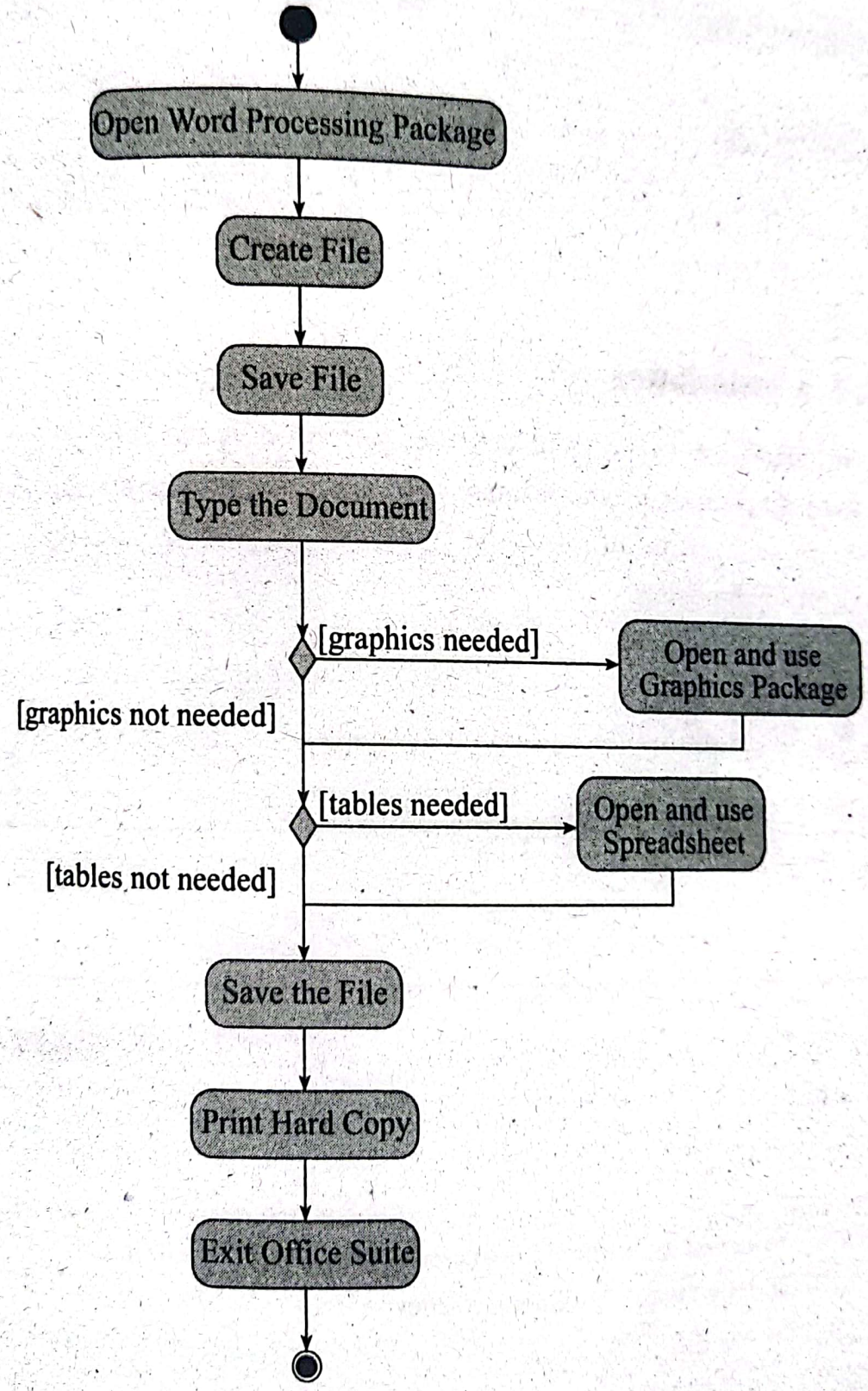
- **Branching**: Simple, sequential transitions are common, but they aren't the only kind of path you'll need to model a flow of control. As in a flowchart, you can include a branch, which specifies alternate paths taken based on some Boolean expression. As you represent a branch as a diamond. A branch may have one incoming transition and two or more outgoing ones. On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch.

- **Forking and Joining:** A fork represents the splitting of a single flow of control into two or more concurrent flows of control. A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. Below the fork, the activities associated with each of these paths continue in parallel. Conceptually, the activities of each of these flows are truly concurrent, although, in a running system, these flows may be either truly concurrent (in the case of a system deployed across multiple nodes) or sequential yet interleaved (in the case of a system deployed across one node), thus giving only the illusion of true concurrency.

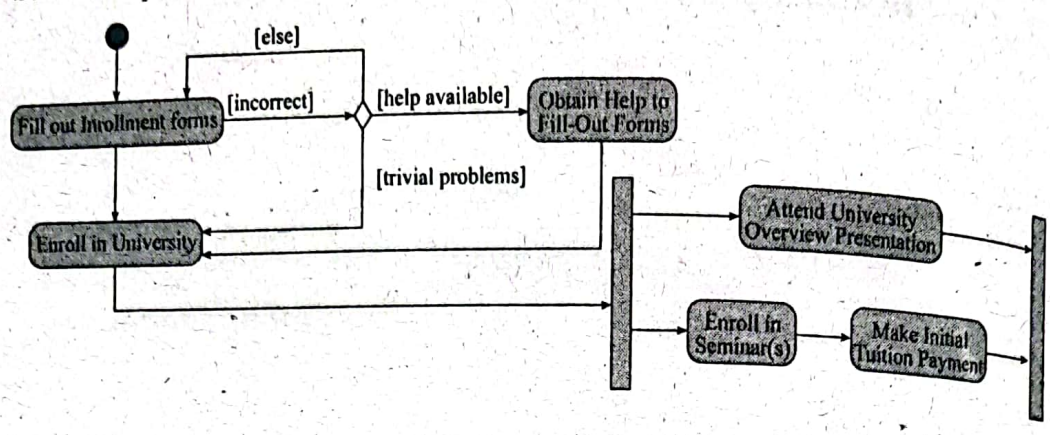
- **Objects:** The things that are involved in an activity diagram are connected using a dependency to the activity or transition that creates, destroys, or modifies them. This use of dependency relationships and objects is called an object flow because it represents the participation of an object in a flow of control.

3.7.2 Some Examples

a. **Word Processor**

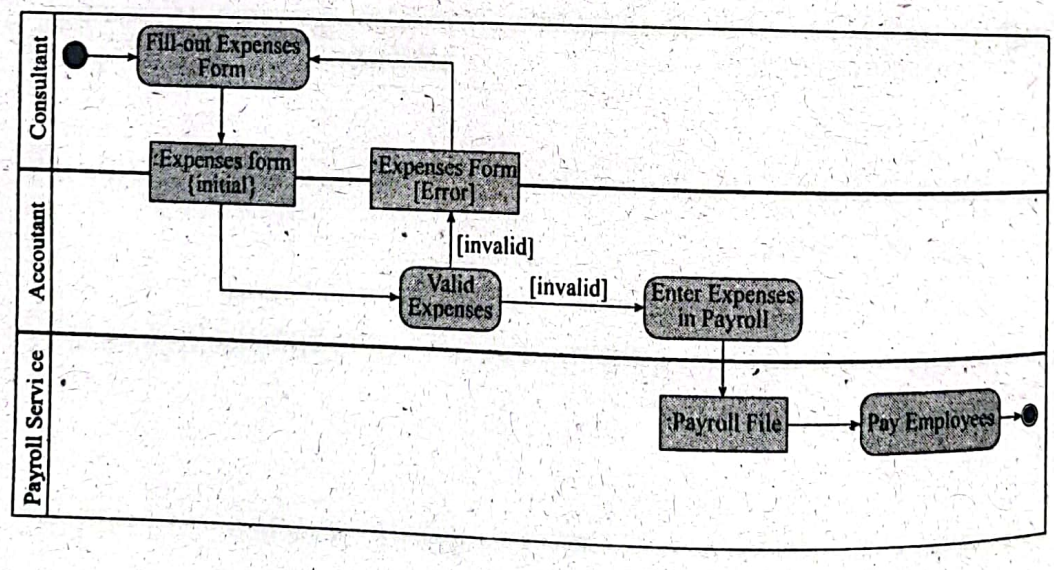


b. Student Enrollment

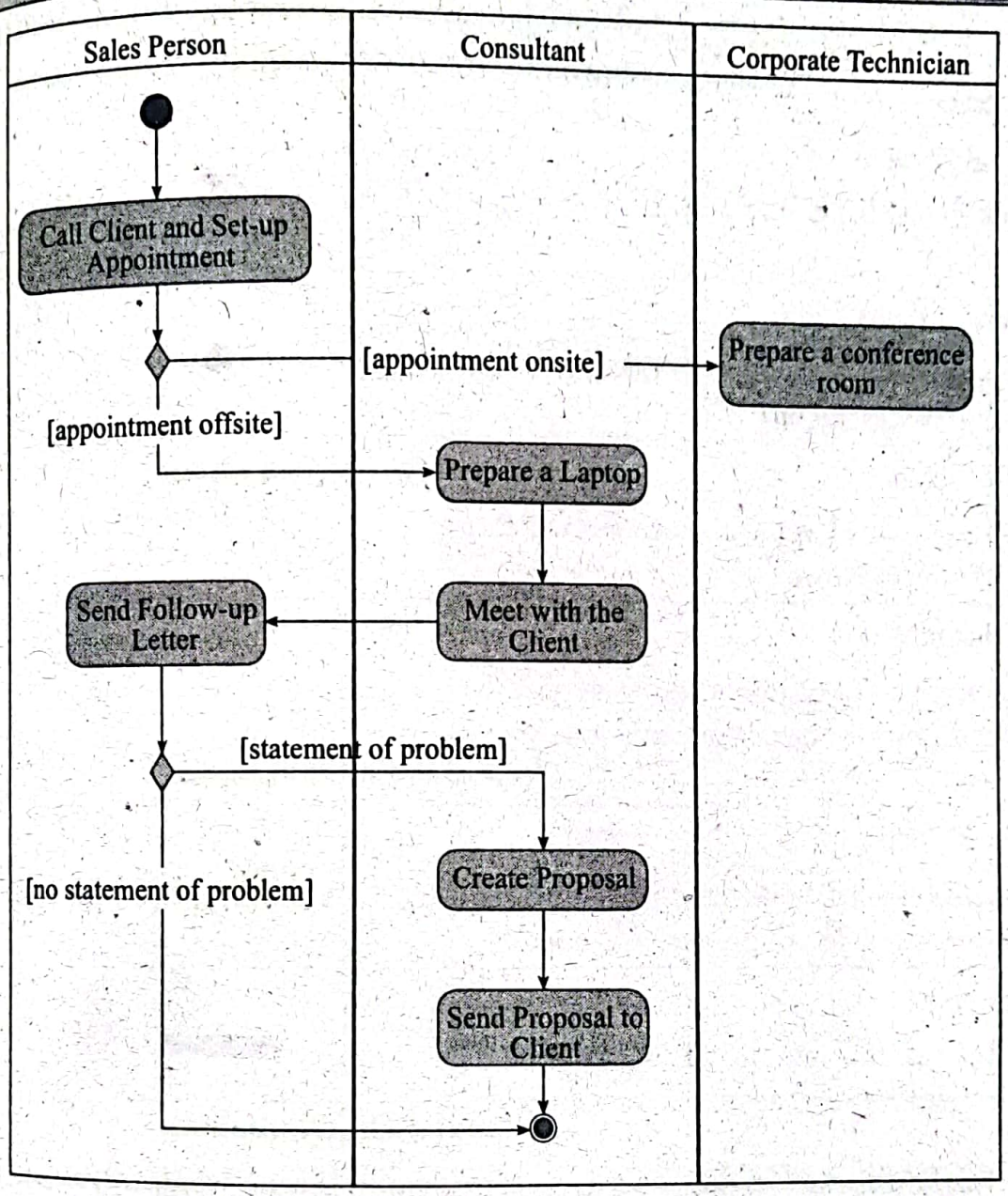


3.7.3 Swimlanes

A swimlane is a way to group activities performed by the same actor on an activity diagram or activity diagram or to group activities in a single thread. Here is an example of a swimlane activity diagram for modeling Staff Expenses Submission:



This figure below describes the business process for meeting a new client using an activity Diagram with swimlanes



3.7.4 Activity Diagram Notations

Notation Description	UML Notation
Activity: Used to represent activity	Activity
Action: Task to be performed.	Action
Control Flow Shows the sequence of execution.	
Object Flow Show the flow of an object from one	

activity (or action) to another activity (or action).	
Initial Node Portrays the beginning of a set of actions or activities	
Activity Final Node Stop all control flows and object flows in an activity (or action)	
Object Node Represent an object that is connected to a set of Object Flows	Object Node
Decision Node Represent a test condition to ensure that the control flow or object flow only goes down one path	[guard-x] [guard-y]
Merge Node Bring back together different decision paths that were created using a decision-node.	
Fork Node Split behavior into a set of parallel or concurrent flows of activities (or actions)	
Join Node Bring back together a set of parallel or concurrent flows of activities (or actions).	
Swimlane and Partition A way to group activities performed by the same actor on an activity diagram or to group activities in a single thread	Partition2 Partition1

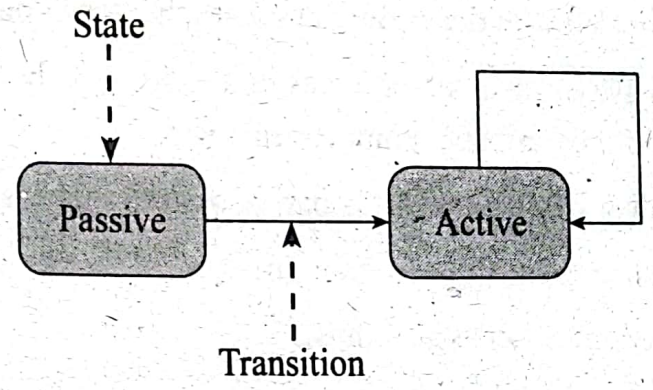
3.8 State Machine Diagram

State machine diagrams typically are used to describe state-dependent behavior for an object. An object responds differently to the same event depending on what state it is in. State machine diagrams are usually applied to objects but can be applied to any element that has behavior to other entities such as: actors, use cases, methods, subsystems systems etc. and they are typically used in conjunction with interaction diagrams (usually sequence diagrams).

Characteristics of State Machine Notations:

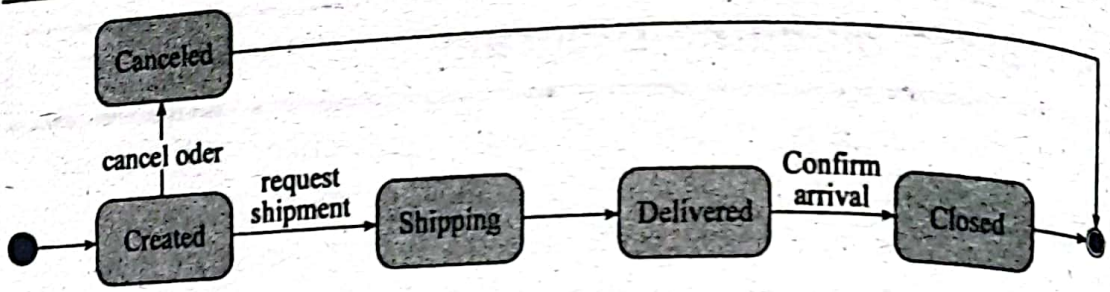
- A state occupies an interval of time.
- A state is often associated with an abstraction of attribute values of an entity satisfying some condition(s).
- An entity changes its state not only as a direct consequence of the current input, but it is also dependent on some past history of its inputs.

State: A state is an abstraction of the attribute values and links of an object. Sets of values are grouped together into a state according to properties that affect the gross behavior of the object.



Initial and Final States

- The initial state of a state machine diagram, known as an initial pseudo-state, is indicated with a solid circle. A transition from this state will show the first real state.
- The final state of a state machine diagram is shown as concentric circles. An open loop state machine represents an object that may terminate before the system terminates, while a closed loop state machine diagram does not have a final state; if it is the case, then the object lives until the entire system terminates.



Events:

An event signature is described as Event-name (comma-separated-parameter-list). Events appear in the internal transition compartment of a state or on a transition between states. An event may be one of four types:

- **Signal event** - corresponding to the arrival of an asynchronous message or signal
- **Call event** - corresponding to the arrival of a procedural call to an operation
- **Time event** - a time event occurs after a specified time has elapsed
- **Change event** - a change event occurs whenever a specified condition is met

Transition

Transition lines depict the movement from one state to another. Each transition line is labeled with the event that causes the transition.

- Viewing a system as a set of states and transitions between states is very useful for describing complex behaviors
- Understanding state transitions is part of system analysis and design
- A Transition is the movement from one state to another state

Transitions between states occur as follows:

- An element is in a source state
- An event occurs
- An action is performed
- The element enters a target state

Actions

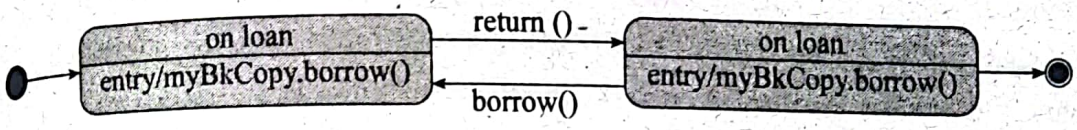
Action is an executable atomic computation, which includes operation calls, the creation or destruction of another object, or the sending of a signal to an object. An action is associated with transitions and during which an action is not interruptible - e.g., entry, exit.

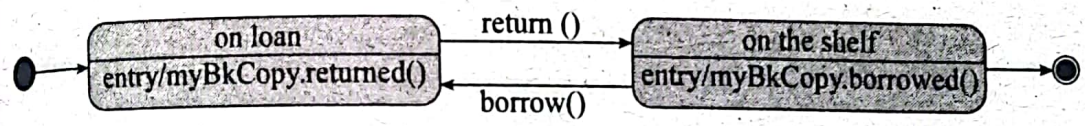
Activity

Activity is associated with states, which is a non-atomic or ongoing computation. Activity may run to completion or continue indefinitely. An Activity will be terminated by an event that causes a transition from the state in which the activity is defined.

Simple State Machine Diagram Notation:

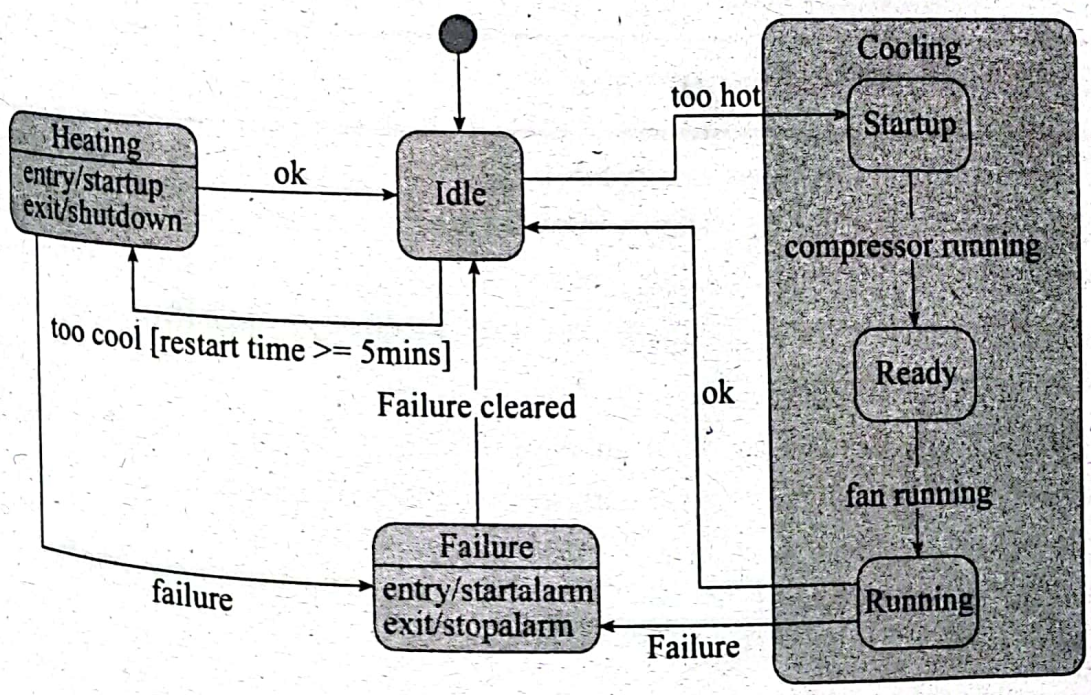
a. **Entry and Exit actions:** Entry and Exit actions specified in the state. It must be true for every entry / exit occurrence. If not, then you must use actions on the individual transition arcs





- Entry Action executed on entry into state with the notation: Entry / action
- Exit Action executed on exit from state with the notation: Exit / action

b. **Substates:** A simple state is one which has no substructure. A state which has substates (nested states) is called a composite state. Substates may be nested to any level. A nested state machine may have at most one initial state and one final state. Substates are used to simplify complex flat state machines by showing that some states are only possible within a particular context (the enclosing state).

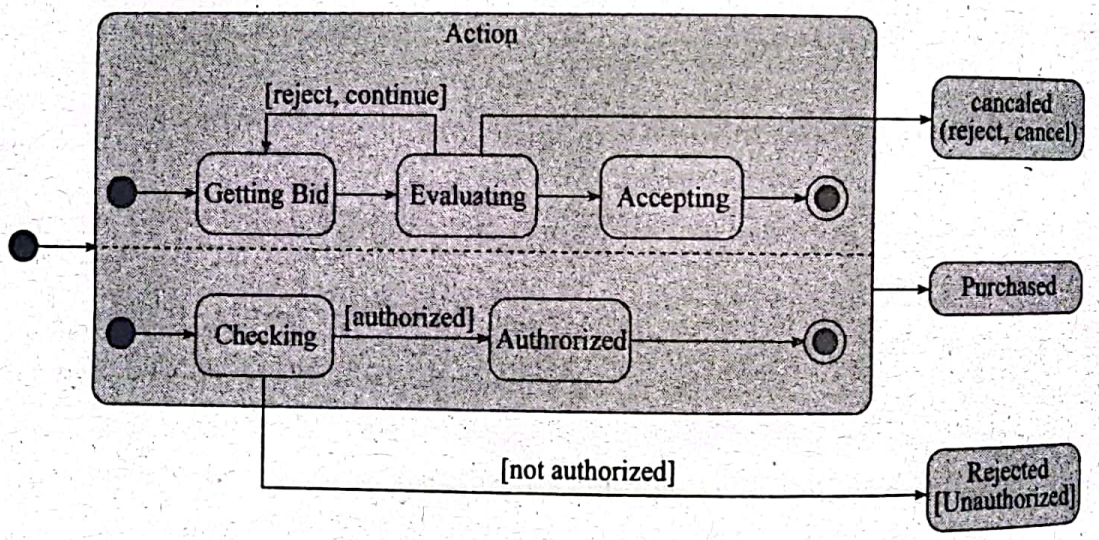


Concurrent states

As mentioned above, states in state machine diagrams can be nested. Related states can be grouped together into a single composite state. Nesting states inside others is necessary when an activity involves concurrent sub-activities. The following state machine diagram models an auction with two concurrent substates: processing the bid and authorizing the payment limit.

Concurrent State Machine Diagram Example - Auction Process

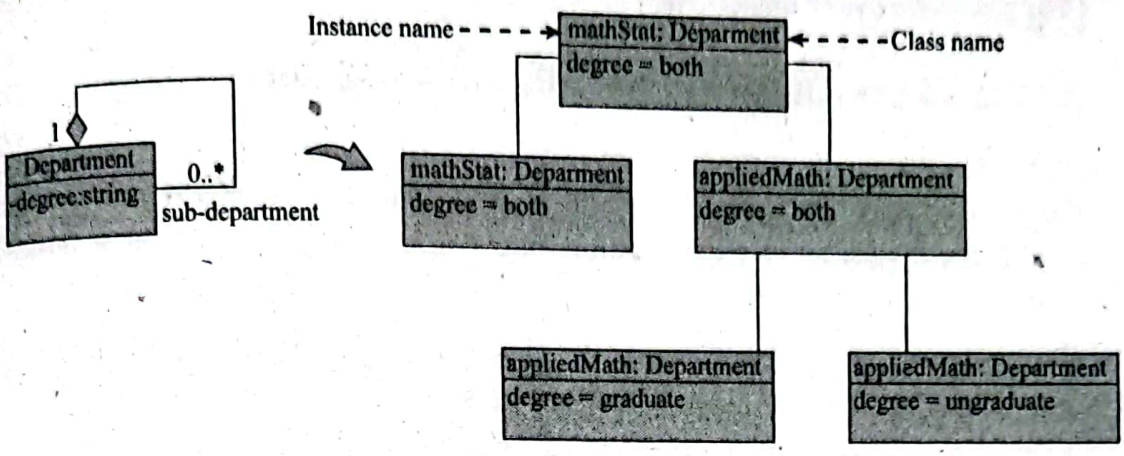
In this example, the state machine first entering the Auction requires a fork at the start into two separate start threads. Each sub-state has an exit state to mark the end of the thread. Unless there is an abnormal exit (Canceled or Rejected), the exit from the composite state occurs when both sub-states have exited.



3.9 Object Diagrams

An object diagram shows this relation between the instantiated classes and the defined class, and the relation between these objects in the system. They are be useful to explain smaller portions of your system, when your system class diagram is very complex, and also sometimes modeling recursive relationship in diagram.

The best way to illustrate what an object diagram look like is to show the object diagram derived from the corresponding class diagram.



3.9.1 Basic Object Diagram Symbols and Notations

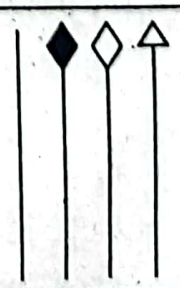
Object Names: Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon.

Object Diagram Notation: Object

Object Attributes: Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, object attributes should have values assigned for them.

Links: Links tend to be instances associated with associations. You can draw a link while using the lines utilized in class diagrams.

The table below shows the different notations:

Object Names: Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon.	Object: Class
Object Attributes: Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, object attributes should have values assigned for them.	Object: Class attribute = value
Links: Links tend to be instances associated with associations. You can draw a link while using the lines utilized in a class diagram	

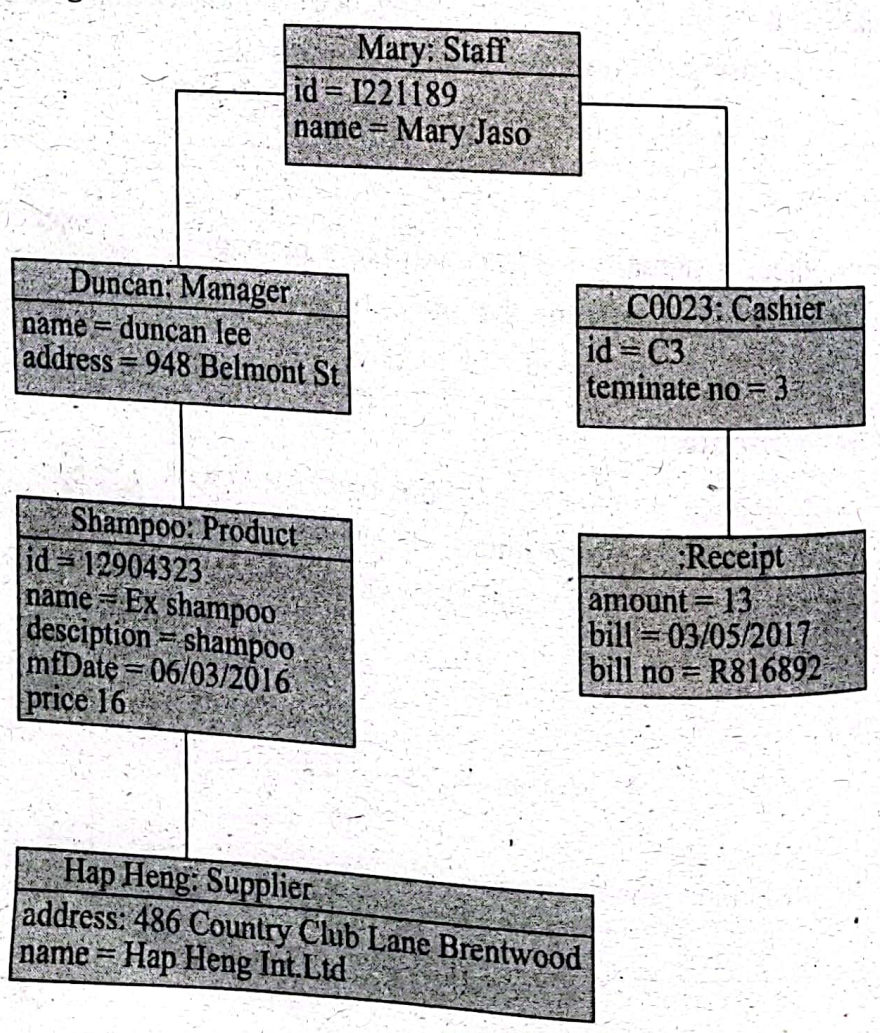
3.9.2 Class Diagram vs. Object Diagram

In UML, object diagrams provide a snapshot of the instances in a system and the relationships between the instances. By instantiating the model elements in a class diagram, you can explore the behavior of a system at a point in time.

- An object diagram is a UML structural diagram that shows the instances of the classifiers in models.
- Object diagrams use notation that is similar to that used in class diagrams.
- Class diagrams show the actual classifiers and their relationships in a system
- Object diagrams show specific instances of those classifiers and the links between those instances at a point in time.
- You can create object diagrams by instantiating the classifiers in class, deployment, component, and use-case diagrams.

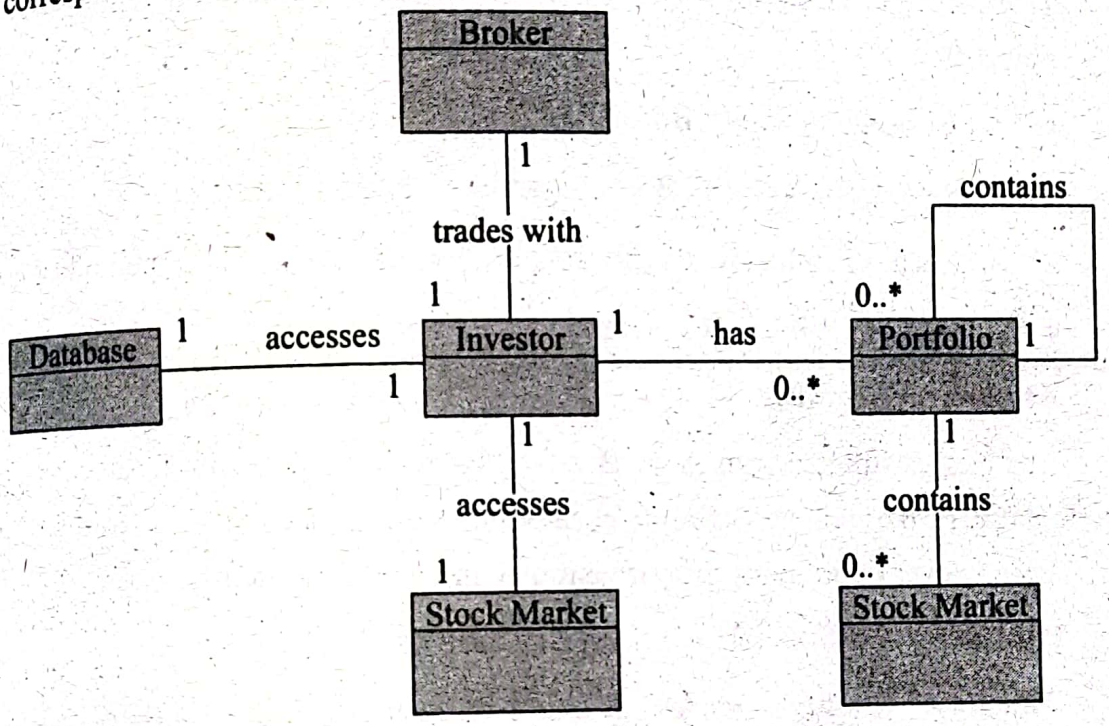
3.9.3 Example

Object Diagram of POS

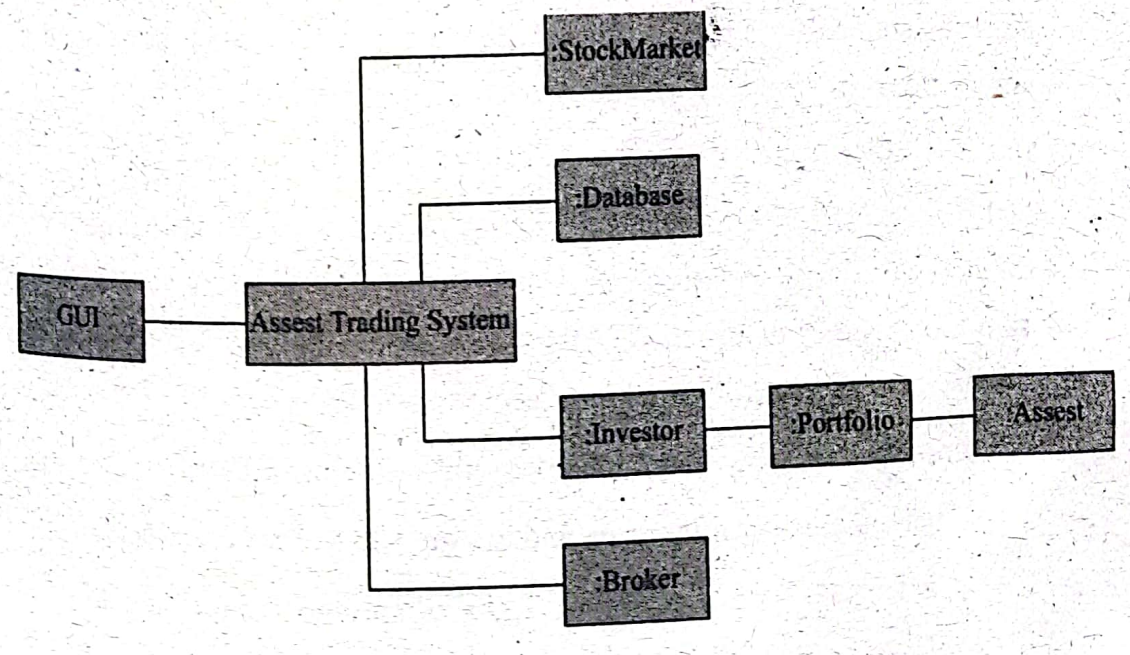


Class diagrams to Objects Diagram:

Besides showing the objects' state at a particular point in time, an object diagram can also be used to represent the occurrences of interactions between classes in runtime. The result looks a bit like a communication diagram. The figure below provides an example of such a class diagram and its corresponding object diagram:



The Object Diagram is:



3.10 Visibility

Visibility is the ability of an object to "see" or have a reference to another object. More generally, it is related to the issue of scope: Is one resource (such as an instance) within the scope of another?

There are four common ways that visibility can be achieved from object **A** to object **B**:

- Attribute visibility **B** is an attribute of **A**.
- Parameter visibility **B** is a parameter of a method of **A**.
- Local visibility **B** is a (non-parameter) local object in a method of **A**.
- Global visibility **B** is in some way globally visible.

Attribute Visibility

Attribute visibility from **A** to **B** exists when **B** is an attribute of **A**. It is a relatively permanent visibility because it persists as long as **A** and **B** exist. This is a very common form of visibility in object-oriented systems.

Parameter Visibility

Parameter visibility from **A** to **B** exists when **B** is passed as a parameter to a method of **A**. It is relatively temporary visibility because it persists only within the scope of the method. After attribute visibility, it is the second most common form of visibility in object-oriented systems.

Local Visibility

Local visibility from **A** to **B** exists when **B** is declared as a local object within a method of **A**. It is relatively temporary visibility because it persists only within the scope of the method. After parameter visibility, it is the third most common form of visibility in object-oriented systems.

Two common means by which local visibility is achieved are:

- Create a new local instance and assign it to a local variable.
- Assign the returning object from a method invocation to a local variable.

As with parameter visibility, it is common to transform locally declared visibility into attribute visibility.

Global Visibility

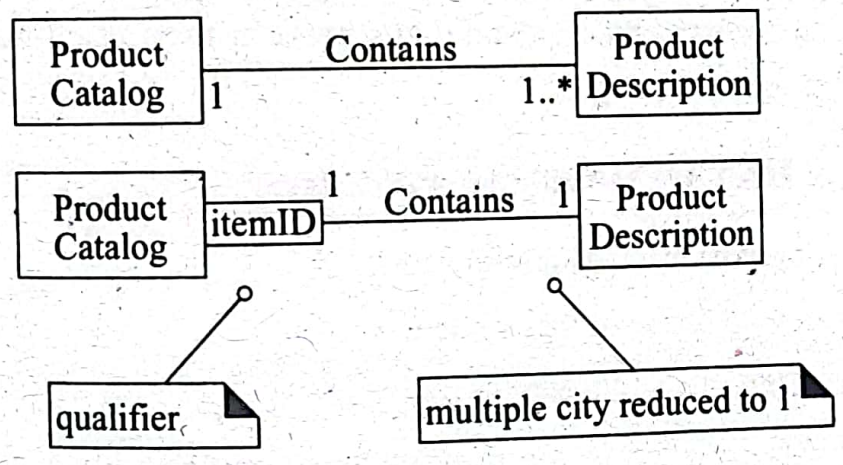
Global visibility from **A** to **B** exists when **B** is global to **A**. It is a relatively permanent visibility because it persists as long as **A** and **B** exist. It is the least common form of visibility in object-oriented systems. One way to achieve global visibility is to assign an instance to a global variable, which is possible in some languages, such as **C++**, but not others, such as Java.

Constraints

Constraints may be used on most UML diagrams, but are especially common on class diagrams. A UMLconstraint is a restriction or condition on a UML element. It is visualized in text between braces; for example: { size >= 0 }. The text may be natural language or anything else, such as UML's formal specification language, the Object Constraint Language.

Qualified Association

A qualified association has a qualifier that is used to select an object (or objects) from a larger set of related objects, based upon the qualifier key.



3.11 What are Patterns?

Patterns are the principles and solutions in structured format which describe a problem and a solution. A named problem or solution pair that can be applied in a new variety of concepts. It is generic advice from previous designers to help designers in a new situation.

The idea behind design patterns is simple: Write down and catalog common interactions between objects that programmers have frequently found useful.

3.11.1 Characteristics of Good patterns

- It solves a problem
- The solution isn't obvious
- The pattern has a significant human component
- It is a proven concept
- It describes a relationship

3.11.2 Types of Design Pattern

- Creational Patterns : Deals with initializing classes and objects.
- Structural Patterns : Deals with decoupling interface and implementation of classes and objects.
- Behavioral Patterns : Deals with dynamic behavior of classes and objects.

Types of Patterns

- ***Architectural Pattern:*** Expresses a fundamental structural organization or schema for software systems.
- ***Design Pattern:*** Provides a scheme for refining the subsystems or components of a software system, or the relationships between them.
- ***Idioms:*** An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

3.11.3 How to Describe Patterns?

Name: It must have a meaningful name.

Problem: A statement of the problem.

Context: This tells us the pattern's applicability.

Forces: A description of the relevant forces and constraints and how they interact/conflict with one another..

Solution: Static relationships and dynamic rules describing how to realize the desired outcome.

Consequences: Implications(good and bad) of using the solution.

Examples: One or more sample applications of the pattern .

Thus a design pattern has four important components:

- The problem
- The context in which the problem occurs.
- The solution.
- The context within which the solution works.

3.12 GRASP Patterns

GRASP is an acronym for "General Responsibility Assignment Software Patterns", is a set of nine fundamental principles in OOD and responsibility assignment.

- Information Expert
- Low Coupling
- Controller
- Pure Fabrication
- Indirection
- Creator
- High Cohesion
- Polymorphism
- Protected Variations

1. **Information Expert:**

   Problem: What is a basic principle by which to assign responsibilities to objects?

   Solution: Assign responsibility to the class that has the information needed to fulfill it.

   Information expert (also expert or the expert principle) is a principle used to determine where to delegate responsibilities such as methods, computed fields, and so on. Using the principle of information expert, a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored.

2. **Creator:** The creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.

   Problem: Who creates object A?

   Solution: In general, Assign class B the responsibility to create object A if one, or preferably more, of the following apply:

   - Instances of B contain or compositely aggregate instances of A
   - Instances of B record instances of A
   - Instances of B closely use instances of A
   - Instances of B have the initializing information for instances of A and pass it on creation.

3. **Controller:**

   A controller object is a non-user interface object responsible for receiving or handling a system event.

Problem: Who should be responsible for handling an input system event?

Solution: A use case controller should be used to deal with all system events of a use case, and may be used for more than one use case. For instance, for the use cases Create User and Delete User, one can have a single class called UserController, instead of two separate use case controllers. Alternatively a facade controller would be used; this applies when the object with responsibility for handling the event represents the overall system or a root object.

4. **Indirection:**

The indirection pattern supports low coupling and reuses potential between two elements by assigning the responsibility of mediation between them to an intermediate object.

Problem: Where to assign responsibility, to avoid direct coupling between two (or more) things? How to decouple objects so that low coupling is supported and reuse potential remains higher?

Solution: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.

5. **Low Coupling:**

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. Low coupling is an evaluative pattern that dictates how to assign responsibilities for the following benefits:

- lower dependency between the classes,
- change in one class having a lower impact on other classes,
- higher reuse potential.

6. **High Cohesion:**

High cohesion is generally used in support of low coupling. High cohesion means that the responsibilities of a given set of elements are strongly related and highly focused on a rather specific topic. Breaking programs into classes and subsystems, if correctly done, is an example of activities that increase the cohesive properties of named classes and subsystems.

7. **Polymorphism:**

According to the **polymorphism** principle, responsibility for defining the variation of behaviors based on type is assigned to the type for which this variation happens. This is achieved using polymorphic operations. The user of the type should use polymorphic operations instead of explicit branching based on type.

Problem: How to handle alternatives based on type? How to create pluggable software components?

Solution: When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies.

8. **Protected Variations:**

The protected variations pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

Problem: How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?

Solution: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

9. **Pure Fabrication:**

A **pure fabrication** is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived (when a solution presented by the information expert pattern does not). This kind of class is called a "**service**" in domain-driven design.

Old Exam Question-Answer

1. **Explain controller pattern and polymorphism pattern of GRASP pattern with suitable examples.** **[6] [2079 Jestha]**

⇒ Refer to theory

2. **Describe and list out all types of visibility with suitable examples.** **[6] [2079 Jestha]**

⇒ Refer to theory

3. List out all the associations used in the class diagram with suitable examples. **[2079 Jestha]**

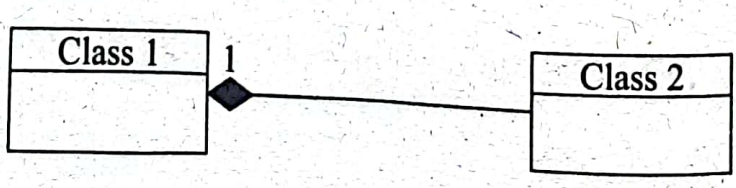
⇒ In UML diagrams, relationships are used to link several things. It is a connection between structural, behavioral, or grouping things. Following are the standard UML relationships enlisted below:

1. **Association:** Association relationship is a structural relationship in which different objects are linked within the system. It exhibits a binary relationship between the objects representing an activity. It depicts the relationship between objects, such as a teacher, can be associated with multiple teachers.

   It is represented by a line between the classes followed by an arrow that navigates the direction, and when the arrow is on both sides, it is then called a bidirectional association. We can specify the multiplicity of an association by adding the adornments on the line that will denote the association.

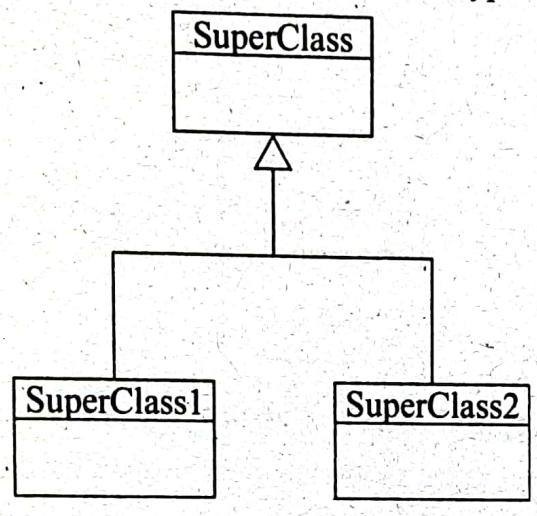
2. **Aggregation:** Aggregation is a subset of association, is a collection of different things. It represents a relationship. It is more specific than an association. It describes a part-whole or part-of relationship. It is a binary association, i.e., it only involves two classes. It is a kind of relationship in which the child is independent of its parent. It represents a "part of" relationship.

3. **Composition:** The composition is a part of aggregation, and it portrays the whole-part relationship. It depicts dependency between a composite (parent) and its parts (children), which means that if the composite is discarded, so will its parts get deleted. It exists between similar objects.



4. **Inheritance (Generalization/Specialization):** A generalization relationship is a relationship in which one model element (the child) is based on another model element (the parent). Generalization relationships are used in class, component, deployment, and use-case diagrams to indicate that the child receives all of the attributes, operations, and relationships that are defined in the parent.

To comply with UML semantics, the model elements in a generalization relationship must be the same type.

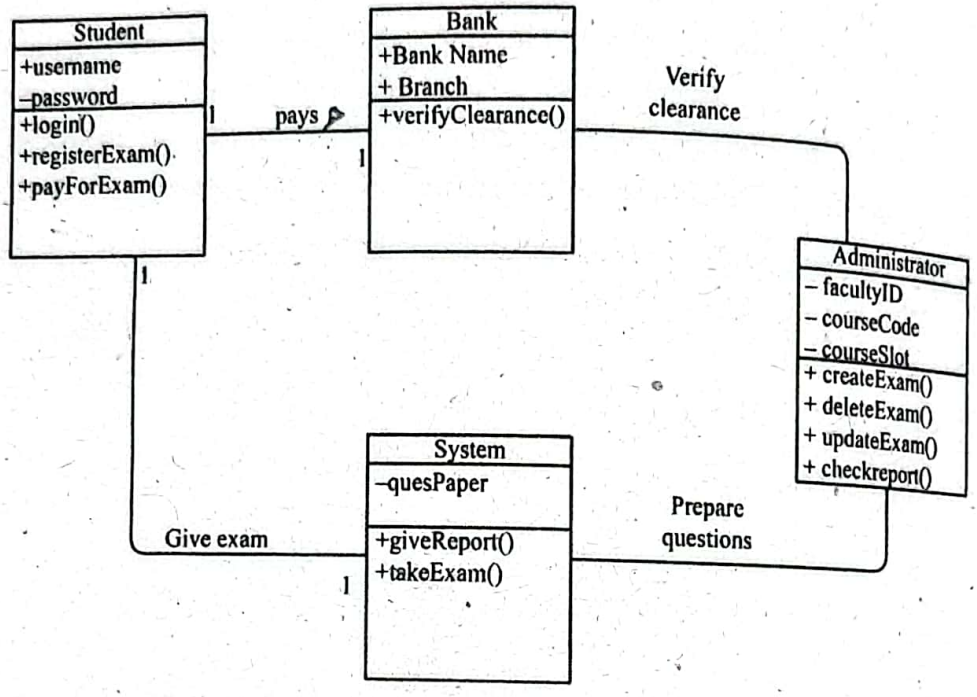


5. **Dependency:** A dependency is a relationship that exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around). Here, Class1 depends on Class2. It is represented by a dashed line with an open arrow.

6. **Identify the different classes, relationships and draw a Class Diagram for "<u>Online Examination System</u>" as following: Students can login to the system with the username and password provided to them. Students can be new or existing. Students can register for the exam through the online form provided in the system. Students are requested to pay for the examination after the registration of the exam. Payments for the registration are accepted through the Bank. Bank verifies the clearance of the fee. Administrator manages the system through faculty id, course code, course slot, create exam, delete exam, update exam and check report. Question paper provided to students through the system. Students can give exams and check reports after exams.**

[8][2078 Chaitra]

⇒



7. **Identify conceptual class and its supportive attributes for a Photocopier machine from the description given below and draw the conceptual class diagram for the same. Initially the machine is off. When the operator switches on the machine, it first warms up during which it performs some internal tests. Once the tests are over, the machine is ready for making copies. When the operator loads a page to be photocopied and presses the 'start' button, the machine starts making copies according to the number of copies selected. When the machine is making copies, the machine may run out of paper. Once an operator loads sufficient pages, it can start making copies again. During the photocopy process, if paper jam occurs in the machine, the operator may need to clean the path by removing the jammed paper to make the machine ready.**

[9] [74 Bhadra]

⇒ The conceptual classes and their attributes are along with their attributes are listed below:

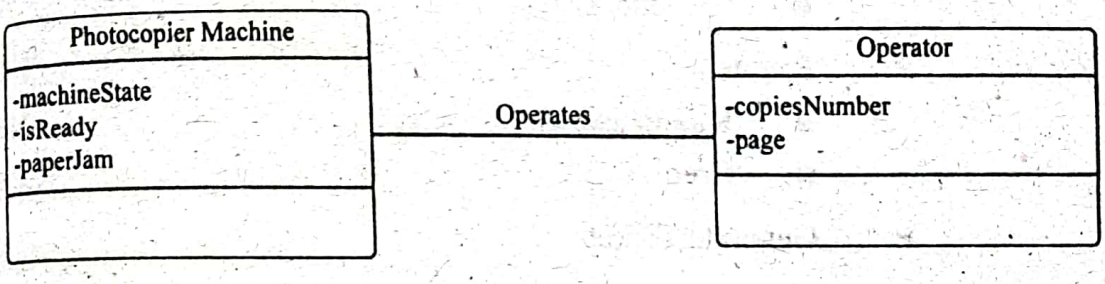
a) Photocopier Machine

- machine State
- is Ready
- paperjam

b) Operator

- page
- copies Number

Thus, the conceptual class diagram from the given properties is drawn below.



8. How does static and dynamic analysis differ in OOAD? Explain in brief. [5] [2073 Magh]

⇒

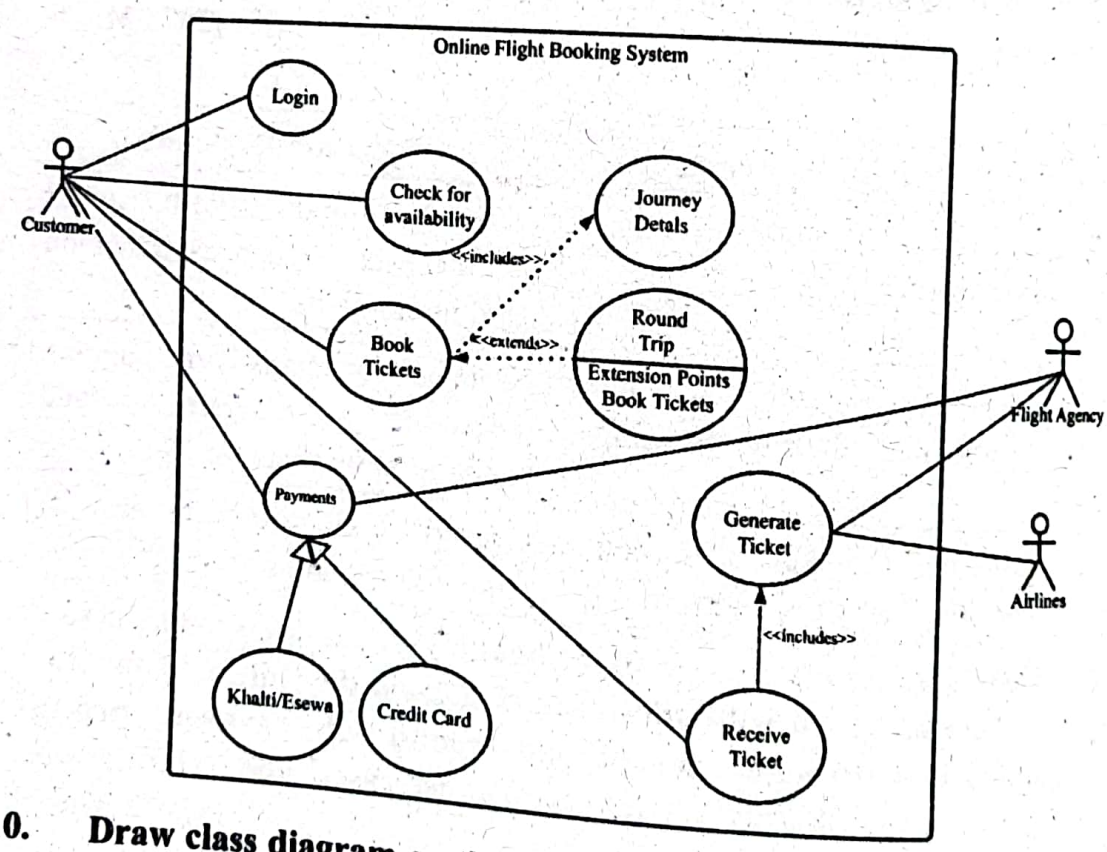
Static Modeling	Dynamic Modeling
Describes the static structure of the system.	Describes the dynamic behaviour of the system such as transition and interactions.
Consists of set of objects (classes) and their relationships.	Consists of statechart diagram, sequence diagram and collaboration diagram.
Describes the skeletal framework and structure of the system.	Describes the system in terms of interactions and states.

9. Consider the scenario of flight booking through an online system at first and then with allowance of certain days of delay for final purchase through the second round of payment process completion. Identify all the actors, use-cases, and relationships. Also draw use case diagram. [8] [2073 Magh]

⇒ For online flight booking system, all the components are as follows:

- Actors
  - Customer
  - Flight Agency
  - Airlines
- Use-cases
  - Login
  - Check for availability

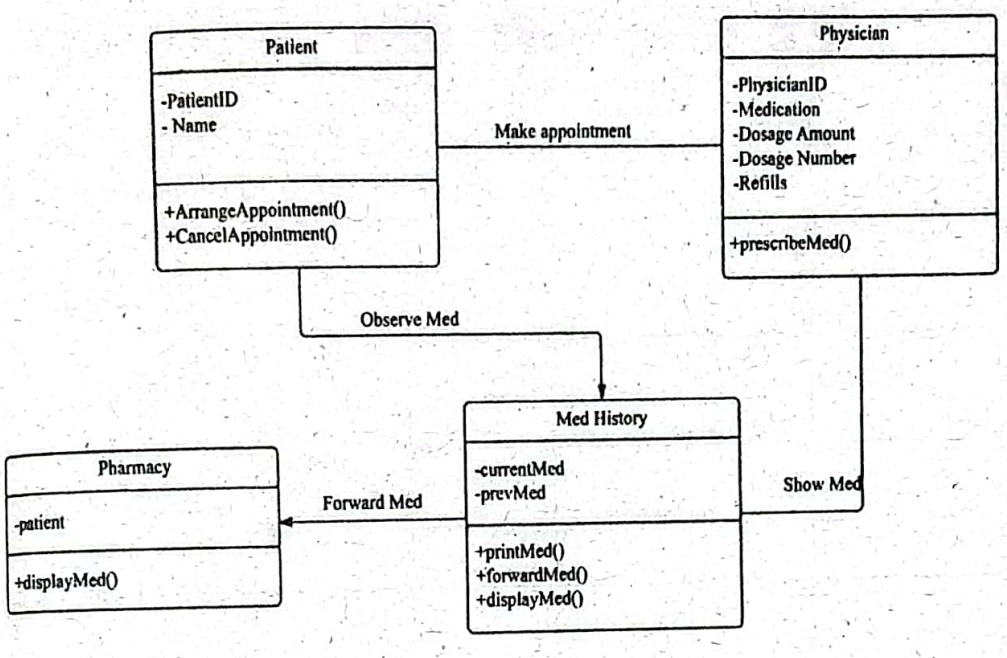
- Booking Tickets
- This requires / includes
  - Journey Details
    Customers may book tickets for Round Trips as well .(extends)
  - Round Trip
- Payments
  - Khalti/Esewa
  - Credit Card
- Generate Ticket
- Receive Ticket



10. Draw class diagram on the following scenario operating condition system:

Patients can arrange and cancel appointments with physicians using a scheduler. Physician succeeded in prescribing Medication for the patient. Physician specificies Drug Info: Medication name, Dosage amount, Number dosages and Refills. Computer cross checks for conflict between Medication and current Medications / Medicals history prescription forwarded electronically to Pharmacy and printed for patient as well. **[5+5] [73 Magh]**

⇒ **Class Diagram:**

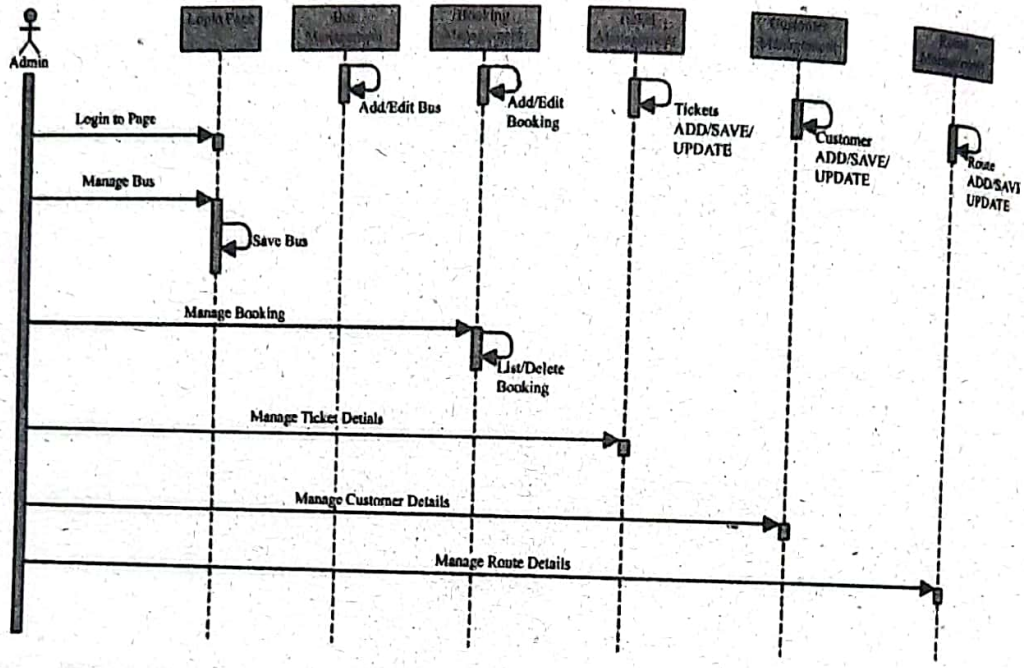


11. **In what aspects does the sequence diagram is different from the collaboration diagram? Prepare sequence diagram of bus ticket reservation system.** **[2+6][2073 Magh]**

⇒ **Part 1:**

Sequence Diagrams	Collaboration Diagrams
The sequence diagram represents the UML, which is used to visualize the sequence of calls in a system that is used to perform a specific functionality.	The collaboration diagram also comes under the UML representation which is used to visualize the organization of the objects and their interaction.
The sequence diagrams are used to represent the sequence of messages that are flowing from one object to another.	The collaboration diagrams are used to represent the structural organization of the system and the messages that are sent and received.
The sequence diagram is used when the time sequence is the main focus.	The collaboration diagram is used when object organization is the main focus.
The sequence diagrams are better suited for analysis activities.	The collaboration diagrams are better suited for depicting simpler interactions of the smaller number of objects.

Part 2: The sequence diagram for Bus ticket reservation system:



12. Explain the concepts of Controller and Polymorphism as per the definition outlined from GRASP design pattern with illustration.

[8][73 Magh]

⇒ Refer to the theory part.

13. How can you represent the dynamic behavior of a system in Object Oriented Analysis (OOA)? Explain with an example.

⇒ The dynamic model represents the time-dependent aspects of a system. It is concerned with the temporal changes in the states of the objects in a system. The main concepts are –

- State, which is the situation at a particular condition during the lifetime of an object.
- Transition, a change in the state
- Event, an occurrence that triggers transitions
- Action, an uninterrupted and atomic computation that occurs due to some event, and
- concurrency of transitions.

A state machine models the behavior of an object as it passes through a number of states in its lifetime due to some events as well as the actions occurring due to the events. A state machine is graphically represented through a state transition diagram.

States and State Transitions

State

The state is an abstraction given by the values of the attributes that the object has at a particular time period. It is a situation occurring for a finite time period in the lifetime of an object, in which it fulfils certain conditions, performs certain activities, or waits for certain events to occur. In state transition diagrams, a state is represented by rounded rectangles.

Parts of a state

- **Name** – A string differentiates one state from another. A state may not have any name.
- **Entry/Exit Actions** – It denotes the activities performed on entering and on exiting the state.
- **Internal Transitions** – The changes within a state that do not cause a change in the state.
- **Sub–states** – States within states.

Initial and Final States

The default starting state of an object is called its initial state. The final state indicates the completion of execution of the state machine. The initial and the final states are pseudo-states, and may not have the parts of a regular state except name. In state transition diagrams, the initial state is represented by a filled black circle. The final state is represented by a filled black circle encircled within another unfilled black circle.

Transition

A transition denotes a change in the state of an object. If an object is in a certain state when an event occurs, the object may perform certain activities subject to specified conditions and change the state. In this case, a state–transition is said to have occurred. The transition gives the relationship between the first state and the new state. A transition is graphically represented by a solid directed arc from the source state to the destination state.

The five parts of a transition are –

- **Source State** – The state affected by the transition.
- **Event Trigger** – The occurrence due to which an object in the source state undergoes a transition if the guard condition is satisfied.

- **Guard Condition** – A Boolean expression which if True, causes a transition on receiving the event trigger.
- **Action** – An un-interruptible and atomic computation that occurs on the source object due to some event.
- **Target State** – The destination state after completion of transition.

14. In many ways, a deployment diagram is just a special kind of class diagram, which focuses on a system's nodes. Justify this statement. **[5] [72 Ashwin]**

⇒ Deployment diagrams are used to visualize the topology of the physical components of a system, where the software components are deployed. Deployment diagrams are used to describe the static deployment view of a system. Deployment diagrams consist of nodes and their relationships.

The term Deployment itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components, where software components are deployed. Component diagrams and deployment diagrams are closely related.

Component diagrams are used to describe the components and deployment diagrams show how they are deployed in hardware. UML is mainly designed to focus on the software artifacts of a system. However, these two diagrams are special diagrams used to focus on software and hardware components. Most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on the hardware topology of a system.

15. Where to Use Deployment Diagrams?

⇒ Deployment diagrams are mainly used by system engineers. These diagrams are used to describe the physical components (hardware), their distribution, and association. Deployment diagrams can be visualized as the hardware components/nodes on which the software components reside.

Software applications are developed to model complex business processes. Efficient software applications are not sufficient to meet the business requirements. Business requirements can be described as the need to support the increasing number of users, quick response time, etc.

Therefore, a deployment diagram models the system in terms of deployable units. It clearly defines the relationships between the deployed components. In the same way a class diagram shows the relationship between the attributes of the classes. A deployment diagram can be said also as a special type of class diagram.

16. Compare the following: **[2072 Ashwin]**

a) Flowchart vs Structure chart

⇒ **Flow Chart:** Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing. Flow chart is a convenient technique to represent the flow of control in a program.

Structure Chart: Structure Chart represents the hierarchical structure of modules. It represents the software architecture that means the various modules making up the system and the dependency. Structure chart representation can be easily implemented using some common programming language. The main focus in the structure chart is on the module structure of the software.

b) Structure Model vs Implementation Model

⇒ Structural modeling captures the static features of a system. The static model describes the framework of the system. They consist of the following –

- Classes diagrams
- Objects diagrams
- Deployment diagrams
- Package diagrams
- Composite structure diagram
- Component diagram

Structural models represent the framework for the system and this framework is the place where all other components exist. Hence, the class diagram, component diagram and deployment diagrams are part of structural modeling. They all represent the elements and the mechanism to assemble them.

The structural model never describes the dynamic behavior of the system. Class diagram is the most widely used structural diagram.

Behavioral Modeling

Behavioral model describes the interaction in the system. It represents the interaction among the structural diagrams. Behavioral modeling shows the dynamic nature of the system. They consist of the following –

- Activity diagrams
- Interaction diagrams
- Use case diagrams

17. Explain all types of external actors in relation to System under Discussion (SuD). Draw a system sequence diagram for the Library Management system with the following requirements. You can add additional elements if necessary.

A college library has 4 librarians to manage and issue the books to the users who are either students or faculty staff. The library contains the books belonging to Computer and Humanities streams. The books are course books, reference books, book banks etc. The users must log into the system to search for the required books and may reserve the books earlier. The librarian issues the books to the users and also charges a fine in case of delayed return or loss of the book. The librarian asks for the "Sanu Publisher" to supply the necessary books into the library. The librarian manages all the users. **[2+6][72 Ashwin]**

⇒ **Part 1:** Actors can be people, other systems, time triggers, or event triggers.

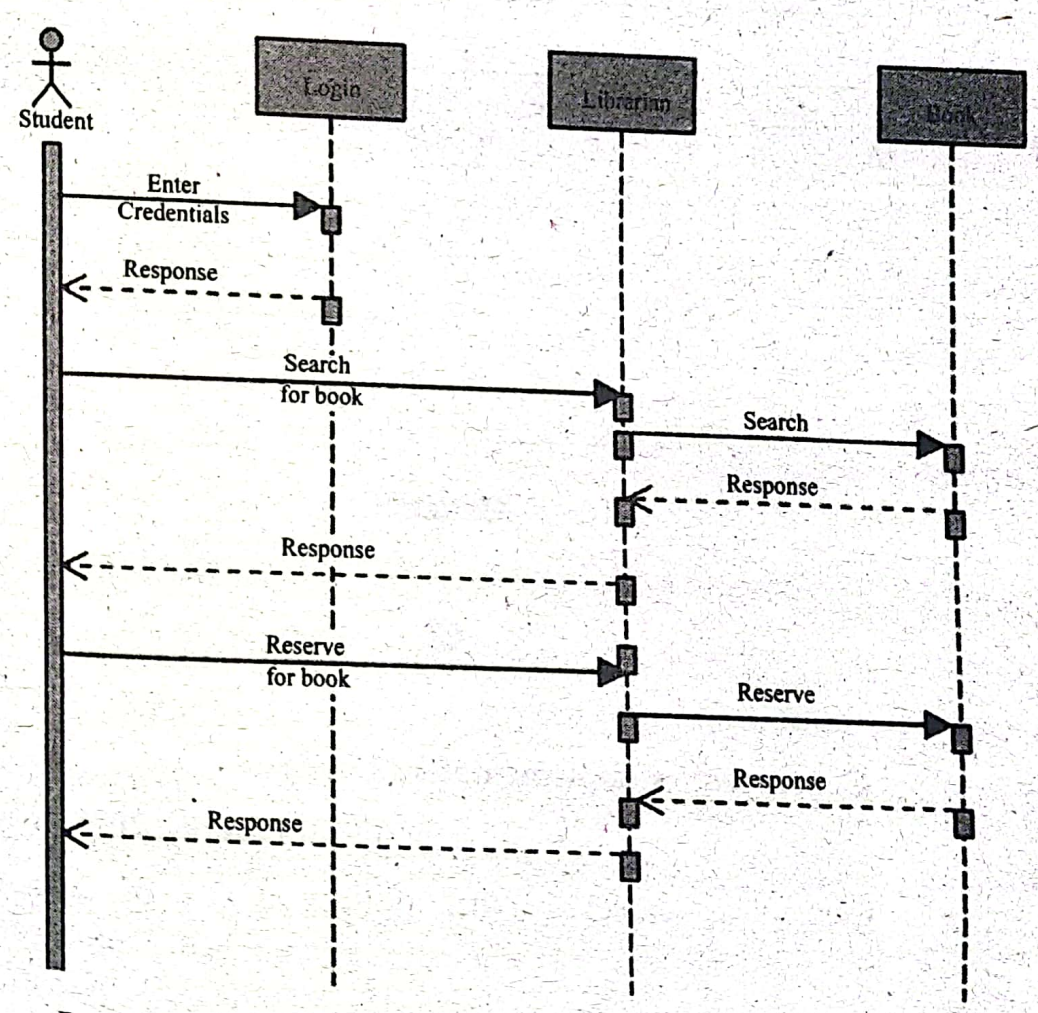
An actor specifies a role played by a user or any other system that interacts with the subject. It may represent roles played by human users, external hardware, or other subjects. Actors are always outside the system and interact directly with it by initiating a use case, providing input to it, and/or receiving outputs from it. Based on the fulfillment of the service provided by the system the actors or external entities can be categorized as follows:

a. **Primary Actor:** a user whose defined user goal is fulfilled by the system.

The primary actor of a use case is the stakeholder that calls on the system to deliver one of its services. It has a goal with respect to the system – one that can be satisfied by its operation. The primary actor is often, but not always, the actor who triggers the use case.

b. **Supporting Actors:** a user who provides a service (e.g information) to the system. A supporting actor (also known as a secondary actor) in a use case in an external actor that provides a service to the system under design. It might be a high-speed printer, a web service, or humans that have to do some research and get back to us.

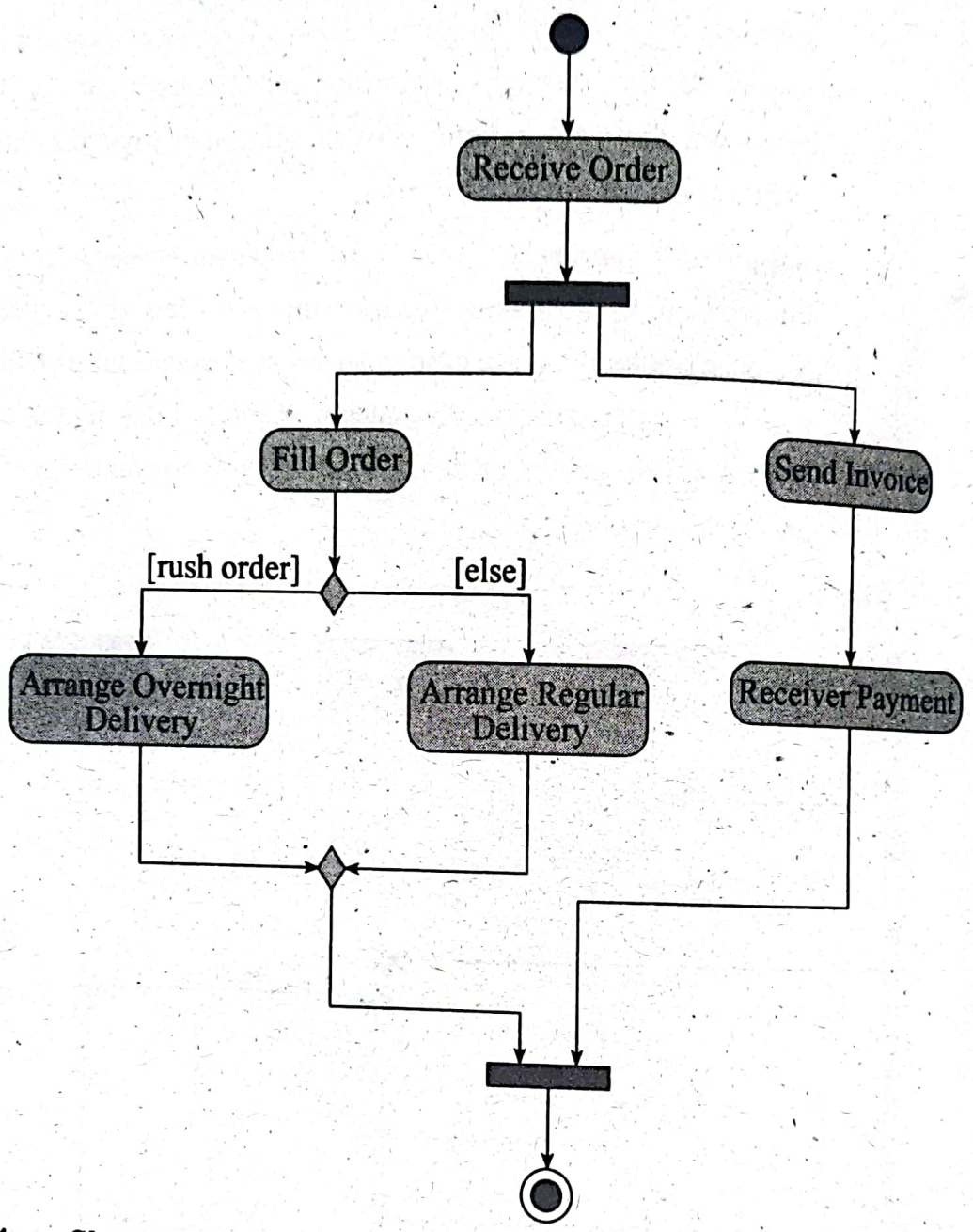
Part 2:



18. **Prepare a list of essential components to be identified in building an activity diagram. Illustrate with an example of your own choice.** **[2070 Magh]**

⇒ **Part 1:** Refer to the theory section.

Part 2: Activity diagram of Process Order:



14. Short Notes

a) Conceptual vs Implementation Model:

[2078 Chaitra, 2075 Bhadra]

⇒ Conceptual model is the abstract model of a system, representing only the required attributes and methods. They also depict only the main relationship between the classes. Conceptual models should help you to better analyze, understand and communicate about the system. A good programmer should be able to directly implement the conceptual model into source code.

On the other hand, implementation models are the code structure implemented using any object oriented programming language. Implementation models tend to be more precise, lengthy and carry all the logical functions and attributes of the real world scenario. Implementation models are much longer to do and are only a visualization of the source code.

b) CRC Cards **[2078 Chaitra]**

⇒ Refer to the theory part.

c) Association types vs Cardinality Constraints **[2075 Bhadra]**

⇒ **Association Types:** *Refer to theory part*

Cardinality Constraints: The cardinality constraints refer to the multiplicity between the association / relationship between classes. Multiplicity notations are placed near the ends of an association. These symbols indicate the number of instances of one class linked to one instance of the other class. For example, one company will have one or more employees, but each employee works for one company only.

Multiplicity examples:

1	Exactly one
0..1	Zero or one
*	Many
0..*	Zero or Many
1..*	One or Many

d) Design Patterns vs Components reuse **[2075 Bhadra]**

⇒ **Design Patterns:** ***Refer to theory part***

Components Reuse: Most Object-Oriented design and programming centers around reuse and reusable code (classes, methods, etc). Some types of reuse found in software design:

- integration of off-the-shelf components
- use of standard and custom class libraries
- use of design patterns to solve common problems and protect specific classes from future change
- use of application frameworks

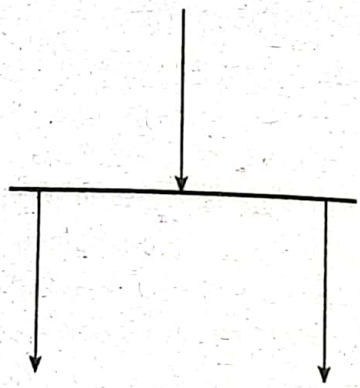
With reuse, developers don't have to unnecessarily put effort on the fundamental components required to build the system. They don't have to make it from scratch. It promotes efficiency, product optimization and faster delivery.

e) Design patterns and its use in OOAD. **[2074 Bhadra]**

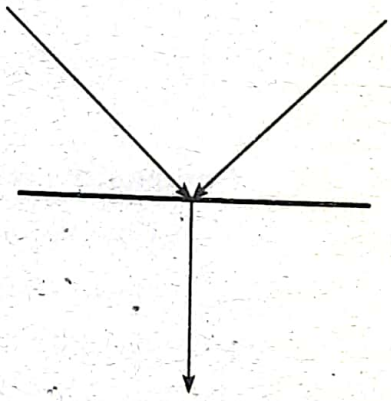
⇒ *Refer to theory part*

f) Forking vs Joining **[2073 Magh]**

⇒ Forks and join nodes generate the concurrent flow inside the activity. A fork node consists of one inward edge and several outward edges. It is the same as that of various decision parameters. Whenever data is received at an inward edge, it gets copied and split across various outward edges. It split a single inward flow into multiple parallel flows.



Join nodes are the opposite of fork nodes. A Logical AND operation is performed on all of the inward edges as it synchronizes the flow of input across one single output (outward) edge.



h) Swimlanes in activity diagram **[2073 Magh]**

⇒ Swimlanes are used to show which activities are performed by which organization in the activity diagram. The lanes are boundaries are drawn and the activities of a particular organization are drawn in the same lane as that of the organization. Swimlanes have to be ordered in a Logical Manner. It is suggested to have less than five swimlanes in an activity diagram. Swimlanes are good in that they combine the activity diagram's depiction of logic with the interaction diagram's depiction of responsibility.

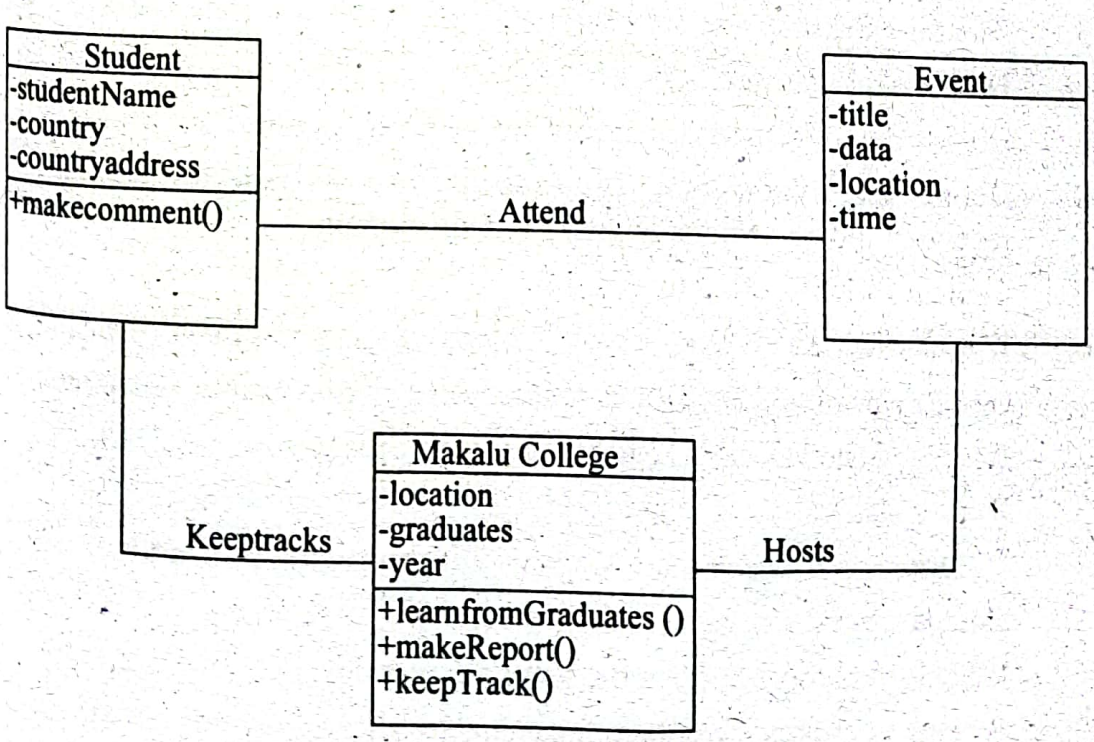
15. **Describe the term Design pattern? Explain the purpose and benefits of *Information-Expert, Creator, Controller* and *Polymorphism* design patterns defined by *GRASP*.**

⇒ Refer to the theory part.

16. **For the case study given below identify all classes, their relationships, attributes and methods for each class. Also draw class diagram using standard UML Notation. [2+6]**

Makalu College operates international business in 10 locations throughout the globe. The college had its first 9000 graduates in 2010. The second keeps track of each student's name, country of birth, and current address. In order to maintain strong ties to its alumni, the school holds various events around the world. Events have title, date, location and time. The school needs to keep track of which graduates have attended which events. For an attendance by a graduate at an event, a comment is recorded about information. School officials learn from that graduate at that event. As with the events, school records information learned from graduates. When an official knows that he or she will be meeting or talking to graduate, a report is produced showing the latest information about the graduate and the information learned during the past two years from that graduate from all contacts and events the graduate attended.

⇒



Practice Questions

1. Explain Object-Oriented Design(OOD) in brief. What are the inputs for Object-Oriented Design?
2. What are the important concepts under Object-Oriented Design as well as Programming?
3. Explain about static and dynamic modelling?
4. What are CRC cards?
5. Describe in short about all the components that fall under Use-Case diagrams.
6. What are the types of relationships between use cases?
7. What is a sequence diagram? Explain in detail about its terms and notations.
8. How do you apply loop and conditional statements in a sequence diagram?
9. Draw sequence diagram for the following systems:
   a. Library system
   b. Online/Virtual class
   c. Movie ticket booking
10. Explain about the basic terminology and notations used in Collaboration or Communication diagrams.
11. What is a class diagram? What are Design Class Diagrams?
12. What are the types of relationships in class diagrams?
13. What are cardinality constraints?
14. Give an example of a class diagram that can be used to model Online Entrance Examination System.
15. What is an association class?
16. Define deployment diagram.
17. Draw a deployment diagram for Client-Server Architecture.
18. Define activity diagram. Draw an activity diagram for sending / receiving money using a Digital payment method.
19. Explain forking and joining.
20. What are swimlanes? Draw a swimlane for Changing Passwords via email.
21. Describe briefly about all the notations used in the Activity Diagram.
22. What is a State Machine Diagram? Briefly describe its terms and notations.
23. Draw a state machine diagram for User Authentication Process.
24. Differentiate between State Machine Diagram and Flowchart Process.
25. Compare and contrast class diagram and object diagram.
26. What is visibility? Explain the types of visibility ..
27. What are patterns ? What are the characteristics of a good pattern?

28. What are GRASP patterns?

29. Write in short about Information Expert, Creator and Controller Patterns.

30. Consider the Library management system. It should provide the facility to issue the book, to calculate the fine during book return, the placement of the books on the shelfs, adding new books to the shelfs and removing old books from the shelfs. Draw the activity diagram with swim lanes for each and every components of the above scenario.

31. Consider a House keeping system in a five star hotel. Draw a sequence and collaboration diagram for the given scenario.

32. To buy a book electronically from chapters.com, a customer needs to select the book from a list provided by Chapters' e-Commerce system, provide credit card information to the system, then the system gets authorization from the bank for the payment, and – if positive – confirms the sale. The order is then sent to the orders department and when the book becomes available, it is shipped to the customer. Also, the order department charges the customer's credit card by informing the bank of the amount. Draw a sequence diagram that models this process. Make sure that the model has all relevant actors and the interactions between them. Do show explicitly the time intervals when different actors actively participate in the modelling process.

33. A library has books, journals and laptops. Undergraduate students can borrow only books and journals. Post graduate students can borrow only journals and laptops. A maximum of three books, two journals and one laptop will be issued per student. If they return these items beyond ten days, a fine of Rs. 10 per day is levied for each book/journal. However, for late return of laptop a fine of Rs. 1,000 is levied. The librarian also periodically takes stock of books, journals and laptops and generates the report of lent out items and lost items. Construct the necessary activity diagrams for the operations carried out by the librarian.

34. Consider an elevator that has the basic functions such as moving up and down open and close doors and pick up passengers. The elevator is supposed to be used in a building having floors numbered from 1 to n. There are cll 1 and n, there are two floor call buttons for the passengers to call elevator for going up and down. There is only one down call button at floor n and one up call button in floor 1. Then the car stops at a floor, the doors are opened and the elevator light indicating the current direction the elevator is going is illuminated so that the passengers can get to know the current moving direction of the elevator. When the elevator is moving a music audio is played inside the elevator.

Draw class diagram, activity diagram and component diagram for designing this system.

35. Model a class diagram a "Banking system". State the functional requirements you are considering.

36. Model a use case diagram for the following scenario:

Deepthi super market wants a subsystem to process supply orders via the Web. The user will supply via a form their name, passowrd, account number and a list of supplies along with an indication of the quantities desired. The subsystem will validate the input, enter the order into a database and generate a receipt with the order number, expected ship date and the total cost of the order. If the validation step fails, the subsystem will generate an error message describing the cause of the failure.

37. Consider an elevator that has the basic functions such as moving up and down open and close doors and pick up passengers. The elevator is supposed to be used in a building having floors numbered from 1 to n. THere are call buttons in the elevator corresponding to each floor. For every floor except floors 1 and n, there are two floor call buttons for the passengers to call elevator for going up and down. There is only one down call butto at floor n and one up call button in floor 1. Then the car stops at a floor, the doors are opened and the elevator light indicating the current direction the elevator is going is illuminated so that the passengers can get to know the current moving direction of the elevator. When the elvator is moving a music audio is played inside the elevator.

Draw class diagram, activity diagram and component diagram for designing this system.

38. Library Management System (LMS):

Problem statement:

Library management software is used to monitor and conrolling the trasactions in a library. This case study on the Library Management System gives us the complete information and daily transaction done in a library. We need to record and retrieve the details of books o=in the library which mainly focus on basic operation like adding a new book, new member, searching books, facility to borrow books and returns.

Perform the following functions:

i. Class diagram.
ii. Use-case diagram.
iii. Sequenec diagram
iv. Collaboration diagram.
v. Activity diagram.

❑❑❑

Chapter 4 Implementation

4.1 Introduction

"Without requirements or design, programming is the art of adding bugs to an empty text file."

With the requirements and completion of the design diagrams, there exists sufficient detail to generate code representing the objects. The UML diagrams, the interaction diagrams and DCDs, created during the design work, will be used as input to the code generation process. While the term "coding" implies the mechanical translation of a pre-existing design into a computer language, "implementation" (or "software construction") is far from mechanical and requires considerable creativity and judgment.

The Unified Process (UP) defines the term "Implementation Model". The implementation model is a collection of components, and the implementation subsystems that contain them. Components include both deliverable components, such as executables, JSP/XML/HTML pages, and components from which the deliverables are produced, such as source code files, database definitions, and so forth. Thus, the code generation process in this chapter is part of the Implementation Model.

4.2 Programming and Development Process

The creation of code in an object-oriented programming language is not part of OOAD; it is an end goal. The artifacts created in the UP-Design Model provide some of the information necessary to generate the code. The strength of use cases along with OOAD and OO programming is that it provides an end-to-end roadmap from requirements and artifacts through to code.

Some decision-making and creative work was accomplished during design work. However, in general, the results generated during design are an incomplete first step; during programming and testing, many changes will be made and detailed problems will be discovered and resolved. Thus, the concept that programming is based on prior design work does not imply that no prototyping occurs while programming. In fact, modern development tools provide an excellent environment for exploring new approaches, such that concurrent design & programming is usually worthwhile. Thus, creativity and change are considered during implementation, especially in iterative and

evolutionary development approaches, where results of a prior iteration can be fed into the beginning of the next iteration such that subsequent analysis and design results are continually being refined and enhanced.

Additionally, it is desirable for the diagrams generated during design to be semi-automatically updated to reflect changes in the subsequent coding work. Ideally this should be done with a CASE tool that can read source code and automatically generate, for example, package, class, and sequence diagrams. This is an aspect of reverse-engineering, the activity of generating diagrams from source (or sometimes, executable) code.

4.2.1 Software Engineering

Software Engineering is a collection of techniques, methodologies and tools that help with the production of

- high quality software system
- with a given budget
- before a given deadline

Software engineering is an engineering discipline that is concerned with all aspects of software production. The four fundamental activities in a software process are:

- **Software specification**, where customers (and engineers) define the software that is to be produced and the constraints on its operation.
- **Software development**, where the software is designed and programmed (implemented).
- **Software validation**, where the software is checked to ensure that it is what the customer requires.
- **Software evolution**, where the software is modified to reflect changing customer and market requirements.

Actually, object-oriented software development comprises of following stages:

- Requirements elicitation
- Analysis
- System design
- Object design
- Implementation
- Testing

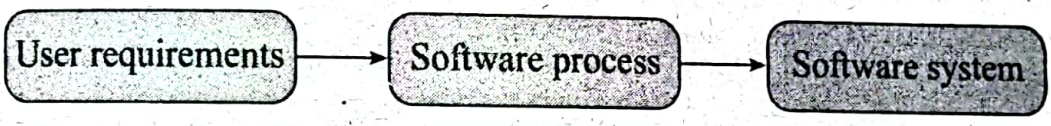
4.2.2 Unified Software Development Process

A software development process is the set of activities needed to transform a user's requirements to a software system. The Unified Software Development Process (USDP) is a software development process which is characterized by Use-case driven, Architecture centered and Iterative and incremental.

There are many different types of requirements.

- Functional requirements
- Data requirements
- Performance requirements
- Capacity requirements
- Accuracy requirements
- Test/Robustness requirements
- Maintainability, extensibility, portability, etc.

All of these requirements drive use-cases and architectures.



A software system should be used by the user. Thus the developer of the system must know the users' needs. The term user refers not only to human users but also to other system which interacts with the system being developed. An interaction from/to the user is a use-case. A use-case is a piece of functionality in the system which captures a requirement.

The role of software architecture is similar in nature to the role of architecture plays in building construction. A plan of building is looked at from various viewpoints, such as structure, services, heat conduction, electricity. This allows the builder to see a complete picture before actual construction. The software architecture must be influenced by the requirements of both use-case dependent and use-case independent. Architecture is not a framework.

The Unified Modeling Language (UML) has various diagrams. Not the all diagrams must be used for the software development. But it must be stressed that many of them are quite useful for the Object-Oriented software design and implementation. A software process has four steps, Object-Oriented Analysis, Object-Oriented Design, Implementation and Test. These steps must be taken many turns (so-called spiral approach) instead of once (so-called water fall approach). The first turn should be the "proof of concepts" demonstration. User's requirements document and diagrams should be regularly updated during the turns. All of these is conveyed by the Unified Software Development Process.

4.3 Mapping Design to Code

Implementation in an object-oriented programming language requires writing source code for:

- Class and Interface definitions
- Method definitions

In this chapter, C++ is used for the coding examples because of its widespread use and familiarity. However, this is not meant to imply a special endorsement of C++; C#, Java, Visual Basic, Smalltalk, Python, and many more languages can be used.

4.4 Creating Class Definitions from Design Class Diagrams

Design Class Diagrams (DCDs) depict the class or interface name, superclasses, method signatures, and simple attributes of a class which is sufficient to create a basic class definition in an object-oriented programming language.

The major sections of a sample UML class diagram are as:

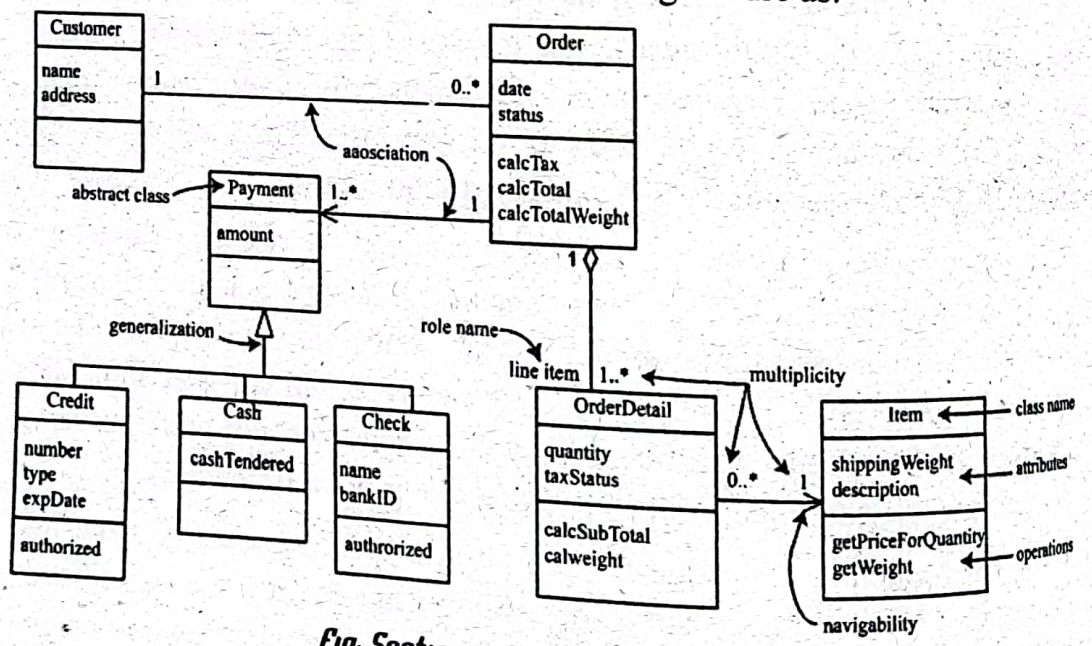


Fig: Sections of a UML Class diagram

4.4.1 Classes in Object Oriented Programming

Object-Oriented Programming (OOP) is a data-centered view of programming in which data and behavior are strongly linked. Data and behavior are conceived of as classes whose instances are objects. OOP also views computation as simulating behavior. What is simulated are objects represented by a computational abstraction.

The term abstract data type (ADT) means a user-defined extension to the native types available in the language. ADT consists of a set of values and a collection of operators and methods that can act on those values. Class objects are variables of an ADT. OOP allows ADT to be easily created and used. For example, integer objects, floating point number objects, complex number objects, four momentum objects, etc., all understand addition and each type has its own way (implementation) of executing addition. An ADT object can be used in exactly same manner as a variable of native type.

4.4.1.1 Defining Classes

The fundamental program structure in an object oriented program is the **object**. The class mechanism in object-oriented languages provides a way to create such objects. The concept of a **class** is similar to ADT in that it can be used to create objects of types that are not directly supported by that language. An **attribute** is a property that we associate with an object; it serves to describe the object and holds some value that is required for processing. Before such objects are created, we create classes that serve as blue-prints for those objects as a whole. For example, the classes for students, teacher and course objects is created as:

```
class Student {
  //code
};
class Teacher {
  //code
};
class Course {
  //code
};
```

The above definitions show how to create three classes, without giving any details. The process of creating an object using these classes– also called *instantiation*–can be done in different ways as shown below:

```
//static initialization
className objectName;

//dynamic initialization
className *objectName = new className();
//delete objectname;

//copy constructor initialization
className object1;
className object2 = object1; //or
className object2(object1);
```

4.4.1.2 Encapsulation

Encapsulation consists of the internal implementation details of a specific type and the externally available operators and functions that can act on objects of that type. The implementation details should be inaccessible to client code that uses the type. It is mandatory to make data members private and provide public Set/Get methods accessible to them. Also, make all Get and other methods which do not modify any data member "cosnt". The "const" methods can be accessed even for constant objects. Strict use of constant objects allows you the safe programming.

4.4.1.3 Implementing Classes

Here, we implement the attributes and methods for the classes. Specifically taking the `Student` class that we initially declared, we add the details as required. We would give each student a name as `String`, roll no. as `int`, etc. all of which are the attributes(*member data*) of the `Student` class defining the behavior of each object. Also, we need to add methods or operations to perform on those data which is referred as *member function* such as `setName()`, `getName()`, etc.

```
class Student {
  private:
      String name;
      int rollNum;
  public:
      void setName(String stName){
      name = stName;
      }
      String getName(){
      return name;
      }
};
```

To illustrate this further, consider two objects of type `Student`.

```
void main {
      Student student1;
      Student student2;
      student1.setName("Jon");
      student2.setName("Jane");
      cout << student1.getName() << endl;
      cout << student2.getName() << endl;
};
```

4.4.1.4 Constructors

The Student class has a method for setting the name of a student. Here we set the name of the student after creating the object. Since every student has a name, when we create a student object, it would be convenient to store the student's name in the object as we create the student object. For this, a constructor can be used. A constructor in C++ is a special *member function* having the same name as that of its class which is used to initialize some valid values to the data members of an object. It is executed automatically whenever an object of a class is created.

There are 3 types of constructors in C++, *Default Constructor*, *Parameterized Constructor* and *Copy Constructor*. For the class Student we can write the following constructor along with other member functions as:

```
Student(){
        //default constructor
 }
Student(String stName){
                //parametrized constructor
        name = stName;
}
Student(Student &student){
                //copy constructor
        name = student.name;
}
```

4.4.1.5 Static Members

The static member variable is member variable that belong to the class rather than objects of the class. If the static member variables are public, we can access them directly using the class name and the scope resolution operator. But if the static member variables are private, static member function is used to access it. Like static member variables, static member functions are not attached to any particular object. Here is an example with a static member function accessor:

```
class className {
  private:
        static int s_value;
  public:
                static void setValue(int val){
        s_value = val;
                }
        static int getValue(){
        return s_value;
        }
};
void main {
        className::setValue(2080);
          cout << className::getValue() << endl;
}
```

Because static member functions are not attached to a particular object, they can be called directly by using the class name and the scope resolution operator. Like static member variables, they can also be called through objects of the class type.

4.4.1.6 Abstraction and Polymorphism

Abstraction and Polymorphism enable "Rapid Prototyping". High level class diagrams and scenario diagrams should be made first before going to the detailed design/implementation of actual concrete classes." Proof of concepts" demonstration must be done with just a couple of concrete classes (or just one dummy concrete class) for each abstract base class. Abstraction and polymorphism localizes responsibility for an abstracted behavior. They also help the modularity and portability of the code.

Polymorphism has lots of forms.

- Function and operator overloading
- Function overriding
- Parametric polymorphism

In C++, an operator is overloadable. A function or an operator is called according to its signature, which is the list of argument types. If the arguments to the addition operator are integral, then integer addition is used. However, if one or both arguments are floating point, then floating point addition is used. Operator overloading helps the readability. Also, using virtual member functions in an inheritance hierarchy allows run-time selection of the appropriate member function. Such functions can have different implementations that are invoked by a run-time determination of the subtype, which is called virtual method invocation or dynamic binding.

4.4.1.7 Interface

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration as follows –

```
class Box {
    public:
        // pure virtual function
        virtual double getVolume() = 0;

    private:
        double length;          // Length of a box
        double breadth;         // Breadth of a box
        double height;          // Height of a box
};
```

The purpose of an abstract class is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

Thus, if a subclass of an abstract class needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the abstract class. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error. Classes that can be used to instantiate objects are called concrete classes.

4.4.2 UML to Class definition

Finally, we are able to transform a given UML class diagram to class definition in our preferred language. A sample UML class symbol is as:

<< abstract >> shape
-id: int #color: String #totalNum: int
+shape(color: String) ±getTotaNum(): int +getArea() = 0: int

Class definition for the above class is shown below:

#Shape.h

```
class Shape {
  private:
    int id;
  protected:
    String color;
    static int totalNum;
  public:
    Shape (String color);
    static int getTotalNum ();
    virtual int getArea () = 0;
};
```

Here the visibility symbols appearing in a UML class diagram correspond to the C++ keywords:

- "-" → "private:"
- "+" → "public:"
- "#" → "protected:"

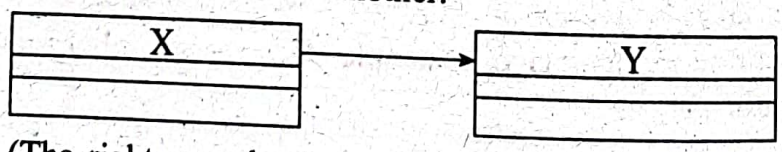
Also, different type of member functions in a C++ class corresponds to operations in class diagram as:

- Static members are represented as <u>underlined</u>.
- Virtual functions are represented as *italics*.

In a system class diagram, the different types of relationship can be implemented to C++ code as:

1. **Association (*knows a*)**

   One object is aware of another.



   (The right arrowhead symbol is optional; a simple straight line will denote association.)

   Code: An object contains a pointer or reference to another object.

```
Class X {
    ...
    Y *y_ptr;
    //or
    Y y;
    ...
};
```

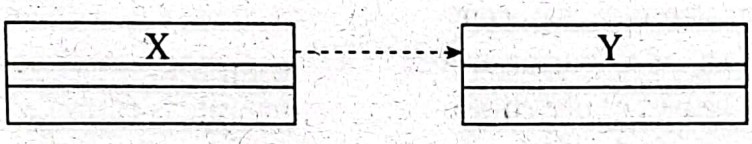
Also, **multiplicity** information can be linked to an association to show how many instances of class Y are linked with instances of class X. It can be implemented using a simple array or C++ STL array, list or any similar structure.

For e.g.,

```
Class X {
   ...
   Y y_instance[2];  //instance of class X
is linked to 2 instance of class Y
   ...
};
```

2. **Dependency (*uses a*)**

One class depends on another but another class may not depend on the first class.

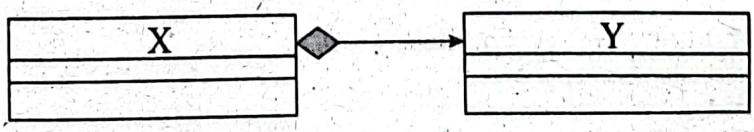


Code: The independent class is a parameter variable or local variable of a method of the dependent class.

```
Class X {
   ...
   void func1(Y y){y.foo();}
   //or
   void func2()    {Y y; y.foo();}
   ...
};
```

3. **Aggregation *(has a)***

A class is collection or container of other classes where contained classes' existence is independent of container class.



It is a weak '*has a*' relationship between **whole** and **part** (*no ownership*). A part object can be created/accessed by others as well. Its life may continue even after the destruction of the whole.

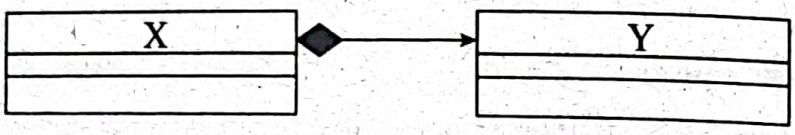
Example: Department has professors. If department is shutdown, they are still professors (they exist).

Code: Association differs from aggregation only in that it does not imply any containment.

```
Class X {
  ...
    Y *y_ptr;
  //or
  vector<Y> y;
  ...
};
```

4. **Composition (uses a/consists of)**

A class is collection or container of other classes where contained classes' existence is dependent of container class.



It is a strong '*has a*' relationship (*ownership*). A composite creates/destroy the component parts. For e.g. a composite 'Car' consists of a component part 'Engine'. In this sense, one object may only be part of one composite; if a composite is destroyed, its parts are destroyed.

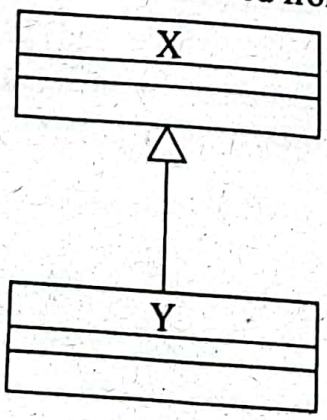
Example: The folder could contain many files, while each File has exactly one Folder parent. If a folder is deleted, all contained files are removed as well.

Code: Similar implementation as aggregation.

```
Class X {
  ...
    Y *y_ptr;
//or
  vector<Y> y;
  ...
};
```

5. **Inheritance (Generalization) (*is a*)**

A class is derived from another class.



Example: Square and Rectangle are derived from same base class Quadrilateral.

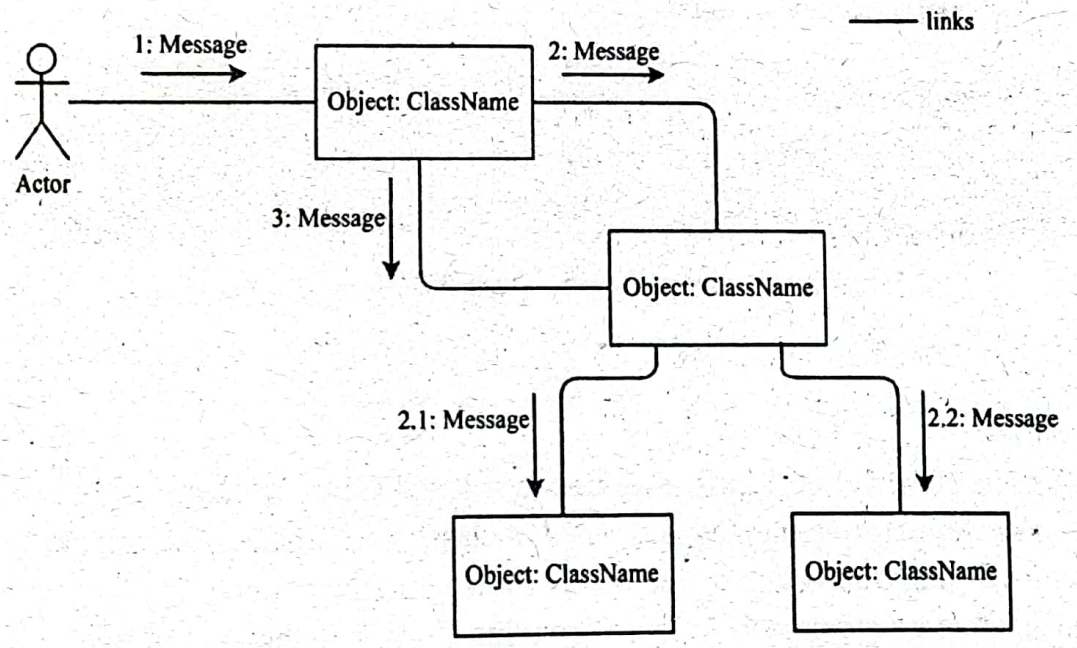
Code: Specified as 'derived_class : <access_specifier> base_class'

```
Class X : public Y
{
  //body

};
```

4.5 Creating Methods from Collaboration Diagram

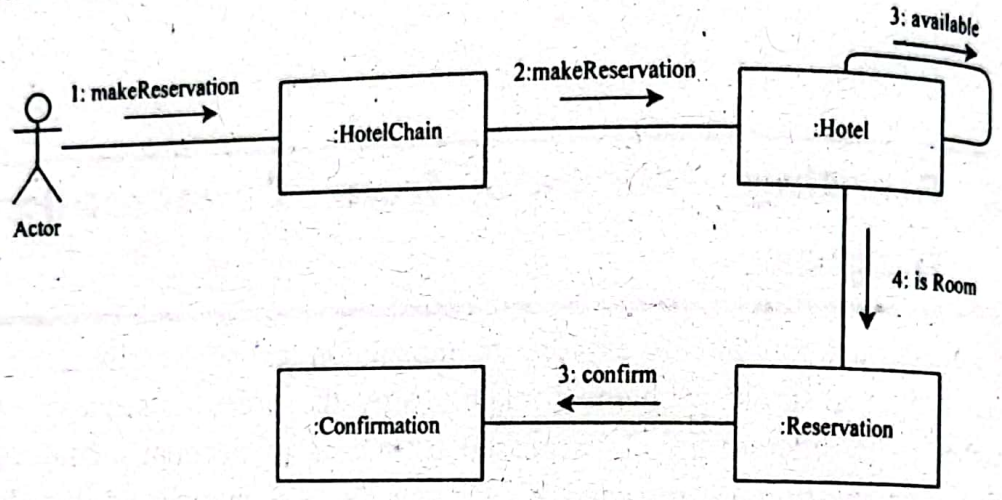
Collaboration diagram aids to explore the interaction and relationship among objects within a single problem domain. Here, the order or sequence of messages is considered and is expressed by means of decimal numbering classification, nesting is indicated by Dewey decimal numbering, but the sequence of time is not considered. The basic components of collaboration diagram are as:



Actually, the major elements in collaboration diagram are: Objects, Links and Messages. Objects are of two types: *Supplier objects*, that supply the method that is being called, and therefore receive the message and *Client objects*, that call methods on supplier objects, and therefore send messages. The connecting lines drawn between objects in a collaboration diagram are *links* which enables us to see the relationships between objects. In fact, associations specified in a class diagram can be represented as links in a collaboration diagram. Messages in collaboration diagrams are shown as

arrows pointing from the Client object to the Supplier object. Typically, messages represent a client invoking an operation or method on a supplier object.

The sequence of the messages in the diagram translates to a series of methods in the class definition. For example, a simplified hotel management collaboration diagram can be shown as:



In this example, *HotelChain* class will be used whose C++ definition is shown as:

```
Class HotelChain
{
 private:
 Hotel *hotel;
 ...
 public:
 void makeReservation(ItemID id, Date date, int
qty) {...}
 ...
}
```

The make Reservation message is sent to a Hotel Chain instance; therefore, the make Reservation method is defined in class Hotel Chain. Similarly, each sequenced message within a method, as shown on the collaboration diagram, is mapped to a statement in the corresponding Class's method definition.

4.6 Container/Collection Classes in Code

It is often necessary for a container of objects in many cases; the need for this is usually reflected as the multiplicity value in a class diagram may be greater than one. For example, a Sale class must maintain visibility to a group of ProductItems instances. In OO programming languages, these relationships

are often implemented with the introduction of an intermediate container or collection, such as a *List* or *Map*, or even a simple *Array*. The class defines a reference pointing to a container/collection instance, which contains instances of the many class-objects. For example, C++'s Standard Template Library (STL) provides collection classes such as *array, vector, list, set, map,* etc.

4.6.1 List

A list is a type of container that is built-in sequentially and allows allocation. Since the list is doubly linked (data can be accessed bi-directionally and sequentially), it makes it possible for the insertion and deletion to be done faster.

For example:

```
#include<list>
int main() {
        std::list<int>my_list={1,3,5,98};
        mylist.push_back(4);
        mylist.push_front(100);
        std::list<int>::iterator it;
        for(it= mylist.begin();it !=
mylist.end();it++) {
                cout << *it <<'\n';
        }
}
```

4.6.2 Set

In C++, sets are used to store and retrieve data from containers. It contains *HashSet* classes which are unordered collections with unique elements.

For example:

```
#include<unordered_set>
int main() {
        std::unordered_set<int> X { 2080, 2081, 2082 };
        for (auto Y: X)
                std::cout << Y << '\n';
        return 0;
}
```

4.6.3 Vector

Vectors in C++ are sequence containers representing arrays that can change their size during runtime. They use contiguous storage locations for their elements just as efficiently as in arrays, which means that their elements can also be accessed using offsets on regular pointers to its elements.

For example:

```
#include <vector>
int main() {
  vector<int> num {1, 2, 3, 4, 5};

  // add the integers 6 and 7 to the vector
  num.push_back(6);
  num.push_back(7);

  cout << "Element at Index 0: " << num.at(0) << endl;
  cout << "Element at Index 2: " << num.at(2) << endl;
  cout << "Element at Index 4: " << num.at(4);

  // change elements at indexes 1 and 4
  num.at(1) = 9;
  num.at(4) = 7;

  // remove the last element
  num.pop_back();

  // declare iterator
  vector<int>::iterator iter;

  // initialize the iterator with the first element
  iter = num.begin();

  // print the vector element
  cout << "num[0] = " << *iter << endl;

  // iterator points to the 3rd element
  iter = num.begin() + 2;
  cout << "num[2] = " << *iter;

  // iterator points to the last element
  iter = num.end() - 1;
  cout << "num[4] = " << *iter;

  for (iter = num.begin(); iter != num.end(); ++iter) {
    cout << *iter << "  ";
  }
  return 0;
}
```

4.7 Exception and Error Handling

Exception are those unusual runtime errors due to which normal execution is ceased. An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications, or the system has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner. For handling these kinds of abnormalities, exception handling should be used. This process consists of steps: *Hit, Throw, Catch* and *Handle* the exception.

In C++, *try* and *catch* are used in exception handling. The *try* block hits and throws an exception object which is then caught and handled in *catch* block.

```
try
{
    //code
    throw parameter;
}
catch(exceptionname ex)
{
    //code to handle exception
}
```

For example, a division by zero exception is as,

```
#include <iostream>
float division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero!";
   }
   return a/b;
}
void main () {
   int x = 50, y = 0;
   float z;
   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* excepMsg) {
      cerr << excepMsg << endl;
   }
}
```

Also, an access to unallocated space is as,

```
int main() {
    vector<int> vec;
    vec.push_back(0);
    vec.push_back(1);
    // access the third element, that doesn't exist
    try
    {
        vec.at(2);
    }
    catch (exception& ex)
    {
        cout << "Exception occurred!" << endl;
    }
    return 0;
}
```

Similarly, error handling refers to the anticipation, detection and resolution of programming, application and communications errors including syntactical and logical errors. Specialized programs called error handlers are available for some applications. Error handler captures a stack back trace by walking up the stack and recording the information for each frame, displays the message box and terminates the application when the message box is closed and the error can be retrieved.

4.7.1 Testing and Quality Assurance

Once a program code is written, it must be tested to detect and subsequently handle all errors in it. A number of schemes are used for testing purposes. Another important aspect is the fitness of purpose of a program that ascertains whether the program serves the purpose which it aims for. The fitness defines the software quality.

Testing is a continuous activity during software development. In object-oriented systems, testing encompasses three levels, namely, unit testing, subsystem testing, and system testing.

- **Unit Testing:** In unit testing, the individual classes are tested. It is seen whether the class attributes are implemented as per design and whether the methods and the interfaces are error-free. Unit testing is the responsibility of the application engineer who implements the structure.
- **Subsystem Testing:** This involves testing a particular module or a subsystem and is the responsibility of the subsystem lead. It involves testing the associations within the subsystem as well as the interaction

of the subsystem with the outside. Subsystem tests can be used as regression tests for each newly released version of the subsystem. The two main approaches of subsystem testing are:

- **Thread based testing:** All classes that are needed to realize a single use case in a subsystem are integrated and tested.
- **Use based testing:** The interfaces and services of the modules at each level of hierarchy are tested. Testing starts from the individual classes to the small modules comprising of classes, gradually to larger modules, and finally all the major subsystems.

- **System Testing:** System testing involves testing the system as a whole and is the responsibility of the quality assurance team. The team often uses system tests as regression tests when assembling new releases. The major categories of System Testing are:
  - **Alpha testing:** This is carried out by the testing team within the organization that develops software.
  - **Beta testing:** This is carried out by select group of co-operating customers.
  - **Acceptance testing:** This is carried out by the customer before accepting the deliverables.

Software quality assurance is a methodology that determines the extent to which a software product is fit for use. The activities that are included for determining software quality are:

- Auditing Development of standards and guidelines
- Production of reports
- Review of quality system

The software quality factors include:

- **Correctness:** Correctness determines whether the software requirements are appropriately met.
- **Usability:** Usability determines whether the software can be used by different categories of users (beginners, non-technical, and experts).
- **Portability:** Portability determines whether the software can operate in different platforms with different hardware devices.
- **Maintainability:** Maintainability determines the ease at which errors can be corrected and modules can be updated.
- **Reusability:** Reusability determines whether the modules and classes can be reused for developing other software products.

4.7.1.1 Object Oriented Metrics

Metrics can be broadly classified into three categories: project metrics, product metrics, and process metrics.

1. **Project Metrics:** Project Metrics enable a software project manager to assess the status and performance of an ongoing project. The following metrics are appropriate for object-oriented software projects:
   - Number of scenario scripts
   - Number of key classes
   - Number of support classes
   - Number of subsystems
2. **Product Metrics:** Product metrics measure the characteristics of the software product that has been developed. The product metrics suitable for object-oriented systems are:
   - **Methods per Class:** It determines the complexity of a class. If all the methods of a class are assumed to be equally complex, then a class with more methods is more complex and thus more susceptible to errors.
   - **Inheritance Structure:** Systems with several small inheritance lattices are more well– structured than systems with a single large inheritance lattice. As a thumb rule, an inheritance tree should not have more than 7 (± 2) number of levels and the tree should be balanced.
   - **Coupling and Cohesion:** Modules having low coupling and high cohesion are considered to be better designed, as they permit greater reusability and maintainability.
   - **Response for a Class:** It measures the efficiency of the methods that are called by the instances of the class.
3. **Process Metrics:** Process metrics help in measuring how a process is performing. They are collected over all projects over long periods of time. They are used as indicators for long-term software process improvements. Some process metrics are:
   - Number of KLOC (Kilo Lines of Code)
   - Defect removal efficiency
   - Average number of failures detected during testing
   - Number of latent defects per KLOC

SAMPLE PROBLEM

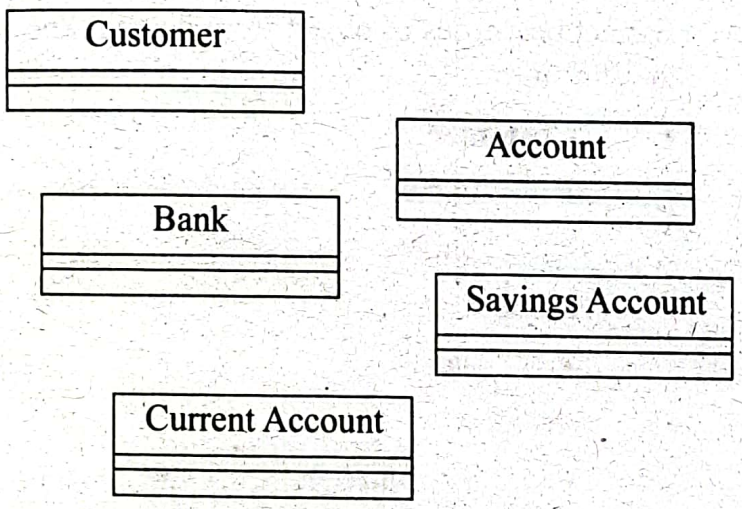
Q. Convert a problem statement into class diagram and implement it.

⇒ **Problem Statement**: Design and develop a simple 'Banking System' where customers can open personal accounts. There are two kinds of accounts: Savings and Current. The account has the functionality to deposit, withdraw and request balance. The account is uniquely identified by account number.

1. Problem Statement to Class Diagram:

Identify Classes

Extract all the nouns in the problem statement which acts as probable classes. You can eliminate irrelevant cases and add more classes by analyzing requirements. In given problem, sufficient classes would be: **Customer, Bank, Account, Savings Account, Current Account.**



Identify Relationships like Associations, Dependency or Inheritance

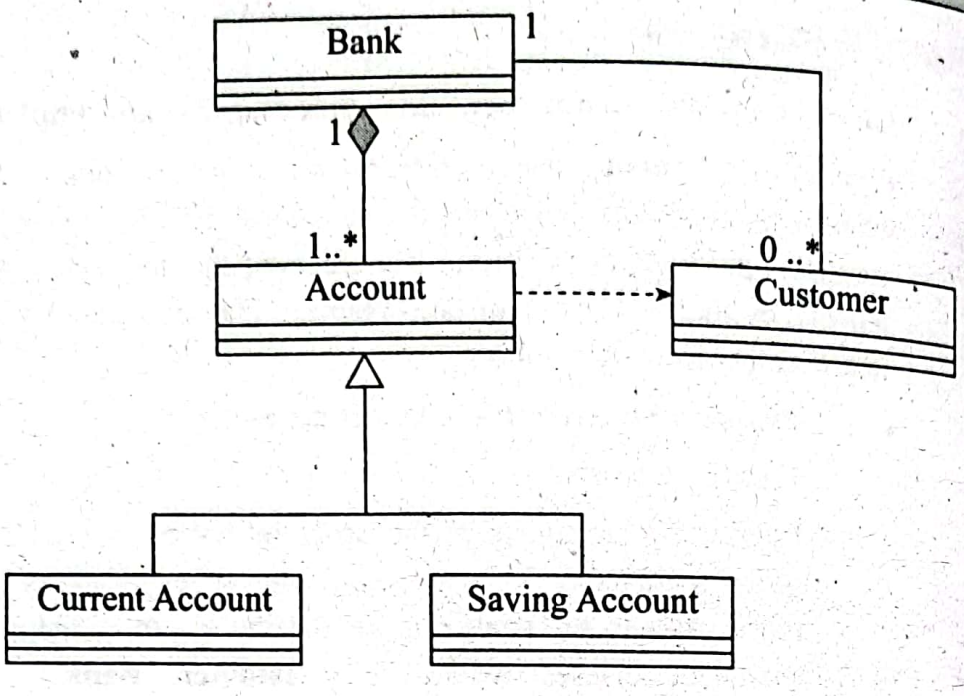
Extract verb connecting objects/entities or their relation.

"Customers can open personal accounts in bank"

This may denote that Account and Bank classes are associated. This creates link between those classes. Also, determine the multiplicity. E.g. a bank has multiple accounts corresponding customers. This may vary upon different types of approaches in same domain as well.

"Accounts are of two types: Savings and Current"

This denotes inheritance of Savings and Current account from Account class.



Identify Attributes and Operations

Extract properties of object from domain knowledge which are the attributes.

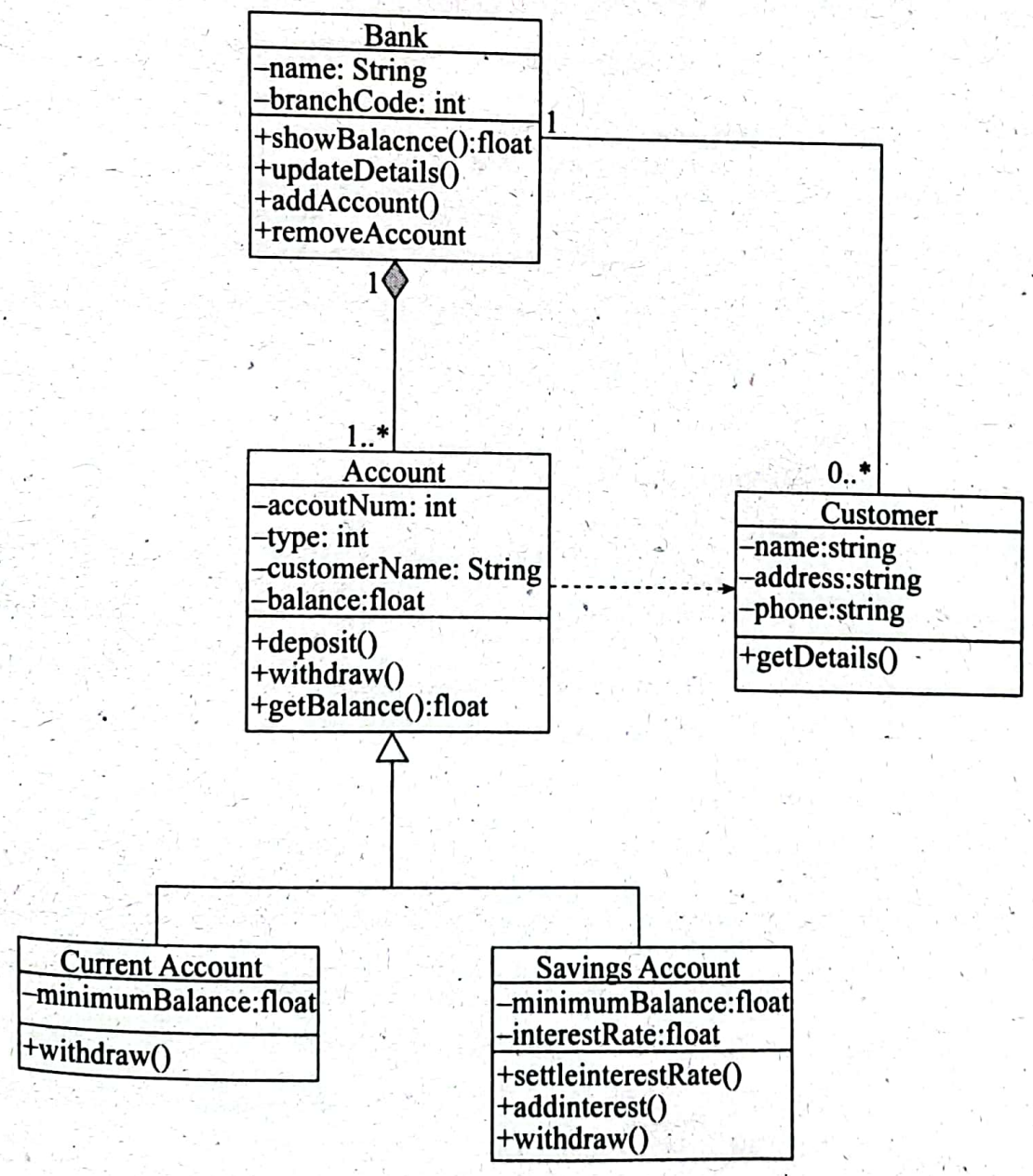
- **Customer**
  - name
  - address
  - phone
- **Bank**
  - name
  - branchCode
- **Account**
  - accountNum
  - type
  - customerName
  - balance
- **Savings Account**
  - minimumBalance
  - interestRate
- **Current Account**
  - minimumBalance

Extract verbs connected with an object and make adjustments with domain knowledge which forms its methods or operations.

- **Customer**
  - getDetails
- **Bank**
  - showBalance
  - addAccount
  - removeAccount

- **Account**
  - deposit
  - withdraw
  - getBalance
- **Savings Account**
  - setInterestRate
  - addInterest
  - withdraw
- **Current Account**
  - withdraw

Finally determine the data types of attributes and return type of methods then the final class diagram looks as follows:



2. **UML Class Diagram to Code(C++):**

//Customer.cpp

```
class Customer{
private:
        String phone;
        String name;
        String address;
public:
   Customer(String name, String phone,
String address):name(name), phone(phone),
address(address){}
   ~Customer(){}
        String getName(){
        return name;
        }
        void getDetails(){
            cout<<"Name: "<<name<<endl;
            cout<<"Phone: "<<phone<<endl;
            cout<<"Address: "<<address<<endl;
        }
};
```

//Account.cpp

```
class Account{
private:
        int accountNum;
        String customerName;
        float balance;
public:
   Account(Customer c1):accountNum(0),
customerName(c1.getName()), balance(0.0){}
   ~Account(){}
   void deposit(float amount){
            if (amount < 0.0)
          cout << "ERROR: Deposit amount
cannot be negative." << endl;
```

```
        else {
        balance+=amount;
        cout << amount << " has been
deposited to account #: " << accountNum <<
endl;
    }
    void withdraw(int amount){
        if (amount < 0.0)
        cout << "ERROR: Withdrawn amount
cannot be negative." << endl;
        else if(amount > balance){
        cout << "ERROR: Withdrawn amount
exceeds available balance." << endl;
        }
        else {
        balance-=amount;
        cout << amount << " has been
withdrawn from account #: " << accountNum
<< endl;
        }
    }
    float getBalance(){
        cout<<"Balance for
"<<customerName<<" is:"<<balance;
    }
};

class CurrentAccount:public Account{
private:
  float minimumBalance = 150.0;
public:
  CurrentAccount(Customer
c1):Account(c1){}
    ~CurrentAccount(){}
  void withdraw(float amount){
```

```
        cout << "Current account." << endl;
        Account::withdraw(amount);
    }
};

class SavingsAccount:public Account{
private:
    float minimumBalance = 100.0;
    float interestRate = 10/100;
public:
    SavingsAccount(Customer
c1):Account(c1){}
        ~SavingsAccount(){}
    void withdraw(float amount){
        cout << "Savings account." << endl;
        Account::withdraw(amount);
    }
};
```

//bank.cpp

```
class Bank {
private:
    String name;
    int branchCode;
    vector<Account> accountList;
public:
    Bank() {name=""; branchCode=0;}
        Bank(String name, int
bCode):name(name), branchCode(bCode) {}
        Bank(String name, int bCode, Account
acc):name(name), branchCode(bCode)
{accountList.push(acc);}
        ~Bank() {}
    void showBalance(){
        mAccount->PrintBalance();
```

```
  }
  addAccount(Account acc){
        accountList.push(acc);
  }
  removeAccount(int aNum){
  int index = 1;
  for (vector::iterator
it=accountList.begin(); it !=
accountList.end(); it++, index++){
        if(*it->accountNum==aNum) break;

        }
      vec.erase(vec.begin() + index);
      }
};
```

//main.cpp

```
#include <iostream>
#include <String.h>
#include <vector>
//include all classes

void main(){
   Customer c1("Ram", "Pulchowk,
Kathmandu", "9876543210");
   CurrentAccount c1Account(c1);
      Bank bank1("XYZ Bank Limited", 123,
&c1Account);

      c1Account.deposit(15000);
   c1Account.withdraw(5000);
   c1Account.getBalance();          // prints
10000
}
```

Old Exam Question-Answer

1. Conceptual model vs. Implementation model.

[2078 Chaitra, 2075 Bhadra, 2072 Aswin]

⇒ Implementation model or system model is a representation of how a program actually works. It is much longer and complex to create and are only a visualization of the source code. It makes writing code easier.

The conceptual model is just a rough draft of the domain. It should help to better analyze, understand and communicate about the system. A good programmer should be able to implement the conceptual model directly into source code.

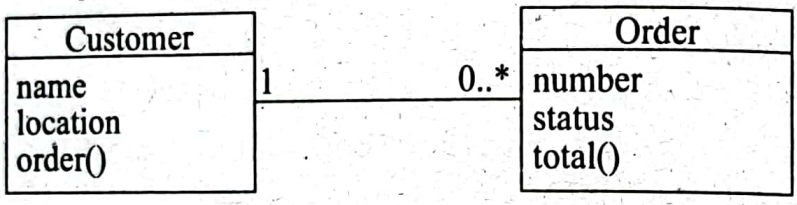


Fig: Conceptual Model

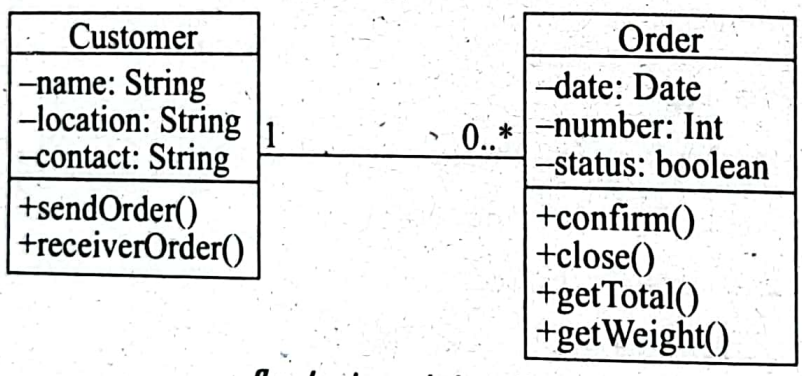


Fig.: Implementation Model

2. Forward Engineering vs. Reverse Engineering.

[2079 Jestha, 2075 Bhadra, 2072 Aswin, 2069 Bhadra]

⇒

Forward Engineering	Reverse Engineering
1. The application are developed with the given requirements.	1. The information is collected from the given application.
2. It is a high proficiency skill.	2. It is a low proficiency skill.
3. Prescriptive Nature: Developers are told how to work.	3. Adaptive Nature: Engineers figure out how it was done.
4. Flow is : Model => System.	4. Flow is : System => Model.
5. More Time for development.	5. Less Time for development.
6. E.g. Developing a new s/w from scratch.	6. E.g. Cloning Facebook, YouTube, Instagram.

Representation:

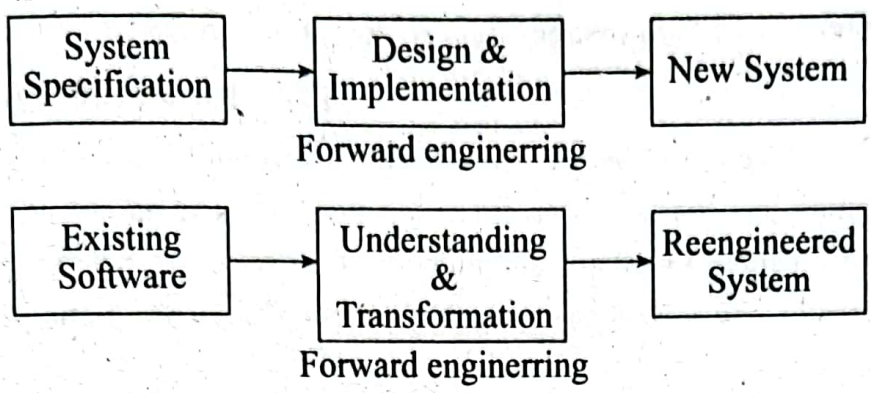


Fig.: forward and reverse engineering

3. **Distributed System implementation issues. [2075 Baisakh, 2070 Magh]**

⇒ Major issues relating to Distributed Systems are:

- **Heterogeneity**: Distributed systems is constructed from a variety of different networks, OS, hardware, and programming languages.
- **Openness**: Distributed system should be open to be extended or reimplemented which poses quite a challenge due to the differences.
- **Security**: Resource in distributed system should be made secure and must adhere to confidentiality, integrity and availability.
- **Scalability**: The system should be stable and remain effective even if number of resources and users are increased.
- **Failure handling**: Each components should be aware of any possible faults and failures and handle it effectively.
- **Concurrency**: Presence of shared resources should not create problems and concurrent sharing should be ensured.
- **Transparency**: The system, regardless of heterogeneity, must be perceived as a whole rather than collection of independent components.

4. **Explain error and exception handling with suitable example.**

[2078 Chaitra, 2073 Magh]

⇒ Please refer to theory section.

5. **How can we create class definition and methods from the domain class diagram and interaction diagram? Explain with example.**

[2073 Magh, 2071 Magh]

⇒ Please refer to theory section.

6. How interface differs from implementation? Explain the concept of interface and implementation in any OO language.

[2073 Bhadra, 2072 Aswin]

⇒ An interface describes a set of methods provided by a class or operation that can be performed on an object. It is an empty shell with only signature of methods and no body. It is that part of a class that is accessible to the outside, namely what is in the *public* section of the class.

Interface in C++:

```
//Stack interface
class Stack{
private:
int values[MAXSTACK];
int top;
public:
    Stack();
~Stack();
    void clear();
    bool empty() const;
    void push(int newvalue);
    int pop();
};
```

An implementation section implements such interface (*the code that carries out required behaviour*). It is the set of *private* data structures and the code for that implements the methods that act on the data.

Implementation in C++:

```
//Stack implementation
//push a new value onto the stack
void Stack::push(int newvalue){
    if(top < MAXSTACK)
    values[top++] = newvalue;
    else{
    cerr << "stack overflow" << endl;
```

```
        exit(1);
        }
}
//return a value from the top of the stack
int Stack::pop(){
        if(!empty())
        return values[--top];
        else{
        cerr << "stack underflow" << endl;
        exit(2);
        }
}
```

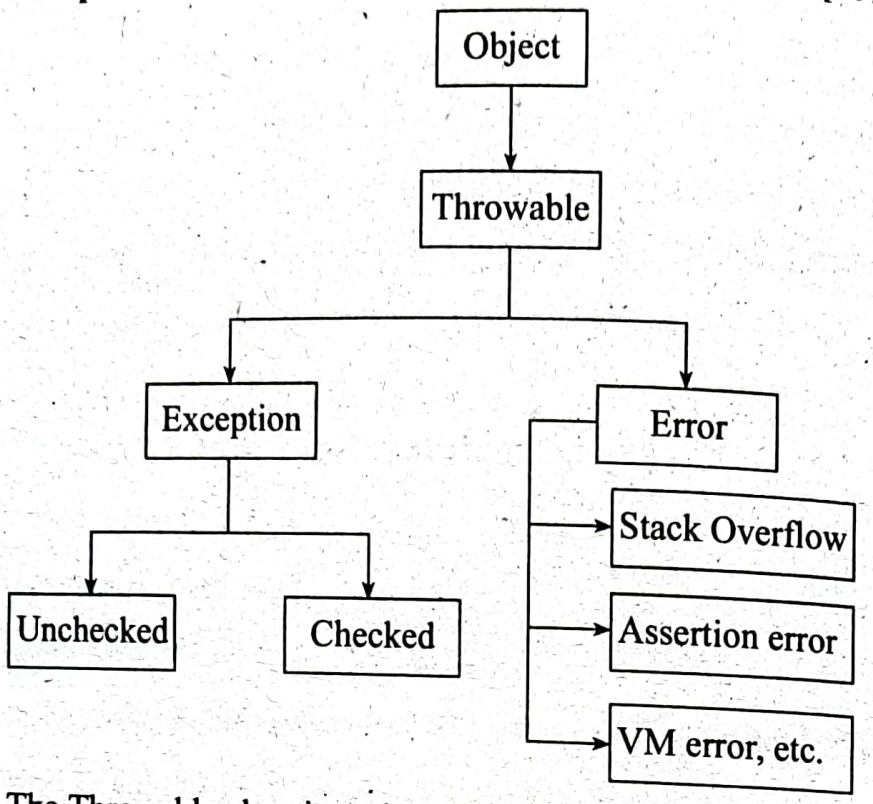
7. **Why is exception handling necessary in program development?**

[2073 Bhadra]

⇒ The necessity of exception handling are as follows:

- It can control run-time errors that occur in the program.
- It can avoid abnormal termination of the program and also shows the behavior of program to users.
- It can provide a facility to handle exceptions, throws message regarding exception and completes the execution of program by catching the exception.
- It can separate the error handling code and normal code by using try-catch block.
- It can produce the normal execution flow for a program.
- A failed constructor can also be handled by throwing an exception.
- It can implement a clean way to propagate error.
- It develops a powerful coding which ensures that the exceptions can be prevented.

8. Draw an exception class hierarchy to present the errors and exceptions derived from the Throwable class. [2072 Aswin]



The Throwable class is at the top of the exception hierarchy following which are the subclasses-Error and Exception. An exception is an unwanted event in a program that can be handled within the program the error cannot be handled in a program as it is a much serious condition that causes damage to the architecture. Exceptions are further divided into two categories:

- **Checked Exceptions** – These are also known as compile-time exceptions.
- **Unchecked Exceptions** – These are also known as runtime exceptions.

9. What are the perceptions of applying UML? [2074 Bhadra]

⇒ Raw diagram types like class diagrams and sequence diagrams are described in the UML. It doesn't apply a modeling perspective over top of these. For instance, the same UML class diagram notation can be used to depict software classes in C++ or real-world concepts.

- **Conceptual perspective**

  The diagrams are viewed as describing events in contexts seen in the actual world or related fields.

- **Specification (Software) perspective**

  The diagrams, which employ the same notation as in the conceptual perspective, depict software abstractions or components with requirements and interfaces without making any claims about a specific implementation (for instance, not particularly a class in C++ or Java).

- **Implementation (Software) perspective**

  The diagrams show how software in a specific technology (like C++) is implemented.

10. **Dewey Decimal Numbering and its use in modeling [2074 Bhadra]**

⇒ The Dewey Decimal Classification (DDC) system divides recorded knowledge into ten broad classes that cover the entire world of knowledge using simple decimal notation. Each main class is further subdivided into ten divisions, with each division further subdivided into ten sections, for a total of 100 divisions and 1,000 sections. All of the division and section numbers may not be used.

The Dewey Decimal Classification system is the most widely used method for classifying books in the library. This system is a general knowledge organization tool that is continuously revised to keep pace with knowledge. This classification is used in OO modeling for marking the sequence of time or events in UML diagrams including sequence diagram and collaboration diagrams.

Practice Questions

1. What are Steps for Mapping Designs to Code?
2. Why is a Unified approach to software development necessary? Discuss it.
3. Explain different categories of conceptual classes with examples and discuss the three strategies to find a conceptual class.
4. Develop the various testing strategies for Software quality assurance.
5. List out the errors that you may encounter when you execute your programs.
6. Illustrate with neat sketch the software development life cycle of object-oriented system.
7. Discuss briefly about Issues in OO Testing.
8. Write short notes on Class Testing.
9. Explain in detail about the OO Integration testing.
10. Explain in detail about GUI testing.
11. Write short notes on OO System testing.
12. Explain the operation of Mapping Designs to Code.

13. Explain in detail about coding and testing in OOAD.
14. Explain about the implementation model (Mapping design to code) and give the NextGen POS program solution.
15. Comparison between OO integration testing and OO system testing.
16. Explain in detail about the different types of testing in OOAD.
17. Comparison between GUI testing and class testing.
18. Explicate in detail the various testing strategies and the impact of object orientation on testing.
19. Illustrate and provide an interface for creating families of related or dependent objects without specifying their concrete classes using factory method.
20. What are the advantages of ignoring implementation-related aspects while performing analysis?
21. A Library lends books and magazines to member, who is registered in the system. It also maintains the purchase of new books and magazines for the Library. A member can reserve a book or magazine that is not currently available in the library, so that when it is returned or purchased by the library, that person is notified. The library can easily create, replace and delete information about the books, members, and reservation in the system. The books transactions are stored in the database. The fine list while the member returns the book after the due date must be generated. Design the use case diagram and implement it using your language of choice.
22. A mobile device has to be fitted with an alarm clock. The clock has a display unit to show the time of day. Using buttons, the user can set the hours and minutes fields individually. It supports a 24-hour display. It is possible to set one or two alarms. When an alarm fires, it will sound some noise. The user can turn it off, or choose to 'snooze'. If the user does not respond at all the alarm will turn off itself after 2 minutes. 'Snoozing' means to turn off the sound, but the alarm will fire again after some minutes of delay. This 'snoozing time' is pre-adjustable. Identify the functional requirements for the clock ,model it with a use case diagram and implement it using your language of choice.
23. A small university would like to create a registration system for its students. The students will use this system to obtain information about courses, when and where the classes meet, register for classes, print transcripts, drop classes, etc. The faculty will be using this system to find out what classes they are assigned to teach, when and where these classes meet, get a list of students registered for each class, and assign grades to students in their classes. The university administrative staff will be using this database to add new faculty and students, remove faculty and students who have left, put in and update information about each course the university offers, enter the schedules for classes that are being offered in each term, and any other housekeeping tasks that need to be performed. Design the use case diagram and implement it using your language of choice.

24. An airline has a weekly flight schedule. Associated with each flight is an aircraft, a list of crew, and a list of passengers. The airline would like to create and maintain a database that can perform the following functions: For passengers Add a passenger to the database, reserve a seat on a flight, print out an itinerary, request seating and meal preferences, and update frequent flier records. For crew Assign crew members to each flight, allow crew members to view their schedule, keep track of what kinds of aircraft the crew member has been trained to operate. For flights Keep track of crew list, passenger list, and aircraft to be used for that flight. For aircraft Maintain all records about the aircraft and a schedule of operation. Make an exhaustive list of queries that this system may be required to answer. Carry out a requirements analysis for the system and model it as a collection of use cases. Find the conceptual classes and their relationships.

25. A hotel reservation system supports the following functionality: (a) Room reservation (b) Changing the properties of a room (for example, from non-smoking to smoking) (c) Customer check-in (d) Customer check-out. Come up with system use cases and conceptual classes for the above functionality.

26. Consider the policies maintained by an automobile insurance company. A policy has a primary policy holder, a set of autos insured, and a list of people who are covered by the insurance. From your knowledge of insurance, come up with system use cases and model conceptual classes for (a) creating a new policy (b) adding a new person to a policy (c) adding a new automobile to a policy (d) recording a claim.

27. A company has several projects, and each employee works in a single project. The human resource system evaluates the personnel needs of each project and matches them against the personnel file to find the best possible employees to be assigned to the project. Come up with the conceptual classes by conducting use case analysis.

28. Suppose the library system has to be augmented so that it can support interlibrary loans. That is, a customer can ask the clerk if a certain book, which is not locally available, is available in some other library. What changes are needed (classes and use cases) to incorporate this new functionality?

29. A university registration system has a class Student that tracks student information. When a student's GPA falls below a certain level, he/she is placed on academic probation. Would you model this by creating a subclass WeakStudent that extends Student?

❑❑❑