1) Define UML? Explain the need for it.

_Ans:- **UML, or Unified Modeling Language,** is a standardized modeling language used in software engineering for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. It provides a set of graphic notation techniques to create visual models of object-oriented software systems.

Need for UML:

1. **Visual Representation:** UML provides a visual way to represent complex systems and their components, making it easier for stakeholders to understand and discuss system architectures and behaviors.
2. **Communication Tool:** UML acts as a common language between developers, analysts, designers, and other stakeholders. It bridges the gap between technical and non-technical team members, ensuring everyone understands the system's structure and behavior.
3. **Analysis and Design:** UML aids in the analysis and design phases of software development. It helps in capturing requirements, defining system structure, and planning the software architecture.
4. **Documentation:** UML diagrams serve as valuable documentation for software projects. They provide detailed insights into the system's architecture, interactions, and behavior, making it easier for developers to understand and modify the codebase.
5. **Blueprint for Implementation:** UML diagrams serve as a blueprint for implementing the system. Developers can refer to these diagrams to understand class relationships, object interactions, and system flow, aiding in the coding process.
6. **Tool Integration:** Several software development tools support UML, allowing developers to create UML diagrams and generate code from these diagrams. This integration streamlines the development process, ensuring consistency between the visual models and the actual implementation.

2) Explain types of UML models and their notations.

Ans:- 1. Class Diagrams:

- **Purpose:** Represents the static structure of the system, including classes, attributes, methods, and their relationships.
- **Notations:**
    - **Class:** Rectangle with the class name.
    - **Attributes:** Listed below the class name with a data type.

2. Use Case Diagrams:

- **Purpose:** Illustrates the interactions between actors (users or external systems) and the system.
- **Notations:**
    - **Actors:** Stick figures or rectangles representing external entities interacting with the system.
    - **Use Cases:** Ovals representing specific functionalities or tasks performed by the system.
    - **Associations:** Lines connecting actors and use cases, indicating interactions.

3. Sequence Diagrams:

- **Purpose:** Displays the interactions between objects and the order in which those interactions occur, representing the dynamic behavior of the system.
- **Notations:**
    - **Lifelines:** Vertical dashed lines representing objects participating in the interaction.
    - **Messages:** Horizontal arrows indicating the flow of communication between objects.
    - **Activation Bar:** Vertical rectangles on lifelines showing the duration of an object's activity.
    - **Return Message:** Dotted arrow indicating the return of a message.

4. Activity Diagrams:

- **Purpose:** Describes the flow of activities within the system, showcasing actions, decisions, and concurrency.
- **Notations:**
    - **Actions:** Rounded rectangles representing activities or tasks.
    - **Decisions:** Diamond shapes indicating decision points.
    - **Control Flow:** Arrows connecting actions and decisions, showing the flow of control.
    - **Fork and Join Nodes:** Splitting and merging points for concurrent activities.

5. State Machine Diagrams:

- **Purpose:** Represents the different states an object can be in and transitions between these states based on events.
- **Notations:**
    - **States:** Rounded rectangles representing different states (e.g., idle, active, error).
    - **Transitions:** Arrows connecting states, indicating the conditions triggering state changes.
    - **Events:** Triggers causing state transitions (e.g., events, messages).

3) Differentiate b/w structural and behavioural models with example.

Ans:- **Structural Models:**

1. Purpose: Structural models focus on the static aspects of a system, depicting its components, relationships, and organization without detailing their dynamic behavior.

a) Class Diagrams:

- **Purpose:** Represent the static structure of the system, including classes, attributes, methods, and their relationships.
- **Example:** Consider an online bookstore system. In a class diagram, you could have classes like `Book`, `Author`, and `Customer`. The `Book` class might have attributes such as `title`, `author`, and `price`, along with methods like `calculateDiscount()`.

b) Object Diagrams:

- **Purpose:** Provide a snapshot of the system at a particular point in time, showing objects and their relationships.
- **Example:** Using the same online bookstore system, an object diagram could illustrate specific instances of the `Book` class, such as a book titled "Example Book" authored by "John Doe" and priced at "$20.00". This diagram captures concrete instances of the classes.

Behavioral Models:

1. Purpose: Behavioral models focus on the dynamic aspects of a system, describing how objects collaborate, communicate, and change states over time.

a) Use Case Diagrams:

- **Purpose:** Depict the interactions between external entities (actors) and the system, showcasing different use cases and their relationships.
- **Example:** In the online bookstore system, a use case diagram could show actors like `Customer` and `Admin` interacting with the system through use cases like `Browse Books`, `Add to Cart`, and `Manage Inventory`. This diagram highlights the system's functionalities from a user's perspective.

b) Sequence Diagrams:

- **Purpose:** Illustrate the interactions between objects over time, indicating the order of messages exchanged.
- **Example:** Continuing with the online bookstore system, a sequence diagram could demonstrate the process of a `Customer` placing an order. It would show messages sent between objects, such as `Customer`, `Shopping Cart`, and `Payment Gateway`, indicating the sequence of actions during the order placement process.

4) Define Design Pattern? Bring out the categories of Design Patterns.

Ans:- **Design patterns** are general, reusable solutions to common problems that occur during software development. They represent best practices for software design and are solutions to recurring design problems. Design patterns can speed up the development process by providing tested, proven development paradigms.

Categories of Design Patterns:

Design patterns are categorized into three main types:

1. **Creational Patterns:**
   - **Purpose:** Focus on object creation mechanisms, trying to create objects in a manner suitable to the situation. They help in making a system independent of how its objects are created, composed, and represented.
2. **Structural Patterns:**
   - **Purpose:** Deal with object composition, creating relationships between objects to form larger structures. They help ensure that when one part of a system changes, the entire structure doesn't need to change.
3. **Behavioral Patterns:**
   - **Purpose:** Focus on communication between objects, how they operate together, and how one object knows about another's behavior. They help in making objects collaborate and avoid direct dependencies.

5) Write the Java code to implement Command Pattern.

Ans:- **Command Interface:**

```
// Command interface

interface Command {

   void execute();

}
```

Concrete Command Classes:

```
// Concrete Command 1

class LightOnCommand implements Command {

   private Light light;

  public LightOnCommand(Light light) {

      this.light = light;   }
```

```
    public void execute() {
        light.turnOn();
    }
}

// Concrete Command 2
class LightOffCommand implements Command {
    private Light light;
  public LightOffCommand(Light light) {
        this.light = light;  }
public void execute() {
        light.turnOff();
    }
}
```

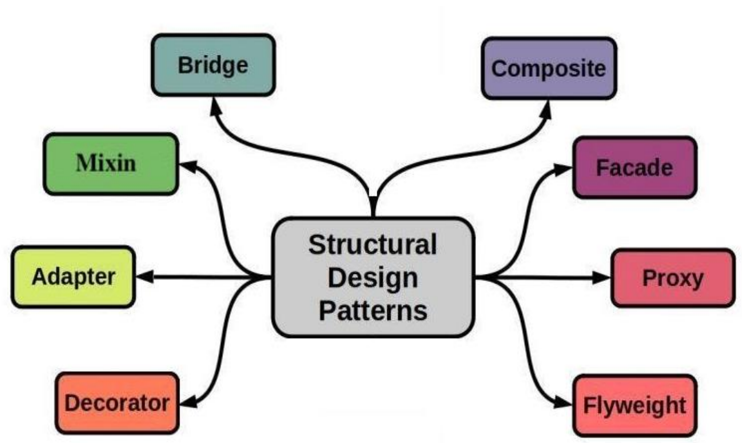
Receiver Class:

```
// Receiver
class Light {
    public void turnOn() {
        System.out.println("Light is ON");   }
   public void turnOff() {
        System.out.println("Light is OFF");
    }
}
```

Main Class:

```
public class Main {
    public static void main(String[] args) {
        Light light = new Light();
        Command lightOnCommand = new LightOnCommand(light);
        Command lightOffCommand = new LightOffCommand(light);
        RemoteControl remote = new RemoteControl();
        remote.setCommand(lightOnCommand);
        remote.pressButton();
        remote.setCommand(lightOffCommand);
        remote.pressButton();
    }
}
```

6) What are Structural Design Patterns? Explain.



-

Structural design patterns focus on simplifying the composition of classes and objects to form larger structures, making it easier to manage relationships and interfaces between different components of a system.

Here are some common structural design patterns along with brief explanations:

1. Adapter Pattern:

- **Purpose:** Allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces by converting the interface of a class into another interface that a client expects.

2. Bridge Pattern:

- **Purpose:** Separates abstraction from implementation so that the two can vary independently. It allows the abstraction and implementation to be developed independently and the client code can access only the abstraction part without being concerned about the implementation.

3. Composite Pattern:

- **Purpose:** Allows you to compose objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly.

4. Decorator Pattern:

- **Purpose:** Allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class. It is often used to extend the functionalities of classes in a flexible and reusable way.

5. Facade Pattern:

- **Purpose:** Provides a simplified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use.

6. Flyweight Pattern:

- **Purpose:** Minimizes memory usage or computational expenses by sharing as much as possible with related objects. It is used when a large number of similar objects need to be created and the overhead of creating each instance is high.