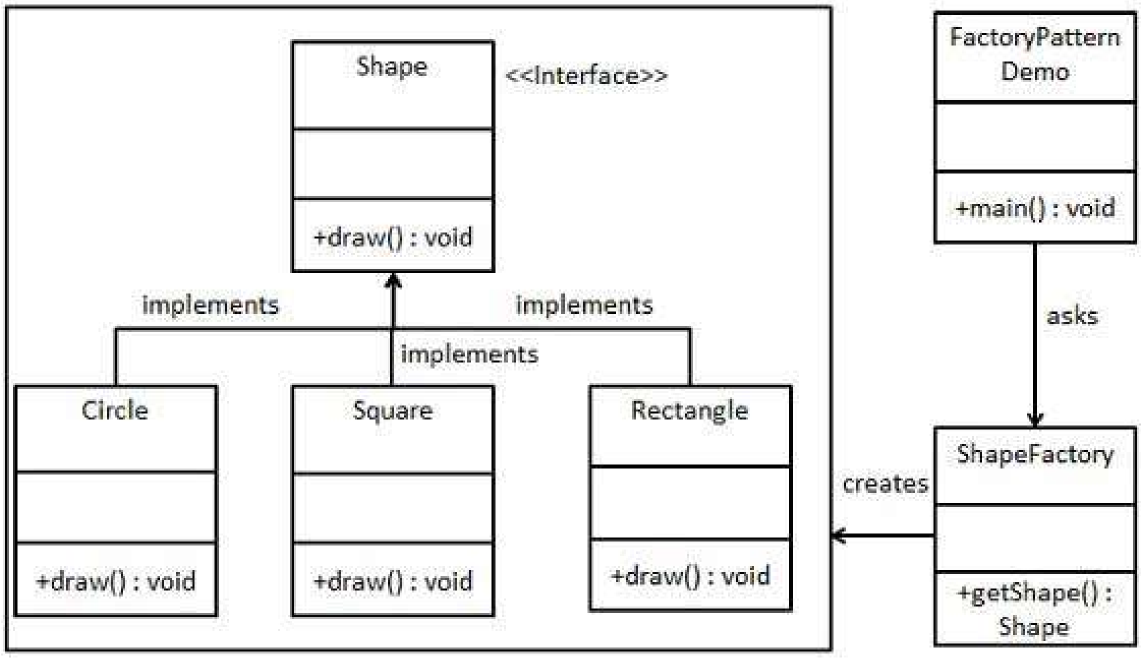
GUJARAT TECHNOLOGICAL UNIVERSITY
MCA INTEGRATED– SEMESTER - IX EXAMINATION- SUMMER-2023
Subject Code: 2698601 **Date: 27/06/2023**
Subject Name: Design Pattern

Q.1(a) Define the terms.
1. What are design patterns?
Ans.: Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers face during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.
There mainly four main types of design patterns which are given below:-

1. **Creational Patterns:** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using a new operator. This gives the program more flexibility in deciding which objects need to be created for a given use case.
2. **Structural Patterns:** These design patterns concern class and object composition. The concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3. **Behavioral Patterns:** These design patterns are specifically concerned with communication between objects.
4. **J2EE Patterns:** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

2. Draw the structure of the Factory method.
Ans.:



3. Describe patterns in software.

Ans.: Software engineering, design patterns are reusable solutions to common problems that occur during software development. These patterns represent best practices and provide a general, reusable template for solving a particular type of problem. They can speed up the development process by providing tested, proven development paradigms. Here are some key aspects of software design patterns:

1. **Reusability:** Design patterns promote reusability by providing well-established solutions to common problems. Developers can apply these patterns in different contexts to solve similar issues without reinventing the wheel.
2. **Abstraction:** Design patterns often involve abstraction, allowing developers to work at a higher level of abstraction by hiding complex implementation details. This makes the code more maintainable and adaptable to changes.
3. **Flexibility:** Design patterns make software systems more flexible and adaptable to change. They help to manage dependencies and provide a structure that allows components to be easily replaced or extended.
4. **Common Vocabulary:** Design patterns provide a common vocabulary and terminology for developers to communicate about software design. This shared understanding can improve collaboration and communication within development teams.
5. **Scalability:** Patterns can contribute to scalability by offering solutions that can be applied at different scales. Whether developing a small application or a large enterprise system, design patterns provide a foundation for scalable and maintainable code.
6. **Documentation:** Design patterns serve as a form of documentation. When developers encounter a familiar design pattern in the codebase, they can quickly understand the intent and purpose of that portion of the code.
7. **Problem Solving:** Design patterns encapsulate solutions to recurring design problems. They help developers avoid pitfalls and common mistakes by following proven solutions to well-known challenges.
8. **Guidance:** Design patterns offer guidance on how to structure code to achieve specific goals. They are not strict rules but rather guidelines that can be adapted to fit the specific needs of a project.

(b) Explain how to use design patterns.

Ans.: The various steps required to use the design problem are given below:

1. **Identify Problem: R**ecognize recurring design challenges in your project.
2. **Understand Pattern:** Learn about relevant design patterns that address the identified problem.
3. **Apply Pattern:** Integrate the pattern into your code, creating necessary classes and relationships.
4. **Adapt to Needs:** Tailor the pattern to fit your application's specific requirements.
5. **Implement and Test:** Code the pattern and thoroughly test to ensure proper functionality.

6. **Refine and Document:** Make refinements as needed and document the pattern's purpose and usage.
7. **Consider Trade-offs:** Be aware of any trade-offs associated with using the design pattern.
8. **Iterate and Learn:** Revisit and adjust patterns as needed based on project evolution and experience.

Q.2(a) What are the common causes of redesign? Explain.

Ans.: Redesign in software development is often prompted by various factors that indicate the need for significant changes to the existing system. Some common causes of the redesign include:

1. **Changing Requirements:** Evolving project requirements, whether due to shifts in business goals, user needs, or regulatory changes, can necessitate a redesign to align the software with the updated specifications.
2. **Poor Performance:** If the system experiences performance issues, such as slow response times or inefficient resource utilization, a redesign may be required to optimize and enhance performance.
3. **Scalability Challenges:** As user loads increase or when expanding the system to handle a larger volume of data, the current architecture may prove insufficient. Redesigning for scalability ensures the system can handle growth effectively.
4. **Outdated Technologies:** When the technologies or libraries used in the current system become obsolete or no longer supported, a redesign may be necessary to migrate to newer, more sustainable technologies.
5. **Maintenance Challenges:** Cumbersome maintenance, high bug counts, or difficulties in adding new features might indicate a design that is hard to understand or modify. A redesign can simplify and improve maintainability.
6. **Security Concerns:** Changes in security threats or the discovery of vulnerabilities may necessitate a redesign to implement stronger security measures and ensure the protection of sensitive data.
7. **User Feedback:** Feedback from end-users, whether through usability testing or direct user input, can reveal shortcomings in the design or functionality, prompting a redesign to enhance the user experience.
8. **Technological Advances:** New technological advancements, such as the introduction of better programming languages, frameworks, or tools, may prompt a redesign to take advantage of improved capabilities.
9. **Integration Requirements:** If the software needs to integrate with new systems, services, or platforms, a redesign may be necessary to accommodate seamless integration.
10. **Regulatory Changes:** Changes in industry regulations or compliance requirements may demand modifications to the system to ensure it remains in compliance with legal standards.
11. **Business Strategy Shifts:** A change in the overall business strategy or objectives may require a redesign to align the software with the updated organizational goals.

12. **Unanticipated Issues:** The discovery of unforeseen issues or limitations in the existing design during development or post-release may prompt a redesign to address these issues comprehensively.
13. **Technology Stack Upgrade:** Upgrading the underlying technology stack (e.g., database systems, server infrastructure) may require a redesign to adapt the software to the new environment.

(b) Explain the steps to be followed for selecting a design pattern.

Ans.: The various required for selecting a design pattern are given below:

1. **Understand the Problem:** Clearly know what problem you're solving.
2. **Identify Key Aspects:** Figure out the main aspects of the problem, like how objects interact.
3. **Consider Design Principles:** Think about basic design principles (SOLID).
4. **Review Existing Solutions:** Check if there are known solutions or patterns for similar problems.
5. **Learn Pattern Catalogs:** Get familiar with design pattern catalogs.
6. **Consider Applicability:** See if each pattern fits your problem well.
7. **Analyze Trade-offs:** Understand the pros and cons of each pattern.
8. **Match Patterns to Aspects:** Connect your problem's aspects with suitable patterns.
9. **Consider Combinations:** Think about using more than one pattern if needed.
10. **Think About Future Changes:** Choose a pattern that's flexible for future updates.
11. **Document Your Decision:** Write down why you chose a specific pattern.
12. **Review and Validate:** Check your decision with others to make sure it makes sense.
13. **Iterate if Necessary:** Be open to changing your choice if needed.

(b) Explain the sample code for the Flyweight pattern.

Ans.: The Flyweight pattern is a structural design pattern that aims to minimize memory usage or computational expenses by sharing as much as possible with other similar objects. It's particularly useful when there are a large number of similar objects in an application.

For Example:

```
class TreeType:
  def __init__(self, name, color, texture):
    self.name = name
    self.color = color
    self.texture = texture
class Tree:
  def __init__(self, x, y, tree_type):
    self.x = x
    self.y = y
    self.tree_type = tree_type
  def draw(self):
        print(f"Drawing a {self.tree_type.name} tree at ({self.x}, {self.y}) with color {self.tree_type.color} and texture {self.tree_type.texture}")
class TreeFactory:
    tree_types = {}
    @classmethod
    def get_tree_type(cls, name, color, texture):
        key = (name, color, texture)
        if key not in cls.tree_types:
            cls.tree_types[key] = TreeType(name, color, texture)
        return cls.tree_types[key]
def main():
    factory = TreeFactory()
    tree1 = Tree(1, 2, factory.get_tree_type("Pine", "Green", "Needle"))
    tree2 = Tree(3, 4, factory.get_tree_type("Oak", "Brown", "Broadleaf"))
    tree3 = Tree(5, 6, factory.get_tree_type("Pine", "Green", "Needle"))
    tree1.draw()
    tree2.draw()
    tree3.draw()
if __name__ == "__main__":
    main()
```

Q.3(a) Explain the toolkit and frameworks.

Ans.: **Toolkit:** A toolkit, in the context of software development, refers to a set of tools, libraries, and components that provide developers with pre-built functionalities to simplify and accelerate the development of applications. Toolkits typically include a collection of reusable software components, such as user interface elements, data structures, and algorithms, that developers can leverage in their projects. These components are designed to work together cohesively and can be customized to suit specific application requirements.

Toolkits can be general-purpose or specialized for certain domains (e.g., GUI toolkits for building graphical user interfaces). Examples of toolkits include Qt, GTK, and JavaFX for building graphical user interfaces, TensorFlow and PyTorch for machine learning, and Bootstrap for web development.

Frameworks: A framework is a more comprehensive software structure that provides a foundation for developing applications. It typically includes a set of rules, conventions, and pre-defined functionalities that guide the development process. Developers build their applications within the framework, following its structure and utilizing its components.

Frameworks often impose a specific architecture or design pattern to promote consistency and maintainability across applications. They provide a higher level of abstraction compared to toolkits and can include tools, libraries, and services. Frameworks may be general-purpose or tailored for specific domains, such as web development (e.g., Django, Ruby on Rails), application development (e.g., Spring for Java), or game development (e.g., Unity for game development).

(b) Write the code the for prototype pattern.

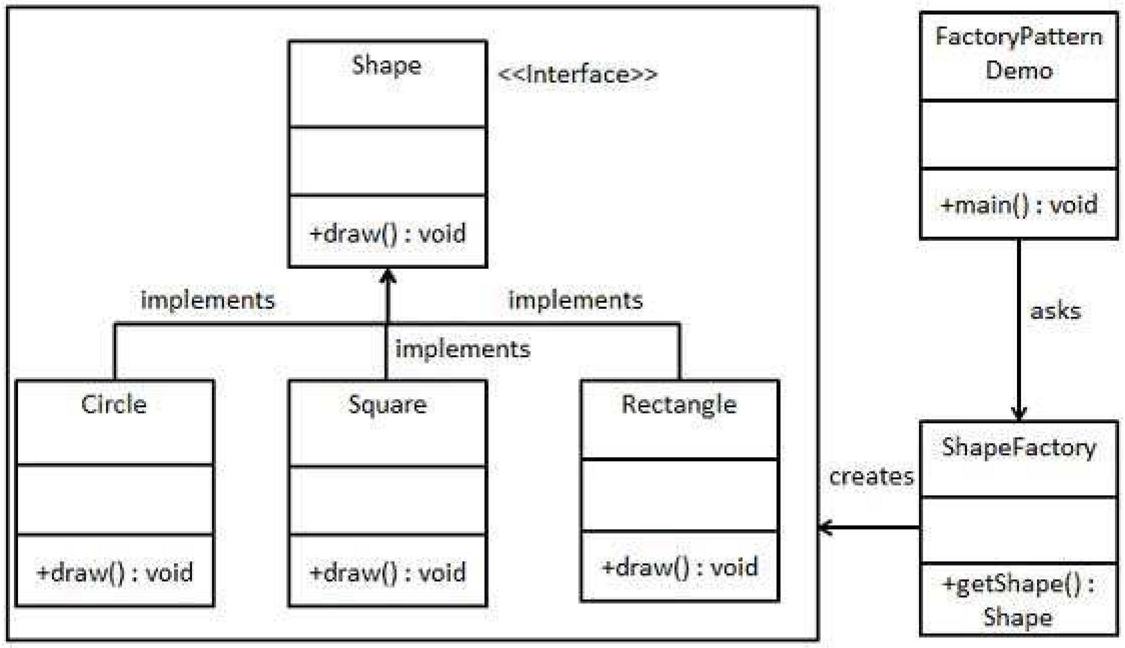
Ans.:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
interface Prototype{
        public Prototype getClone();
}
class EmployeeRecord implements Prototype{
        private int id;
        private String name, designation;
        private double salary;
        public EmployeeRecord(){
        System.out.println("Employee Records ");
        System.out.println("---------------------------------------------");
        System.out.println("Eid"+"\t"+"Ename"+"\t"+"Edesignation"+"\t"+"Esalary"); }
        public EmployeeRecord(int id, String name, String designation, double salary){
                this();
                this.id= id;
                this.name= name;
                this.designation= designation;
                this.salary= salary;
        }
        public void showRecord(){
                System.out.println(id+"\t"+name+"\t"+designation+"\t"+salary);}
                @Override
                public Prototype getClone(){
                        return new EmployeeRecord(id,name,designation,salary);
        }
}
public class PrototypeDemo{
public static void main(String[] args) throws IOException{
        BufferedReader br =new BufferedReader(new
        InputStreamReader(System.in));
        System.out.print("Enter Employee Id: ");
        int eid=Integer.parseInt(br.readLine());
        System.out.print("\n");
        System.out.print("Enter Employee Name: ");
        String ename=br.readLine();
        System.out.print("\n");
        System.out.print("Enter Employee Designation: ");
        String edesignation=br.readLine();
        System.out.print("\n");
        System.out.print("Enter Employee Salary: ");
        double esalary= Double.parseDouble(br.readLine());
        System.out.print("\n");
        EmployeeRecord e1=new
        EmployeeRecord(eid,ename,edesignation,esalary);
        e1.showRecord();
        System.out.println("\n");
        EmployeeRecord e2=(EmployeeRecord) e1.getClone();
        EmployeeRecord e3=(EmployeeRecord) e2.getClone();
        e1.showRecord();
        e2.showRecord();
        e3.showRecord();
        }
}
```

(a) Draw the structure the of Factory method and explain.

Ans.: Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.



1. Create an interface named *Shape.java*
2. Create concrete classes implementing the same interface(Rectangle.java, Square.java, Circle.java).

3. Create a Factory to generate object of concrete class based on given information (ShapeFactory.java)
4. Use the Factory to get object of concrete class by passing an information such as type (FactoryPatternDemo.java)

(b) Differentiate Structural and Behavioral Design patterns.

Ans.: **Structural Design Patterns:**

1. **Purpose:**
   - **Structural patterns** are concerned with the composition of classes or objects to form larger structures.
   - They focus on how classes and objects are organized to form a larger, more complex system.
2. **Key Elements:**
   - **Classes and Objects:** Structural patterns deal with the way classes and objects are assembled to form larger structures.
   - **Composition:** They emphasize the composition of classes and objects to create more flexible and reusable systems.
3. **Examples:**
   - **Adapter Pattern:** Allows incompatible interfaces to work together.
   - **Decorator Pattern:** Adds new functionalities to objects by wrapping them with additional classes.
   - **Composite Pattern:** Treats individual objects and compositions of objects uniformly.
4. **Use Cases:**
   - Useful when you want to compose objects into larger structures or when you need to adapt interfaces.

Behavioral Design Patterns:

1. **Purpose:**
   - **Behavioral patterns** are concerned with the interaction and responsibility of objects.
   - They define patterns for communication between classes and objects.
2. **Key Elements:**
   - **Object Interaction:** Behavioral patterns focus on how objects collaborate and communicate with each other.
   - **Responsibility:** They define patterns for distributing responsibilities among objects in a system.
3. **Examples:**
   - **Observer Pattern:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
   - **Strategy Pattern:** Defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable.

- **Command Pattern:** Encapsulates a request as an object, thereby allowing for parameterization of clients with different requests, queuing of requests, and logging of the requests.

4. **Use Cases:**
   - Useful when you want to define how objects interact and collaborate, or when you need to define the responsibility and algorithms of objects.

Q.4(a) Explain the implementation of composite design pattern with an example.

Ans.: The Composite Design Pattern is a structural pattern that allows you to compose objects into tree structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly.

For Example:

```
import java.util.ArrayList;
import java.util.List;
// Component - Abstract base class for all components
interface Graphic {
   void draw();
}
// Leaf - Represents individual graphic objects
class Circle implements Graphic {
   private String name;
   public Circle(String name) {
      this.name = name;
   }
   @Override
   public void draw() {
      System.out.println("Drawing Circle " + name);
   }
}
// Composite - Represents a collection of graphic objects
class CompoundGraphic implements Graphic {
   private List<Graphic> graphics = new ArrayList<>();
   public void add(Graphic graphic) {
      graphics.add(graphic);
   }
   public void remove(Graphic graphic) {
      graphics.remove(graphic);
   }
   @Override
   public void draw() {
      System.out.println("Drawing Compound Graphic:");
      for (Graphic graphic : graphics) {
          graphic.draw();
      }
   }
}
// Client code
public class CompositeExample {
   public static void main(String[] args) {
      // Create leaf objects
      Circle circle1 = new Circle("1");
      Circle circle2 = new Circle("2");
      Circle circle3 = new Circle("3");
      // Create a composite object
      CompoundGraphic composite = new CompoundGraphic();
      // Add leaf objects to the composite
      composite.add(circle1);
      composite.add(circle2);
      composite.add(circle3);
      // Create another leaf object
      Circle circle4 = new Circle("4");
      // Add the new leaf object to the composite
      composite.add(circle4);
      // Draw the composite object (which includes individual circles)
      composite.draw();
   }
}
```

- **Graphic** is the interface declaring the common method **draw()** for all components.
- **Circle** is a leaf class implementing the **Graphic** interface.
- **CompoundGraphic** is a composite class that contains a collection of **Graphic** objects, either individual **Circle** objects or other **CompoundGraphic** objects.
- The client code in the **CompositeExample** class creates individual **Circle** objects and a **CompoundGraphic** object. It adds the **Circle** objects to the **CompoundGraphic** to create a composite structure.
- When the **draw()** method is called on the **CompoundGraphic**, it recursively calls the **draw()** method on all its child components, resulting in the drawing of both individual circles and the composite graphic.

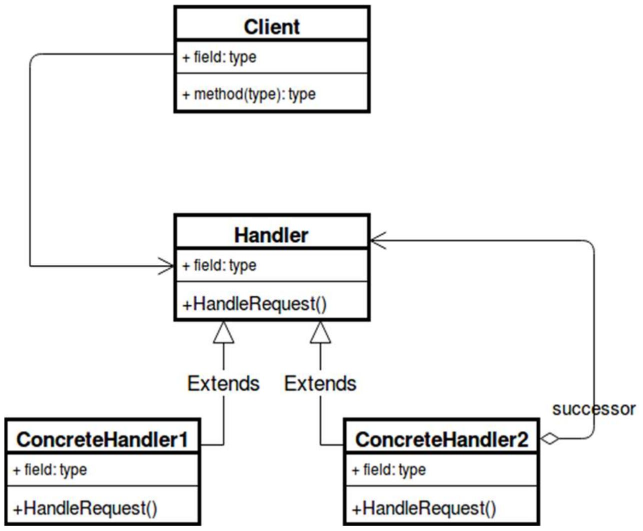
(b) Explain the following.

1. Interpreter: All high-level languages need to be converted to machine code so that the computer can understand the program after taking the required inputs. The software by which the conversion of the high-level instructions is performed line-by-line to machine-level language, other than compiler and assembler, is known as INTERPRETER.

The interpreter in the compiler checks the source code line-by-line and if an error is found on any line, it stops the execution until the error is resolved. Error correction is quite easy for the interpreter as the interpreter provides a line-by-line error. But the program takes more time to complete the execution successfully. Interpreters were first used in 1952 to ease programming within the limitations of computers at the time. It translates source code into some efficient intermediate representation and executes them immediately.

2. Chain of responsibility: It is a behavioral design pattern that allows an object to pass a request along a chain of potential handlers. Upon receiving a request, each handler decides either to process the request or to pass it along to the next handler in the chain. This pattern promotes loose coupling between senders and receivers of requests.

- Handler : This can be an interface which will primarily receive the request and dispatches the request to a chain of handlers. It has reference to the only first handler in the chain and does not know anything about the rest of the handlers.
- Concrete handlers : These are actual handlers of the request chained in some sequential order.
- Client : Originator of request and this will access the handler to handle it.



(a) What are the implementation issues raises by State pattern? Explain

Ans.: The State pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. While the State pattern provides a clean and modular way to manage state-specific behavior, there are some implementation issues that developers should be aware of:

1. **Complexity with Many States:** As the number of states increases, the class implementing the context (the object whose behavior is influenced by its state) can become large and complex. Each state may require a separate class, leading to a proliferation of classes.
2. **Transitions Between States:** Handling state transitions can be error-prone if not managed carefully. Incorrect transitions or missing transitions might lead to unexpected behavior. Ensuring that all transitions are correctly implemented and tested is crucial.
3. **Duplication of Code:** Depending on the complexity of state-specific behavior, there might be code duplication across state classes. This can occur when multiple states share similar actions, leading to redundancy in the codebase.

4. **Maintaining State Consistency:** Ensuring that the state of the context object is consistent across state transitions is important. Developers must carefully manage any data or attributes that are tied to specific states to avoid inconsistencies.
5. **Context Knowledge of States:** The context object needs to have knowledge of all possible states. If new states are added later, the context must be updated, which can impact the Open/Closed Principle (OCP) if the modification is frequent.
6. **Testing Challenges:** Testing the state transitions and ensuring that the correct behavior is exhibited for each state can be challenging. Comprehensive testing is essential to validate the correctness of the state machine.
7. **Concurrency Issues:** In a multithreaded environment, managing state changes becomes more complex. Proper synchronization mechanisms might be necessary to avoid race conditions or inconsistencies in the state.
8. **Increased Number of Classes:** The State pattern can lead to a larger number of classes in the codebase, which might be overwhelming for smaller projects. It's important to balance the benefits of the pattern against the complexity it introduces.

(b) Explain the Template Method.

Ans.: Template method design pattern is to define an algorithm as a skeleton of operations and leave the details to be implemented by the child classes. The overall structure and sequence of the algorithm are preserved by the parent class.
Template means Preset format like HTML templates which has a fixed preset format. Similarly in the template method pattern, we have a preset structure method called template method which consists of steps. These steps can be an abstract method that will be implemented by its subclasses.
This behavioral design pattern is one of the easiest to understand and implement. This design pattern is used popularly in framework development. This helps to avoid code duplication.
The two main components of Template method are given below:

- **AbstractClass** contains the templateMethod() which should be made final so that it cannot be overridden. This template method makes use of other operations available in order to run the algorithm but is decoupled for the actual implementation of these methods. All operations used by this template method are made abstract, so their implementation is deferred to subclasses.
- **ConcreteClass** implements all the operations required by the templatemethod that were defined as abstract in the parent class. There can be many different ConcreteClasses.

For Example:

```
abstract class OrderProcessTemplate
{
        public boolean isGift;
        public abstract void doSelect();
        public abstract void doPayment();
        public final void giftWrap()
        {
                try
                {
                        System.out.println("Gift wrap successful");
                }
                catch (Exception e)
                {
                        System.out.println("Gift wrap unsuccessful");
                }
        }
        public abstract void doDelivery();
        public final void processOrder(boolean isGift)
        {
                doSelect();
                doPayment();
                if (isGift) {
                        giftWrap();
                }
                doDelivery();
        }
class NetOrder extends OrderProcessTemplate
{
        @Override
        public void doSelect()
        {
                System.out.println("Item added to online shopping cart");
                System.out.println("Get gift wrap preference");
                System.out.println("Get delivery address.");
        }
        @Override
        public void doPayment()
        {
                System.out.println
                                ("Online Payment through Netbanking, card or Paytm");
        }
        @Override
        public void doDelivery()
        {
                System.out.println
                                        ("Ship the item through post to delivery address");
        }
}
class StoreOrder extends OrderProcessTemplate
{
        @Override
        public void doSelect()
        {
                System.out.println("Customer chooses the item from shelf.");
        }
        @Override
        public void doPayment()
        {
                System.out.println("Pays at counter through cash/POS");
        }
        @Override
        public void doDelivery()
        {
                System.out.println("Item delivered to in delivery counter.");
        }
}
class TemplateMethodPatternClient
{
        public static void main(String[] args)
        {
                OrderProcessTemplate netOrder = new NetOrder();
                netOrder.processOrder(true);
                System.out.println();
                OrderProcessTemplate storeOrder = new StoreOrder();
                storeOrder.processOrder(true);
        }
}
```

Q.5(a) 1. Write the implementation steps for strategy patterns.

Ans.: It is a behavioral design pattern that defines a family of algorithms, encapsulates each algorithm, and makes them interchangeable. It allows the client to choose the appropriate algorithm at runtime without altering the client's code. The Strategy Pattern is useful when you have a set of related algorithms and want to make them interchangeable, allowing the client to choose the most suitable one.

The various step required for the implementation of strategy Pattern which are given below:-

1. **Identify the Context:** Identify the class (the context) that will have a behavior that varies. This class should be designed to delegate the behavior to a strategy interface.

2. **Define the Strategy Interface:** Create an interface that defines the set of algorithms or behaviors that can be used interchangeably. This is the strategy interface.
3. **Implement Concrete Strategies:** Create one or more classes that implement the strategy interface. Each class represents a specific algorithm or behavior.
4. **Modify the Context to Use Strategies:** Modify the context class to include a reference to the strategy interface. This reference should be set via a setter method or through the constructor.
5. **Delegate Behavior to Strategies:** In the context class, delegate the behavior that varies to the strategy interface. Instead of implementing the behavior directly in the context class, call the corresponding method on the strategy interface.
6. **Client Code:** In the client code, create an instance of the context class and set its strategy. The client can choose the appropriate strategy dynamically or statically.

Q.5(a) 2. Write the implementation steps for Visitor patterns.

Ans.: It is a behavioral design pattern that allows adding new behaviors to existing classes without altering their structure. It achieves this by defining a separate visitor class that encodes the new behavior, and this visitor class can traverse the elements of a class hierarchy.

The various step required for the implementation of Visitor Pattern which are given below:-

1. Define Element Interface: Create an interface or abstract class (Element) that declares an accept method.
2. Implement Concrete Elements: Create concrete classes (ConcreteElementA, ConcreteElementB) that implement the Element interface and provide their own accept method.
3. Define Visitor Interface: Create a visitor interface (Visitor) that declares a visit method for each concrete element.
4. Implement Concrete Visitors: Create concrete visitor classes (ConcreteVisitor1, ConcreteVisitor2) that implement the Visitor interface and provide specific behaviors for each element.
5. Implement Object Structure: Create a class (ObjectStructure) that holds a collection of elements and provides a method (acceptVisitor) to accept a visitor.
6. Client Code: In the client code, create concrete elements, concrete visitors, and an object structure. Use the object structure to accept different visitors.

Q.5 (b) Discuss what to expect from Design pattern.

Ans.: Design patterns offer a set of general, reusable solutions to common problems encountered in software design. design patterns in software development are like blueprints or templates for solving common problems in a smart and proven way. They offer a set of guidelines or best practices that help make your code more organized, reusable, and easier to understand.

Here's what you can expect from using design patterns:

1. **Smart Solutions:** Design patterns provide clever and effective solutions to problems that developers often encounter.

2. **Reuse of Solutions:** They encourage reusing solutions that have worked well in the past, saving time and effort.
3. **Clear and Modular Code:** Design patterns help make your code neat and organized, making it easier to understand and maintain.
4. **Better Communication:** They give developers a common language to discuss and share solutions, fostering better communication within a team.
5. **Scalability:** Design patterns help your code grow smoothly as your project becomes more complex.
6. **Readability:** They contribute to writing code that is easy to read, even for someone who didn't originally write it.
7. **Following Best Practices:** Design patterns often follow best practices, helping you write code that is aligned with industry standards.
8. **Guidance for New Developers:** They serve as a helpful guide for new developers, providing them with established ways to solve common problems.
9. **Recognizing Problems:** They help developers quickly recognize and solve recurring issues in a way that has been proven to work.
10. **Time-Tested Wisdom:** Design patterns are like wisdom gained from experience—they are solutions that have stood the test of time.

Q.5 (a) Explain structure, participants and implementation of Iterator pattern.

Ans.: The Iterator Pattern is a behavioral design pattern that provides a way to access elements of a collection (e.g., a list) sequentially without exposing the underlying representation of the collection.

Structure:

- Iterator: Defines an interface for accessing and traversing elements.
- ConcreteIterator: Implements the Iterator interface. Keeps track of the current position in the traversal of the aggregate.
- Aggregate: Defines an interface for creating an Iterator object.
- ConcreteAggregate: Implements the Aggregate interface to create an Iterator. Represents the collection of objects to be traversed.

Participants:

1. Iterator (Interface): Declares methods for traversing elements (first, next, isDone, current).
2. ConcreteIterator: Implements the Iterator interface.
3. Aggregate (Interface): Declares an interface for creating an Iterator.
4. ConcreteAggregate: Implements the Aggregate interface.

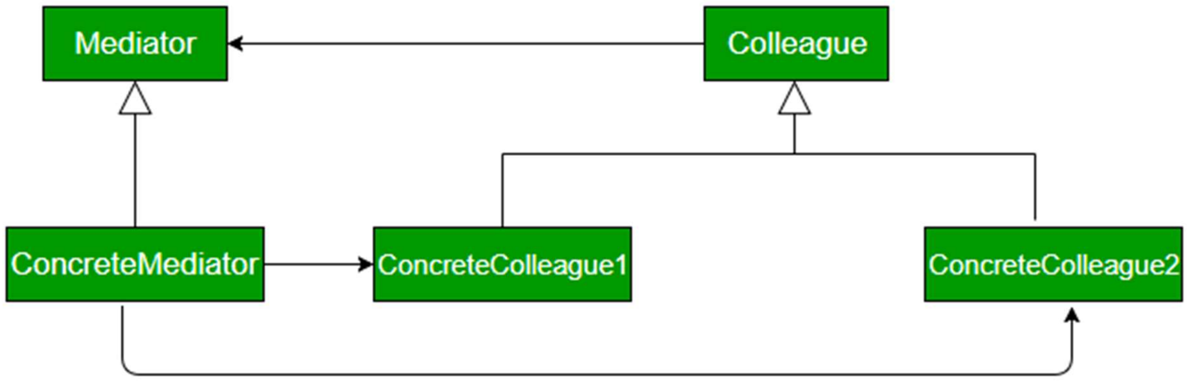
Implementation:

Client Code: Uses the Iterator and Aggregate interfaces to traverse the collection.

the Client uses the ConcreteAggregate class to create an aggregate of elements and obtain an iterator. The iterator (ConcreteIterator) is then used to traverse the collection, retrieving and processing each element.

Q.5 (b) 1. Explain the structure of Mediator pattern.

Ans.: Mediator design pattern is one of the important and widely used behavioral design pattern. Mediator enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the layer. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes higher maintainability cost and not hard to extend. Mediator pattern focuses on providing a mediator between objects for communication and help in implementing loose-coupling between objects. Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights.



- **Mediator :**It defines the interface for communication between colleague objects.
- **ConcreteMediator :** It implements the mediator interface and coordinates communication between colleague objects.
- **Colleague :** It defines the interface for communication with other colleagues
- **ConcreteColleague :** It implements the colleague interface and communicates with other colleagues through its mediator

Q.5 (B) 2. Explain the implementation of Memento pattern.

Ans.: Memento pattern is a behavioral design pattern. Memento pattern is used to restore the state of an object to a previous state. As your application progresses, you may want to save checkpoints in your application and restore back to those checkpoints later. The intent of Memento Design pattern is without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

The various step required for the implementation of Momento Pattern which are given below:-

1. Create a class that represents the originator, which is the object whose state needs to be saved.
2. Create a Memento class to represent the state of the originator at a particular point in time.
3. Create a Caretaker class that keeps track of the history of the originator's state.
4. Write a client code to demonstrate how to use the Memento Pattern to save and restore the state of the originator.