1.

a) what are the classifications of design pattern?name two pattern in each class?

Ans:

1.Creational Patterns:

i.*Factory Method Pattern*: Defines an interface for creating objects, but lets subclasses decide which class to instantiate. It allows a class to defer the instantiation to its subclasses.

ii.*Singleton Pattern*: Ensures a class has only one instance and provides a global point of access to that instance.

2.Structural Patterns:

i.*Adapter Pattern*: Allows objects with incompatible interfaces to work together by p roviding a wrapper or adapter that converts one interface into another.

ii.*Decorator Pattern*: Dynamically adds responsibilities or behaviors to objects without modifying their code directly. It provides a flexible alternative to subclassing for extending functionality.

3.Behavioral Patterns:

i.*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.

ii.*Strategy Pattern*: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the client to choose the algorithm at runtime.

b) Describe the purpose of using factory method pattern?

Ans:

The Factory Method Pattern is a creational design pattern that provides an interface for creating objects but allows subclasses to decide which class to instantiate. The main purpose of using the Factory Method Pattern is to promote loose coupling between the client code and the concrete classes being instantiated. It achieves this by delegating the responsibility of creating objects to subclasses rather than having the client directly instantiate concrete classes.

- Abstraction and Encapsulation:
- Flexibility
- Dependency Inversion Principle
- Extensibility
- Decoupling Object Creation
- Decoupling Object CreationDecoupling Object Creation

c) Define code smell and refactoring.

Ans:

- **Code Smell:**

  Code smell refers to certain characteristics in the source code that indicate potential design or implementation issues, such as duplicated code, long methods, complex logic, and inappropriate naming.

- **Refactoring:**

  Refactoring is the process of restructuring existing code to improve its internal structure, design, and other non-functional attributes without changing its external behavior. It aims to enhance code quality, readability, maintainability, and testability.

d) What are the characteristics of enterprise application.

Ans:

Enterprise applications are large-scale software systems designed to meet the complex and diverse needs of organizations and businesses. They often handle critical business processes, serve a large number of users, and require robustness, scalability, and high performance. Here are some key characteristics of enterprise applications:

- Scalability
- Reliability
- Security
- Integration
- Performance
- Maintainability
- Extensibility
- Compliance
- User Interface (UI) and User Experience (UX)
- Centralized Data Management
- Support and Maintenance

e) Explain pull up field and collapse hierarchy.

Ans:

- **Pull Up Field**:
  - Move a common field from subclasses to the superclass to eliminate duplication.
- **Collapse Hierarchy**:
  - Eliminate unnecessary intermediate subclasses and move their functionalities directly into the superclass.

f) Explain SOLID principle

Ans*:*

SOLID Principles:

1. SRP (Single Responsibility Principle): A class should have only one reason to change - do one thing well.

2. OCP (Open/Closed Principle): Software entities should be open for extension but closed for modification.

3. LSP (Liskov Substitution Principle): Subtypes should be substitutable for their base types without altering program correctness.

4. ISP (Interface Segregation Principle): Clients should not be forced to depend on interfaces they don't use.

5. DIP (Dependency Inversion Principle): High-level modules should not depend on low-level modules; both should depend on abstractions.

g) Describe the purpose of using singleton design pattern.

Ans:

The purpose of using the Singleton design pattern is to ensure that a class has only one instance and provides a global point of access to that instance. It is used when you need to control and restrict the instantiation of a class to a single object, such as a global configuration, logging mechanism, or shared resource, ensuring consistency and avoiding unnecessary duplication of resources.

h) what are the basic feature of OOP?which one is the most important and why?

Ans:

The basic features of Object-Oriented Programming (OOP) are:

1. Encapsulation: Bundling data and methods together within a class to hide internal details and provide a clean external interface.

2. Abstraction: Defining essential characteristics while hiding unnecessary details, allowing for generic designs.

3. Inheritance: Creating hierarchical relationships between classes for code reuse and polymorphism.

4. Polymorphism: Treating objects of different classes uniformly through a common interface.

The most important feature is "Encapsulation" because it enhances code organization, maintenance, and security. It hides internal details, allows changes without affecting clients, and improves information hiding and data protection.

i) "Comments represent a failure to express an idea"-describe with an appropriate example.

Ans:

"Comments represent a failure to express an idea" is a quote attributed to Robert C. Martin, an influential software engineer and author. The statement implies that well-written code should be self-explanatory and expressive enough to convey its intentions without relying heavily on comments. In other words, if code requires excessive comments to be understood, it might indicate that the code itself lacks clarity and readability.

Example -

```
// Bad approach with comments
int result = 0; // Initialize result
for (int i = 1; i <= n; i++) { // Loop to calculate factorial
    result *= i; // Calculate factorial
}

// Better approach without comments
int factorial = 1;
for (int i = 1; i <= n; i++) {
    factorial *= i;
}
```

In the second example, the code is self-explanatory without the comments, making them unnecessary. The code is clear, concise, and expressive, following the principle that "good code does not require comments to explain its intent."

2.

a) ***Expalin what the team feature envy means,giving a short code or diagram example to illustrate the remedies.***

Ans:

Feature Envy is an anti-pattern that occurs when a method in a class is more interested in the features (fields or methods) of another class rather than its own. In other words, the method excessively uses or manipulates the data of another class,

indicating that it might belong to the other class instead. This can lead to poor code organization and decrease maintainability.

Example code illustrating Feature Envy:

```java
class Customer {
    private String name;
    private Address address;

    // Getters and setters...

    public String getAddressCity() {
        return address.getCity(); // Feature Envy - using Address class's feature excessively
    }
}
```

In this example, the `getAddressCity()` method in the `Customer` class is excessively using the `Address` class's feature (`getCity()`). This method seems more interested in the `Address` class's data than its own. It might be more appropriate for the `getCity()` method to be in the `Address` class itself.

Remedies for Feature Envy:

1. Move the method to the class that owns the data:
   ```java
   class Address {
       private String city;

       // Getters and setters...

       // Move the method to the Address class
       public String getCity() {
           return city;
       }
   }

   class Customer {
       private String name;
       private Address address;

       // Getters and setters...
   }
   ```

b) "strategy changes the guts of an object whereas decorator changes skin of an object".Explain this statement with approprite reasoning.

Ans:

The statement "Strategy changes the guts of an object whereas Decorator changes the skin of an object" highlights the fundamental difference between the Strategy and Decorator design patterns and how they affect an object's behavior and structure.

1. Strategy Pattern:
The Strategy pattern is a behavioral design pattern that allows you to define a family of algorithms or strategies, encapsulate each one, and make them interchangeable at runtime. It changes the "guts" or internal behavior of an object by swapping the underlying algorithm, which directly affects how the object performs a certain task.

Reasoning:
In the Strategy pattern, the object itself doesn't change; instead, the internal algorithm it uses is changed. The object delegates the task to a separate strategy object, and different strategies can be easily plugged into the object. This allows for flexibility and adaptability, as you can switch between different strategies without modifying the object's core functionality.

2. Decorator Pattern:
The Decorator pattern is a structural design pattern that allows you to add additional functionalities to an object dynamically. It changes the "skin" or the external appearance of an object by wrapping it with one or more decorators, adding new behaviors or responsibilities to the object without altering its core implementation.

Reasoning:
In the Decorator pattern, the object's original behavior remains intact. Instead, decorators are used to enhance or modify the object's functionality by adding new features to it. Each decorator wraps the original object and adds specific behavior, but the core object's structure and behavior remain unchanged. Decorators can be stacked or combined to provide multiple enhancements to the original object, and they can be added or removed at runtime, providing flexibility and dynamic behavior modification.

In summary, the Strategy pattern changes the internal behavior of an object by swapping algorithms (guts), while the Decorator pattern enhances or extends an object's functionality by adding new responsibilities or behaviors (skin) without changing its core implementation. Both patterns promote code flexibility, but they do so in different ways and for different purposes. The Strategy pattern allows an object to adapt to different algorithms, while the Decorator pattern enhances an object's capabilities while keeping its core intact.

c)Which design pattern provides us a way to reduce memory requirement ? Explain how this memory optimization actually works?what other design pattern automatically comes with this pattern implementation?

Ans:
The Flyweight design pattern provides a way to reduce memory requirements by sharing common state among multiple objects. The key idea behind the Flyweight pattern is to store intrinsic (shared) state externally and maintain a reference to it in multiple objects, while keeping the intrinsic state unchanged and immutable. This way, objects can share the same state, reducing memory consumption and improving performance, especially when dealing with large numbers of similar objects.

How the memory optimization works:

1. Intrinsic and Extrinsic State: In the Flyweight pattern, objects have two types of state - intrinsic and extrinsic state. Intrinsic state represents the shared, immutable part of an object

that can be shared among multiple objects. Extrinsic state represents the unique, context-specific part of an object.

2. External State Storage: The intrinsic state is stored externally in a Flyweight Factory, which is responsible for creating and managing Flyweight objects. The Flyweight Factory ensures that only one instance of the intrinsic state is created and shared among the objects that require it.

3. Sharing Intrinsic State: When an object needs to use the shared intrinsic state, it requests it from the Flyweight Factory. This way, multiple objects can share the same intrinsic state, reducing memory consumption.

By utilizing the Flyweight pattern, memory optimization is achieved by minimizing the storage of redundant data across multiple objects. Instead of duplicating the common state for each object, the pattern allows them to share a single instance, resulting in significant memory savings.

Other design pattern that comes with this pattern implementation:

The Composite design pattern often comes hand-in-hand with the Flyweight pattern. In the Composite pattern, a tree-like structure is created, where individual objects (leaves) and composite objects (containers) are treated uniformly. When using the Flyweight pattern in conjunction with the Composite pattern, the intrinsic state is typically shared among all the leaf objects, while the composite objects handle the extrinsic state specific to each individual object in the tree. This combination allows for efficient memory usage, especially when dealing with complex hierarchical structures containing a large number of similar objects.

d)which patterns seem to be similar to proxy pattern?Explain why they are similar.What are the dissimilarities that made these patterns exist separately?
Ans:
Two patterns that seem to be similar to the Proxy pattern are the Decorator pattern and the Adapter pattern. While they share some similarities with the Proxy pattern, each has distinct purposes and implementations that set them apart.

1. Decorator Pattern:

Similarity:
Both the Proxy and Decorator patterns involve wrapping an object to control access or add additional behavior. In both cases, the wrapper (Proxy or Decorator) and the wrapped object have the same interface, allowing them to be used interchangeably.

Difference:
The main difference lies in their intent and focus. The Proxy pattern focuses on controlling access to the underlying object, while the Decorator pattern focuses on adding responsibilities or behaviors to the object without changing its interface. Proxies can add additional functionalities like lazy initialization, access control, caching, or logging, while Decorators are primarily used for adding or modifying an object's behavior.

2. Adapter Pattern:

Similarity:

Both the Proxy and Adapter patterns act as intermediaries between the client code and the underlying object. They both provide a way to interact with an object through an intermediate layer.

Difference:
The key difference is in their purposes and use cases. The Proxy pattern serves to control access or provide additional features to the underlying object. On the other hand, the Adapter pattern is used to make two incompatible interfaces work together. It acts as a bridge between the client and the target interface, allowing them to interact seamlessly.

Reasons for Separate Existence:

The Proxy, Decorator, and Adapter patterns serve different design goals and scenarios:

1. Proxy Pattern: It is used to control access to an object, add lazy initialization, provide logging or security checks, and create a level of indirection to manage the underlying object's complexity.

2. Decorator Pattern: It is used to add or modify an object's behavior transparently without altering its interface. It is primarily focused on providing additional responsibilities dynamically.

3. Adapter Pattern: It is used to make two incompatible interfaces work together. It enables objects with different interfaces to collaborate without modifying their code.

While there are some similarities among these patterns in terms of their use of intermediaries, their core intents and functionalities differ significantly, leading to their separate and distinct existence in the realm of design patterns.

e)Can you use the memento design pattern along with the commad pattern?If yes then explain with an example

Ans:

Yes, the Memento design pattern and the Command design pattern can be used together to implement a feature that supports undo and redo functionality in an application.

4.

a) Can we replace interface with abstract class?justify your answer.

Ans:

Yes, in some cases, an abstract class can replace an interface. Abstract classes can provide a default implementation, shared state, and backward compatibility. However, interfaces are preferable for multiple inheritance, polymorphism, and strict contract enforcement. The choice depends on the specific needs of the application.

b) what does Delegation and Object Composition means?

Ans:

Delegation:

- Delegation is when an object forwards certain tasks to another object to perform on its behalf.
- Promotes code reuse and separation of concerns.
- Allows collaboration without tightly coupling implementations.

Example: PrinterController delegates printing to InkjetPrinter or LaserPrinter.

Object Composition:
- Object Composition is creating complex objects by combining smaller, simpler objects.
- Allows flexible and modular code construction.
- Avoids deep inheritance hierarchies.

Example: Car composed of an Engine object instead of inheriting from an Engine class.

c) what is remote facade?

Ans:
In simple terms, a Remote Facade is a design pattern that provides a simple and unified interface for clients to interact with complex remote systems. It shields clients from the complexities of remote communication, making it easier to use and maintain the remote services. Clients interact with a single facade instead of dealing with multiple remote components directly.

d) Describe the motivation of Observer pattern.

Ans:
The motivation behind the Observer pattern is to establish a one-to-many dependency between objects, so that when one object (the subject) changes state, all its dependents (observers) are automatically notified and updated. This allows objects to be loosely coupled, enabling changes in one object to propagate to multiple other objects without them needing to know the details of each other. The Observer pattern promotes flexibility, reusability, and maintainability by decoupling the subject from its observers.

e) write down two example of Enterprise applicaton.

Ans:
1. Customer Relationship Management (CRM) System: Manages customer interactions, sales leads, and marketing campaigns for businesses.

2. Enterprise Resource Planning (ERP) System: Integrates and manages various business processes, including finance, human resources, inventory, and production, across an organization.

5.

a)what are the patterns that seems to be similar to composite patterns?Explain why they are similar,what are the dissimilarities that made these patterns exist separately?

Ans:
Patterns that seem to be similar to the Composite pattern are the Decorator pattern and the Visitor pattern. Let's explore the similarities and differences between these patterns:

1. Decorator Pattern:

Similarity:
Both the Composite and Decorator patterns involve composing objects in a tree-like structure. In both cases, the patterns utilize a recursive composition of objects to create hierarchical structures.

Difference:
The main difference lies in their intent and use cases. The Composite pattern is used to treat individual objects and composite objects (containers) uniformly. It allows clients to interact

with both types of objects in a transparent manner. In contrast, the Decorator pattern is used to add responsibilities or behaviors to an object dynamically without modifying its structure. It enhances the functionality of a single object, rather than treating multiple objects as a unified entity.

2. Visitor Pattern:

Similarity:
The Visitor pattern also involves traversing a composite structure, similar to the Composite pattern. Both patterns allow you to visit each element in the structure and perform operations on them.

Difference:
The Visitor pattern focuses on separating algorithms or operations from the elements on which they operate. It allows you to add new operations to a complex structure without modifying the classes of the elements. The Composite pattern, on the other hand, is about treating individual objects and composites uniformly, organizing them into a tree-like structure.

Dissimilarities:

1. Intent:
- Composite Pattern: Organizes objects into tree-like structures to create a whole-part hierarchy and treat individual objects and composites uniformly.
- Decorator Pattern: Adds responsibilities or behaviors to an object without modifying its structure.
- Visitor Pattern: Separates operations from the elements on which they operate, allowing for dynamic addition of new operations.

2. Use Cases:
- Composite Pattern: Best suited for representing whole-part hierarchies and tree-like structures.
- Decorator Pattern: Useful for adding responsibilities or behaviors dynamically to objects.
- Visitor Pattern: Suitable for performing operations on a complex structure without modifying its elements.

3. Focus:
- Composite Pattern: Focuses on creating a unified interface for individual objects and composites.
- Decorator Pattern: Focuses on enhancing the behavior of a single object.
- Visitor Pattern: Focuses on separating algorithms from the elements they operate on.

In summary, the Composite, Decorator, and Visitor patterns share similarities in how they structure and traverse objects in a hierarchical manner. However, they have distinct intents and serve different purposes, which led to their separate existence as independent design patterns.

b)Name two different patterns that are used to eliminate if else statement in code? Give and example each of these two patterns expalining how these patterns eleiminationg if eles statement.
Ans:

Two different patterns that are used to eliminate if-else statements in code are the Strategy pattern and the Chain of Responsibility pattern.

Strategy Pattern:

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows you to select the desired algorithm at runtime without using conditional statements.

```
// Strategy interface
interface DiscountStrategy {
    double applyDiscount(double amount);
}

// Concrete strategy classes
class RegularCustomerDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double amount) {
        return amount * 0.1; // 10% discount for regular customers
    }
}

class NewCustomerDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double amount) {
        return amount * 0.15; // 15% discount for new customers
    }
}

class HolidaySaleDiscount implements DiscountStrategy {
    @Override
    public double applyDiscount(double amount) {
        return amount * 0.2; // 20% discount during holiday sale
    }
}

// Context class
class DiscountCalculator {
    private DiscountStrategy discountStrategy;

    public void setDiscountStrategy(DiscountStrategy discountStrategy) {
        this.discountStrategy = discountStrategy;
    }

    public double calculateDiscount(double amount) {
        return discountStrategy.applyDiscount(amount);
    }
}

// Client code
public class Main {
    public static void main(String[] args) {
        DiscountCalculator calculator = new DiscountCalculator();
```

```
        // Select discount strategy dynamically
        // No need for if-else statements
        calculator.setDiscountStrategy(new RegularCustomerDiscount());
        double discountedAmount = calculator.calculateDiscount(100.0);
        System.out.println("Discounted amount: " + discountedAmount);
    }
}
```

In this example, the Strategy pattern eliminates the need for if-else statements to determine the discount. The DiscountCalculator class can use different DiscountStrategy implementations to calculate the discount based on the customer type, providing a cleaner and more maintainable solution.

Chain of Responsibility Pattern:

The Chain of Responsibility pattern creates a chain of handler objects, each responsible for processing a specific request. The request is passed along the chain until it is handled, removing the need for if-else cascades.

```
// Handler interface
interface Logger {
    void setNextLogger(Logger nextLogger);
    void logMessage(LogLevel level, String message);
}

// Concrete handler classes
class ConsoleLogger implements Logger {
    private Logger nextLogger;

    @Override
    public void setNextLogger(Logger nextLogger) {
        this.nextLogger = nextLogger;
    }

    @Override
    public void logMessage(LogLevel level, String message) {
        // Log to console
    }
}

class FileLogger implements Logger {
    private Logger nextLogger;

    @Override
    public void setNextLogger(Logger nextLogger) {
        this.nextLogger = nextLogger;
    }

    @Override
    public void logMessage(LogLevel level, String message) {
        // Log to file
    }
}
```

```
class EmailLogger implements Logger {
    @Override
    public void setNextLogger(Logger nextLogger) {
        // EmailLogger is the last in the chain
    }

    @Override
    public void logMessage(LogLevel level, String message) {
        // Send log via email
    }
}

// LogLevel enumeration
enum LogLevel {
    DEBUG, INFO, WARNING, ERROR
}

// Client code
public class Main {
    public static void main(String[] args) {
        Logger consoleLogger = new ConsoleLogger();
        Logger fileLogger = new FileLogger();
        Logger emailLogger = new EmailLogger();

        // Chain the loggers
        consoleLogger.setNextLogger(fileLogger);
        fileLogger.setNextLogger(emailLogger);

        // Start logging
        consoleLogger.logMessage(LogLevel.ERROR, "An error occurred.");
    }
}
```

c)what does object inheritance and object composition mean?what are the advantages and disadvantages of these two approaches while trying to achieve reusability?

Ans:

Object Inheritance:

- Inheritance allows a subclass to inherit properties and behaviors from a superclass.
- Advantages: Code reuse, polymorphism, clear relationships.
- Disadvantages: Tight coupling, fragile base class problem, limited flexibility.

Object Composition:

- Composition involves one class containing other objects as member variables.
- Advantages: Code reuse, flexibility, encapsulation.
- Disadvantages: Increased complexity, indirect access, learning curve.

Choose between inheritance and composition based on design goals and system complexity. Often, a mix of both is used for reusability and maintainability.

d)what is the motivation behind Memento pattern? Draw the UML or class diagram for memento pattern.Identify the participants and their roles from diagram.

Ans:

The motivation behind the Memento pattern is to capture the internal state of an object at a particular moment and store it externally so that the object can be restored to that state later if needed. It allows an object to be restored to a previous state without revealing its internal details, promoting encapsulation and providing the ability to undo or rollback changes.

Participants and their roles in the Memento pattern:

1. Originator:
   - Creates and holds the object's internal state.
   - Creates a Memento object to store its state.
   - Can restore its state from a Memento.

2. Memento:
   - Stores the internal state of the Originator.
   - Should have no direct access to the state except for the Originator that created it.
   - Immutable or effectively immutable to prevent external modification.

3. Caretaker:
   - Manages and keeps track of Memento objects.
   - Requests the Originator to save and restore its state using Mementos.
   - Acts as a caretaker or guardian for the Originator's state history.

In summary, the Memento pattern provides a mechanism to save and restore an object's state without exposing its internal details. The Originator creates and maintains its state, the Memento stores the state, and the Caretaker manages the Mementos, enabling undo and rollback functionality in an application.