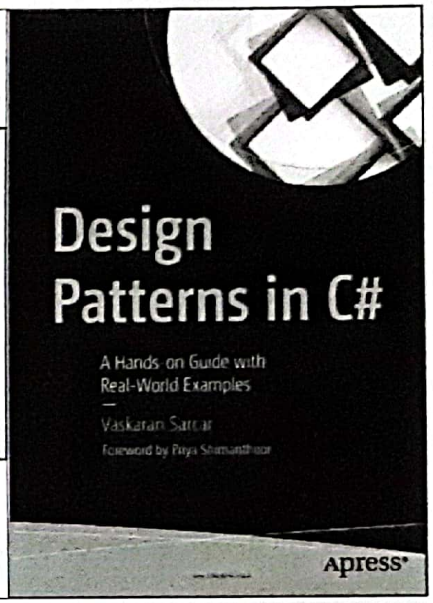
Software Design Patterns & Frameworks

Dinesh Asanka

References

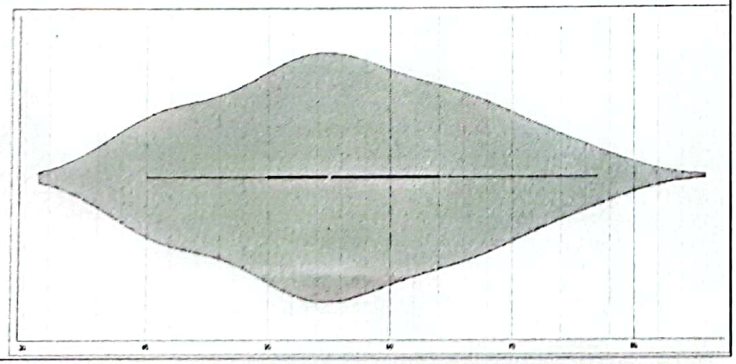
UNIVERSITY OF ALBERTA
FACULTY OF SCIENCE

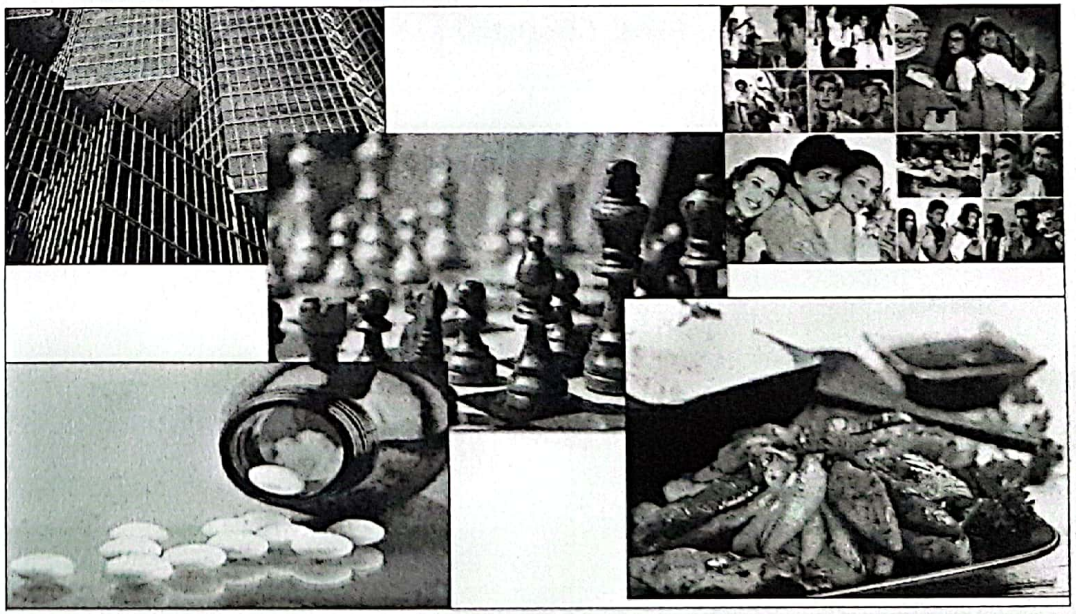
Department of Computing Science



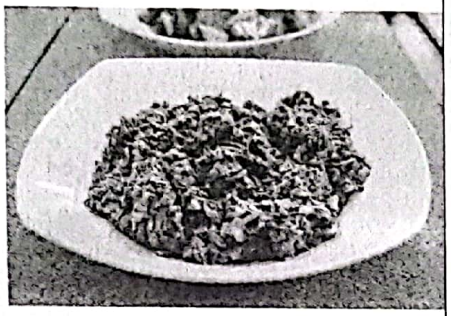
Assessment

Method	Description	Allocation
Quiz / In Class	Daily, 2 Marks each In Class	20%
Exam	Open Book	80%



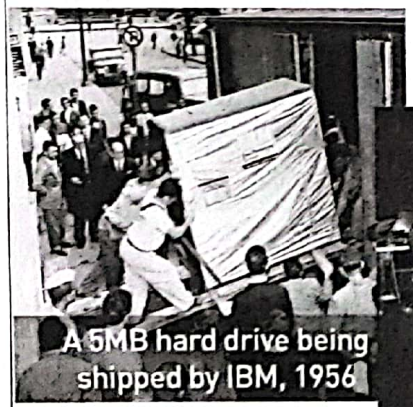


How do you cook?

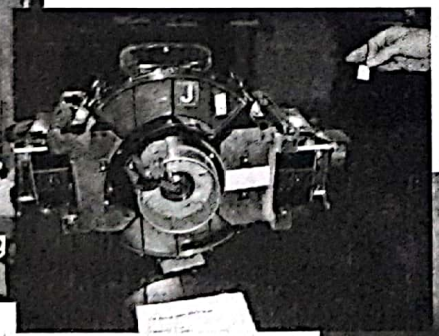


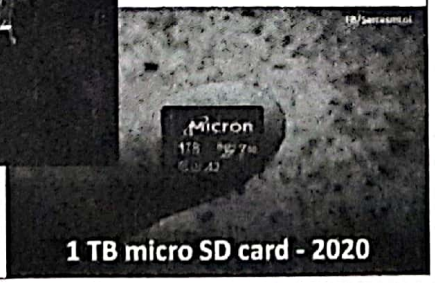
Patterns may change with Environment & Time

How Software Design have changed



A 5MB hard drive being shipped by IBM, 1956





1 TB micro SD card - 2020

What is Design Patterns?

- The concept of design pattern was introduced in building architecture by Christopher Alexander in the 1970s
- Building structures and town planning should be supported by design patterns.

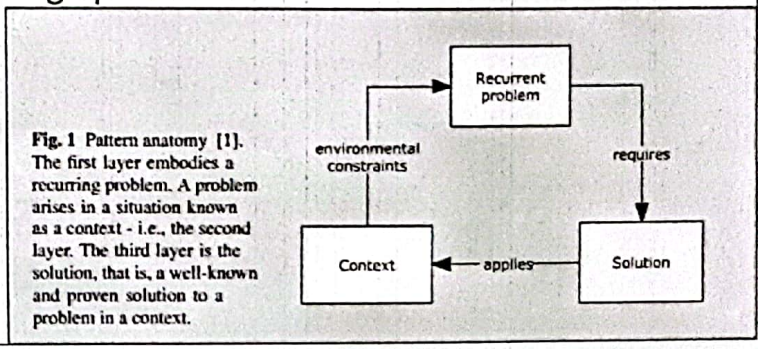
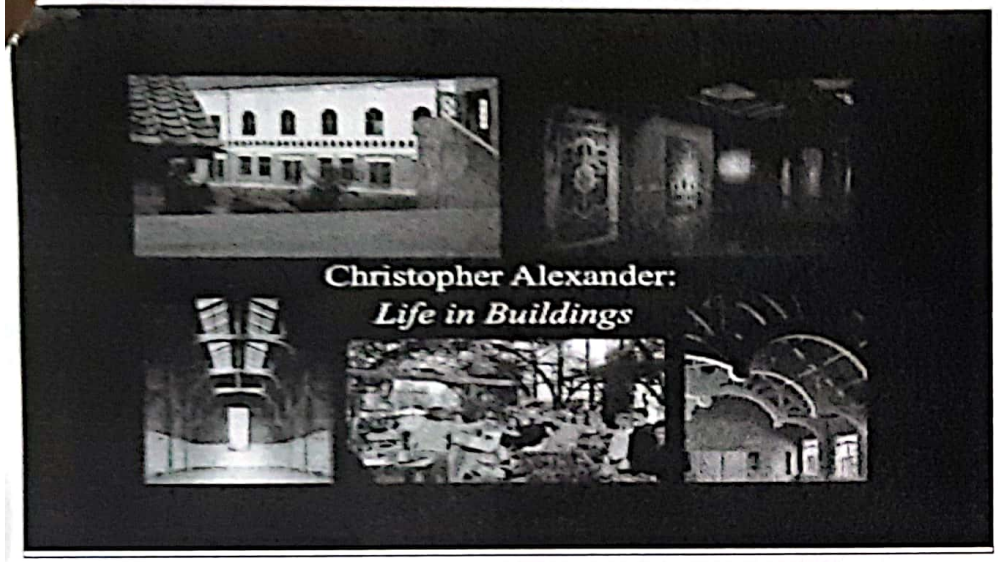


Fig. 1 Pattern anatomy [1]. The first layer embodies a recurring problem. A problem arises in a situation known as a context - i.e., the second layer. The third layer is the solution, that is, a well-known and proven solution to a problem in a context.



GoF Design Patterns Catalog

- Later the Gang of Four - Design patterns, elements of reusable object-oriented software book was written by a group of four persons named as Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides in 1995.
- That's why all the above 23 Design Patterns are known as Gang of Four (**GoF**) Design Patterns.

↑ for software development

What is Design Patterns

- Practical, effective, and proven solution for recurring problem.
- Design Patterns are used instead of using basic OOP & Programming concepts
  - Inheritance
  - Interface
  - For Loop
  - IF .. ELSE
- Design Patterns are NOT theoretical solutions.
- 23 Design Patterns are defined.

Important

- Design patterns don't guarantee an absolute solution to a problem.
  - You may have to modify the design pattern
  - You may have to combine two or more design patterns
- They provide clarity to the system architecture and the possibility of building a better system.

Anti-Patterns

* Antipatterns are the things what we should not do for the problems.

- Common mistakes
- What you should not do
- Tactical solutions that lead to long term problems.
  - "We need to deliver the product as soon as possible."
  - "Currently we do not need to analyze the impact."
  - "I am an expert of reuse. I know design patterns very well."
  - "We will use the latest technologies and features to impress our customers. We do not need to care about legacy systems."
  - "More complicated code will reflect my expertise in the subject."

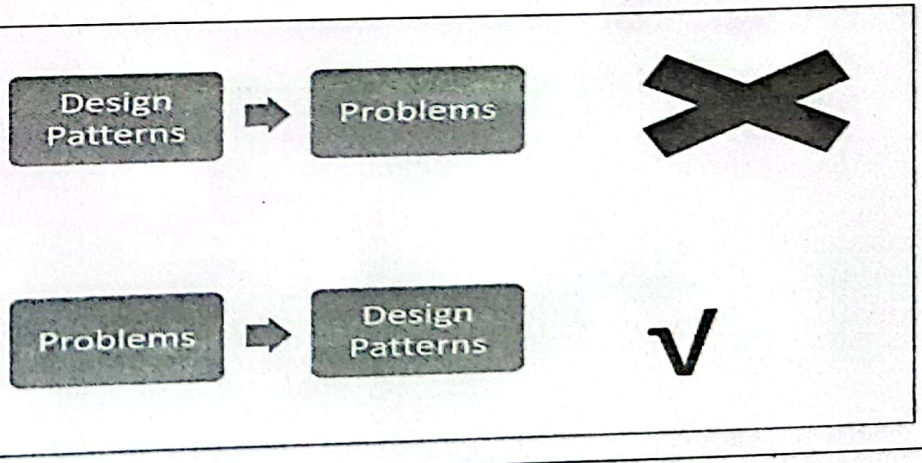
Design Patterns

Creational (5)	Structural (7)	Behavioral (11)
• Factory	• Adapter	• Chain Of Responsibility
• Abstract Factory	• Bridge	• Command
• Singleton	• Composite	• Interpreter
• Prototype	• Decorator	• Iterator
• Builder	• Facade	• Mediator
	• Flyweight	• Memento
	• Proxy	• Observer
		• State
		• Strategy
		• Template
		• Visitor

- Microservices.

Pattern Type	Description
Creational	How objects are initiated. Initiating or Cloning
Structural	How objects are connected to each other
Behavioural	How objects are behaving to achieve common objective

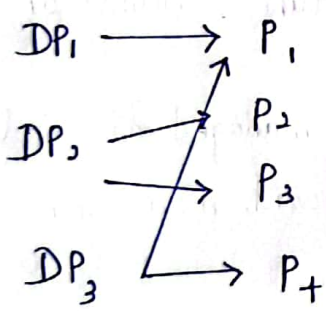
Remember



Software Design Patterns and Framework.

* Patterns always change with the time and the environment.
* Design pattern is a practical, effective and a proven solution for a recurring problem.
  (Always the design patterns should be well known, proved and practical. Design patterns are always applied for the recurring problems.) The problem should always be recurring.
* Design patters are used to solve problems. It is a solution.
* Design pattern is a kind of template. (what we should do to solve the problem).
* Design patterns of buildings paved the path to design patter
* Next, Gang of Four design patterns (23 patterns) came to us

DP → Design pattern
P → problem



* These design patterns are not the absolute solution for a problem. We should get the pattern, change it, combine it (2 o more) design patterns to get the required solution.

* Designs provide clarity ~~so~~ to the architect so, they can develop better software.

Design pattern —solves→ Problem

studing process

reality happens.

design patterns (23)
- Creational (5) – How you create the object eg:- from classes we get objects
- Structural (7) – How objects are related
- behavioral (11) – How objects are performing.

First identify to which category or categories the problem belongs to

01 Singleton design pattern

* Creational design pattern / to create objects.
* One class with one instance.
* Singleton design pattern ensures that class has only one instance and provides a way to access that instance from any point in the application.

Possible implementations.

(01) Game settings:

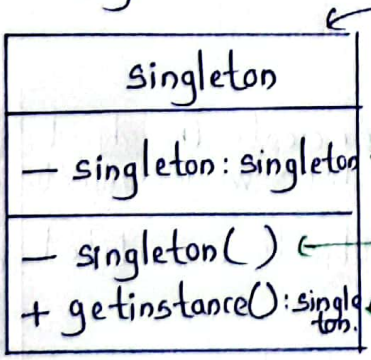
I. Game Manager → Can use a singleton for a game manager that oversees various aspects of the game as managing levels, ~~stori~~ scoring and game progression.
Here game manager can be accessed from different game objects to communicate or retrieve global information.

II Resource manager → Manage resource like textures, sounds or models.
* Resources are loaded and managed effciently.

III Input manager → Handle input devices keyboard, mouse or controllers.

- Limitations ⟹ overusing singleton cause.
  hard to control
  make the code less modular.

* class diagram of the singleton design pattern;



← single class.
(private static variable — to keep the single instance/object created). Made only once.
← private instance (same type of class)
← private constructor — to restrict creating instances from other classes.
← public method (return type is a same object type as the class)
// Public static method (return the instance of the class global access point)

- → private
+ → public.

~~Singleton Design pattern~~ ~~Creational design pattern~~ *

Singleton design pattern

1) early instantiation
created at the loading time
* increase startup time and resource usage.

2) lazy instantiation.
created when required.
* more optimized.
* used in multithreaded environme

Factory Method Design Pattern.

* Creational design pattern.

· In factory design pattern it defines an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.

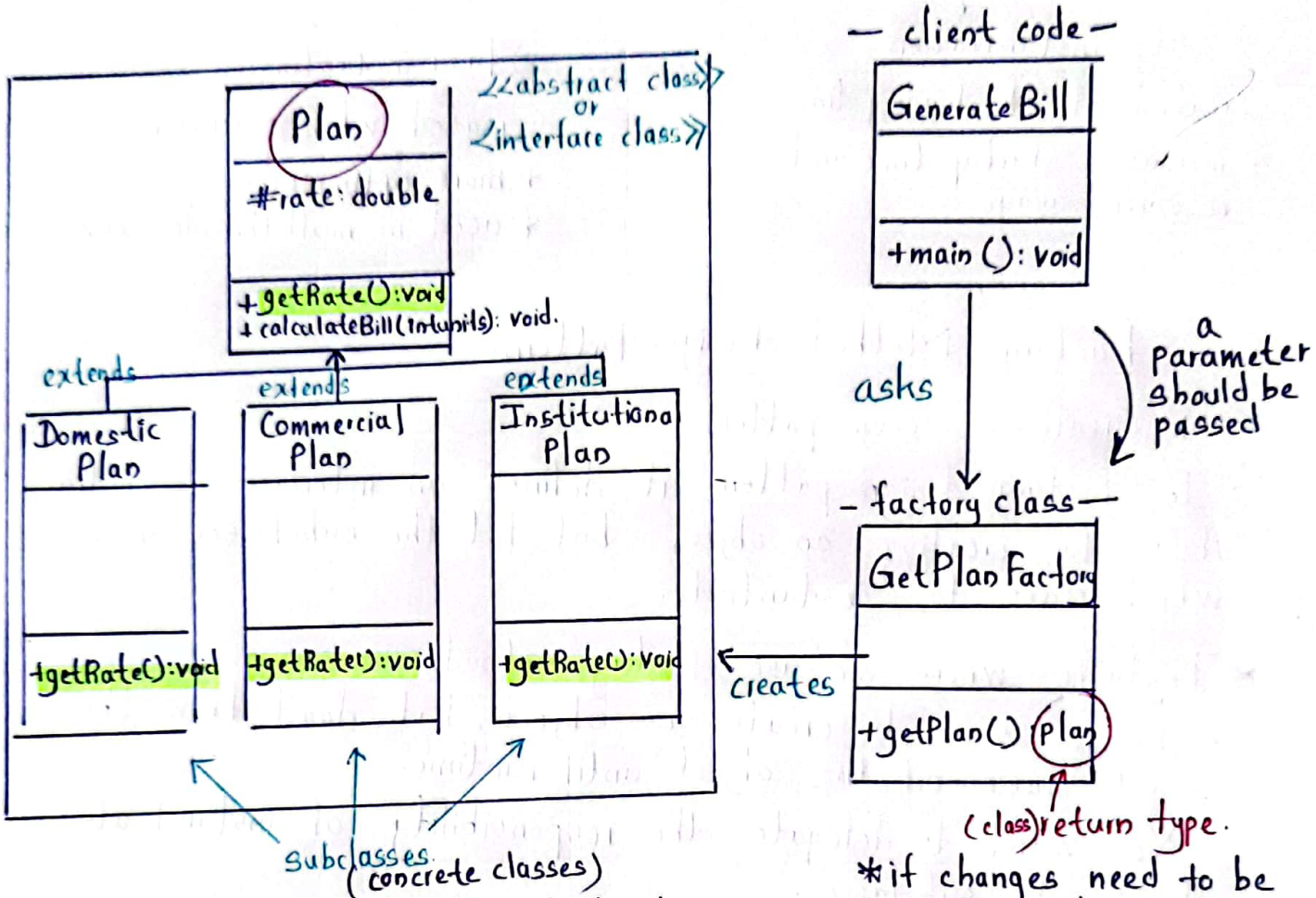
* Instance where we use factory method,
· If we want to create an object, but don't know the exact class of the object until runtime.
· You want to delegate the responsibility of instantiating objects to subclasses.

* Process;
· Define an interface/abstract class for creating an object.
· Let the subclasses implement this interface, providing their ow concrete implementation of the factory method.
· Factory method is responsible for creating an instance of a cla which may be a subclass of the defined interface.

* Here the object creating code is encapsulated.

* Used for other patterns as Prototype, Builder etc...

ex:- • class diagram:

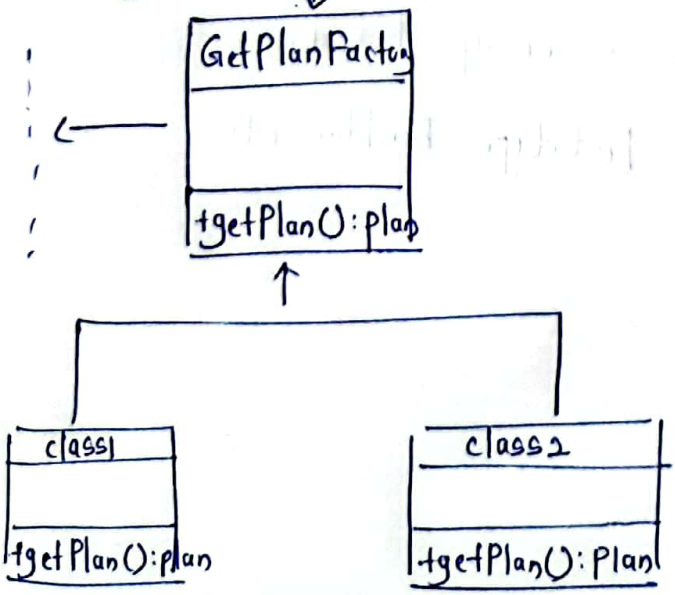


→ same action, different objects. (overrides)

*if changes need to be done they should be done to the factory class.

* Client code always contact with the factory method (with a parameter.

The above diagram shows the factory method, in factory method design pattern the object creation of the factory class is ~~also~~ delegated for subclasses.



Design Patterns

Singleton & Factory Method

Design Patterns

Creational (5)	Structural (7)	Behavioral (11)
• Factory	• Adapter	• Chain Of Responsibility
• Abstract Factory	• Bridge	• Command
• Singleton	• Composite	• Interpreter
• Prototype	• Decorator	• Iterator
• Builder	• Facade	• Mediator
	• Flyweight	• Memento
	• Proxy	• Observer
		• State
		• Strategy
		• Template
		• Visitor

Singleton Pattern

- Way to create an object.
- Simplest example of design pattern, but powerful techniques.
- Enforces one and only one object of a singleton class
- Define a class that has only one instance and provides a global point of access to it
- Saves memory because object is not created at each request. Only single instance is reused again and again.
- There are two forms of singleton design pattern

01) Early Instantiation: creation of instance at load time.

02) Lazy Instantiation: creation of instance when required.

Possible Implementations

- Game Settings
- Printer Queue
- Software Drivers
- Building Automation Solutions
- Database Connection

when accessing external resources / hardware.
(The driver/manager class.) / To centralize the access of to the resource.

* Driver / Manager classes
* To read configurations in an application. (only read one time)
* When cashing an application (single access point)

* To build other design patterns like Facade, Builder, Prototype. etc...

Implementation

```
public class NotSingleton {

    public NotSingleton() {
        -
    }
}
```

```
public class ExampleSingleton { // lazy construction
    // the class variable is null if no instance is
    // instantiated
    private static ExampleSingleton uniqueInstance = null;

    private ExampleSingleton() {
        -
    }

    // lazy construction of the instance
    public static ExampleSingleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new ExampleSingleton();
        }

        return uniqueInstance;

    }
    -
}
```

instance is created when required.

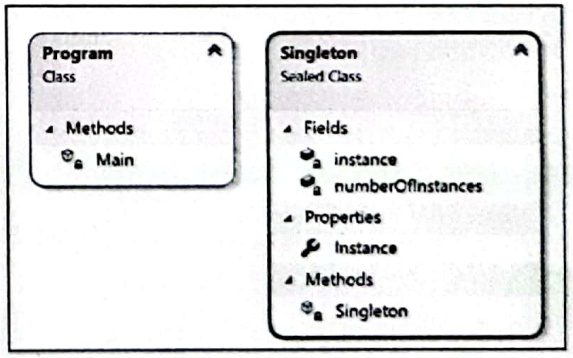
Reference

Page Number 05 - 11

Additional Notes:

A sealed class, in C#, is a class that cannot be inherited by any class but can be instantiated.

The design intent of a sealed class is to indicate that the class is specialized and there is no need to extend it to provide any additional functionality through inheritance to override its behavior.

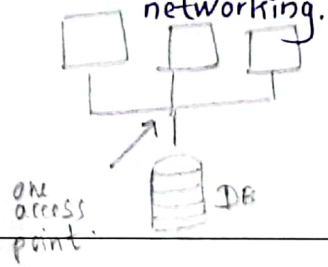


Group Exercise

- What are the places that you can implement singleton class.

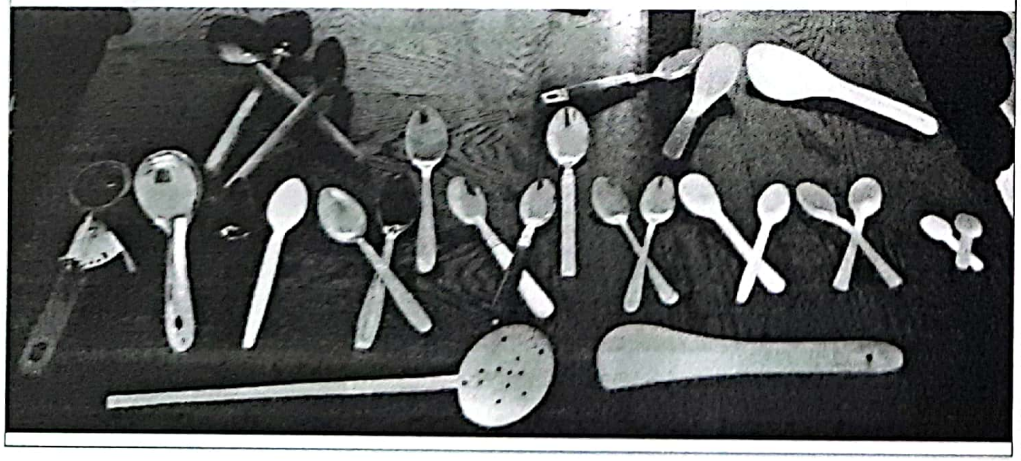
* Instances that should only have one class and provide a global point of access.
  - configuration management
  - database connecting pools
  - resource managers
  - printer queues
  - caching mechanisms
  - device drivers
  - user authentication
  - game managers.

- http client in networking.



Factory Method Pattern

Same Type, Different usages



```
Knife orderKnife(String knifeType) {

    Knife knife;

    // create Knife object - concrete instantiation
    if (knifeType.equals("steak")) {
        knife = new SteakKnife();
    } else if (knifeType.equals("chefs")) {
        knife = new ChefsKnife();
    }

    // prepare the Knife
    knife.sharpen();
    knife.polish();
    knife.package();

    return knife;

}
```

```
Knife orderKnife(String knifeType) {

    Knife knife;

    // create Knife object - concrete instantiation
    if (knifeType.equals("steak")) {
        knife = new SteakKnife();
    } else if (knifeType.equals("chefs")) {
        knife = new ChefsKnife();
    } else if (knifeType.equals("bread")) {
        knife = new BreadKnife();
    } else if (knifeType.equals("paring")) {
        knife = new ParingKnife();
    }

    // prepare the Knife
    knife.sharpen();
    knife.polish();
    knife.package();

    return knife;
```

Common Properties

- Different objects
- Same actions

Solution: Delegate the object creation for a Factory class

```
public class KnifeFactory {
    public Knife createKnife(String knifeType) {
        Knife knife = null;

        // create Knife object
        if (knifeType.equals("steak")) {
            knife = new SteakKnife();
        } else if (knifeType.equals("chefs")) {
            knife = new ChefsKnife();
        }

        return knife;
    }
```

```
public class KnifeStore {

    private KnifeFactory factory;

    // require a KnifeFactory object to be passed to
    // this constructor:
    public KnifeStore(KnifeFactory factory) {
        this.factory = factory;
    }

    public Knife orderKnife(String knifeType) {

        Knife knife;

        // use the create method in the factory
        knife = factory.createKnife(knifeType);

        // prepare the Knife
        knife.sharpen();
        knife.polish();
        knife.package();

        return knife;
    }
```

Advantages

- Easy unit testing
- Increase of code usability
- Improve code readability
- Code extensibility
- Code transparency

Unit Testing is a software testing technique by means of which individual units of software i.e. group of computer program modules, usage procedures, and operating procedures are tested to determine whether they are suitable for use or not.

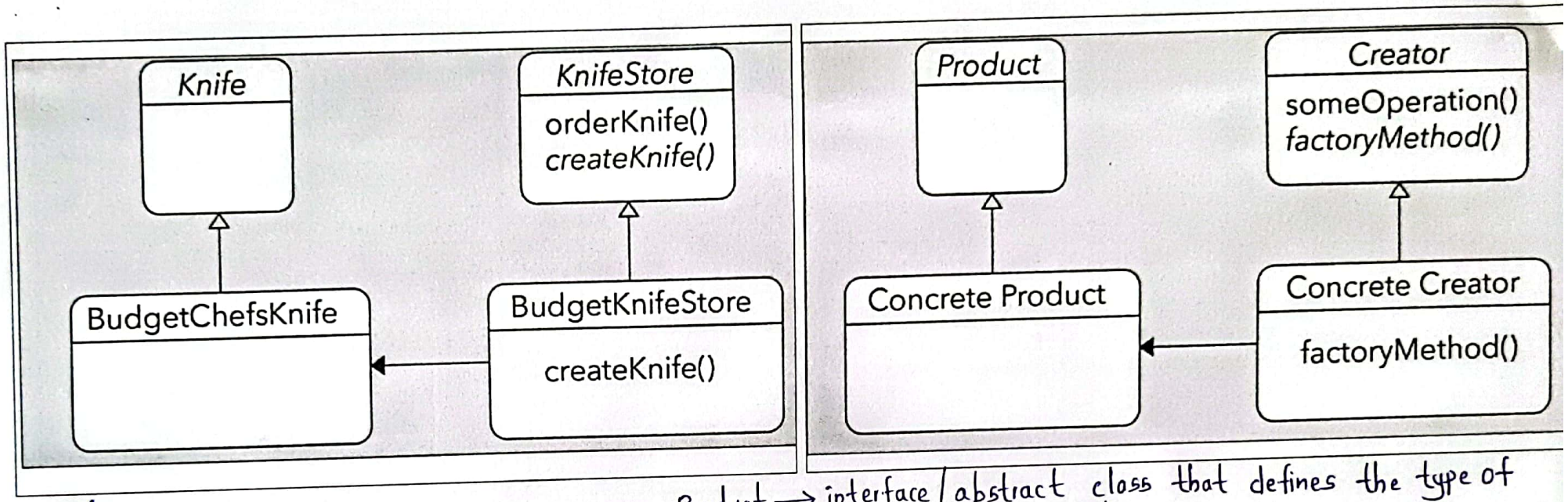
```
    public Knife orderKnife(String knifeType) {
        Knife knife;

        // now creating a knife is a method in the class
        knife = createKnife(knifeType);

        // this is still the same as before!
        knife.sharpen();
        knife.polish();
        knife.package();

        return knife;
    }

    abstract Knife createKnife(String knifeType);

}
```

Abstract Classes

- Though abstract methods should be declared inside a abstract class, it is not required to have abstract methods for every abstract class.
- There is no any possibility to create objects from an abstract class. Only possible way is to create an object through a child class of that abstract class if it is inherited.

Sub Classes

```
public BudgetKnifeStore extends KnifeStore {

    // up to any subclass of KnifeStore to define this method
    Knife createKnife(String knifeType) {
        if (knifeType.equals("steak")) {
            return new BudgetSteakKnife();
        } else if (knifeType.equals("chefs")) {
            return new BudgetChefsKnife();
        }
        //.. more types
        else return null;
    }
}
```



class diagram.

Product → interface / abstract class that defines the type of object to be created.

Concreteproduct → class that implements product interface.

Creator → This is the abstract or interface class that declares the factory method which returns an object of type Product.

ConcreteCreator → Class that implements the creater interfa and overrides the factory method to create a specific concrete product.

Facade Design Pattern

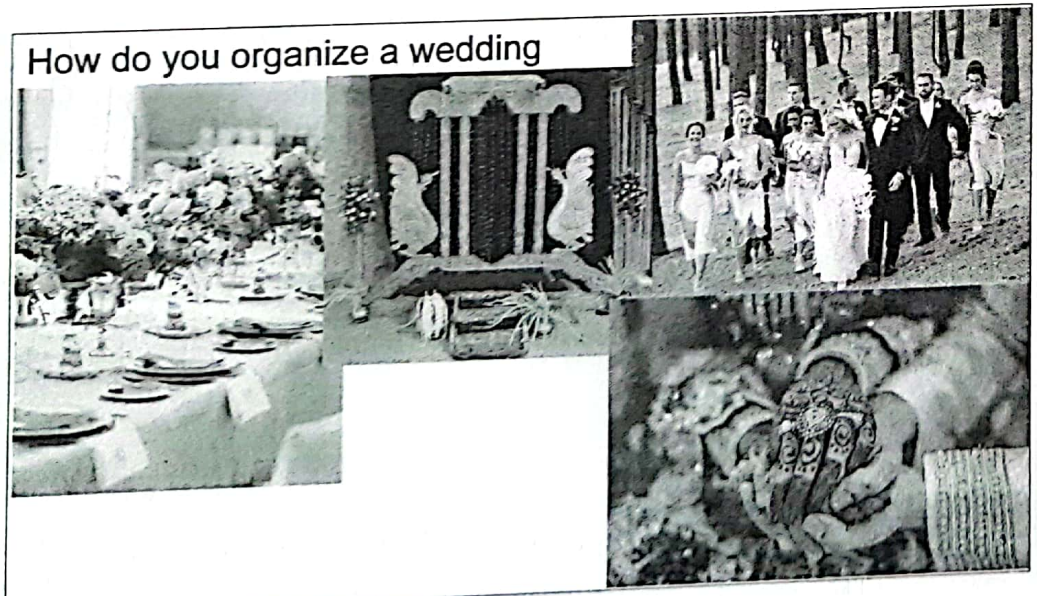
1

Design Patterns

Creational (5)	Structural (7)	Behavioral (11)
• Factory	• Adapter	• Chain Of Responsibility
• Abstract Factory	• Bridge	• Command
• Singleton	• Composite	• Interpreter
• Prototype	• Decorator	• Iterator
• Builder	• Facade	• Mediator
	• Flyweight	• Memento
	• Proxy	• Observer
		• State
		• Strategy
		• Template
		• Visitor

2

How do you organize a wedding



FAGADE Design Pattern

- Provide a unified interface to a set of interfaces in a subsystem.
- Facade defines a higher-level interface that makes the subsystem easier to use.
- Structural type design pattern
- Wrapper class that encapsulates a subsystem in order to hide subsystem's complexity.
- Can be used in complex systems that have multiple and different sub systems..
- Reduce dependencies of outside code on the inner workings of a library.

4

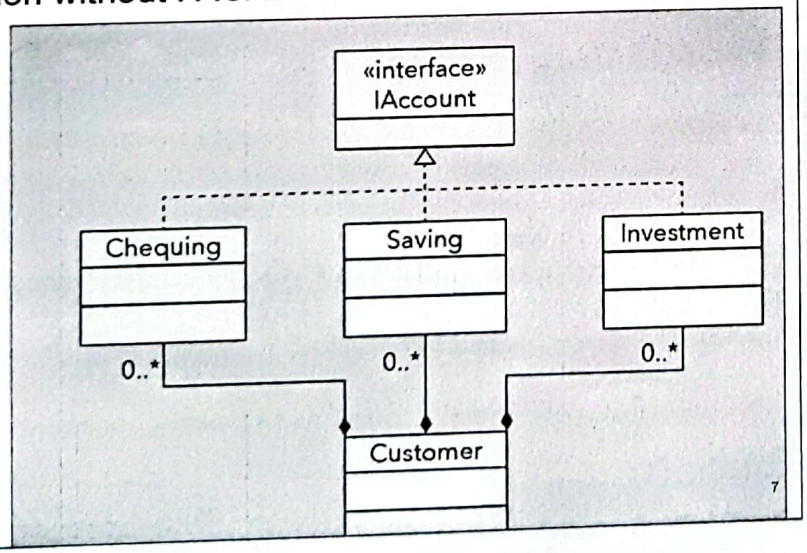
Think about a waiter

5

- Facade design pattern does NOT add any functionality. We should try to avoid add any functionally to the interface.
- Facade design pattern is point of entry into system / sub system.
- Facade is a wrapper class that encapsulates a subsystem in order to hide the sub system complexity.
- It interfaces multiple classes to interact.
- User should have the option of directly connect to the subsystems if needed without the interface.

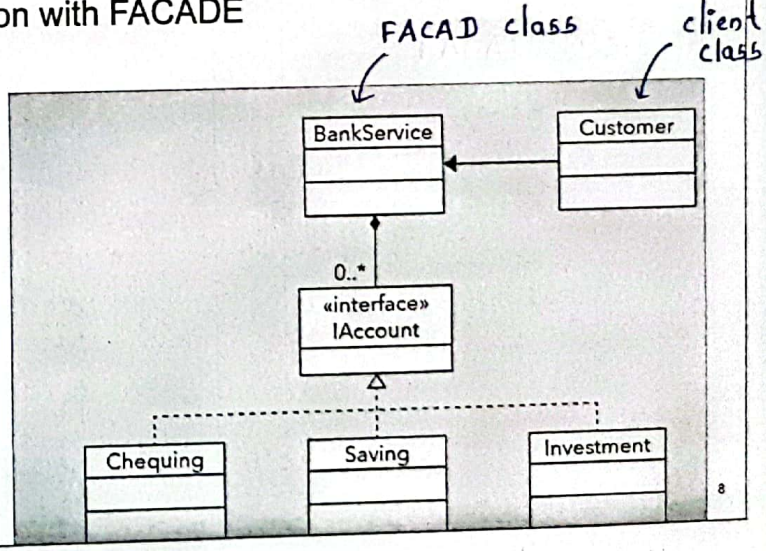
6

Implementation without FACADE



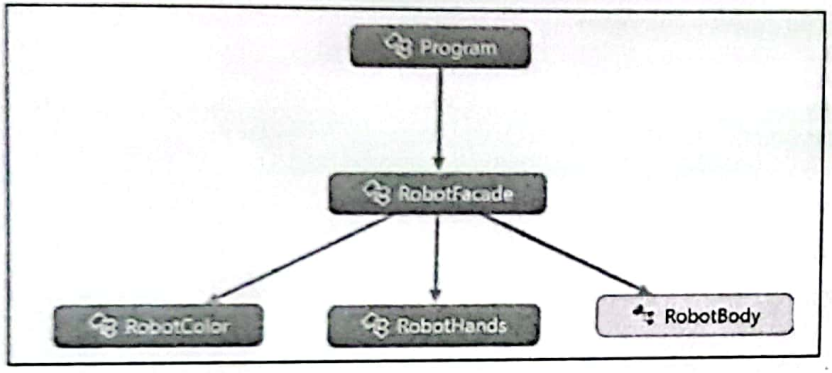
7

Implementation with FACADE

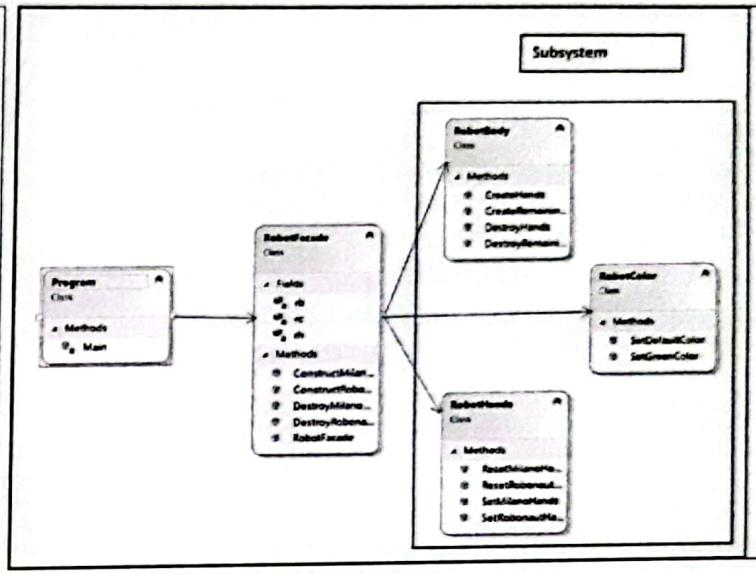


8

Refer 116 - 120



9



10

What are the two conditions required for you to use the facade design pattern?

1. You need a class to act as an interface between your subsystem and a client class.
2. You need a class to translate messages between two existing subsystems because one is expecting a specific interface to use but is provided with an interface that is incompatible.
3. You need to simplify the interaction with your subsystem for client classes.
4. You need a class to that will instantiate other classes within your system and provide these instances to a client class.

11

What are the two conditions required for you to use the facade design pattern?

1. **You need a class to act as an interface between your subsystem and a client class.**
2. You need a class to translate messages between two existing subsystems because one is expecting a specific interface to use but is provided with an interface that is incompatible.
3. **You need to simplify the interaction with your subsystem for client classes.**
4. You need a class to that will instantiate other classes within your system and provide these instances to a client class.

12

Step 1: Design the interface

```
public interface IAccount {
      public void deposit(BigDecimal amount);
      public void withdraw(BigDecimal amount);
      public void transfer(BigDecimal amount);
      public int getAccountNumber();
}
```


What is an interface class?

- A Java interface is a bit like a class, except a Java interface can only contain method signatures and fields.
- Interfaces will define the specifications for Classes that govern how code will behave.
- An Java interface cannot contain an implementation of the methods, only the signature (name, parameters and exceptions) of the method.
- In interface, variables can be defined only as constants.

```
interface A{
      void m();
      int x=0;
}
```


Step 2: Implement the interface with one or more classes

```
public class Chequing implements IAccount { … }
public class Saving implements IAccount { … }
public class Investment implements IAccount { … }
```


Step 3: Create the facade class and wrap the classes that implements the interface

```
public class BankService {
    private Hashtable<int, IAccount> bankAccounts;
    public BankService() {
        this.bankAccounts = new Hashtable<int, IAccount>;
    }
    public int createNewAccount(String type, BigDecimal initAmount) {
        IAccount newAccount = null;
        switch (type) {
            case "chequing":
                newAccount = new Chequing(initAmount);
                break;
            case "saving":
                newAccount = new Saving(initAmount);
                break;
            case "investment":
                newAccount = new Investment(initAmount);
                break;
            default:
                System.out.println("Invalid account type");
                break;
        }
        if (newAccount != null) {
            this.bankAccounts.put(newAccount.getAccountNumber(), newAccount);
            return newAccount.getAccountNumber();
        }
        return -1;
    }
    public void transferMoney(int to, int from, BigDecimal amount) {
        IAccount toAccount = this.bankAccounts.get(to);
        IAccount fromAccount = this.bankAccounts.get(from);
        fromAccount.transfer(toAccount, amount);
```



```
public class Customer {

    public static void main(String args[]) {
        BankService myBankService = new BankService();

        int mySaving = myBankService.createNewAccount("saving",
            new BigDecimal(500.00));

        int myInvestment = myBankService.createNewAccount(
            "investment", new BigDecimal(1000.00));

        myBankService.transferMoney(mySaving, myInvestment, new
            BigDecimal(300.00));
    }

}
```


What are the key design principles are used to implement the facade design pattern?

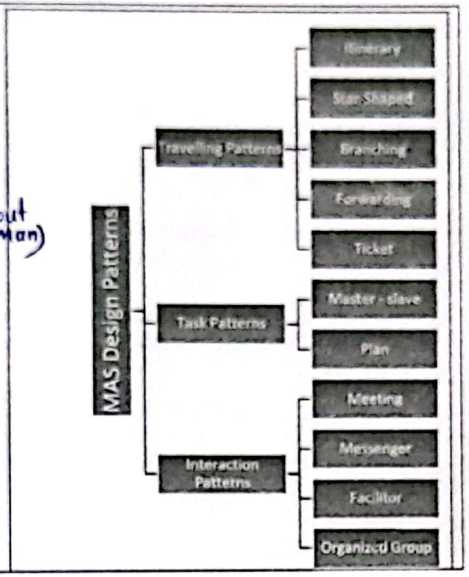
1. Information hiding, encapsulation, generalization, coupling
2. Encapsulation, information hiding, separation of concerns
3. Inheritance, information hiding, separation of concerns, generalization
4. Encapsulation, implementing interfaces, information hiding, inheritance

Answer

While the facade design pattern uses a number of different design principles, it purpose is to provide ease of access to a complex subsystem. This is done by encapsulating the subsystem classes into a Facade class, and then hiding them from the client classes so that the clients do not know about the details of the subsystem. Therefore, this is the correct answer.

Extension of Facade Design Patterns-Multi Agents

- Multi Agent System (MAS) — computational system where multiple autonomous agents (can operate without a human) interact with each other to achieve individual and collective goals
- MAS can be simple or complex and the agents exhibit some degree of intelligence.



Multi Agent Design Patterns

Blackboard Pattern	Agents contribute their knowledge and expertise to a shared blackboard-like data structure. Agents monitor the blackboard for relevant information and make decisions based on the current state of the blackboard.
Master-Slave Pattern	This pattern involves a master agent that assigns tasks to multiple slave agents. The master agent distributes the workload among the slaves and coordinates their activities, ensuring efficient execution of the tasks.
Contract Net Pattern	This pattern is useful when there is a need for task allocation among multiple agents. A task is advertised, and interested agents submit bids. The task is then assigned to the agent with the most suitable bid
Mediator Pattern	The mediator pattern provides a centralized component, the mediator, that handles communication and coordination among multiple agents. Agents communicate with the mediator instead of directly with each other, simplifying the overall system architecture.

22

Multi Agent Design Patterns

Team Pattern	This pattern involves organizing agents into teams or groups to accomplish complex tasks. Each team may have a team leader or coordinator responsible for managing the interactions and cooperation among team members.
State Machine Pattern	The state machine pattern represents the behavior of an agent or a system as a set of states and transitions. Agents can switch between different states based on certain conditions or events, enabling dynamic adaptation and decision-making.
Auction Pattern	The auction pattern is suitable for scenarios where multiple agents compete to acquire a limited resource. Agents submit bids, and the resource is allocated to the agent with the highest bid, following predefined auction rules.
Observer Pattern	The observer pattern facilitates the communication between agents by establishing a publish-subscribe mechanism. Agents can subscribe to events or changes in other agents' states and receive notifications when relevant updates occur.

23

Facade Design Pattern

- Use to hide the complexity of a subsystem by encapsulating it behind a unified wrapper class called a facade class.
- Removes the need for client classes to manage a subsystem on their own
- Handles instantiation and redirection of tasks to appropriate class within the subsystem.
- Provides client classes with a simplified interface for the subsystem.
- Acts simply as a point of entry to a substem and does not add more functional to the subsystem.

24

Q&A

1. What are the key advantages of using the Facade pattern?
2. Can you access each of the subsystems directly?
3. There should be only one facade for a complex subsystem. Is this understanding correct?
4. What are the challenges associated with the Facade pattern?

25

Answers

(01)
- Simplified interface
- encapsulate the complexity
- easy maintenance
- Improve readability and understandability.
- Easy adoption for clients
- Consistent interfaces.

(02) No, accessed through facade class.

(03) Single facade for a complex subsystem is common, using multiple facades can be appropriate when clients require speacial interfaces or when different subsystem functionalities evolve independently. Decision is based on requirements of the client & managing subsystem complexity.

(04)
- Limited flexibility : may not accomadate all possible client needs.
- Maintaining consistency : changes in subsyst may impact the facade's effectiveness.
- Need to maintain a proper documentation to guide the clients.
- Including functionalities that is never used by the clients in the facade may increase the complexity.

extra :- Interface class in OOP :-

In OOP interface is a programming construct that defines a contract for set of methods that a class must implement. A interface class, often simply reffered to as an "interface" does not provide any implementation for the methods it declares. Instead, it serves as a bluePrint for classes that implement it. (No implementation)

- keyword 'implement' is used to indicate that they implement a particular interface.

Facade Design Pattern

* Facade design pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.
* This pattern involves a single class which provides simplified methods required by the client and delegate calls to methods of existing system classes.

Instances used;

In software development senarios where a simplified interface is needed to interact with a complex system or subsystem.

(01) Payment Processing System;

PaymentFacade that encapsulates the interactions with
- different payment gateways
- handles encryption
- manage transaction logging

clients only interact with PaymentFacade.

(02) Travel booking system.

BookingFacade for booking process for the clients & other multiple external services.

(03) Ecommerce inventory system:

InventoryFacade — manage products
update stock
process orders.

Steps of implementation:

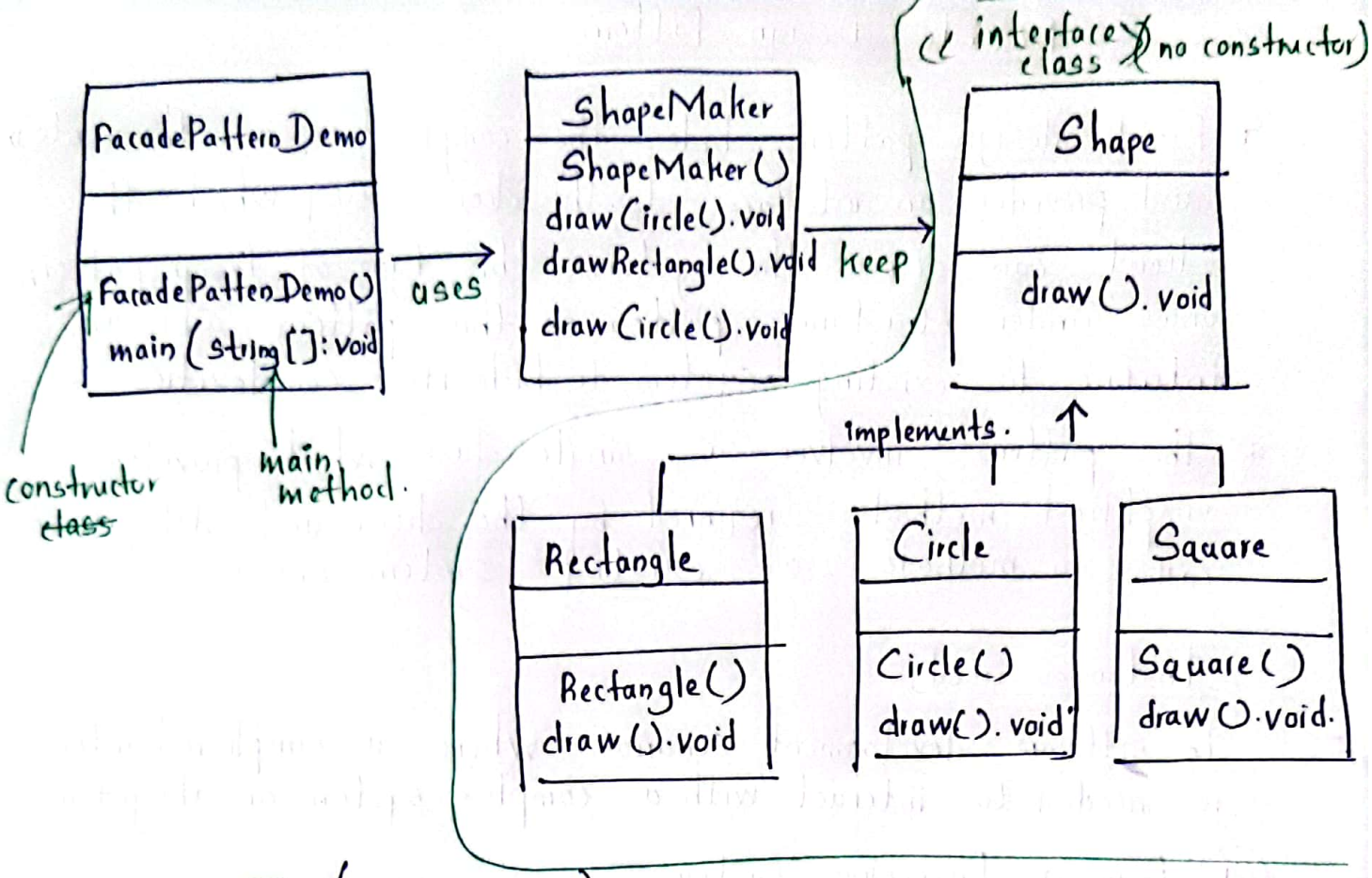
[01] Create an interface (Shape.java)

[02] Create concrete classes implementing the same interface. (Rectangle.java, Square.java, Circle.java)

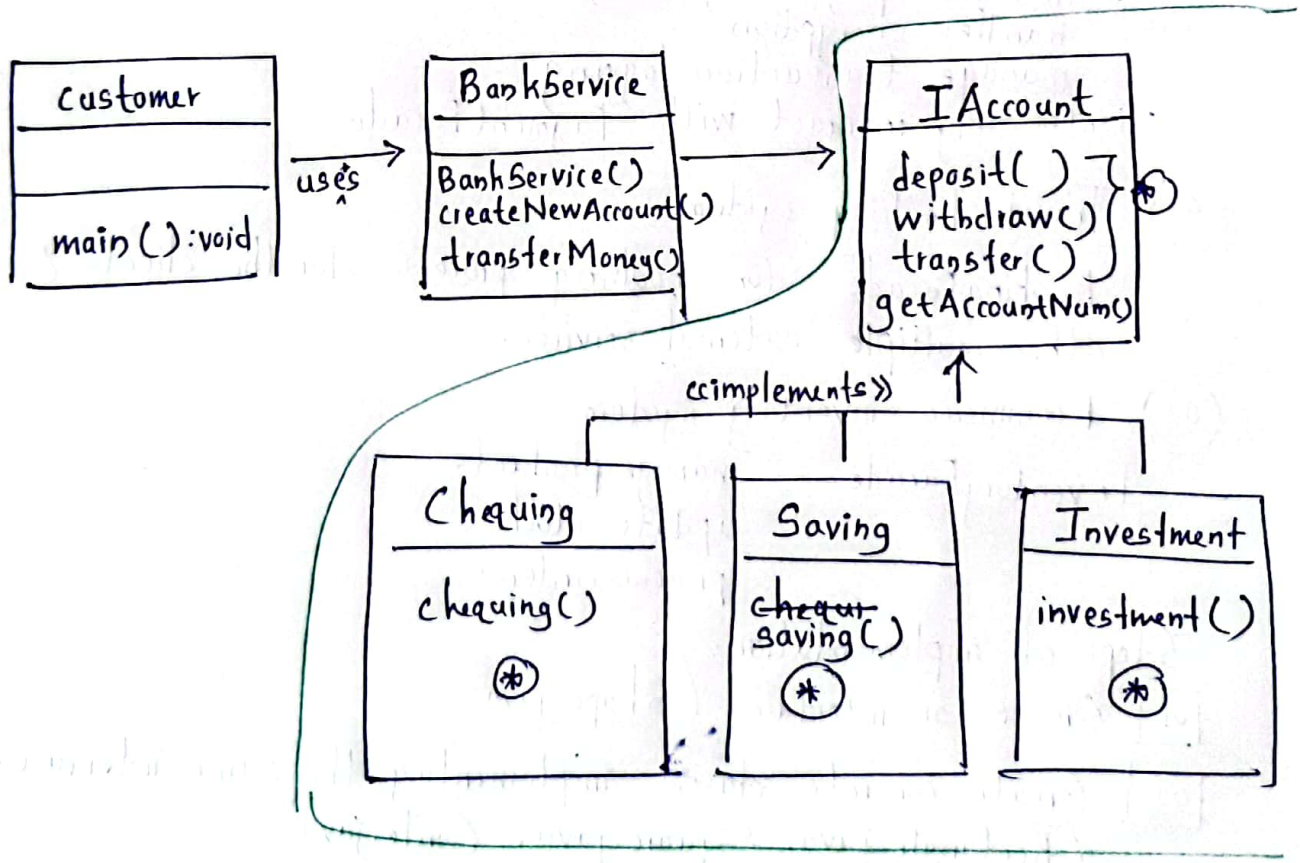
[03] Create the facade class. (ShapeMaker.java)

[04] Use the facade to (draw various shapes): (FacadePatternDemo.Java)

[05] Verify the output.



example (02) (slide 13–18)



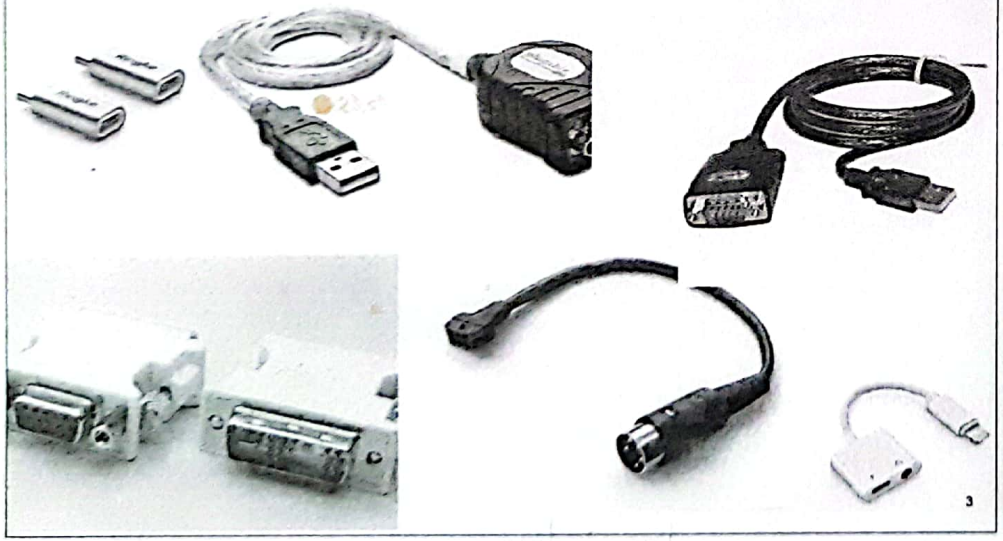
Adapter Design Pattern

1

Design Patterns

Creational (5)	Structural (7)	Behavioral (11)
• Factory	• Adapter	• Chain Of Responsibility
• Abstract Factory	• Bridge	• Command
• Singleton	• Composite	• Interpreter
• Prototype	• Decorator	• Iterator
• Builder	• Facade	• Mediator
	• Flyweight	• Memento
	• Proxy	• Observer
		• State
		• Strategy
		• Template
		• Visitor

2



3

- Adapter Design pattern is used to communicate between two non-compatible existing systems by providing a compatible interface.
- Common term is wrapper class.

* Structural design pattern.

4

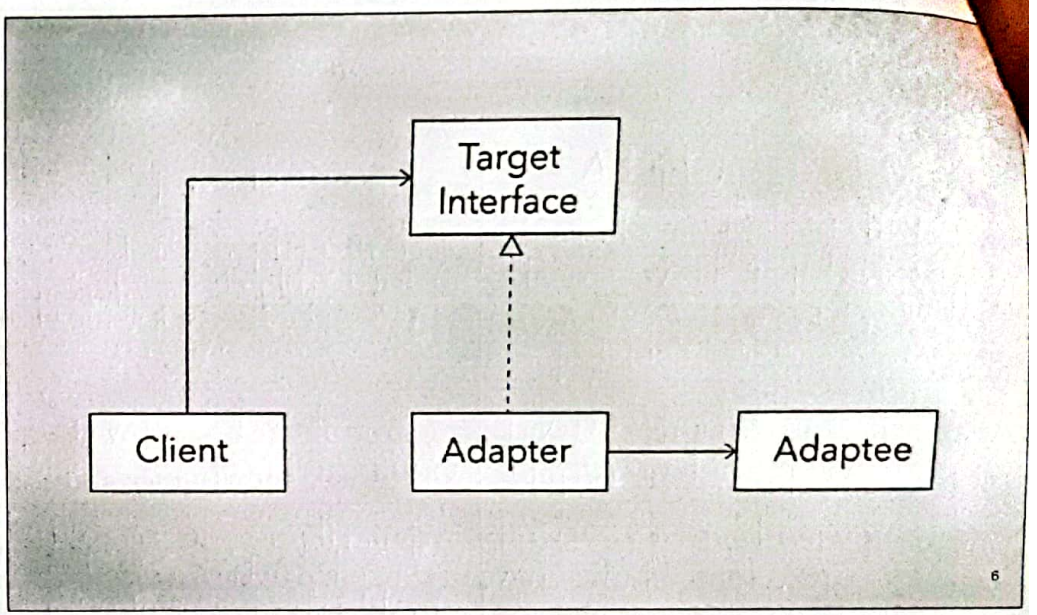
Question

- If the interfaces are incompatible, why we can't change them.

Changing interfaces may not be feasible due to reasons as;

- legacy code → code can be a part of legacy system, modifing can cause problems.
- Third party libraries → if working with third party libraries ~~It will~~ you might not have the ability to modify interfaces.
- Will be hard in terms of code maintainability.
- Will reduce the stability and compatability.
- It is not practical.

5



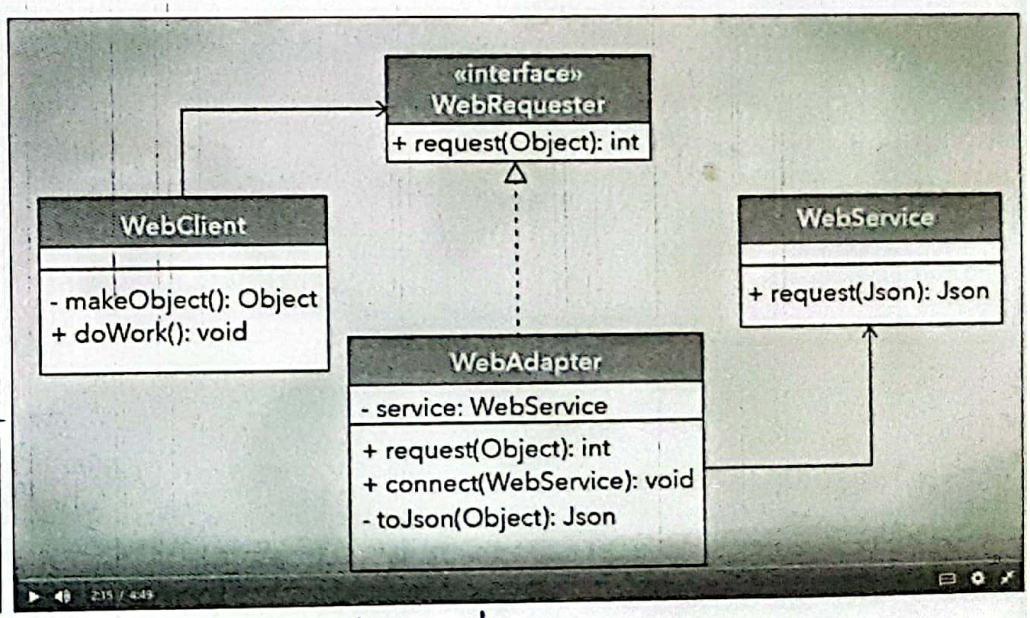
6

Implementations

- Third party libraries → (make third party components compatible with the existing system).
- System Integration
- Web Services
- System Upgrades

- Legacy system integration (bridge old & new interfaces)
- Database connectivity (switching between different DBMS)

7



- class diagram -

Step 1: Design the target interface

```
public interface WebRequester {
    public int request(Object);
}
```

Step 2: Implement the target interface with the adapter class

```
public class WebAdapter implements WebRequester {
    private WebService service;

    public void connect(WebService currentService) {
        this.service = currentService;
    }

    public int request(Object request) {
        Json result = this.toJson(request);
        Json response = service.request(result);
        if (response != null)
            return 200; // OK status code
        return 500; // Server error status code
    }

    private Json toJson(Object input) { … }
}
```

Step 3: Send the request from the client to the adapter using the target interface

```
public class WebClient {
    private WebRequester webRequester;

    public WebClient(WebRequester webRequester) {
        this.webRequester = webRequester;
    }

    private Object makeObject() { … } // Make an Object

    public void doWork() {
        Object object = makeObject();
        int status = webRequester.request(object);

        if (status == 200) {
            System.out.println("OK");
        } else {
            System.out.println("Not OK");
        }
        return;
    }
}
```

```
public class Program {
    public static void main(String args[]) {
        String webHost = "Host: https://google.com\n\r";
        WebService service = new WebService(webHost);
        WebAdapter adapter = new WebAdapter();
        adapter.connect(service);
        WebClient client = new WebClient(adapter);
        client.doWork();
    }
}
```

What are the characteristics of the adapter design pattern?

1. The client and adaptee classes have incompatible interfaces.
2. An adapter is a wrapper class that wraps the adaptee, hiding it from the client.
3. The client sends requests indirectly to the adaptee by using the adapter's target interface.
4. The adapter translates the request sent by the client class into a request that the adaptee class is expecting.

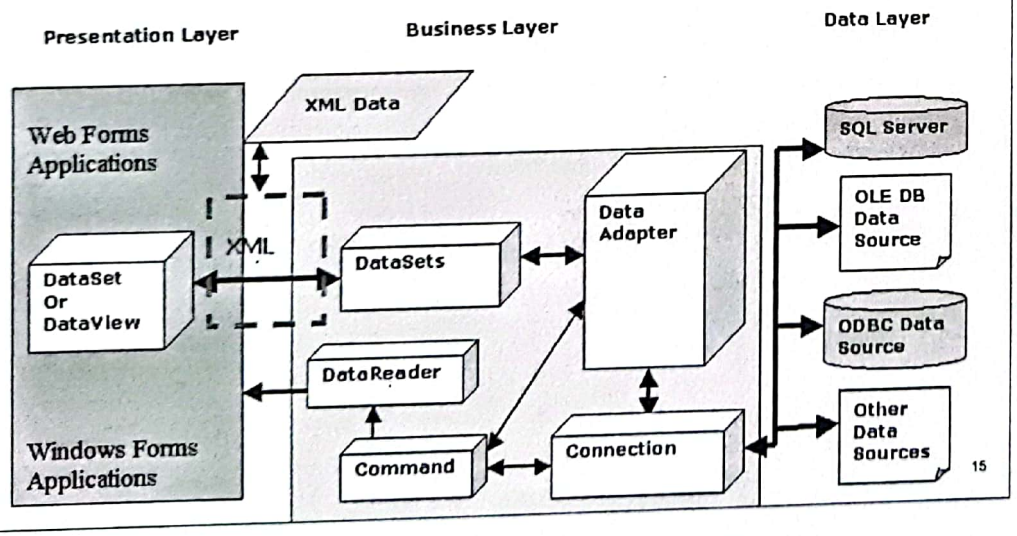
13

How is Facade different from the Adapter design pattern?

Facade	Adapter
Used to simplify and provide a simplified interface to set of interfaces or subsystems.	Used to make existing classes work together (to make incompatible interfaces compatible)
Involves facade and the subsystem classes	Involve traget interface, adapter and adaptee.
focuses on simplifying a set of interfaces by providing a higher-level interface to a subsystem.	focuses on making one class work with another by providing a compatible interface.

14

Database Connector



15

How is Facade different from the Adapter design pattern?

In the Adapter pattern, you are trying to alter an interface so that your clients do not see any difference between the interfaces. By contrast, the Facade pattern simplifies the interface. It presents the client with a simple interface to interact with (instead of a complex subsystem).

16

Adapter design needs more code. Do you agree?

You may need to write some additional code. However, the payoff is great, particularly for those legacy systems that cannot be changed but you still want to use them for their stability

Adapter Design Pattern

* Adaptor pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces. This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. (Adaptor)

Key components of the adaptor design pattern.

- Target Interface — This is the interface that the client code expects to interact with. Desired interface the client uses.
- Adaptee — Existing class or interface that needs to be integrated into the client code. It has an interface that is incompatible with the target interface.
- Adapter — The class that bridges the gap between the adaptee and the target. It implements the target interface and contains an instance of the adaptee.

Decorator Design Pattern

*Decorator design pattern is a structural pattern in softwar design that allows behavior to be added to an individual object statically or dynamically, without affecting the behavic of other objects from the same class.

Key Components:

(01) Component interface / Abstract component →
Defines the interface for objects that can have responsibilites added dynamically.
It usually declares the methods that concrete component & decorators have in common.

(02) Concrete Component
Implements Concrete interface.
Represent the base object to which additional functionalities are added.

(03) Decorator.
Also implement concrete interface.
Contains a reference to a Component object & defines an interface that conforms to the Component's interface.
Can have additional responsibilities (methods & behaviors) that can be added dynamically to the component.

(04) Concrete Decorator.
Extends the Decorator class
Adds specific ~~te~~ methods & behaviors to the component.

* The decorator pattern is primarily used to extend the functionalities of objects dynamically at runtime. It involves attaching additional responsibilities to an object by wrapping it in a class that provides the desired functionality.
Here the modification to the behavior of an object can occur at runtime, making it more dynamic.

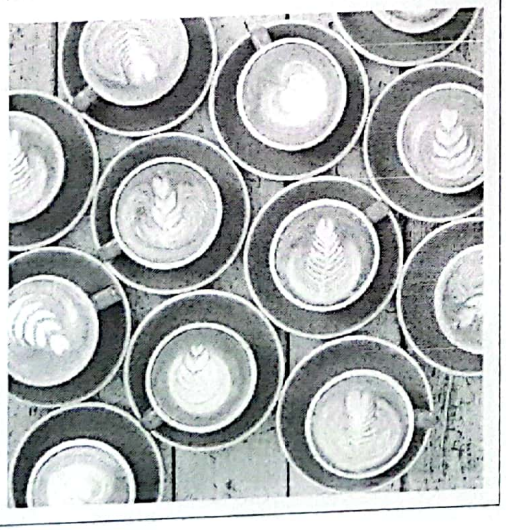
Decorator Design Pattern

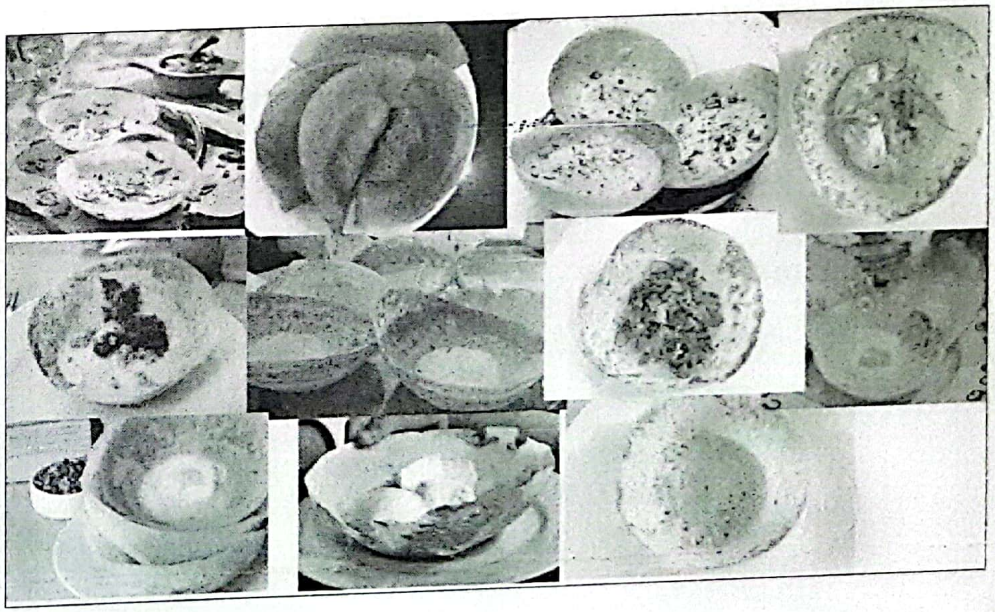
Design Patterns

Creational(5)	Structural(7)	Behavioral(11)
• Factory	• Adapter	• Chain Of Responsibility
• Abstract Factory	• Bridge	• Command
• Singleton	• Composite	• Interpreter
• Prototype	• Decorator	• Iterator
• Builder	• Facade	• Mediator
	• Flyweight	• Memento
	• Proxy	• Observer
		• State
		• Strategy
		• Template
		• Visitor

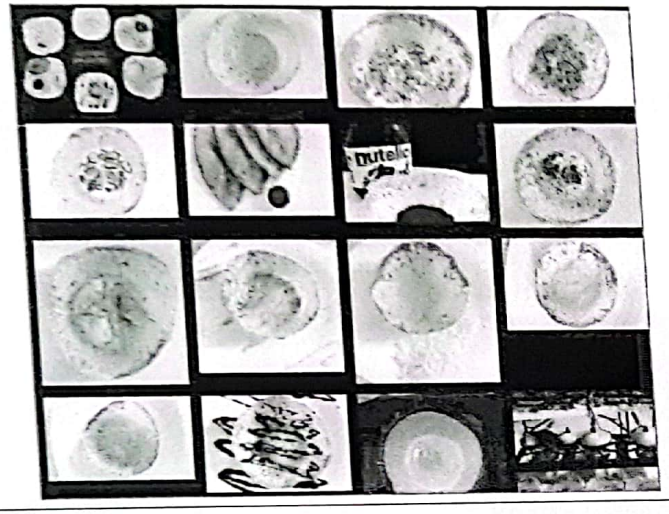
How would you model different types of coffee?

- One class many properties?
- Many classes?





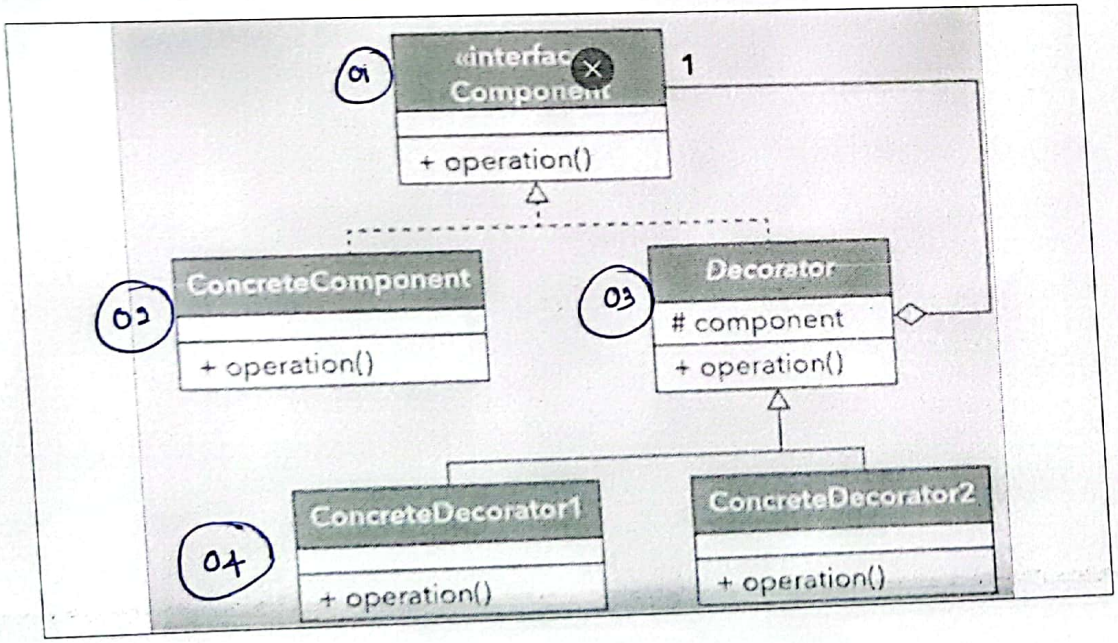
https://hoppersunlimited.com.au/varieties/

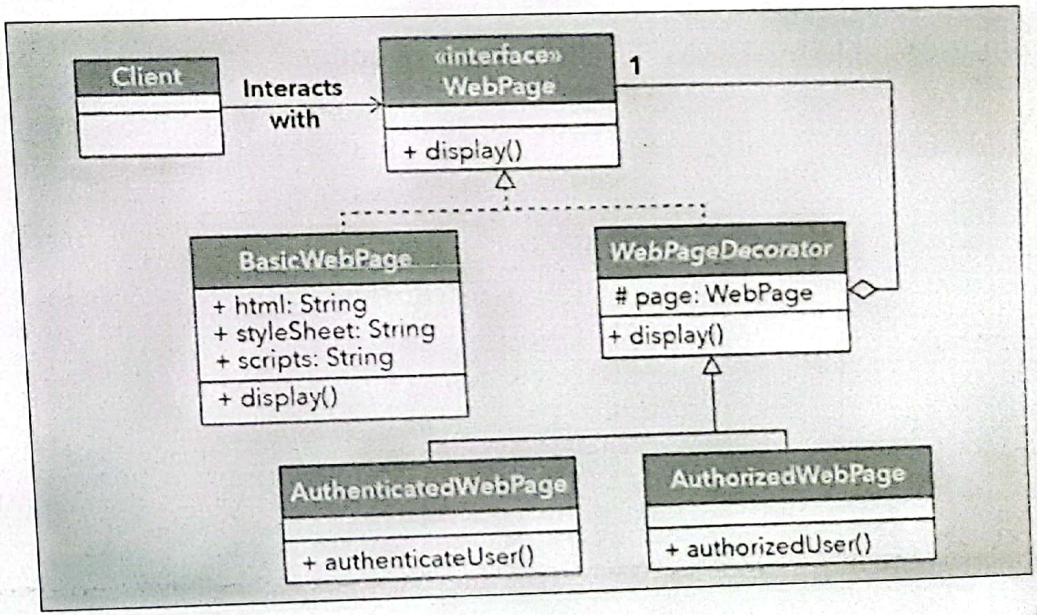


Introduction

- Allows behaviour / property to be added to an individual object without impacting the behaviour of the class.
- This can be added statistically or dynamically.
- Alternative for sub classing and avoid many number of classes or many number of attributes

* Decorator design pattern allows a user to add new functionality to an existing object without changing its structure.
* Structural design patten
  Also called wrapper.





Step 1: Design the component interface

```
public interface WebPage {
    public void display();
}
```

Step 2: Implement the interface with your base concrete component class

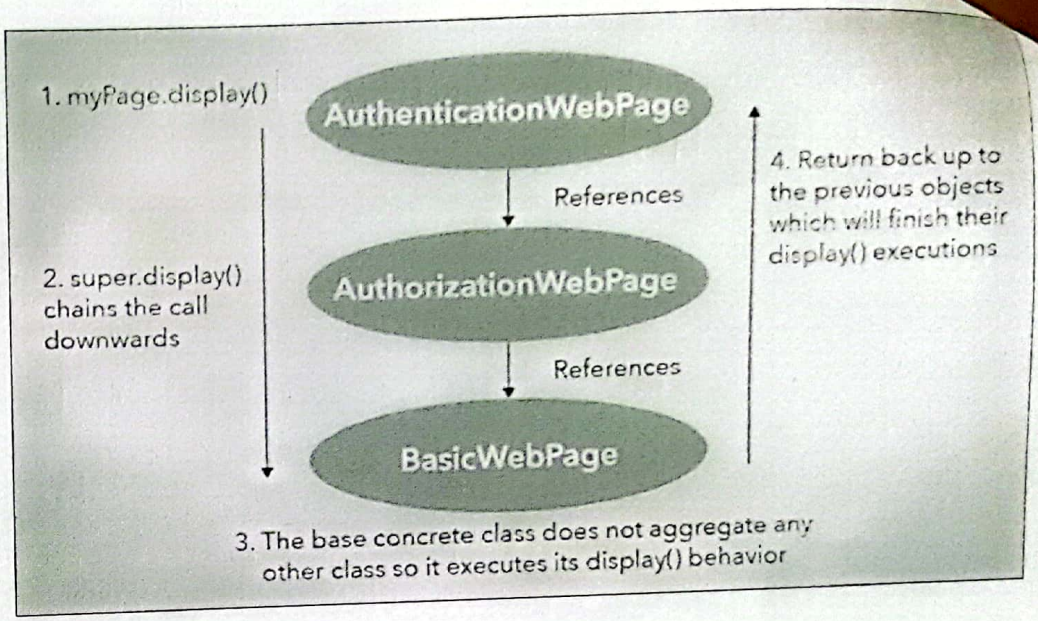
```
public class BasicWebPage implements WebPage {
    private String html = ...;
    private String styleSheet = ...;
    private String scripts = ...;

    public void display() {
        /* Renders the HTML to the stylesheet, and run any
        embedded scripts */
        System.out.println("Basic web page");
    }
}
```

Step 3: Implement the interface with your abstract decorator class

```
public abstract class WebPageDecorator implements WebPage {
    protected WebPage page;

    public WebPageDecorator(WebPage webpage) {
        this.page = webpage;
    }

    public void display() {
        this.page.display();
    }
}
```

```
public class AuthorizedWebPage extends WebPageDecorator {
    public AuthorizedWebPage(WebPage decoratedPage) {
        super(decoratedPage);
    }
    public void authorizedUser() {
        System.out.println("Authorizing user");
    }
    public display() {
        super.display();
        this.authorizedUser();
    }
}

public class AuthenticatedWebPage extends WebPageDecorator {
    public AuthenticatedWebPage(WebPage decoratedPage) {
        super(decoratedPage);
    }
    public void authenticateUser() {
        System.out.println("Authenticating user");
    }
    public display() {
        super.display();
        this.authenticateUser();
    }
}
```

```
public class Program {
    public static void main(String args[]) {
        WebPage myPage = new BasicWebPage();
        myPage = new AuthorizedWebPage(myPage);
        myPage = new AuthenticatedWebPage(myPage);
        myPage.display();
    }
}
```



What are the reasons for using the decorator design pattern?

1. The decorator design pattern allows objects to dynamically add behaviors to others.
2. To hide a complex, and resource demanding object until it is needed.
3. To reduce the number of classes needed to offer a combination of behaviors.
4. It lets you build a tree-like structure of objects that can be treated as a single, uniform object type.

What are the reasons for using the decorator design pattern?

1. The decorator design pattern allows objects to dynamically add behaviors to others.
2. **To hide a complex, and resource demanding object until it is needed.**
3. To reduce the number of classes needed to offer a combination of behaviors.
4. It lets you build a tree-like structure of objects that can be treated as a single, uniform object type.

1. You can build functionality by stacking objects with the Decorator design pattern.
2. The Design Pattern you use to hide a complex and resource demanding object is Proxy!
3. you may find that by using a Decorator, you have more objects to get the same behaviour. That's okay, because the pattern brings you other benefits!
4. Actually this describes a Composite pattern. Both use aggregation, but Decorator is more like a stack of objects, where Composite is a tree.

What is the difference between proxy and decorator design pattern?

A protection proxy might be implemented like a decorator, but you should not forget the intent of a proxy.

Decorators focus on adding responsibilities, but proxies focus on controlling the access to an object.

Proxies differ from each other through their types and implementations.

If you can remember their purposes, in most cases you will be able to clearly distinguish proxies from decorators.

What are the key advantages of using a decorator

- The existing structure is untouched, so you cannot introduce bugs there.
- New functionalities can be added to an existing object easily.
- You do not need to predict/implement all the supported functionalities at once (in other words, in the initial design phase). You can develop incrementally. For example, you can add decorator objects one by one to support your needs. You must acknowledge that if you make a complex class first and then want to extend the functionalities, that will be a tedious process.

What are the disadvantages associated with this pattern?

- Separate core and decorator features.
- If you create too many decorators in the system, it will be hard to maintain and debug. So, in that case, they can create unnecessary confusion.

References

If the design involves need ~~of~~ for a combination of features that can change at runtime, the decorator pattern can be a valuble tool in acheiving these objectives.

Instances

(01) GUI frameworks → decorators can be used to add new behaviors to graphical components as to add borders, scroll bars or tool tips.

(02) Logging — features that can be added timestamping, log level filtering etc..

(03) Security — encryption / authentication

(04) Web development - add authetication logging or cash

(05) Text formatting — add styling, fonds, colour to plain text.

Template Method

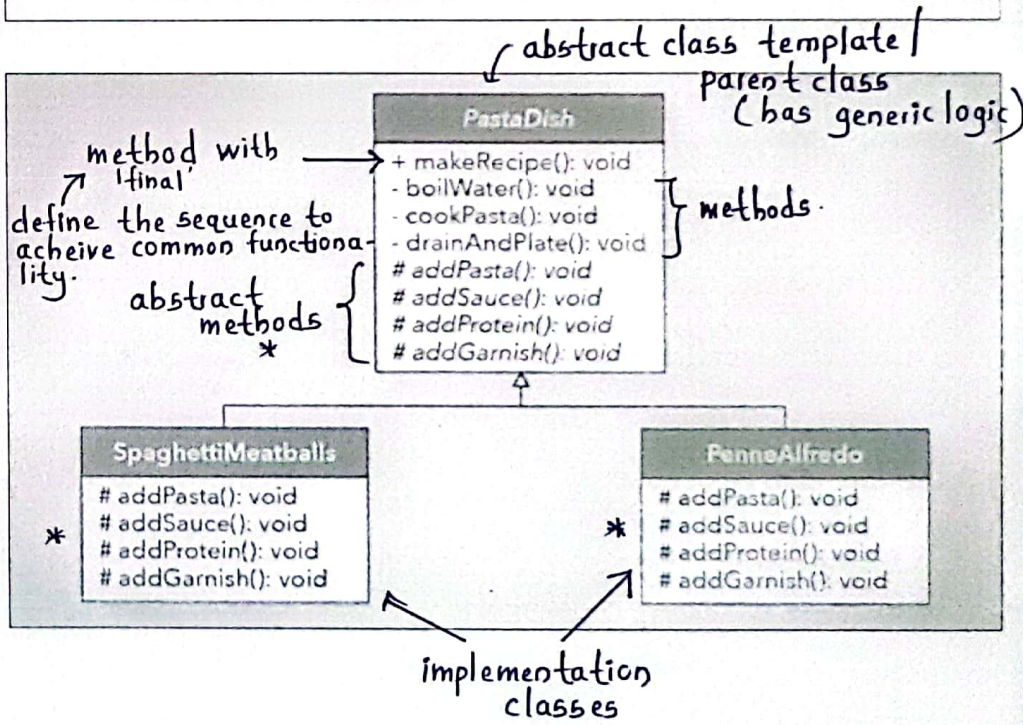
A Behavioural Design Pattern

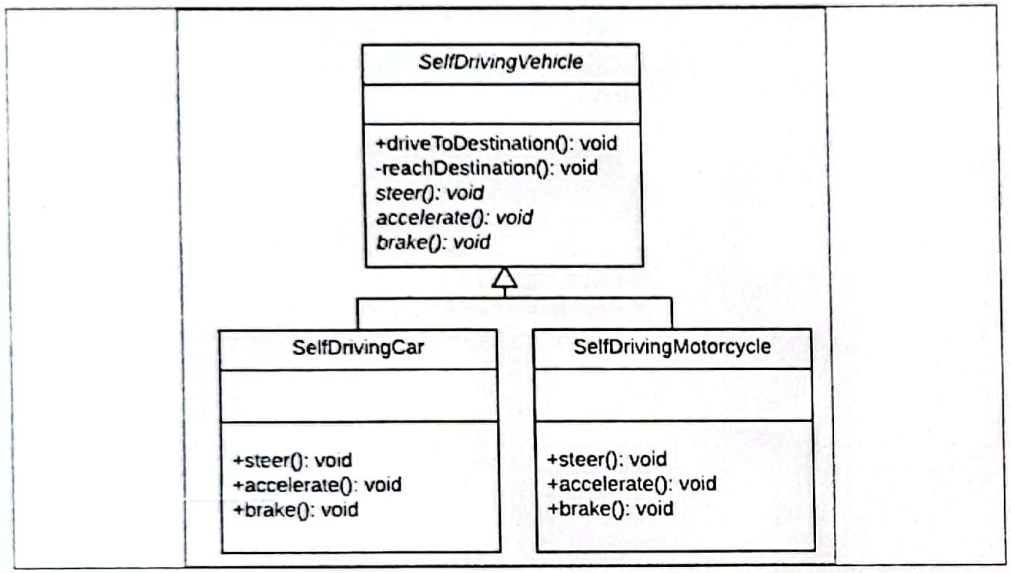
Design Patterns

Creational (5)	Structural (7)	Behavioral (11)
• Factory	• Adapter	• Chain Of Responsibility
• Abstract Factory	• Bridge	• Command
• Singleton	• Composite	• Interpreter
• Prototype	• Decorator	• Iterator
• Builder	• Facade	• Mediator
	• Flyweight	• Memento
	• Proxy	• Observer
		• State
		• Strategy
		• Template
		• Visitor

Template Method Pattern

- Behavioral patterns also focus on how independent objects work towards a common goal.
- The template method defines an algorithm's steps deferring the implementation of some steps to subclasses.
- It is a behavioral design pattern and is concerned with the assignment of responsibilities.
- Template Method Pattern defines the skeleton of an algorithm in the superclass but allows subclasses to override specific steps of the algorithm without changing the structure.






```
public abstract class PastaDish {
    public final void makeRecipe() {
        boilWater();
        addPasta();
        cookPasta();
        drainAndPlate();
        addSauce();
        addProtein();
        addGarnish();
    }
    protected abstract void addPasta();
    protected abstract void addSauce();
    protected abstract void addProtein();
    protected abstract void addGarnish();

    private void boilWater() {
        System.out.println("Boiling water");
    }
}
```

template method.

sequence to be followed in order.

Final - Cannot be overwritten at the sub classes.

```
public class SpaghettiMeatballs
extends PastaDish {
    protected void addPasta() {
        System.out.println("Add
        spaghetti");
    }
    protected void addProtein() {
        System.out.println("Add
        meatballs");
    }
    protected void addSauce() {
        System.out.println("Add
        tomato sauce");
    }
    protected void addGarnish() {
        System.out.println("Add
        Parmesan cheese");
    }
}
```

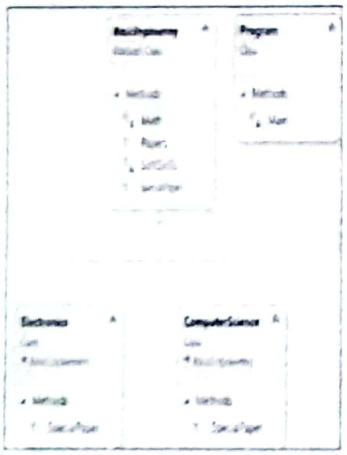
```
public class PenneAlfredo extends
PastaDish {
    protected void addPasta() {
        System.out.println("Add
        penne");
    }
    protected void addProtein() {
        System.out.println("Add
        chicken");
    }
    protected void addSauce() {
        System.out.println("Add
        Alfredo sauce");
    }
    protected void addGarnish() {
        System.out.println("Add
        parsley");
    }
}
```

Usage

- Multiple classes with same functional operations with different implementations and same order of executions.
- Practical application of generalization and inheritance
- Popular in framework development
- Avoid code duplication

In this pattern, subclasses can simply redefine the methods based on their needs. Is this understanding correct?

Answer: Yes



In the abstract class BasicEngineering, only one method is abstract, and the other two methods are concrete methods. What is the reason behind this?

Answer: This is a simple example with only three methods, and you want the subclasses to override only the SpecialPaper() method here. Other methods are common to both courses, and they do not need to be overridden by the subclasses

What are the key advantages of using a Template Method design pattern?

Answer:

You can control the flow of the algorithms. Clients cannot change them.

Common operations will be in a centralized location. For example, in an abstract class, the subclasses can redefine only the varying parts so that you can avoid redundant code.

What are the key challenges associated with a Template Method design pattern?

Answer:

- The client code cannot direct the sequence of steps. If you want that type of functionality, use the Builder pattern.
- A subclass can override a method defined in the parent class (in other words, hiding the original definition in the parent class)
- Having more subclasses means more scattered code, which is difficult to maintain

Abstract and interface classes.

Similarities;

- Both abstraction and interface class support abstraction.
- " " " " contributes to polymorphism.
- " " " " involve inheritance.

Differences.

Abstract	Interface.
Have both abstract and concrete members.	only declare method signatures without providing any implementation.
May contain feilds, constructors and properties in addition to methods.	Cannot have constructors or properties with implementation.
Have access modifiers.	All members are implicitly public & abstract. access modifiers are not allowed.
Single inheritance	Support multiple inheritance.

Templates are needed when you need data in a single format.
We use templates in OOP in the interfaces.
In Java if we have the 'final' keyword we cannot override the method.

Template Method design pattern.

* Template Method pattern defines a sequence of steps of an algorithm. The subclasses are allowed to override the steps but not allowed to change the sequence.
* Here the general logic is kept in the abstract parent class and child class define specifics.

- In template design pattern it can have multiple classes with same functional operations with different implementations. (abstract class) and same order of execution, from the 'final' keyword that execution cannot be overriden.

Possible implementation.

1) Salary calculation.

Chain of Responsibility (CoR) Method

A Behavioural Design Pattern

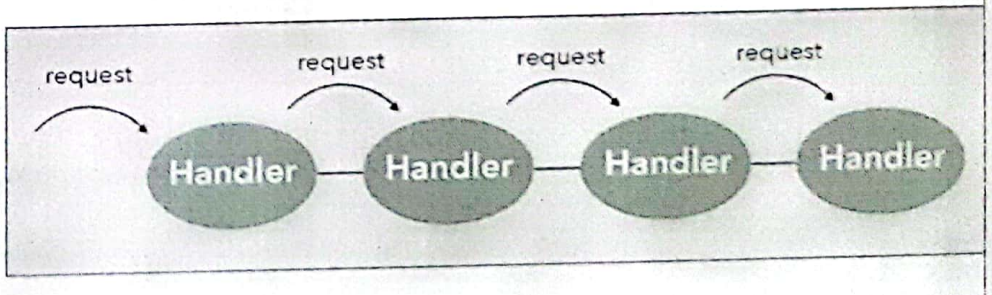
Design Patterns

Creational (5)	Structural (7)	Behavioral (11)
• Factory	• Adapter	• Chain Of Responsibility
• Abstract Factory	• Bridge	• Command
• Singleton	• Composite	• Interpreter
• Prototype	• Decorator	• Iterator
• Builder	• Facade	• Mediator
	• Flyweight	• Memento
	• Proxy	• Observer
		• State
		• Strategy
		• Template
		• Visitor

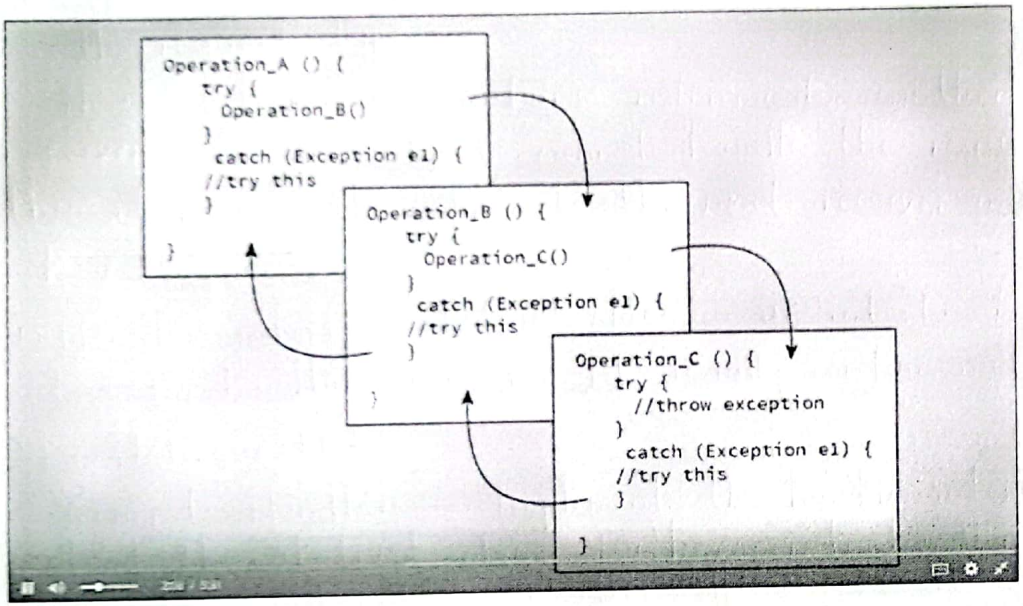
Chain of Responsibility Pattern

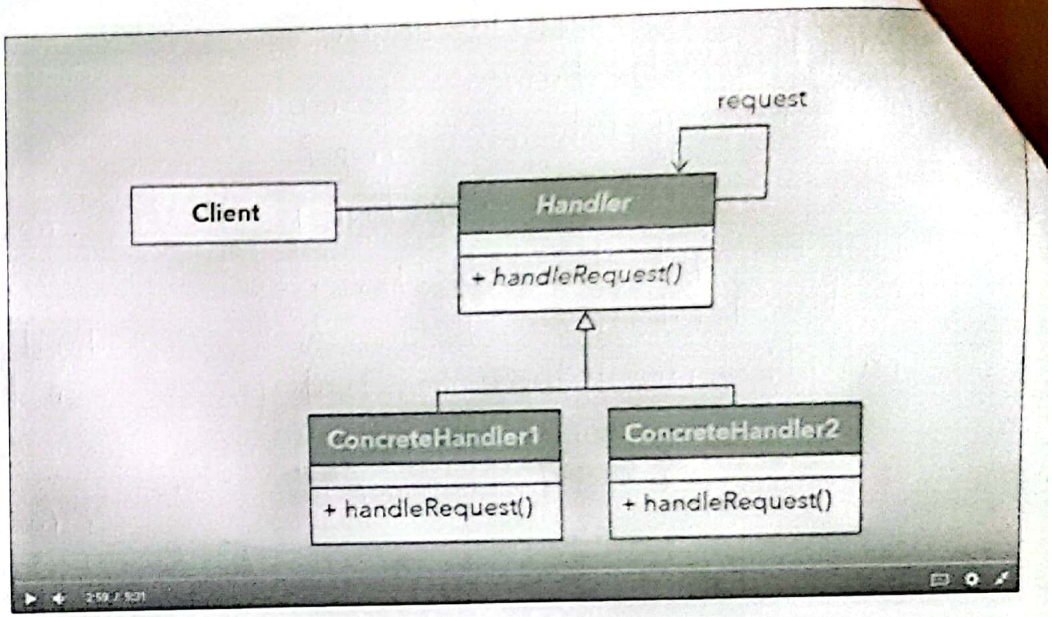
- A Behavioral pattern
- Avoid coupling the sender of a request to its receiver by giving more than one object to handle request.
- Chain the receiving objects and pass the request along the chain until an object handles it.
  - Ex: Consulting a doctor.
- The processing chain may end in either of these two scenarios:
  - A handler can process the request completely.
  - You have reached the end of the chain.

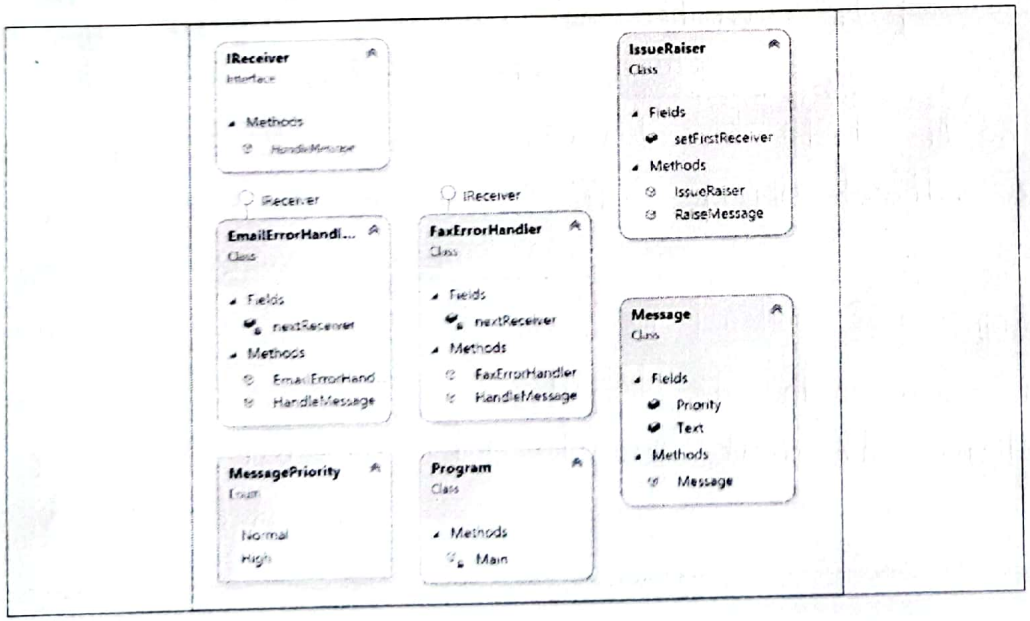
Chain of Responsibility



class diagram.







Applications

- Email Spam detectors
- Medical Testing
- Troubleshooting
- Classifier Systems

. request processing pipelines
. access control systems

Setting up the Priorities

- Performance
- Accuracy
- Less complex

What are the advantages of using the Chain of Responsibility design pattern?

- You have more than one object to handle a request. (Notice that if a handler cannot handle the whole request, it may forward the responsibility to the next handler in the chain.)
- The nodes of the chain can be added or removed dynamically. Also, you can shuffle their order. For example, in the previous application, if you see that most issues are e-mail issues, then you may place EmailErrorHandler as the first handler to save the average processing time of the application.
- A handler does not need to know how the next handler in the chain will handle the request. It can focus on its own handling mechanism only.
- In this pattern, you are decoupling the senders (of requests) from the receivers.
- Can improve High Availability (HA) & Disaster Recovery (DR)

What are the challenges associated with using the Chain of Responsibility design pattern?

- There is no guarantee that the request will be handled because you may reach at the end of chain but have not found any explicit receiver to handle the request.
- Debugging becomes tricky with this kind of design.
- Unnecessary handlers may lead to inefficiency.
  - Priority can be set.
  - Bypassing the handlers depending on the information received.

* Chain of responsibility pattern creates a chain of receiver objects or a request. This pattern decouples sender and receiver of a request based on the request type.

Here each receiver contains reference to another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

Key components :-

1) Handler interface / abstract class — an interface / abstract class for handling requests. It usually contain methods as 'handleRequest()'

2) Concrete Handlers — Implement the handler interface or extend the abstract class. Each concrete handler is responsible for handling specific type of requests. If it can handle a request, it does so; otherwise, it passes the request to the next handler in the chain.

3) Client — Initiates the request and starts it on the chain. The client is unaware of the specific handlers in the chain; it just knows the first one.

4) Chain — Manages the chain of handlers. It typically has a method to add a handler to the chain and a method to process requests by starting from the first handler in the chain.

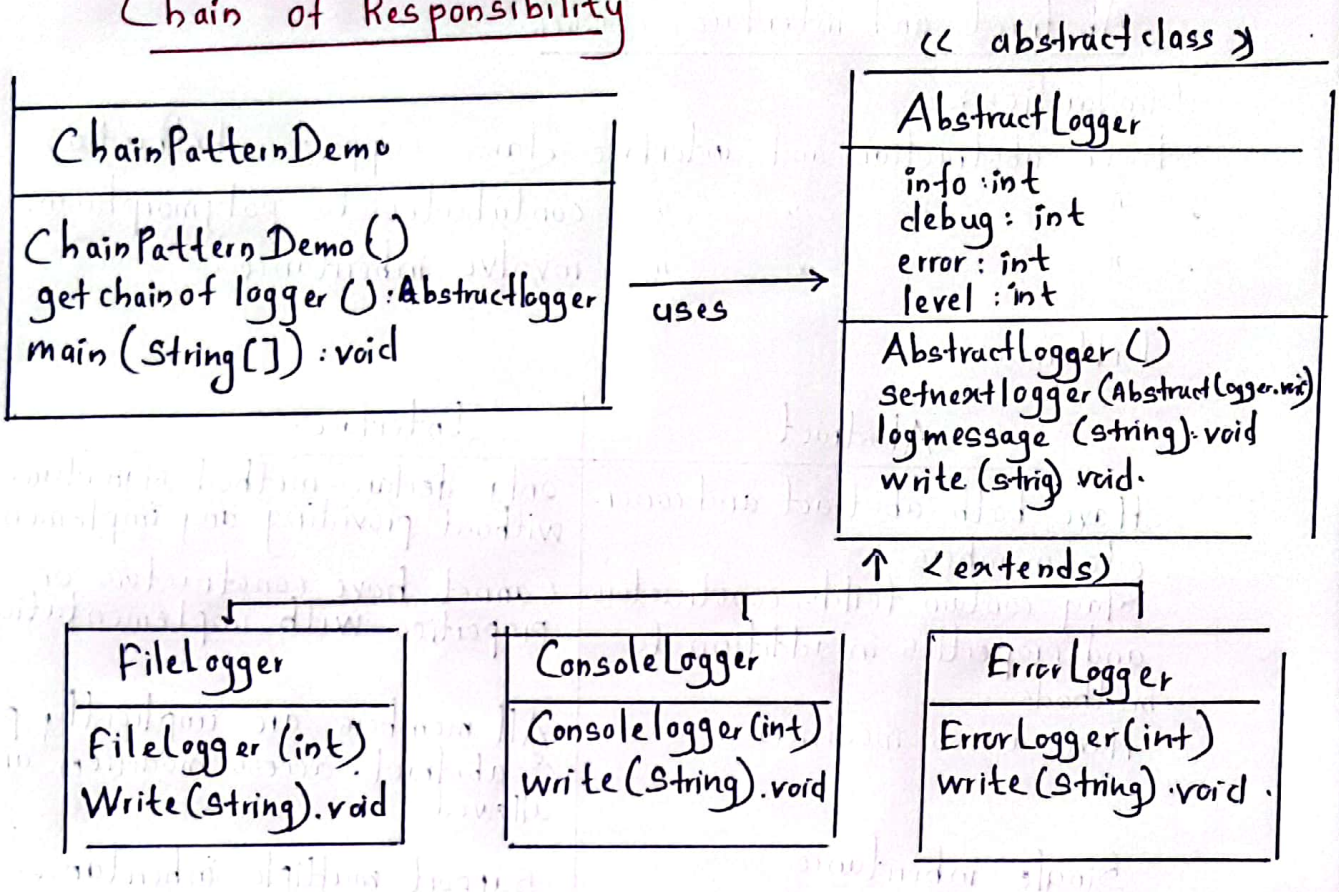
steps in implementation.

01) Create the abstract class / logger (AbstractLogger)

02) Create concrete class extending logger. (consoleLogger, FileLogger, ErrorLogger)

03) Create different types of loggers. Assign them error levels and set next logger in each logger. This represent part of the chain. (ChainPatterDemo.)

Chain of Responsibility



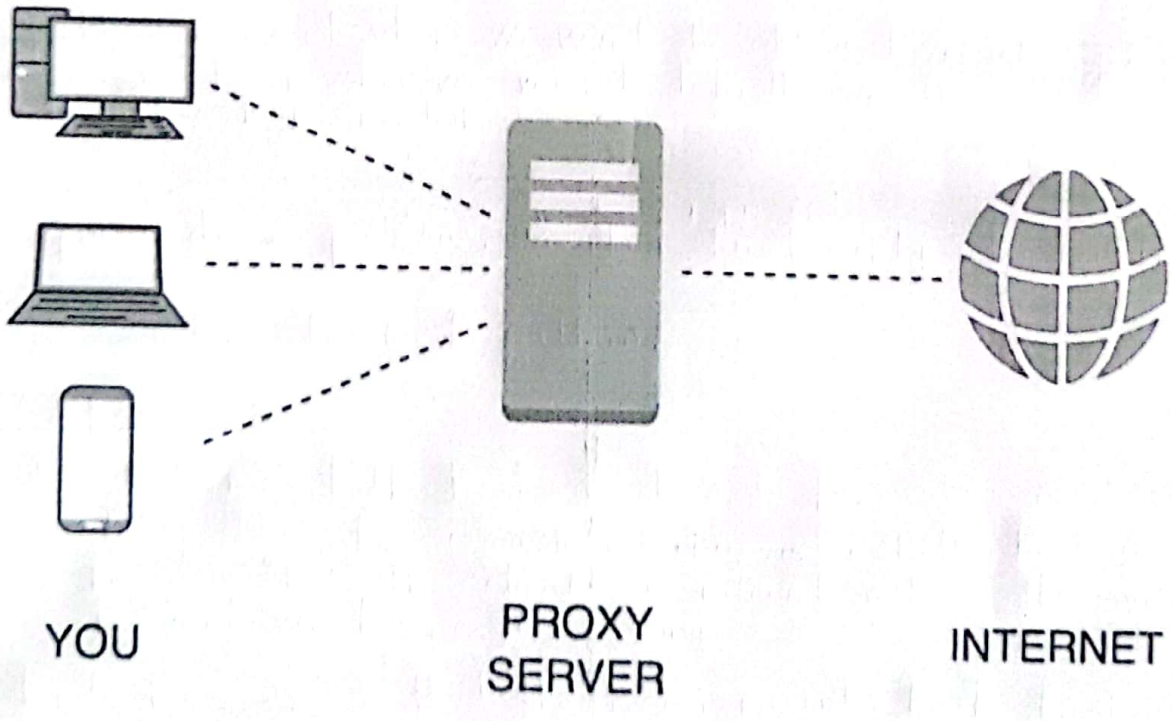
proxy → onbehalf of / surrogate / represent.

Proxy (Ambassador) Design Pattern

Design Patterns

Creational (5)	Structural (7)	Behavioral (11)
• Factory	• Adapter	• Chain Of Responsibility
• Abstract Factory	• Bridge	• Command
• Singleton	• Composite	• Interpreter
• Prototype	• Decorator	• Iterator
• Builder	• Facade	• Mediator
	• Flyweight	• Memento
	• Proxy	• Observer
		• State
		• Strategy
		• Template
		• Visitor

Who is a Proxy?



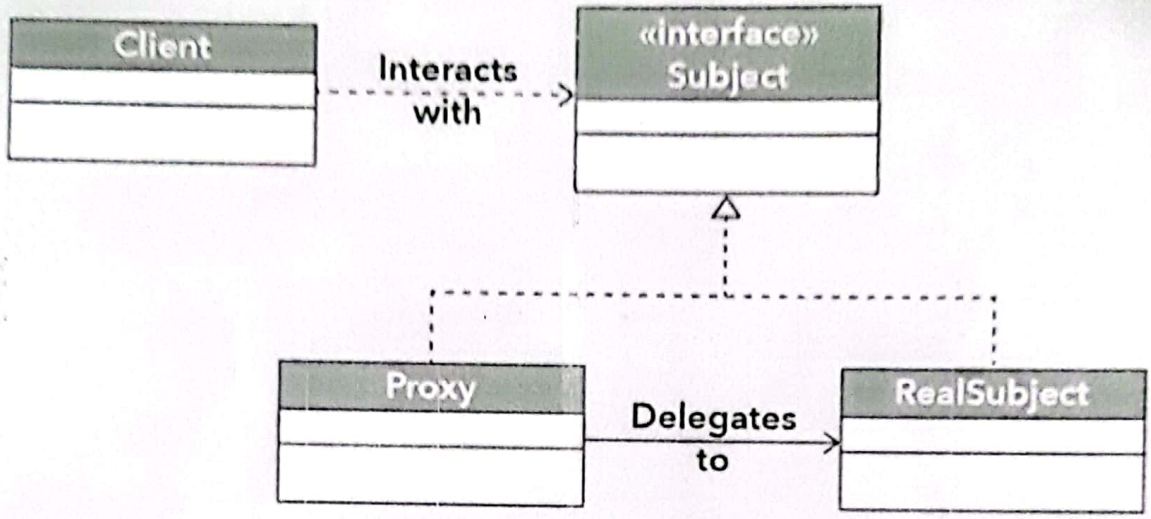
What is a Proxy?

- This design pattern provides a substitute for another object and controls access to that object, allowing you to perform something before or after the request reaches the original object.
- The proxy acts as a simplified or lightweight version of the original object.
- A proxy object is still able to accomplish the same tasks.
- It but may delegate requests to the original object to achieve them.

Common Scenarios

- Virtual Proxy
  - Resource incentive operations,
  - Lazy Loading
    - Image Editing, Gaming
- Protection Proxy
  - Authorization of different roles.
  - Users can access only their roles,
  - Data level security
- Remote Proxy
  - Working Remotely
- Smart Proxy.

- class diagram -



* Proxy design pattern provides a placeholder for another object to control access to it.
* This pattern is used when we want to add a level of control ove the access to an object, or when we want to defer the creation & initialization of an object until actually needed.
  - Proxy class should have the same methods as the real class.
* In proxy design pattern, a class represents functionality of another class.
* Comes under structural pattern.

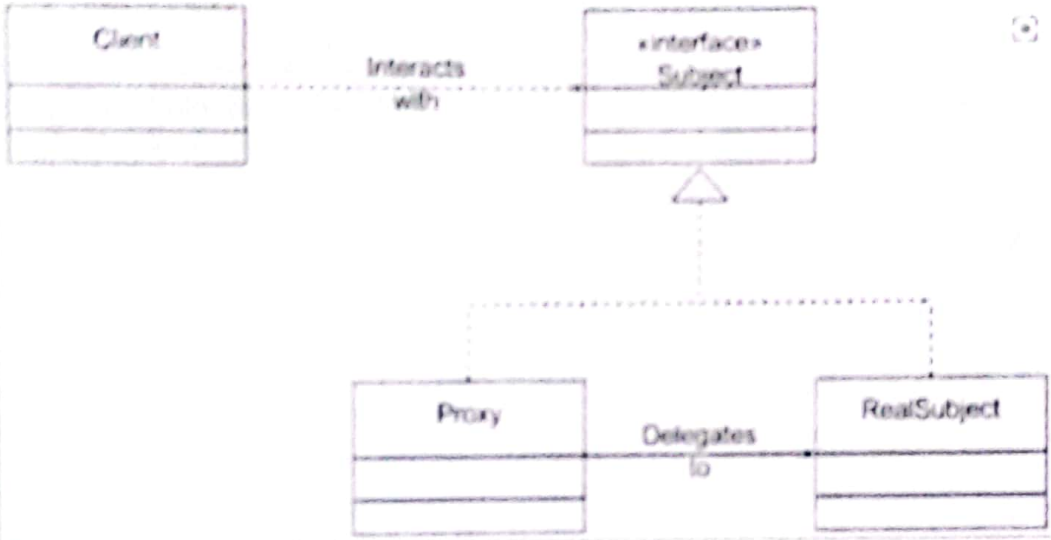
Key components;

- Subject: Interface both RealSubject and Poxy class must implement. It declares the common interface for both RealSubject and Proxy so that the client can interact with either without knowing which one it is using.
- RealSubject: Real object that the Proxy represents. RealSubject is the class that the Proxy is meant to stand in for control access to.
- Poxy : Placeholder to RealSubject. Control access to RealSubject. Can perform additional tasks as lazy initialization, access control, logging.

Question

(You may need to scroll down, or use full-screen mode, to review all four choices)

Which conclusions can be drawn about the proxy design pattern based on its structure as shown in the UML class diagram?



1. Proxy and RealSubject are subtypes of Subject.
2. The proxy design pattern achieves polymorphism through implementing a Subject interface.
3. The RealSubject class knows about the Proxy class, meaning that the RealSubject class has an attribute that refers to an instance of the Proxy class.
4. The proxy design pattern cannot achieve polymorphism because the Proxy and RealSubject classes are both implementing the Subject.

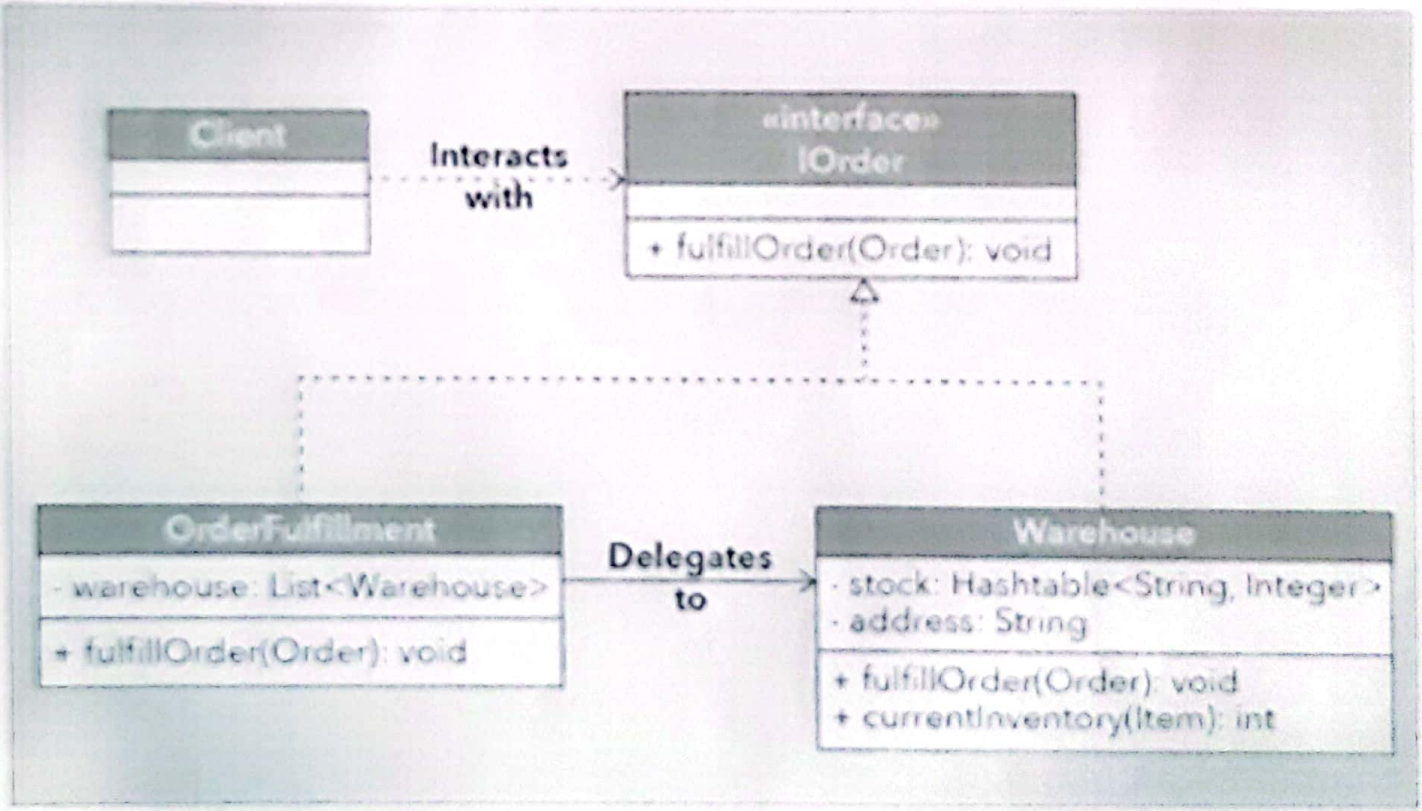
1. **Proxy and RealSubject are subtypes of Subject**

   Correct! This is clear from the way interface inheritance is shown with an open arrowhead. Since both Proxy and RealSubject must implement the methods of Subject, they subtypes

2. **The proxy design pattern achieves polymorphism through implementing a Subject interface.**

   Polymorphism is achieved with the interface, ensuring that both the Proxy and the RealSubject can be interacted with using the same methods, even if the internal implementation of these methods is different.

3. **The RealSubject class knows about the Proxy class, meaning that the RealSubject class has an attribute that refers to an instance of the Proxy class.**
4. **The proxy design pattern cannot achieve polymorphism because the Proxy and RealSubject classes are both implementing the Subject.**



Step 1: Design the subject interface

```
public interface IOrder {
    public void fulfillOrder(Order);
}
```

only one non implemented method.

Step 2: Implement the real subject class

```
public class Warehouse implements IOrder {
    private Hashtable<String, Integer> stock;
    private String address;

    /* Constructors and other attributes would go here */
    …

    public void fulfillOrder(Order order) {
        for (Item item : order.itemList)
            this.stock.replace(item.sku, stock.get(item)-1);

        /* Process the order for shipment and delivery */
        …
    }

    public int currentInventory(Item item) {
        if (stock.containsKey(item.sku))
            return stock.get(item.sku).intValue();
        return 0;
    }
}
```

```
public class OrderFulfillment implements IOrder {
    private List<Warehouse> warehouses;

    /* Constructors and other attributes would go here */

    public void fulfillOrder(Order order) {
        /* For each item in a customer order, check each warehouse
        to see if it is in stock.

        If it is then create a new Order for that warehouse. Else
        check the next warehouse.

        Send the all the Orders to the warehouse(s) after you finish
        iterating over all the items in the original Order. */
        for (Item item: order.itemList) {
            for (Warehouse warehouse: warehouses) {
                ...
            }
        }
        return;

    }
}
```

Step 3: Implement the proxy class

Which are key responsibilities of the proxy class?

1. It is receives its tasks from the real subject class.
2. It protects the real subject class by checking the client's request and controlling access to the real subject class.
3. It protects the real subject class by checking the client's request and controlling access to the real subject class.
4. It acts as a wrapper class for the real subject class.

1. It is receives its tasks from the real subject class.

   The proxy class delegates to the real subject class, it is not delegated tasks from the real subject class.

2. It protects the real subject class by checking the client's request and controlling access to the real subject class.

   One of the common uses for the proxy design pattern is to protect the real subject class. Access control is one of the ways the proxy class is able to accomplish this goal. In our order fulfillment example, the proxy class controls access by ensuring orders are sent,to warehouses that are able to service them.

3. It protects the real subject class by checking the client's request and controlling access to the real subject class.

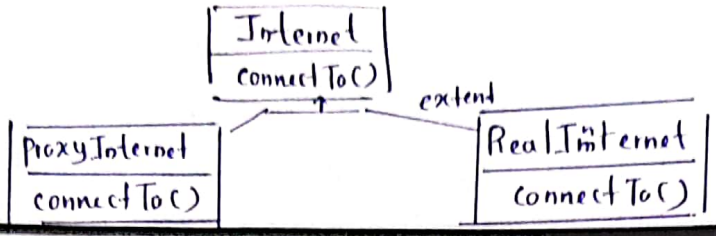
   One of the common uses for the proxy design pattern is to protect the real subject class. Access control is one of the ways the proxy class is able to accomplish this goal. In our order fulfillment example, the proxy class controls access by ensuring orders are sent to warehouses that are able to service them.

4. It acts as a wrapper class for the real subject class.

Actually the interface should be the same! Since the proxy and real subject class implement the same interface, the proxy class does not provide clients with an interface that is different from the real subject's interface (This is used in the adaptor design pattern)

Usage of Proxy Design Pattern

- Protect the real subject class
- To have a polymorphic design so that the client class can expect the same interface for the proxy and real subject classes,
- To use a lightweight proxy in place of a resource intensive object until it is actually needed,
- To implement some form of intelligent verification of requests from client code in order to determine if, how, and to whom the requests should be forwarded to
- To present a local representation of a system that is not in the same physical or virtual space.
- Manages the lifecycle of the service object and proxy will work even if the service object isn't ready or is not completely available.




```java
public interface Internet {
    void connectTo(String host);
}

public class RealInternet implements Internet {
    @Override
    public void connectTo(String host) {
        System.out.println("Opened connection to " + host);
    }
}
```

```java
public class ProxyInternet implements Internet {
    private static final List<String> bannedSites;
    private final Internet internet = new RealInternet();

    static {
        bannedSites = new ArrayList<>();
        bannedSites.add("banned.com");
    }

    @Override
    public void connectTo(String host) {
        if (bannedSites.contains(host)) {
            System.out.println("Access Denied!");
            return;
        }
        internet.connectTo(host);
    }
}
```

*now, the user who wants to benefit from the banned-websites functionality can use the proxy **without affecting other users***

```java
public static void main(String[] args) {
    Internet internet = new ProxyInternet();
    internet.connectTo("google.com");
    internet.connectTo("banned.com");
}
```

```java
public interface VideoDownloader {
    Video getVideo(String videoName);
}

public class RealVideoDownloader implements VideoDownloader {
    @Override
    public Video getVideo(String videoName) {
        System.out.println("Connecting to https://www.youtube.com/");
        System.out.println("Downloading Video");
        System.out.println("Retrieving Video Metadata");
        return new Video(videoName);
    }
}
```

```java
public static void main(final String[] arguments) {
    VideoDownloader videoDownloader = new RealVideoDownloader();
    videoDownloader.getVideo("geekific");
    videoDownloader.getVideo("geekific");
    videoDownloader.getVideo("likeNsub");
    videoDownloader.getVideo("likeNsub");
    videoDownloader.getVideo("geekific");
}
```

```
Connecting to https://www.youtube.com/
Downloading Video
Retrieving Video Metadata
Connecting to https://www.youtube.com/
Downloading Video
Retrieving Video Metadata
Connecting to https://www.youtube.com/
Downloading Video
Retrieving Video Metadata
Connecting to https://www.youtube.com/
Downloading Video
Retrieving Video Metadata
Connecting to https://www.youtube.com/
Downloading Video
Retrieving Video Metadata
```

```java
public interface VideoDownloader {
    Video getVideo(String videoName);
}

public class RealVideoDownloader implements VideoDownloader {
    @Override
    public Video getVideo(String videoName) {
        System.out.println("Connecting to https://www.youtube.com/");
        System.out.println("Downloading Video");
        System.out.println("Retrieving Video Metadata");
        return new Video(videoName);
    }
}
```

```java
public class ProxyVideoDownloader implements VideoDownloader {
    private final Map<String, Video> videoCache = new HashMap<>();
    private final VideoDownloader downloader = new RealVideoDownloader();

    @Override
    public Video getVideo(String videoName) {
        if (!videoCache.containsKey(videoName)) {
            videoCache.put(videoName, downloader.getVideo(videoName));
        }
        return videoCache.get(videoName);
    }
}
```

Implementation:

1) Lazy loading
2) Protection proxy – access control.

Microservices

Monolithic applications

A monolithic application is a traditional software architecture where all the components and functionalities are tightly integreted and packaged as a single unit. This is a single entity.

Key features;

- Single codebase
- Tight integration
- Single deployment unit
- Centralized database

Due to the limitations such as when application grow in complexity and scale, monolithic architecture can become harder to maintain, deploy and scale. So patterns like microservices came to use. (selection is based on requirements)

Microservices.

Microservices architecture is an approach of software develop-ment where complex application is divided to small, independent services that can be developed, deployed and scaled independently.

Each microservire is designed to perform a specific business function and communicates with other services through well defined API's (Application Programming interfaces)

Key characters.

- Decentralized
- Single responsibility – focused to one task.
- Scalable
- Continuous dilivery & deployment
- Diverse technology
- Independent datamanagement.

Limitations in microservices;

- High complexity in managing interservice communication.
- needed effective monitoring and management.
- Issues with consistency cross services.

Selecting monolithic or microservices depends on the requirements.