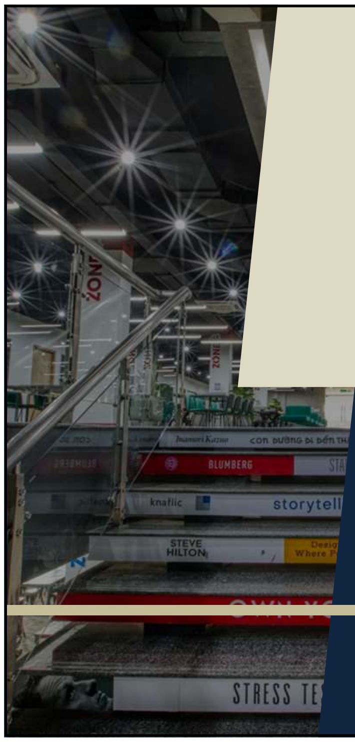


EIU
TRƯỜNG ĐẠI HỌC QUỐC TẾ MIỀN ĐÔNG
EASTERN INTERNATIONAL UNIVERSITY

CSE203 – OBJECT ORIENTED PROGRAMMING

Design Patterns

1

1

EIU

What is Design Pattern

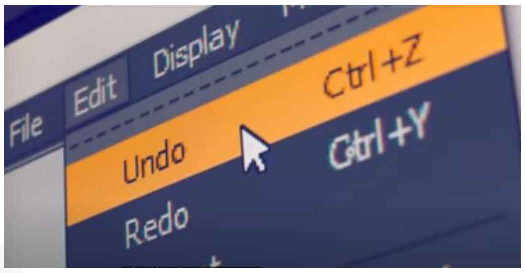
- Any engineer need the right tools to solve problems,
    - For developers and software designers, those tools are ideas, algorithms, and pre-made solutions
    - Design Pattern is such a tool
- Software developers may face a lot of problems in the time of developing a software
    - In fact, there are a number of problems that we would actually consider common across a variety of industries.
    - A group of design experts came together to identify these common problems and develop matching generalized solutions
    - Design patterns are these solutions. Each design pattern identifies two things, the problem that exists and the solution or process by which to solve the problem.
    - These are elegant solutions to repeating problems in software design

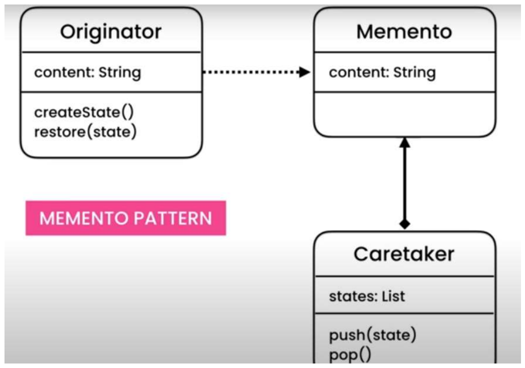
8 March 2024 2

2

EIU

What is Design Pattern





- For example, making undo mechanisms in your application
- This is an example of repeating problem in software design
- Different design patterns can be used to implement this feature
- Design patterns tell us how to structure classes and how those classes should talk to each other

.

8 March 2024 3

3

EIU

What is Design Pattern

- Design Pattern
  - supports design activities by providing general, reusable solutions to commonly occurring design problems

Design Patterns
Elements of Reusable Object-Oriented Software
By GoF

- Industry has some professionals who has provided some best practices
  - We can call these best practices as design pattern
  - There was a book published in 1994 by four authors
  - In this book, authors have mentioned about **23 design patterns**
  - These four authors are commonly known as “The Gang of Four” (GoF)
  - The design patterns provided by GoF can be applied in any object oriented language

8 March 2024 4

4

EIU

Categories of GoF Design Pattern

Creational
Structural
Behavioural

- Which design pattern should we use depends on software requirements
- **Creational Design Patterns** are different way to create objects
- **Structural Design Patterns** are about relationships between these objects
- **Behavior Design Patterns** are about interaction or communication between these objects
- **There is no rush in applying of Design Patterns**
  - At first, think about your problem domain, then logic and then choose which design pattern can you apply.
- **23 design patterns are classified in these three categories** and they are classic design patterns

EIU

Benefits of using Design Pattern

- **Why Should we use Design Patterns**?
  - Vocabulary to talk about design: Design Patterns can help us to communicate with other developers in more abstract level.
  - For example, you can tell your coworker, “Hey we can use this **Façade Pattern** to improve this code”
  - You just need to name of the design pattern to communicate your idea
  - You do not need to write a lots of code to express your idea

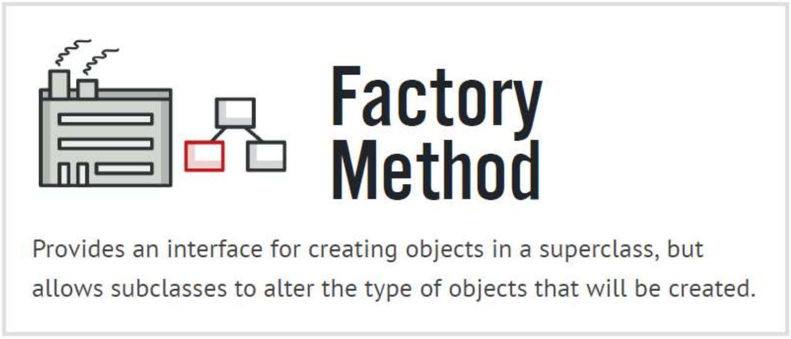
Benefits of using Design Pattern

- **It helps you to make a better designer**
  - Elements of reusable design: **You can learn how to make reusable, extensible and maintainable software**, no matter what kind of programming language you have used or what kind of application you have created
  - This help you to **learn any frameworks quickly, since same design patterns are used in various frameworks and libraries**

Creational Design Pattern

- In software design, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.
  - The basic form of object creation could result in design problems or added complexity to the design.
  - Creational design patterns solve this problem by some how controlling this object creation.
- Different design patterns included in this category are:
  - Abstract Factory: Creates an instance of several families of classes
  - Builder: Separates object construction from its representation
  - Factory Method: Creates an instance of several derived classes
  - Object Pool: Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
  - Prototype: A fully initialized instance to be copied or cloned
  - Singleton: A class of which only a single instance can exist

Factory Method


Factory Method

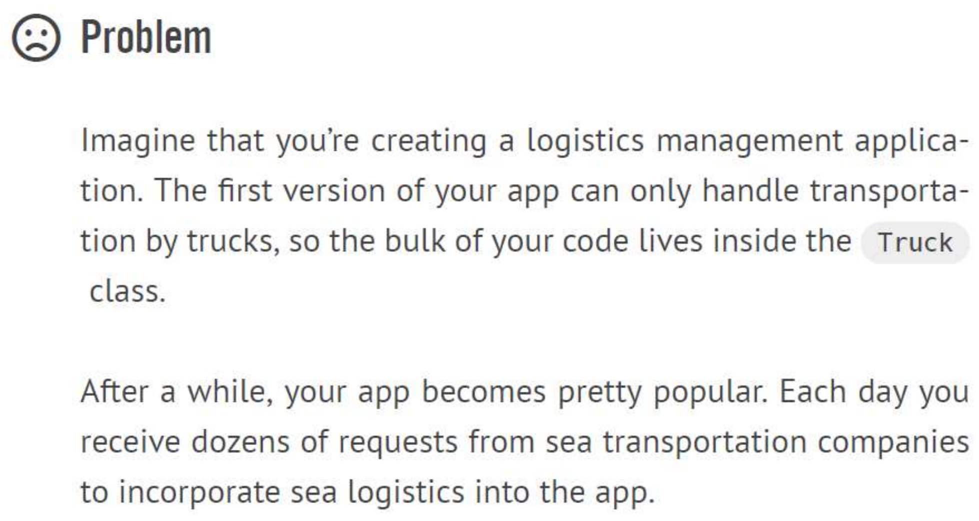
Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

❖ This design pattern is also known as Virtual Constructor

8 March 2024 9

9

Factory Method



☹ Problem

Imagine that you're creating a logistics management application. The first version of your app can only handle transportation by trucks, so the bulk of your code lives inside the `Truck` class.

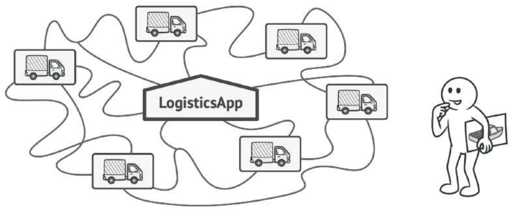
After a while, your app becomes pretty popular. Each day you receive dozens of requests from sea transportation companies to incorporate sea logistics into the app.

8 March 2024 10

10

Factory Method

Great news, right? But how about the code? At present, most of your code is coupled to the `Truck` class. Adding `Ships` into the app would require making changes to the entire codebase. Moreover, if later you decide to add another type of transportation to the app, you will probably need to make all of these changes again.



Adding a new class to the program isn't that simple if the rest of the code is already coupled to existing classes.

Factory Method

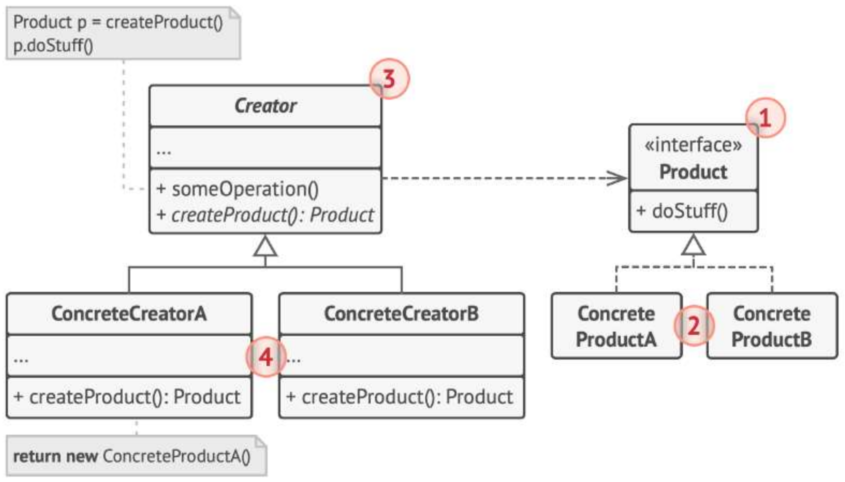
😊 Solution

The Factory Method pattern suggests that you replace direct object construction calls (using the `new` operator) with calls to a special *factory* method. Don't worry: the objects are still created via the `new` operator, but it's being called from within the factory method. Objects returned by a factory method are often referred to as *products*.

EIU

Factory Method

Structure



8 March 2024 13

13

EIU

Factory Method

1. The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

2. **Concrete Products** are different implementations of the product interface.

3. The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

4. **Concrete Creators** override the base factory method so it returns a different type of product.

8 March 2024 14

14

Example of Factory Method Design Pattern

- Consider a pizza store. In an application, you can model the pizzas as concrete Pizza objects, such as CheesePizza, PepperoniPizza, and VeggiePizza. Just as in any pizza store, a customer can only order a pizza and not make it.

- In the application, you can create an abstract class, say BasePizzaFactory with a factory method to create a pizza. You can then create a subclass of BasePizzaFactory, called PizzaFactory to implement the factory method. I n the factory method, you can create and return a proper Pizza object.

Example of Factory Method Design Pattern

- The components of the factory method pattern in the context of the pizza store can be summarized as:
  - ❖ Product (Pizza): Is an interface or an abstract class whose subclasses are instantiated by the factory method.
  - ❖ ConcreteProduct (CheesePizza, PepperoniPizza, and VeggiePizza): Are the concrete subclasses that implement/extend Product.
  The factory method instantiates these subclasses.
  - ❖ Creator (BasePizzaFactory): Is an interface or an abstract class that declares the factory method, which returns an object of type Product.
  - ❖ ConcreteCreator (PizzaFactory): Is a concrete class that implements the factory method to create and return a ConcreteProduct to Client.
  - ❖ Client: Asks the Creator for a Product.

Example of Factory Method Design Pattern

- let us first create the abstract Pizza class and its subclasses.

```
public abstract class Pizza {
    public abstract void addIngredients();
    public void bakePizza() {
        System.out.println("Pizza baked at 400 for 20 minutes.");
    }
}
```

- The Pizza class with an abstract addIngredients() method and defined a bakePizza() method.

Example of Factory Method Design Pattern

- SubClass- CheesePizza

```
public class CheesePizza extends Pizza {
    @Override
    public void addIngredients() {
        System.out.println("Preparing ingredients for cheese pizza.");
    }
}
```

Example of Factory Method Design Pattern

➢ SubClass- PepperoniPizza

```
public class PepperoniPizza extends Pizza {
    @Override
    public void addIngredients() {
        System.out.println("Preparing ingredients for pepperoni pizza.");
    }
}
```

Example of Factory Method Design Pattern

➢ SubClass- VeggiePizza

```
public class VeggiePizza extends Pizza {
    @Override
    public void addIngredients() {
        System.out.println("Preparing ingredients for veggie pizza.");
    }
}
```

➢ All subclasses provide their own implementation of the addIngredients() method. The subcl asses being derived from Pizza inherit the bakePizza() method defined in Pizza

Example of Factory Method Design Pattern

➢ Next, we will create the abstract BasePizzaFactory class with a createPizza() factory method. createPizza() is an abstract method

```
public abstract class BasePizzaFactory
{
  public abstract Pizza createPizza(String type);
}
```

Example of Factory Method Design Pattern

➢ The factory method implementation in a concrete subclass, PizzaFactory

```
public class PizzaFactory extends BasePizzaFactory{
   @Override
   public  Pizza createPizza(String type){
      Pizza pizza;
      switch (type.toLowerCase())
      {case "cheese":
            pizza = new CheesePizza();
            break;
         case "pepperoni":
            pizza = new PepperoniPizza();
            break;
         case "veggie":
            pizza = new VeggiePizza();
            break;
         default: throw new IllegalArgumentException("No such pizza.");
      }pizza.addIngredients();
      pizza.bakePizza();
      return pizza;
   }}
```

Example of Factory Method Design Pattern

- The client

```
public class ClientPizza {

    public static void main(String args[ ]){
        BasePizzaFactory  pizzaFactory = new PizzaFactory();
        Pizza cheesePizza = pizzaFactory.createPizza("cheese");
        Pizza veggiePizza = pizzaFactory.createPizza("veggie");
    }
}
```

EIU

Structural Design Pattern

- Structural patterns, describe how objects are connected to each other.
- Previously, we examined the major design principles like Association, Aggregation, Composition and Generalization, and how they are expressed in UML class diagrams
- There are many different ways you can structure objects depending on the relationship you'd like between them.
- Not only do structural patterns describe how different objects have relationships, but also how subclasses and classes interact through inheritance.
- Structural Patterns use these relationships, and describe how they should work to achieve a particular design goal.

EIU

Structural Design Pattern

- The Structural Design Patterns are classified as
  - Adapter. Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.
  - Bridge. Decouples an abstraction so two classes can vary independently.
  - Composite. Takes a group of objects into a single object.
  - Decorator. Allows for an object's behavior to be extended dynamically at run time.
  - Facade. Provides a simple interface to a more complex underlying object.
  - Flyweight. Reduces the cost of complex object models.
  - Proxy. Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

EIU

Facade Design Pattern

- If software system become larger, they naturally become more complex.
- This can be potentially confusing for the client classes in your system to use.
- System complexity is not always a sign of poor design, though.
- The scope of the problem you're trying to solve may be so large, it requires a complex solution.
- Client classes, however, would prefer a simpler interaction.
- The facade design pattern provides a single simplified interface for client classes to interact with the subsystem.

EIU

What is Facade?

- ➢ If you've ever gone shopping at a retail store, eaten at a restaurant, or ordered anything online, you've had experience interacting with the real world facade design pattern.
- ➢ Imagine yourself walking down the street looking for a place to have dinner. What do you look for when you're at street? Naturally, you look for a sign on the face of a building that indicates that the contents within the structure can provide you with the dining service.

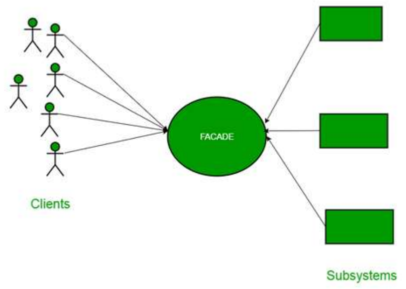
EIU

What is Facade?

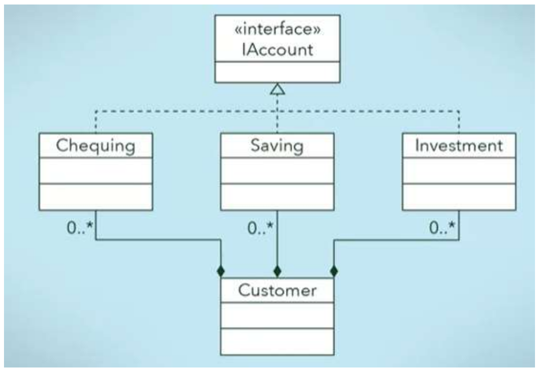
- ➢ Similarly, when you're shopping online, you look for some outward indication that the website you're on has a virtual storefront. These outward facing indicators are used to communicate to the public what types of services are available.
- ➢ When a waiter takes and places your order, or when an online sales platform sends your order to be fulfilled, they are acting as part of a facade by hiding away all the extra work that needs to be done
- ➢ You are able to purchase goods and services without having to know how the request is processed. The entire construction of the buildings, wait-staff, menus, we bsites and many other components are behind facades.
- ➢ Keep in mind that a facade does not actually add more functionality, a facade simply acts as a point of entry into your subsystem.

What is Facade?

- In software, the facade design pattern does exactly what a waiter or salesperson would do in real life.
- A facade is a wrapper class that encapsulate the subsystem in order to hide the subsystem's complexity.
- This wrapper class will allow a client class to interact with the subsystem through a façade
- Let's take a look at how client code would interact with the subsystem without a facade class for a simple banking system



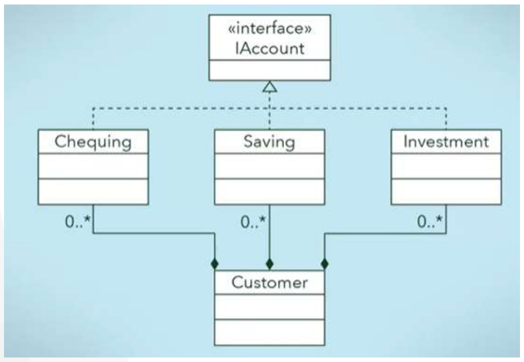
Client – Subsystem Interaction without Facade



- Without a facade class, the customer class would contain instances of the checking, saving and investment classes.
- This means that the customer is responsible for properly instantiating each of these constituent classes and knows about all their different attributes and methods.

EIU

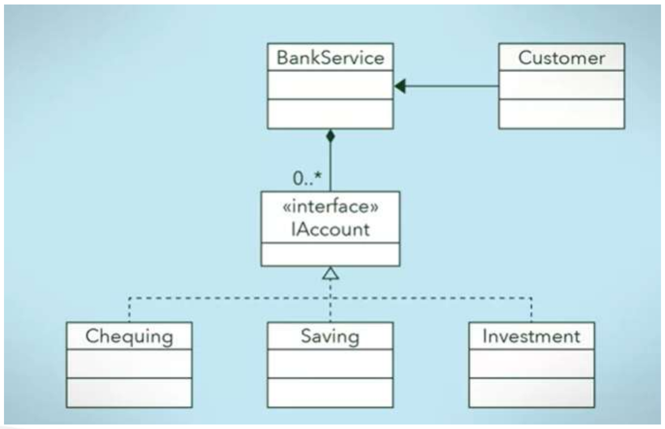
Client – Subsystem Interaction without a Façade class



➢ This is like having to manage all of your own financial accounts in real life, which can be complex with lots of accounts, instead of letting a financial institution do it for you.

EIU

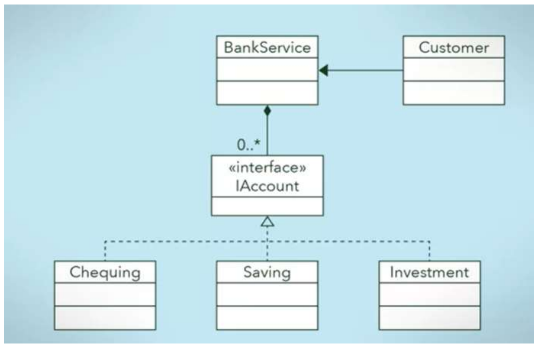
Client – Subsystem Interaction using a Facade class



➢ Instead, we introduce the bank service class to act as a facade for the checking, saving, and investment classes.
➢ The customer class no longer needs to handle instantiation or deal with any of the other complexities of financial management.

EIU

Client – Subsystem Interaction using a Façade class



- Since the three different accounts all implement the IAccount interface, the bank's service class is effectively wrapping the account interfacing classes, and presenting a simpler front to them for the customer client class to use..
- The facade design pattern is simple to apply in our bank accounts example.
- It combines interface implementation by one or more classes which then gets wrapped by the facade class.

EIU

Question?

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

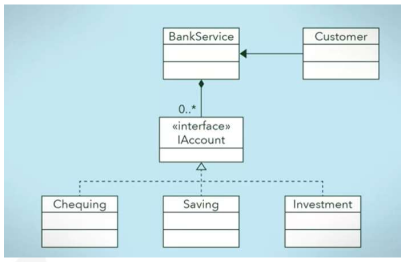
a) You need a class that will instantiate other classes within your system and provide these instances to a client class.

b) You need a class to translate messages between two existing subsystems because one is exp ecting a specific interface to use but is provided with an interface that is incompatible.

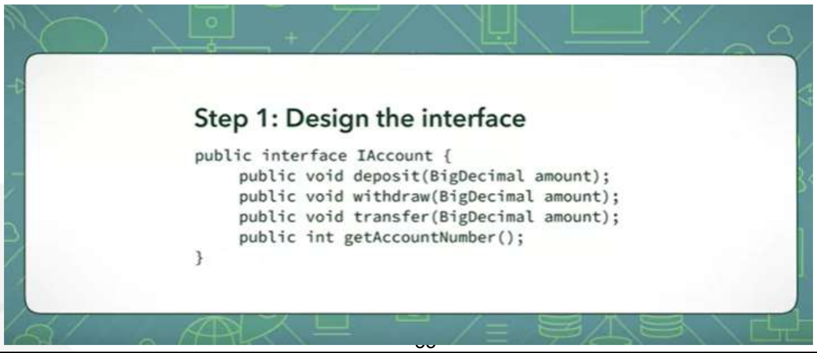
c) You need a class to act as an interface between your subsystem and a client class.

d) You need to simplify the interaction with your subsystem for client classes.

Example of Facade Design Pattern



- This Java interface is the one that will be implemented by the different account classes and will not be known to the customer class



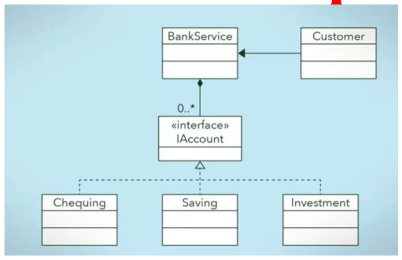
Step 1: Design the interface

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

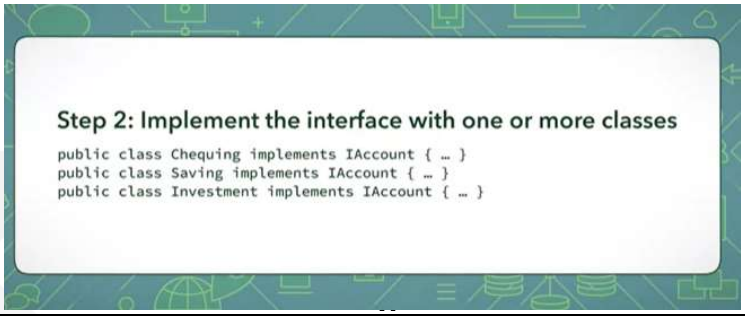
8 March 2024

35

Example of Facade Design Pattern



- Remember that interfaces allow us to create subtypes which means that
- checking, saving, and investment are subtypes of i-Account and are expected to Behave like an account type



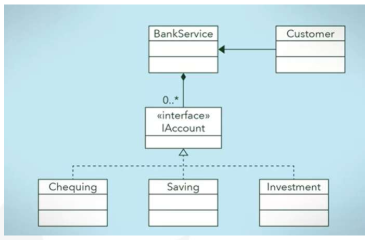
Step 2: Implement the interface with one or more classes

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

8 March 2024

36

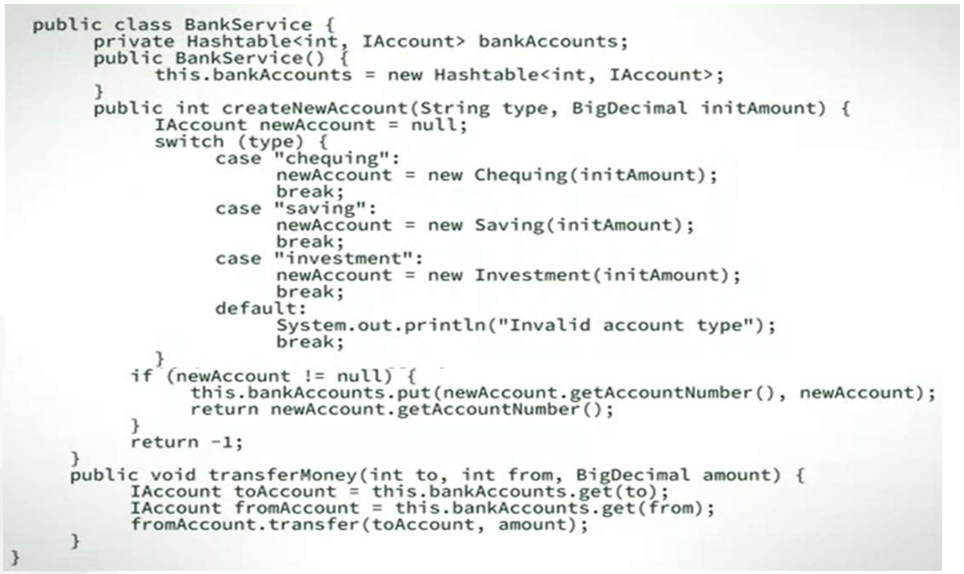
Example of Facade Design Pattern



- In this example, we are only implementing and hiding one interface, but in practice, a facade class can be used to wrap all the interfaces and classes for a subsystem.
- It is our decision as to what we want to wrap
- In this example, BankService class is the facade

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

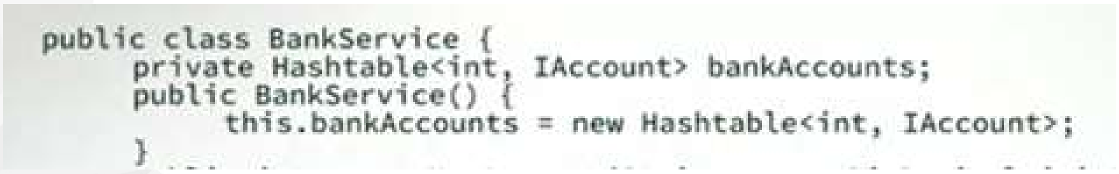
Example of Facade Design Pattern



```
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);
    }
}
```

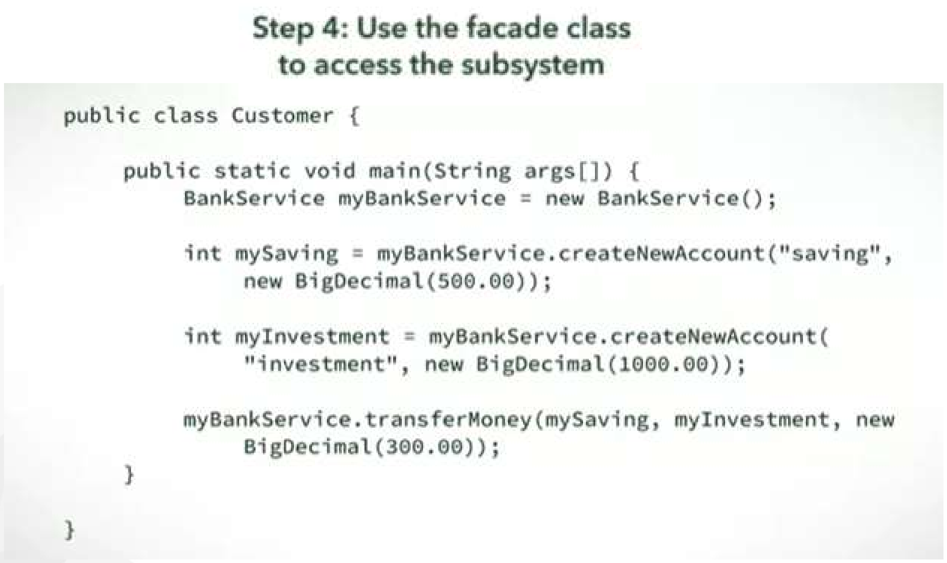
Example of Facade Design Pattern

- Notice that its public methods are simple to use and show no hint of the underlying interface and implementing classes.
- Another thing to note is that we set the access modifiers for each account to be private.
- Since the entire point of the facade design pattern is to hide complexity, we use the information hiding design principle to prevent all client classes from seeing the account objects and how these accounts behave.



```
public class BankService {
    private Hashtable<int, IAccount> bankAccounts;
    public BankService() {
        this.bankAccounts = new Hashtable<int, IAccount>;
    }
```

Example of Facade Design Pattern



Step 4: Use the facade class to access the subsystem

```
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));
    }

}
```

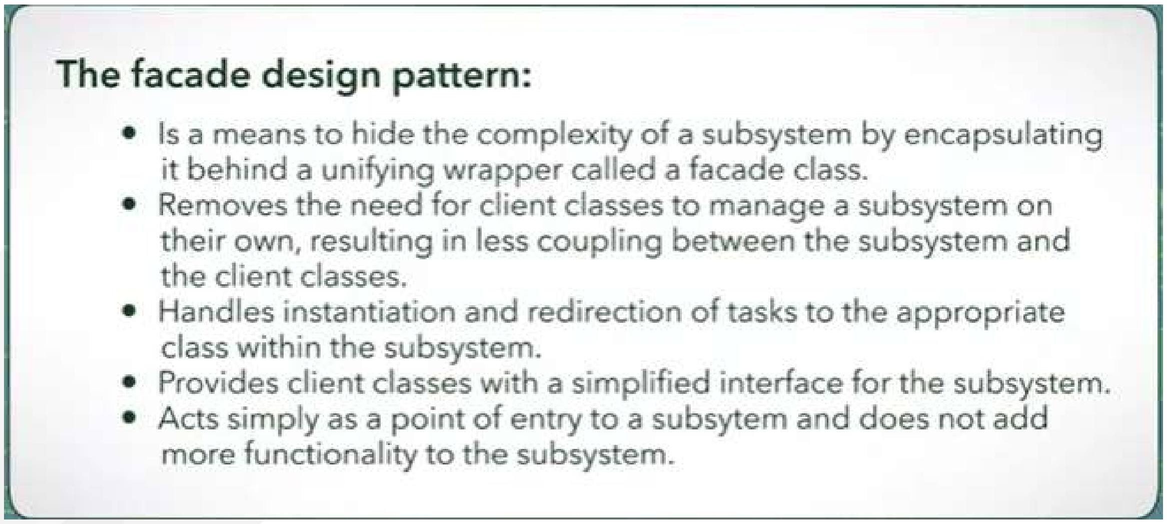
EIU

Example of Facade Design Pattern

- ➢ Client classes can access the functionalities of the different accounts through the methods of the BankService class.
- ➢ The BankService class will tell the client what type of actions it will allow the client to call upon, and then will delegate that action to the appropriate account objects.
- ➢ Now that we have our facade in place, our client class can access its accounts through the BankService.
- ➢ Customer class does not need to worry about creating and managing its own accounts.
- ➢ The customer simply needs to know about the BankService and the set of behaviors the BankService is capable of performing.
- ➢ We have effectively hidden the complexity of account management from the customer using the BankService facade class.

EIU

Summary: Facade Design Pattern



The facade design pattern:

- Is a means to hide the complexity of a subsystem by encapsulating it behind a unifying wrapper called a facade class.
- Removes the need for client classes to manage a subsystem on their own, resulting in less coupling between the subsystem and the client classes.
- Handles instantiation and redirection of tasks to the appropriate class within the subsystem.
- Provides client classes with a simplified interface for the subsystem.
- Acts simply as a point of entry to a subsytem and does not add more functionality to the subsystem.

Behavioral Design Pattern

- ➢ Behavioral Design Pattern focus on how objects distribute work.
- ➢ They describe how each object does a single cohesive function.
- ➢ Behavioral patterns also focus on how independent objects work towards a common goal.
- ➢ Think of a behavioral pattern like a race car pit crew at a track.
- ➢ In the pit crew, the roles of the members describe how the team is able to achieve victory, which is their common goal.

Behavioral Design Pattern

➢ Each member has a specific responsibility, their role in the race.
➢ Some members change the tires, others unmount, and mount the wheel nuts, others refuel the car, but all must work together to win.
➢ Like a game plan, a Behavioral Pattern lays out the overall goal and the purpose for each of the objects.

EIU

Behavioral Design Pattern

- The Behavioural Design Patterns are classified as
  - Command. Creates objects which encapsulate actions and parameters.
  - Iterator. Accesses the elements of an object sequentially without exposing its underlying representation.
  - Mediator. Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
  - Observer. Is a publish/subscribe pattern which allows a number of observer objects to see an event.
  - State. Allows an object to alter its behavior when its internal state changes.
  - Strategy. Allows one of a family of algorithms to be selected on-the-fly at run-time.
  - Template Method. Defines the skeleton of an algorithm as an abstract class, allowing its sub-classes to provide concrete behavior.
  - Visitor. Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

EIU

Observer Pattern

- Imagine that you have a favorite blog. Every day you visit it multiple times per day to check for new blog posts.
- After a while, you get fed up with this routine. There must be a better way you think to yourself.
- The first solution that crosses your mind is to write a script to check the blog for new posts every fraction of a second.
- But upon further consideration, you realize most sites would not appreciate this barrage of requests, and would block your IP address.
- To avoid this, you instead write a script to check the blog for new posts once per hour.

EIU

Observer Pattern

- But to your disappointment, this means that you're now missing out on blog posts that provide live changes, such as real time posts about your favorite TV show.
- A better solution to this problem is for the blog to notify you every time a new post is added.
- You would subscribe to the blog, every time a new post was published, the blog would notify each subscriber, inc luding you.
- This way, you would be notified of the new content when it is created, rather than having to pull for this informati on at some interval

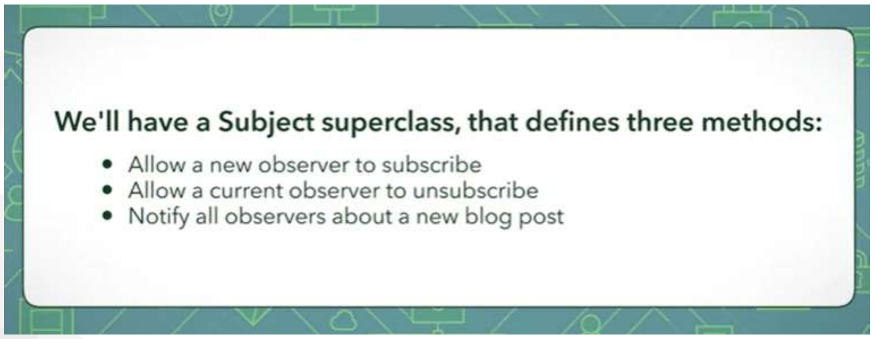
EIU

Observer Pattern

- The observer design pattern is a pattern where a subject keeps a list of observers.
- Observers rely on the subject to inform them of changes to the state of the subject.
- In an observer design pattern, there is generally a Subject superclass, which would have an attribute to keep track of all the observers.
- There is also an Observer interface with a method so that an observer can be notified of state changes to the subject.
- The Subject superclass may also have subclasses that implement the Observer interface.
- These elements create the relationship between the subject and observer.

Observer Pattern

- The subject in our example, the blog, would keep a list of observers.
- In this example, we might think of these observers as subscribers to the blog
- The observers rely on the blog to inform them of any changes to the state of the blog such as, when a new blog post is added.
- how we could apply this idea to solve this problem related to the blog?
- First:



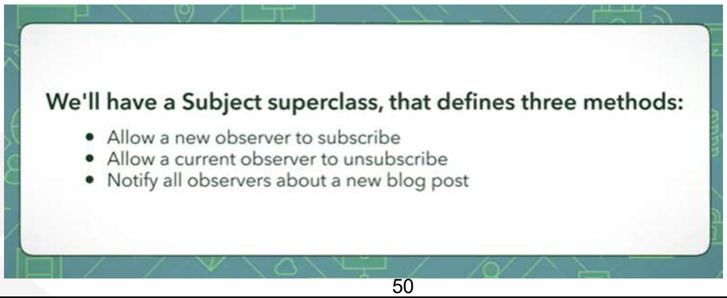
We'll have a Subject superclass, that defines three methods:

- Allow a new observer to subscribe
- Allow a current observer to unsubscribe
- Notify all observers about a new blog post

Observer Pattern

- This superclass would also have an attribute to keep track of all the observers, and will make an observer interface with methods that an observer can be notified to update itself
- Next, the blog Class will be a subclass of the Subject superclass, and the Subscriber Class will implement the observer interface.
- Observer design elements are essential to forming a Subject and Observer relationship.

For example, for the blog and subscriber

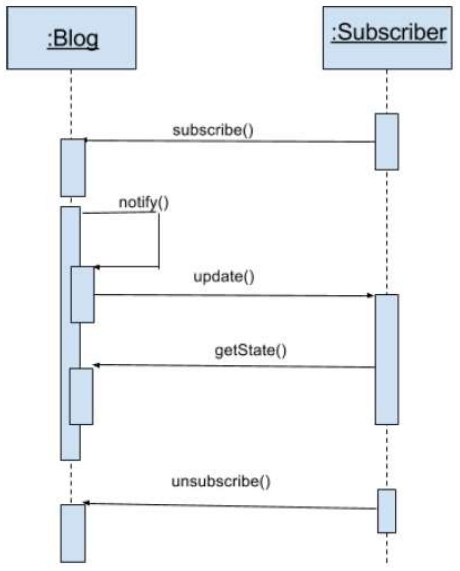


We'll have a Subject superclass, that defines three methods:

- Allow a new observer to subscribe
- Allow a current observer to unsubscribe
- Notify all observers about a new blog post

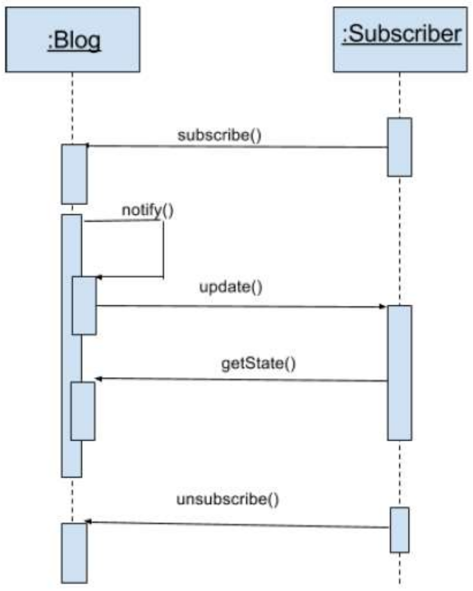
Observer Pattern

- ➢ Using Sequence Diagram, we can describe this kind of relationship
- ➢ A sequence diagram for observe patterns will have two major roles: the subject (the blog) and the observer (a subscriber).
- ➢ In order to form the subject and observer relationship, a subscriber must subscribe to the blog.
- ➢ The blog then needs to be able to notify subscribers of a change.
- ➢ The notify function keeps subscribers consistent, and is only called when a change has been made to the blog.



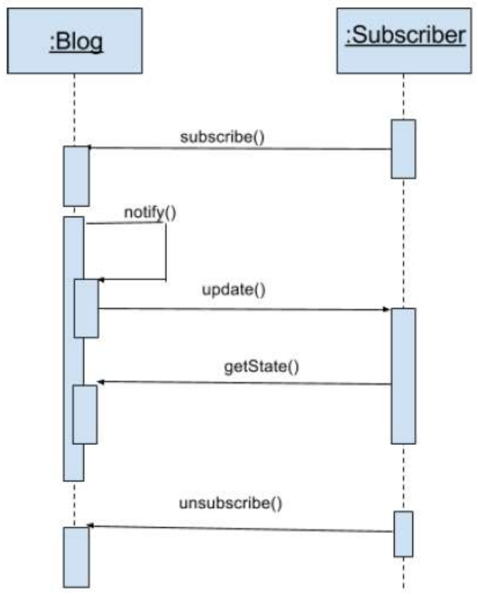
Observer Pattern

- ➢ If a change is made, the blog will make an update call to update subscribers.
- ➢ Subscribers can get the state of the blog through a getState call.
- ➢ It is up to the blog to ensure its subscribers get the latest information.
- ➢ To unsubscribe from the blog, subscribers could use the last call in the sequence diagram.
- ➢ unsubscribe() originates from the subscriber and lets the blog know the subscriber would like to be removed from the list of observers.



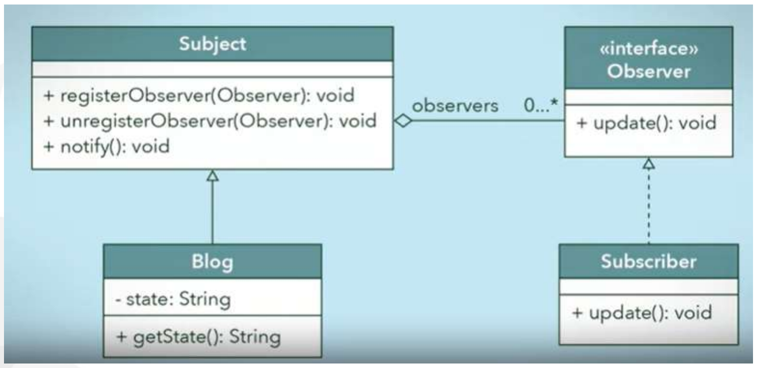
Observer Pattern

- If a change is made, the blog will make an update call to update subscribers.
- Subscribers can get the state of the blog through a getState call.
- It is up to the blog to ensure its subscribers get the latest information.
- To unsubscribe from the blog, subscriber could use the last call in the sequence diagram
- unsubscribe() originates from the subscriber and lets the blog know the Subscriber would like to be removed from the list of observers.



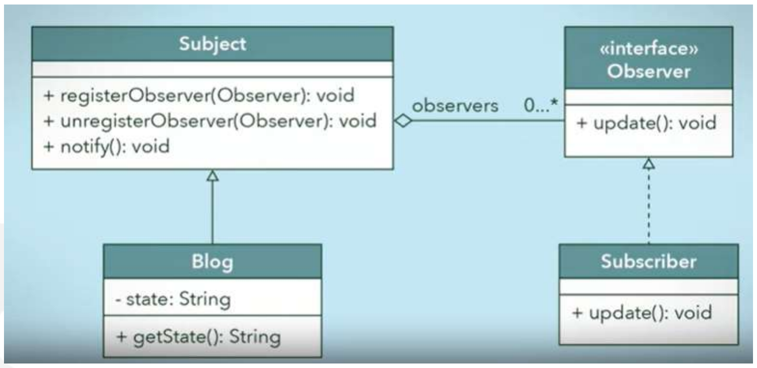
Observer Pattern: UML Diagram

- Let's take a look, in the UML diagram of this example of Observer Pattern.
- The Subject superclass has three methods: register observer, unregister observer, and notify. These are essential for a subject to relate to its observers.
- A subject may have zero or more observers registered at any given time.



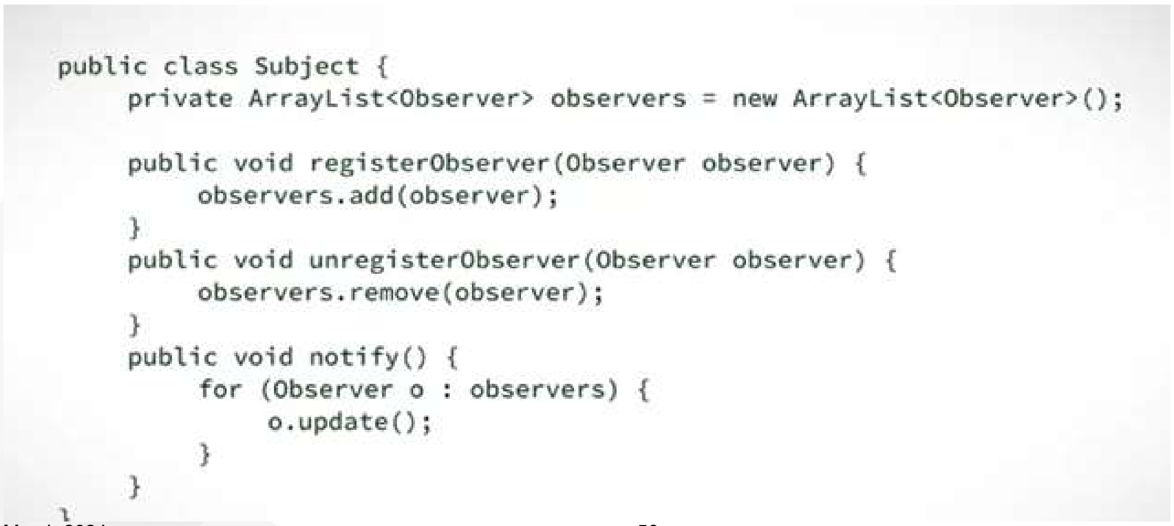
Observer Pattern: UML Diagram

- The Blog subclass would inherit these methods.
- The Observer interface only has the update method. An observer must have some way to update itself.
- The Subscriber class implements the Observer interface, providing the body of an update method so a subscriber can get what changed in the blog.



Observer Pattern: Java Code

- Let's see the java code for subject superclass.

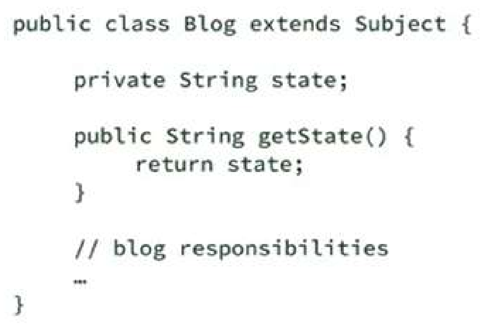


```
public class Subject {
    private ArrayList<Observer> observers = new ArrayList<Observer>();

    public void registerObserver(Observer observer) {
        observers.add(observer);
    }
    public void unregisterObserver(Observer observer) {
        observers.remove(observer);
    }
    public void notify() {
        for (Observer o : observers) {
            o.update();
        }
    }
}
```

EIU

Observer Pattern: Java Code

- ➢ Next the blog class is ta subclass of the subject.
- ➢ which will inherit the register observer, unregister observer and notify methods and have the other responsibilities of managing a blog and posting messages.

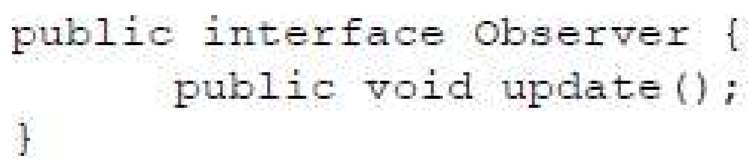


```
public class Blog extends Subject {

    private String state;

    public String getState() {
        return state;
    }

    // blog responsibilities
    …
}
```

EIU

Observer Pattern: Java Code

- ➢ The Observer interface makes sure all observer objects behave the same way.
- ➢ There is only a single method to implement, update(), which is called by the subject.
- ➢ The subject makes sure when a change happens, all its observers are notified to update themselves.



```
public interface Observer {
        public void update();
}
```

EIU

Observer Pattern: Java Code

➢ The subject makes sure when a change happens, all its observers are notified to update themselves. In this example, there is a class Subscriber the implements the Observer interface.

```
class Subscriber implements Observer {

        public void update() {
                // get the blog change
                ...
        }
}
```

EIU

Observer Pattern: Summary

➢ The observer design pattern can save you a lot of time when you're implementing your system.
➢ If you know that you have many objects that rely on the state of one, the value of the observer pattern becomes more pronounced.
➢ Instead of managing all the observer objects individually, you can have your subject manage them and make sure the observers are updating themselves as needed.
➢ There are many different ways and situations you can apply the observer design pattern.
➢ As a behavioral pattern, it makes it easy to distribute and handle notifications of changes across systems, in a manageable and controlled way.