# Concept of GRASP

- In analysis phase requirements are collected to find objects and use cases.
- Other design domain model, supplementary specification and SSD are developed during analysis phase.
- In design phase methods or responsibilities are added to the software and design.
- Object design defines the responsibilities of the system.
- Responsibility defines the behavior of an object.

# Responsibilities and Methods

- Some of the responsibilities during OOAD.
  - Creating an Object
  - Initiating action in other object.
  - Controlling and Coordinating activities)
  - Knowing type of data (Private, Protected and Public)
  - Knowing about related object
  - Knowing about things it can derive or calculate

# GRASP

- It stands for General Responsibility Assignment Software Pattern.
- It is the fundamental principle which guides in assigning responsibility for collaborating objects.
- These principle guides
  - Creating an Object
  - Information required to create object
  - Assigning the responsibility to collaborating object
  - Dependency between the objects

# Fundamental GRASP Pattern

- Creater
- Information Expert
- Controller
- Low Coupling
- High Cohesion
- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variation

# Creator

- **Creator** is a **GRASP Pattern** which helps to decide which class should be responsible for creating a new instance of a class. Object creation is an important process, and it is useful to have a principle in deciding who should create an instance of a class.
- **Problem**
- Who should be responsible for creating a new instance of some class?

# Solution of Creator Design Pattern

- Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.
- In general, a class **B** should be responsible for creating instances of class **A** if one, or preferably more, of the following apply:
- Instances of **B** contain or compositely aggregate instances of **A**
- Instances of **B** record instances of **A**
- Instances of **B** closely use instances of **A**
- Instances of **B** have the initializing information for instances of **A** and pass it on creation

# Example of Creator

- Let's consider **Point of Sale** (POS) application: Here is a brief overview of the **POS** application.
- POS application for a shop, restaurant, etc. that registers sales.
- Each sale is of one or more items of one or more product types and happens at a certain date.
- A product has a specification including a description, unitary price, an identifier.
- The application also registers payments (say, in cash) associated with sales.
- Payment is for a certain amount, equal to or greater than the total of the sale.

# Information Expert

- **Information expert (also expert or the expert principle)** is a principle used to determine where to delegate responsibilities such as methods, computed fields, and so on.
- **Problem**
- What is a general principle of assigning responsibilities to objects?

# Solution of Information Expert

- Assign responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.
- **Example**
- Let's consider Point of Sale (POS) application: a brief overview of the POS application.
- Application for a shop, restaurant, etc. that registers sales.
- Each sale is of one or more items of one or more product types and happens at a certain date.
- A product has a specification including a description, unitary price, an identifier.
- The application also registers payments (say, in cash) associated with sales.
- Payment is for a certain amount, equal to or greater than the total of the sale. Problem statement: Who's responsible for knowing the grand total of a Sale?

# Example

- Let us consider we need to calculate the grand total.
- For this we should find out who should be responsible for knowing the grand total of a sale.
- By Information Expert we should look for that class of object that has sufficient information to calculate total.
- To calculate grand total we need sufficient information.
- We need to know about SaleLineItem instance and sum of their sub total.

# Example

- These all information is contained by sale object(i.e Sale contains SaleLineItem) and to fulfill such responsibility getTotal) and to fulfill such responsibility getTotal method is added to sale class.
- Therefore, from the information expert sale is a suitable class of object for this responsibility( i.e sale is information expert for calculating grand Total.

# Low Coupling

- Coupling is the measure of how strongly one element is connected to, has knowledge of or relies on other elements.
- An element with low coupling is not depedent on too many other elements.
- Low coupling supports the design of the classes that are more independent which reduces the impact of change.
- It encourages to assign the responsibility in such a way that its placement does not increase the dependency to a level that leads to the negative results.

# Benefit of Low Coupling

- Not affected by changes in other components
- Simple to understand in isolation
- Convenient to reuse.

# Key Points about Low Coupling

- Low dependence between artifacts (“ Classes, Modules and Components”).
- There shouldn’t bee to much of dependency between the modules, even if there is a dependency it should be via the interfaces and should be minimal.
- Strive for loosely coupled design between objects that interacts.

# Source Code for Low Coupling

Step 1: `Vehicle` interface to allow loose coupling implementation.

```
interface Vehicle {
    public void move();
}
```

Step 2: `Car` class implements `Vehicle` interface.

```
class Car implements Vehicle {
    @Override
    public void move() {
        System.out.println("Car is moving");
    }
}
```

# Source Code for Low Coupling

Step 3: `Bike` class implements `Vehicle` interface.

```
class Bike implements Vehicle {

    @Override

    public void move() {

         System.out.println("Bike is moving");

    }

}
```

# Source Code for Low Coupling

Step 4: Now create a `Traveler` class which holds the reference to the `Vehicle` interface.

```
class Traveler {
    private Vehicle v;
    public Vehicle getV() {
         return v;
    }
    public void setV(Vehicle v) {
        this.v = v;
    }

    public void startJourney() {
         v.move();
    }
}
```

# Source Code for Low Coupling

Step 5: Test class for loose coupling example - `Traveler` is an example of loose coupling.

```
public static void main(String[] args) {
    Traveler traveler = new Traveler();
    traveler.setV(new Car()); // Inject Car dependency
    traveler.startJourney(); // start journey by Car
    traveler.setV(new Bike()); // Inject Bike dependency
    traveler.startJourney(); // Start journey by Bike
}
```

# High Cohesion

- High Cohesion is the measure of how strongly related and focused the responsibilities of an element with highly related responsibilities that does not do a tremendous amount of work has high cohesion.
- **High cohesion** is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of **low coupling.**

# Key Points about High Cohesion

- The code has to be very specific in its operations.
- The responsibilities/methods are highly related to the module.
- The term cohesion is used to indicate the degree to which a class has a single, well-focused responsibility.
- A class is identified as a low cohesive class when it contains many unrelated functions within it.

# Code for High Cohesion

```
class HighCohesive {
    // -------------- functions related to read resource
    // read the resource from disk
    public String readFromDisk(String fileName) {
         return "reading data of " + fileName;
    }

    // read the resource from web
    public String readFromWeb(String url) {
         return "reading data of " + url;
    }

    // read the resource from network
    public String readFromNetwork(String networkAddress) {
         return "reading data of " + networkAddress;
    }
}
```

# The Controller Pattern

- The Controller is responsible for handling the requests of actors. The Controller is the middle-man between your user clicking "Send" and your back-end making that happen.

- A Controller is a non-user interface object responsible for receiving or handling a system event. A Controller defines the method for the system operation.

# Key Points

- When a request comes from the UI layer object, Controller pattern helps UI layer to determine what is that first object that receives the message from the UI layer objects.
- This object is called a controller object which receives a request from the UI layer object and then controls/coordinates with other objects of the domain layer to fulfill the request.
- It delegates the work to other class and coordinates the overall activity.

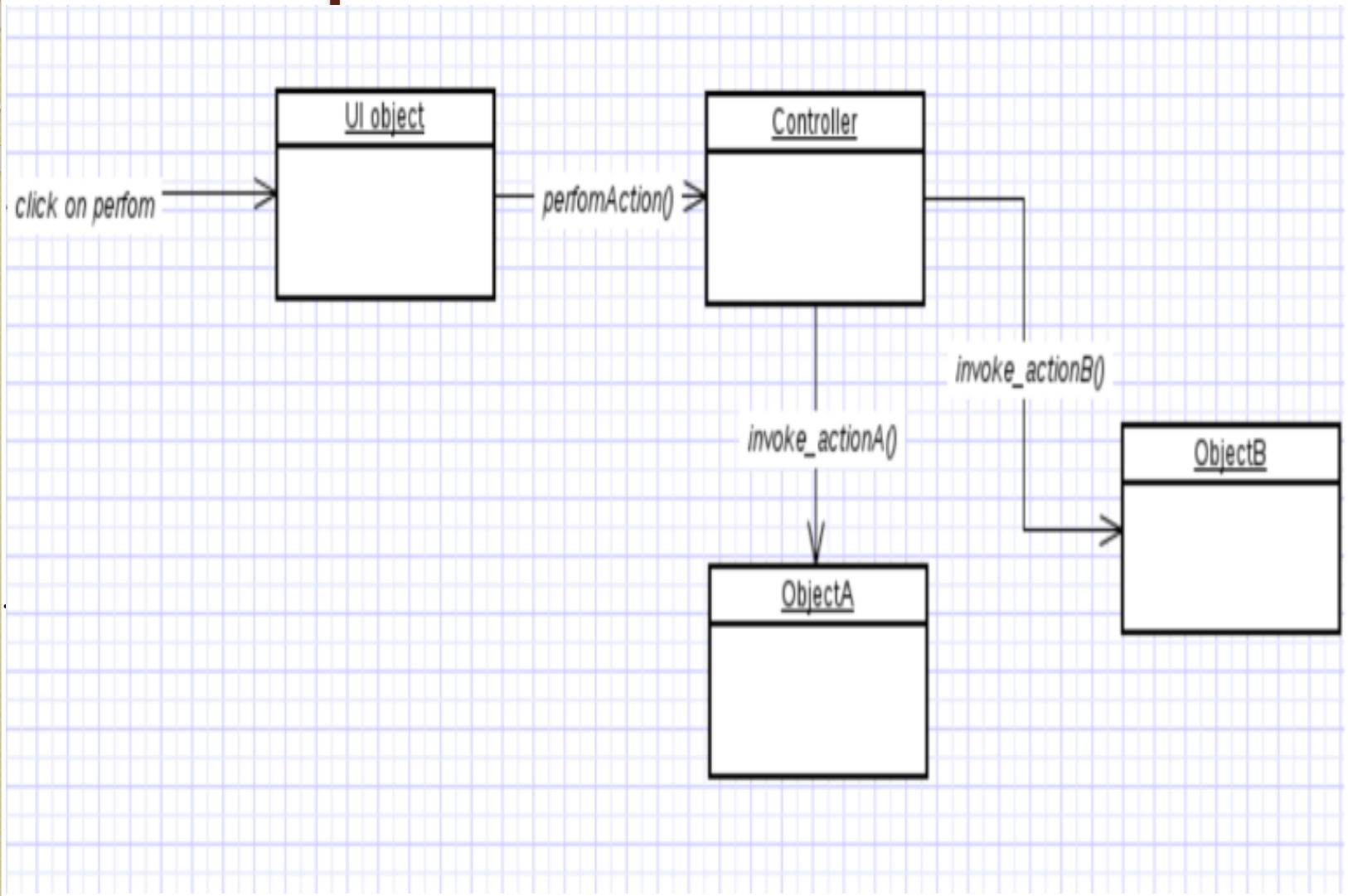
# Problem of Controller

- Who should be responsible for handling an input system event?
- An input system event is an event generated by an external actor. They are associated with system operations of the system in response to system events, just as messages and methods are related.
- Example when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check.“

# Solution

- Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:
  - Represents the overall system, device, or subsystem (facade controller).
  - Represents a use case scenario within which the system event occurs, often named Handler, Coordinator, or Session (use-case or session controller).
  - Use the same controller class for all system events in the same use case scenario.

# Example of Controller



# Benefits

- Either the UI classes or the problem/software domain classes can change without affecting the other side.
  - The controller is a simple class that mediates between the UI and problem domain classes, just forwards
  - Event handling requests
  - Output requests
- The Controller is also an important idiom in modern web development frameworks, in forming a pillar of the Model-View-Controller architectural pattern.

# Designing for Visibility

- A visibility is the ability of an object to see another object.
- Visibility can be determined if there is a message passing between sender and receiver.
- It is related to the issue of scope.
- To receive a message sender object must be visible to the receiver object.

.

# Types of Visibility

- Attribute Visibility
- Parameter Visibility
- Local Visibility
- Global Visibility

# Types of Visibility

- Attribute Visibility
  - Attribute visibility from A to B exist when B is an attribute of A. It is relatively permanent visibility because it persists as long as A and B exist. Let us consider the Register instance that may have attribute visibility to a prductcatalog.

```
Public Class Register{
Private ProductCatalog catalog;
Public void enterItem( Itemid, quantity){
catalog.getSpecification(ItemId)
}
```

# Types of Visibility

- Parameter Visibility
  - Attribute visibility from A to B exist when B is passed as parameter to a method of A. It is temporary because it is exist if method require parameter.

```
makeLineItem( ProductSpecification spec, int qty)
{
sl = new SalesLineItem(Spec,Qty)
}
```

# Types of Visibility

- Local Visibility
  - Local visibility from A to B exist when B is declared as a local object within a method a of A. It is temporary because it persist only within the scope of the method.

It can be achieved by two ways.

- Create a new local instance and assign it to a local variable
- Assign the returning object.

```
enterItem(itemId, quantity)
{
ProductSpecification spec = catalog.getSpecification(itemId)
}
```

# Types of Visibility

- Global Visibility
  - Global visibility from A to B exist when B is global to A. It is common for most of the object oriented programming language. It can be achieved by some programming language such as C++. ( But not by Java).

# GoF Pattern

- In 1994 Four authors Eric Gamma, Richard Helm, Ralph Johnson and John Vlissidees publish a book design patterns of reusable object oriented software which initiated the concept of design pattern in software development. These authors are collectively Gang of Four(GoF).

GoF Pattern Principles

- Program to an interface not an implementation
- Favor object composition over inheritance

# Singleton Design Pattern

- The singleton pattern is one of the simplest design patterns: it involves only one class which is responsible to make sure there is no more than one instance.
- It does it by instantiating itself and in the same time it provides a global point of access to that instance.
- By doing it, the singleton class ensures the same instance can be used from everywhere, preventing direct invocation of the singleton constructor.

# Implementation of Singleton

- The implementation involves a static member in the singleton class which keeps the reference to the instance.
- A private constructor and a static public method that returns the static member reference.
- The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is is responsible for creating its class unique instance in case it is not created yet and to return that instance.

# Code for Singleton

```
class Singleton {
private static Singleton instance;
private Singleton() {
...
}
public static synchronized Singleton getInstance(){
 if (instance == null)
instance = new Singleton();
return instance;
}
 ... public void doSomething()
{
 ...
 }
 }
```

# Adapter Design Pattern

- The adapter pattern is adapting between classes and objects.
- Like any adapter in the real world it is used to be an interface, a bridge between two objects.
- In real world we have adapters for power supplies, adapters for camera memory cards, and so on.
- Of course, you want to use both of them so you don't to implement again one of them, and you don't want to change existing classes, so why not create an adapter.
- It convert the instance of a class into another interface clients expect.

# Advantages and Disadvantages of Adapter

**Advantages**

- It allows two or more previously incompatible objects to interact.
- It allows reusability of existing functionality and flexibility.
- Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.

**Disadvantages**

- All requests are forwarded, so there is a slight increase in the overhead.
- Sometimes many adaptations are required along an adapter chain to reach the type which is required.

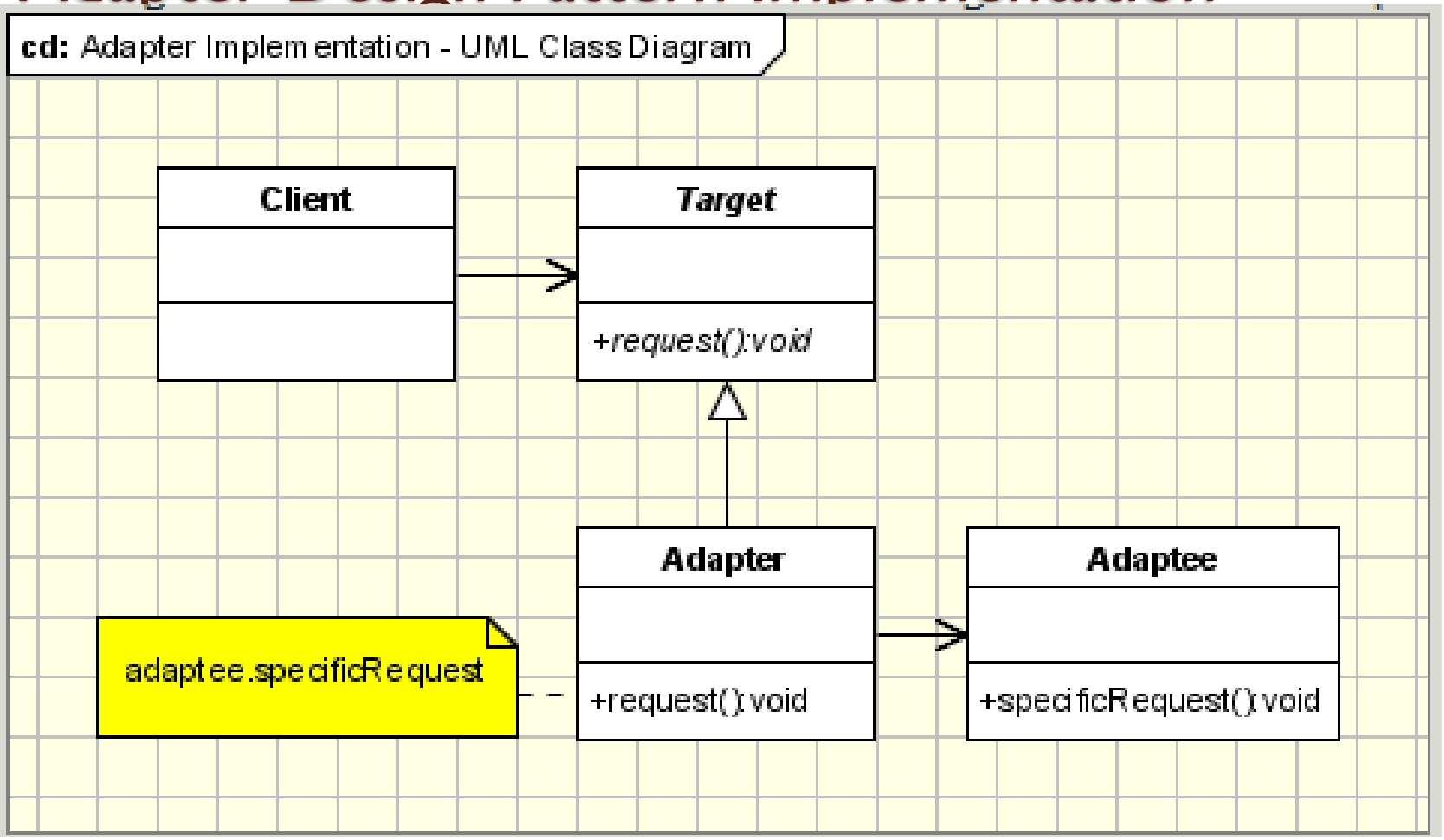
# When to use Adapter Pattern

- Legacy System Integration
- Library Compatibility
- Polymorphism and code reusability

# Real world Examples of Adapter Pattern

- **File Format Adapters:** An application that can read data from different file formats (e.g., CSV, XML, JSON). Each file format may have its own parsing interface. To maintain a consistent interface for reading data, adapters can be used to wrap the different parsing classes and provide a unified interface for this application.
- **Database Adapters:** When working with databases, different database vendors might provide their own specific APIs and interfaces. An adapter can be used to wrap these vendor-specific APIs and provide a standard database interface that an application can use regardless of the underlying database technology.
- **Third-Party API Integration:** When integrating with external services or APIs that have different interfaces, adapters can be used to convert the external API calls into a format that matches an application's interface. This allows seamless integration without changing the application's code.
- **Language Translation Services:** Language translation functionality is required for an application, and a third-party translation service that has its own API can be used. Adapters

# Adapter Design Pattern Implementation



# Factory Pattern

- It is most commonly used design pattern in Java and provide a best way to create an object.
- It can be used when we have superclass and its multiple subclass and based on the input we need to return.
- In this pattern, An object is created without exposing the creation logic to the client and refer to newly created object using a common interface.

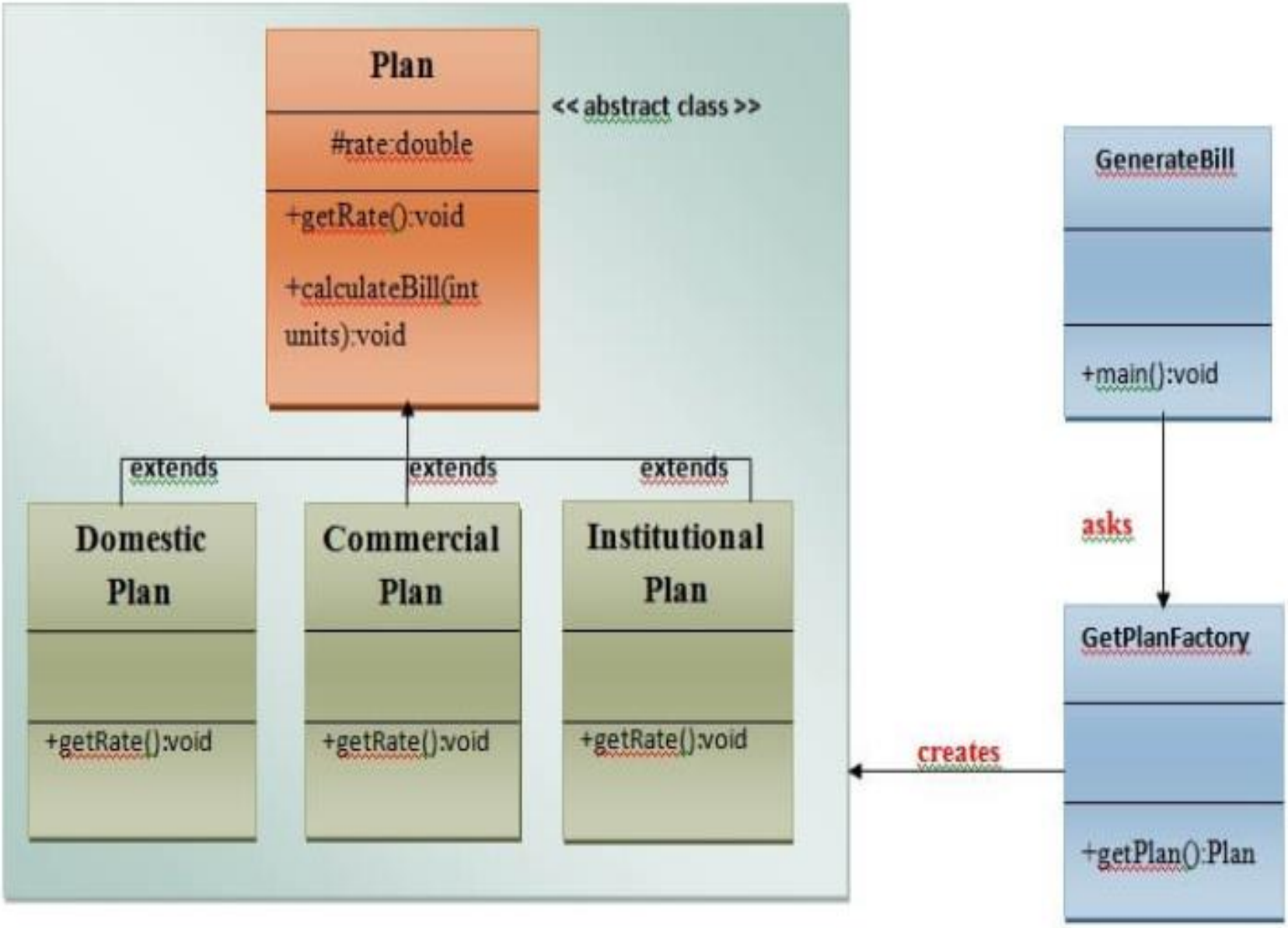
# Factory Pattern Example

- To create multiple currency e.g. INR, SGD, USD and code should be extensible to accommodate new Currency as well.
- Here we have made Currency as interface and all currency would be concrete implementation of Currency interface.
- Factory Class will create Currency based upon country and return concrete implementation which will be stored in interface type. This makes code dynamic and extensible.

## When to use Factory Method Design

- A class cannot predict the type of objects it needs to create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of multiple helper subclasses, and you aim to keep the information about which helper subclass is the delegate within a specific scope or location.

# Real example with class diagram



# Observer Pattern

- **Observer Pattern** is one of the **behavioral design pattern**.
- Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change.
- In observer pattern, the object that watch on the state of another object are called **Observer** and the object that is being watched is called **Subject**.

# Benefits and usage

- **Benefits**
  - It describes the coupling between the objects and the observer.
  - It provides the support for broadcast-type communication.
- **Usage**
  - When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
  - When the framework we writes and needs to be enhanced in future with new observers with minimal changes.