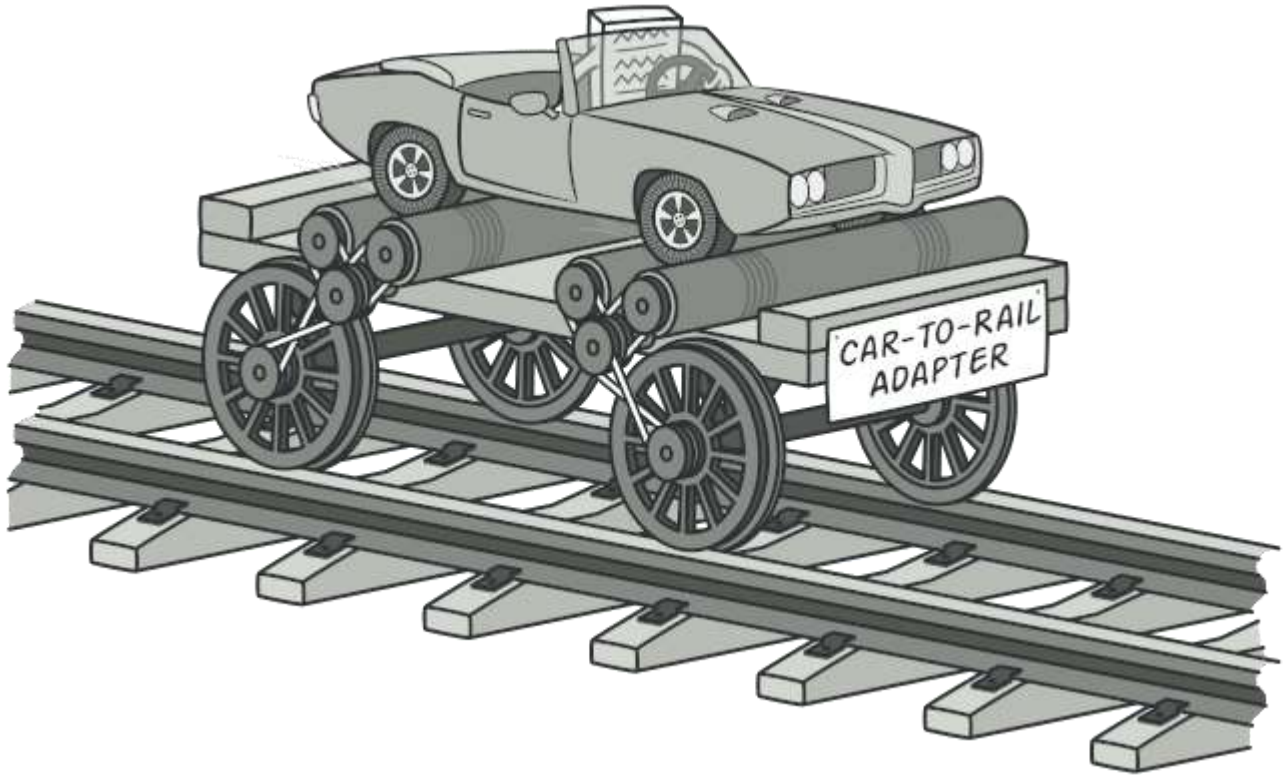
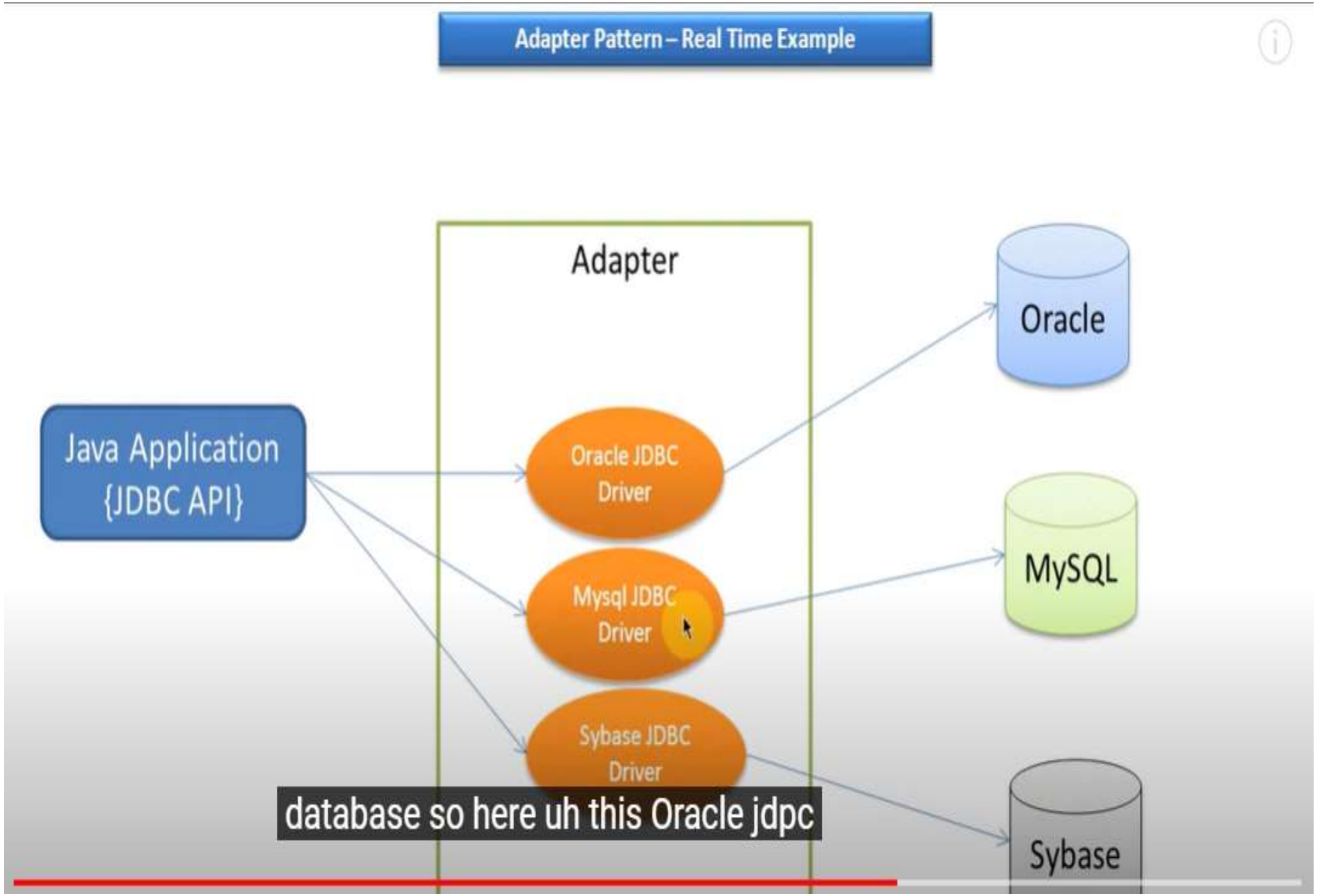
# Structural Design Patterns

- A structural design pattern deals with class and object composition, or how to assemble objects and classes into larger structures.
- **Adapter**: How to change or adapt an interface to that of another existing class to allow incompatible interfaces to work together.
- **Bridge**: A method to decouple an interface from its implementation.
- **Composite**: Leverages a tree structure to support manipulation as one object.
- **Decorator**: Dynamically extends (adds or overrides) functionality.
- **Façade**: Defines a high-level interface to simplify the use of a large body of code.
- **Flyweight**: Minimize memory use by sharing data with similar objects.
- **Proxy**: How to represent an object with another object to enable access control, reduce cost and reduce complexity.

# Adapter Design Pattern

- **Adapter** is a structural design pattern that allows objects with incompatible interfaces to collaborate.




Adapter Pattern – Real Time Example
Adapter
Java Application
{JDBC API}
Oracle JDBC
Driver
Mysql JDBC
Driver
Sybase JDBC
Driver
Oracle
MySQL
Sybase
database so here uh this Oracle jdpc

- The Adapter Design Pattern is a structural pattern that allows objects with incompatible interfaces to work together.
- It acts as a bridge between two incompatible interfaces by converting the interface of a class into another interface expected by the client.

**Key Components**

1. **Target:** Defines the domain-specific interface that the client uses.
2. **Adapter:** Implements the Target interface and adapts the interface of the Adaptee to the Target interface.
3. **Adaptee:** The existing interface that needs to be adapted to the Target interface.
4. **Client:** Interacts with the Target interface.

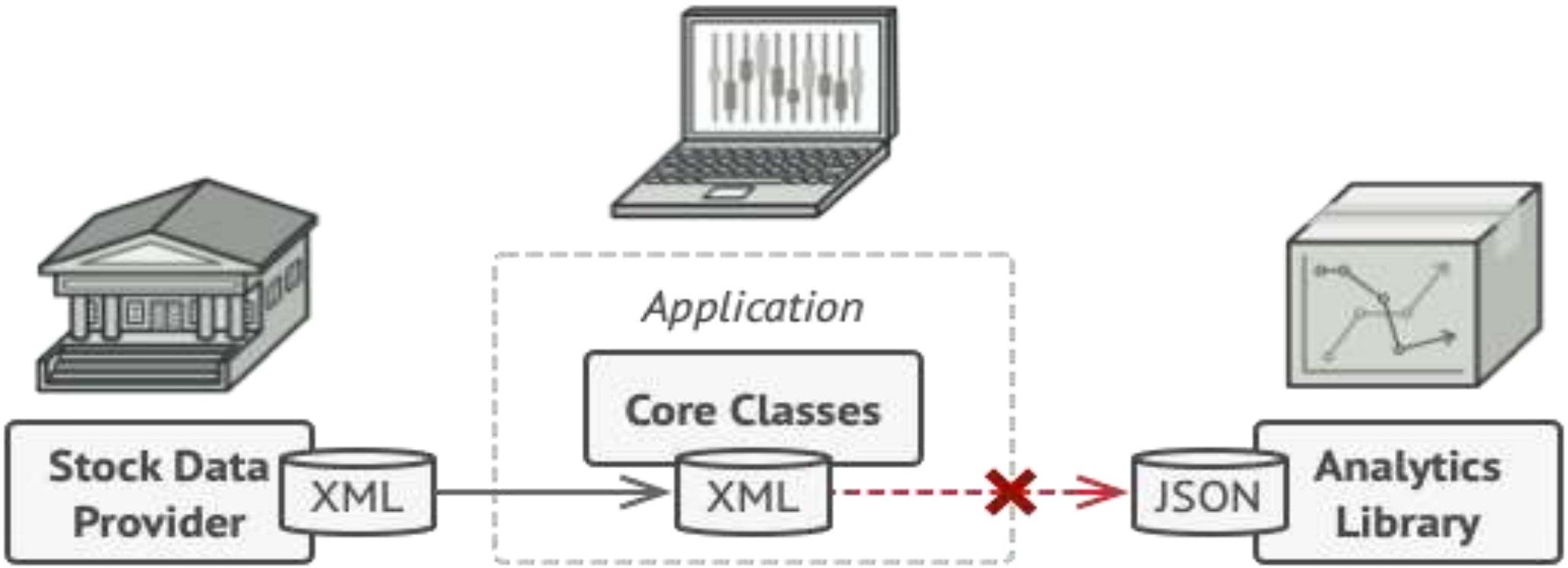
# When to use

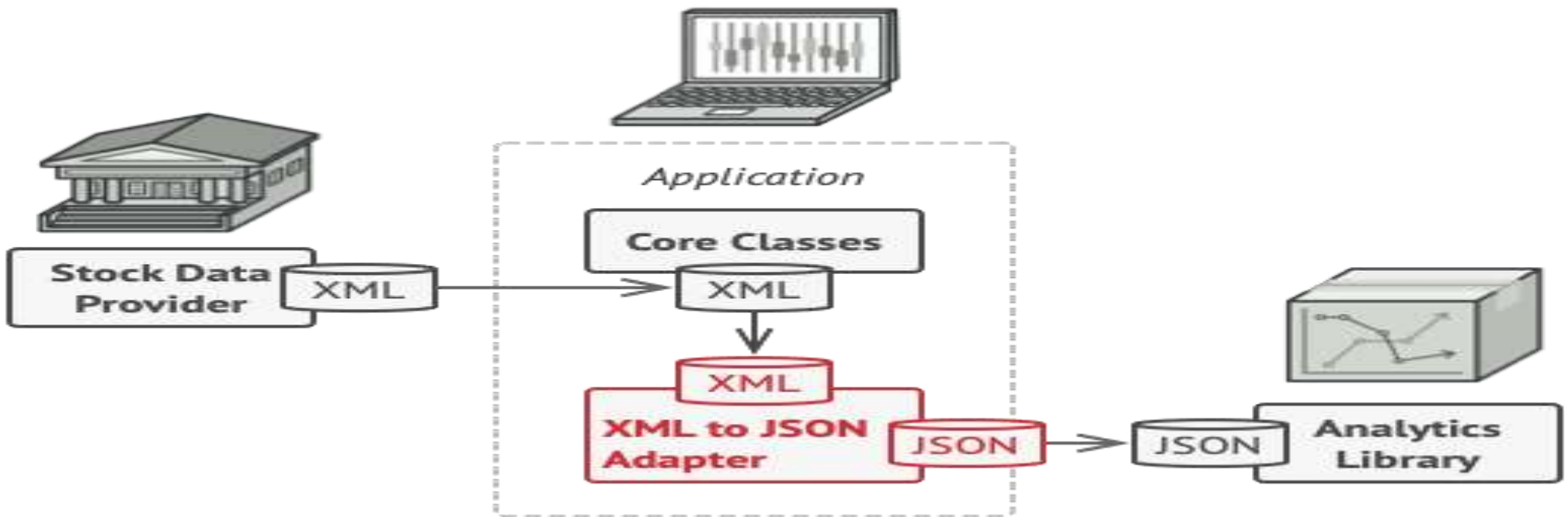
- When you want to use an existing class but its interface is incompatible with the code you’re working with.

- When you need to reuse several existing subclasses that lack some common functionality.

# Problem

- Imagine that you're creating a stock market monitoring app. The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.
- At some point, you decide to improve the app by integrating a smart 3rd-party analytics library. But there's a catch: the analytics library only works with data in JSON format.


Application
Core Classes
Stock Data Provider
XML
XML
JSON
Analytics Library


Application
Core Classes
XML
Stock Data Provider
XML
XML
XML to JSON Adapter
JSON
JSON
Analytics Library

# Solution

- You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.
- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.
- Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate. Here's how it works:

1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter's methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

- Sometimes it's even possible to create a two-way adapter that can convert the calls in both directions.

- Let’s get back to our stock market app. To solve the dilemma of incompatible formats, you can create XML-to-JSON adapters for every class of the analytics library that your code works with directly. Then you adjust your code to communicate with the library only via these adapters. When an adapter receives a call, it translates the incoming XML data into a JSON structure and passes the call to the appropriate methods of a wrapped analytics object.

# Advantages of Adapter Design Pattern

- Improves Reusability: Allows using existing incompatible classes by adapting them to a common interface.
- Enhances Flexibility: New adapters can be introduced without changing existing code.
- Single Responsibility Principle: Divides the interface adaptation work from the primary business logic.
- Promotes Legacy Code Integration: Allows integration of old components with new systems without modifications.

# Disadvantages of Adapter Design Pattern

- Increased Complexity: Introducing an adapter adds extra layers, which can make the design more complex.
- Limited Functionality: If the Adaptee interface is too different, the adapter may only provide partial functionality.
- Overuse Risk: Excessive use of adapters can lead to cluttered code and maintenance difficulties.
- Performance Overhead: Can introduce slight overhead due to additional interface conversions.

# Scenario: Integrating Electric Car Charging System with a Gasoline Car Interface

- Imagine a system where we have a GasolineCar interface used throughout an existing application.
- Now, we want to integrate a new ElectricCar system which has a different interface.
- We will use the Adapter Pattern to make the ElectricCar compatible with the GasolineCar interface.

| Role | Class | Description |
| --- | --- | --- |
| Target | `GasolineCar` | What the client wants. |
| Adaptee | `ElectricCar` | Incompatible class. |
| Adapter | `ElectricCarAdapter` | Makes `ElectricCar` compatible. |
| Client | `AdapterPatternDemo` | Uses the system through `GasolineCar`. |

- 1. Target Interface (What the client expects)
- This is what the rest of the system or client knows and expects.
- Any car that implements GasolineCar must have a refuel() method.
- 2. Adaptee (Incompatible class)
- This class doesn't match the GasolineCar interface.
- It has a charge() method instead of refuel().
- Problem: You can't pass this directly where a GasolineCar is expected.

- Adapter Class (The bridge)
- ElectricCarAdapter implements GasolineCar, so it can be used anywhere a GasolineCar is expected.
- It holds an instance of ElectricCar.
- Inside refuel(), it delegates to electricCar.charge().
- This makes ElectricCar look like a GasolineCar from the outside!
- 4. Client Code (Using the adapter)
- The client works with GasolineCar.
- It doesn’t need to know anything about ElectricCar or charge().
- The adapter handles the translation.

# Java Code Implementation

```
// Target Interface: The interface expected by the client.
interface GasolineCar {
    void refuel();
}

// Adaptee: The incompatible interface that needs to be adapted.
class ElectricCar {
    public void charge() {
        System.out.println("Electric car is charging...");
    }
}
```

```
// Adapter: Makes the ElectricCar compatible with GasolineCar interface.
class ElectricCarAdapter implements GasolineCar {
    private final ElectricCar electricCar;

    public ElectricCarAdapter(ElectricCar electricCar) {
        this.electricCar = electricCar;
    }

    @Override
    public void refuel() {
        electricCar.charge(); // Delegating the call to the adaptee's method
    }
}
```

```
// Client: Uses the Target interface (GasolineCar).
public class AdapterPatternDemo {
    public static void main(String[] args) {
        GasolineCar myElectricCar = new ElectricCarAdapter(new ElectricCar());

        System.out.println("Using ElectricCar through GasolineCar interface:");
        myElectricCar.refuel(); // Outputs: Electric car is charging...
    }
}
```

## Explanation

- Target Interface (GasolineCar):
- Defines a common interface refuel() which the client expects.

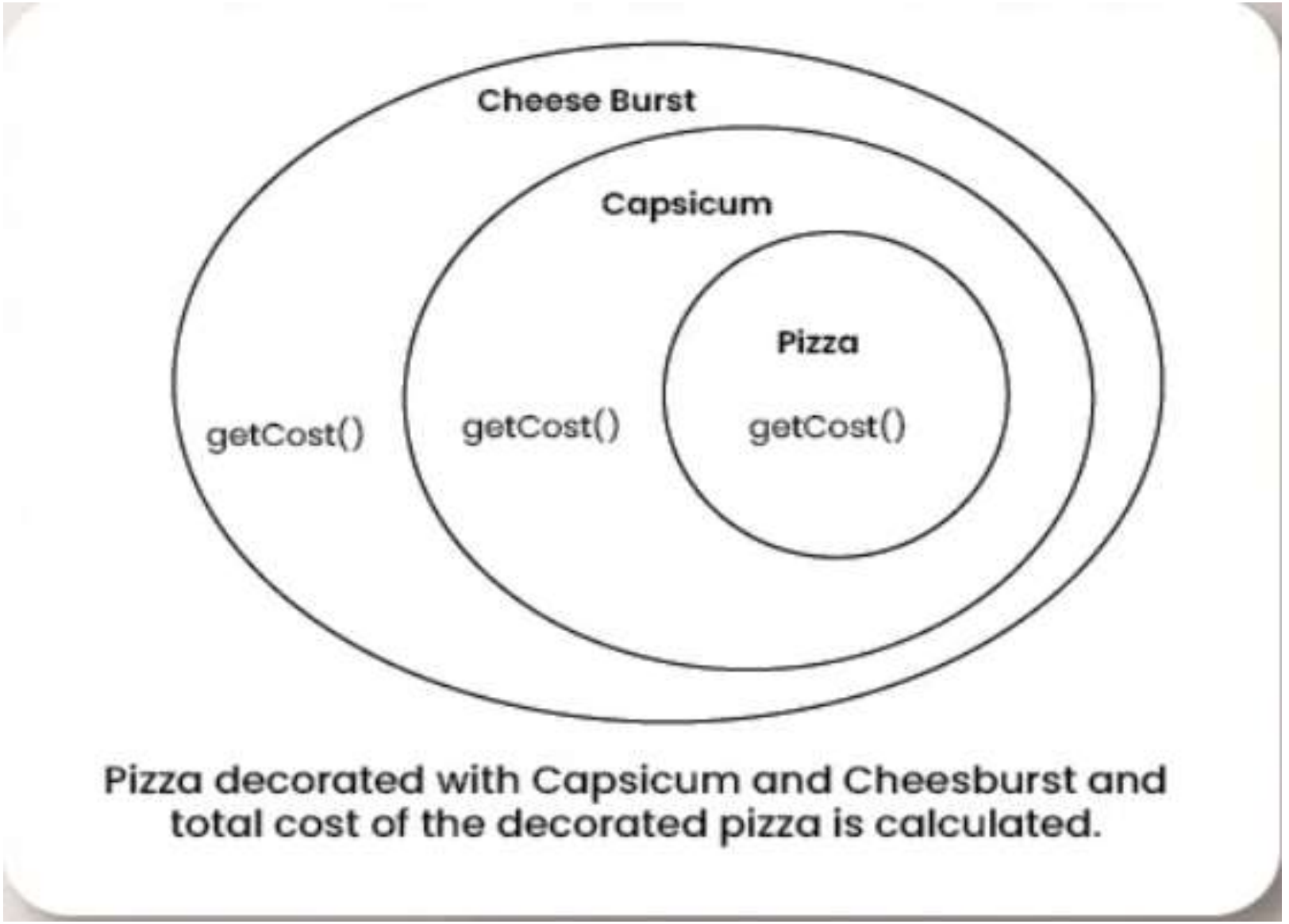
- Adaptee (ElectricCar):
- Has a different method charge() which is incompatible with the existing GasolineCar interface.

- Adapter (ElectricCarAdapter):
- Implements the GasolineCar interface and holds a reference to an ElectricCar object.
- Converts the refuel() call to charge() for compatibility.

- Client (AdapterPatternDemo):
- Uses the GasolineCar interface to interact with the ElectricCar through the adapter.

# Decorator Design Pattern

- The Decorator Design Pattern is a structural design pattern used in software development. It allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class.
- This pattern is useful when you need to add functionality to objects in a flexible and reusable way.




Pizza decorated with Capsicum and Cheesburst and total cost of the decorated pizza is calculated.

- Characteristics of the Decorator Pattern
- This pattern promotes flexibility and extensibility in software systems by allowing developers to compose objects with different combinations of functionalities at runtime.
- It follows the open/closed principle, as new decorators can be added without modifying existing code, making it a powerful tool for building modular and customizable software components.
- The Decorator Pattern is commonly used in scenarios where a variety of optional features or behaviors need to be added to objects in a flexible and reusable manner, such as in text formatting, graphical user interfaces, or customization of products like coffee or ice cream.

# Use Cases for the Decorator Pattern

- Extending Functionality: When you have a base component with basic functionality, but you need to add additional features or behaviors to it dynamically without altering its structure. Decorators allow you to add new responsibilities to objects at runtime.
- Multiple Combinations of Features: When you want to provide multiple combinations of features or options to an object. Decorators can be stacked and combined in different ways to create customized variations of objects, providing flexibility to users.
- Legacy Code Integration: When working with legacy code or third-party libraries where modifying the existing codebase is not feasible or desirable, decorators can be used to extend the functionality of existing objects without altering their implementation.
- GUI Components: In graphical user interface (GUI) development, decorators can be used to add additional visual effects, such as borders, shadows, or animations, to GUI components like buttons, panels, or windows.
- Input/Output Streams: Decorators are commonly used in input/output stream classes in languages like Java. They allow you to wrap streams with additional functionality such as buffering, compression, encryption, or logging without modifying the original stream classes.

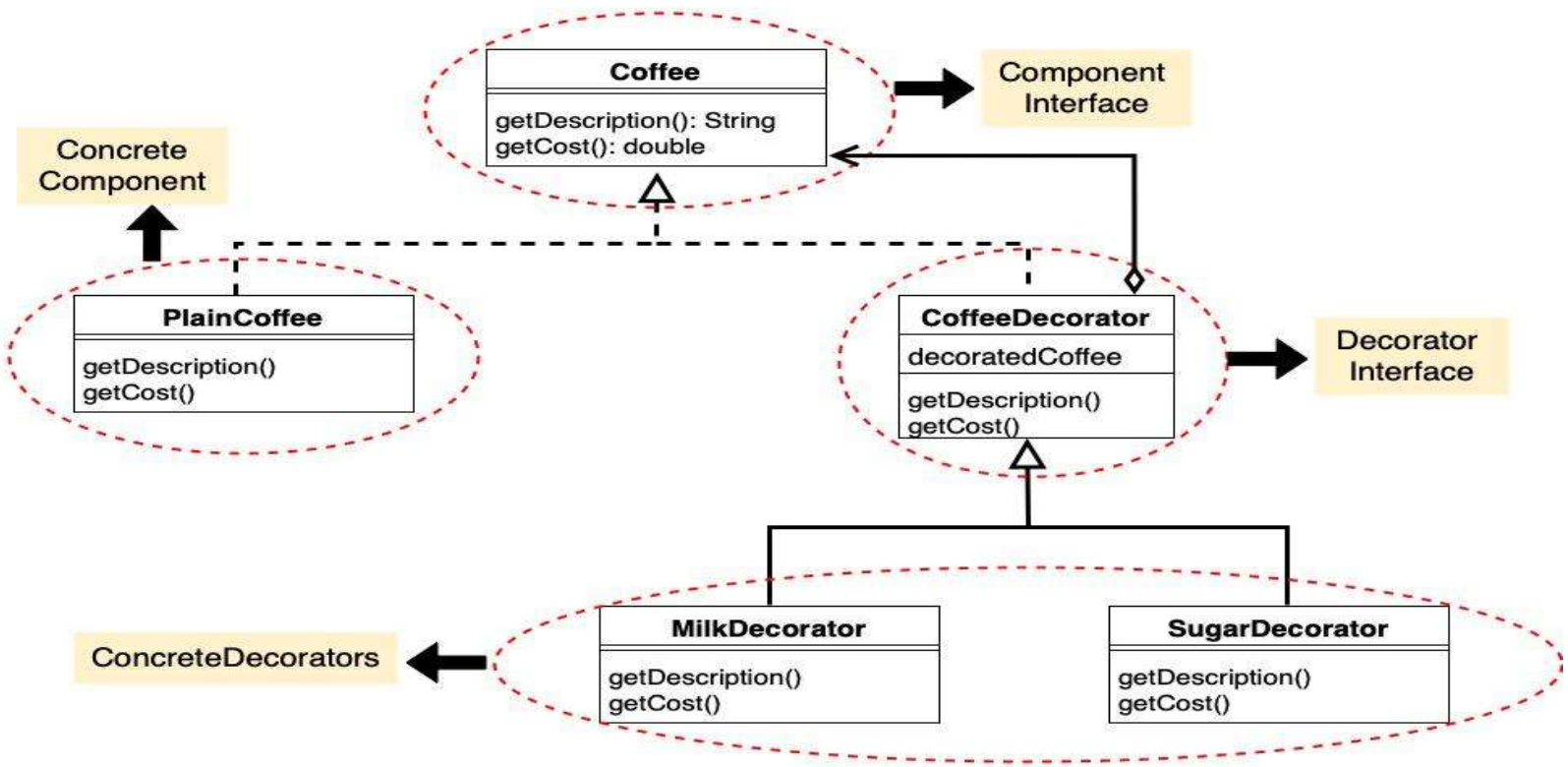
## Key Components of the Decorator Design Pattern

- **Component Interface:** This is an abstract class or interface that defines the common interface for both the concrete components and decorators. It specifies the operations that can be performed on the objects.
- **Concrete Component:** These are the basic objects or classes that implement the Component interface. They are the objects to which we want to add new behavior or responsibilities.
- **Decorator**: This is an abstract class that also implements the Component interface and has a reference to a Component object. Decorators are responsible for adding new behaviors to the wrapped Component object.
- **Concrete Decorator**: These are the concrete classes that extend the Decorator class. They add specific behaviors or responsibilities to the Component. Each Concrete Decorator can add one or more behaviors to the Component.

# Example of Decorator Design Pattern

- *Suppose we are building a coffee shop application where customers can order different types of coffee. Each coffee can have various optional add-ons such as milk, sugar, whipped cream, etc. We want to implement a system where we can dynamically add these add-ons to a coffee order without modifying the coffee classes themselves.*

# Class Diagram of Decorator Design Pattern



- 1. Component Interface(Coffee)
- This is the interface Coffee representing the component.
- It declares two methods getDescription() and getCost() which must be implemented by concrete components and decorators.

- // Coffee.java
- public interface Coffee {
-  String getDescription();
-  double getCost();
- }

- 2. ConcreteComponent(PlainCoffee)
- PlainCoffee is a concrete class implementing the Coffee interface.
- It provides the description and cost of plain coffee by implementing the getDescription() and getCost() methods.

```
// PlainCoffee.java
public class PlainCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Plain Coffee";
    }

    @Override
    public double getCost() {
        return 2.0;
    }
}
```

- 3. Decorator(CoffeeDecorator)
- CoffeeDecorator is an abstract class implementing the Coffee interface.
- It maintains a reference to the decorated Coffee object.
- The getDescription() and getCost() methods are implemented to delegate to the decorated coffee object.

```
// CoffeeDecorator.java
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}
```

- 4. ConcreteDecorators(MilkDecorator,SugarDecorator)
- MilkDecorator and SugarDecorator are concrete decorators extending CoffeeDecorator.
- They override getDescription() to add the respective decorator description to the decorated coffee’s description.
- They override getCost() to add the cost of the respective decorator to the decorated coffee’s cost.

```java
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}

// SugarDecorator.java
public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.2;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        // Plain Coffee
        Coffee coffee = new PlainCoffee();
        System.out.println("Description: " + coffee.getDescription());
        System.out.println("Cost: $" + coffee.getCost());


```

- // Coffee with Milk
- Coffee milkCoffee = new MilkDecorator(new PlainCoffee());
- System.out.println("\nDescription: " +
- milkCoffee.getDescription());
- System.out.println("Cost: $" + milkCoffee.getCost());

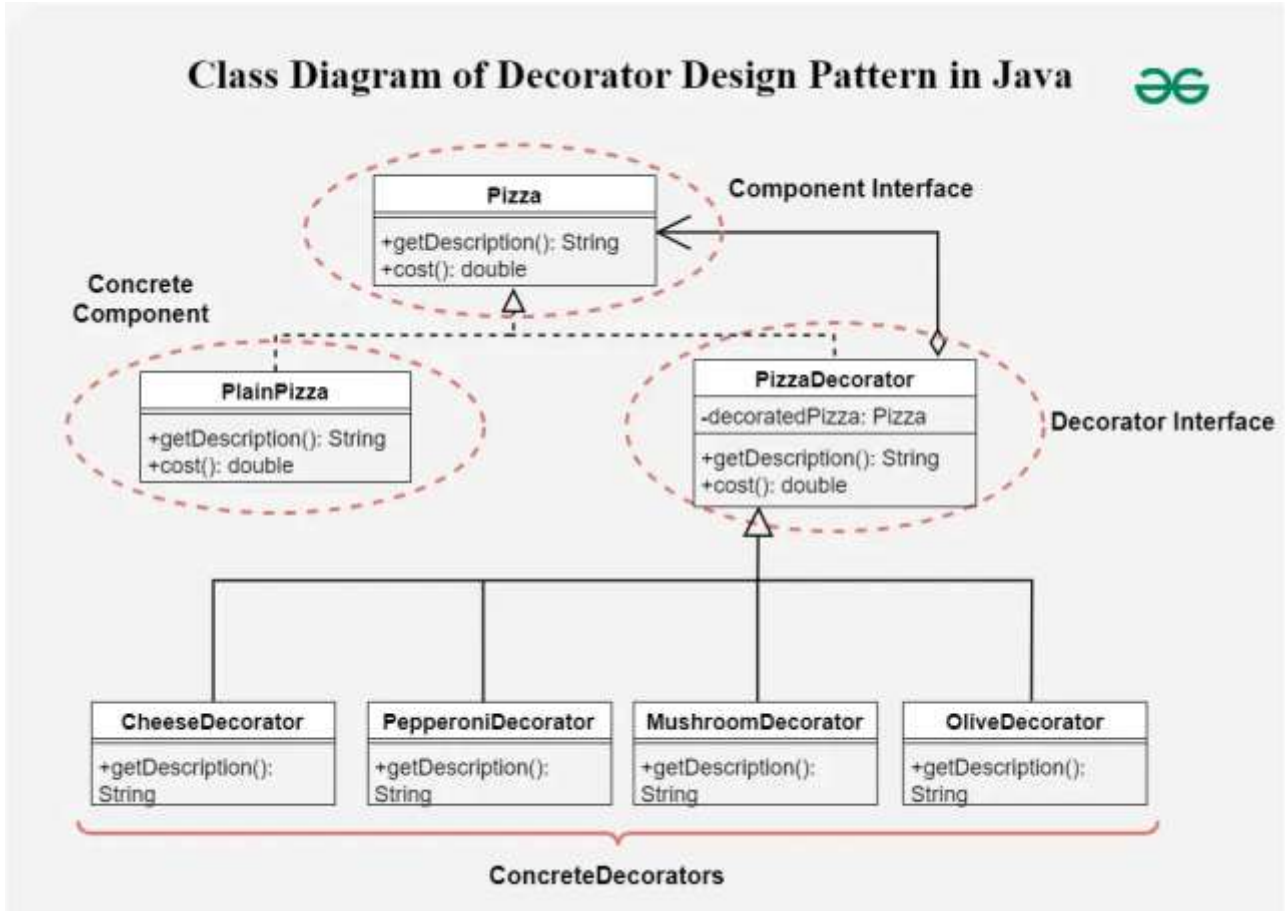
- A MilkDecorator is wrapping a PlainCoffee.
- Now we’ve decorated the plain coffee with milk.
- This adds to both description and cost.

- System.out.println("\nDescription: " + milkCoffee.getDescription());

- Calls MilkDecorator.getDescription(), which adds "Milk" to "Plain Coffee" → "Plain Coffee, Milk".

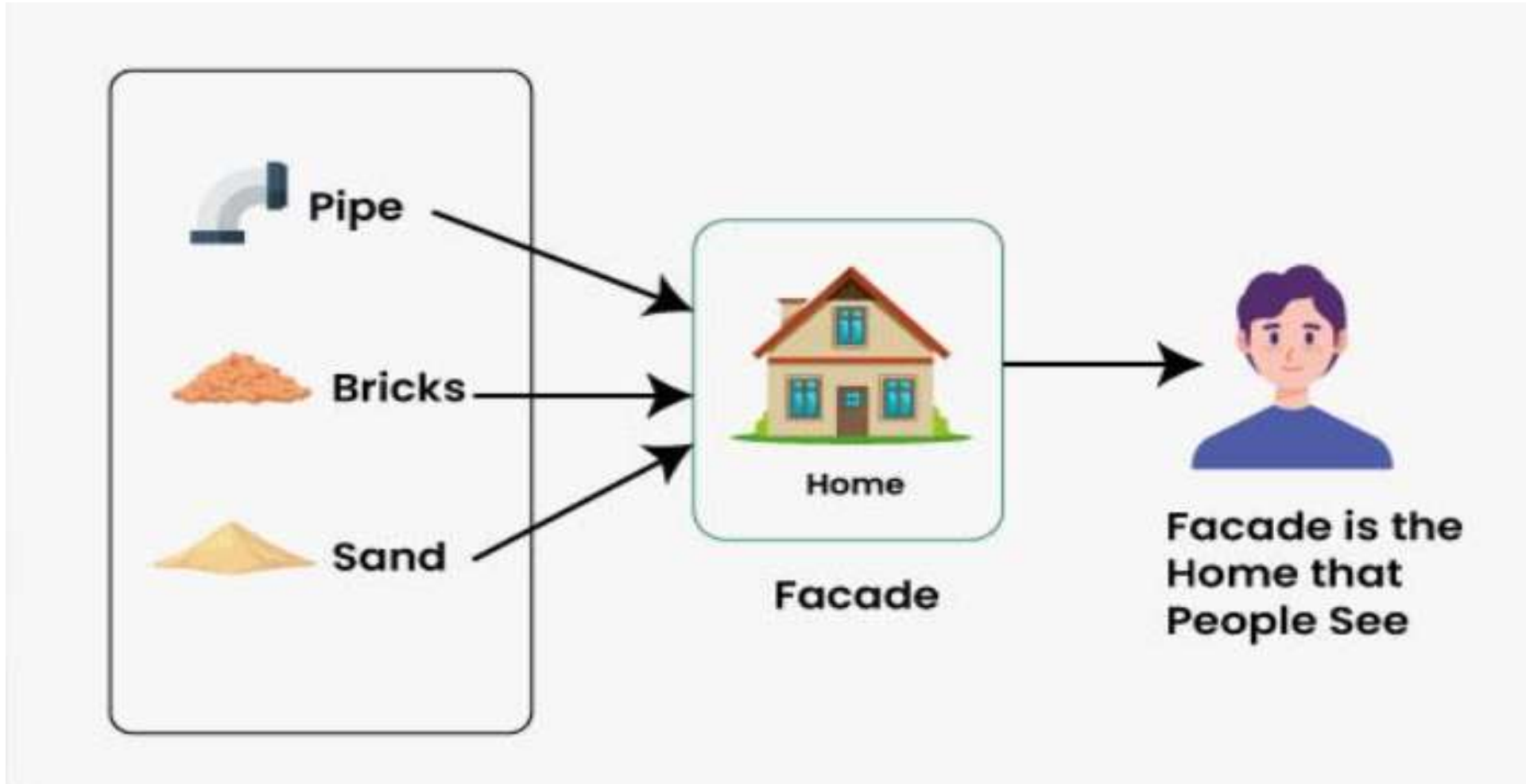
- // Coffee with Sugar and Milk
- //This wraps the PlainCoffee first with MilkDecorator then wraps the result with SugarDecorator.

- Coffee sugarMilkCoffee = new SugarDecorator(new MilkDecorator(new PlainCoffee()));
- System.out.println("\nDescription: " + sugarMilkCoffee.getDescription());
- System.out.println("Cost: $" + sugarMilkCoffee.getCost());
- }
- }


Class Diagram of Decorator Design Pattern in Java
Pizza
+getDescription(): String
+cost(): double
Component Interface
Concrete Component
PlainPizza
+getDescription(): String
+cost(): double
PizzaDecorator
-decoratedPizza: Pizza
+getDescription(): String
+cost(): double
Decorator Interface
CheeseDecorator
+getDescription(): String
PepperoniDecorator
+getDescription(): String
MushroomDecorator
+getDescription(): String
OliveDecorator
+getDescription(): String
ConcreteDecorators

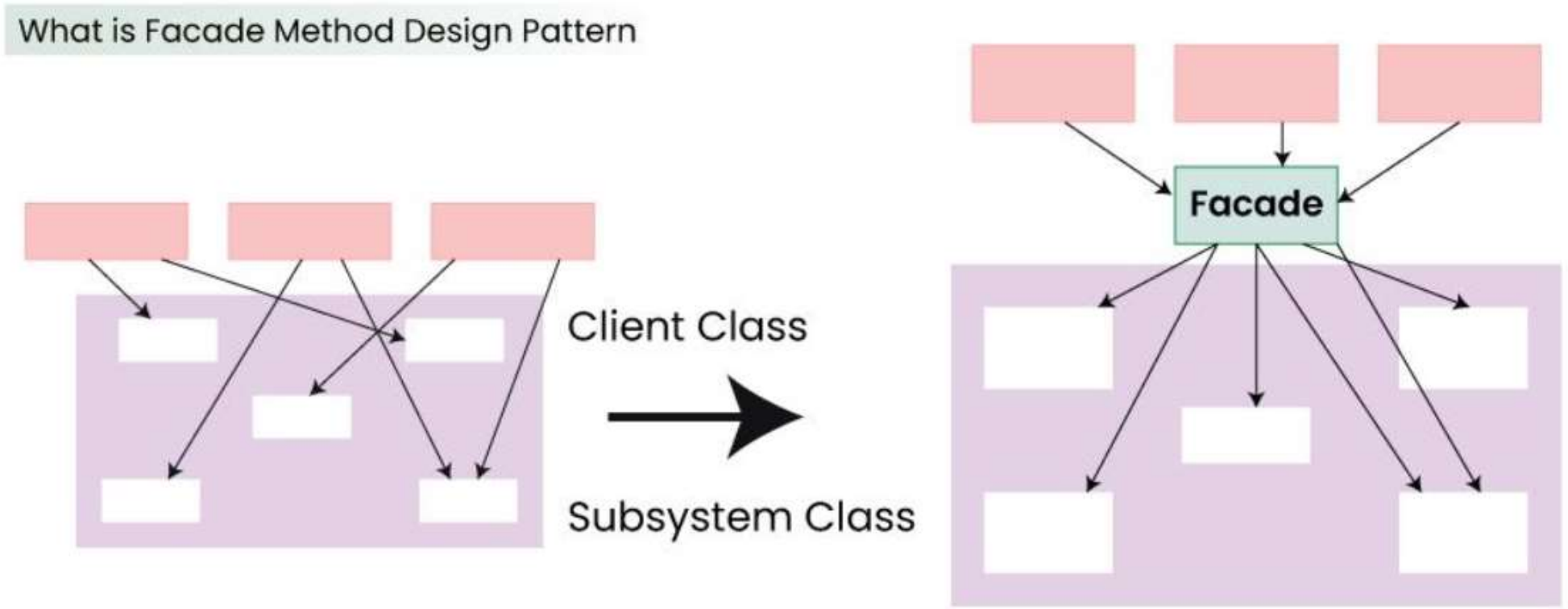
- Component Interface (Pizza):
- Defines the methods getDescription() and cost() that all concrete pizza and decorators will implement.
- Concrete Component (PlainPizza):
- Implements the Pizza interface and represents a basic pizza with a description and cost.
- Abstract Decorator (PizzaDecorator):
- Implements the Pizza interface and contains a reference to a Pizza object. This class forwards method calls to the wrapped pizza object and can be extended by concrete decorators.
- Concrete Decorators (CheeseDecorator, PepperoniDecorator, MushroomDecorator, OliveDecorator):
- Extend PizzaDecorator and add specific toppings. Each decorator modifies the description and cost of the pizza by adding the new topping.
- Client Code (PizzaShop):
- Demonstrates how to create a PlainPizza and then decorate it with various toppings. Each topping is added dynamically, and the description and cost are updated accordingly.

# Facade Method Design Pattern


Pipe
Bricks
Sand
Home
Facade
Facade is the Home that People See

- Facade Method Design Pattern is a part of the Gang of Four design patterns and it is categorized under Structural design patterns.
- Imagine a building, the facade is the outer wall that people see, but behind it is a complex network of wires, pipes, and other systems that make the building function.
- The facade pattern is like that outer wall. It hides the complexity of the underlying system and provides a simple interface that clients can use to interact with the system.
- It acts as a "**front door**." concealing the internal complexity of the subsystem and making it easier for clients to interact with it.

***Facade Method Design Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a high-level interface that makes the subsystem easier to use.***



- Structuring a system into subsystems helps reduce complexity.
- A common design goal is to minimize the communication and dependencies between subsystems.
- One way to achieve this goal is to introduce a Facade object that provides a single simplified interface to the more general facilities of a subsystem.

- **When to use Facade Method Design Pattern**
- A Facade provide a simple default view of the subsystem that is good enough for most clients.
- There are many dependencies between clients and the implementation classes of an abstraction.
- A Facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- Facade define an entry point to each subsystem level. If subsystem are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

# Key Component of Facade Method Design Pattern

- **Facade**
- Facade knows which subsystem classes are responsible for a request.
- It delegate client requests to appropriate subsystem objects.

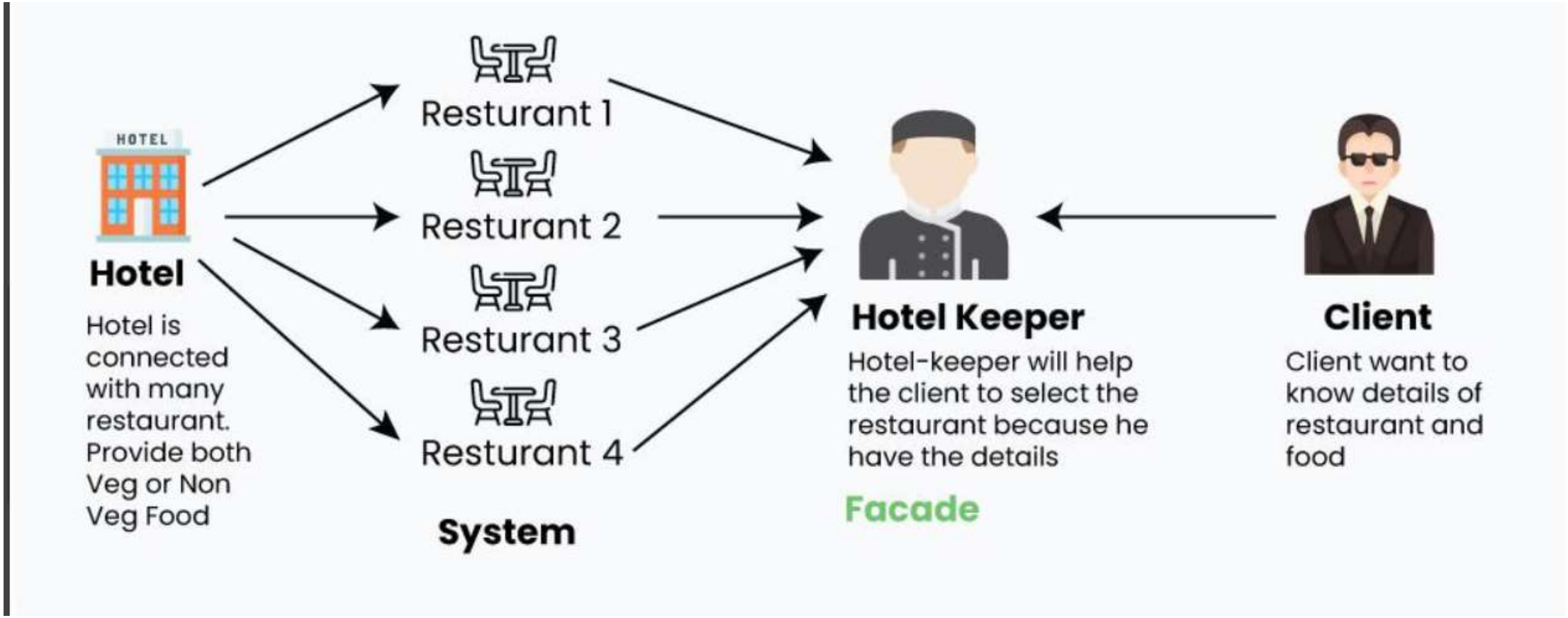
- **Subsystem classes**
- It implement subsystem functionality.
- It handle work assigned by the Facade object.
- It have no knowledge of the facade; that is, they keep no references to it.

- **Facade Method Design Pattern collaborate in different way**
- Client communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem objects.
- The Facade may have to do work of its own to translate it inheritance to subsystem interface.
- Clients that use the Facade don’t have to access its subsystem objects directly.

# Problem Statement for the Facade Method Design Pattern

- *Let's consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside the hotel e.g. Veg restaurants, Non-Veg restaurants, and Veg/Non Both restaurants. You, as a client want access to different menus of different restaurants. You do not know what are the different menus they have. You just have access to a hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of the respective restaurants and hands it over to you.*

# Problem Statement for the Facade Method Design Pattern




**Hotel-Keeper** is Facade and respective **Restaurants** is system.

**Step wise Step Implementation of above problem**

- **Interface of Hotel**
- The hotel interface only returns Menus. Similarly, the Restaurant are of three types and can implement the hotel interface

- package structural.facade;
- public interface Hotel
- {
- public Menus getMenus();
- }

- **NonVegRestaurant.java**

```
package structural.facade;
public class NonVegRestaurant implements Hotel {

        public Menus getMenus()
        {
                NonVegMenu nv = new NonVegMenu();
                return nv;
        }
}
```

- **VegRestaurant.java**

```
package structural.facade;
public class VegRestaurant implements Hotel {

        public Menus getMenus()
        {
                VegMenu v = new VegMenu();
                return v;
        }
}
```

- **VegNonBothRestaurant.java**
- package structural.facade;
- public class VegNonBothRestaurant implements Hotel {

- 	public Menus getMenus()
- 	{
- 		Both b = new Both();
- 		return b;
- 	}
- }

- **HotelKeeper.java** (façade)

- package structural.facade;

- public interface HotelKeeper
- {
- public VegMenu getVegMenu();
- public NonVegMenu getNonVegMenu();
- public Both getVegNonMenu();
- }

- **HotelKeeperImplementation.java**

```
package structural.facade;

public class HotelKeeperImplementation implements HotelKeeper {

    public VegMenu getVegMenu()
    {
        VegRestaurant v = new VegRestaurant();
        VegMenu vegMenu = (VegMenu)v.getMenus();
        return vegMenu;
    }

    public NonVegMenu getNonVegMenu()
    {
        NonVegRestaurant v = new NonVegRestaurant();
        NonVegMenu NonvegMenu = (NonVegMenu)v.getMenus();
        return NonvegMenu;
    }

    public Both getVegNonMenu()
    {
        VegNonBothRestaurant v = new VegNonBothRestaurant();
        Both bothMenu = (Both)v.getMenus();
        return bothMenu;
    }
```

- The complex implementation will be done by HotelKeeper himself. The client will just access the HotelKeeper and ask for either Veg, NonVeg or VegNon Both Restaurant menu.

- **How will the client program access this façade?**

```
package structural.facade;

public class Client
{
    public static void main (String[] args)
    {
        HotelKeeper keeper = new HotelKeeperImplementation();

        VegMenu v = keeper.getVegMenu();
        NonVegMenu nv = keeper.getNonVegMenu();
        Both = keeper.getVegNonMenu();

    }
}
```

- In this way, the implementation is sent to the facade. The client is given just one interface and can access only that. This hides all the complexities.

# Use Cases of Facade Method Design Pattern

- **Simplifying Complex External Systems:**
  - A facade encapsulates database connection, query execution, and result processing, offering a clean interface to the application.
  - A facade simplifies the usage of external APIs by hiding the complexities of authentication, request formatting, and response parsing.
  - A facade can create a more user-friendly interface for complex or poorly documented libraries.
- **Layering Subsystems:**
  - **Decoupling subsystems:** Facades define clear boundaries between subsystems, reducing dependencies and promoting modularity.
  - **Providing high-level views:** Facades offer simplified interfaces to lower-level subsystems, making them easier to understand and use.
- **Providing a Unified Interface to Diverse Systems:**
  - **Integrating multiple APIs:** A facade can combine multiple APIs into a single interface, streamlining interactions and reducing code duplication.
  - **Bridging legacy systems:** A facade can create a modern interface for older, less accessible systems, facilitating their integration with newer components.

- **Protecting Clients from Unstable Systems:**
  - **Isolating clients from changes:** Facades minimize the impact of changes to underlying systems by maintaining a stable interface.
  - **Managing third-party dependencies:** Facades can protect clients from changes or issues in external libraries or services.

- **Advantages of Facade Method Design Pattern**
- **Simplified Interface:**
  - Provides a clear and concise interface to a complex system, making it easier to understand and use.
  - Hides the internal details and intricacies of the system, reducing cognitive load for clients.
  - Promotes better code readability and maintainability.
- **Reduced Coupling:**
  - Decouples clients from the underlying system, making them less dependent on its internal structure.
  - Promotes modularity and reusability of code components.
  - Facilitates independent development and testing of different parts of the system.
- **Encapsulation:**
  - Encapsulates the complex interactions within a subsystem, protecting clients from changes in its implementation.
  - Allows for changes to the subsystem without affecting clients, as long as the facade interface remains stable.
- **Improved Maintainability:**
  - Easier to change or extend the underlying system without affecting clients, as long as the facade interface remains consistent.
  - Allows for refactoring and optimization of the subsystem without impacting client code.

- **Disadvantages of Facade Method Design Pattern**
- **Increased Complexity:**
  - Introducing a facade layer adds an extra abstraction level, potentially increasing the overall complexity of the system.
  - This can make the code harder to understand and debug, especially for developers unfamiliar with the pattern.
- **Reduced Flexibility:**
  - The facade acts as a single point of access to the underlying system.
  - This can limit the flexibility for clients who need to bypass the facade or access specific functionalities hidden within the subsystem.
- **Overengineering:**
  - Applying the facade pattern to very simple systems can be overkill, adding unnecessary complexity where it's not needed.
  - Consider the cost-benefit trade-off before implementing a facade for every situation.
- **Potential Performance Overhead:**
  - Adding an extra layer of indirection through the facade can introduce a slight performance overhead, especially for frequently used operations.
  - This may not be significant for most applications, but it's worth considering in performance-critical scenarios.

- The facade pattern is appropriate when you have a **complex system** that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system that is incompatible with the system. Facade deals with interfaces, not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.