SLIIT UNI

THE KNOWLEDGE UNIVERSITY

Sri Lanka Institute of Information Technology
Faculty of Computing

Design Patterns-Part 02

Behavioral: Strategy Pattern

What is Strategy Pattern?

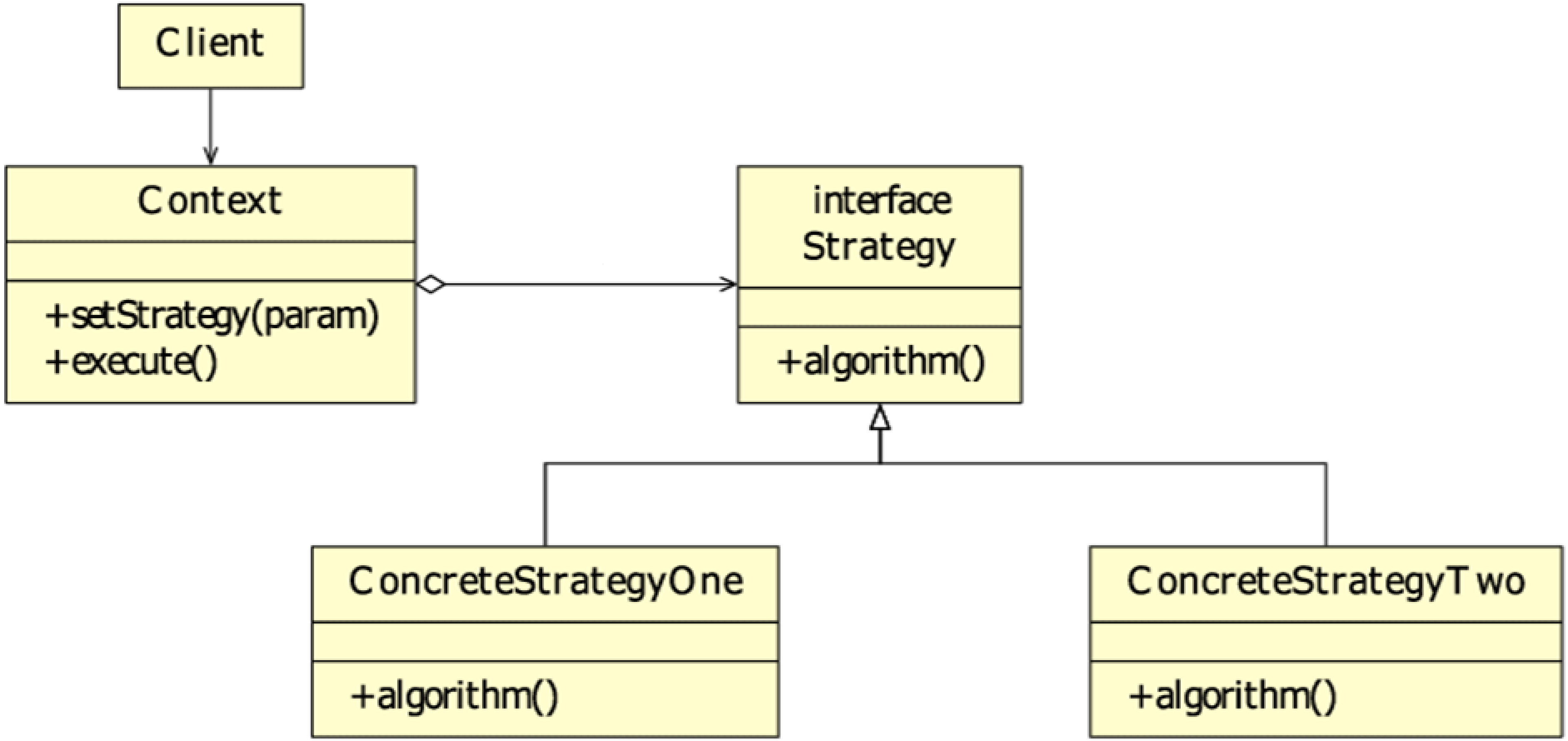
The **Strategy Design Pattern** is one of the **behavioral design patterns** in Object-Oriented Programming .

- Belongs to Behavioral Design Patterns.

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

- Allows algorithms to vary independently from the clients that use them.

Class Diagram of Strategy Design Pattern



Key Concepts of Strategy Pattern

1. Defines a Family of Algorithms

Example: Sorting a list (Bubble Sort, Quick Sort, Merge Sort).

2. Encapsulates Each One

Each algorithm implemented in its own class.

3. Makes Them Interchangeable

All follow a common interface, so they can be swapped at runtime.

Strategy lets the algorithm vary independently from clients that use it.

1. Defines a Family of Algorithms

A family of algorithms means several different ways to solve the same type of problem.

Example: Sorting a list. You could use:

Bubble Sort

Quick Sort

Merge Sort

All of these are different **algorithms** for the same task (sorting).

2. Encapsulates Each One

Encapsulation means wrapping each algorithm inside its own class so it is separate from others.

Instead of writing big if-else code, each sorting method is written in its own class.

Example:

BubbleSort class
QuickSort class
MergeSort class

3. Makes Them Interchangeable

- All follow a common interface so that they can be swapped at runtime.

- Each algorithm follows a common interface (e.g., Sort Strategy), so we can swap one with another at runtime without changing the main code.

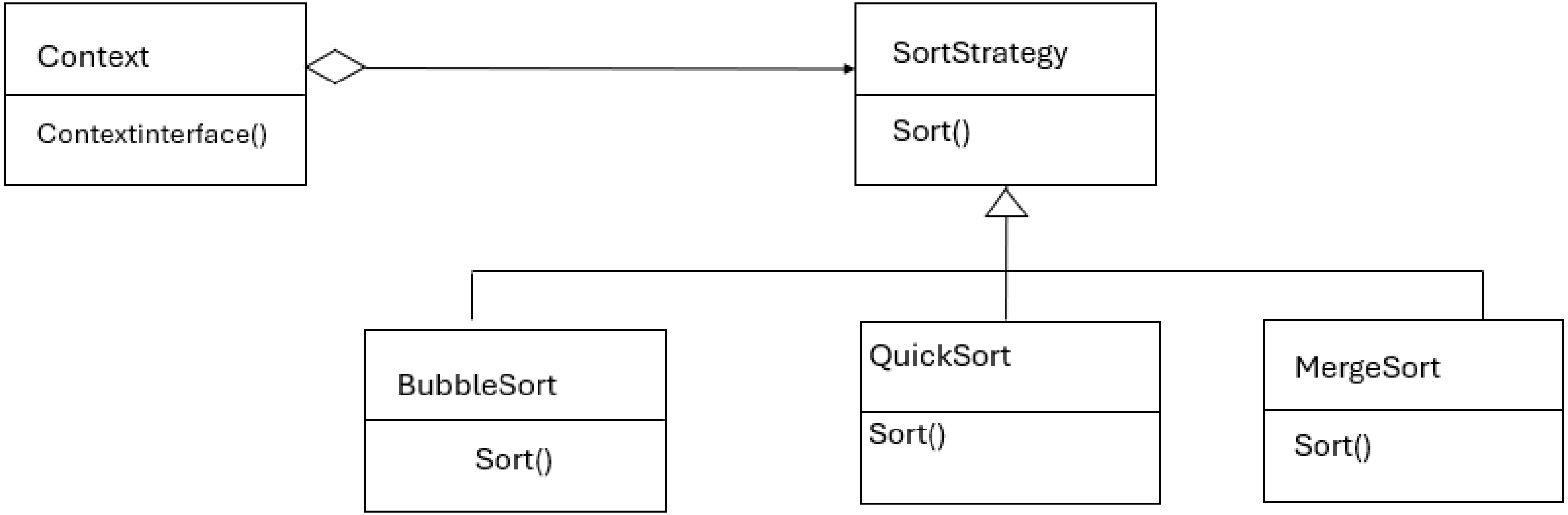
UML Class Diagram (Sorting Algorithms)

Classes:

Context → Uses a Strategy object.

Strategy Interface → Defines the method(s).

Concrete Strategies → Implement the interface.



Example 01: E-commerce Payment

Imagine you are building an E-commerce platform. After adding items to the cart, customers must choose a payment method to complete their purchase. The system needs to support multiple ways of payment:

- Credit Card – customers enter their card details, and the system processes the transaction through a card gateway.
- PayPal – customers log in with their PayPal account and authorize payment securely.
- Bank Transfer – customers directly transfer the amount from their bank account using online banking.

Example 01: E-commerce Payment

- Customer chooses payment method at checkout.

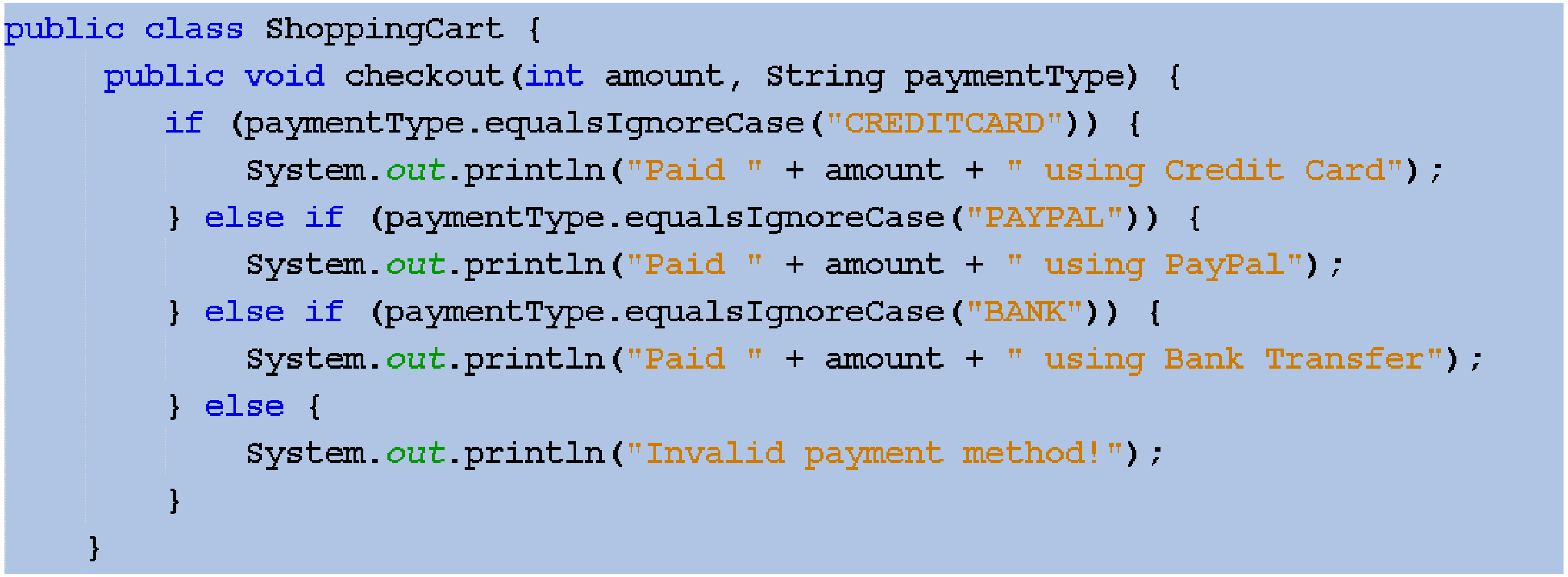
- Strategies:
  - CreditCardPayment
  - PayPalPayment
  - BankTransferPayment

- Each implements a common interface PaymentStrategy.

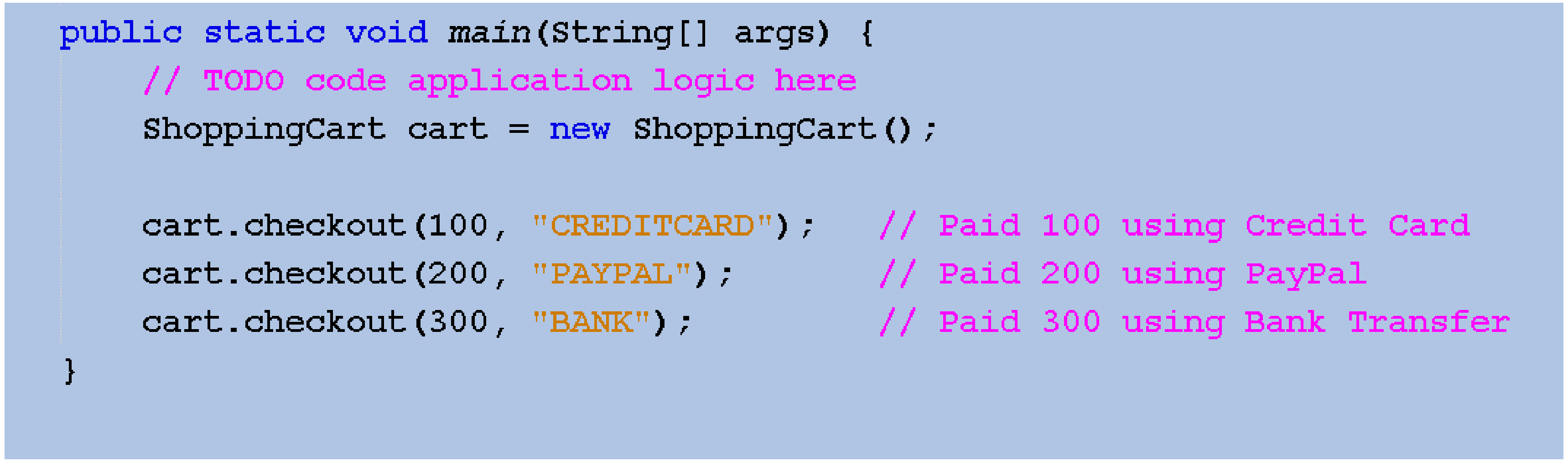
- Context: ShoppingCart → uses chosen payment strategy.

Traditional Approach (Without Strategy Pattern)

- Create a ShoppingCart class
- Add a checkout() method Inside ShoppingCart class
- Put all the payment logic into one big method with several **if-else** or **switch** statements.



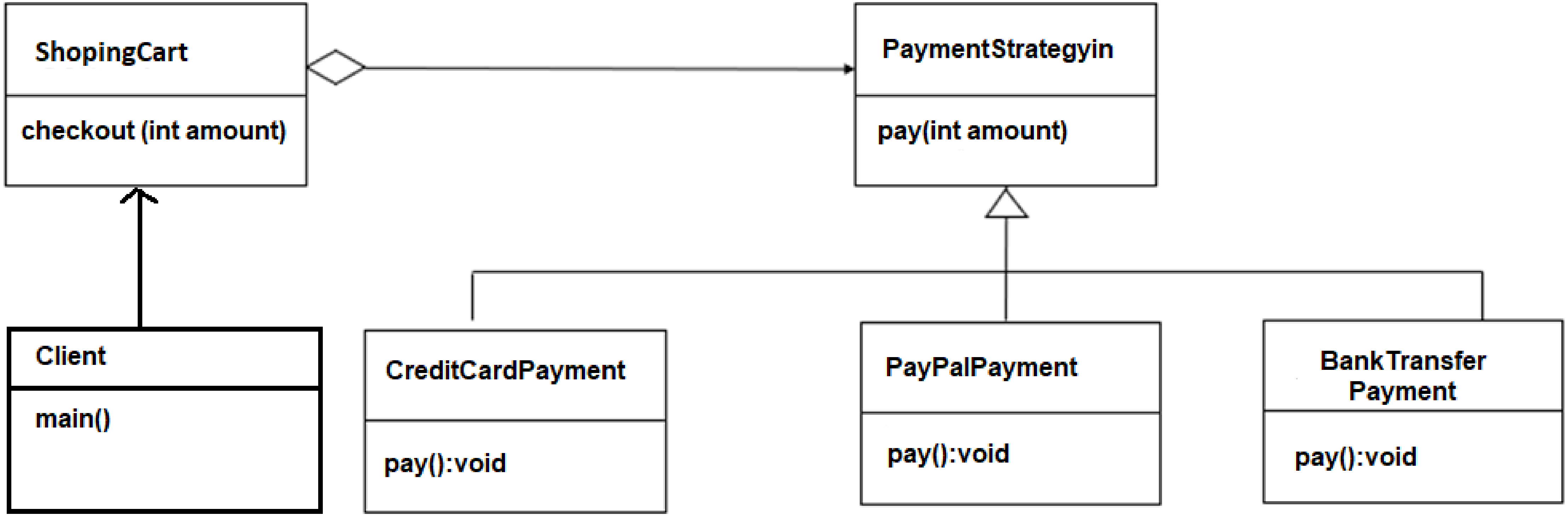
```java
public class ShoppingCart {
    public void checkout(int amount, String paymentType) {
        if (paymentType.equalsIgnoreCase("CREDITCARD")) {
            System.out.println("Paid " + amount + " using Credit Card");
        } else if (paymentType.equalsIgnoreCase("PAYPAL")) {
            System.out.println("Paid " + amount + " using PayPal");
        } else if (paymentType.equalsIgnoreCase("BANK")) {
            System.out.println("Paid " + amount + " using Bank Transfer");
        } else {
            System.out.println("Invalid payment method!");
        }
    }
```



```java
public static void main(String[] args) {
    // TODO code application logic here
    ShoppingCart cart = new ShoppingCart();

    cart.checkout(100, "CREDITCARD");   // Paid 100 using Credit Card
    cart.checkout(200, "PAYPAL");       // Paid 200 using PayPal
    cart.checkout(300, "BANK");         // Paid 300 using Bank Transfer
}
```

- **Problem with this Approach**: Payment logic tied up in long if–else chains. Hard to extend, tightly coupled, and violates good OOP principles.
- **Solution**: Use Strategy Pattern to separate each payment into its own class and keep ShoppingCart clean and flexible.

Strategy Pattern - UML Class Diagram



Components of Strategy Design Pattern

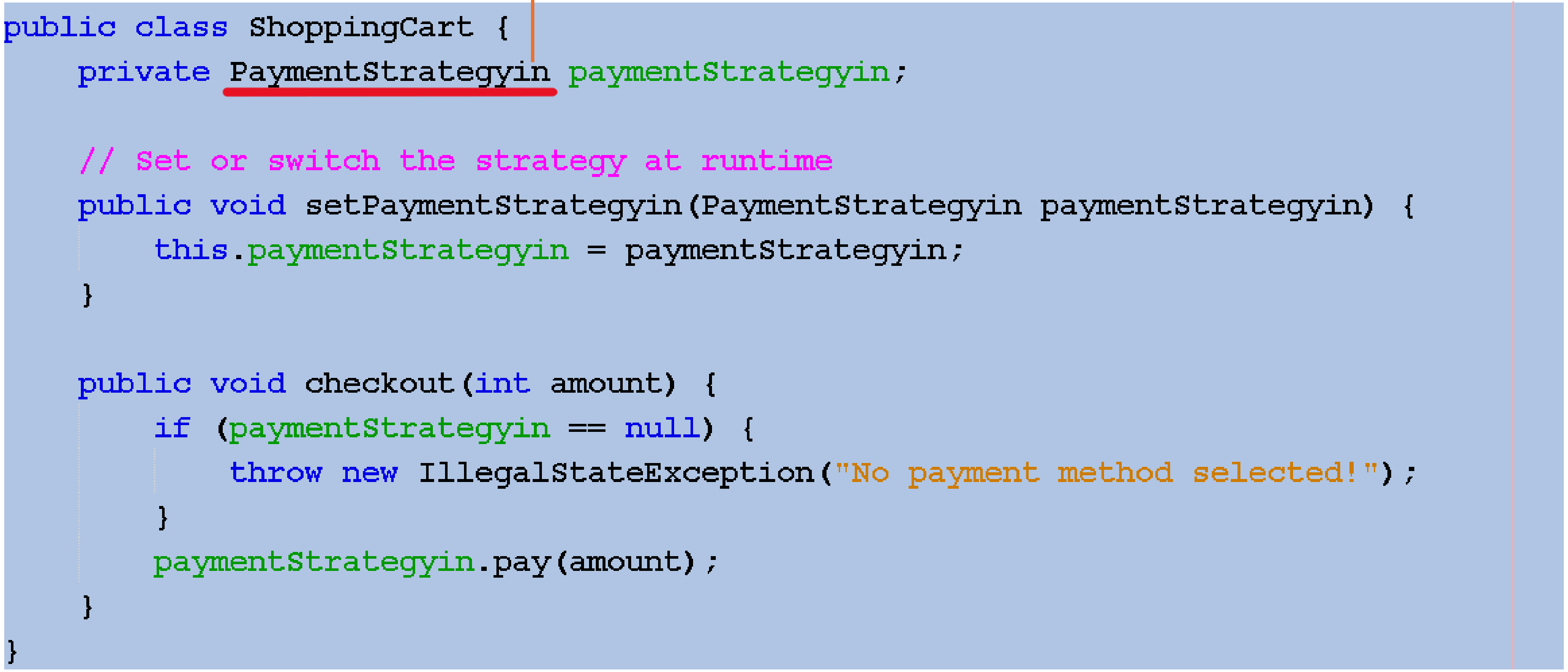
1.Context

- The Context is the main class that the client interacts with.
- It doesn’t implement the algorithm itself but keeps a reference to a Strategy object and uses it.

Example:

ShoppingCart → It holds a reference to a PaymentStrategy and uses it during checkout.

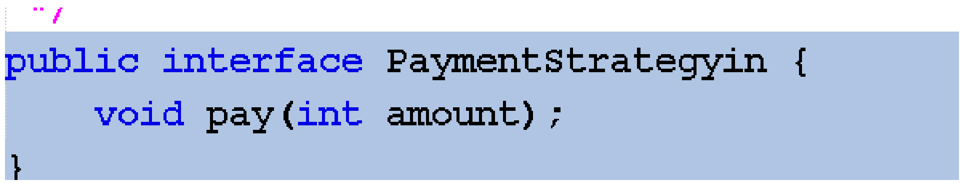
Strategy Interface



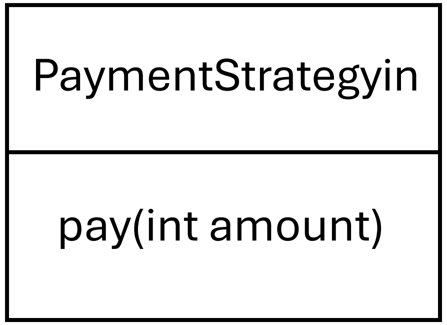
```
public class ShoppingCart {
    private PaymentStrategyin paymentStrategyin;

    // Set or switch the strategy at runtime
    public void setPaymentStrategyin(PaymentStrategyin paymentStrategyin) {
        this.paymentStrategyin = paymentStrategyin;
    }

    public void checkout(int amount) {
        if (paymentStrategyin == null) {
            throw new IllegalStateException("No payment method selected!");
        }
        paymentStrategyin.pay(amount);
    }
}
```

2. Strategy

The **common interface** that all strategies must follow. It defines a method but doesn’t provide implementation.



```
 */
public interface PaymentStrategyin {
    void pay(int amount);
}
```



Example:

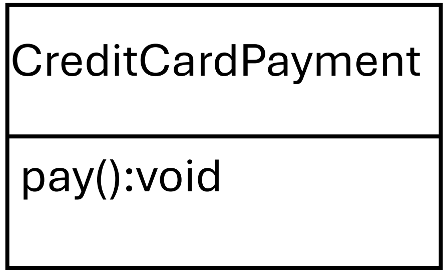
PaymentStrategy → Declares the method pay(int amount) but doesn’t define how payment happens.

3.Concrete Strategies

- These are the **real classes** that implement the Strategy interface. Each one provides a different version of the algorithm.

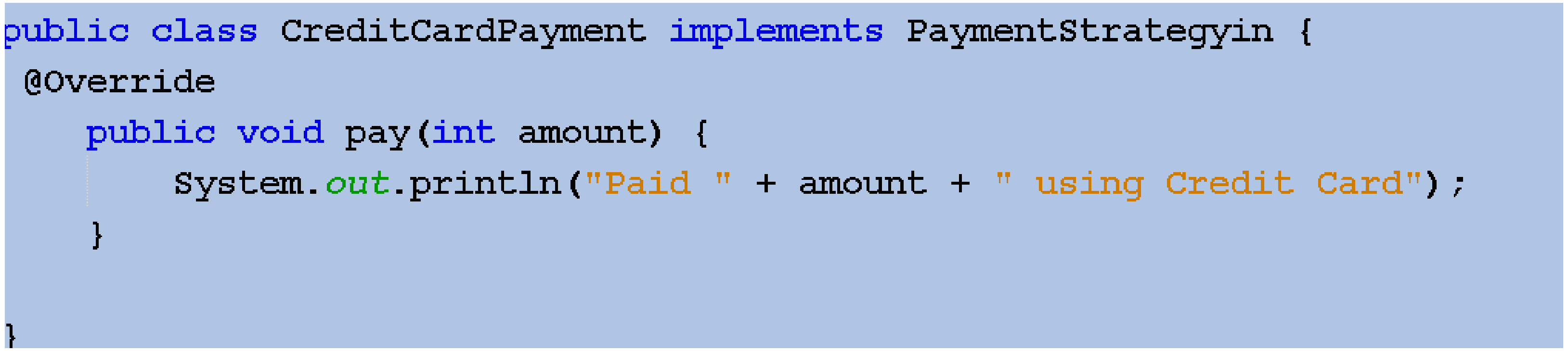
Example:

- CreditCardPayment → Defines how to process payment using a credit PayPalPayment → Defines how to process payment via paypal BankTransferPayment → Defines how to process payment via bank transfer.

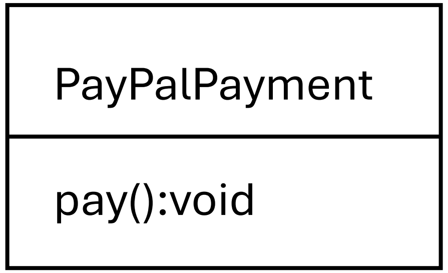


CreditCardPayment

pay():void

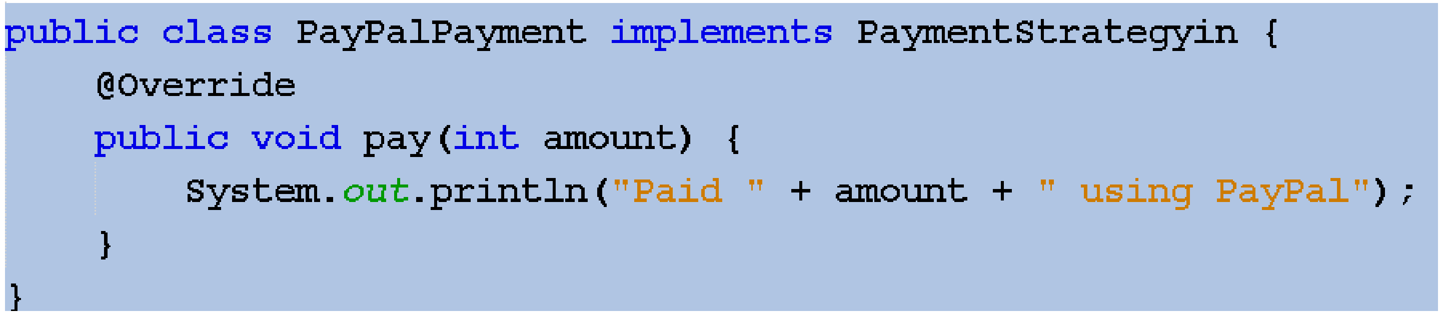


```
public class CreditCardPayment implements PaymentStrategyin {
 @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card");
    }


}
```

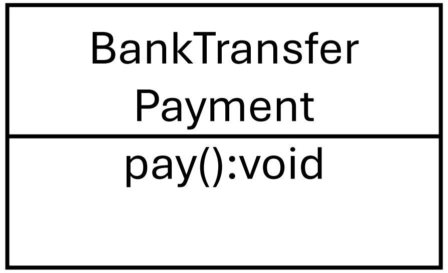


PayPalPayment

pay():void

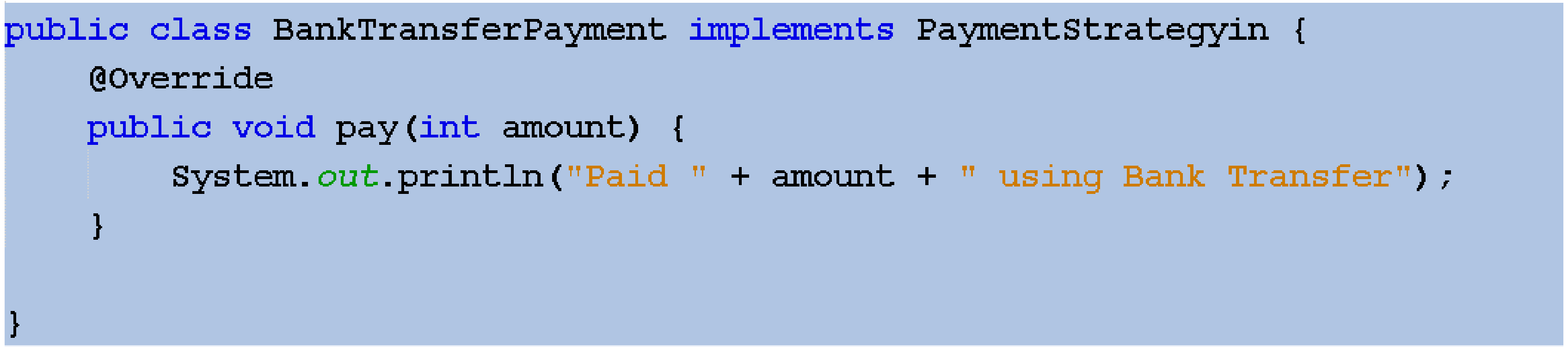


```
public class PayPalPayment implements PaymentStrategyin {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal");
    }
}
```



BankTransfer Payment

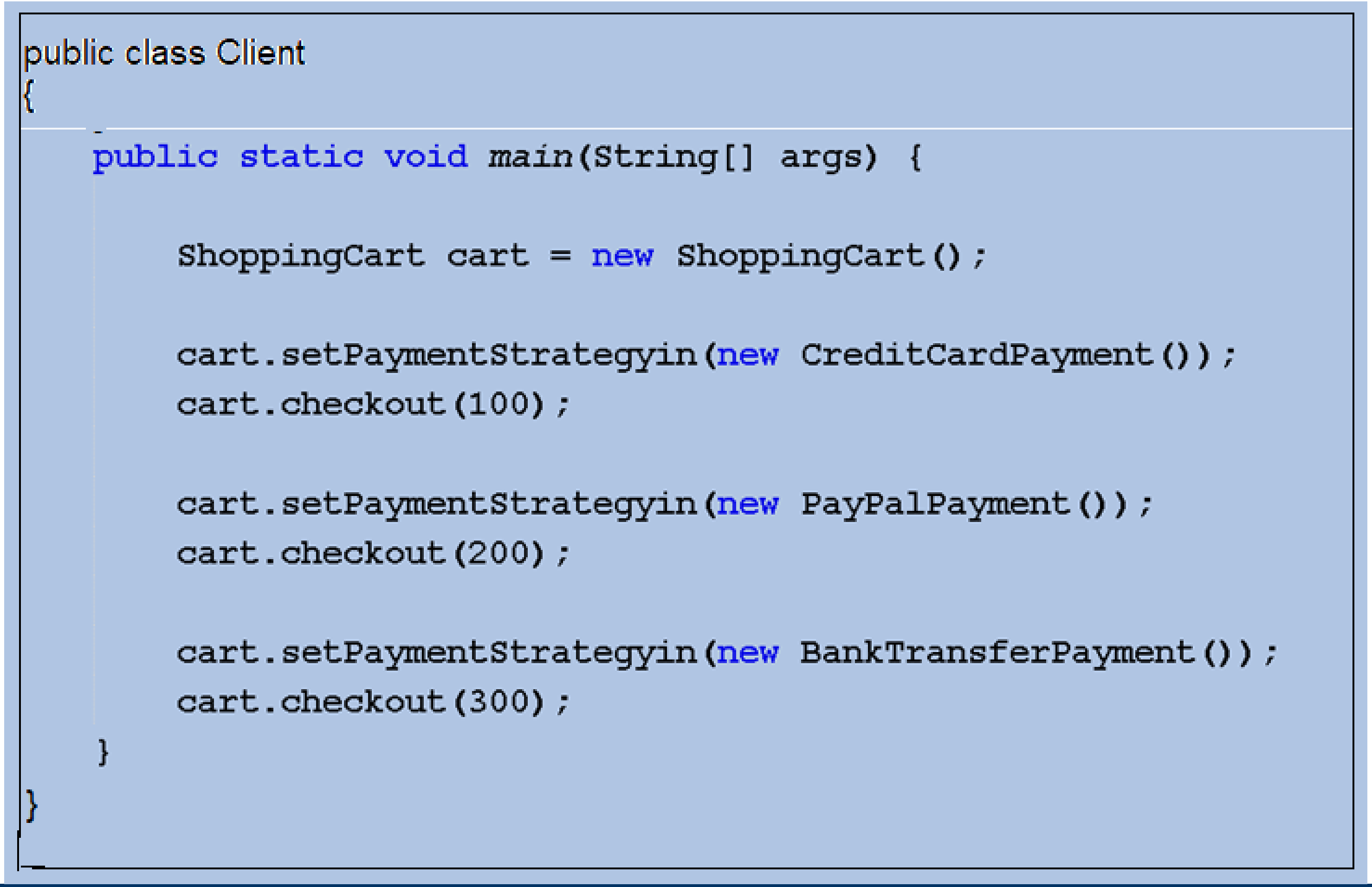
pay():void



```
public class BankTransferPayment implements PaymentStrategyin {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Bank Transfer");
    }

}
```

4.Client

- The end-user code or the application logic that selects which strategy should be applied.



```
public class Client
{
    public static void main(String[] args) {

        ShoppingCart cart = new ShoppingCart();

        cart.setPaymentStrategyin(new CreditCardPayment());
        cart.checkout(100);

        cart.setPaymentStrategyin(new PayPalPayment());
        cart.checkout(200);

        cart.setPaymentStrategyin(new BankTransferPayment());
        cart.checkout(300);
    }
}
```

Output

```
Paid 100 using Credit Card.
Paid 200 using PayPal.
Paid 300 using Bank Transfer.
```

Practical Uses of Strategy Pattern

Use Strategy Pattern when there are **multiple interchangeable ways** to achieve the same goal, and you want to **switch easily at runtime**.

- **Payment Systems (E-commerce)** → Credit Card, PayPal, Bank Transfer, Crypto.
- **Sorting Algorithms** → QuickSort, MergeSort, BubbleSort.
- **Compression Tools** → ZIP, RAR, GZIP.
- **Navigation Apps** → Fastest Route, Shortest Distance, Avoid Tolls.
- **Authentication** → Username/Password, OAuth, Biometric.
- **Game AI** → Aggressive, Defensive, Stealth attack behaviors.

Advantages of Strategy Pattern

- **Eliminates Long if–else / switch statements**
  → Each algorithm lives in its own class, so code is cleaner.
- **Easy to Extend**
  → Add a new algorithm by creating a new class (no need to change old code).
- **Runtime Flexibility**
  → Change behavior (algorithm/strategy) while the program is running.
- **Follows OOP Principles**
  → Supports **Open/Closed Principle** (add new code without modifying old code).
  → Respects **Single Responsibility Principle** (separates business logic from algorithms).
- **Improves Reusability**
  → Strategies can be reused in other projects without depending on the context.

Disadvantages of Strategy Pattern

- **More Classes to Manage**

→ Each new algorithm = a new class → can increase project size.

- **Client Awareness Needed**

→ The client (e.g., programmer or system) must know which strategy to pick and when.

- **Slightly More Complex Design**

→ Compared to a simple if–else, setting up interfaces and classes takes extra effort.

- **Overhead for Small Problems**

→ If only one or two algorithms exist and they rarely change, Strategy may feel like over-engineering.

Creational: Factory Pattern

What is Factory Pattern?

- A Creational Design Pattern.
- Defines an interface for creating objects.
- Subclasses/factory decide which class to instantiate.
- Client code depends only on factory + interface, not on concrete classes.

Why Use Factory Pattern?

- Hides object creation details.
- Reduces tight coupling between client and classes.
- Easy to add new product types (Open/Closed Principle).
- Centralizes object creation logic.
- Makes client code cleaner & more readable.

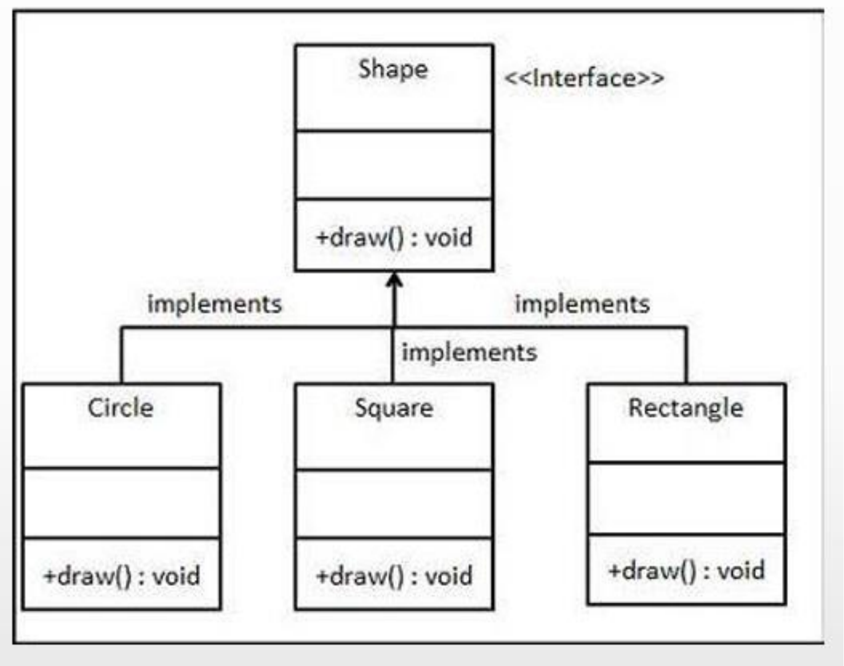
Factory Pattern

The **Factory Pattern** is one of the **creational design patterns** in Object-Oriented Programming .

It allows you to,

1. Defines an interface for creating objects.
2. Let subclasses decide which class to instantiate.
3. Client code only depends on the factory, not the

concrete classes.



Example:

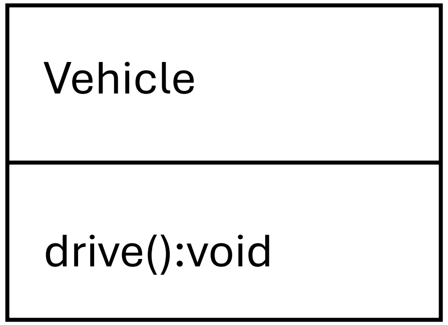
Imagine you are building a Transport Management System for a vehicle rental company Customers can request different types of vehicles based on their needs:

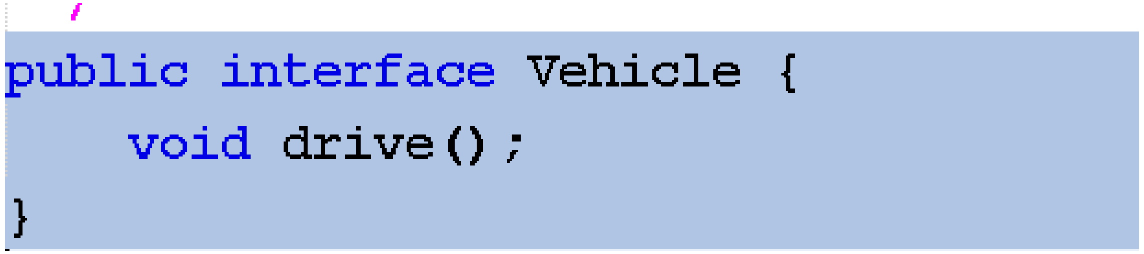
Car for family or business trips.

Bike for quick deliveries or solo rides.

Truck for moving goods or heavy transport.

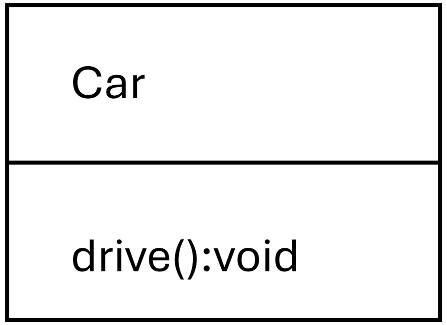
1.Defines Common Interface





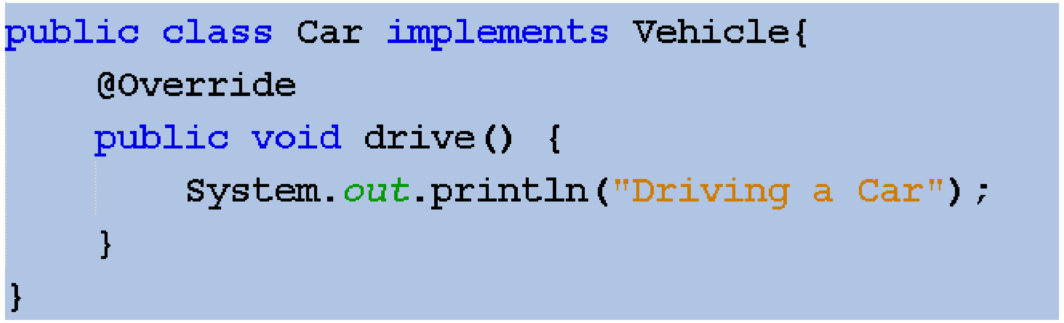
```
public interface Vehicle {
    void drive();
}
```

2. Create concrete classes implementing the same interface.

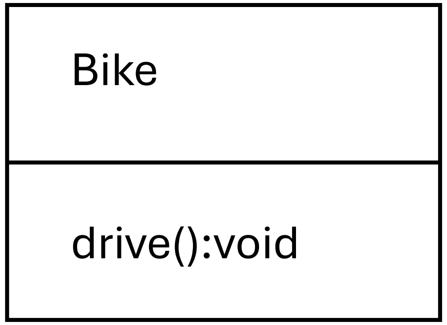


Car

drive():void

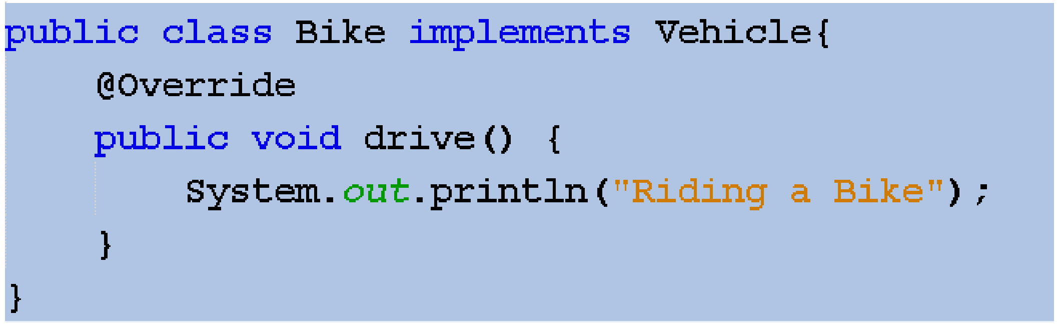


```java
public class Car implements Vehicle{
    @Override
    public void drive() {
        System.out.println("Driving a Car");
    }
}
```

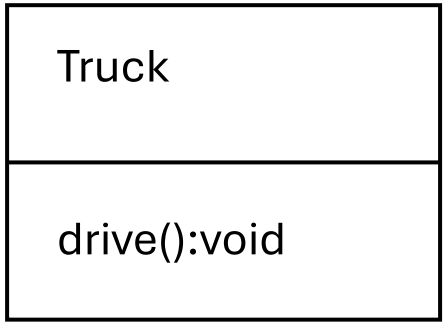


Bike

drive():void

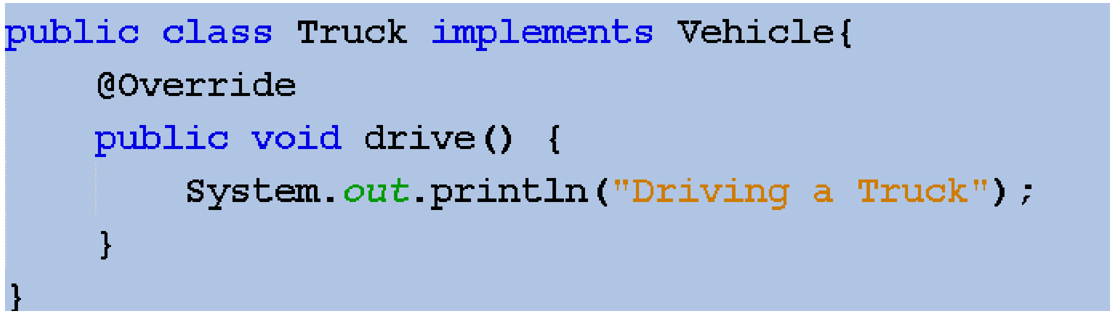


```java
public class Bike implements Vehicle{
    @Override
    public void drive() {
        System.out.println("Riding a Bike");
    }
}
```



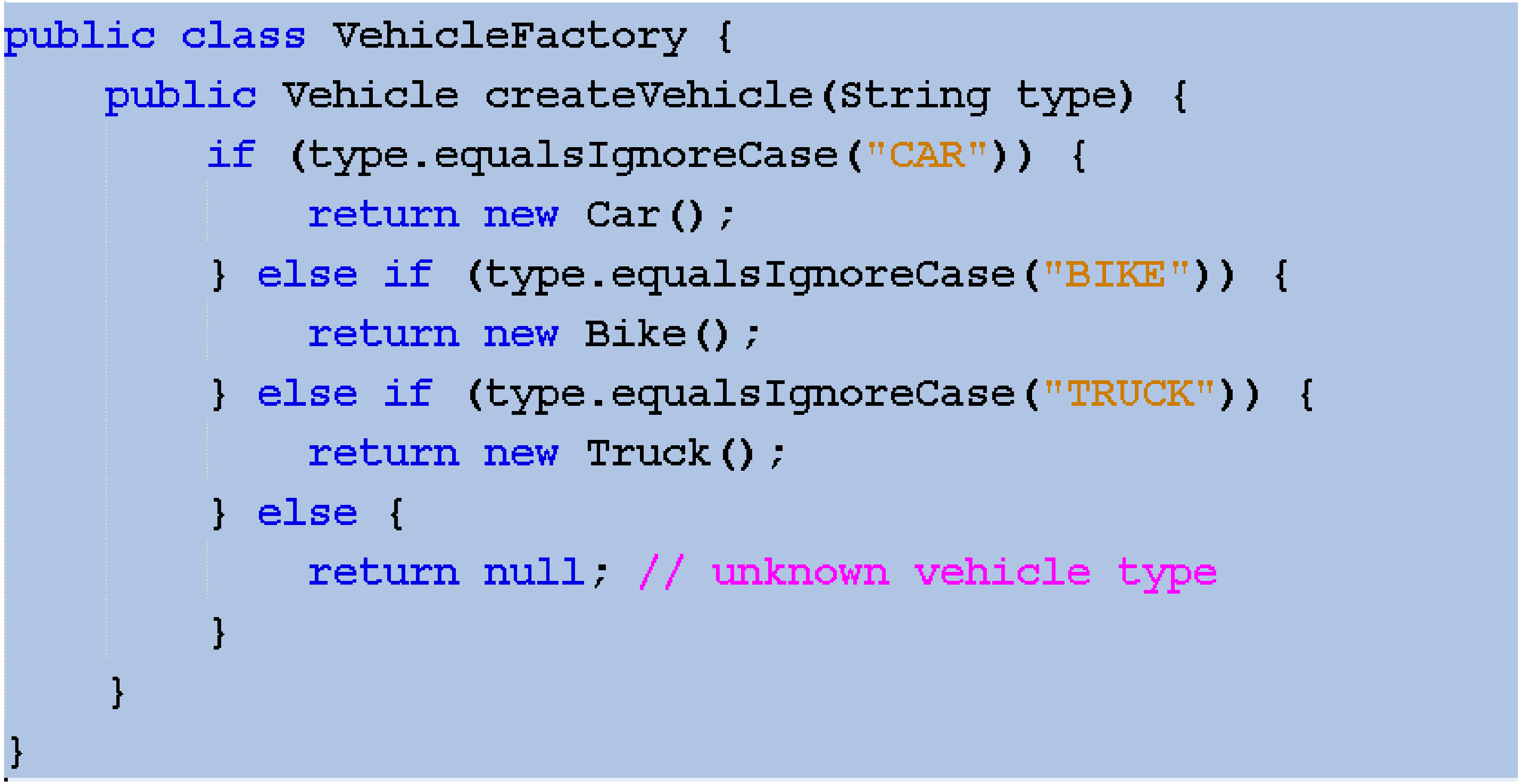
Truck

drive():void



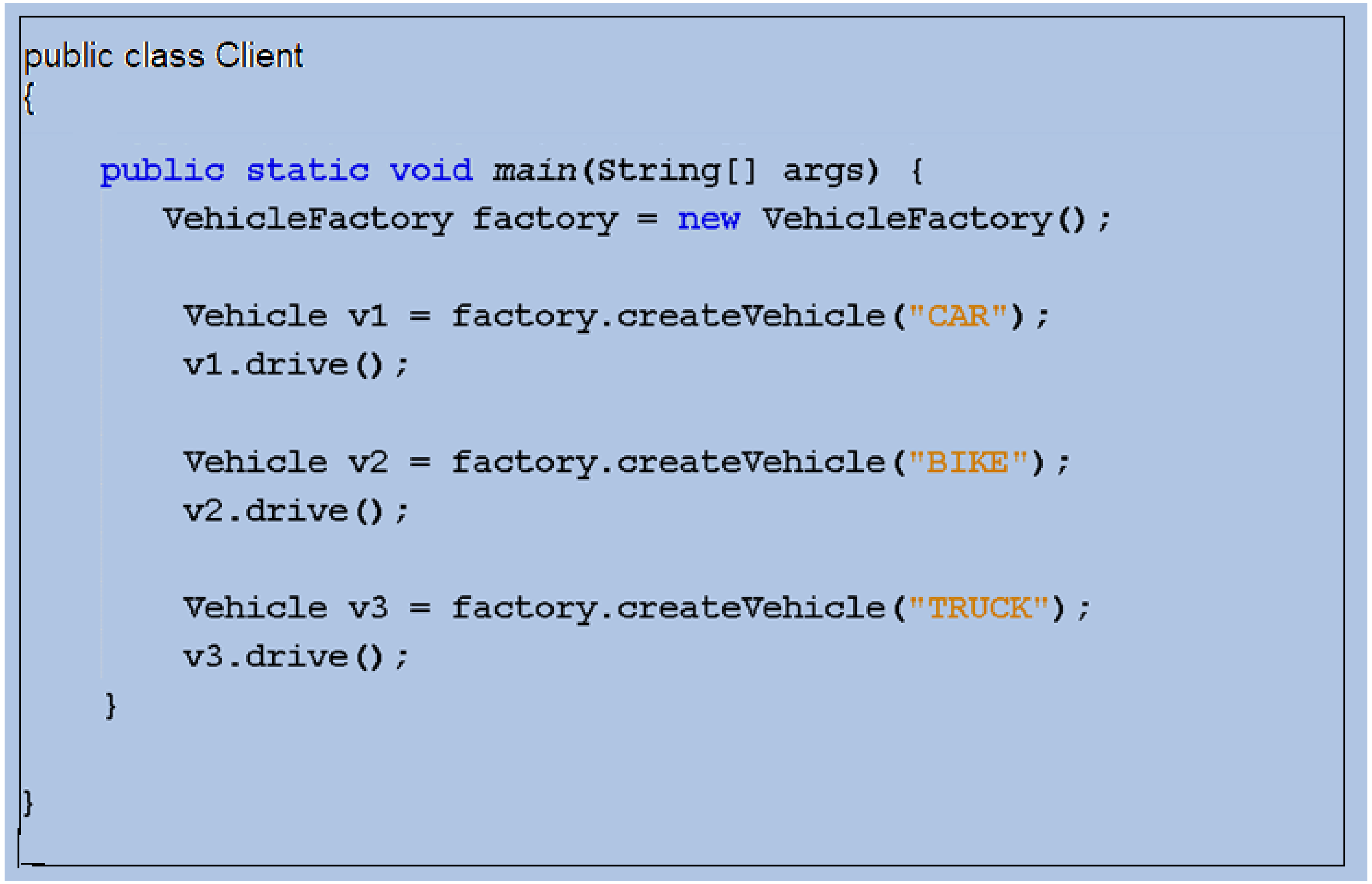
```java
public class Truck implements Vehicle{
    @Override
    public void drive() {
        System.out.println("Driving a Truck");
    }
}
```

3.Create a Factory Class



```java
public class VehicleFactory {
    public Vehicle createVehicle(String type) {
        if (type.equalsIgnoreCase("CAR")) {
            return new Car();
        } else if (type.equalsIgnoreCase("BIKE")) {
            return new Bike();
        } else if (type.equalsIgnoreCase("TRUCK")) {
            return new Truck();
        } else {
            return null; // unknown vehicle type
        }
    }
}
```

4.Use Factory in Client(Context) Code



```
public class Client
{

    public static void main(String[] args) {
       VehicleFactory factory = new VehicleFactory();

        Vehicle v1 = factory.createVehicle("CAR");
        v1.drive();

        Vehicle v2 = factory.createVehicle("BIKE");
        v2.drive();

        Vehicle v3 = factory.createVehicle("TRUCK");
        v3.drive();
    }


}
```

Practical Uses of Factory Pattern

- **UI Frameworks** – creating buttons, dialogs (Java Swing, Android).
- **Database Connections** – JDBC drivers.
- **File Parsers** – PDF, Word, Excel reader factories.

  Suppose you open a file. The factory chooses the correct parser (PDFParser, WordParser, ExcelParser).)
- **Games** – enemies, weapons, power-ups.
- **Cross-platform apps** – return Windows or Mac GUI elements

Advantages of Factory Pattern

- Hides object creation details.
- Cleaner & more reusable code.
- Easy to extend with new classes.
- Promotes loose coupling.

Disadvantages of Factory Pattern

- Adds **extra classes** (factories + products).
- Slightly more complex design.
- If not used carefully, it can lead to “factory explosion” (too many factories).

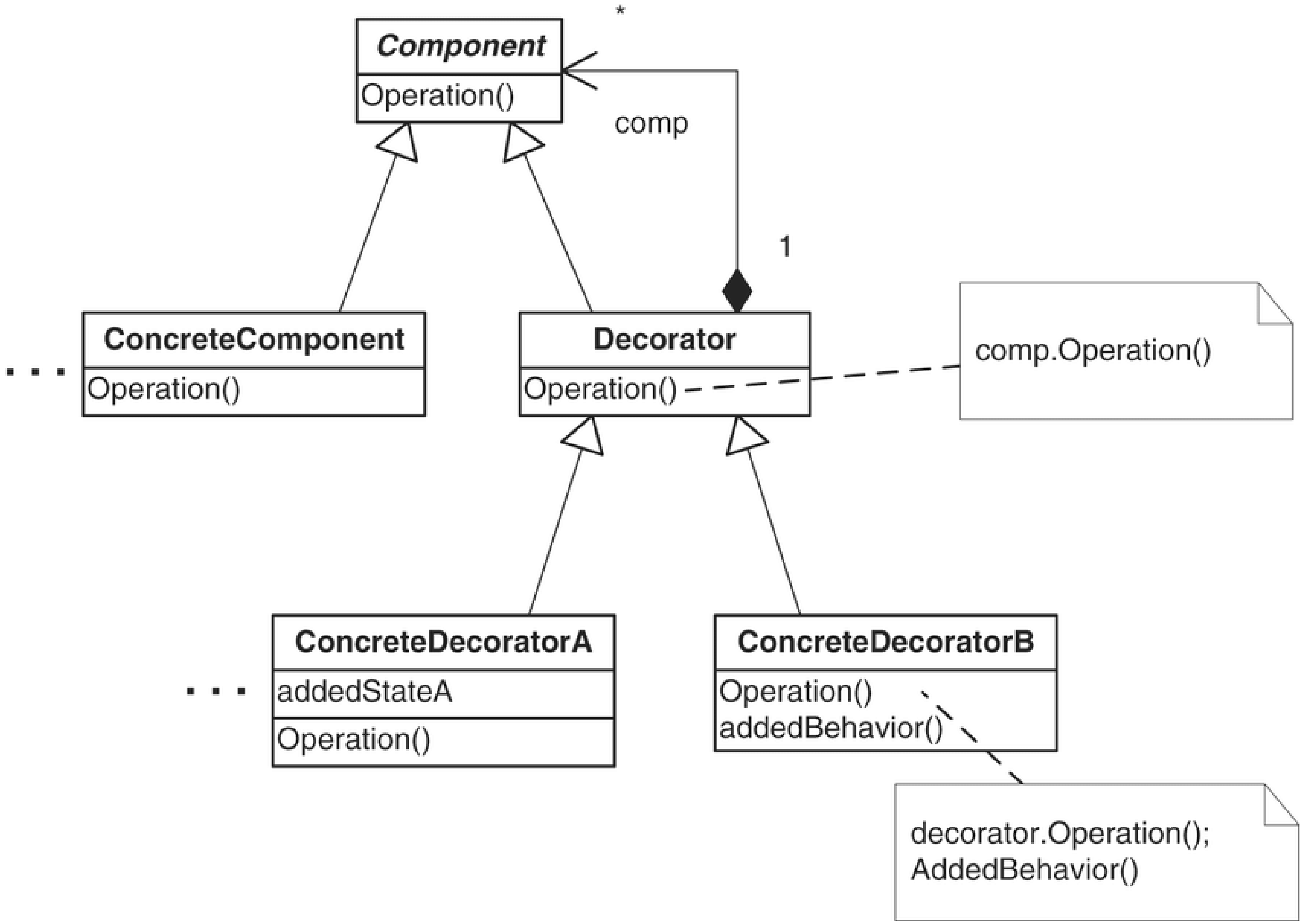
Structural :Decorator Design Pattern

(Self Study)

What is Decorator Pattern

- A **Structural Design Pattern.**
- Attaches new behavior to objects **dynamically at runtime.**
- Works by **wrapping** the original object with a decorator object.
- Provides flexibility without modifying the original class.

Class Diagram of Decorator Design Pattern



Example : Online Coffee Ordering

- Imagine you are building an online coffee ordering app.
- The customer can order a basic coffee.
- They can customize it with add-ons like:
  - Milk
  - Sugar

Problem with the Traditional Approach:

If we try to use inheritance, we'd need too many classes:

MilkCoffee, SugarCoffee, MilkSugarCoffee, MilkSugarCreamCoffee, etc..

Solution:

Use Decorator Pattern

Example : Online Coffee Ordering

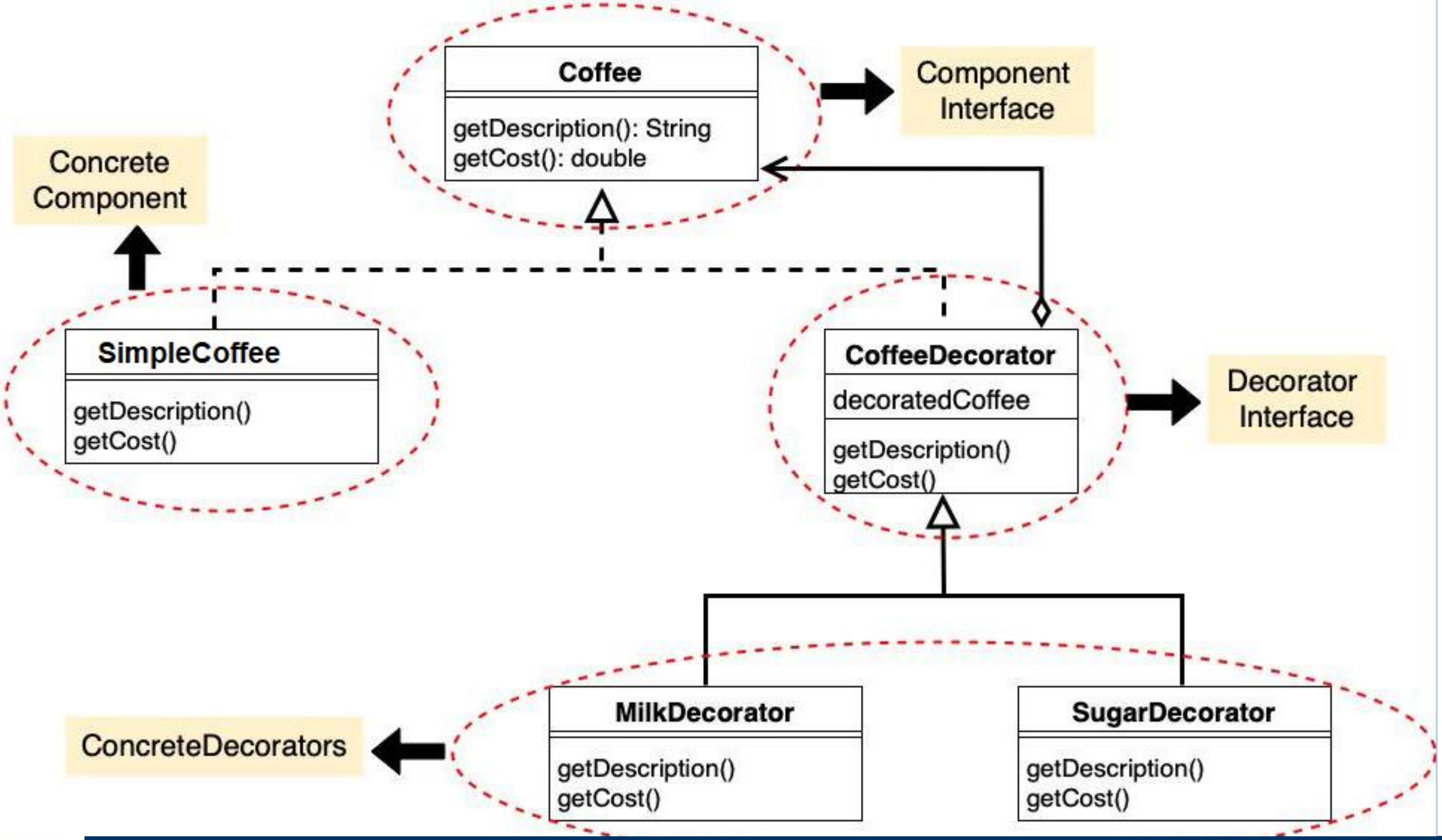
- Using the Decorator Pattern, we start with a SimpleCoffee as the base object, and each add-on such as Milk, Sugar, or Whipped Cream is implemented as a separate Decorator class.

- When a customer wants a customized coffee, for example with milk and sugar, we simply wrap the base coffee object with the MilkDecorator and then with the SugarDecorator, creating a chain like,

  SimpleCoffee → MilkDecorator → SugarDecorator

- This approach makes it easy to mix and match add-ons dynamically, without the need to create dozens of separate subclasses for every possible combination.

Class Diagram of Decorator Design Pattern



Component & Concrete Component

```
// Component
public interface Coffee {
   String getDescription();
   double getCost();
}

// Concrete Component
public class SimpleCoffee implements Coffee {
   public String getDescription() { return "Simple Coffee"; }
   public double getCost() { return 2.0; }
}
```

Abstract Decorator

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

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

    public String getDescription() { return decoratedCoffee.getDescription(); }
    public double getCost() { return decoratedCoffee.getCost(); }
}
```

Concrete Decorators

```
// Milk Decorator
public class MilkDecorator extends CoffeeDecorator {
   public MilkDecorator(Coffee coffee) { super(coffee); }
   public String getDescription() { return
decoratedCoffee.getDescription() + ", Milk"; }
   public double getCost() { return decoratedCoffee.getCost() + 0.5; }
}

// Sugar Decorator
public class SugarDecorator extends CoffeeDecorator {
   public SugarDecorator(Coffee coffee) { super(coffee); }
   public String getDescription() { return
decoratedCoffee.getDescription() + ", Sugar"; }
   public double getCost() { return decoratedCoffee.getCost() + 0.2; }
}
```

Client Code (Main)

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

        coffee = new MilkDecorator(coffee);
        System.out.println(coffee.getDescription() + " $" + coffee.getCost());

        coffee = new SugarDecorator(coffee);
        System.out.println(coffee.getDescription() + " $" + coffee.getCost());
    }
}
```

Practical Uses of Decorator Pattern

- UI Components → Adding scrollbars, borders, colors dynamically to windows or text fields.

- Like in food places (coffee shops, pizza, ice cream), you start with a base item and add different toppings or extras dynamically.

- Messaging Apps → Add encryption, compression, logging as decorators to message objects.

- Game Development → Equip a character with weapons, armor, or powers without creating many subclasses.

Advantages of Decorator Pattern

- Add behavior at runtime → wrap objects flexibly.
- Avoids subclass explosion → no need many classes.
- Highly flexible → combine multiple decorators in any order.
- Reusable decorators → same `MilkDecorator` can be used on any `Coffee`.

Disadvantages of Decorator Pattern

- More classes & objects → each feature = a new class
- Debugging is harder → behavior is spread across multiple wrappers.
- Not always necessary → for simple problems, may overcomplicate the design.

Thank You