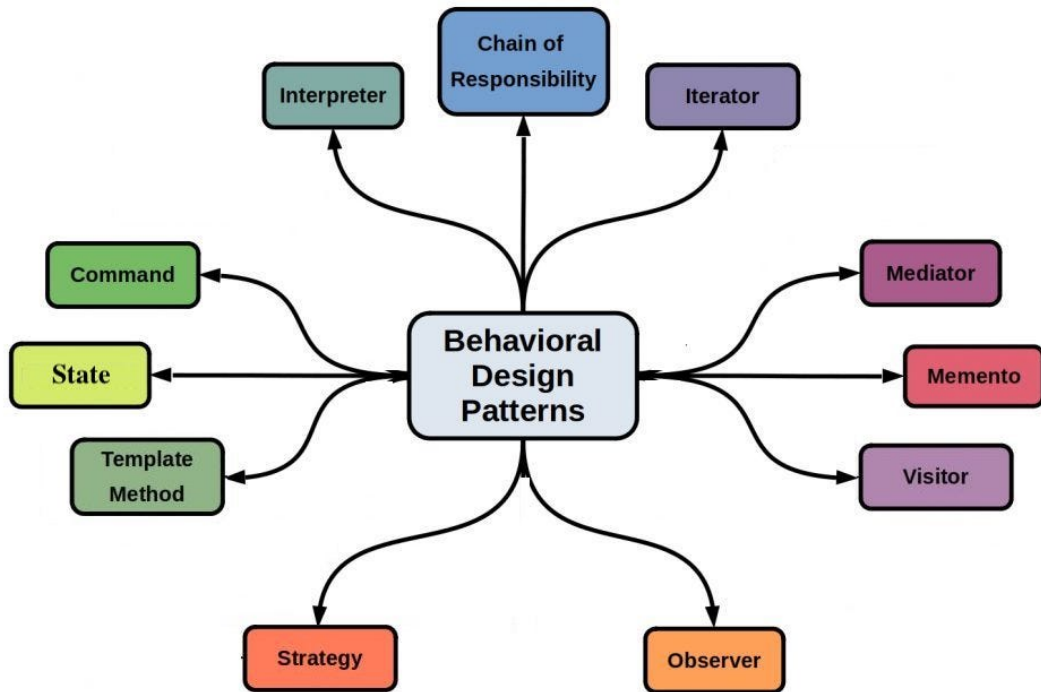


Design Patterns: Behavioural

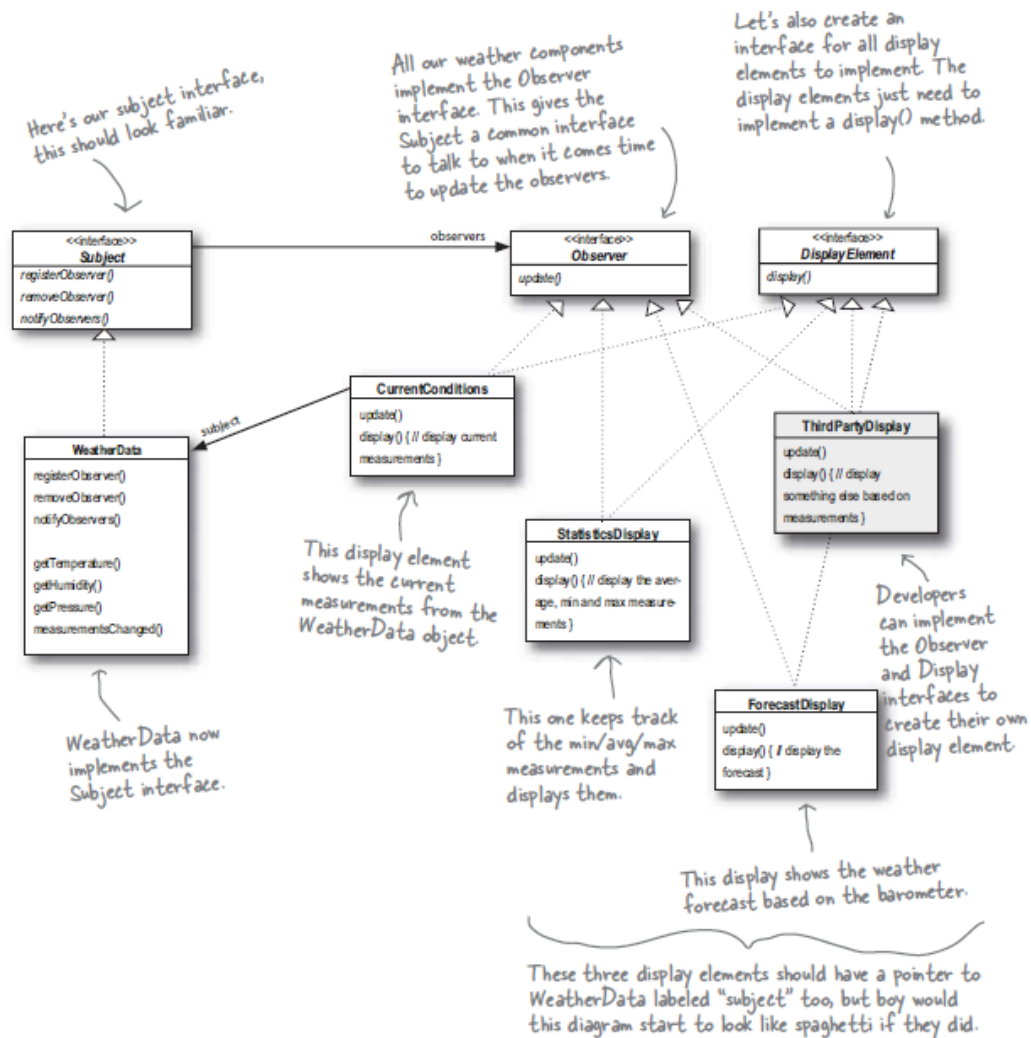
▼ Behavioural Design Patterns:

- Behavioural patterns deal with object collaboration and delegation — They help define how objects interact and communicate with each other.
- These patterns help manage **complex control flows** and communication between objects.
- They define **how responsibilities are divided**, how **objects request actions** from one another, and how they can **communicate changes in state**.
- Behavioural patterns are critical in maintaining **flexibility** and **scalability** in systems with a large number of interacting objects.
- They help avoid **spaghetti code** by organising and defining object relationships and promoting loose coupling and high cohesion.



▼ What is the Observer Pattern?

- The **Observer Pattern** defines a one-to-many relationship between objects, where a **subject** (the one) maintains a list of **observers** (the many) that need to be notified of any changes in its state.
- When the subject's state changes, it notifies all of its observers automatically.
- This pattern is widely used in **event-driven** systems and **graphical user interfaces** (GUIs).



Why Use the Observer Pattern?


- The observer and the subject are loosely coupled. The subject doesn't need to know any details about the observers—it just needs to notify them.
- Ideal for systems where several parts need to respond to the state of another object.

Examples:

- A **weather station** application, where various displays (observers) need to update when the weather changes (subject).
- A **stock price tracking system**, where multiple screens update with price changes.

Observer


Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to

 <https://refactoring.guru/design-patterns/observer>



The Observer Pattern in Java | Baeldung

Learn a few ways to implement the Observer design pattern in Java

 <https://www.baeldung.com/java-observer-pattern>



▼ Basic Implementation of the Observer Pattern

- The **Observer Pattern** establishes a one-to-many relationship between a subject (or observable) and multiple observers.
- When the state of the subject changes, all its observers are notified.
- This pattern is commonly used in **event-driven** systems, where one object's change affects many others.

```
import java.util.ArrayList;
import java.util.List;

// Step 1: Create the Subject (Observable) class
class Subject {
    private List<Observer> observers = new ArrayList<>();
    private int state;

    // Method to get the state
    public int getState() {
        return state;
    }

    // Method to change the state
    public void setState(int state) {
        this.state = state;
    }
}
```

```

        notifyAllObservers(); // Notify observers
when state changes
    }

    // Method to attach observers
    public void attach(Observer observer) {
        observers.add(observer);
    }

    // Method to notify all observers about the sta
te change
    private void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update(); // Each observer's
update method is called
        }
    }
}

// Step 2: Create an abstract Observer class
abstract class Observer {
    protected Subject subject;

    public abstract void update();
}

// Step 3: Concrete Observer classes implementing t
he Observer interface
class HexObserver extends Observer {
    public HexObserver(Subject subject) {
        this.subject = subject;
        this.subject.attach(this); // Attach this
observer to the subject
    }

    @Override
    public void update() {
        System.out.println("Hex String: " + Intege

```

```

        r.toHexString(subject.getState()));
    }
}

class BinaryObserver extends Observer {
    public BinaryObserver(Subject subject) {
        this.subject = subject;
        this.subject.attach(this); // Attach this
observer to the subject
    }

    @Override
    public void update() {
        System.out.println("Binary String: " + Integer.toBinaryString(subject.getState()));
    }
}

// Step 3: Test the Observer Pattern
public class ObserverPatternDemo {
    public static void main(String[] args) {
        Subject subject = new Subject();

        new HexObserver(subject); // Create and
attach observers
        new BinaryObserver(subject); // Create and
attach observers

        System.out.println("First state change: 1
5");
        subject.setState(15); // Set state and see
the update in observers

        System.out.println("Second state change: 1
0");
        subject.setState(10); // Set state again a
nd see the update
    }
}

```

```
}  
}
```

▼ Key Concepts:

- The **Subject** class manages a list of **observers**. When its state changes, it triggers the `notifyAllObservers` method to call each observer's `update` method.
- **Observers** are abstract classes that must implement the `update` method. This allows flexibility in how each observer reacts to the subject's state changes.
- **Concrete Observers:** In this example, two observers (`HexObserver` and `BinaryObserver`) are created, each with their own implementation of the `update` method.
- **Notification Logic** — When the state of the **Subject** is changed, it calls `notifyAllObservers`, and each observer's `update()` method gets called.
- **Key Insight:** This demonstrates the **loose coupling** achieved between the subject and observers. The subject only knows it has to notify its observers, but it doesn't need to know what actions they will take.

▼ Benefits of the Observer Pattern

- You can add or remove observers at runtime without modifying the subject. This dynamic nature allows the system to scale easily as more observers are added.
- The observer pattern allows a subject to broadcast updates to multiple observers, which can be useful in systems where changes need to be reflected in various parts of the application.
- The pattern divides the system into multiple modules (subject and observers) that can evolve independently, improving maintainability.
- Observers can be reused across different subjects, which increases the flexibility and reusability of the code.
- The subject doesn't need to know the specific details of how the observer processes the data. This allows for easier maintenance

and updates to either the subject or the observers without affecting the other.

▼ Common Pitfalls in Observer Pattern

- **Memory Leaks**

- If observers aren't properly removed from the subject's list when they are no longer needed, it can lead to **memory leaks**.
- This happens especially in long-running applications where observers accumulate over time, even if they are no longer required.

Example:

Imagine a scenario where observers are created for monitoring the state of a UI component, but those observers are never removed even after the UI component is no longer in use.

```
class Subject {
    private List<Observer> observers = new Array
List<>();

    public void attach(Observer observer) {
        observers.add(observer);
    }

    public void detach(Observer observer) {
        observers.remove(observer); // Observer
should be removed when no longer needed
    }

    public void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}

// Observer pattern implementation without detac
hing observers
public class MemoryLeakExample {
```



```

public static void main(String[] args) {
    Subject subject = new Subject();

    Observer observer1 = new ConcreteObserver();
    Observer observer2 = new ConcreteObserver();

    subject.attach(observer1);
    subject.attach(observer2);

    // Later on, observer1 and observer2 are
    // no longer used but are never detached
    // They still reside in the observers list,
    // contributing to a memory leak.
}
}

```

In this example, the observers remain in memory even if they are no longer needed because they are not **detached**.

Solution:

Always ensure that

observers are removed when they are no longer necessary.

```

subject.detach(observer1); // Properly detach observer
                             when no longer needed

```

- **Performance Overhead**

- When there are too many observers, the subject might end up **notifying** a large number of observers, which can degrade performance, especially if all observers are performing computationally heavy tasks.

Example:

Let's say you have a stock price tracking system, where thousands of users (observers) are tracking the price of a single stock (subject). If the stock price updates frequently and the

system notifies every observer for each update, it could cause performance bottlenecks.

```
class StockPriceSubject {
    private List<Observer> observers = new Array
List<>();
    private int price;

    public void setPrice(int newPrice) {
        this.price = newPrice;
        notifyAllObservers(); // Notifies a lar
ge number of observers for each price change
    }

    public void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}
```

Imagine this system scaling up to thousands of users. Each update would notify all observers, and the performance might degrade with every price change.

Solution:

- Instead of notifying all observers for every small change, you can **batch** updates or use **rate-limiting** to reduce the number of notifications, especially when updates happen frequently.

- **Circular Dependencies**

- If two subjects observe each other, this can result in a **circular dependency** where both objects continuously notify each other, leading to infinite loops.

Example:

Imagine an application where two different modules monitor each other's state. If Module A is updated and notifies Module B, which

then updates itself and notifies Module A again, this would create an infinite loop of updates.

```
class ModuleA extends Subject implements Observer {
    public void update() {
        System.out.println("Module A updated.");
        setState(); // Changes state and notifies observers (Module B)
    }
}

class ModuleB extends Subject implements Observer {
    public void update() {
        System.out.println("Module B updated.");
        setState(); // Changes state and notifies observers (Module A)
    }
}
```

In this scenario, Module A updates Module B, which in turn updates Module A, and so on, leading to an **infinite loop**.

Solution:

- Introduce a **check** to avoid recursive updates. You can also decouple these modules or re-design the system to prevent circular dependencies.

• **Lack of Control Over Updates**

- The **Observer Pattern** assumes that all observers are equally interested in every state change. However, some observers might only be interested in certain types of updates. Not distinguishing between different types of updates can lead to unnecessary processing in observers.

Example:

Let's say we have a

`WeatherStation` subject that notifies observers whenever the

temperature, humidity, or pressure changes. Some observers might only care about temperature, but they will still be notified for all changes.

```
class WeatherStation {
    private List<Observer> observers = new ArrayList<>();
    private int temperature;
    private int humidity;

    public void setWeatherData(int temp, int humidity) {
        this.temperature = temp;
        this.humidity = humidity;
        notifyAllObservers(); // Notifies observers for both temperature and humidity changes
    }
}
```

If an observer only needs temperature data, it would still get updates for humidity, which might be irrelevant.

Solution:

- Implement more **granular notifications**, where observers can specify which types of changes they are interested in.

```
class WeatherStation {
    private List<Observer> temperatureObservers = new ArrayList<>();
    private List<Observer> humidityObservers = new ArrayList<>();

    public void setTemperature(int temp) {
        this.temperature = temp;
        notifyTemperatureObservers(); // Notify only temperature observers
    }

    public void setHumidity(int humidity) {
```

```

        this.humidity = hum;
        notifyHumidityObservers(); // Notify on
ly humidity observers
    }
}

```

- **Tight Coupling Between Subject and Observers**

- Even though the **Observer Pattern** is meant to reduce coupling, it can still result in **tight coupling** if the subject and observers are implemented too specifically. This can make it difficult to extend or modify the system.

Example:

If a subject has a very specific type of observer, adding new types of observers could require changes to both the subject and the existing observers, defeating the purpose of decoupling.

```

class SpecificSubject {
    private SpecificObserver observer;

    public void setObserver(SpecificObserver obs
erver) {
        this.observer = observer;
    }

    public void notifyObserver() {
        observer.update(); // Subject is tightl
y coupled with a specific observer
    }
}

```

Solution:

- Ensure that the subject and observers communicate through **abstract interfaces**, so the system can be extended without needing to modify the existing code.

```

class Subject {
    private List<Observer> observers = new Array

```

```

List<>();

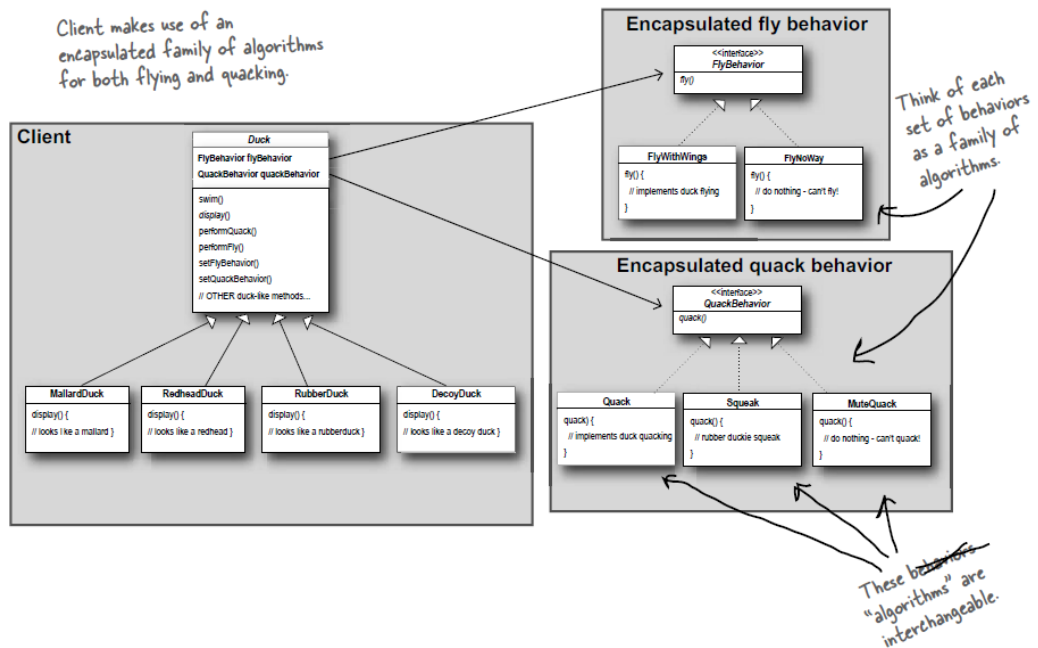
public void attach(Observer observer) {
    observers.add(observer);
}

public void notifyAllObservers() {
    for (Observer observer : observers) {
        observer.update();
    }
}
}
}

```

▼ What is the Strategy Pattern?

- The **Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- The key idea is to allow an algorithm's behaviour to be selected at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which of a family of algorithms to use.



Why Use the Strategy Pattern?


- **Flexibility:** The strategy pattern allows the behavior of an object to be changed at runtime by changing the algorithm it uses.
- **Decoupling:** The algorithm implementations are decoupled from the context class, allowing easier maintenance and testing.

Examples:

- Different **sorting algorithms** (e.g., bubble sort, quicksort) that can be selected at runtime.
- Different **payment methods** (e.g., credit card, PayPal) in an e-commerce system.

Strategy

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects

 <https://refactoring.guru/design-patterns/strategy>



REFACTORING
· GURU ·

Strategy Design Pattern in Java | Baeldung

Implementation of Strategy design pattern in the light of Java 8 features.

 <https://www.baeldung.com/java-strategy-pattern>



▼ Basic Implementation of Strategy Pattern

- The **Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- This pattern allows algorithms to vary independently from clients that use them.
- **Key Points** of the Strategy Pattern:
 - **Context:** The class that uses a strategy to complete a task.
 - **Strategy Interface:** The common interface that all strategies must implement.
 - **Concrete Strategies:** The various algorithms that implement the strategy interface.

- In the Strategy Pattern, the context object receives the strategy at runtime instead of being tightly coupled to a particular algorithm, thus promoting flexibility.

Example: Payment Strategy

Imagine an e-commerce system where users can choose different methods of payment (like credit cards, PayPal, etc.). Each payment method is a **strategy**.

1. Define a Strategy Interface:

```
public interface PaymentStrategy {  
    void pay(int amount);  
}
```

2. Implement Concrete Strategies:

```
public class CreditCardPayment implements PaymentStrategy {  
    private String cardNumber;  
  
    public CreditCardPayment(String cardNumber)  
    {  
        this.cardNumber = cardNumber;  
    }  
  
    @Override  
    public void pay(int amount) {  
        System.out.println("Paid " + amount + "  
using Credit Card: " + cardNumber);  
    }  
}  
  
public class PayPalPayment implements PaymentStrategy {  
    private String email;  
  
    public PayPalPayment(String email) {  
        this.email = email;  
    }  
}
```



```

    }

    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + "
using PayPal: " + email);
    }
}

```

3. Context Class:

The

Context holds a reference to the strategy and interacts with it. This allows for dynamic changes to the strategy at runtime.

```

public class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}

```

4. Usage:

```

public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        // User selects Credit Card payment
        cart.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));
        cart.checkout(100);
    }
}

```

```
// User switches to PayPal payment
    cart.setPaymentStrategy(new PayPalPayment
t("user@example.com"));
    cart.checkout(200);
}
}
```

- **Encapsulation of Algorithms:** In this example, the payment methods (credit card, PayPal) are strategies encapsulated in their own classes. The client (shopping cart) interacts with them through a common interface.
- **Flexibility:** The key benefit is flexibility. You can add new payment methods (strategies) without modifying the shopping cart logic.
- **Open/Closed Principle:** This pattern adheres to the **Open/Closed Principle**, as you can introduce new strategies without changing existing code.
- **Interchangeability:** You can switch between different algorithms at runtime, making the system more adaptable to varying needs.

▼ Benefits of Strategy Pattern

- **Flexibility in Algorithm Selection:**
 - The Strategy Pattern allows you to switch between different algorithms at runtime, giving the system dynamic behavior without altering the client code.
 - **Example:** In a payment processing system, switching between PayPal, credit card, or bank transfer is simple because each payment method is encapsulated as a separate strategy.
- **Code Maintainability:**
 - Strategies are encapsulated, promoting clean separation of concerns and ensuring that each class has a **single responsibility**.
 - Since each algorithm is in its own class, the system becomes more modular and maintainable, making future modifications easier.

- **Adheres to SOLID Principles:**
 - The Strategy Pattern supports **Open/Closed Principle** (you can add new strategies without modifying existing code) and **Single Responsibility Principle** (each strategy focuses on one behavior).
 - This helps reduce the need for large, complex `if-else` or `switch` statements and keeps classes focused and extensible.
- **Improved Testability:**
 - Since each strategy is encapsulated separately, individual strategies can be unit-tested in isolation, which simplifies debugging and improves reliability.
 - This makes it easier to pinpoint issues during testing since the behaviour of each strategy can be independently verified.
- **Reduces Code Duplication:**
 - The Strategy Pattern eliminates the need for redundant conditional logic by delegating algorithm-specific behaviour to strategy classes.
 - By removing duplicate logic and centralising behavior into strategy classes, it simplifies code management and reduces bugs.

▼ Common Pitfalls in Strategy Pattern

- **Increased Number of Classes**
 - Each strategy requires its own class, which can lead to a bloated class hierarchy when you have too many strategies.

```
// Strategy Interface
public interface SortingStrategy {
    void sort(int[] arr);
}

// Concrete Strategy 1: QuickSort
public class QuickSort implements SortingStrategy {
    @Override
```

```

        public void sort(int[] arr) {
            System.out.println("Performing QuickSort");
            // Implement QuickSort logic here
        }
    }

    // Concrete Strategy 2: MergeSort
    public class MergeSort implements SortingStrategy {
        @Override
        public void sort(int[] arr) {
            System.out.println("Performing MergeSort");
            // Implement MergeSort logic here
        }
    }

    // Concrete Strategy 3: BubbleSort (for simplicity's sake)
    public class BubbleSort implements SortingStrategy {
        @Override
        public void sort(int[] arr) {
            System.out.println("Performing BubbleSort");
            // Implement BubbleSort logic here
        }
    }

    // Context
    public class SortingContext {
        private SortingStrategy strategy;

        public void setStrategy(SortingStrategy strategy) {
            this.strategy = strategy;
        }
    }

```

```

    public void sort(int[] arr) {
        strategy.sort(arr);
    }
}

```

- If more strategies are needed (like HeapSort, InsertionSort, etc.), you must create more classes, leading to an explosion in the number of classes.

Solution:

Use lambdas or anonymous classes to reduce the number of individual strategy classes, especially in functional languages.

```

// Example of using lambda for strategy (Java 8 +)
SortingContext context = new SortingContext();
context.setStrategy((arr) -> {
    System.out.println("Lambda: Perform sorting logic");
    // Implement the sorting logic here (QuickSort, MergeSort, etc.)
});
context.sort(new int[]{1, 2, 3});

```

- **Complexity in Context Setup**

- The context class may become complicated due to passing multiple parameters to different strategies.

```

// Example: PaymentContext class passing transaction data to strategies
public class PaymentContext {
    private PaymentStrategy strategy;
    private String accountNumber;
    private double amount;

    public PaymentContext(PaymentStrategy strate

```

```

    gy, String accountNumber, double amount) {
        this.strategy = strategy;
        this.accountNumber = accountNumber;
        this.amount = amount;
    }

    public void executePayment() {
        strategy.processPayment(accountNumber, amount);
    }
}

// PaymentStrategy interface and implementations
public interface PaymentStrategy {
    void processPayment(String accountNumber, double amount);
}

public class PayPalPayment implements PaymentStrategy {
    @Override
    public void processPayment(String accountNumber, double amount) {
        System.out.println("Processing PayPal payment");
    }
}

public class CreditCardPayment implements PaymentStrategy {
    @Override
    public void processPayment(String accountNumber, double amount) {
        System.out.println("Processing Credit Card payment");
    }
}

```

```
// Client code
PaymentContext context = new PaymentContext(new
PayPalPayment(), "12345", 100.0);
context.executePayment();
```

- If `PaymentContext` requires many parameters, it might pass a lot of unnecessary information to certain strategies, complicating the design.

Solution:

Simplify the context by passing only what is essential to each strategy, or use **data objects** to encapsulate parameters.

- **Lack of Awareness of Strategy Capabilities**

- Clients may unknowingly use an inappropriate strategy, leading to performance issues or undesired outcomes.

```
// SortingContext unaware of the performance of
BubbleSort for large arrays
public class SortingContext {
    private SortingStrategy strategy;

    public void setStrategy(SortingStrategy stra
tegy) {
        this.strategy = strategy;
    }

    public void sort(int[] arr) {
        strategy.sort(arr);
    }
}

// Client code chooses BubbleSort without knowin
g its inefficiency for large datasets
SortingContext context = new SortingContext();
context.setStrategy(new BubbleSort()); // Ineff
icient for large arrays
```

```
context.sort(new int[]{5, 2, 9, 1, 5});
```

- Inappropriate use of strategies (e.g., BubbleSort for large datasets) leads to performance issues.

Solution:

The context can be made smarter by adding logic to choose the correct strategy based on the input's characteristics (e.g., input size).

- **Overhead of Strategy Creation**

- If a new strategy object is created every time it's needed, this can lead to performance issues, particularly in resource-constrained environments.

```
// Inefficient: New strategy instance created every time a payment is made
public class PaymentContext {
    public void pay(double amount, String account) {
        PaymentStrategy strategy = new CreditCardPayment();
        strategy.processPayment(account, amount);
    }
}
```

- Repeated creation of strategy instances results in performance overhead and memory waste.

Solution:

Reuse strategy instances where possible or use a singleton pattern for strategies that don't hold state.

```
// Efficient: Reuse the same strategy instance
public class PaymentContext {
```



```

        private PaymentStrategy strategy = new CreditCardPayment(); // Reuse instance

        public void pay(double amount, String account) {
            strategy.processPayment(account, amount);
        }
    }
}

```

- **Tight Coupling in the Context**

- The context may end up being tightly coupled to specific strategies if not designed properly, making future changes difficult.

```

// Problem: Directly calling specific strategy methods from the context
public class PaymentContext {
    private PayPalPayment paypal = new PayPalPayment(); // Tight coupling to PayPalPayment

    public void executePayPalPayment(String account, double amount) {
        paypal.processPayment(account, amount);
    }
}

```

- In this design, adding or switching to new payment methods (like `CreditCardPayment`) requires modifications to the `PaymentContext` class.

Solution:

Keep the context loosely coupled by programming to interfaces and swapping strategies dynamically.

```

public class PaymentContext {
    private PaymentStrategy strategy;
}

```

```
    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void executePayment(String account, double amount) {
        strategy.processPayment(account, amount);
    }
}
```