🧠 What Is Strategy Design Pattern?

> **Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable **at runtime**.

It lets you **choose the algorithm dynamically** without changing the business logic.

The **Strategy Design Pattern** is an elegant and highly used behavioral pattern — especially in **shopping cart**, **payment**, and **algorithm-selection** systems. Let's break it down in **simple language**, and then look at **real Java**, **JDK**, and **Spring Boot** examples.

🛒 Real-Life Example: Shopping Cart Payment System

You have:

- A `ShoppingCart`
- Multiple payment methods:
  - `PayPal`
  - `CreditCard`
  - `UPI`

Each payment method is a **Strategy** that can be injected at runtime.

✅ Key Players in Strategy Pattern

Role	Description	Shopping Cart Example
Strategy	Interface for all strategies	`PaymentStrategy`
ConcreteStrategy	Implements one algorithm	`PayPalStrategy`, `CreditCardStrategy`
Context	Uses a strategy to perform the task	`ShoppingCart`

✅ Java Code – Strategy Pattern in Shopping Cart

1️⃣ Strategy Interface

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

---

2 Concrete Strategies

```
public class PayPalStrategy implements PaymentStrategy {
    private String email;

    public PayPalStrategy(String email) {
        this.email = email;
    }

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

public class CreditCardStrategy implements PaymentStrategy {
    private String cardNumber;

    public CreditCardStrategy(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    public void pay(int amount) {
        System.out.println("Paid ₹" + amount + " using Credit Card (" +
cardNumber + ")");
    }
}
```

---

3 Context – ShoppingCart

```
public class ShoppingCart {
    private List<Integer> items = new ArrayList<>();

    public void addItem(int price) {
        items.add(price);
    }

    public void pay(PaymentStrategy strategy) {
        int total = items.stream().mapToInt(Integer::intValue).sum();
        strategy.pay(total);  // Delegate to selected strategy
    }
}
```

---

4 Client Code

```
java
CopyEdit
public class StrategyDemo {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();
        cart.addItem(1000);
        cart.addItem(2500);

        cart.pay(new PayPalStrategy("user@example.com"));
        cart.pay(new CreditCardStrategy("1234-5678-9012-3456"));
    }
}
```

✅ Output:

```
Paid ₹3500 using PayPal (user@example.com)
Paid ₹3500 using Credit Card (1234-5678-9012-3456)
```

🔬 JDK Internal Use of Strategy Pattern

✅ A. `Comparator<T>` and `Collections.sort()`

```
java
CopyEdit
List<String> names = List.of("Zara", "Amit", "John");

names.sort((a, b) -> a.compareTo(b));  // Strategy = lambda comparator
```

- `Comparator` is the **Strategy**
- `sort()` is the **context**
- You can plug in multiple sorting algorithms via different Comparators

---

✅ B. `java.util.function.Predicate` with `filter()`

```
java
CopyEdit
List<String> names = List.of("Zara", "Amit", "John");
names.stream().filter(name -> name.startsWith("A")).toList();
```

- `Predicate<T>` is the **Strategy**
- `filter()` method is the **Context**
- You inject different strategies to filter data

🚀 Spring Boot – Strategy in Real REST API

✅ Scenario: Payment Gateway Switching via Strategy

Step 1: Define `PaymentStrategy` Interface

```
public interface PaymentStrategy {
    void pay(int amount);
    String getType();  // e.g., "paypal", "card"
}
```

Step 2: Create Implementations

```
@Component
public class UpiPaymentStrategy implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid ₹" + amount + " using UPI");
```

```
    }

    public String getType() {
        return "upi";
    }
}

@Component
public class CardPaymentStrategy implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid ₹" + amount + " using Card");
    }

    public String getType() {
        return "card";
    }
}
```

Step 3: Register Strategies in Context

```
@Service
public class PaymentService {

    private final Map<String, PaymentStrategy> strategies = new HashMap<>();

    @Autowired
    public PaymentService(List<PaymentStrategy> strategyList) {
        for (PaymentStrategy strategy : strategyList) {
            strategies.put(strategy.getType(), strategy);
        }
    }

    public void processPayment(String type, int amount) {
        PaymentStrategy strategy = strategies.get(type.toLowerCase());
        if (strategy == null) throw new RuntimeException("No such payment
type");
        strategy.pay(amount);
    }
}
```

Step 4: REST Controller

```
@RestController
public class PaymentController {

    @Autowired
    private PaymentService paymentService;

    @PostMapping("/pay")
    public String pay(@RequestParam String type, @RequestParam int amount) {
        paymentService.processPayment(type, amount);
        return "Payment processed with " + type;
    }
}
```

✅ Example API Calls:

- `/pay?type=upi&amount=1500` → "Paid ₹1500 using UPI"
- `/pay?type=card&amount=2000` → "Paid ₹2000 using Card"

✅ You added a new strategy (e.g., NetBanking) without touching controller or service code!

🧠 Summary — Strategy Pattern

Concept	Explanation
Purpose	Choose algorithm at **runtime** dynamically
Strength	Avoids many `if-else` or `switch` statements
Pattern Type	Behavioral
JDK Use	`Comparator`, `Predicate`, `sort()`, `filter()`
Spring Use	Strategy injection via `@Component`, `@Autowired`, and REST
Common Uses	Payment, Sorting, Filtering, Validation

Note: Spring Boot automatically creates beans for **all** `@Component` **classes** that implement the `PaymentStrategy` interface and **injects them as a** `List<PaymentStrategy>` into the constructor.

✅ How It Works (Behind the Scenes)

```
@Autowired
public PaymentService(List<PaymentStrategy> strategyList) {
    for (PaymentStrategy strategy : strategyList) {
        strategies.put(strategy.getType(), strategy);
    }
}
```

Here's what Spring does under the hood:

1. It scans for all `@Component`, `@Service`, `@Repository`, etc. that implement `PaymentStrategy`.
2. Spring **creates beans** for:
   - `UpiPaymentStrategy`
   - `CardPaymentStrategy`
   - Any other future strategies.
3. It **collects them all into a** `List<PaymentStrategy>` and injects them into the constructor.

This is called **"auto-wiring by type collection"** — very powerful and **Strategy Pattern–friendly**!

✅ Benefits:

Benefit	Explanation
🔄 Open/Closed Principle	Easily add new strategy classes without touching existing logic.
♻️ Loosely coupled	Service doesn't care *how many* strategies exist.
🧠 Dynamic selection	You can choose a strategy based on runtime parameters (like `"upi"`, `"card"`).

✅ Add Another Strategy (no code change):

```
@Component
public class NetBankingStrategy implements PaymentStrategy {
    public void pay(int amount) {
        System.out.println("Paid ₹" + amount + " using Net Banking");
    }
    public String getType() {
        return "netbanking";
    }
}
```

No changes to:

- `PaymentService`
- `Controller`

✅ It "just works" thanks to Spring's DI + Strategy Pattern combo.
