Week 3 Lab — Strategy Pattern Café POS & Delivery Project

In Week 3, you will extend the Café POS & Delivery System by introducing the Strategy design pattern. In a real café, payment options evolve (cash, cards, wallets, vouchers, QR, loyalty points). The Strategy pattern lets us swap **how we pay** without changing the **what we pay** (your Week-2 Order totals). Today you'll keep the core domain untouched and plug in interchangeable behaviors (PaymentStrategy) that can be extended in future labs without modifying Order. This builds directly on your Week 2 foundation where you implemented core domain objects (Product, Order, and LineItem). *Like last lab, you may work alone or in Pair.*

Before starting this lab, make sure that your Week 2 project is working correctly. You should already have the Money, Product, Order, and LineItem classes implemented. Open the project in IntelliJ IDEA or any IDE of your choice and confirm that the **Project SDK** is set to **Java 21** (that's where solutions are produced). In a terminal, run the command mvn -q -DskipTests compile. If it compiles without errors, you are ready to continue.

Learning Objectives

- Apply the Strategy pattern to support multiple, interchangeable payment methods.
- Program to interfaces rather than concrete implementations.
- Extend the domain model while preserving clean design principles.
- Demonstrate dynamic selection of payment strategies at runtime.

Introduction to the Task

This week you will extend the Café POS system by adding support for **different payment methods**. In real cafés, customers can pay in cash, by card, or with a digital wallet. Instead of writing separate payment logic inside the Order class, you will apply the **Strategy design pattern**, which allows you to define a family of algorithms (payment methods), encapsulate each one, and make them interchangeable. This keeps your code flexible and avoids one large "God method" handling all cases.

Tasks to Perform in this Lab

1. Define the PaymentStrategy interface to represent generic payment behavior.
2. Implement concrete strategies: CashPayment, CardPayment, WalletPayment.
3. Extend the Order class to accept a PaymentStrategy and delegate the payment behavior.
4. Implement a CLI demo (Week3Demo) that creates orders and pays them with different strategies.
5. Write JUnit tests to confirm that payment strategies are invoked correctly.

This lab builds directly on your Week-2 domain (Order, LineItem, Product, Money). In the coming weeks you will follow the same approach: adding new capabilities while keeping the

core stable. The Strategy pattern here shows how to extend behavior without changing existing classes — for example, later you could add LoyaltyPointsPayment or GiftCardPayment by writing new strategy classes instead of editing Order.

Step-by-Step Instructions/Guidelines for Each Task

1. **Step1: Define the PaymentStrategy Interface**

PaymentStrategy is a small interface that models how an order is paid. Order shouldn't know about card numbers or wallet IDs. By delegating to a strategy, Order stays stable while payment options can evolve independently. This also makes testing easier: we can inject a fake strategy to verify delegation. Following are the fine-level steps needs to be performed.

- Create a new package called payment.
- Inside it, define an interface PaymentStrategy with a single method:
- void pay(Order order);
- This interface is the contract that all payment methods must follow.

To implement these, you will introduce a new package called payment. Within this package, define an interface and concrete implementations. Below is the code for `PaymentStrategy` to help you.

PaymentStrategy.java

```
public interface PaymentStrategy {
    void pay(Order order);
}
```

2. **Step2: Implement the CashPayment Strategy**

Each concrete strategy prints a different confirmation message but never changes totals (CashPayment, is not different). This demonstrates polymorphism (same call, different behavior) without leaking payment details into Order.

- Create the class CashPayment that implements PaymentStrategy.
- When called, it should print a message such as:
- [Cash] Customer paid 9.35 EUR
- Note that this strategy does not modify the order total; it only represents how the payment is made.

Consider the following code for `CashPayment` an implementation of `PaymentStrategy:`

CashPayment.java

```
public final class CashPayment implements PaymentStrategy {
    @Override
    public void pay(Order order) {
        System.out.println("[Cash] Customer paid " +
order.totalWithTax(10) + " EUR");
    }
}
```

3. **Step3: Implement the CardPayment Strategy**

- Create the class CardPayment with a constructor that accepts a card number.

- Inside the pay method, mask all digits except the last four.
- The message should look like:
- [Card] Customer paid 9.35 EUR with card ****1234

CardPayment.java

```
public final class CardPayment implements PaymentStrategy {
    private final String cardNumber;
    public CardPayment(String cardNumber) { ... }
    @Override
    public void pay(Order order) {
        // mask card and print payment confirmation
    }
}
```

4. **Step4: Implement the WalletPayment Strategy**
   - Create the class WalletPayment with a constructor that takes a wallet ID (for example, "alice-wallet-01").
   - Its pay method should confirm payment using that wallet ID, e.g.:
   - [Wallet] Customer paid 9.35 EUR using wallet alice-wallet-01

Consider the following WalletPayment class and fill-in the required code:

WalletPayment.java

```
public final class WalletPayment implements PaymentStrategy {
    private final String walletId;
    public WalletPayment(String walletId) { ... }
    @Override
    public void pay(Order order) {
        System.out.println("[Wallet] Customer paid " +
order.totalWithTax(10) + " EUR via wallet " + walletId);
```

5. **Update the Order Class**

Order.pay(strategy) should only delegate; it must not format messages or handle payment policies. Keep it thin and enforce a non-null strategy (fail fast with a clear error). This preserves Order as the domain aggregate, not a payment orchestrator. You need to do following:

- Add a method pay(PaymentStrategy strategy) to Order.
- This method should check that the strategy is not null, then delegate the payment process to the strategy.
- The order itself should not care *how* the payment is done, only that it is done.

Order.java (extension)

```
public void pay(PaymentStrategy strategy) {
    if (strategy == null) throw new
IllegalArgumentException("strategy required");
```

```
    strategy.pay(this);
}
```

6. **Run the Week3Demo**
    - In the demo class, create one or more orders, add items, and then pay them using different strategies.
    - For example, try one order with cash and another with card.
    - You should notice that the order total remains unchanged, but the payment messages vary depending on the strategy chosen.

Demo Expectations

By the end of this lab, you should be able to run Week3Demo. The demo creates sample orders and pays them using different payment strategies. The output should demonstrate that while the order totals remain unchanged, the payment behavior varies depending on the strategy selected.

Week3Demo.java

```
public final class Week3Demo {
    public static void main(String[] args) {
        Catalog catalog = new InMemoryCatalog();
        catalog.add(new SimpleProduct("P-ESP", "Espresso",
Money.of(2.50)));
        catalog.add(new SimpleProduct("P-CCK", "Chocolate
Cookie", Money.of(3.50)));

        // Cash payment
        Order order1 = new Order(OrderIds.next());
        order1.addItem(new LineItem(catalog.findById("P-
ESP").orElseThrow(), 2));
        order1.addItem(new LineItem(catalog.findById("P-
CCK").orElseThrow(), 1));
        System.out.println("Order #" + order1.id() + " Total: " +
order1.totalWithTax(10));
        order1.pay(new CashPayment());

        // Card payment
        Order order2 = new Order(OrderIds.next());
        order2.addItem(new LineItem(catalog.findById("P-
ESP").orElseThrow(), 2));
        order2.addItem(new LineItem(catalog.findById("P-
CCK").orElseThrow(), 1));
        System.out.println("Order #" + order2.id() + " Total: " +
order2.totalWithTax(10));
        order2.pay(new CardPayment("1234567812341234"));
    }
}
```

Lab coordinator will expect the following type of output as an outcome of this lab:

```
Order #1002 Total: 9.35
[Cash] Customer paid 9.35 EUR
Order #1003 Total: 9.35
[Card] Customer paid 9.35 EUR with card ****1234
```

Testing

Good design requires testing. To verify your implementation:

1. In your test folder, create a fake PaymentStrategy class that sets a boolean flag when pay is called.
2. Write a unit test where you create an order, call order.pay(fakeStrategy), and then assert that the flag is true.
3. This proves that Order delegates correctly to the chosen strategy.
4. Run all your tests in IntelliJ or via the terminal with mvn -q test. You should see a green bar.

You should write JUnit tests to verify that payment strategies are invoked correctly. For instance, you can create a fake PaymentStrategy to record whether it has been called. Here is an example:

```
@Test void payment_strategy_called() {
    var p = new SimpleProduct("A", "A", Money.of(5));
    var order = new Order(42);
    order.addItem(new LineItem(p, 1));
    final boolean[] called = {false};
    PaymentStrategy fake = o -> called[0] = true;
    order.pay(fake);
    assertTrue(called[0], "Payment strategy should be called");
}
```

Reflection Prompts: Answers Need to Submit

At the end of the lab, write a short reflection (2–3 sentences) answering the following questions:

1. Point to one if/else you avoided in Order by introducing PaymentStrategy?
2. Show (1–2 sentences) how the same Order can be paid by different strategies in the demo without changing Order code?

Deliverables

By the end of this lab, you must submit:
1. The extended codebase with the payment package and all three strategy implementations.
2. The Week3Demo run showing Cash and Card payment outputs.
3. All JUnit test cases passing.
4. An updated “Class diagram” the classes introduced in this lab (lab coordinator will check).

Submit deliverables in one zip file name after your IDs(Pair or Individual).