Week 5 Lab — Decorator + Factory Café POS & Delivery Project

In this week you extend the *Café POS system* by introducing two new design patterns, Decorator and Factory. The Decorator pattern allows you to add optional features to a base product without changing the original class. The Factory pattern gives you a central place to construct products, so that other parts of the program do not need to know the details of constructors or decoration recipes. Together these two patterns reinforce the Open/Closed Principle: you can add new add-ons or recipes by writing new classes or tokens, not by editing existing code.

Before starting, make sure your Week 2–4 code compiles and runs correctly. You should already have Money, Product, SimpleProduct, Catalog (with InMemoryCatalog), Order, LineItem, the payment strategies from Week 3, and the observer wiring from Week 4. Confirm that the project SDK is set to Java 21, and run mvn -q -DskipTests compile in your terminal to ensure you are starting from a clean state.

Important Notes

Per module policy, the Week 5 lab session will **assess Week 4 work** (Observer Pattern). However, during Week 5 you still complete the Decorator + Factory tasks described here; they will be assessed in Week 6.

Learning Objectives

- Apply the Decorator pattern to compose add-ons around existing products without modifying base classes.
- Design a small Factory that creates products (and decorated products) from short codes/recipes.
- Preserve OCP and program to interfaces.
- Produce a CLI demo and unit tests proving behavior and pricing are correct.

Prerequisites (from Weeks 2–4)

You should already have: Money, Product, SimpleProduct, Catalog (in-memory), Order, LineItem, and the Week 3 payment strategy + Week 4 observer changes.

Step-by-Step Instructions to Complete this Lab

Step 1 — Create a Decorator base type

The first step is to create a base class for all decorators. The abstract ProductDecorator implements the Product interface and wraps another Product. The constructor enforces that the wrapped product is never null. The id and basePrice methods simply delegate to the wrapped product. Concrete decorators will override the name method to add their label, and they will provide a price method that adds their surcharge to the base price.

```
package com.cafepos.decorator;

import com.cafepos.catalog.Product;
import com.cafepos.common.Money;
public abstract class ProductDecorator implements Product {
    protected final Product base;
    protected ProductDecorator(Product base) {
        if (base == null) throw new
IllegalArgumentException("base product required");
        this.base = base;
    }
    @Override public String id() { return base.id(); } //
id may remain the base product id
    @Override public Money basePrice() { return
base.basePrice(); } // original price (not total)
    // Concrete decorators will override name() and provide
a finalPrice() helper if desired.
}
```

Step 2 — Implement concrete decorators (add-ons)

Once the base decorator is ready, you can implement concrete add-ons. For example, ExtraShot adds 0.80, OatMilk adds 0.50, Syrup adds 0.40, and SizeLarge adds 0.70. Each of these classes extends ProductDecorator, holds its surcharge, overrides name() to append the label, and calculates the final price as the base price plus the surcharge. The important part is that each decorator behaves like a product, so you can wrap them around each other. This makes it possible to chain them together: an espresso wrapped with an extra shot, then oat milk, then large size, becomes “Espresso + Extra Shot + Oat Milk (Large)” with the base price plus all three surcharges. Following code should do adds-on work:

```
package com.cafepos.decorator;

import com.cafepos.catalog.Product;
import com.cafepos.common.Money;

public final class ExtraShot extends ProductDecorator {
    private static final Money SURCHARGE = Money.of(0.80);
    public ExtraShot(Product base) { super(base); }
    @Override public String name() { return base.name() + "
+ Extra Shot"; }
    public Money price() { return (base instanceof Priced p
? p.price() : base.basePrice()).add(SURCHARGE);  }
}

public final class OatMilk extends ProductDecorator {
    private static final Money SURCHARGE = Money.of(0.50);
    public OatMilk(Product base) { super(base); }
    @Override public String name() { return base.name() + "
+ Oat Milk"; }
    public Money price() { return (base instanceof Priced p
```

```
? p.price() : base.basePrice()).add(SURCHARGE);  }
}

public final class Syrup extends ProductDecorator {
    private static final Money SURCHARGE = Money.of(0.40);
    public Syrup(Product base) { super(base); }
    @Override public String name() { return base.name() + "
+ Syrup"; }
    public Money price() { return (base instanceof Priced p
? p.price() : base.basePrice()).add(SURCHARGE);  }
}

public final class SizeLarge extends ProductDecorator {
    private static final Money SURCHARGE = Money.of(0.70);
    public SizeLarge(Product base) { super(base); }
    @Override public String name() { return base.name() + "
(Large)"; }
    public Money price() { return (base instanceof Priced p
? p.price() : base.basePrice()).add(SURCHARGE);  }
}
```

Each decorator's final unit price should reuse your existing Money.add operation. Do not reconstruct Money from internal fields; simply take the wrapped product's unit price and add the surcharge. If your solution uses a Priced interface, the decorator's price() should return the wrapped price() plus its own surcharge. If your solution exposes only basePrice(), keep the same idea but be consistent with the interface you chose in Week 2.

Step 3 — Allow stacked decorations (chaining)

Demonstrate that add-ons can be stacked. For example: new SizeLarge(new OatMilk(new ExtraShot(espresso))). Each decorator should compute price() using its wrapped product's price (or basePrice if you keep base semantics). See example code below:

```
// Example usage
Product espresso = new SimpleProduct("P-ESP", "Espresso",
Money.of(2.50));
Product decorated = new SizeLarge(new OatMilk(new
ExtraShot(espresso)));
// decorated.name() => "Espresso + Extra Shot + Oat Milk (Large)"
// decorated.price() => 2.50 + 0.80 + 0.50 + 0.70 = 4.50
```

Surcharges add commutatively, so changing the order of wrappers does not change the numeric total; only the rendered name() string changes.

Step 4 — Integrate decorators with Orders

The next step is to integrate decorators with orders. Your LineItem should not just multiply the base price. Instead it should use the decorated price when available. A common way to do this is to introduce a small interface such as Priced with a price() method. Both SimpleProduct and all decorators implement this interface. In LineItem.lineTotal(), check if

the product is Priced, and if so use price(); otherwise fall back to basePrice(). This ensures that decorated products are always charged correctly without adding complex instanceof chains. See solution below:

```
// Introduce a new interface for pricing
public interface Priced {
    Money price();
}
// Make SimpleProduct implement Priced (price() == basePrice())
// Make all decorators implement Priced (price() == basePrice() +
surcharges)
// In LineItem:
public Money lineTotal() {
    Money unit = (product instanceof Priced p) ? p.price() :
product.basePrice();
    return unit.multiply(quantity);
}
```

Step 5 — Implement a Product Factory (recipes)

With decorators working, you can now add the factory. The ProductFactory accepts a recipe string such as ESP+SHOT+OAT+L. The first token chooses the base product (ESP for Espresso, LAT for Latte, CAP for Cappuccino). Each subsequent token applies a decorator in order: SHOT for extra shot, OAT for oat milk, SYP for syrup, and L for large size. The factory returns the fully decorated product. This design means that no other class needs to know how to construct a decorated product; they just ask the factory to create it.

```
package com.cafepos.factory;
import com.cafepos.catalog.*;
import com.cafepos.common.Money;
import com.cafepos.decorator.*;

public final class ProductFactory {
    public Product create(String recipe) {
        if (recipe == null || recipe.isBlank()) throw new
IllegalArgumentException("recipe required");
String[] raw = recipe.split("\\+"); // literal '+'
String[] parts = java.util.Arrays.stream(raw)
        .map(String::trim)
        .map(String::toUpperCase)
        .toArray(String[]::new);
        Product p = switch (parts[0]) {
            case "ESP" -> new SimpleProduct("P-ESP", "Espresso",
Money.of(2.50));
            case "LAT" -> new SimpleProduct("P-LAT", "Latte",
Money.of(3.20));
            case "CAP" -> new SimpleProduct("P-CAP",
"Cappuccino", Money.of(3.00));
            default -> throw new
IllegalArgumentException("Unknown base: " + parts[0]);
```

```
        };
        for (int i = 1; i < parts.length; i++) {
            p = switch (parts[i]) {
                case "SHOT" -> new ExtraShot(p);
                case "OAT"  -> new OatMilk(p);
                case "SYP"  -> new Syrup(p);
                case "L"    -> new SizeLarge(p);
                default -> throw new
IllegalArgumentException("Unknown addon: " + parts[i]);
            };
        }
        return p;
    }
}
```

Always trim and uppercase tokens so that esp+shot+oat and ESP + SHOT + OAT behave the same; fail fast with a clear message on unknown tokens.

Step 6 — CLI Demo (Week5Demo)

To prove that everything works, write a CLI demo called Week5Demo. In the demo, use the factory to create two drinks: an espresso with extra shot and oat milk, and a large latte. Add them to an order, print each line with the product name and line total, and then print the subtotal, tax, and total. If everything is correct, your receipt will show the add-on names and the correct prices. Note: you may reuse your tax calculation from Week 2. Example flow below.

```
public final class Week5Demo {
    public static void main(String[] args) {
        ProductFactory factory = new ProductFactory();
        Product p1 = factory.create("ESP+SHOT+OAT"); // Espresso
+ Extra Shot + Oat
        Product p2 = factory.create("LAT+L");      // Large Latte
        Order order = new Order(OrderIds.next());
        order.addItem(new LineItem(p1, 1));
        order.addItem(new LineItem(p2, 2));
        System.out.println("Order #" + order.id());
        for (LineItem li : order.items()) {
            System.out.println(" - " + li.product().name() + " x"
+ li.quantity() + " = " + li.lineTotal());
        }
        System.out.println("Subtotal: " + order.subtotal());
        System.out.println("Tax (10%): " +
order.taxAtPercent(10));
        System.out.println("Total: " + order.totalWithTax(10));
    }
}
```

This demo proves that the Week 2 order math still works unchanged while Decorator and Factory only affect unit names and unit prices.

Step 7 — Tests

Focus the tests on two properties: adding a supported decorator increases the unit price by exactly its surcharge, and the numeric total is independent of decoration order even though the name() text may change. These properties keep your design stable when you add new add-ons later. Add following unit tests that verify:

1) Single decorator changes price and name as expected.
2) Multiple decorators stack correctly (associative addition of surcharges).
3) Factory parses recipes and returns correctly decorated products.
4) Order totals use the decorated price.

```
@Test void decorator_single_addon() {
    Product espresso = new SimpleProduct("P-ESP", "Espresso",
Money.of(2.50));
    Product withShot = new ExtraShot(espresso);
    assertEquals("Espresso + Extra Shot", withShot.name());
    // if using Priced interface:
    assertEquals(Money.of(3.30), ((Priced) withShot).price());
}

@Test void decorator_stacks() {
    Product espresso = new SimpleProduct("P-ESP", "Espresso",
Money.of(2.50));
    Product decorated = new SizeLarge(new OatMilk(new
ExtraShot(espresso)));
    assertEquals("Espresso + Extra Shot + Oat Milk (Large)",
decorated.name());
    assertEquals(Money.of(4.50), ((Priced) decorated).price());
}

@Test void factory_parses_recipe() {
    ProductFactory f = new ProductFactory();
    Product p = f.create("ESP+SHOT+OAT");
    assertTrue(p.name().contains("Espresso") &&
p.name().contains("Oat Milk"));
}

@Test void order_uses_decorated_price() {
    Product espresso = new SimpleProduct("P-ESP", "Espresso",
Money.of(2.50));
    Product withShot = new ExtraShot(espresso); // 3.30
    Order o = new Order(1);
    o.addItem(new LineItem(withShot, 2));
    assertEquals(Money.of(6.60), o.subtotal());
}
```

Activity: Factory vs. Manual Chaining—Prove They Build the Same Drink

Create a new test class under src/test/java. First, build a drink via the factory: viaFactory = new ProductFactory().create("ESP+SHOT+OAT+L"). Second, build the same drink by manual wrapping: viaManual = new SizeLarge(new OatMilk(new ExtraShot(new SimpleProduct("P-ESP","Espresso", Money.of(2.50))))). Assert that viaFactory.name() equals viaManual.name(). If your solution uses a Priced interface, cast both to Priced and assert that the unit prices are equal. Create two separate orders, add each product with quantity one, and assert the subtotals and totalWithTax(10) are equal. In your README, write three to four sentences explaining which construction approach you would expose to application developers and why.

Consider the following as expected outcomes

The coordinator's 30-second check is to run the demo and see decorated names on each line with the correct totals. To show your work to the coordinator in the last ten minutes of lab, compile and run the demo from the project root. On macOS or Linux, you can run

mvn -q -DskipTests -Dexec.mainClass=com.cafepos.demo.Week5Demo exec:java

On Windows PowerShell, use the command:

***Compile**: mvn -q -DskipTests compile*
***Run**: java -cp "target/classes" com.cafepos.demo.Week5Demo*

The coordinator will expect to see an order printed with decorated names and correct totals. A sample output might be:

```
Order #2001
 - Espresso + Extra Shot + Oat Milk x1 = 3.80
 - Latte (Large) x2 = 7.80
Subtotal: 11.60
Tax (10%): 1.16
Total: 12.76
```

Reflection and Deliverables

Finally, reflect briefly on the design. Explain why you chose your pricing interface, identify one place where you preserved the Open/Closed Principle, and describe what would be required to add a new add-on next week. If you added any new tokens to the factory during experimentation, explain how your change preserved the Open/Closed Principle.

Your deliverables are the code for decorators and the product factory, a console run of the demo showing correct output, passing unit tests, and a short reflection. You must need to provide the updated Class Diagram with these deliverables (class diagram will be assessed by lab coordinator)
Reminder: Week 5 lab marks cover Week 4 outputs; your Week 5 work will be assessed in Week 6.