Week 6 Lab — Refactoring Code Smells (Smelly → Clean) Café POS & Delivery Project

In Week 6, your task is to refactor an intentionally "smelly" module into a clean, testable, and extensible design. You will lock the current behavior with characterization tests, identify specific smells (God Class, Long Method, Primitive Obsession, Duplicated Code, Feature Envy, Shotgun Surgery risk), and then apply refactorings aligned with SOLID principles. This lab builds directly on your Week 2–5 code: Money, Product/Catalog, Order/LineItem, Strategy (payments), Observer (notifications), and Decorator + Factory (recipes/add-ons).

This week does not add new features. You are learning to protect behavior (via characterization tests) and then improve the design underneath. You must keep outputs identical to the "smelly" version. Everything you extracted (discounts, tax, receipt formatting, payment delegation) will be assessed in the midterm (Week 7) and re-used in final project. Think of today as building the "clean seams" your future changes will plug into.

Important Assessment Note

Per module policy, the **Week 6 lab session assesses Week 5 outputs** (Decorator + Factory). You still complete the Week 6 refactoring work during this week; it will be part of your **Week 7 midterm** review.

Learning Objectives

- Recognize code smells and articulate their risks.
- Write characterization tests to preserve behavior during refactoring.
- Apply core refactorings: Extract Class, Extract Method, Introduce Strategy, Replace Conditional with Polymorphism, Introduce Interface, Dependency Injection, Remove Global/Static State.
- Improve design to honor design principles.

Setup & Context

You are given a single large class `OrderManagerGod` that mixes domain logic (pricing, tax, discounts), I/O (receipt printing), and orchestration (payment switching). Your mission is to refactor it to:

• Separate concerns into cohesive classes/interfaces
• Reuse your existing Week 3 `PaymentStrategy` and Week 5 `ProductFactory`/decorators
• Make behavior easy to extend without editing the core class
• Keep the **externally visible behavior identical** (verified by tests)

Use your Java 21 toolchain. From the project root, verify a clean baseline with:

```
mvn -q -DskipTests compile
```

All steps in this lab keep the **public behavior identical**. Never delete or edit *OrderManagerGod* before your characterization tests pass; keep it as a reference while you refactor behind tests. **Do not change any printed labels or spacing** (e.g., “Tax (10%): 0.74”). Your refactor must keep the exact text.

Given “Smelly” Starting Point (read-only input)

Study the following implementation carefully. Your first step will be to write tests that “pin down” its behavior. Do NOT refactor until you have tests. Copy this file into your project under `src/main/java/com/cafepos/smells/OrderManagerGod.java`.

```java
package com.cafepos.smells;

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

public class OrderManagerGod {
    public static int TAX_PERCENT = 10;
    public static String LAST_DISCOUNT_CODE = null;

    public static String process(String recipe, int qty, String
paymentType, String discountCode, boolean printReceipt) {
        ProductFactory factory = new ProductFactory();
        Product product = factory.create(recipe);

        Money unitPrice;
        try {
            var priced = product instanceof com.cafepos.decorator.Priced
p ? p.price() : product.basePrice();
            unitPrice = priced;
        } catch (Exception e) {
            unitPrice = product.basePrice();
        }

        if (qty <= 0) qty = 1;
        Money subtotal = unitPrice.multiply(qty);

        Money discount = Money.zero();
        if (discountCode != null) {
            if (discountCode.equalsIgnoreCase("LOYAL5")) {
                discount = Money.of(subtotal.asBigDecimal()
                        .multiply(java.math.BigDecimal.valueOf(5))
                        .divide(java.math.BigDecimal.valueOf(100)));
            } else if (discountCode.equalsIgnoreCase("COUPON1")) {
                discount = Money.of(1.00);
            } else if (discountCode.equalsIgnoreCase("NONE")) {
                discount = Money.zero();
            } else {
                discount = Money.zero();
            }
            LAST_DISCOUNT_CODE = discountCode;
        }
        Money discounted =
Money.of(subtotal.asBigDecimal().subtract(discount.asBigDecimal()));
        if (discounted.asBigDecimal().signum() < 0) discounted =
Money.zero();

        var tax = Money.of(discounted.asBigDecimal()
                .multiply(java.math.BigDecimal.valueOf(TAX_PERCENT))
                .divide(java.math.BigDecimal.valueOf(100)));

        var total = discounted.add(tax);

        if (paymentType != null) {
            if (paymentType.equalsIgnoreCase("CASH")) {
                System.out.println("[Cash] Customer paid " + total + "
EUR");
            } else if (paymentType.equalsIgnoreCase("CARD")) {
                System.out.println("[Card] Customer paid " + total + "
EUR with card ****1234");
            } else if (paymentType.equalsIgnoreCase("WALLET")) {
                System.out.println("[Wallet] Customer paid " + total + "
EUR via wallet user-wallet-789");
            } else {
                System.out.println("[UnknownPayment] " + total);
            }
        }

        StringBuilder receipt = new StringBuilder();
        receipt.append("Order (").append(recipe).append(")
x").append(qty).append("\n");
        receipt.append("Subtotal: ").append(subtotal).append("\n");
        if (discount.asBigDecimal().signum() > 0) {
            receipt.append("Discount: -").append(discount).append("\n");
        }
        receipt.append("Tax (").append(TAX_PERCENT).append("%):
").append(tax).append("\n");
        receipt.append("Total: ").append(total);
        String out = receipt.toString();

        if (printReceipt) {
            System.out.println(out);
        }
        return out;
    }
}
```

Step 1 — Write Characterization Tests (lock current behavior)

Before refactoring, write tests that capture what the smelly code does (even if you dislike it). These tests will prevent accidental behavior changes while you clean the design. Place the following in `src/test/java/com/cafepos/Week6CharacterizationTests.java` and make sure they pass.

```
package com.cafepos;

import com.cafepos.smells.OrderManagerGod;
```

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class Week6CharacterizationTests {

    @Test void no_discount_cash_payment() {
        String receipt = OrderManagerGod.process("ESP+SHOT+OAT", 1,
"CASH", "NONE", false);
        assertTrue(receipt.startsWith("Order (ESP+SHOT+OAT) x1"));
        assertTrue(receipt.contains("Subtotal: 3.80"));
        assertTrue(receipt.contains("Tax (10%): 0.38"));
        assertTrue(receipt.contains("Total: 4.18"));
    }

    @Test void loyalty_discount_card_payment() {
        String receipt = OrderManagerGod.process("LAT+L", 2, "CARD",
"LOYAL5", false);
        // Latte (Large) base = 3.20 + 0.70 = 3.90, qty 2 => 7.80
        // 5% discount => 0.39, discounted=7.41; tax 10% => 0.74;
total=8.15
        assertTrue(receipt.contains("Subtotal: 7.80"));
        assertTrue(receipt.contains("Discount: -0.39"));
        assertTrue(receipt.contains("Tax (10%): 0.74"));
        assertTrue(receipt.contains("Total: 8.15"));
    }

    @Test void coupon_fixed_amount_and_qty_clamp() {
        String receipt = OrderManagerGod.process("ESP+SHOT", 0, "WALLET",
"COUPON1", false);
        // qty=0 clamped to 1; Espresso+SHOT = 2.50 + 0.80 = 3.30;
coupon1 => -1 => 2.30; tax=0.23; total=2.53
        assertTrue(receipt.contains("Order (ESP+SHOT) x1"));
        assertTrue(receipt.contains("Subtotal: 3.30"));
        assertTrue(receipt.contains("Discount: -1.00"));
        assertTrue(receipt.contains("Tax (10%): 0.23"));
        assertTrue(receipt.contains("Total: 2.53"));
    }
}
```

Run `*mvn -q test*`. If tests fail, ***do not refactor yet***. If your Money.toString() renders with two decimals (as in Weeks 2–5), the string assertions above will pass exactly. If your local Money renders differently (e.g., currency symbols), write assertions on substrings (as shown) rather than full-line equality. Also, do not “fix” the smelly code yet; the point of Step 1 is to lock in whatever it currently does—even when you disagree with it.

Step 2 — Identify & Label Smells

Open OrderManagerGod.process(...) and add inline comments in your branch **at the exact lines** that exhibit each smell. For example, put // Primitive Obsession on the discountCode branch and // Shotgun Surgery risk on each tax computation line. Your commit should show these comments; they will be graded in the midterm as evidence you can **identify** smells before refactoring them. Annotate (as comments in your branch) where these smells occur:

– God Class & Long Method: One method performs creation, pricing, discounting, tax, payment I/O, and printing.
– Primitive Obsession: `discountCode` strings; `TAX_PERCENT` as primitive; magic numbers for rates.
– Duplicated Logic: Money and BigDecimal manipulations scattered inline.
– Feature Envy / Shotgun Surgery: Tax/discount rules embedded inline; any change requires editing this method.
– Global/Static State: `LAST_DISCOUNT_CODE`, `TAX_PERCENT` are global — risky and hard to test.

Step 3 — Refactor in Small, Safe Steps

Apply these steps, running tests after each change:
1) **Extract DiscountPolicy** — move discount computation out of `OrderManagerGod`. Use polymorphism.
2) **Extract TaxPolicy** — compute tax via a small interface; remove inline math.
3) **Extract ReceiptPrinter** — move string formatting & printing out of core logic.
4) **Replace paymentType string switch** with a call to an injected `PaymentStrategy`.
5) **Inject dependencies** (factory, discount policy, tax policy, printer, payment strategy) via constructor.
6) **Delete global state** (`LAST_DISCOUNT_CODE`, `TAX_PERCENT`).

```
// Example interfaces you create (put them under com.cafepos.pricing or
com.cafepos.checkout)

package com.cafepos.pricing;
import com.cafepos.common.Money;
public interface DiscountPolicy {
    Money discountOf(Money subtotal);
}

package com.cafepos.pricing;
import com.cafepos.common.Money;
public final class NoDiscount implements DiscountPolicy {
    @Override public Money discountOf(Money subtotal) { return
Money.zero(); }
}

package com.cafepos.pricing;
import com.cafepos.common.Money;
public final class LoyaltyPercentDiscount implements DiscountPolicy {
    private final int percent;
    public LoyaltyPercentDiscount(int percent) { if (percent < 0) throw
new IllegalArgumentException(); this.percent = percent; }
    @Override public Money discountOf(Money subtotal) {
        var d =
subtotal.asBigDecimal().multiply(java.math.BigDecimal.valueOf(percent))
                .divide(java.math.BigDecimal.valueOf(100));
        return Money.of(d);
    }
```

```
}

package com.cafepos.pricing;
import com.cafepos.common.Money;
public final class FixedCouponDiscount implements DiscountPolicy {
    private final Money amount;
    public FixedCouponDiscount(Money amount) { this.amount = amount; }
    @Override public Money discountOf(Money subtotal) {
        // cap at subtotal
        if (amount.asBigDecimal().compareTo(subtotal.asBigDecimal()) > 0)
return subtotal;
        return amount;
    }
}

public interface TaxPolicy { Money taxOn(Money amount); }

package com.cafepos.pricing;
import com.cafepos.common.Money;
public final class FixedRateTaxPolicy implements TaxPolicy {
    private final int percent;
    public FixedRateTaxPolicy(int percent) { if (percent < 0) throw new
IllegalArgumentException(); this.percent = percent; }
    @Override public Money taxOn(Money amount) {
        var t =
amount.asBigDecimal().multiply(java.math.BigDecimal.valueOf(percent))
                .divide(java.math.BigDecimal.valueOf(100));
        return Money.of(t);
    }
}
public final class PricingService {
    private final DiscountPolicy discountPolicy;
    private final TaxPolicy taxPolicy;

    public PricingService(DiscountPolicy discountPolicy, TaxPolicy
taxPolicy) {
        this.discountPolicy = discountPolicy;
        this.taxPolicy = taxPolicy;
    }

    public PricingResult price(Money subtotal) {
        Money discount = discountPolicy.discountOf(subtotal);
        Money discounted =
Money.of(subtotal.asBigDecimal().subtract(discount.asBigDecimal()));
        if (discounted.asBigDecimal().signum() < 0) discounted =
Money.zero();
        Money tax = taxPolicy.taxOn(discounted);
        Money total = discounted.add(tax);
        return new PricingResult(subtotal, discount, tax, total);
    }

    public static record PricingResult(Money subtotal, Money discount,
Money tax, Money total) {}
}
public final class ReceiptPrinter {
    public String format(String recipe, int qty,
PricingService.PricingResult pr, int taxPercent) {
        StringBuilder receipt = new StringBuilder();
        receipt.append("Order (").append(recipe).append(")
x").append(qty).append("\n");
        receipt.append("Subtotal: ").append(pr.subtotal()).append("\n");
        if (pr.discount().asBigDecimal().signum() > 0) {
            receipt.append("Discount: -
").append(pr.discount()).append("\n");
        }
        receipt.append("Tax (").append(taxPercent).append("%):
").append(pr.tax()).append("\n");
        receipt.append("Total: ").append(pr.total());
        return receipt.toString();
    }
}
```

Keep the printed tax rate and the applied tax rate **the same**. Either (a) inject the same percent into both FixedRateTaxPolicy and the format(..., taxPercent) call, or (b) add a getter on the policy and print that value. **Do not** change the printed label text.

After each micro-change, commit with a message that names the smell and the refactoring, for example:

- test: add characterization tests for OrderManagerGod
- refactor(discount): Extract Class DiscountPolicy (behavior preserved)
- refactor(tax): Extract Class FixedRateTaxPolicy (behavior preserved)
- refactor(io): Extract ReceiptPrinter (behavior preserved)
- refactor(payment): Replace Conditional with Polymorphism (PaymentStrategy)
- refactor(di): Constructor Injection; remove global TAX_PERCENT/LAST_DISCOUNT_CODE

This sequence will be checked in the midterm.

Activity: Characterize Discounts, Then Extract Them Create *DiscountPolicyCharacterizationTests* that call *OrderManagerGod.process(...)* with LOYAL5, COUPON1, and an unknown code, asserting the presence/absence of the "Discount:" line and the final total for each case. Then implement NoDiscount, LoyaltyPercentDiscount(5), and FixedCouponDiscount(1.00) and write isolated unit tests for each policy (no I/O, no receipt). You pass the activity when your isolated policy tests are green and swapping your PricingService into CheckoutService still yields exactly the same receipt text as OrderManagerGod for those three scenarios. This proves you can first lock behavior, then extract it without changing outputs.

Step 4 — Replace OrderManagerGod with a small orchestrator

Create a new class `CheckoutService` that does orchestration only: create product via `ProductFactory`, compute subtotal from `Priced.price()` or `basePrice()`, delegate pricing to `PricingService`, delegate payment to a `PaymentStrategy`, and delegate formatting to `ReceiptPrinter`. Keep the public method signature simple.

Payment must reuse your Week-3 PaymentStrategy (Cash, Card, Wallet). Do not add new strings or switches. If your strategies print based on an Order, pass the real order there; if they print based on totals, use a tiny adapter that calls the strategy after pricing.

Keep the CheckoutService code as you have it. Immediately above the line where you call payment.pay(...), add these two comment lines in the snippet (students will see how to adapt to their own Week-3 signature):

```
// Adapt to your Week-3 signature; if your strategy expects an Order,
pass the real one here.
// If your strategy prints based on totals, wrap in a tiny adapter and
call after pricing.

public final class CheckoutService {
    private final ProductFactory factory;
    private final PricingService pricing;
    private final ReceiptPrinter printer;
    private final int taxPercent;

    public CheckoutService(ProductFactory factory, PricingService
pricing, ReceiptPrinter printer, int taxPercent) {
        this.factory = factory;
        this.pricing = pricing;
        this.printer = printer;
        this.taxPercent = taxPercent;
    }

    public String checkout(String recipe, int qty) {
        Product product = factory.create(recipe);
        if (qty <= 0) qty = 1;
        Money unit = (product instanceof com.cafepos.decorator.Priced p)
? p.price() : product.basePrice();
        Money subtotal = unit.multiply(qty);
        var result = pricing.price(subtotal);
        return printer.format(recipe, qty, result, taxPercent);
    }
}
```

Note: Wire your real `Order` and payment flow as in Week 3. The snippet above shows delegation only; you should integrate with your existing `Order` entity if needed (e.g., to record payments).

Step 5 — Make Tests Pass After Refactor

Re-run your characterization tests. They should still pass. Then, add new **unit tests** for your policies and services. Example tests:

```
// DiscountPolicy test
@Test void loyalty_discount_5_percent() {
    DiscountPolicy d = new LoyaltyPercentDiscount(5);
    assertEquals(Money.of(0.39), d.discountOf(Money.of(7.80)));
}

// TaxPolicy test
```

```
@Test void fixed_rate_tax_10_percent() {
    TaxPolicy t = new FixedRateTaxPolicy(10);
    assertEquals(Money.of(0.74), t.taxOn(Money.of(7.41)));
}

// PricingService test
@Test void pricing_pipeline() {
    var pricing = new PricingService(new LoyaltyPercentDiscount(5), new
FixedRateTaxPolicy(10));
    var pr = pricing.price(Money.of(7.80));
    assertEquals(Money.of(0.39), pr.discount());
    assertEquals(Money.of(7.41),
Money.of(pr.subtotal().asBigDecimal().subtract(pr.discount().asBigDecimal
())));
    assertEquals(Money.of(0.74), pr.tax());
    assertEquals(Money.of(8.15), pr.total());
}
```

Step 6 — CLI Demo (Week6Demo)

Create a simple demo to show the old and new flows produce the same receipt text for given inputs. Your 30-second proof for the coordinator is that the “Old Receipt” (smelly) and “New Receipt” (clean) blocks are text-identical for one input. This is your 30-second proof.

```
package com.cafepos.demo;
public final class Week6Demo {
    public static void main(String[] args) {
        // Old behavior
        String oldReceipt = OrderManagerGod.process("LAT+L", 2, "CARD",
"LOYAL5", false);

        // New behavior with equivalent result
        var pricing = new PricingService(new LoyaltyPercentDiscount(5),
new FixedRateTaxPolicy(10));
        var printer = new ReceiptPrinter();
        var checkout = new CheckoutService(new ProductFactory(), pricing,
printer, 10);
        String newReceipt = checkout.checkout("LAT+L", 2);

        System.out.println("Old Receipt:\n" + oldReceipt);
        System.out.println("\nNew Receipt:\n" + newReceipt);
        System.out.println("\nMatch: " + oldReceipt.equals(newReceipt));
    }
}
```

Expected Sample Output (for the demo case LAT+L, qty 2, LOYAL5, CARD, 10% tax)

```
Old Receipt:
Order (LAT+L) x2
Subtotal: 7.80
Discount: -0.39
Tax (10%): 0.74
Total: 8.15
```

```
New Receipt:
Order (LAT+L) x2
Subtotal: 7.80
Discount: -0.39
Tax (10%): 0.74
Total: 8.15

Match: true
```

"30-Second Proof" Commands

From the project root (where pom.xml lives):
Unix/macOS/Linux: *mvn -q -DskipTests test*
Windows PowerShell (if needed): *mvn -q test*
Optional: add Week6Demo and run:

```
mvn -q -DskipTests -Dexec.mainClass=com.cafepos.demo.Week6Demo exec:java
```

Windows PowerShell (fallback fully-qualified):

```
mvn -q -DskipTests org.codehaus.mojo:exec-maven-plugin:3.3.0:java `
  -Dexec.mainClass=com.cafepos.demo.Week6Demo
```

If the exec plugin fails on Windows, compile and run directly:

```
mvn -q -DskipTests compile
java -cp target/classes com.cafepos.demo.Week6Demo
```

Reflection & Evidence of Effort

In your README, add a short (6–10 lines) design note:
• Which smells did you remove, and where?
• Which refactorings did you apply (by name), and why?
• How does your new design satisfy specific SOLID principles?
• What would be required to add a **new** discount type without editing existing classes?

Deliverables

1) Characterization tests (green) that lock the smelly behavior.
2) Refactored code following the target design (policies/services/printer, no global state).
3) New unit tests for policies/services (green).
4) A demo (optional) showing old vs new receipt match.
5) Commit history that shows incremental refactorings with meaningful messages.
6) A one-page summary (list or diagram) of final responsibilities: CheckoutService orchestrates → ProductFactory builds → PricingService uses DiscountPolicy and TaxPolicy → ReceiptPrinter formats → PaymentStrategy handles payment I/O. No globals; all dependencies injected

Reminder: *Week 6 lab marks cover **Week 5 outputs**; the refactoring you do now will be evaluated in the **Week 7 midterm**.*