Week 8 Lab — Command + Adapter Patterns Café POS & Delivery Project

This lab is about **clean orchestration and safe integration**. In Part A, you will introduce the **Command** pattern so button presses (the invoker) are decoupled from what actually happens in the domain (the receiver). This gives you undo and macro behaviors for free and keeps UI logic out of business code. In Part B, you will add an **Adapter** so a legacy vendor printer can be used without changing your existing domain or receipt code. By the end, you will (1) press buttons that build and pay orders via commands, and (2) print receipts to a simulated legacy device, all while keeping your core POS model unchanged.

Following is the current, post-mid-term project map so you can quickly locate files as you work:

```
cafe-pos/
 pom.xml
 src/
  main/java/com/cafepos/
   common/        ← Money, value objects
   catalog/       ← Product, SimpleProduct, Catalog, InMemoryCatalog
   decorator/     ← ProductDecorator, ExtraShot, OatMilk, Syrup, SizeLarge (+ Priced)
   factory/       ← ProductFactory (recipes: ESP, LAT, CAP + add-ons)
   order/         ← Order, LineItem, OrderIds
   payment/       ← PaymentStrategy, CashPayment, CardPayment, WalletPayment
   checkout/      ← PricingService, DiscountPolicy*, TaxPolicy*, ReceiptPrinter  (*from Week 6)
   smells/        ← OrderManagerGod (read-only reference from Week 6)
   command/       ← (you'll add this today)
   printing/      ← (you'll add this today)
  main/java/vendor/legacy/
   LegacyThermalPrinter (adaptee for Part B)
  demo/
   Week8Demo_Commands, Week8Demo_Adapter (you'll add/run these)
  test/java/com/cafepos/
   ... your JUnit tests for command & adapter
```

Keep this tree open beside you; when the lab text says "create class X", you'll know exactly where it belongs.

Important Assessment Note

Per module policy, the **Week 8 lab** builds on your Mid-Term (Week 7). The next lab assessment slot is **Week 9**, which will assess the outputs you produce in this Week 8 lab.

Learning Objectives

- Implement the **Command pattern** to decouple invokers (UI/buttons) from actions on the POS domain.

- Support **undo** and simple **macro commands** (sequence of actions).
- Apply the **Adapter pattern** to integrate a legacy vendor component without changing the core POS code.
- Write tests that prove the invoker/command/receiver collaboration and the adapter's compatibility.

Prerequisites (from Weeks 2–7)

You should have: Money, Product, SimpleProduct, Decorators & Factory (recipes), Order/LineItem, Payment strategies, Observer notifications (optional in this demo), and Week 6 pricing/receipt utilities.

Part A — Command Pattern

Concept Recap

The **Command** pattern encapsulates a request as an object, allowing you to parameterize clients with different requests, queue or log requests, and support undoable operations. We'll create:

- a `Command` interface with `execute()` and `undo()`;
- concrete commands (e.g., `AddItemCommand`, `RemoveItemCommand`, `PayOrderCommand`, `MarkReadyCommand`);
- a receiver (domain service) that actually performs the work (e.g., `OrderService`);
- an invoker that triggers commands (e.g., `PosRemote` with programmable buttons and a history stack).

Step 1 — Define the Command interface

You are about to create a tiny command protocol so a **remote** (the invoker) can trigger **actions** without knowing any business details. This makes it easy to bind different actions to different buttons at runtime and to **undo** the last command by asking the command how to reverse itself. Your only job in this step is to define a small interface; do not add logic here. Create a new package com.cafepos.command and add:

```
package com.cafepos.command;

public interface Command {
    void execute();
    default void undo() { /* optional */ }
}
```

Purpose of the step is to establish a uniform shape for all user-triggered actions. You can perform it by creating the file and compiling; nothing should run yet. No domain references in this interface—keep it generic.

Step 2 — Create the Receiver (OrderService)

Commands don't change your domain—they **call** it. The receiver is the place where domain work actually happens. You will create OrderService, a thin façade over your existing Order,

LineItem, ProductFactory, and PaymentStrategy. Keep behavior cohesive: adding items, removing the last item, computing totals, and delegating payment. Paste the following class into com.cafepos.command:

```
package com.cafepos.command;
import com.cafepos.catalog.Product;
import com.cafepos.common.Money;
import com.cafepos.factory.ProductFactory;
import com.cafepos.order.LineItem;
import com.cafepos.order.Order;
import com.cafepos.payment.PaymentStrategy;
public final class OrderService {
    private final ProductFactory factory = new ProductFactory();
    private final Order order;
    public OrderService(Order order) { this.order = order; }
    public void addItem(String recipe, int qty) {
        Product p = factory.create(recipe);
        order.addItem(new LineItem(p, qty));
        System.out.println("[Service] Added " + p.name() + " x" + qty);
    }
    public void removeLastItem() {
        var items = new java.util.ArrayList<>(order.items());
        if (!items.isEmpty()) {
            var last = items.get(items.size()-1);
            items.remove(items.size()-1);
            // replace items
            try {
                var field = Order.class.getDeclaredField("items");
                field.setAccessible(true);
                field.set(order, items);
            } catch (Exception e) {
                throw new IllegalStateException("Could not remove item
reflectively; adapt your Order API.");
            }
            System.out.println("[Service] Removed last item");
        }
    }
    public Money totalWithTax(int percent) { return
order.totalWithTax(percent); }
    public void pay(PaymentStrategy strategy, int taxPercent) {
        // Usually you'd call order.pay(strategy), here we display a
payment using the computed total
        var total = order.totalWithTax(taxPercent);
        strategy.pay(order);
        System.out.println("[Service] Payment processed for total " +
total);
    }
    public Order order() { return order; }
}
```

Purpose of the step is to isolate domain operations behind a single, testable API that commands can call. And you are performing it by creating the class, ensuring imports point to your existing types.
Note: *if your Order cannot remove items yet, expose a removeLastItem() method on Order and call it here instead of using reflection. The **receiver** should be the only place that knows how to mutate orders. The idea is that the **receiver** owns the domain operations; commands just call the receiver.*

Step 3 — Implement Concrete Commands

You will now write small "wrappers" that translate a button press into a service call. Keep commands dumb: no pricing math or I/O here—just call OrderService.

For **AddItemCommand** paste your class:

```
// AddItemCommand
package com.cafepos.command;
public final class AddItemCommand implements Command {
    private final OrderService service;
    private final String recipe;
    private final int qty;
    public AddItemCommand(OrderService service, String recipe,
int qty) {
        this.service = service; this.recipe = recipe; this.qty =
qty;
    }
    @Override public void execute() { service.addItem(recipe,
qty); }
    @Override public void undo() { service.removeLastItem(); }
}
```

For **PayOrderCommand** paste your class:

```
// PayOrderCommand
package com.cafepos.command;
import com.cafepos.payment.PaymentStrategy;
public final class PayOrderCommand implements Command {
    private final OrderService service;
    private final PaymentStrategy strategy;
    private final int taxPercent;
    public PayOrderCommand(OrderService service, PaymentStrategy
strategy, int taxPercent) {
        this.service = service; this.strategy = strategy;
this.taxPercent = taxPercent;
    }
    @Override public void execute() { service.pay(strategy,
taxPercent); }
}
```

This will capture a request as an object so it can be queued, logged, invoked, and undone. To achieve it, constructor stores parameters; execute() forwards to the service; undo() calls the service's inverse if available.

Step 4 — Build the Invoker (PosRemote) with Undo

The invoker holds slots (buttons) and a history. It neither knows nor cares what each command does. You'll bind commands to slots and then **press** them. Paste your remote into com.cafepos.command:

```
package com.cafepos.command;
import java.util.ArrayDeque;
import java.util.Deque;
public final class PosRemote {
    private final Command[] slots;
    private final Deque<Command> history = new ArrayDeque<>();
    public PosRemote(int n) { this.slots = new Command[n]; }
    public void setSlot(int i, Command c) { slots[i] = c; }
    public void press(int i) {
        Command c = slots[i];
        if (c != null) {
            c.execute();
            history.push(c);
        } else {
            System.out.println("[Remote] No command in slot " +
i);
        }
    }
    public void undo() {
        if (history.isEmpty()) { System.out.println("[Remote]
Nothing to undo"); return; }
        history.pop().undo();
    }
}
```

By creating the class; you will bind real commands in the demo and this decouples UI wiring from domain code and add an **undo** stack with minimal logic.

Step 5 — MacroCommand (Optional Stretch)

If you want multi-step actions (e.g., "Lunch Combo"), bundle commands into a **MacroCommand**. Execution runs forward; undo walks backwards. Paste as provided:

```
package com.cafepos.command;
public final class MacroCommand implements Command {
    private final Command[] steps;
    public MacroCommand(Command... steps) { this.steps = steps; }
    @Override public void execute() { for (Command c : steps)
c.execute(); }
    @Override public void undo() { for (int i = steps.length-1; i
>= 0; i--) steps[i].undo(); }
}
```

This demonstrates how commands compose into larger workflows. The class you created will be used in demos/tests if you attempt the stretch.

Step 6 — CLI Demo (Week8Demo) for Command

You now bind concrete commands to buttons and press them in sequence. Create com.cafepos.demo.Week8Demo_Commands with your demo code:

```
package com.cafepos.demo;
import com.cafepos.order.*;
import com.cafepos.payment.*;
import com.cafepos.command.*;

public final class Week8Demo_Commands {
    public static void main(String[] args) {
        Order order = new Order(OrderIds.next());
        OrderService service = new OrderService(order);
        PosRemote remote = new PosRemote(3);

        remote.setSlot(0, new AddItemCommand(service,
"ESP+SHOT+OAT", 1));
        remote.setSlot(1, new AddItemCommand(service, "LAT+L",
2));
        remote.setSlot(2, new PayOrderCommand(service, new
CardPayment("1234567890123456"), 10));
        remote.press(0);
        remote.press(1);
        remote.undo();   // remove last add
        remote.press(1); // add again
        remote.press(2); // pay
    }
}
```

Run it from the project root:

```
mvn -q -DskipTests -Dexec.mainClass=com.cafepos.demo.Week8Demo_Commands exec:java
```

This should prove the invoker/command/receiver flow and see **undo** working.
What to expect: console lines for "Added …", "Removed last item", and a payment line from your strategy.
If something looks off: check slot bindings and confirm your OrderService methods are being called. Expected (example) console output is following:

```
[Service] Added Espresso + Extra Shot + Oat Milk x1
[Service] Added Latte (Large) x2
[Service] Removed last item
[Service] Added Latte (Large) x2
[Card] Customer paid 12.12 EUR with card ****3456
[Service] Payment processed for total 12.12
```

Part B — Adapter Pattern

Concept Recap

The **Adapter** pattern converts the interface of a class (adaptee) into another interface clients expect. We will adapt a **legacy thermal printer** to the printer abstraction our checkout flow expects, without changing the core POS code.

Step 1 — Define the Target Interface (what our app expects)

Legacy devices won't change their APIs for you. Instead of modifying your POS model or the vendor class, you'll **adapt** the legacy printer to your app's expected printing interface. This keeps your checkout code stable while you swap real printers later.
To define a simple, app-friendly target interface, create com.cafepos.printing.Printer:

```
package com.cafepos.printing;
public interface Printer {
    void print(String receiptText);
}
```

Step 2 — A Legacy Vendor Class (Adaptee)

We simulate a legacy ESC/POS thermal printer that only accepts byte[]. Put it under vendor.legacy so it's clearly "not ours":

```
package vendor.legacy;
public final class LegacyThermalPrinter {
    public void legacyPrint(byte[] payload) {
        // imagine this talks to a serial port / ESC-POS device
        System.out.println("[Legacy] printing bytes: " +
payload.length);
    }
}
```

Step 3 — The Adapter

Now bridge from your Printer interface to the vendor class. Put the adapter in com.cafepos.printing. The purpose is to translate from your receipt text to the byte[] protocol expected by the legacy device, **without** touching either side's source.

```
package com.cafepos.printing;
import vendor.legacy.LegacyThermalPrinter;
import java.nio.charset.StandardCharsets;
public final class LegacyPrinterAdapter implements Printer {
    private final LegacyThermalPrinter adaptee;
    public LegacyPrinterAdapter(LegacyThermalPrinter adaptee) {
this.adaptee = adaptee; }
    @Override public void print(String receiptText) {
        byte[] escpos =
receiptText.getBytes(StandardCharsets.UTF_8);
        adaptee.legacyPrint(escpos);
    }
}
```

Step 4 — Use the Adapter in Checkout

Assuming you have a `ReceiptPrinter` from Week 6 that returns a formatted string, you can inject a `Printer` and send the string to it. Create com.cafepos.demo.Week8Demo_Adapter with your demo code:

```
package com.cafepos.demo;
import com.cafepos.checkout.ReceiptPrinter;
import com.cafepos.printing.*;
import vendor.legacy.LegacyThermalPrinter;
public final class Week8Demo_Adapter {
    public static void main(String[] args) {
        String receipt = "Order (LAT+L) x2\nSubtotal: 7.80\nTax
(10%): 0.78\nTotal: 8.58";
        Printer printer = new LegacyPrinterAdapter(new
LegacyThermalPrinter());
        printer.print(receipt);
        System.out.println("[Demo] Sent receipt via adapter.");
    }
}
```

Run the following:

```
mvn -q -DskipTests -Dexec.mainClass=com.cafepos.demo.Week8Demo_Adapter exec:java
```

This should prove that adapter is called and the legacy class receives bytes.
What to expect: a "[Legacy] printing bytes: …" line, then "[Demo] Sent receipt via adapter."
Expected (example) console output is following:

```
[Legacy] printing bytes: 48
[Demo] Sent receipt via adapter.
```

Testing

You are testing **collaboration**, not just numbers. Focus on "does the invoker call the right command?", "does undo reverse the last change?", and "does the adapter convert text to bytes?".

Following is the explanation of how we will achieve this testing:

1. Create a **fake** OrderService or use a real Order and count items before/after an undo() to prove the effect. The purpose is to lock the command→receiver path and the undo stack behavior.
2. For **MacroCommand**, execute two "add item" commands, then call undo() once and check that exactly one add is reverted. The purpose is to show reverse-order undo.
3. For **Adapter**, subclass the legacy printer in tests, capture the last payload length, and assert that print("ABC") yields a payload of length ≥ 3. The purpose is to confirm the conversion to bytes happens inside the adapter.
4. For a small **integration** test, bind two AddItemCommands and a PayOrderCommand to a PosRemote, press them in sequence, and assert the order subtotal equals what you expect from your Week 5 prices. The purpose is to show end-to-end wiring without touching UI or domain internals.

Here's a tiny adapter test scaffold (students paste in src/test/java and finish the assertion details):

```
// Example adapter test idea (pseudo-JUnit)
class FakeLegacy extends vendor.legacy.LegacyThermalPrinter {
    int lastLen = -1;
    @Override public void legacyPrint(byte[] payload) { lastLen =
payload.length; }
}
@org.junit.jupiter.api.Test
void adapter_converts_text_to_bytes() {
    var fake = new FakeLegacy();
    com.cafepos.printing.Printer p = new
com.cafepos.printing.LegacyPrinterAdapter(fake);
    p.print("ABC");
    org.junit.jupiter.api.Assertions.assertTrue(fake.lastLen >= 3);
}
```

Run all tests with: *mvn -q test*

"30-Second Proof" Commands

From the project root in Windows (where pom.xml lives):
Compile: *mvn -q -DskipTests compile*
Commands demo: *java -cp target/classes com.cafepos.demo.Week8Demo_Commands*
Adapter demo: *java -cp target/classes com.cafepos.demo.Week8Demo_Adapter*

The coordinator expects to see the four or five console lines from the commands demo (including an undo) and one "[Legacy] printing bytes: …" line from the adapter demo.

Drawing a Sequence Diagram for the Command Pattern

To consolidate your understanding of the Command pattern, you will draw a **sequence diagram** that shows how a single button press flows through the system. The purpose is not to reproduce every method call in fine detail, but to demonstrate that you understand the roles and collaborations involved. Start your diagram with the **Cashier actor**, who presses a button on the POS remote. The next lifeline is the **Invoker (PosRemote)**, which looks up the command assigned to that slot and invokes its execute() method. The command object (for example, AddItemCommand or PayOrderCommand) appears next: its role is simply to forward the request to the **Receiver (OrderService)**. The receiver then carries out the real work, such as adding a product to an order or processing a payment. Finally, the receiver may call into the existing domain classes—such as the ProductFactory, Order, and a PaymentStrategy—to complete the operation.

Deliverables

Submit your command and printing packages, the two demos, passing JUnit tests, and a short reflection. In your reflection, explicitly answer: **Where does Command decouple UI from business logic in your codebase?** and **Why is adapting the legacy printer better than changing your domain or vendor class? Also submit the sequence diagram prepared.**