Week 10 Lab — Architectural Design: Layered Architecture + MVC + Components/Connectors Café POS & Delivery Project

This lab will not be assessed or graded, and there is no submission required. Please use this time to continue working on your final project this week.

This week we take a step back from individual patterns and look at architecture. You will learn how to organize your Café POS project into clear layers, introduce MVC in the Presentation layer, and make component boundaries and connectors explicit. The goal is not to add new features, but to restructure the system so it is easier to reason about, test, and evolve. By the end of this lab, you will understand how design patterns plug into a broader architecture.

Learning Objectives

- Apply an architectural perspective to your project by introducing a clean Layered Architecture (Presentation, Application, Domain, Infrastructure).
- Implement a lightweight MVC interaction within the Presentation layer for a console UI.
- Make component boundaries explicit (components) and define how they communicate (connectors) via ports and adapters.
- Understand the trade-offs between Layering and Partitioning, and document your chosen decomposition.
- Produce a short "architecture evidence" demo with commands that a coordinator can verify in 30 seconds.

Prerequisites (Weeks 2–9)

You should have: Money/Order/LineItem, Product factory + decorators, Payment strategies, Pricing/Receipt services, Observer/Command/Adapter, Composite/Iterator menus, and a State-based order lifecycle.

Part A — Layered Architecture

We are now structuring the code into four layers:

- **Presentation (UI)** — handles I/O, console view, and controllers.
- **Application (Use Cases)** — orchestrates domain operations, but does not do I/O.
- **Domain (Core Model)** — contains business entities, value objects, and policies.
- **Infrastructure (Adapters)** — provides repositories, printers, and integration with external systems.

This separation of concerns is what makes the architecture maintainable: the domain core does not know about UI or databases, and dependencies only point inwards.

Step A1 — Define Domain Interfaces & Entities

You start by moving your Order entity into the domain layer and defining an OrderRepository interface. This creates a stable business core that other layers depend on, but which itself is independent.

```
// com/cafepos/domain/Order.java (existing, move here)
package com.cafepos.domain;

import java.util.*;
import java.math.BigDecimal;

public final class Order {
    private final long id;
    private final List<LineItem> items = new ArrayList<>();

    public Order(long id) { this.id = id; }
    public long id() { return id; }
    public List<LineItem> items() { return List.copyOf(items); }
    public void addItem(LineItem li) { items.add(li); }
    public void removeLastItem() { if (!items.isEmpty())
items.remove(items.size()-1); }

    public Money subtotal() {
        return
items.stream().map(LineItem::lineTotal).reduce(Money.zero(), Money::add);
    }
    public Money taxAtPercent(int percent) {
        var bd =
subtotal().asBigDecimal().multiply(BigDecimal.valueOf(percent)).divide(Bi
gDecimal.valueOf(100));
        return Money.of(bd);
    }
    public Money totalWithTax(int percent) { return
subtotal().add(taxAtPercent(percent)); }
}
```

```
// com/cafepos/domain/OrderRepository.java
package com.cafepos.domain;

import java.util.Optional;

public interface OrderRepository {
    void save(Order order);
    Optional<Order> findById(long id);
}
```

Step A2 — Application Services (Use Cases, no UI/DB code)

The application layer orchestrates use cases. It coordinates the domain and infrastructure but does not itself handle I/O. For example, CheckoutService computes totals and returns a receipt string, but does not print it.

```
// com/cafepos/app/CheckoutService.java
package com.cafepos.app;
import com.cafepos.domain.*;
import com.cafepos.pricing.PricingService;

public final class CheckoutService {
    private final OrderRepository orders;
    private final PricingService pricing;
    public CheckoutService(OrderRepository orders, PricingService
pricing) {
        this.orders = orders; this.pricing = pricing;
    }

    /** Returns a receipt string; does NOT print. */
    public String checkout(long orderId, int taxPercent) {
        Order order = orders.findById(orderId).orElseThrow();
        var pr = pricing.price(order.subtotal());
        return new ReceiptFormatter().format(orderId, order.items(), pr,
taxPercent);
    }
}
// com/cafepos/app/ReceiptFormatter.java
package com.cafepos.app;

import com.cafepos.common.Money;
import com.cafepos.domain.LineItem;
import java.util.List;

public final class ReceiptFormatter {
    public String format(long id, List<LineItem> items,
com.cafepos.pricing.PricingService.PricingResult pr, int taxPercent) {
        StringBuilder sb = new StringBuilder();
        sb.append("Order #").append(id).append("\n");
        for (LineItem li : items) {
            sb.append(" - ").append(li.product().name()).append("
x").append(li.quantity())
              .append(" = ").append(li.lineTotal()).append("\n");
        }
        sb.append("Subtotal: ").append(pr.subtotal()).append("\n");
        if (pr.discount().asBigDecimal().signum() > 0) {
            sb.append("Discount: -").append(pr.discount()).append("\n");
        }
        sb.append("Tax (").append(taxPercent).append("%):
").append(pr.tax()).append("\n");
        sb.append("Total: ").append(pr.total());
        return sb.toString();
    }
}
```

Step A3 — Infrastructure Adapters (Repositories, Printers)

Now you implement infrastructure, which is how your system talks to the outside world. An example is an in-memory repository, which is a stand-in for a database. Infrastructure depends on the domain, but not the other way around. This makes swapping persistence mechanisms easier later.

```
// com/cafepos/infra/InMemoryOrderRepository.java
package com.cafepos.infra;

import com.cafepos.domain.*;
import java.util.*;

public final class InMemoryOrderRepository implements OrderRepository {
    private final Map<Long, Order> store = new HashMap<>();
    @Override public void save(Order order) { store.put(order.id(),
order); }
```

```
    @Override public Optional<Order> findById(long id) { return
Optional.ofNullable(store.get(id)); }
}
```

Step A4 — Composition Root (Wiring)

Finally, you centralize wiring in one place. This is where you choose concrete implementations and pass them to constructors. Keeping object creation in one place makes dependencies explicit and testable.

```
// com/cafepos/infra/Wiring.java
package com.cafepos.infra;
import com.cafepos.app.CheckoutService;
import com.cafepos.pricing.*;
import com.cafepos.domain.*;

public final class Wiring {
    public static record Components(OrderRepository repo, PricingService
pricing, CheckoutService checkout) {}

    public static Components createDefault() {
        OrderRepository repo = new InMemoryOrderRepository();
        PricingService pricing = new PricingService(new
LoyaltyPercentDiscount(5), new FixedRateTaxPolicy(10));
        CheckoutService checkout = new CheckoutService(repo, pricing);
        return new Components(repo, pricing, checkout);
    }
}
```

Part B — MVC in the Presentation Layer (Console UI)

Now you practice a simple MVC design for the console UI. The idea is that the Controller translates user actions into application service calls, the Model is your domain, and the View handles only I/O. This separation prevents UI code from creeping into the domain. We'll build a tiny MVC: the Model is your domain objects; the Controller translates user intents into application service calls; the View manages console I/O. Keep the Controller free of formatting and the View free of business logic.

Step B1 — Controller

The controller creates orders, adds items, and triggers checkout. Notice that it does not do formatting or printing — it delegates that responsibility.

```
// com/cafepos/ui/OrderController.java
package com.cafepos.ui;
import com.cafepos.app.CheckoutService;
import com.cafepos.domain.*;
import com.cafepos.factory.ProductFactory;

public final class OrderController {
    private final OrderRepository repo;
    private final CheckoutService checkout;
    private final ProductFactory factory = new ProductFactory();
    public OrderController(OrderRepository repo, CheckoutService
checkout) {
        this.repo = repo; this.checkout = checkout;
    }
```

```java
    public long createOrder(long id) {
        repo.save(new Order(id));
        return id;
    }
    public void addItem(long orderId, String recipe, int qty) {
        Order order = repo.findById(orderId).orElseThrow();
        order.addItem(new LineItem(factory.create(recipe), qty));
        repo.save(order);
    }
    public String checkout(long orderId, int taxPercent) {
        return checkout.checkout(orderId, taxPercent);
    }
}
```

Step B2 — View (Console) and Demo

The ConsoleView is responsible only for printing. The demo shows how a controller and view work together with your layered architecture to produce a receipt.

```java
// com/cafepos/ui/ConsoleView.java
package com.cafepos.ui;

public final class ConsoleView {
    public void print(String s) { System.out.println(s); }
}
// com/cafepos/demo/Week10Demo_MVC.java
package com.cafepos.demo;

import com.cafepos.infra.Wiring;
import com.cafepos.ui.OrderController;
import com.cafepos.ui.ConsoleView;

public final class Week10Demo_MVC {
    public static void main(String[] args) {
        var c = Wiring.createDefault();
        var controller = new OrderController(c.repo(), c.checkout());
        var view = new ConsoleView();
        long id = 4101L;
        controller.createOrder(id);
        controller.addItem(id, "ESP+SHOT+OAT", 1);
        controller.addItem(id, "LAT+L", 2);
        String receipt = controller.checkout(id, 10);
        view.print(receipt);
    }
}
```

Expected output shows the receipt formatted properly (example):

```
Order #4101
 - Espresso + Extra Shot + Oat Milk x1 = 3.80
 - Latte (Large) x2 = 7.80
Subtotal: 11.60
Discount: -0.58
Tax (10%): 1.10
Total: 12.12
```

Part C — Components & Connectors (Ports & Adapters)

Now you introduce a simple notion of components and connectors. Components are the units (like OrderController, CheckoutService), and connectors are how they talk. You will implement a lightweight in-process EventBus so that components can publish and subscribe to events without tight coupling. We will define components as deployable/replaceable units with clear interfaces (ports). We'll add a simple in-process event bus so Presentation can react to Application events without tight coupling.

Step C1 — Define a Domain Event & EventBus

First define events (OrderCreated, OrderPaid) and a bus to deliver them. This allows you to publish/subscribe without hard dependencies.

```
// com/cafepos/app/events/OrderEvents.java
package com.cafepos.app.events;

public sealed interface OrderEvent permits OrderCreated, OrderPaid { }

public record OrderCreated(long orderId) implements OrderEvent { }
public record OrderPaid(long orderId) implements OrderEvent { }

// com/cafepos/app/events/EventBus.java
package com.cafepos.app.events;

import java.util.*;
import java.util.function.Consumer;

public final class EventBus {
    private final Map<Class<?>, List<Consumer<?>>> handlers = new
HashMap<>();

    public <T> void on(Class<T> type, Consumer<T> h) {
        handlers.computeIfAbsent(type, k -> new ArrayList<>()).add(h);
    }

    @SuppressWarnings("unchecked")
    public <T> void emit(T event) {
        var list = handlers.getOrDefault(event.getClass(), List.of());
        for (var h : list) ((Consumer<T>) h).accept(event);
    }
}
```

Step C2 — Use the Connector

You wire an OrderController to the EventBus and show that events are published and received. The console output confirms the interaction.

```
// com/cafepos/ui/EventWiringDemo.java
package com.cafepos.ui;
import com.cafepos.app.events.*;
import com.cafepos.infra.Wiring;

public final class EventWiringDemo {
    public static void main(String[] args) {
        var bus = new EventBus();
        var comp = Wiring.createDefault();
        var controller = new OrderController(comp.repo(),
comp.checkout());
        bus.on(OrderCreated.class, e -> System.out.println("[UI] order
created: " + e.orderId()));
        bus.on(OrderPaid.class, e -> System.out.println("[UI] order paid:
" + e.orderId()));
        long id = 4201L;
        controller.createOrder(id);
        bus.emit(new OrderCreated(id));
        // after a payment in your real flow:
        bus.emit(new OrderPaid(id));
    }
}
```

MVC demo

```
Order #4101
 - Espresso + Extra Shot + Oat Milk x1 = 3.80
 - Latte (Large) x2 = 7.80
Subtotal: 11.60
Discount: -0.58
Tax (10%): 1.10
Total: 12.12
```

Event wiring demo

```
[UI] order created: 4201
[UI] order paid: 4201
```

Part D — Trade-offs: Layering vs Partitioning

Finally, you reflect on layering versus partitioning. The purpose is to think ahead: why do we keep everything in one layered monolith for now, and what seams (like Payments or Notifications) might become separate services later. You capture this in a short README note. Write a 150–200 word note in your README answering:
• Why did you choose a Layered Monolith (for now) rather than partitioning into multiple services?
• Which seams are natural candidates for future partitioning (e.g., Payments, Notifications)?
• What are the connectors/protocols you would define if splitting (events, REST APIs)?
Keep the system simple today, but be deliberate about future evolution.

30-Second Proof commands:

```
# Run MVC demo
mvn -q -DskipTests
java -cp target/classes com.cafepos.demo.Week10Demo_MVC
java -cp target/classes com.cafepos.ui.EventWiringDemo
```

Deliverables for Final

1) Four-layer package structure with clear dependencies (UI→App→Domain; Infra as adapters).
2) Application service (`CheckoutService`) and a repository interface + in-memory implementation.
3) MVC console demo (`Week10Demo_MVC`) producing a receipt like the example.
4) Event bus + small demo wiring (`EventWiringDemo`).
5) README trade-off note (Layering vs Partitioning).

6) Full architecture diagram of project showing all entities, relationships and interactions (no need of attributes and methods in this diagram). Architecture diagram must present layers and tiers where required.