Week 9 Lab — Iterator + Composite + State Patterns Café POS & Delivery Project

The purpose of this lab is to extend your Café POS project with three classic design patterns: Composite, Iterator, and State. Composite will allow you to structure hierarchical menus so that categories and individual items can be treated uniformly. Iterator will let you traverse menu trees in depth-first order and filter items such as vegetarian-only choices. State will allow you to manage the lifecycle of an order without relying on long chains of if/else statements. By the end of this lab, you will see how these patterns help you navigate complex object structures and delegate behavior to states in a way that makes your system both extensible and robust.

Remember that this week's session still assesses your Week 8 outputs (Command and Adapter). The work you complete in this session will be assessed during Week 10.

Learning Objectives

- Implement **Composite** to model hierarchical menus and treat groups and items uniformly.
- Implement **Iterators** to traverse menu trees (depth-first), supporting selective traversal (e.g., desserts only).
- Implement the **State** pattern to model the order lifecycle with behavior delegated to state objects.
- Write tests to verify traversal order, filtering, and valid/invalid state transitions.
- Understand **transparency vs. safety** trade-offs when designing Composite APIs.

Prerequisites (Weeks 2–8)

You should already have: Money type, Order/LineItem, factory + decorators for products, payment strategies, receipts/pricing (Week 6), and Command/Adapter (Week 8). We will add menu structures (Composite + Iterator) and an explicit Order State Machine.

Summary of Part A

In the first part of the lab, you will implement the Composite pattern to build hierarchical menus. The first step is to create a base type, MenuComponent. This abstract class provides a common API for both composite menus and leaf menu items. By default, its operations throw an exception, which is a safe design choice because it prevents misuse (for example, trying to add a child to a leaf). This balances transparency and safety: the API is visible, but only supported in the right subclasses.

Next, you will implement MenuItem, which is the leaf of the composite. This class stores the name, price, and vegetarian flag for a single dish. It overrides only the operations that are relevant to leaf nodes, such as returning its name and price. By design, it does not support child management.

After that, you will implement Menu, the composite node that can hold other menus or menu items. It stores its children in a list and allows adding, removing, and accessing them. By overriding iterator() and print(), the composite can be traversed uniformly with its children. This is what allows you to treat an entire sub-menu just like an item in the larger menu tree.

The final piece in this part is the CompositeIterator. Unlike a normal iterator, which works only on flat collections, this iterator uses a stack of iterators to traverse nested menus in depth-first order. As it walks through the tree, it automatically pushes child iterators onto the stack, allowing seamless traversal of arbitrarily deep menu structures.

Summary of Part B

In the second part of the lab, you will see how to use the Composite and Iterator together. You will build a menu tree with categories such as “Drinks,” “Coffee,” and “Desserts.” You will then print the full menu by simply calling print() on the root menu, showing how easy traversal becomes. Finally, you will demonstrate filtering by retrieving only vegetarian items, proving that you can use the iterator and simple stream filters to query complex structures without exposing internal lists. This part of the lab demonstrates the power of combining Composite and Iterator: clients get a uniform API and the ability to query or filter without breaking encapsulation.

Summary of Part C

In the third part of the lab, you will model the order lifecycle using the State pattern. Instead of using a status enum with long conditional chains, you will define a State interface and multiple concrete states, such as NewState, PreparingState, ReadyState, DeliveredState, and CancelledState. The context class, OrderFSM, will hold the current state and delegate all actions—such as pay, prepare, markReady, deliver, or cancel—to the state object. Each state decides which transitions are valid and what happens when they are invoked.

For example, when an order is in the NEW state, calling pay() will move it into the PREPARING state. If you try to call prepare() before paying, the NewState will reject that transition and print a message explaining why. Once in PREPARING, calling markReady() moves it to the READY state, and so on. By encapsulating transitions in state objects, you avoid tangled conditional logic and make it much easier to add or modify states later.

The demo will show how an order flows through its lifecycle. It starts in the NEW state, rejects preparation before payment, then moves through PREPARING, READY, and finally DELIVERED. Console output will confirm each step and the final status.

Part A — Composite: Hierarchical Menus

Step A1 — Base Type (MenuComponent)

What you are doing: Creating a common abstraction so both Menu (composite) and MenuItem (leaf) can be used with the same API.
How: Define abstract methods, with safe defaults that throw

UnsupportedOperationException.

Why: This balances **safety vs. transparency** — only the right operations are supported on each type.

```
package com.cafepos.menu;
import com.cafepos.common.Money;
import java.util.Iterator;
public abstract class MenuComponent {
    // Composite ops (unsupported by default → safe)
    public void add(MenuComponent c) { throw new
UnsupportedOperationException(); }
    public void remove(MenuComponent c) { throw new
UnsupportedOperationException(); }
    public MenuComponent getChild(int i) { throw new
UnsupportedOperationException(); }
    // Leaf data (unsupported by default → safe)
    public String name() { throw new UnsupportedOperationException(); }
    public Money price() { throw new UnsupportedOperationException(); }
    public boolean vegetarian() { return false; }
    // Iteration / printing hooks
    public Iterator<MenuComponent> iterator() { throw new
UnsupportedOperationException(); }
    public void print() { throw new UnsupportedOperationException(); }
}
```

Step A2 — Leaf (MenuItem)

What you are doing: Implementing the leaf nodes of the composite.

How: MenuItem stores name, price, and vegetarian flag, and overrides only leaf operations.

Why: Leaves represent actual menu items; they don't need to support child manipulation.

```
package com.cafepos.menu;
import com.cafepos.common.Money;
import java.util.Collections;
import java.util.Iterator;
public final class MenuItem extends MenuComponent {
    private final String name;
    private final Money price;
    private final boolean vegetarian;
    public MenuItem(String name, Money price, boolean vegetarian) {
        if (name == null || name.isBlank()) throw new
IllegalArgumentException("name required");
        if (price == null) throw new IllegalArgumentException("price
required");
        this.name = name; this.price = price; this.vegetarian =
vegetarian;
    }
    @Override public String name() { return name; }
    @Override public Money price() { return price; }
    @Override public boolean vegetarian() { return vegetarian; }
    @Override public Iterator<MenuComponent> iterator() { return
Collections.emptyIterator(); }
    @Override public void print() {
        String veg = vegetarian ? " (V)" : "";
        System.out.println(" - " + name + veg + " = " + price);
    }
}
```

Step A3 — Composite (Menu)

What you are doing: Implementing a menu category that can hold children (menus or items).

How: Store a List<MenuComponent> and expose add/remove/iterator methods.
Why: Enables uniform traversal and printing of complex menu trees.

```
package com.cafepos.menu;
import java.util.*;
import java.util.stream.Collectors;

public final class Menu extends MenuComponent {
    private final String name;
    private final List<MenuComponent> children = new ArrayList<>();
    public Menu(String name) {
        if (name == null || name.isBlank()) throw new
IllegalArgumentException("name required");
        this.name = name;
    }
    @Override public void add(MenuComponent c) { children.add(c); }
    @Override public void remove(MenuComponent c) { children.remove(c); }
    @Override public MenuComponent getChild(int i) { return
children.get(i); }
    @Override public String name() { return name; }
    // Expose child iterator for CompositeIterator
    public Iterator<MenuComponent> childrenIterator() { return
children.iterator(); }
    @Override public Iterator<MenuComponent> iterator() {
        return new CompositeIterator(childrenIterator());
    }
    @Override public void print() {
        System.out.println(name);
        for (MenuComponent c : children) c.print();
    }
    public List<MenuComponent> allItems() {
        List<MenuComponent> out = new ArrayList<>();
        var it = iterator();
        while (it.hasNext()) out.add(it.next());
        return out;
    }
    public List<MenuItem> vegetarianItems() {
        return allItems().stream()
            .filter(mc -> mc instanceof MenuItem mi && mi.vegetarian())
            .map(mc -> (MenuItem) mc)
            .collect(Collectors.toList());
    }
}
```

Step A4 — Depth-First CompositeIterator

What you are doing: Writing a custom iterator that walks the tree depth-first.
How: Use a stack of iterators to traverse composites and their children.
Why: Standard iterators only work on flat collections; this supports nested menus.

```
package com.cafepos.menu;
import java.util.*;
public final class CompositeIterator implements Iterator<MenuComponent> {
    private final Deque<Iterator<MenuComponent>> stack = new
ArrayDeque<>();
    public CompositeIterator(Iterator<MenuComponent> root) {
stack.push(root); }
    @Override public boolean hasNext() {
        while (!stack.isEmpty()) {
            if (stack.peek().hasNext()) return true;
            stack.pop();
```

```
        }
        return false;
    }
    @Override public MenuComponent next() {
        if (!hasNext()) throw new NoSuchElementException();
        Iterator<MenuComponent> it = stack.peek();
        MenuComponent comp = it.next();
        if (comp instanceof Menu m) {
            stack.push(m.childrenIterator());
        }
        return comp;
    }
}
```

Part B — Iterator Usage & Filtering

What you are doing: Building and traversing a real menu.
How: Create nested menus, print them, then filter for vegetarian items.
Why: Demonstrates the **power of Composite + Iterator**: uniform traversal and selective queries. Now construct a menu tree and traverse it depth-first. We'll also filter for vegetarian items to show selective iteration.

```
package com.cafepos.demo;
import com.cafepos.menu.*;
import com.cafepos.common.Money;
public final class Week9Demo_Menu {
    public static void main(String[] args) {
        Menu root = new Menu("CAFÉ MENU");
        Menu drinks = new Menu(" Drinks ");
        Menu coffee = new Menu("  Coffee ");
        Menu desserts = new Menu(" Desserts ");
        coffee.add(new MenuItem("Espresso", Money.of(2.50), true));
        coffee.add(new MenuItem("Latte (Large)", Money.of(3.90), true));
        drinks.add(coffee);
        desserts.add(new MenuItem("Cheesecake", Money.of(3.50), false));
        desserts.add(new MenuItem("Oat Cookie", Money.of(1.20), true));
        root.add(drinks);
        root.add(desserts);
        // Print entire menu
        root.print();
        // List vegetarian items only
        System.out.println("\nVegetarian options:");
        for (MenuItem mi : root.vegetarianItems()) {
            System.out.println(" * " + mi.name() + " = " + mi.price());
        }
    }
}
```

Expected console output (example):

```
CAFÉ MENU
 Drinks
  Coffee
 - Espresso (V) = 2.50
 - Latte (Large) (V) = 3.90
 Desserts
 - Cheesecake = 3.50
 - Oat Cookie (V) = 1.20
Vegetarian options:
 * Espresso = 2.50
 * Latte (Large) = 3.90
 * Oat Cookie = 1.20
```

Part C — State: Order Lifecycle as an FSM

We will model the **Order** lifecycle using **State** objects. Instead of `if/else` chains on a status enum, each state class will implement the allowed transitions and behavior. The context `OrderFSM` delegates to its current state.

Step C1 — State interface & Context

What you are doing: Defining the interface for all states and the FSM context.
How: State declares actions; OrderFSM delegates to the current state.
Why: Removes messy if/else logic by encapsulating transitions in state classes.

```
package com.cafepos.state;
public interface State {
    void pay(OrderFSM ctx);
    void prepare(OrderFSM ctx);
    void markReady(OrderFSM ctx);
    void deliver(OrderFSM ctx);
    void cancel(OrderFSM ctx);
    String name();
}
```

```
package com.cafepos.state;
public final class OrderFSM {
    private State state;
    public OrderFSM() { this.state = new NewState(); }
    void set(State s) { this.state = s; }
    public String status() { return state.name(); }
    public void pay() { state.pay(this); }
    public void prepare() { state.prepare(this); }
    public void markReady() { state.markReady(this); }
    public void deliver() { state.deliver(this); }
    public void cancel() { state.cancel(this); }
}
```

Step C2 — Concrete States

What you are doing: Implementing each order state (NEW, PREPARING, READY, DELIVERED, CANCELLED).
How: Each class overrides only valid transitions and prints appropriate messages.
Why: Encapsulates allowed/forbidden behavior in one place per state.

```
package com.cafepos.state;

final class NewState implements State {
    @Override public void pay(OrderFSM ctx) { System.out.println("[State]
Paid → Preparing"); ctx.set(new PreparingState()); }
    @Override public void prepare(OrderFSM ctx) {
System.out.println("[State] Cannot prepare before pay"); }
    @Override public void markReady(OrderFSM ctx) {
System.out.println("[State] Not ready yet"); }
    @Override public void deliver(OrderFSM ctx) {
System.out.println("[State] Cannot deliver yet"); }
    @Override public void cancel(OrderFSM ctx) {
System.out.println("[State] Cancelled"); ctx.set(new CancelledState()); }
    @Override public String name() { return "NEW"; }
}

final class PreparingState implements State {
```

```
    @Override public void pay(OrderFSM ctx) { System.out.println("[State]
Already paid"); }
    @Override public void prepare(OrderFSM ctx) {
System.out.println("[State] Still preparing..."); }
    @Override public void markReady(OrderFSM ctx) {
System.out.println("[State] Ready for pickup"); ctx.set(new
ReadyState()); }
    @Override public void deliver(OrderFSM ctx) {
System.out.println("[State] Deliver not allowed before ready"); }
    @Override public void cancel(OrderFSM ctx) {
System.out.println("[State] Cancelled during prep"); ctx.set(new
CancelledState()); }
    @Override public String name() { return "PREPARING"; }
}

final class ReadyState implements State {
    @Override public void pay(OrderFSM ctx) { System.out.println("[State]
Already paid"); }
    @Override public void prepare(OrderFSM ctx) {
System.out.println("[State] Already prepared"); }
    @Override public void markReady(OrderFSM ctx) {
System.out.println("[State] Already ready"); }
    @Override public void deliver(OrderFSM ctx) {
System.out.println("[State] Delivered"); ctx.set(new DeliveredState()); }
    @Override public void cancel(OrderFSM ctx) {
System.out.println("[State] Cannot cancel after ready"); }
    @Override public String name() { return "READY"; }
}

final class DeliveredState implements State {
    @Override public void pay(OrderFSM ctx) { System.out.println("[State]
Completed"); }
    @Override public void prepare(OrderFSM ctx) {
System.out.println("[State] Completed"); }
    @Override public void markReady(OrderFSM ctx) {
System.out.println("[State] Completed"); }
    @Override public void deliver(OrderFSM ctx) {
System.out.println("[State] Already delivered"); }
    @Override public void cancel(OrderFSM ctx) {
System.out.println("[State] Completed"); }
    @Override public String name() { return "DELIVERED"; }
}

final class CancelledState implements State {
    @Override public void pay(OrderFSM ctx) { System.out.println("[State]
Cancelled"); }
    @Override public void prepare(OrderFSM ctx) {
System.out.println("[State] Cancelled"); }
    @Override public void markReady(OrderFSM ctx) {
System.out.println("[State] Cancelled"); }
    @Override public void deliver(OrderFSM ctx) {
System.out.println("[State] Cancelled"); }
    @Override public void cancel(OrderFSM ctx) {
System.out.println("[State] Already cancelled"); }
    @Override public String name() { return "CANCELLED"; }
}
```

Step C3 — FSM Demo

What you are doing: Running the order FSM through transitions.
How: Call pay(), prepare(), markReady(), etc., and observe console output.
Why: Shows how the state pattern cleanly models lifecycle progression.

```
package com.cafepos.demo;

import com.cafepos.state.OrderFSM;

public final class Week9Demo_State {
    public static void main(String[] args) {
        OrderFSM fsm = new OrderFSM();
        System.out.println("Status = " + fsm.status());
        fsm.prepare();     // invalid before pay
        fsm.pay();         // NEW -> PREPARING
        fsm.prepare();     // still preparing
        fsm.markReady();   // PREPARING -> READY
        fsm.deliver();     // READY -> DELIVERED
        System.out.println("Status = " + fsm.status());
    }
}
```

Expected console output (example):

Menu demo:

```
CAFÉ MENU
 Drinks
  Coffee
 - Espresso (V) = 2.50
 - Latte (Large) (V) = 3.90
 Desserts
 - Cheesecake = 3.50
 - Oat Cookie (V) = 1.20
Vegetarian options:
 * Espresso = 2.50
 * Latte (Large) = 3.90
 * Oat Cookie = 1.20
```

State demo:

```
Status = NEW
[State] Cannot prepare before pay
[State] Paid → Preparing
[State] Still preparing...
[State] Ready for pickup
[State] Delivered
Status = DELIVERED
```

Tests (JUnit 5)

Create tests to verify behavior:
1) **Composite/Iterator** — build a menu with nested categories and items. Assert depth-first traversal order and that `vegetarianItems()` returns only veg items.
2) **State** — assert legal transitions change the `status()` and illegal ones print messages (or raise exceptions if you choose). Example:

NEW→pay→PREPARING→markReady→READY→deliver→DELIVERED.
3) **Integration** — optional: select a menu item by name, create it via `ProductFactory`, and assert totals match your Money logic.

```
// Example test sketch (pseudo)
@Test void depth_first_iteration_collects_all_nodes() {
    Menu root = new Menu("ROOT");
    Menu a = new Menu("A");
    Menu b = new Menu("B");
    root.add(a); root.add(b);
    a.add(new MenuItem("x", Money.of(1.0), true));
    b.add(new MenuItem("y", Money.of(2.0), false));

    List<String> names =
root.allItems().stream().map(MenuComponent::name).toList();
    assertTrue(names.contains("x"));
    assertTrue(names.contains("y"));
}

@Test void order_fsm_happy_path() {
    OrderFSM fsm = new OrderFSM();
    assertEquals("NEW", fsm.status());
    fsm.pay();
    assertEquals("PREPARING", fsm.status());
    fsm.markReady();
    assertEquals("READY", fsm.status());
    fsm.deliver();
    assertEquals("DELIVERED", fsm.status());
}
```

"30-Second Proof" Commands

From the project root (where pom.xml lives):
Compile: *mvn -q -DskipTests compile*
Menu Demo: *java -cp target/classes com.cafepos.demo.Week9Demo_Menu*
State Demo: *java -cp target/classes com.cafepos.demo.Week9Demo_State*

Draw a State Transition Table

To show that you understand how the **State pattern** models the order lifecycle, you will not draw a class diagram or sequence diagram this week. Instead, you will create a **state transition table**.

What you will do

- List all possible states of an order down the rows (NEW, PREPARING, READY, DELIVERED, CANCELLED).
- List all possible events across the columns (pay, prepare, markReady, deliver, cancel).
- For each cell in the table, mark whether the transition is **allowed (✓)** or **not allowed (✗)**. If allowed, you should also note the **target state** (e.g., NEW → PREPARING).

How you will do it

- Read your State implementations (NewState, PreparingState, etc.) and look at what each method does.
- Translate the behavior into a small grid: if the method changes to another state, put ✓ and write the new state. If it just prints a "not allowed" message, put ✗.

Why you are doing this

The purpose is not to repeat your code in another form, but to prove that you really understand the *rules of the lifecycle*. A state transition table makes these rules explicit at a glance. It also shows why the State pattern is cleaner than a mess of if/else conditions: the legal moves are defined once, in one place per state.

You Must Fill the Following Table to be Assessed in the Next Week

State	pay	prepare	markReady	deliver	cancel
NEW					
PREPARING					
READY					
DELIVERED					
CANCELLED					

Submit your own completed state transition table as part of this lab. This will be checked in your Week 10 assessment together with your code.

Deliverables

1) `menu` package with `MenuComponent`, `Menu`, `MenuItem`, and `CompositeIterator`.
2) `state` package with `State` interface, concrete states, and `OrderFSM` context.
3) Demos: `Week9Demo_Menu`, `Week9Demo_State` with outputs similar to the examples.
4) JUnit tests green.
5) **Short reflection**: **Where did you choose **safety** over **transparency** in your Composite API and why? What new behavior becomes easy with State that was awkward with conditionals?**