13 Challenge 2 — Restaurant Ordering System with Builder + Abstract Factory

**Scenario** Customers can craft their own meal—main course, side, drink, plus per-item options (portion, cooking style).  
 At the same time, the restaurant sells ready-made combos (“Lunch Box”, “Kids Meal”), and management wants to add new menu items or combos later **without touching ordering logic**.

You must combine two creational patterns to satisfy:

* **Step-by-step customization** of a single order.
* **Plug-in predefined packages** that the UI can add to the cart with one click.
* **Open/Closed** for new dishes or combos.

#### **Tasks**

1 analysis/pattern\_choice.md Argue which two creational patterns solve the requirements (hint: Builder + Abstract Factory) and map each requirement to one pattern.  
 2 **Domain model** (src/main/java/clean/)

* **Products** – MainCourse, Side, Drink; each implements MenuItem with price() and description().
* **Builder layer**
  + Order (immutable) containing lists of MenuItem and per-item Preference objects.
  + Fluent OrderBuilder with methods addMain(...), addSide(...), addDrink(...), pref(...), build().
* **Abstract Factory layer**
  + MealFactory interface with createMain(), createSide(), createDrink().
  + Concrete factories: LunchComboFactory, KidsMealFactory.
* OrderService  
  + createCustomOrder(OrderBuilder builder) and
  + createCombo(MealFactory factory) which plugs items from a factory into a new builder then builds.  
     3 **Unit tests** (src/test/java/clean/)
* BuilderBuildsCustomMealTest – verify builder assembles three items and total price.
* ComboFactoryProducesConsistentFamilyTest – assert all items of a LunchComboFactory carry "Lunch" tag in description().  
   4 reflection.md
* How does Builder keep fluent, readable customization?
* How does Abstract Factory keep combos consistent and extensible?
* What must change to add a “Vegan Feast” combo or “Bubble Tea” drink?

#### **Deliverables**

analysis/pattern\_choice.md

src/main/java/clean/menu/MenuItem.java

src/main/java/clean/menu/MainCourse.java

src/main/java/clean/menu/Side.java

src/main/java/clean/menu/Drink.java

src/main/java/clean/order/Preference.java

src/main/java/clean/order/Order.java

src/main/java/clean/order/OrderBuilder.java

src/main/java/clean/factory/MealFactory.java

src/main/java/clean/factory/LunchComboFactory.java

src/main/java/clean/factory/KidsMealFactory.java

src/main/java/clean/service/OrderService.java

src/test/java/clean/\*\* ← two tests

reflection.md

README.md

## **Detailed Solution**

### **1 pattern\_choice.md (excerpt)**

We need:

• step-wise, optional parts → Builder

• one-shot family of related items (combo) → Abstract Factory

Therefore we combine:

1. Builder – constructs an Order incrementally with fluent API.

2. Abstract Factory – produces a family of menu items that always ‘match’.

### **2 Core implementation**

/\* menu/MenuItem.java \*/

package clean.menu;

public interface MenuItem { String description(); double price(); }

/\* menu/MainCourse.java \*/

package clean.menu;

public class MainCourse implements MenuItem {

private final String desc; private final double price;

public MainCourse(String d,double p){ this.desc=d; this.price=p; }

public String description(){ return desc; } public double price(){ return price; }

}

/\* menu/Side.java & menu/Drink.java similar \*/

/\* order/Preference.java \*/

package clean.order;

public class Preference {

private final String itemDesc, note;

public Preference(String itemDesc,String note){ this.itemDesc=itemDesc; this.note=note; }

public String toString(){ return itemDesc+" -> "+note; }

}

/\* order/Order.java \*/

package clean.order;

import clean.menu.MenuItem;

import java.util.\*;

public final class Order {

private final List<MenuItem> items; private final List<Preference> prefs;

Order(List<MenuItem> i,List<Preference> p){ items=i; prefs=p; }

public List<MenuItem> items(){ return Collections.unmodifiableList(items); }

public List<Preference> preferences(){ return Collections.unmodifiableList(prefs); }

public double total(){ return items.stream().mapToDouble(MenuItem::price).sum(); }

}

/\* order/OrderBuilder.java \*/

package clean.order;

import clean.menu.\*;

import java.util.\*;

public class OrderBuilder {

private final List<MenuItem> items = new ArrayList<>();

private final List<Preference> prefs = new ArrayList<>();

public OrderBuilder addMain(String desc,double price){

items.add(new MainCourse(desc,price)); return this; }

public OrderBuilder addSide(String desc,double price){

items.add(new Side(desc,price)); return this; }

public OrderBuilder addDrink(String desc,double price){

items.add(new Drink(desc,price)); return this; }

public OrderBuilder pref(String itemDesc,String note){

prefs.add(new Preference(itemDesc,note)); return this; }

public Order build(){ return new Order(new ArrayList<>(items), new ArrayList<>(prefs)); }

}

/\* factory/MealFactory.java \*/

package clean.factory;

import clean.menu.\*;

public interface MealFactory {

MainCourse createMain();

Side createSide();

Drink createDrink();

}

/\* factory/LunchComboFactory.java \*/

package clean.factory;

import clean.menu.\*;

public class LunchComboFactory implements MealFactory {

public MainCourse createMain(){ return new MainCourse("Lunch Burger",8.0); }

public Side createSide(){ return new Side("Fries (Lunch)",2.5); }

public Drink createDrink(){ return new Drink("Soda (Lunch)",1.5); }

}

/\* factory/KidsMealFactory.java similar \*/

/\* service/OrderService.java \*/

package clean.service;

import clean.order.\*;

import clean.factory.\*;

public class OrderService {

public Order createCustomOrder(OrderBuilder b){ return b.build(); }

public Order createCombo(MealFactory f){

return new OrderBuilder()

.addMain(f.createMain().description(), f.createMain().price())

.addSide(f.createSide().description(), f.createSide().price())

.addDrink(f.createDrink().description(),f.createDrink().price())

.build();

}

}

### **3 Tests (outline)**

/\* BuilderBuildsCustomMealTest \*/

Order o = new OrderBuilder()

.addMain("Steak",15)

.addDrink("Wine",5)

.pref("Steak","medium-rare").build();

assertEquals(20.0,o.total());

/\* Pool tests analogous for builder reuse \*/

/\* ComboFactoryProducesConsistentFamilyTest \*/

MealFactory f = new LunchComboFactory();

assertTrue(f.createMain().description().contains("Lunch"));

assertTrue(f.createSide().description().contains("Lunch"));

### **Reflection (highlights)**

*Builder* gives end-user freedom: new OrderBuilder().addMain(...).pref(...)…build() reads like natural language; object is immutable → safe to hand across threads.  
 *Abstract Factory* guarantees a combo’s components are from the same “family” (Lunch vs Kids). Adding “VeganFeastFactory” requires no change to OrderService.  
 Trade-off: extra classes and boiler-plate, but ordering logic remains closed for modification while menu expands indefinitely—achieving Open/Closed Principle.