Parte 1: Approccio Base Ristorante Pizzeria

Analisi del Dominio

Il ristorante produce pizze con:

- **Stato variabile**: ingredienti, dimensione, tipo impasto
- **Comportamento comune**: cottura, taglio, confezionamento
- **Varianti**: pizze classiche predefinite vs personalizzate

Soluzione Diretta

```
abstract class Pizza {
    protected String nome;
    protected TipoImpasto impasto;
    protected List<Ingrediente> ingredienti;
    protected Dimensione dimensione;

    public void prepara() {
        stendereImpasto();
        aggiungereSalsa();
        aggiungiIngredienti();
    }

    protected abstract void aggiungiIngredienti();

    public void cuoci() {
        System.out.println("Cottura a 300°C per 4 minuti");
    }

    public void taglia() {
        System.out.println("Taglio in " + dimensione.getSpicchi() + "
spicchi");
    }
}

class Margherita extends Pizza {
    public Margherita() {
        this.nome = "Margherita";
        this.ingredienti = new ArrayList<>();
    }

    protected void aggiungiIngredienti() {
        ingredienti.add(new Ingrediente("Mozzarella", 150));
        ingredienti.add(new Ingrediente("Basilico", 5));
    }
```

```
}

class Diavola extends Pizza {
    protected void aggiungiIngredienti() {
        ingredienti.add(new Ingrediente("Mozzarella", 150));
        ingredienti.add(new Ingrediente("Salame piccante", 80));
    }
}

class QuattroStagioni extends Pizza {
    protected void aggiungiIngredienti() {
        ingredienti.add(new Ingrediente("Mozzarella", 150));
        ingredienti.add(new Ingrediente("Prosciutto", 50));
        ingredienti.add(new Ingrediente("Funghi", 50));
        ingredienti.add(new Ingrediente("Carciofi", 50));
        ingredienti.add(new Ingrediente("Olive", 30));
    }
}

// Uso
Pizza ordine1 = new Margherita();
ordine1.prepara();
ordine1.cuoci();
ordine1.taglia();
```

Problemi di Questo Approccio

1. **Esplosione di classi**: ogni nuova pizza = nuova classe
2. **Rigidità**: cliente vuole "Margherita senza basilico"? Servirebbero sottoclassi o logica condizionale
3. **Duplicazione**: ingredienti base ripetuti in più classi
4. **Manutenzione**: cambio prezzo mozzarella richiede modifica di N classi

---

Parte 2: Miglioramenti Progressivi

Livello 1: Builder Pattern (materiale corso)

```
class Pizza {
    private String nome;
    private TipoImpasto impasto;
    private List<Ingrediente> ingredienti;
    private Dimensione dimensione;

    // costruttore privato
    private Pizza(Builder builder) {
```

```
        this.nome = builder.nome;
        this.impasto = builder.impasto;
        this.ingredienti = builder.ingredienti;
        this.dimensione = builder.dimensione;
    }

    static class Builder {
        private String nome;
        private TipoImpasto impasto = TipoImpasto.NORMALE;
        private List<Ingrediente> ingredienti = new ArrayList<>();
        private Dimensione dimensione = Dimensione.MEDIA;

        public Builder(String nome) {
            this.nome = nome;
        }

        public Builder impasto(TipoImpasto impasto) {
            this.impasto = impasto;
            return this;
        }

        public Builder aggiungi(Ingrediente ing) {
            this.ingredienti.add(ing);
            return this;
        }

        public Builder dimensione(Dimensione dim) {
            this.dimensione = dim;
            return this;
        }

        public Pizza build() {
            return new Pizza(this);
        }
    }
}

// Uso
Pizza margherita = new Pizza.Builder("Margherita")
    .aggiungi(new Ingrediente("Mozzarella", 150))
    .aggiungi(new Ingrediente("Basilico", 5))
    .build();

Pizza personalizzata = new Pizza.Builder("Speciale Cliente")
    .impasto(TipoImpasto.INTEGRALE)
    .aggiungi(new Ingrediente("Mozzarella", 200))
    .aggiungi(new Ingrediente("Salsiccia", 100))
    .dimensione(Dimensione.FAMIGLIA)
    .build();
```

Vantaggi: massima flessibilità, no esplosione classi, costruzione leggibile.

Livello 2: Prototype Pattern + Registry

```
class MenuPizze {
    private Map<String, Pizza> ricette = new HashMap<>();

    public MenuPizze() {
        // pizze standard come prototipi
        Pizza margherita = new Pizza.Builder("Margherita")
            .aggiungi(new Ingrediente("Mozzarella", 150))
            .aggiungi(new Ingrediente("Basilico", 5))
            .build();
        ricette.put("Margherita", margherita);

        Pizza diavola = new Pizza.Builder("Diavola")
            .aggiungi(new Ingrediente("Mozzarella", 150))
            .aggiungi(new Ingrediente("Salame", 80))
            .build();
        ricette.put("Diavola", diavola);
    }

    public Pizza crea(String tipo) {
        Pizza prototipo = ricette.get(tipo);
        return prototipo != null ? prototipo.clone() : null;
    }
}

// Uso
MenuPizze menu = new MenuPizze();
Pizza ordine = menu.crea("Margherita");
ordine.personalizza(modifiche -> modifiche.rimuovi("Basilico"));
```

Livello 3: Strategy + Decorator (avanzato)

```
// Strategy per cottura
interface StrategiaCottura {
    void cuoci(Pizza pizza);
}

class CotturaLegna implements StrategiaCottura {
    public void cuoci(Pizza pizza) {
        System.out.println("Forno a legna 350°C, 3 min");
    }
}

class CotturaElettrico implements StrategiaCottura {
    public void cuoci(Pizza pizza) {
```

```
        System.out.println("Forno elettrico 280°C, 5 min");
    }
}

// Decorator per varianti
abstract class VariantePizza extends Pizza {
    protected Pizza pizzaBase;

    public VariantePizza(Pizza base) {
        this.pizzaBase = base;
    }
}

class ConBordoRipieno extends VariantePizza {
    public ConBordoRipieno(Pizza base) {
        super(base);
    }

    public double calcolaPrezzo() {
        return pizzaBase.calcolaPrezzo() + 2.0;
    }
}

class SenzaGlutine extends VariantePizza {
    public SenzaGlutine(Pizza base) {
        super(base);
    }

    public double calcolaPrezzo() {
        return pizzaBase.calcolaPrezzo() + 3.5;
    }
}

// Uso
Pizza base = menu.crea("Margherita");
Pizza finale = new SenzaGlutine(new ConBordoRipieno(base));
```

Livello 4: Composizione Moderna (non corso)

```
record Ingrediente(String nome, int grammi, double costo) {}

record RicettaPizza(
    String nome,
    List<Ingrediente> ingredienti,
    TipoImpasto impasto,
    Dimensione dimensione
) {
    public double calcolaCosto() {
        return ingredienti.stream()
```

```
                .mapToDouble(Ingrediente::costo)
                .sum();
    }

    public RicettaPizza conVariante(UnaryOperator<Builder> modifiche) {
        Builder builder = new Builder(this);
        return modifiche.apply(builder).build();
    }

    static class Builder {
        // builder interno per modifiche immutabili
    }
}

// Database ricette
class RepositoryPizze {
    private Map<String, RicettaPizza> ricette;

    public Optional<RicettaPizza> trova(String nome) {
        return Optional.ofNullable(ricette.get(nome));
    }
}

// Service layer
class ServizioOrdini {
    private RepositoryPizze repo;

    public OrdinePizza crea(String nomePizza, List<String> modifiche) {
        return repo.trova(nomePizza)
            .map(ricetta -> applicaModifiche(ricetta, modifiche))
            .map(OrdinePizza::new)
            .orElseThrow();
    }
}
```

Sintesi Evoluzione

1. **Base**: classe astratta per comportamento comune
2. **Builder**: costruzione flessibile
3. **Prototype**: riuso ricette standard
4. **Strategy/Decorator**: varianti comportamentali
5. **Composizione**: immutabilità + functional style

La scelta dipende da: complessità menu, frequenza personalizzazioni, team size.