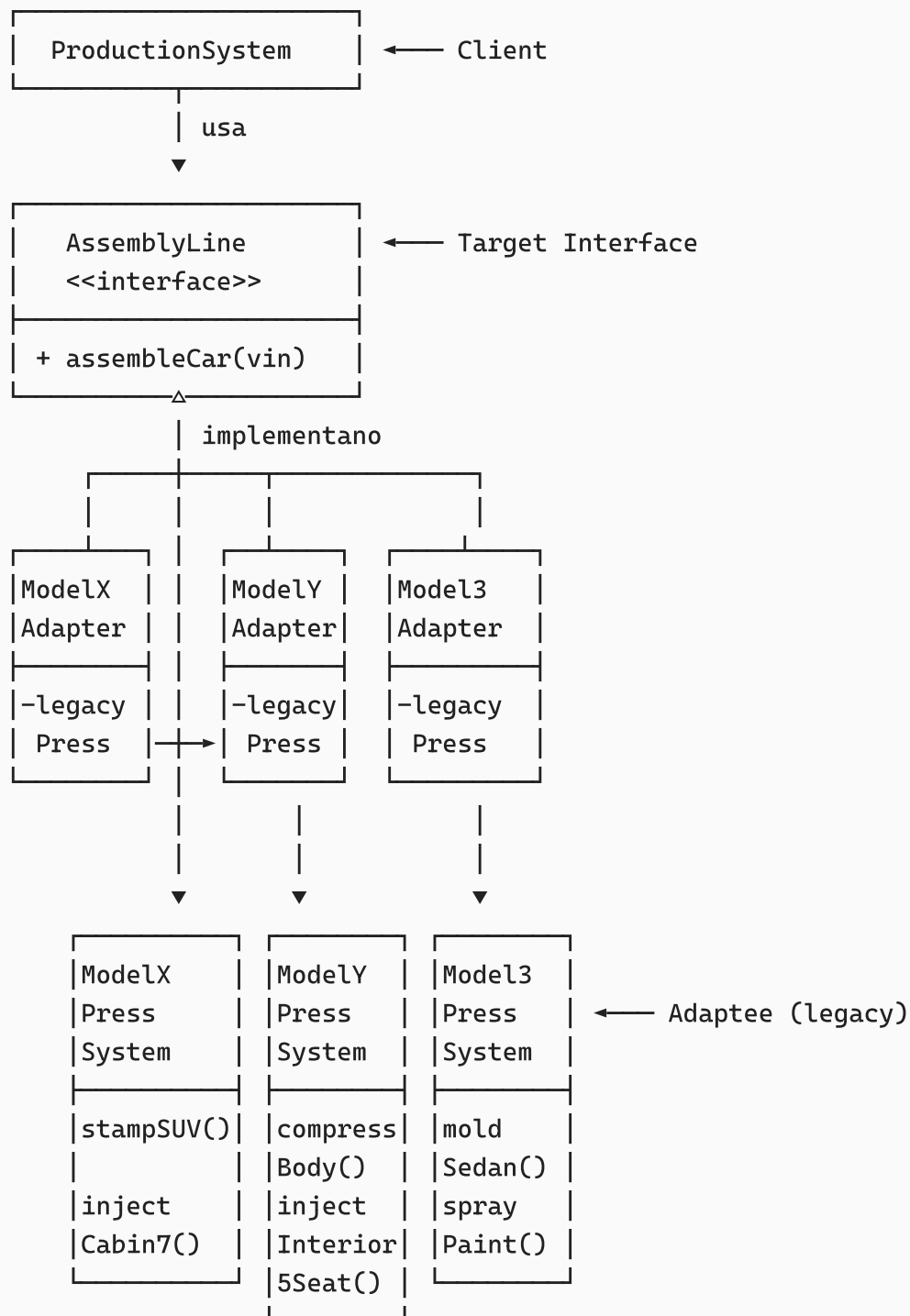


# 1. Adapter Pattern per Linee di Assemblaggio Tesla

## Razionale

Scenario: hai **macchinari legacy incompatibili** per ogni modello. Ogni pressa ha API proprietaria diversa, ma vuoi un'interfaccia uniforme `AssemblyLine` per il sistema di produzione.

## Diagramma



# Implementazione

```
// Target: interfaccia uniforme per il sistema di produzione
public interface AssemblyLine {
    Car assembleCar(String vin);
    String getSupportedModel();
}

// Adaptee 1: Sistema legacy Model X (API proprietaria Fremont)
public class ModelXPressSystem {
    public void initializeSUV Mold(String identifier) {
        System.out.println("[ModelX Legacy Press] Initializing SUV mold for " + identifier);
    }

    public void stampSUVBody(int pressure, boolean falconDoors) {
        System.out.println("[ModelX Legacy Press] Stamping body at " + pressure + " PSI, falcon doors: " + falconDoors);
    }

    public void injectCabin7Seats(String materialCode) {
        System.out.println("[ModelX Legacy Press] Injecting 7-seat cabin with material: " + materialCode);
    }

    public boolean runDiagnostics() {
        System.out.println("[ModelX Legacy Press] Running diagnostics...");
        return true;
    }
}

// Adaptee 2: Sistema legacy Model Y (API proprietaria Shanghai)
public class ModelYPressSystem {
    public void startCompression(String jobId) {
        System.out.println("[ModelY Legacy Press] Starting compression job: " + jobId);
    }

    public void compressBodyPanel(double tonnage) {
        System.out.println("[ModelY Legacy Press] Compressing at " + tonnage + " tonnes");
    }

    public void injectInterior5Seat() {
        System.out.println("[ModelY Legacy Press] Injecting 5-seat vegan interior");
    }

    public int getCompletionStatus() {
```

```

        System.out.println("[ModelY Legacy Press] Checking status...");
        return 100; // percentage
    }
}

// Adaptee 3: Sistema legacy Model 3 (API proprietaria Berlin)
public class Model3PressSystem {
    public void moldSedanProfile(String vin, int temperature) {
        System.out.println("[Model3 Legacy Press] Molding sedan at " +
temperature + "°C for VIN: " + vin);
    }

    public void sprayPaint(String colorCode) {
        System.out.println("[Model3 Legacy Press] Spraying paint: " +
colorCode);
    }

    public void assembleMinimalistInterior() {
        System.out.println("[Model3 Legacy Press] Assembling minimalist
interior");
    }

    public void finalizeQC(String qcCode) {
        System.out.println("[Model3 Legacy Press] QC finalization: " +
qcCode);
    }
}

// Adapter 1: traduce AssemblyLine → ModelXPressSystem
public class ModelXAdapter implements AssemblyLine {
    private final ModelXPressSystem legacyPress;

    public ModelXAdapter(ModelXPressSystem legacyPress) {
        this.legacyPress = legacyPress;
    }

    @Override
    public Car assembleCar(String vin) {
        System.out.println("\n=== ModelX Assembly START (via Adapter) ===");

        // Traduzione chiamate legacy
        legacyPress.initializeSUVMold(vin);
        legacyPress.stampSUVBody(8500, true); // parametri fissi per Model
X
        legacyPress.injectCabin7Seats("LEATHER-PREM-001");

        boolean qcPassed = legacyPress.runDiagnostics();

        Car car = new Car(vin, "Model X", "SUV");
        car.setQualityPassed(qcPassed);
    }
}

```

```

        System.out.println("=== ModelX Assembly COMPLETE ===\n");
        return car;
    }

    @Override
    public String getSupportedModel() {
        return "MODEL_X";
    }
}

// Adapter 2: traduce AssemblyLine → ModelYPressSystem
public class ModelYAdapter implements AssemblyLine {
    private final ModelYPressSystem legacyPress;

    public ModelYAdapter(ModelYPressSystem legacyPress) {
        this.legacyPress = legacyPress;
    }

    @Override
    public Car assembleCar(String vin) {
        System.out.println("\n=== ModelY Assembly START (via Adapter) ===");

        // Traduzione chiamate legacy con logica diversa
        legacyPress.startCompression("JOB-" + vin);
        legacyPress.compressBodyPanel(650.0); // tonnellate
        legacyPress.injectInterior5Seat();

        int completion = legacyPress.getCompletionStatus();
        boolean qcPassed = (completion == 100);

        Car car = new Car(vin, "Model Y", "Compact SUV");
        car.setQualityPassed(qcPassed);

        System.out.println("=== ModelY Assembly COMPLETE ===\n");
        return car;
    }

    @Override
    public String getSupportedModel() {
        return "MODEL_Y";
    }
}

// Adapter 3: traduce AssemblyLine → Model3PressSystem
public class Model3Adapter implements AssemblyLine {
    private final Model3PressSystem legacyPress;

    public Model3Adapter(Model3PressSystem legacyPress) {
        this.legacyPress = legacyPress;
    }
}

```

```

    }

    @Override
    public Car assembleCar(String vin) {
        System.out.println("\n=== Model3 Assembly START (via Adapter) ===");

        // Traduzione con sequenza diversa
        legacyPress.moldSedanProfile(vin, 180);
        legacyPress.sprayPaint("PEARL-WHITE");
        legacyPress.assembleMinimalistInterior();
        legacyPress.finalizeQC("QC-BERLIN-" + System.currentTimeMillis());

        Car car = new Car(vin, "Model 3", "Sedan");
        car.setQualityPassed(true);

        System.out.println("=== Model3 Assembly COMPLETE ===\n");
        return car;
    }

    @Override
    public String getSupportedModel() {
        return "MODEL_3";
    }
}

// Domain object
public class Car {
    private final String vin;
    private final String model;
    private final String bodyStyle;
    private boolean qualityPassed;

    public Car(String vin, String model, String bodyStyle) {
        this.vin = vin;
        this.model = model;
        this.bodyStyle = bodyStyle;
    }

    public void setQualityPassed(boolean passed) {
        this.qualityPassed = passed;
    }

    @Override
    public String toString() {
        return String.format("Car[VIN=%s, Model=%s, Body=%s, QC=%s]",
            vin, model, bodyStyle, qualityPassed ? "PASS" : "FAIL");
    }
}

// Factory per configurare gli adapter con i sistemi legacy

```

```

public class AssemblyLineFactory {
    private static final Map<String, AssemblyLine> assemblyLines = new
    HashMap<>();

    static {
        // Inizializzazione sistemi legacy e relativi adapter
        assemblyLines.put("MODEL_X", new ModelXAdapter(new
    ModelXPRESSSystem()));
        assemblyLines.put("MODEL_Y", new ModelYAdapter(new
    ModelYPRESSSystem()));
        assemblyLines.put("MODEL_3", new Model3Adapter(new
    Model3PRESSSystem()));
    }

    public static AssemblyLine getAssemblyLine(String model) {
        AssemblyLine line = assemblyLines.get(model.toUpperCase());
        if (line == null) {
            throw new IllegalArgumentException("No assembly line for model:
" + model);
        }
        return line;
    }
}

// Production System (Client): usa solo l'interfaccia uniforme
public class ProductionSystem {
    private final List<AssemblyLine> availableLines;

    public ProductionSystem(List<AssemblyLine> lines) {
        this.availableLines = lines;
    }

    public void processOrders(List<ProductionOrder> orders) {
        for (ProductionOrder order : orders) {
            AssemblyLine line =
    AssemblyLineFactory.getAssemblyLine(order.model());
            Car car = line.assembleCar(order.vin());

            System.out.println("✓ Produced: " + car);
        }
    }
}

// Main
public class TeslaFactoryWithAdapters {
    public static void main(String[] args) {
        List<ProductionOrder> orders = List.of(
            new ProductionOrder("VIN-X-001", "MODEL_X"),
            new ProductionOrder("VIN-Y-002", "MODEL_Y"),
            new ProductionOrder("VIN-3-003", "MODEL_3"),

```

```

        new ProductionOrder("VIN-X-004", "MODEL_X")
    );

    List<AssemblyLine> lines = List.of(
        AssemblyLineFactory.getAssemblyLine("MODEL_X"),
        AssemblyLineFactory.getAssemblyLine("MODEL_Y"),
        AssemblyLineFactory.getAssemblyLine("MODEL_3")
    );

    ProductionSystem production = new ProductionSystem(lines);
    production.processOrders(orders);
}
}

record ProductionOrder(String vin, String model) {}

```

## Output

```

=== ModelX Assembly START (via Adapter) ===
[ModelX Legacy Press] Initializing SUV mold for VIN-X-001
[ModelX Legacy Press] Stamping body at 8500 PSI, falcon doors: true
[ModelX Legacy Press] Injecting 7-seat cabin with material: LEATHER-PREM-001
[ModelX Legacy Press] Running diagnostics...
=== ModelX Assembly COMPLETE ===

✓ Produced: Car[VIN=VIN-X-001, Model=Model X, Body=SUV, QC=PASS]

=== ModelY Assembly START (via Adapter) ===
[ModelY Legacy Press] Starting compression job: JOB-VIN-Y-002
[ModelY Legacy Press] Compressing at 650.0 tonnes
[ModelY Legacy Press] Injecting 5-seat vegan interior
[ModelY Legacy Press] Checking status...
=== ModelY Assembly COMPLETE ===

✓ Produced: Car[VIN=VIN-Y-002, Model=Model Y, Body=Compact SUV, QC=PASS]

```

## Quando Usare Adapter vs Template Method

Aspetto	Adapter	Template Method
<b>Problema</b>	Sistemi legacy incompatibili	Stesso algoritmo, step variabili
<b>Controllo codice</b>	Non possiedi gli Adaptee	Possiedi tutto
<b>Variabilità</b>	API completamente diverse	Stesso flusso, override puntuali
<b>Riuso logica</b>	Zero condivisione	Massima condivisione
<b>Accoppiamento</b>	Zero tra adapter	Ereditarietà forte

Nel caso Tesla con presse legacy da fabbriche diverse (Fremont, Shanghai, Berlin) con API proprietarie **non modificabili**, Adapter è la scelta corretta.

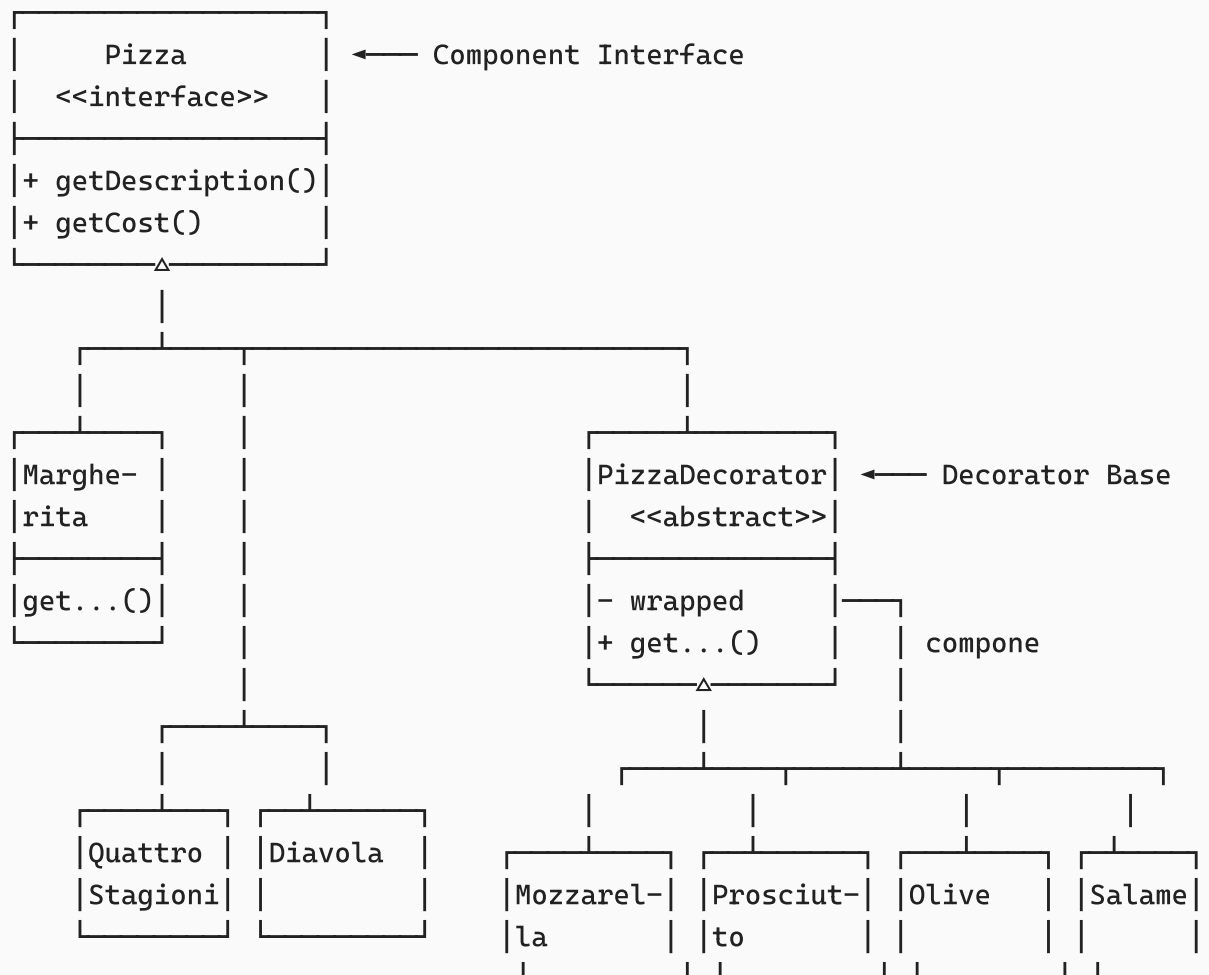
## 2. Decorator Pattern per Pizze

### Razionale

Problema: hai una pizza base e vuoi aggiungere ingredienti dinamicamente a runtime. Ogni ingrediente modifica descrizione e prezzo. Con ereditarietà esploderesti con `PizzaMargheritaConMozzarellaEOlive`, `PizzaMargheritaConProsciutto`, ecc.

**Decorator** wrappa l'oggetto base, aggiunge comportamento, delega al wrapped.

### Diagramma



Esempio wrapping:

Margherita → wrappata da Mozzarella → wrappata da Olive → wrappata da Prosciutto

### Implementazione

```
// Component: interfaccia base
public interface Pizza {
```

```

        String getDescription();
        BigDecimal getCost();
    }

    // Concrete Component 1: Margherita base
    public class Margherita implements Pizza {
        @Override
        public String getDescription() {
            return "Pizza Margherita (pomodoro, basilico)";
        }

        @Override
        public BigDecimal getCost() {
            return new BigDecimal("5.50");
        }
    }

    // Concrete Component 2: Quattro Stagioni base
    public class QuattroStagioni implements Pizza {
        @Override
        public String getDescription() {
            return "Pizza Quattro Stagioni (pomodoro, funghi, carciofi,
prosciutto cotto, olive)";
        }

        @Override
        public BigDecimal getCost() {
            return new BigDecimal("8.00");
        }
    }

    // Concrete Component 3: Diavola base
    public class Diavola implements Pizza {
        @Override
        public String getDescription() {
            return "Pizza Diavola (pomodoro, salame piccante)";
        }

        @Override
        public BigDecimal getCost() {
            return new BigDecimal("7.00");
        }
    }

    // Decorator Base: wrappa una Pizza e delega
    public abstract class PizzaDecorator implements Pizza {
        protected final Pizza wrappedPizza;

        public PizzaDecorator(Pizza pizza) {
            this.wrappedPizza = pizza;
        }
    }

```

```

    }

    @Override
    public String getDescription() {
        return wrappedPizza.getDescription();
    }

    @Override
    public BigDecimal getCost() {
        return wrappedPizza.getCost();
    }
}

// Concrete Decorator 1: Mozzarella
public class ExtraMozzarella extends PizzaDecorator {
    public ExtraMozzarella(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return wrappedPizza.getDescription() + ", extra mozzarella";
    }

    @Override
    public BigDecimal getCost() {
        return wrappedPizza.getCost().add(new BigDecimal("1.50"));
    }
}

// Concrete Decorator 2: Prosciutto Crudo
public class ProsciuttoCrudo extends PizzaDecorator {
    public ProsciuttoCrudo(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return wrappedPizza.getDescription() + ", prosciutto crudo";
    }

    @Override
    public BigDecimal getCost() {
        return wrappedPizza.getCost().add(new BigDecimal("2.50"));
    }
}

// Concrete Decorator 3: Olive
public class Olive extends PizzaDecorator {
    public Olive(Pizza pizza) {

```

```

        super(pizza);
    }

    @Override
    public String getDescription() {
        return wrappedPizza.getDescription() + ", olive nere";
    }

    @Override
    public BigDecimal getCost() {
        return wrappedPizza.getCost().add(new BigDecimal("1.00"));
    }
}

// Concrete Decorator 4: Salame Piccante
public class SalamePiccante extends PizzaDecorator {
    public SalamePiccante(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return wrappedPizza.getDescription() + ", salame piccante extra";
    }

    @Override
    public BigDecimal getCost() {
        return wrappedPizza.getCost().add(new BigDecimal("2.00"));
    }
}

// Concrete Decorator 5: Funghi
public class Funghi extends PizzaDecorator {
    public Funghi(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return wrappedPizza.getDescription() + ", funghi champignon";
    }

    @Override
    public BigDecimal getCost() {
        return wrappedPizza.getCost().add(new BigDecimal("1.20"));
    }
}

// Concrete Decorator 6: Rucola
public class Rucola extends PizzaDecorator {

```

```

    public Rucola(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return wrappedPizza.getDescription() + ", rucola fresca";
    }

    @Override
    public BigDecimal getCost() {
        return wrappedPizza.getCost().add(new BigDecimal("1.00"));
    }
}

// Concrete Decorator 7: Gorgonzola
public class Gorgonzola extends PizzaDecorator {
    public Gorgonzola(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return wrappedPizza.getDescription() + ", gorgonzola DOP";
    }

    @Override
    public BigDecimal getCost() {
        return wrappedPizza.getCost().add(new BigDecimal("2.00"));
    }
}

// Sistema di ordinazione
public class Pizzeria {
    public static void main(String[] args) {
        // Ordine 1: Margherita con extra mozzarella e olive
        Pizza order1 = new Margherita();
        order1 = new ExtraMozzarella(order1);
        order1 = new Olive(order1);

        printOrder(1, order1);

        // Ordine 2: Diavola con salame piccante extra, funghi e gorgonzola
        Pizza order2 = new Diavola();
        order2 = new SalamePiccante(order2);
        order2 = new Funghi(order2);
        order2 = new Gorgonzola(order2);

        printOrder(2, order2);
    }
}

```

```

        // Ordine 3: Quattro Stagioni con prosciutto crudo e rucola (pizza gourmet)
        Pizza order3 = new QuattroStagioni();
        order3 = new ProsciuttoCrudo(order3);
        order3 = new Rucola(order3);
        order3 = new ExtraMozzarella(order3);

        printOrder(3, order3);

        // Ordine 4: Margherita semplice
        Pizza order4 = new Margherita();

        printOrder(4, order4);

        // Ordine 5: Decorazione estrema
        Pizza order5 = new Margherita();
        order5 = new ExtraMozzarella(order5);
        order5 = new ProsciuttoCrudo(order5);
        order5 = new Olive(order5);
        order5 = new Funghi(order5);
        order5 = new Rucola(order5);

        printOrder(5, order5);
    }

    private static void printOrder(int orderNumber, Pizza pizza) {
        System.out.println("=====");
        System.out.println("ORDINE #" + orderNumber);
        System.out.println("-----");
        System.out.println(pizza.getDescription());
        System.out.println("-----");
        System.out.println("TOTALE: €" + pizza.getCost());
        System.out.println("=====\\n");
    }
}

```

## Output

```

=====
ORDINE #1
-----
Pizza Margherita (pomodoro, basilico), extra mozzarella, olive nere
-----
TOTALE: €8.00
=====

=====
ORDINE #2
-----

```

Pizza Diavola (pomodoro, salame piccante), salame piccante extra, funghi champignon, gorgonzola DOP

TOTALE: €12.20

ORDINE #3

Pizza Quattro Stagioni (pomodoro, funghi, carciofi, prosciutto cotto, olive), prosciutto crudo, rucola fresca, extra mozzarella

TOTALE: €13.00

ORDINE #4

Pizza Margherita (pomodoro, basilico)

TOTALE: €5.50

ORDINE #5

Pizza Margherita (pomodoro, basilico), extra mozzarella, prosciutto crudo, olive nere, funghi champignon, rucola fresca

TOTALE: €11.20

## Catena di Wrapping (Ordine #5)

Rucola wraps ↓	getCost() → 1.00 + wrapped.getCost()
Funghi wraps ↓	getCost() → 1.20 + wrapped.getCost()
Olive wraps ↓	getCost() → 1.00 + wrapped.getCost()
ProsciuttoCrudo wraps ↓	getCost() → 2.50 + wrapped.getCost()
ExtraMozzarella wraps ↓	getCost() → 1.50 + wrapped.getCost()

Margherita (base)

getCost() → 5.50

Totale: 5.50 + 1.50 + 2.50 + 1.00 + 1.20 + 1.00 = €11.20

## Vantaggi

1. **Open/Closed Principle**: aggiungi ingredienti senza modificare classi esistenti
2. **Single Responsibility**: ogni decorator gestisce UN ingrediente
3. **Composizione dinamica**: assembli a runtime, non a compile-time
4. **No esplosione combinatoria**: con 10 ingredienti eviti  $2^{10} = 1024$  sottoclassi

## Confronto con Alternative

Approccio	Classi per 5 ingredienti	Flessibilità runtime
Ereditarietà	$2^5 = 32$ sottoclassi	Zero
Decorator	5 decorator + base	Totale
Builder	1 classe + builder	Totale ma meno trasparente

## Quando NON usare Decorator

Se hai bisogno di rimuovere ingredienti o modificare l'ordine del wrapping dopo la costruzione, considera **Builder Pattern** con stato mutabile.

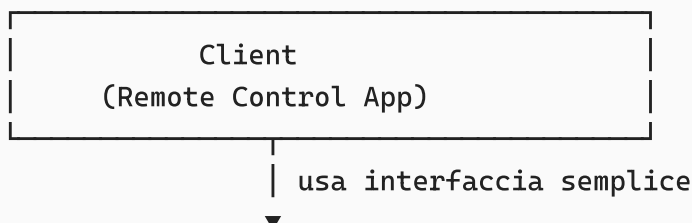
## 3. Facade Pattern per Home Theater System

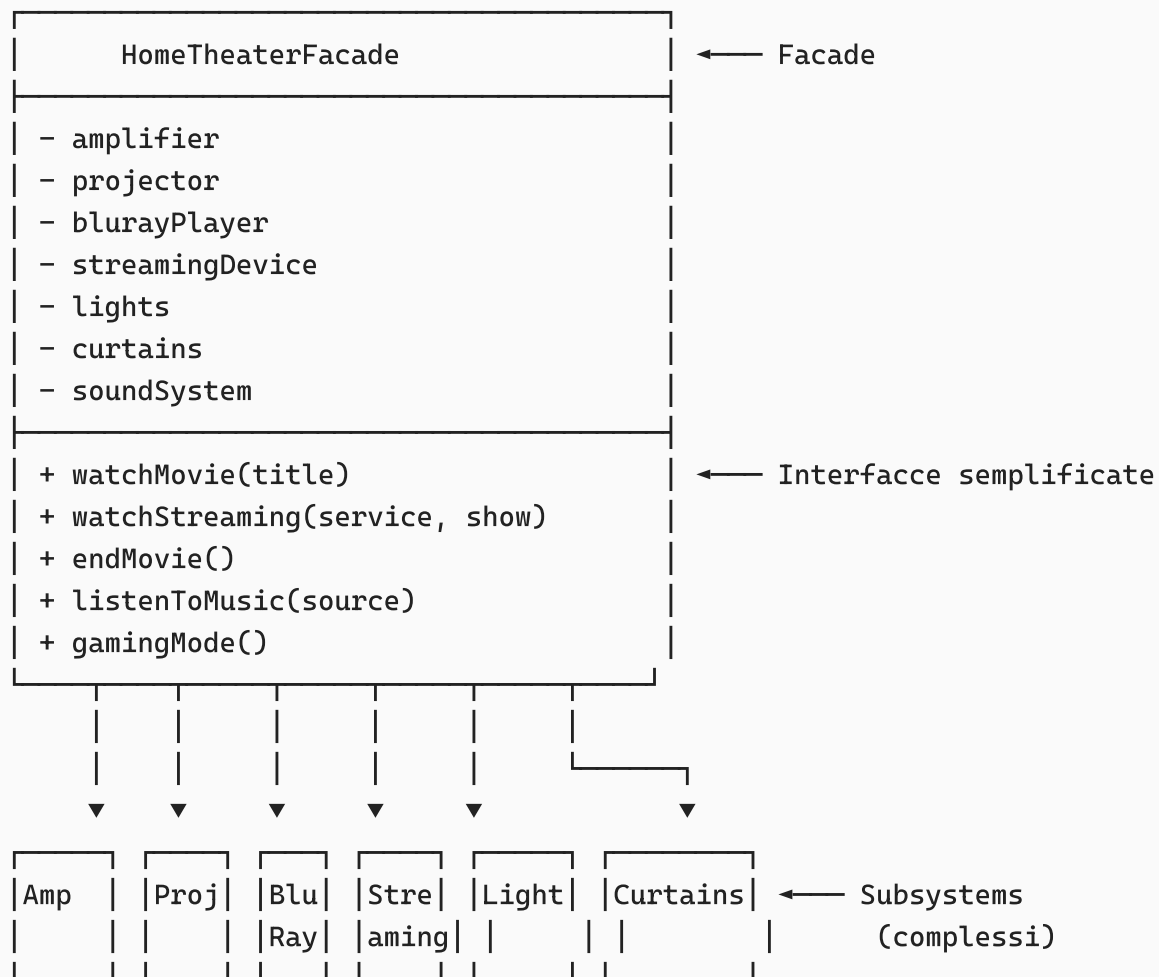
### Razionale

Problema: hai un sistema home theater complesso con **10+ sottosistemi** (amplificatore, proiettore, lettore Blu-ray, luci smart, tende motorizzate, ecc.). Ogni componente ha API articolata. L'utente vuole semplicemente "guarda un film" senza orchestrare manualmente 15 chiamate.

**Facade** fornisce un'interfaccia semplificata che nasconde la complessità dei sottosistemi.

### Diagramma





## Implementazione

```
// ===== SUBSYSTEMS (complessi, API articolate) =====

// Subsystem 1: Amplificatore
public class Amplifier {
    private String description;
    private int volume;
    private String inputSource;

    public Amplifier(String description) {
        this.description = description;
    }

    public void on() {
        System.out.println(description + " acceso");
    }

    public void off() {
        System.out.println(description + " spento");
    }

    public void setInputSource(String source) {
        this.inputSource = source;
    }
}
```

```

        System.out.println(description + " input impostato su: " + source);
    }

    public void setSurroundSound() {
        System.out.println(description + " modalità surround 7.1 attivata");
    }

    public void setStereoSound() {
        System.out.println(description + " modalità stereo attivata");
    }

    public void setVolume(int level) {
        this.volume = level;
        System.out.println(description + " volume impostato a: " + level);
    }
}

// Subsystem 2: Proiettore 4K
public class Projector {
    private String description;
    private boolean wideScreenMode;

    public Projector(String description) {
        this.description = description;
    }

    public void on() {
        System.out.println(description + " acceso");
    }

    public void off() {
        System.out.println(description + " spento");
    }

    public void setWideScreenMode() {
        this.wideScreenMode = true;
        System.out.println(description + " modalità widescreen (21:9)
attivata");
    }

    public void setStandardMode() {
        this.wideScreenMode = false;
        System.out.println(description + " modalità standard (16:9)
attivata");
    }

    public void calibrateColor() {
        System.out.println(description + " calibrazione colore HDR10 in
corso...");
    }
}

```

```

}

// Subsystem 3: Lettore Blu-ray
public class BlurayPlayer {
    private String description;

    public BlurayPlayer(String description) {
        this.description = description;
    }

    public void on() {
        System.out.println(description + " acceso");
    }

    public void off() {
        System.out.println(description + " spento");
    }

    public void eject() {
        System.out.println(description + " espulsione disco");
    }

    public void play(String movie) {
        System.out.println(description + " riproduzione film: " + movie);
    }

    public void stop() {
        System.out.println(description + " riproduzione fermata");
    }

    public void enableHDRMode() {
        System.out.println(description + " HDR Dolby Vision abilitato");
    }
}

// Subsystem 4: Streaming Device
public class StreamingDevice {
    private String description;

    public StreamingDevice(String description) {
        this.description = description;
    }

    public void on() {
        System.out.println(description + " acceso");
    }

    public void off() {
        System.out.println(description + " spento");
    }
}

```

```

    public void openApp(String appName) {
        System.out.println(description + " apertura app: " + appName);
    }

    public void play(String content) {
        System.out.println(description + " riproduzione: " + content);
    }

    public void setVideoQuality(String quality) {
        System.out.println(description + " qualità video: " + quality);
    }
}

// Subsystem 5: Sistema luci smart
public class SmartLights {
    private int brightness;

    public void dim(int level) {
        this.brightness = level;
        System.out.println("Luci attenuate al " + level + "%");
    }

    public void on() {
        this.brightness = 100;
        System.out.println("Luci accese al 100%");
    }

    public void off() {
        this.brightness = 0;
        System.out.println("Luci spente");
    }

    public void setColorTemperature(int kelvin) {
        System.out.println("Temperatura colore impostata: " + kelvin + "K");
    }
}

// Subsystem 6: Tende motorizzate
public class MotorizedCurtains {
    public void close() {
        System.out.println("Tende chiuse completamente");
    }

    public void open() {
        System.out.println("Tende aperte completamente");
    }

    public void setPosition(int percentage) {
        System.out.println("Tende posizionate al " + percentage + "%");
    }
}

```

```

    }
}

// Subsystem 7: Sistema audio surround
public class SoundSystem {
    private String description;

    public SoundSystem(String description) {
        this.description = description;
    }

    public void on() {
        System.out.println(description + " acceso");
    }

    public void off() {
        System.out.println(description + " spento");
    }

    public void setMode(String mode) {
        System.out.println(description + " modalità audio: " + mode);
    }

    public void calibrateRoom() {
        System.out.println(description + " calibrazione acustica ambiente in
corso...");
    }
}

// ===== FACADE (interfaccia semplificata) =====

public class HomeTheaterFacade {
    // Riferimenti a tutti i sottosistemi
    private final Amplifier amplifier;
    private final Projector projector;
    private final BlurayPlayer blurayPlayer;
    private final StreamingDevice streamingDevice;
    private final SmartLights lights;
    private final MotorizedCurtains curtains;
    private final SoundSystem soundSystem;

    public HomeTheaterFacade(
        Amplifier amp,
        Projector projector,
        BlurayPlayer bluray,
        StreamingDevice streaming,
        SmartLights lights,
        MotorizedCurtains curtains,
        SoundSystem sound) {
        this.amplifier = amp;

```

```

    this.projector = projector;
    this.blurayPlayer = bluray;
    this.streamingDevice = streaming;
    this.lights = lights;
    this.curtains = curtains;
    this.soundSystem = sound;
}

// Operazione semplificata: guardare un film Blu-ray
public void watchMovie(String movie) {
    System.out.println("\n=====");
    System.out.println("PREPARAZIONE VISIONE FILM: " + movie);
    System.out.println("=====");

    // Orchestrazione automatica di 15+ chiamate
    curtains.close();
    lights.dim(10);
    lights.setColorTemperature(2700);

    projector.on();
    projector.setWideScreenMode();
    projector.calibrateColor();

    soundSystem.on();
    soundSystem.calibrateRoom();
    soundSystem.setMode("Cinema Dolby Atmos");

    amplifier.on();
    amplifier.setInputSource("Blu-ray");
    amplifier.setSurroundSound();
    amplifier.setVolume(45);

    blurayPlayer.on();
    blurayPlayer.enableHDRMode();
    blurayPlayer.play(movie);

    System.out.println("=====");
    System.out.println("✓ Sistema pronto. Buona visione!");
    System.out.println("=====\\n");
}

// Operazione semplificata: streaming Netflix/Prime
public void watchStreaming(String service, String show) {
    System.out.println("\n=====");
    System.out.println("PREPARAZIONE STREAMING: " + show);
    System.out.println("=====");

    curtains.close();
    lights.dim(15);

```

```

projector.on();
projector.setStandardMode();

soundSystem.on();
soundSystem.setMode("TV Show");

amplifier.on();
amplifier.setInputSource("Streaming");
amplifier.setStereoSound();
amplifier.setVolume(35);

streamingDevice.on();
streamingDevice.setVideoQuality("4K HDR");
streamingDevice.openApp(service);
streamingDevice.play(show);

System.out.println("=====");
System.out.println("✓ Streaming avviato. Buona visione!");
System.out.println("=====\\n");
}

// Operazione semplificata: terminare la visione
public void endMovie() {
    System.out.println("\\n=====");
    System.out.println("SPEGNIMENTO SISTEMA");
    System.out.println("=====");

    blurayPlayer.stop();
    blurayPlayer.eject();
    blurayPlayer.off();

    streamingDevice.off();

    amplifier.off();
    soundSystem.off();
    projector.off();

    lights.on();
    curtains.open();

    System.out.println("=====");
    System.out.println("✓ Sistema spento completamente");
    System.out.println("=====\\n");
}

// Operazione semplificata: ascoltare musica
public void listenToMusic(String source) {
    System.out.println("\\n=====");
    System.out.println("MODALITÀ MUSICA");
    System.out.println("=====");

```

```

        lights.dim(40);
        lights.setColorTemperature(3500);

        soundSystem.on();
        soundSystem.setMode("Stereo Hi-Fi");

        amplifier.on();
        amplifier.setInputSource(source);
        amplifier.setStereoSound();
        amplifier.setVolume(30);

        System.out.println("=====");
        System.out.println("✓ Sistema audio pronto");
        System.out.println("=====\\n");
    }

    // Operazione semplificata: gaming mode
    public void gamingMode() {
        System.out.println("\\n=====");
        System.out.println("MODALITÀ GAMING");
        System.out.println("=====");

        curtains.close();
        lights.dim(30);
        lights.setColorTemperature(6500); // luce fredda per concentrazione

        projector.on();
        projector.setStandardMode();

        soundSystem.on();
        soundSystem.setMode("Gaming (Low Latency)");

        amplifier.on();
        amplifier.setInputSource("HDMI-Gaming");
        amplifier.setSurroundSound();
        amplifier.setVolume(40);

        System.out.println("=====");
        System.out.println("✓ Gaming mode attivo. Have fun!");
        System.out.println("=====\\n");
    }
}

// ===== CLIENT (usa solo la facade) =====

public class HomeTheaterClient {
    public static void main(String[] args) {
        // Inizializzazione sottosistemi (complessità nascosta)
        Amplifier amp = new Amplifier("Denon AVR-X4700H");
    }
}

```

```

Projector projector = new Projector("Sony VPL-VW915ES 4K");
BlurayPlayer bluray = new BlurayPlayer("Panasonic DP-UB9000");
StreamingDevice streaming = new StreamingDevice("Apple TV 4K");
SmartLights lights = new SmartLights();
MotorizedCurtains curtains = new MotorizedCurtains();
SoundSystem sound = new SoundSystem("Klipsch Reference Premiere
7.2.4");

// Facade: interfaccia semplificata
HomeTheaterFacade homeTheater = new HomeTheaterFacade(
    amp, projector, bluray, streaming, lights, curtains, sound
);

// Scenario 1: Serata film Blu-ray
homeTheater.watchMovie("Dune: Part Two [4K UHD]");

// Pausa...
pause(2000);

// Fine film
homeTheater.endMovie();

// Scenario 2: Serie TV su Netflix
pause(1000);
homeTheater.watchStreaming("Netflix", "Stranger Things S05E01");

pause(2000);
homeTheater.endMovie();

// Scenario 3: Ascolto musica
pause(1000);
homeTheater.listenToMusic("Spotify");

pause(2000);

// Scenario 4: Gaming session
homeTheater.gamingMode();
}

private static void pause(int milliseconds) {
    try {
        Thread.sleep(milliseconds);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
}

```

## Output (Estratto)

```

=====
PREPARAZIONE VISIONE FILM: Dune: Part Two [4K UHD]
=====
Tende chiuse completamente
Luci attenuate al 10%
Temperatura colore impostata: 2700K
Sony VPL-VW915ES 4K acceso
Sony VPL-VW915ES 4K modalità widescreen (21:9) attivata
Sony VPL-VW915ES 4K calibrazione colore HDR10 in corso...
Klipsch Reference Premiere 7.2.4 acceso
Klipsch Reference Premiere 7.2.4 calibrazione acustica ambiente in corso...
Klipsch Reference Premiere 7.2.4 modalità audio: Cinema Dolby Atmos
Denon AVR-X4700H acceso
Denon AVR-X4700H input impostato su: Blu-ray
Denon AVR-X4700H modalità surround 7.1 attivata
Denon AVR-X4700H volume impostato a: 45
Panasonic DP-UB9000 acceso
Panasonic DP-UB9000 HDR Dolby Vision abilitato
Panasonic DP-UB9000 riproduzione film: Dune: Part Two [4K UHD]
=====
✓ Sistema pronto. Buona visione!
=====

=====
SPEGNIMENTO SISTEMA
=====
Panasonic DP-UB9000 riproduzione fermata
Panasonic DP-UB9000 espulsione disco
Panasonic DP-UB9000 spento
Apple TV 4K spento
Denon AVR-X4700H spento
Klipsch Reference Premiere 7.2.4 spento
Sony VPL-VW915ES 4K spento
Luci accese al 100%
Tende aperte completamente
=====
✓ Sistema spento completamente
=====

```

## Vantaggi

1. **Semplicità client:** `watchMovie()` vs 15+ chiamate manuali
2. **Disaccoppiamento:** client ignora sottosistemi (Amplifier, Projector, ecc.)
3. **Manutenibilità:** modifiche ai sottosistemi non impattano client
4. **Riuso:** stesse operazioni complesse disponibili ovunque
5. **Testabilità:** mock della facade invece di 7 sottosistemi

# Quando Usare Facade

- Sistema complesso con molti componenti interdipendenti
- Operazioni comuni che richiedono orchestrazione di più sottosistemi
- Necessità di layer di astrazione per client diversi (app mobile, web, vocale)

## Differenza con Adapter

Pattern	Scopo	Componenti
Facade	Semplificare interfaccia complessa	1 facade → N subsystems
Adapter	Rendere compatibile interfaccia incompatibile	1 adapter → 1 adaptee

## 4. Virtual Proxy per Video

Scenario: player video stile VLC. Problema: caricare tutti i video in memoria al lancio dell'applicazione è inutile e costoso se l'utente non li visualizza tutti.

**Soluzione:** Virtual Proxy che carica il video solo quando effettivamente riprodotto.

```
// Subject: interfaccia comune
public interface Video {
    void playVideo();
}

// RealSubject: oggetto pesante che vogliamo caricare lazy
public class RealVideo implements Video {
    private final byte[] fileContent;

    public RealVideo(byte[] fileContent) {
        this.fileContent = fileContent;
        System.out.println("Loading video from disk... (expensive operation)");
    }

    @Override
    public void playVideo() {
        System.out.println("Reproducing video...");
    }
}

// Proxy: surrogato leggero
public class VideoProxy implements Video {
    private final String filePath;
    private RealVideo realVideo; // lazy initialization

    public VideoProxy(String filePath) {
```

```

        this.filePath = filePath;
        // NO caricamento qui - solo path
    }

    @Override
    public void playVideo() {
        if (realVideo == null) {
            // Caricamento on-demand
            byte[] bytes = loadBytesFromDisk(filePath);
            realVideo = new RealVideo(bytes);
        }
        realVideo.playVideo();
    }

    private byte[] loadBytesFromDisk(String path) {
        // Simulazione I/O
        return new byte[0];
    }
}

// Client
public class VideoGallery {
    private List<Video> videos;

    public VideoGallery() {
        videos = new ArrayList<>();
        // Creazione proxy - operazione leggera
        videos.add(new VideoProxy("/path/video1.mp4"));
        videos.add(new VideoProxy("/path/video2.mp4"));
        videos.add(new VideoProxy("/path/video3.mp4"));
        // Nessun video caricato in memoria
    }

    public void playVideo(int index) {
        videos.get(index).playVideo(); // Ora il caricamento avviene
    }
}

```

### Flusso esecutivo:

1. Creazione VideoGallery : istanzia 3 proxy (leggeri, solo path)
2. playVideo(0) : primo accesso → carica RealVideo da disco → esegue play
3. playVideo(0) : secondo accesso → RealVideo già in memoria → esegue play direttamente
4. playVideo(1) : carica secondo video on-demand

**Invariante:** il client interagisce solo con Video , trasparenza totale tra proxy e real subject.