Pattern di Creazione

I pattern di creazione forniscono vari meccanismi per la creazione di oggetti, aumentando la flessibilità e il riutilizzo del codice esistente.

Factory Method

Descrizione

Fornisce un'interfaccia per la creazione di oggetti in una superclasse, ma permette alle sottoclassi di modificare il tipo di oggetti che verranno creati.

Esempi nel Mondo Reale

1. **Creazione Componenti UI**
   - Creazione di diversi tipi di pulsanti (primario, secondario, pericolo) in base al contesto
   - Costruzione di diversi elementi di form (input testo, select, checkbox) in base al tipo di campo
   - Generazione di diversi tipi di notifiche (successo, errore, avviso)
   - Creazione di diversi tipi di modali (conferma, informazione, errore)

Example Implementation

```
// Product interface
public interface Button {
    void render();
}

// Concrete Products
public class PrimaryButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering primary button");
    }
}

public class SecondaryButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering secondary button");
    }
}

// Creator interface
```

```
public interface ButtonCreator {
    Button createButton();
}

// Concrete Creators
public class PrimaryButtonCreator implements ButtonCreator {
    @Override
    public Button createButton() {
        return new PrimaryButton();
    }
}

public class SecondaryButtonCreator implements ButtonCreator {
    @Override
    public Button createButton() {
        return new SecondaryButton();
    }
}

// Usage
public class FactoryMethodExample {
    public static void main(String[] args) {
        ButtonCreator primaryCreator = new PrimaryButtonCreator();
        ButtonCreator secondaryCreator = new SecondaryButtonCreator();

        Button primaryBtn = primaryCreator.createButton();
        Button secondaryBtn = secondaryCreator.createButton();

        primaryBtn.render(); // Renders primary button
        secondaryBtn.render(); // Renders secondary button
    }
}
```

Abstract Factory

Descrizione

Ti permette di produrre famiglie di oggetti correlati senza specificare le loro classi concrete.

Esempi nel Mondo Reale

1. **Sistema di Temi**
   - Creazione di componenti UI con diversi temi (chiaro, scuro, personalizzato)
   - Generazione di diversi set di icone per varie piattaforme
   - Costruzione di diversi set di controlli di form per diversi tipi di input
   - Creazione di diversi set di componenti di layout per il design responsive

Example Implementation

```java
// Abstract Products
public interface Button {
    void render();
}

public interface Input {
    void render();
}

// Concrete Products for Light Theme
public class LightButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering light button");
    }
}

public class LightInput implements Input {
    @Override
    public void render() {
        System.out.println("Rendering light input");
    }
}

// Concrete Products for Dark Theme
public class DarkButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering dark button");
    }
}

public class DarkInput implements Input {
    @Override
    public void render() {
        System.out.println("Rendering dark input");
    }
}

// Abstract Factory
public interface ThemeFactory {
    Button createButton();
    Input createInput();
}

// Concrete Factories
public class LightThemeFactory implements ThemeFactory {
    @Override
    public Button createButton() {
        return new LightButton();
    }
```

```
    @Override
    public Input createInput() {
        return new LightInput();
    }
}

public class DarkThemeFactory implements ThemeFactory {
    @Override
    public Button createButton() {
        return new DarkButton();
    }

    @Override
    public Input createInput() {
        return new DarkInput();
    }
}

// Usage
public class AbstractFactoryExample {
    public static void main(String[] args) {
        ThemeFactory lightFactory = new LightThemeFactory();
        ThemeFactory darkFactory = new DarkThemeFactory();

        Button lightButton = lightFactory.createButton();
        Input lightInput = lightFactory.createInput();

        Button darkButton = darkFactory.createButton();
        Input darkInput = darkFactory.createInput();

        lightButton.render(); // Renders light button
        lightInput.render(); // Renders light input

        darkButton.render(); // Renders dark button
        darkInput.render(); // Renders dark input
    }
}
```

Builder

Descrizione

Ti permette di costruire oggetti complessi passo dopo passo. Il pattern ti permette di produrre diversi tipi e rappresentazioni di un oggetto utilizzando lo stesso codice di costruzione.

Esempi nel Mondo Reale

1. **Costruttore di Form**
    - Creazione di form complessi con diversi tipi di campi e regole di validazione

- Costruzione di sondaggi dinamici con domande condizionali
- Generazione di report complessi con più sezioni e fonti dati
- Creazione di dashboard personalizzabili con diversi widget

Example Implementation

```
// Product
public class Form {
    private List<String> fields = new ArrayList<>();
    private List<String> validations = new ArrayList<>();
    private String layout = "default";

    public void addField(String field) {
        fields.add(field);
    }

    public void addValidation(String validation) {
        validations.add(validation);
    }

    public void setLayout(String layout) {
        this.layout = layout;
    }

    public void render() {
        System.out.println("Rendering form with " + fields.size() + " fields");
        System.out.println("Validations: " + String.join(", ", validations));
        System.out.println("Layout: " + layout);
    }
}

// Builder
public abstract class FormBuilder {
    protected Form form;

    public FormBuilder() {
        this.form = new Form();
    }

    public abstract void addTextField(String name);
    public abstract void addSelectField(String name, List<String> options);
    public abstract void addCheckboxField(String name);

    public Form getForm() {
        return form;
    }
}

// Concrete Builder
public class AdvancedFormBuilder extends FormBuilder {
    @Override
```

```
    public void addTextField(String name) {
        form.addField(name);
        form.addValidation("required");
    }

    @Override
    public void addSelectField(String name, List<String> options) {
        form.addField(name);
        form.addValidation("required");
        if (!options.isEmpty()) {
            form.addValidation("minOptions");
        }
    }

    @Override
    public void addCheckboxField(String name) {
        form.addField(name);
        form.addValidation("minChecked");
    }
}

// Director
public class FormDirector {
    public static Form buildContactForm(FormBuilder builder) {
        builder.addTextField("name");
        builder.addTextField("email");
        builder.addSelectField("country", Arrays.asList("Italy", "France", "Germany"))
        builder.addCheckboxField("newsletter");
        return builder.getForm();
    }
}

// Usage
public class BuilderExample {
    public static void main(String[] args) {
        FormBuilder builder = new AdvancedFormBuilder();
        Form form = FormDirector.buildContactForm(builder);
        form.render();
    }
}
```

Prototype

Descrizione

Ti permette di copiare oggetti esistenti senza rendere il tuo codice dipendente dalle loro classi.

Esempi nel Mondo Reale

1. **Configurazione Componenti**

- Creazione di più istanze di un componente con variazioni lievi
- Generazione di diverse configurazioni per lo stesso tipo di componente
- Creazione di più istanze di un form con valori predefiniti diversi
- Generazione di diversi layout da un template base

Example Implementation

```
// Prototype interface
public interface ComponentPrototype {
    ComponentPrototype clone();
    void configure(Map<String, Object> config);
}

// Concrete Prototype
public class ButtonComponent implements ComponentPrototype {
    private Map<String, Object> properties;

    public ButtonComponent() {
        properties = new HashMap<>();
        properties.put("type", "primary");
        properties.put("size", "medium");
        properties.put("disabled", false);
    }

    @Override
    public ComponentPrototype clone() {
        try {
            ButtonComponent clone = (ButtonComponent) super.clone();
            clone.properties = new HashMap<>(this.properties);
            return clone;
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("Clone not supported", e);
        }
    }

    @Override
    public void configure(Map<String, Object> config) {
        properties.putAll(config);
    }

    public void render() {
        System.out.println("Rendering button with properties: " + properties);
    }
}

// Usage
public class PrototypeExample {
    public static void main(String[] args) {
        ButtonComponent baseButton = new ButtonComponent();
        baseButton.configure(Map.of(
```

```
            "type", "primary",
            "size", "medium",
            "disabled", false
        ));

        ButtonComponent submitButton = (ButtonComponent) baseButton.clone();
        submitButton.configure(Map.of(
            "type", "primary",
            "size", "large",
            "disabled", false
        ));

        ButtonComponent cancelButton = (ButtonComponent) baseButton.clone();
        cancelButton.configure(Map.of(
            "type", "secondary",
            "size", "medium",
            "disabled", false
        ));

        submitButton.render(); // Renders primary large button
cancelButton.render(); // Renders secondary medium button
    }
}
```

Singleton

Descrizione

Ti permette di assicurarti che una classe abbia solo un'istanza, fornendo un punto di accesso globale per questa istanza.

Esempi nel Mondo Reale

1. **Gestione Stato Applicazione**
   - Gestione dello stato globale dell'applicazione
   - Gestione dello stato di autenticazione utente
   - Gestione della configurazione API
   - Gestione delle preferenze di tema

Example Implementation

```
// Singleton class
public class AppState {
    private static AppState instance;
    private Map<String, Object> state;

    private AppState() {
```

```
        state = new HashMap<>();
        state.put("theme", "light");
        state.put("language", "en");
        state.put("user", null);
    }

    public static synchronized AppState getInstance() {
        if (instance == null) {
            instance = new AppState();
        }
        return instance;
    }

    public Map<String, Object> getState() {
        return state;
    }

    public void setState(Map<String, Object> newState) {
        state.putAll(newState);
    }
}

// Usage
public class SingletonExample {
    public static void main(String[] args) {
        AppState appState1 = AppState.getInstance();
        AppState appState2 = AppState.getInstance();

        System.out.println(appState1 == appState2); // true

        appState1.setState(Map.of("theme", "dark"));
        System.out.println(appState2.getState().get("theme")); // 'dark'
    }
}
```

Best Practice

1. Usa Factory Method quando:
    - Vuoi centralizzare la logica di creazione degli oggetti
    - Vuoi nascondere la logica di creazione degli oggetti ai client
    - Vuoi fornire un modo flessibile per creare oggetti
2. Usa Abstract Factory quando:
    - Devi creare famiglie di oggetti correlati
    - Vuoi assicurarti una creazione coerente degli oggetti attraverso diversi temi/varianti
    - Vuoi decouplare la creazione degli oggetti dal codice client

3. Usa Builder quando:
    - Devi creare oggetti complessi con molte parti opzionali
    - Vuoi nascondere la logica di creazione degli oggetti ai client
    - Devi creare oggetti in più passaggi
4. Usa Prototype quando:
    - Devi creare più istanze di un oggetto con variazioni lievi
    - Vuoi evitare logiche complesse di creazione degli oggetti
    - Devi creare oggetti simili a quelli esistenti
5. Usa Singleton quando:
    - Devi assicurarti che una classe abbia solo un'istanza
    - Vuoi fornire un punto di accesso globale a un oggetto
    - Devi controllare l'accesso a risorse condivise

Pattern Strutturali

I pattern strutturali spiegano come assemblare oggetti e classi in strutture più grandi, mantenendo queste strutture flessibili ed efficienti.

Adapter

Descrizione

Permette a oggetti con interfacce incompatibili di collaborare.

Esempi nel Mondo Reale

1. **Integrazione API**
    - Adattamento di API esterne a interfacce standard
    - Conversione di formati dati tra sistemi diversi
    - Integrazione di servizi legacy con sistemi moderni
    - Adattamento di librerie di terze parti

Example Implementation

```
// Target interface
public interface PaymentService {
    void processPayment(double amount);
}
```

```java
// Adaptee class
public class ExternalPaymentSystem {
    public void processTransaction(double amount) {
        System.out.println("Processing transaction of " + amount);
    }
}

// Adapter class
public class PaymentAdapter implements PaymentService {
    private ExternalPaymentSystem externalSystem;

    public PaymentAdapter(ExternalPaymentSystem externalSystem) {
        this.externalSystem = externalSystem;
    }

    @Override
    public void processPayment(double amount) {
        externalSystem.processTransaction(amount);
    }
}

// Usage
public class AdapterExample {
    public static void main(String[] args) {
        ExternalPaymentSystem externalSystem = new ExternalPaymentSystem();
        PaymentService paymentService = new PaymentAdapter(externalSystem);
        paymentService.processPayment(100.0);
    }
}
```

Bridge

Descrizione

Ti permette di separare una classe o un insieme di classi correlate in due gerarchie separate - astrazione e implementazione - che possono essere sviluppate indipendentemente l'una dall'altra.

Esempi nel Mondo Reale

1. **Rendering UI**
    - Separazione della logica di rendering dal tipo di dispositivo
    - Implementazione di diversi stili per lo stesso componente
    - Gestione di diversi motori di rendering
    - Supporto per diverse piattaforme

Example Implementation

```java
// Abstraction interface
public interface DrawingAPI {
    void drawCircle(double x, double y, double radius);
}

// Concrete Implementations
public class DrawingAPI1 implements DrawingAPI {
    @Override
    public void drawCircle(double x, double y, double radius) {
        System.out.println("API1.circle at " + x + ":" + y + ":" + radius);
    }
}

public class DrawingAPI2 implements DrawingAPI {
    @Override
    public void drawCircle(double x, double y, double radius) {
        System.out.println("API2.circle at " + x + ":" + y + ":" + radius);
    }
}

// Abstraction class
public class CircleShape {
    protected DrawingAPI drawingAPI;
    protected double x, y, radius;

    public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {
        this.x = x;
        this.y = y;
        this.radius = radius;
        this.drawingAPI = drawingAPI;
    }

    public void draw() {
        drawingAPI.drawCircle(x, y, radius);
    }
}

// Usage
public class BridgeExample {
    public static void main(String[] args) {
        CircleShape circle1 = new CircleShape(1, 2, 3, new DrawingAPI1());
        CircleShape circle2 = new CircleShape(5, 7, 11, new DrawingAPI2());

        circle1.draw();
        circle2.draw();
    }
}
```

Composite

Descrizione

Ti permette di comporre oggetti in strutture ad albero e di lavorare con queste strutture come se fossero oggetti singoli.

Esempi nel Mondo Reale

1. **Menu e Sottomenu**
    - Struttura ad albero di menu e sottomenu
    - Gestione di componenti UI annidati
    - Organizzazione di sezioni e sottosezioni
    - Gerarchia di componenti

Example Implementation

```
// Component interface
public interface MenuComponent {
    void add(MenuComponent component);
    void remove(MenuComponent component);
    MenuComponent getChild(int index);
    String getName();
    double getPrice();
    void print();
}

// Leaf
public class MenuItem implements MenuComponent {
    private String name;
    private double price;

    public MenuItem(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public void add(MenuComponent component) {
        throw new UnsupportedOperationException();
    }

    @Override
    public void remove(MenuComponent component) {
        throw new UnsupportedOperationException();
    }

    @Override
    public MenuComponent getChild(int index) {
        throw new UnsupportedOperationException();
    }
```

```
    @Override
    public String getName() {
        return name;
    }

    @Override
    public double getPrice() {
        return price;
    }

    @Override
    public void print() {
        System.out.println(" " + getName());
        System.out.println(". " + getPrice());
    }
}

// Composite
public class Menu implements MenuComponent {
    private List<MenuComponent> menuComponents = new ArrayList<>();
    private String name;

    public Menu(String name) {
        this.name = name;
    }

    @Override
    public void add(MenuComponent component) {
        menuComponents.add(component);
    }

    @Override
    public void remove(MenuComponent component) {
        menuComponents.remove(component);
    }

    @Override
    public MenuComponent getChild(int index) {
        return menuComponents.get(index);
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public double getPrice() {
        return 0;
    }

    @Override
```

```
    public void print() {
        System.out.println("\n" + getName());
        System.out.println("--------------------");

        for (MenuComponent component : menuComponents) {
            component.print();
        }
    }
}

// Usage
public class CompositeExample {
    public static void main(String[] args) {
        MenuComponent pancakeHouseMenu = new Menu("Pancake House Menu");
        MenuComponent dinerMenu = new Menu("Diner Menu");
        MenuComponent cafeMenu = new Menu("Cafe Menu");
        MenuComponent dessertMenu = new Menu("Dessert Menu");

        MenuComponent allMenus = new Menu("All Menus");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        pancakeHouseMenu.add(new MenuItem("K&B's Pancake Breakfast", 2.99));
        pancakeHouseMenu.add(new MenuItem("Regular Pancake Breakfast", 2.99));
        pancakeHouseMenu.add(new MenuItem("Blueberry Pancakes", 3.49));
        pancakeHouseMenu.add(new MenuItem("Waffles", 3.59));

        dinerMenu.add(new MenuItem("Vegetarian BLT", 2.99));
        dinerMenu.add(new MenuItem("BLT", 2.99));
        dinerMenu.add(new MenuItem("Soup of the day", 3.29));
        dinerMenu.add(new MenuItem("Hotdog", 3.05));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem("Apple Pie", 1.59));
        dessertMenu.add(new MenuItem("Cheesecake", 1.99));
        dessertMenu.add(new MenuItem("Sorbet", 1.89));

        allMenus.print();
    }
}
```

Decorator

Descrizione

Ti permette di aggiungere nuovi comportamenti agli oggetti inserendoli all'interno di oggetti wrapper speciali che contengono i comportamenti.

Esempi nel Mondo Reale

1. **Componenti UI**
    - Aggiunta di bordi e sfondi a componenti
    - Implementazione di effetti di hover e focus
    - Gestione di stati di disabilitazione
    - Aggiunta di icone e indicatori

Example Implementation

```
// Component interface
public interface Button {
    void render();
    void onClick();
}

// Concrete Component
public class SimpleButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering simple button");
    }

    @Override
    public void onClick() {
        System.out.println("Button clicked");
    }
}

// Decorator base
public abstract class ButtonDecorator implements Button {
    protected Button wrappedButton;

    public ButtonDecorator(Button button) {
        this.wrappedButton = button;
    }

    @Override
    public void render() {
        wrappedButton.render();
    }

    @Override
    public void onClick() {
        wrappedButton.onClick();
    }
}

// Concrete Decorators
public class DisabledButtonDecorator extends ButtonDecorator {
```

```
    public DisabledButtonDecorator(Button button) {
        super(button);
    }

    @Override
    public void render() {
        System.out.println("Rendering disabled button");
        super.render();
    }

    @Override
    public void onClick() {
        System.out.println("Button is disabled");
    }
}

public class HoverButtonDecorator extends ButtonDecorator {
    public HoverButtonDecorator(Button button) {
        super(button);
    }

    @Override
    public void render() {
        System.out.println("Rendering hover button");
        super.render();
    }
}

// Usage
public class DecoratorExample {
    public static void main(String[] args) {
        Button simpleButton = new SimpleButton();
        Button disabledButton = new DisabledButtonDecorator(simpleButton);
        Button hoverButton = new HoverButtonDecorator(simpleButton);

        disabledButton.render();
        disabledButton.onClick();

        hoverButton.render();
        hoverButton.onClick();
    }
}
```

Facade

Descrizione

Fornisce un'interfaccia semplificata per una libreria, un framework o un insieme complesso di classi.

Esempi nel Mondo Reale

1. **Sistema di File**
    - Gestione semplificata di operazioni su file
    - Interazione con sistemi di file diversi
    - Gestione di configurazioni complesse
    - Integrazione di servizi

Example Implementation

```
// Complex subsystems
public class FileSubsystem {
    public void createFile(String path) {
        System.out.println("Creating file at " + path);
    }

    public void deleteFile(String path) {
        System.out.println("Deleting file at " + path);
    }
}

public class DirectorySubsystem {
    public void createDirectory(String path) {
        System.out.println("Creating directory at " + path);
    }

    public void deleteDirectory(String path) {
        System.out.println("Deleting directory at " + path);
    }
}

// Facade
public class FileSystemFacade {
    private FileSubsystem fileSubsystem = new FileSubsystem();
    private DirectorySubsystem directorySubsystem = new DirectorySubsystem();

    public void createFile(String path) {
        fileSubsystem.createFile(path);
    }

    public void deleteFile(String path) {
        fileSubsystem.deleteFile(path);
    }

    public void createDirectory(String path) {
        directorySubsystem.createDirectory(path);
    }

    public void deleteDirectory(String path) {
        directorySubsystem.deleteDirectory(path);
```

```
    }
}

// Usage
public class FacadeExample {
    public static void main(String[] args) {
        FileSystemFacade fileSystem = new FileSystemFacade();

        fileSystem.createDirectory("/home/user/documents");
        fileSystem.createFile("/home/user/documents/file.txt");
        fileSystem.deleteFile("/home/user/documents/file.txt");
        fileSystem.deleteDirectory("/home/user/documents");
    }
}
```

Flyweight

Descrizione

Ti permette di inserire più oggetti nella quantità di RAM disponibile condividendo parti comuni dello stato tra più oggetti invece di mantenere tutti i dati in ogni oggetto.

Esempi nel Mondo Reale

1. **Gestione UI**
    - Condivisione di icone comuni
    - Gestione di componenti simili
    - Ottimizzazione di risorse grafiche
    - Gestione di stati di componenti

Example Implementation

```
// Flyweight interface
public interface ButtonFlyweight {
    void render(String label, String style);
}

// Concrete Flyweight
public class ButtonImplementation implements ButtonFlyweight {
    private String type;
    private String size;

    public ButtonImplementation(String type, String size) {
        this.type = type;
        this.size = size;
    }
```

```
    @Override
    public void render(String label, String style) {
        System.out.println("Rendering button:");
        System.out.println("Type: " + type);
        System.out.println("Size: " + size);
        System.out.println("Label: " + label);
        System.out.println("Style: " + style);
    }
}

// Flyweight Factory
public class ButtonFactory {
    private Map<String, ButtonFlyweight> buttons = new HashMap<>();

    public ButtonFlyweight getButton(String type, String size) {
        String key = type + ":" + size;
        if (!buttons.containsKey(key)) {
            buttons.put(key, new ButtonImplementation(type, size));
        }
        return buttons.get(key);
    }
}

// Usage
public class FlyweightExample {
    public static void main(String[] args) {
        ButtonFactory factory = new ButtonFactory();

        ButtonFlyweight button1 = factory.getButton("primary", "medium");
        ButtonFlyweight button2 = factory.getButton("primary", "medium");
        ButtonFlyweight button3 = factory.getButton("secondary", "small");

        button1.render("Save", "blue");
        button2.render("Submit", "blue");
        button3.render("Cancel", "gray");
    }
}
```

Proxy

Descrizione

Ti permette di fornire un sostituto o un segnaposto per un altro oggetto. Un proxy controlla l'accesso all'oggetto originale, permettendoti di eseguire qualcosa prima o dopo che la richiesta raggiunga l'oggetto originale.

Esempi nel Mondo Reale

1. **Gestione Risorse**

- Caricamento lazy di risorse
- Cache di risorse
- Gestione di connessioni
- Controllo di accesso

Example Implementation

```
// Subject interface
public interface Image {
    void display();
}

// Real Subject
public class RealImage implements Image {
    private String fileName;

    public RealImage(String fileName) {
        this.fileName = fileName;
        loadFromDisk();
    }

    private void loadFromDisk() {
        System.out.println("Loading image: " + fileName);
    }

    @Override
    public void display() {
        System.out.println("Displaying image: " + fileName);
    }
}

// Proxy
public class ImageProxy implements Image {
    private RealImage realImage;
    private String fileName;

    public ImageProxy(String fileName) {
        this.fileName = fileName;
    }

    @Override
    public void display() {
        if (realImage == null) {
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}

// Usage
```

```
public class ProxyExample {
    public static void main(String[] args) {
        Image image1 = new ImageProxy("image1.jpg");
        Image image2 = new ImageProxy("image2.jpg");

        // Images will be loaded only when needed
        image1.display();
        image2.display();
    }
}
```

Best Practice per Pattern Strutturali

1. Usa Adapter quando:
    - Devi integrare classi con interfacce incompatibili
    - Vuoi evitare modifiche al codice esistente
    - Devi convertire formati o protocolli
2. Usa Bridge quando:
    - Devi evitare una gerarchia di classi stretta
    - Vuoi separare l'implementazione dall'astrazione
    - Devi permettere la modifica indipendente di classi correlate
3. Usa Composite quando:
    - Devi rappresentare una gerarchia ad albero di oggetti
    - Vuoi trattare oggetti singoli e gruppi di oggetti in modo uniforme
    - Devi gestire strutture annidate
4. Usa Decorator quando:
    - Devi aggiungere comportamenti a oggetti in modo dinamico
    - Vuoi evitare sottoclassi per ogni combinazione di comportamenti
    - Devi aggiungere funzionalità a oggetti individualmente
5. Usa Facade quando:
    - Devi fornire un'interfaccia semplificata a un sistema complesso
    - Vuoi nascondere la complessità del sistema
    - Devi fornire un punto di accesso unico a un insieme di classi
6. Usa Flyweight quando:
    - Devi gestire un gran numero di oggetti simili

- Vuoi ottimizzare l'uso della memoria
- Devi condividere dati comuni tra oggetti

7. Usa Proxy quando:
   - Devi controllare l'accesso a un oggetto
   - Vuoi implementare il caricamento lazy
   - Devi aggiungere funzionalità di sicurezza
   - Vuoi fornire un'interfaccia locale per un oggetto remoto

Pattern Comportamentali

I pattern comportamentali sono interessati agli algoritmi e all'assegnazione delle responsabilità tra gli oggetti.

Chain of Responsibility

Descrizione

Ti permette di passare richieste lungo una catena di gestori. Ogni gestore decide se processare la richiesta o passarla al gestore successivo nella catena.

Esempi nel Mondo Reale

1. **Gestione Eventi**
   - Gestione di eventi UI in ordine specifico
   - Processamento di comandi in sequenza
   - Validazione di form in cascata
   - Gestione di errori in catena

Example Implementation

```
// Handler interface
public interface EventHandler {
    void setNext(EventHandler next);
    void handle(Event event);
}

// Concrete Handlers
public class AuthHandler implements EventHandler {
    private EventHandler next;

    @Override
    public void setNext(EventHandler next) {
```

```
        this.next = next;
    }

    @Override
    public void handle(Event event) {
        if (event.getType() == EventType.AUTH) {
            System.out.println("Handling auth event: " + event.getData());
        } else if (next != null) {
            next.handle(event);
        }
    }
}

public class ValidationHandler implements EventHandler {
    private EventHandler next;

    @Override
    public void setNext(EventHandler next) {
        this.next = next;
    }

    @Override
    public void handle(Event event) {
        if (event.getType() == EventType.VALIDATION) {
            System.out.println("Handling validation event: " + event.getData());
        } else if (next != null) {
            next.handle(event);
        }
    }
}

// Event class
public class Event {
    private EventType type;
    private String data;

    public Event(EventType type, String data) {
        this.type = type;
        this.data = data;
    }

    public EventType getType() {
        return type;
    }

    public String getData() {
        return data;
    }
}

// EventType enum
public enum EventType {
    AUTH, VALIDATION, OTHER
```

```
}

// Usage
public class ChainOfResponsibilityExample {
    public static void main(String[] args) {
        EventHandler authHandler = new AuthHandler();
        EventHandler validationHandler = new ValidationHandler();

        authHandler.setNext(validationHandler);

        Event authEvent = new Event(EventType.AUTH, "User login");
        Event validationEvent = new Event(EventType.VALIDATION, "Form validation");

        authHandler.handle(authEvent);
        authHandler.handle(validationEvent);
    }
}
```

Command

Descrizione

Trasforma una richiesta in un oggetto standalone che contiene tutte le informazioni sulla richiesta. Questa trasformazione ti permette di passare richieste come argomenti di metodo, di ritardare o mettere in coda l'esecuzione di una richiesta e di supportare operazioni annullabili.

Esempi nel Mondo Reale

1. **Gestione Azioni**
    - Implementazione di operazioni undo/redo
    - Gestione di comandi UI
    - Logging delle operazioni
    - Queue di operazioni

Example Implementation

```
// Command interface
public interface Command {
    void execute();
    void undo();
}

// Receiver
public class Editor {
    private String content;

    public void setContent(String content) {
```

```
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}

// Concrete Commands
public class BoldCommand implements Command {
    private Editor editor;
    private String previousContent;

    public BoldCommand(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void execute() {
        previousContent = editor.getContent();
        editor.setContent("<b>" + editor.getContent() + "</b>");
    }

    @Override
    public void undo() {
        editor.setContent(previousContent);
    }
}

public class ItalicCommand implements Command {
    private Editor editor;
    private String previousContent;

    public ItalicCommand(Editor editor) {
        this.editor = editor;
    }

    @Override
    public void execute() {
        previousContent = editor.getContent();
        editor.setContent("<i>" + editor.getContent() + "</i>");
    }

    @Override
    public void undo() {
        editor.setContent(previousContent);
    }
}

// Invoker
public class Toolbar {
    private List<Command> commands = new ArrayList<>();
    private int currentCommand = -1;
```

```
    public void executeCommand(Command command) {
        command.execute();
        commands.add(command);
        currentCommand++;
    }

    public void undo() {
        if (currentCommand >= 0) {
            commands.get(currentCommand).undo();
            currentCommand--;
        }
    }

    public void redo() {
        if (currentCommand < commands.size() - 1) {
            commands.get(currentCommand + 1).execute();
            currentCommand++;
        }
    }
}

// Usage
public class CommandExample {
    public static void main(String[] args) {
        Editor editor = new Editor();
        Toolbar toolbar = new Toolbar();

        editor.setContent("Hello World");

        toolbar.executeCommand(new BoldCommand(editor));
        toolbar.executeCommand(new ItalicCommand(editor));

        System.out.println("After commands: " + editor.getContent());
        toolbar.undo();
        System.out.println("After undo: " + editor.getContent());
        toolbar.redo();
        System.out.println("After redo: " + editor.getContent());
    }
}
```

Iterator

Descrizione

Ti permette di attraversare gli elementi di una collezione senza esporre la sua rappresentazione sottostante (lista, stack, albero, ecc.).

Esempi nel Mondo Reale

1. **Gestione Dati**
    - Iterazione su liste di componenti UI
    - Traversamento di alberi di elementi
    - Gestione di stack di operazioni
    - Iterazione su mappe di dati

Example Implementation

```
// Collection interface
public interface Collection {
    Iterator createIterator();
    int size();
    Object get(int index);
}

// Concrete Collection
public class Menu implements Collection {
    private List<String> items = new ArrayList<>();

    public void addItem(String item) {
        items.add(item);
    }

    @Override
    public Iterator createIterator() {
        return new MenuIterator(this);
    }

    @Override
    public int size() {
        return items.size();
    }

    @Override
    public Object get(int index) {
        return items.get(index);
    }
}

// Iterator interface
public interface Iterator {
    boolean hasNext();
    Object next();
}

// Concrete Iterator
public class MenuIterator implements Iterator {
    private Collection collection;
    private int position = 0;
```

```
    public MenuIterator(Collection collection) {
        this.collection = collection;
    }

    @Override
    public boolean hasNext() {
        return position < collection.size();
    }

    @Override
    public Object next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        return collection.get(position++);
    }
}

// Usage
public class IteratorExample {
    public static void main(String[] args) {
        Menu menu = new Menu();
        menu.addItem("Pizza");
        menu.addItem("Pasta");
        menu.addItem("Salad");

        Iterator iterator = menu.createIterator();
        while (iterator.hasNext()) {
            String item = (String) iterator.next();
            System.out.println("Menu item: " + item);
        }
    }
}
```

Mediator

Descrizione

Ti permette di ridurre le dipendenze caotiche tra oggetti. Il pattern limita le comunicazioni dirette tra gli oggetti e li forza a collaborare solo attraverso un oggetto mediatore.

Esempi nel Mondo Reale

1. **Gestione Componenti**
   - Coordinazione di componenti UI
   - Gestione di eventi tra componenti
   - Sincronizzazione di stati
   - Gestione di interazioni

Example Implementation

```
// Mediator interface
public interface ChatMediator {
    void sendMessage(String message, User user);
    void addUser(User user);
}

// Concrete Mediator
public class ChatRoom implements ChatMediator {
    private List<User> users = new ArrayList<>();

    @Override
    public void addUser(User user) {
        users.add(user);
    }

    @Override
    public void sendMessage(String message, User user) {
        for (User u : users) {
            if (u != user) {
                u.receive(message);
            }
        }
    }
}

// User class
public class User {
    private String name;
    private ChatMediator mediator;

    public User(String name, ChatMediator mediator) {
        this.name = name;
        this.mediator = mediator;
        mediator.addUser(this);
    }

    public void send(String message) {
        mediator.sendMessage(message, this);
    }

    public void receive(String message) {
        System.out.println(name + " received: " + message);
    }
}

// Usage
public class MediatorExample {
    public static void main(String[] args) {
        ChatMediator chatRoom = new ChatRoom();
```

```
        User user1 = new User("Alice", chatRoom);
        User user2 = new User("Bob", chatRoom);
        User user3 = new User("Charlie", chatRoom);

        user1.send("Hello everyone!");
        user2.send("Hi Alice!");
    }
}
```

Memento

Descrizione

Ti permette di salvare e ripristinare lo stato precedente di un oggetto senza rivelare i dettagli della sua implementazione.

Esempi nel Mondo Reale

1. **Gestione Stato**
    - Implementazione di operazioni undo/redo
    - Salvataggio di checkpoint
    - Ripristino di configurazioni
    - Gestione di versioni

Example Implementation

```
// Originator
public class Editor {
    private String content;

    public void setContent(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }

    public Memento createMemento() {
        return new Memento(content);
    }

    public void restore(Memento memento) {
        content = memento.getContent();
    }
}
```

```
// Memento
public class Memento {
    private String content;

    public Memento(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}

// Caretaker
public class History {
    private List<Memento> history = new ArrayList<>();
    private int current = -1;

    public void addMemento(Memento memento) {
        // Remove future states
        if (current < history.size() - 1) {
            history.subList(current + 1, history.size()).clear();
        }
        history.add(memento);
        current++;
    }

    public Memento getMemento(int index) {
        return history.get(index);
    }

    public boolean canUndo() {
        return current > 0;
    }

    public boolean canRedo() {
        return current < history.size() - 1;
    }

    public void undo() {
        if (canUndo()) {
            current--;
        }
    }

    public void redo() {
        if (canRedo()) {
            current++;
        }
    }
}

// Usage
```

```
public class MementoExample {
    public static void main(String[] args) {
        Editor editor = new Editor();
        History history = new History();

        editor.setContent("Hello");
        history.addMemento(editor.createMemento());

        editor.setContent("Hello World");
        history.addMemento(editor.createMemento());

        editor.setContent("Hello World!");
        history.addMemento(editor.createMemento());

        System.out.println("Current: " + editor.getContent());
        history.undo();
        editor.restore(history.getMemento(history.current));
        System.out.println("After undo: " + editor.getContent());
        history.redo();
        editor.restore(history.getMemento(history.current));
        System.out.println("After redo: " + editor.getContent());
    }
}
```

Observer

Descrizione

Ti permette di definire un meccanismo di sottoscrizione per notificare più oggetti quando accadono eventi sull'oggetto che stanno osservando.

Esempi nel Mondo Reale

1. **Gestione Eventi**
   - Notifiche di aggiornamenti UI
   - Sincronizzazione di dati
   - Gestione di eventi in tempo reale
   - Aggiornamento di componenti

Example Implementation

```
// Subject interface
public interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers(String message);
}
```

```java
// Concrete Subject
public class DataProvider implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String data;

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    public void setData(String data) {
        this.data = data;
        notifyObservers(data);
    }
}

// Observer interface
public interface Observer {
    void update(String message);
}

// Concrete Observers
public class Chart implements Observer {
    @Override
    public void update(String message) {
        System.out.println("Chart updated with: " + message);
    }
}

public class Table implements Observer {
    @Override
    public void update(String message) {
        System.out.println("Table updated with: " + message);
    }
}

// Usage
public class ObserverExample {
    public static void main(String[] args) {
        DataProvider provider = new DataProvider();
```

```
        Chart chart = new Chart();
        Table table = new Table();

        provider.registerObserver(chart);
        provider.registerObserver(table);

        provider.setData("New data point");
        provider.removeObserver(table);
        provider.setData("Another data point");
    }
}
```

State

Descrizione

Ti permette a un oggetto di modificare il suo comportamento quando il suo stato interno cambia. Sembrerà che l'oggetto abbia cambiato la sua classe.

Esempi nel Mondo Reale

1. **Gestione Stato**
    - Gestione di stati di componenti UI
    - Gestione di stati di form
    - Gestione di stati di applicazione
    - Gestione di stati di sessione

Example Implementation

```
// State interface
public interface State {
    void handle(Context context);
}

// Concrete States
public class LoadingState implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Loading state");
        // Simulate loading
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        context.setState(new ReadyState());
    }
```

```
}

public class ReadyState implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Ready state");
        // Simulate user interaction
        context.setState(new ProcessingState());
    }
}

public class ProcessingState implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Processing state");
        // Simulate processing
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        context.setState(new CompletedState());
    }
}

public class CompletedState implements State {
    @Override
    public void handle(Context context) {
        System.out.println("Completed state");
    }
}

// Context
public class Context {
    private State state;

    public Context() {
        state = new LoadingState();
    }

    public void setState(State state) {
        this.state = state;
        state.handle(this);
    }

    public State getState() {
        return state;
    }
}

// Usage
public class StateExample {
    public static void main(String[] args) {
```

```
        Context context = new Context();
        context.setState(new LoadingState());
    }
}
```

Strategy

Descrizione

Ti permette di definire una famiglia di algoritmi, metterli in classi separate e renderli intercambiabili.

Esempi nel Mondo Reale

1. **Gestione Algoritmi**
    - Implementazione di diversi algoritmi di ordinamento
    - Gestione di diversi algoritmi di ricerca
    - Implementazione di diversi algoritmi di routing
    - Gestione di diversi algoritmi di validazione

Example Implementation

```
// Strategy interface
public interface SortingStrategy {
    void sort(List<Integer> list);
}

// Concrete Strategies
public class BubbleSort implements SortingStrategy {
    @Override
    public void sort(List<Integer> list) {
        System.out.println("Bubble sorting...");
        boolean swapped;
        do {
            swapped = false;
            for (int i = 0; i < list.size() - 1; i++) {
                if (list.get(i) > list.get(i + 1)) {
                    Integer temp = list.get(i);
                    list.set(i, list.get(i + 1));
                    list.set(i + 1, temp);
                    swapped = true;
                }
            }
        } while (swapped);
    }
}

public class QuickSort implements SortingStrategy {
```

```
    @Override
    public void sort(List<Integer> list) {
        System.out.println("Quick sorting...");
        quickSort(list, 0, list.size() - 1);
    }

    private void quickSort(List<Integer> list, int low, int high) {
        if (low < high) {
            int pivot = partition(list, low, high);
            quickSort(list, low, pivot - 1);
            quickSort(list, pivot + 1, high);
        }
    }

    private int partition(List<Integer> list, int low, int high) {
        int pivot = list.get(high);
        int i = low - 1;
        for (int j = low; j < high; j++) {
            if (list.get(j) < pivot) {
                i++;
                Integer temp = list.get(i);
                list.set(i, list.get(j));
                list.set(j, temp);
            }
        }
        Integer temp = list.get(i + 1);
        list.set(i + 1, list.get(high));
        list.set(high, temp);
        return i + 1;
    }
}

// Context
public class Sorter {
    private SortingStrategy strategy;

    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sort(List<Integer> list) {
        strategy.sort(list);
    }
}

// Usage
public class StrategyExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>(Arrays.asList(5, 3, 8, 4, 2));

        Sorter sorter = new Sorter();
        sorter.setStrategy(new BubbleSort());
        sorter.sort(numbers);
```

```
        System.out.println("Sorted with BubbleSort: " + numbers);

        numbers = new ArrayList<>(Arrays.asList(5, 3, 8, 4, 2));
        sorter.setStrategy(new QuickSort());
        sorter.sort(numbers);
        System.out.println("Sorted with QuickSort: " + numbers);
    }
}
```

Template Method

Descrizione

Definisce lo scheletro di un algoritmo nella superclasse ma permette alle sottoclassi di sovrascrivere specifici passaggi dell'algoritmo senza modificare la sua struttura.

Esempi nel Mondo Reale

1. **Gestione Processi**
   - Implementazione di processi di rendering
   - Gestione di pipeline di elaborazione
   - Implementazione di cicli di vita
   - Gestione di flussi di lavoro

Example Implementation

```
// Abstract class with template method
public abstract class Renderer {
    // Template method
    public final void render() {
        prepare();
        draw();
        finalize();
    }

    protected abstract void prepare();

    protected abstract void draw();

    protected abstract void finalize();
}

// Concrete classes
public class ChartRenderer extends Renderer {
    @Override
    protected void prepare() {
        System.out.println("Preparing chart data...");
```

```
    }

    @Override
    protected void draw() {
        System.out.println("Drawing chart...");
    }

    @Override
    protected void finalize() {
        System.out.println("Finalizing chart rendering...");
    }
}

public class TableRenderer extends Renderer {
    @Override
    protected void prepare() {
        System.out.println("Preparing table data...");
    }

    @Override
    protected void draw() {
        System.out.println("Drawing table...");
    }

    @Override
    protected void finalize() {
        System.out.println("Finalizing table rendering...");
    }
}

// Usage
public class TemplateMethodExample {
    public static void main(String[] args) {
        Renderer chartRenderer = new ChartRenderer();
        Renderer tableRenderer = new TableRenderer();

        System.out.println("Rendering chart:");
        chartRenderer.render();

        System.out.println("\nRendering table:");
        tableRenderer.render();
    }
}
```

Visitor

Descrizione

Ti permette di separare gli algoritmi dagli oggetti su cui operano.

Esempi nel Mondo Reale

1. **Gestione Operazioni**
    - Implementazione di operazioni su alberi di componenti
    - Gestione di operazioni su grafici
    - Implementazione di operazioni su liste
    - Gestione di operazioni su mappe

Example Implementation

```
// Element interface
public interface Element {
    void accept(Visitor visitor);
}

// Concrete Elements
public class Button implements Element {
    private String label;

    public Button(String label) {
        this.label = label;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    public String getLabel() {
        return label;
    }
}

public class Text implements Element {
    private String content;

    public Text(String content) {
        this.content = content;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    public String getContent() {
        return content;
    }
}
```

```java
// Visitor interface
public interface Visitor {
    void visit(Button button);
    void visit(Text text);
}

// Concrete Visitors
public class HTMLVisitor implements Visitor {
    @Override
    public void visit(Button button) {
        System.out.println("<button>" + button.getLabel() + "</button>");
    }

    @Override
    public void visit(Text text) {
        System.out.println("<p>" + text.getContent() + "</p>");
    }
}

public class JSONVisitor implements Visitor {
    @Override
    public void visit(Button button) {
        System.out.println("{\"type\":\"button\",\"label\":\"" + button.getLabel() + "
    }

    @Override
    public void visit(Text text) {
        System.out.println("{\"type\":\"text\",\"content\":\"" + text.getContent() + "
    }
}

// Usage
public class VisitorExample {
    public static void main(String[] args) {
        List<Element> elements = new ArrayList<>();
        elements.add(new Button("Click me"));
        elements.add(new Text("Hello World"));

        Visitor htmlVisitor = new HTMLVisitor();
        Visitor jsonVisitor = new JSONVisitor();

        System.out.println("HTML representation:");
        for (Element element : elements) {
            element.accept(htmlVisitor);
        }

        System.out.println("\nJSON representation:");
        for (Element element : elements) {
            element.accept(jsonVisitor);
        }
    }
}
```

Best Practice per Pattern Comportamentali

1. Usa Chain of Responsibility quando:
    - Devi processare una richiesta in modo sequenziale
    - Vuoi evitare l'uso di molte condizioni if-else
    - Devi gestire richieste in modo dinamico
    - Vuoi creare una catena di gestori
2. Usa Command quando:
    - Devi implementare operazioni undo/redo
    - Vuoi parametrizzare oggetti con operazioni
    - Devi mettere in coda richieste
    - Vuoi supportare operazioni annullabili
3. Usa Iterator quando:
    - Devi attraversare elementi di una collezione
    - Vuoi nascondere la rappresentazione della collezione
    - Devi supportare diverse strategie di traversamento
    - Vuoi semplificare l'accesso ai dati
4. Usa Mediator quando:
    - Devi ridurre le dipendenze tra oggetti
    - Vuoi centralizzare la comunicazione
    - Devi gestire interazioni complesse
    - Vuoi semplificare il codice
5. Usa Memento quando:
    - Devi salvare lo stato di un oggetto
    - Vuoi implementare operazioni undo/redo
    - Devi gestire checkpoint
    - Vuoi preservare lo stato senza esporlo
6. Usa Observer quando:
    - Devi implementare un meccanismo di sottoscrizione
    - Vuoi notificare oggetti di eventi
    - Devi gestire aggiornamenti in tempo reale
    - Vuoi decouplare oggetti

7. Usa State quando:
    - Devi gestire stati di un oggetto
    - Vuoi cambiare il comportamento in base allo stato
    - Devi evitare condizioni if-else
    - Vuoi rendere il codice più manutenibile
8. Usa Strategy quando:
    - Devi definire una famiglia di algoritmi
    - Vuoi rendere gli algoritmi intercambiabili
    - Vuoi evitare condizioni if-else
    - Vuoi rendere il codice più flessibile
9. Usa Template Method quando:
    - Devi definire lo scheletro di un algoritmo
    - Vuoi permettere alle sottoclassi di sovrascrivere passaggi
    - Vuoi evitare duplicazione di codice
    - Vuoi rendere il codice più manutenibile
10. Usa Visitor quando:
    - Devi separare algoritmi dagli oggetti
    - Vuoi evitare modifiche alla gerarchia
    - Vuoi aggiungere operazioni dinamicamente
    - Vuoi rendere il codice più flessibile