Azzolini Riccardo 2019-12-03

Design Patterns

1 Design pattern

I problemi incontrati nello sviluppo di grossi progetti software sono spesso ricorrenti e prevedibili. I **design pattern** sono "schemi di soluzioni" riutilizzabili per tali problemi.

Definizione: descrizione di oggetti e classi comunicanti, adattabili per risolvere un problema ricorrente di progettazione in un contesto specifico.

I design pattern sono abbastanza astratti, non particolarmente complessi, e non domain-specific, in modo da poter essere usati in applicazioni per domini diversi.

1.1 Caratteristiche

Un design pattern *nomina*, *astrae*, e *identifica* gli aspetti chiave di una struttura comune e riutilizzabile di design, che la rendono utile nell'ambito dello sviluppo object-oriented.

In particolare, esso identifica

- le classi (e le istanze) partecipanti;
- le associazioni e i ruoli;
- le modalità di collaborazione tra le classi coinvolte;
- la distribuzione delle responsabilità nella soluzione del particolare problema di design considerato.

1.2 Vantaggi

Un design pattern fornisce al progettista software:

- una soluzione consolidata, già testata, per un problema ricorrente;
- un livello di astrazione solitamente più elevato di una classe;
- un modo per progettare software con caratteristiche già note;
- un supporto alla progettazione di sistemi complessi.

Infatti, i design pattern permettono di non inventare da capo soluzioni ai problemi già risolti, ma piuttosto riutilizzare dei "mattoni" di provata efficacia, sfruttando l'esperienza e la "saggezza" degli esperti.

Essi forniscono inoltre un vocabolario comune, che facilita la comunicazione tra progettisti. Infatti, un bravo progettista sa riconoscerli, sia nella documentazione che direttamente nel codice, e utilizzarli per comprendere i programmi scritti da altri.

1.3 Svantaggi

L'uso dei design pattern può rendere la struttura del progetto/codice più complessa del necessario. Per questo, è necessario decidere di volta in volta se adottare semplici soluzioni ad hoc, oppure riutilizzare pattern noti, potenzialmente più complessi. In generale, è utile ricordare i seguenti motti:

- "when in doubt, leave it out": quando si è in dubbio relativamente all'uso di un pattern, è meglio non usarlo;
- "keep it simple": cercare di non complicare il progetto/codice più del necessario.

2 Factory Method

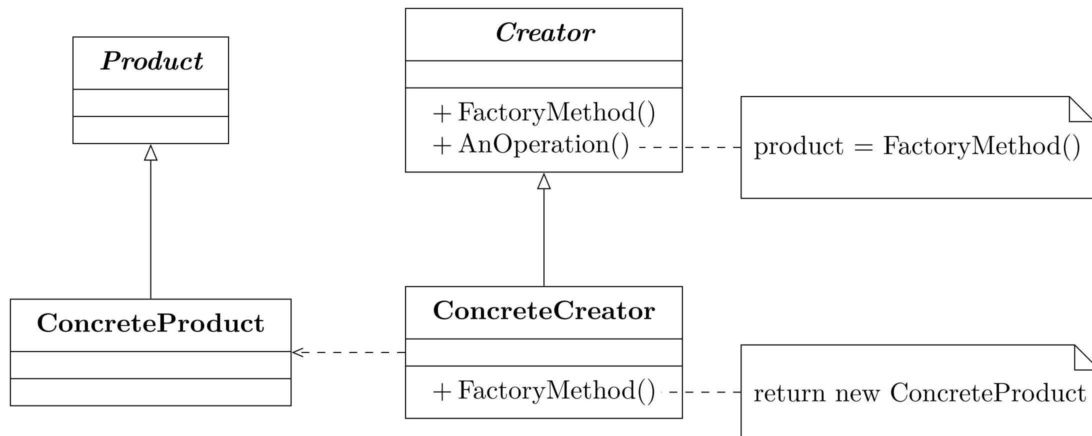
Il codice di un programma ad oggetti per lo più non dipende dalla precisa classe a cui appartiene un certo oggetto (grazie al meccanismo del polimorfismo): serve solo che l'oggetto rispetti il contratto corrispondente al suo tipo. Questo permette di limitare le dipendenze dalle classi, e quindi di sostituire un'implementazione con un'altra.

Ciò non vale, però, per le *chiamate ai costruttori*: il codice utente che chiama il costruttore è obbligato a specificare esattamente la classe da istanziare, e rimane quindi vincolato a essa.

La soluzione è disaccoppiare (separare) il codice che fa uso di un tipo da quello che sceglie quale implementazione del tipo, cioè quale classe, utilizzare: si nasconde l'operazione di creazione in un apposito metodo, detto **factory**, che restituisce un oggetto di una classe, senza essere costruttore di quella classe.

Non essendo un costruttore, il metodo factory può creare oggetti di classi diverse, purché essi abbiano tutti lo stesso tipo.

2.1 UML



In questo diagramma, viene mostrato un solo prodotto concreto, ma ce ne potrebbe essere più di uno. Analogamente, potrebbe esserci più di un creatore concreto: spesso, ce n'è uno per ogni prodotto concreto.

2.2 Esempi di codice

```
public abstract class Product { }
public class ConcreteProductA extends Product { }
public class ConcreteProductB extends Product { }

public abstract class Creator {
    public abstract Product factoryMethod();
}
public class ConcreteCreatorA extends Creator {
    public Product factoryMethod() {
        return new ConcreteProductA();
    }
}
public class ConcreteCreatorB extends Creator {
    public Product factoryMethod() {
        return new ConcreteProductB();
    }
}

public class MainApp {
    public static void main(String[] args) {
        Creator[] creators = {
```

```
            new ConcreteCreatorA(),
            new ConcreteCreatorB()
        };

        for (Creator creator : creators) {
            Product product = creator.factoryMethod();
            System.out.println(
                "Created " + product.getClass().getName()
            );
        }
    }
}
```

L'output di questo programma è:

```
Created ConcreteProductA
Created ConcreteProductB
```

Come altro esempio, si suppone di avere un tipo `Poly`, che rappresenta polinomi. Esso ha due implementazioni:

- `DensePoly` rappresenta un polinomio qualsiasi ($a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$);
- `SparsePoly` rappresenta, in modo più efficiente, un polinomio del tipo $ax^n + b$.

In questo caso, l'uso di un metodo factory permette di scegliere l'implementazione più adeguata in base ai valori dei coefficienti:

```
public static Poly createPoly(int[] coeffs) {
    int degree = -1;
    int numCoeffs = 0;
    for (int n = 0; n < coeffs.length; n++) {
        if (coeffs[n] != 0) {
            numCoeffs++;
            degree = n;
        }
    }

    if ((numCoeffs == 2 && coeffs[0] != 0) || numCoeffs == 1) {
        return new SparsePoly(degree, coeffs[degree], coeffs[0]);
    }
    return new DensePoly(degree, coeffs);
}
```

3 Singleton

A volte, si vuole che una classe possa essere istanziata da un solo oggetto. Questo tipo di classe è detta **singleton**:

- il costruttore è privato (o almeno protetto), per impedire la creazione di più istanze;
- un metodo factory crea la sola istanza della classe quando viene chiamato per la prima volta, e restituisce la stessa istanza alle chiamate successive.

Singleton
− instance: Singleton
− Singleton() + Instance(): Singleton

3.1 Esempio di codice

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // ...
    }

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

    // altri metodi...
}

public class MainApp {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        if (s1 == s2) {
            System.out.println("Objects are the same instance");
        }
```

```
    }
}
```

Questo programma stampa:

```
Objects are the same instance
```

4 Flyweight

Quando molti oggetti identici e *immutabili* vengono utilizzati contemporaneamente, è utile costruire e condividere un solo oggetto per ogni classe di equivalenza. Questi oggetti condivisi vengono chiamati **flyweight** (pesi mosca), perché spesso sono molto piccoli. Un esempio comune è il tipo `String`, che in molti linguaggi (compreso Java) è gestito appunto come un flyweight.

Per implementare questo pattern, si usa una factory che tiene gli oggetti già creati in una tabella. Quando viene richiesto un oggetto, la factory controlla se questo esiste già nella tabella, e in tal caso restituisce un riferimento a esso, altrimenti costruisce un nuovo oggetto (chiamando un costruttore privato) e lo aggiunge alla tabella.

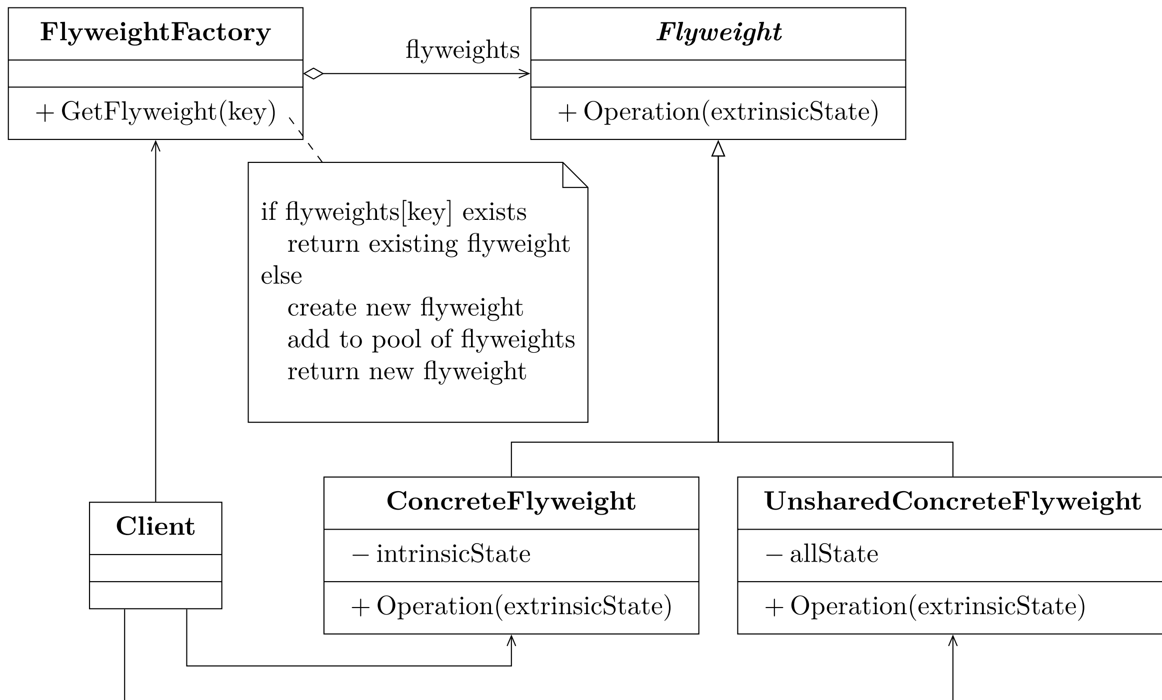
Gli oggetti flyweight devono essere immutabili per evitare problemi di aliasing: se uno di essi venisse modificato, tale modifica sarebbe visibile da tutti i client che lo condividono (cosa che invece non accadrebbe senza l'uso dei flyweight, perché allora i client avrebbero copie separate dell'oggetto).

Usando questo pattern, se c'è un alto grado di condivisione degli oggetti:

- si risparmia memoria;
- non si perde tempo a inizializzare oggetti duplicati;
- si può usare `==` per il confronto, al posto di `equals`.

Il flyweight pattern può essere visto come un'evoluzione del singleton: invece di avere una singola istanza di una classe, si ha una singola copia per ogni oggetto diverso.

4.1 UML



- I flyweight possono essere di tipi diversi.[1]
- Il client fornisce alla factory una qualche chiave di accesso, che identifica l'oggetto che si vuole ottenere (creandolo, se non esiste già).
- Le istanze di ConcreteFlyweight contengono solo la parte di stato immutabile (intrinsicState). Se lo stato ha anche una parte mutabile, questa deve essere gestita separatamente dall'oggetto (altrimenti si avrebbero problemi di aliasing), e passata come argomento (extrinsicState) ai metodi.

4.2 Esempio di codice

```
import java.util.*;

public abstract class Flyweight {
    public abstract void operation(int extrinsicState);
}
```

[1] È anche possibile che solo alcuni tipi di oggetti flyweight siano condivisi. Ad esempio, in questo diagramma, la classe UnsharedConcreteFlyweight rappresenta oggetti che non possono essere condivisi, ma che implementano comunque l'interfaccia Flyweight perché supportano le stesse operazioni degli altri tipi di flyweight.

```java
public class ConcreteFlyweight extends Flyweight {
    private final int intrinsicState;

    ConcreteFlyweight(int intrinsicState) {
        this.intrinsicState = intrinsicState;
    }

    public void operation(int extrinsicState) {
        System.out.println(
            "ConcreteFlyweight: "
            + intrinsicState + ", " + extrinsicState
        );
    }
}

public class FlyweightFactory {
    private Map<String, Flyweight> flyweights = new HashMap<>();

    public FlyweightFactory() {
        flyweights.put("X", new ConcreteFlyweight(1));
        flyweights.put("Y", new ConcreteFlyweight(2));
        flyweights.put("Z", new ConcreteFlyweight(3));
    }

    public Flyweight getFlyweight(String key) {
        return flyweights.get(key);
    }
}

public class MainApp {
    public static void main(String[] args) {
        int extrinsicState = 22;
        FlyweightFactory factory = new FlyweightFactory();

        Flyweight fx = factory.getFlyweight("X");
        fx.operation(--extrinsicState);
        Flyweight fy = factory.getFlyweight("Y");
        fy.operation(--extrinsicState);
        Flyweight fz = factory.getFlyweight("Z");
        fz.operation(--extrinsicState);

        Flyweight fx2 = factory.getFlyweight("X");
        fx2.operation(--extrinsicState);
        if (fx == fx2) {
```

```
            System.out.println("fx and fx2 are the same instance");
        }
    }
}
```

L'output del programma è:

```
ConcreteFlyweight: 1, 21
ConcreteFlyweight: 2, 20
ConcreteFlyweight: 3, 19
ConcreteFlyweight: 1, 18
fx and fx2 are the same instance
```

5 State

A volte, la variabilità, sia di stato che di comportamento, degli oggetti mutabili può essere molto elevata. Il codice di questi oggetti tende allora a diventare assai complicato, e pieno di condizioni.

Lo **state** pattern risolve questo problema mediante oggetti che cambiano configurazione a seconda dello stato in cui si trovano. A tale scopo, esso introduce un ulteriore strato[2] (classe mediatrice) tra il tipo implementato e l'implementazione:

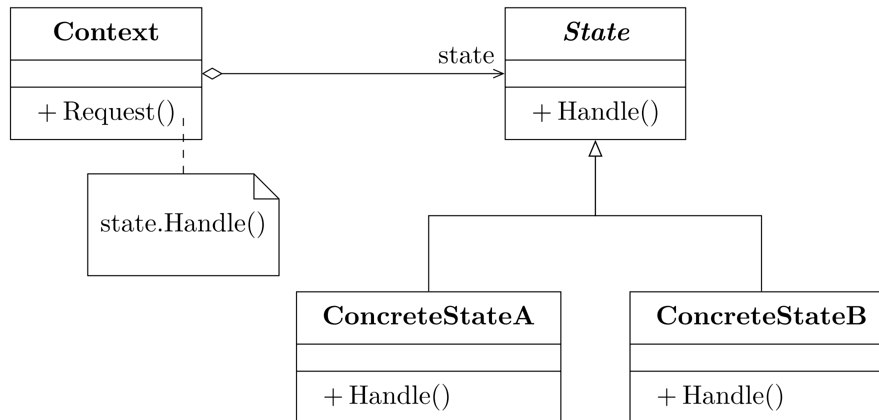
- per un unico tipo, vengono definite più classi che lo implementano, ciascuna corrispondente a un diverso stato in cui possono trovarsi gli esemplari del tipo;
- nel corso della vita dell'oggetto, l'implementazione usata può cambiare, senza che l'utente se ne accorga.

Per implementarlo, si crea un'interfaccia o una classe astratta, che rappresenta le parti dell'oggetto che possono essere sostituite nel corso della sua vita:

- ciascuna delle possibili rappresentazioni (stati) diventa un'implementazione dell'interfaccia o una sottoclasse della classe astratta;
- la classe principale contiene il codice per scegliere la rappresentazione più adatta e per delegare a essa l'implementazione.

[2] L'introduzione di un nuovo strato è un "trucco" molto comune, in quanto permette di risolvere numerosi problemi: "Every problem in computer science can be solved by adding another level of indirection".

5.1 UML



Il client lavora sulla classe Context (una classe mediatrice), che incapsula un'istanza di State (l'implementazione vera e propria), permettendo di sostituirla quando cambia lo stato dell'oggetto.

5.2 Esempio di codice

```
public abstract class State {
    public abstract void handle(Context context);
    public abstract String info();
}
public class ConcreteStateA extends State {
    public void handle(Context context) {
        context.setState(new ConcreteStateB());
    }
    public String info() { return "ConcreteStateA"; }
}
public class ConcreteStateB extends State {
    public void handle(Context context) {
        context.setState(new ConcreteStateA());
    }
    public String info() { return "ConcreteStateB"; }
}

public class Context{
    private State state;

    public Context(State state) {
        this.state = state;
```

```java
    }

    public State getState() {
        return state;
    }

    public void setState(State value) {
        state = value;
        System.out.println(state.info());
    }

    public void request() {
        state.handle(this);
    }
}

public class MainApp {
    public static void main(String[] args) {
        Context c = new Context(new ConcreteStateA());

        // Stato: ConcreteStateA
        c.request(); // Invoca ConcreteStateA.handle
        // Stato: ConcreteStateB
        c.request(); // Invoca ConcreteStateB.handle
        // Stato: ConcreteStateA
        c.request(); // Invoca ConcreteStateA.handle
        // Stato: ConcreteStateB
        c.request(); // Invoca ConcreteStateB.handle
        // Stato: ConcreteStateA
    }
}
```

In questo esempio, per semplicità, l'implementazione del metodo `handle` di ciascuno stato non fa altro che passare all'altro stato. Il programma produce quindi in output:

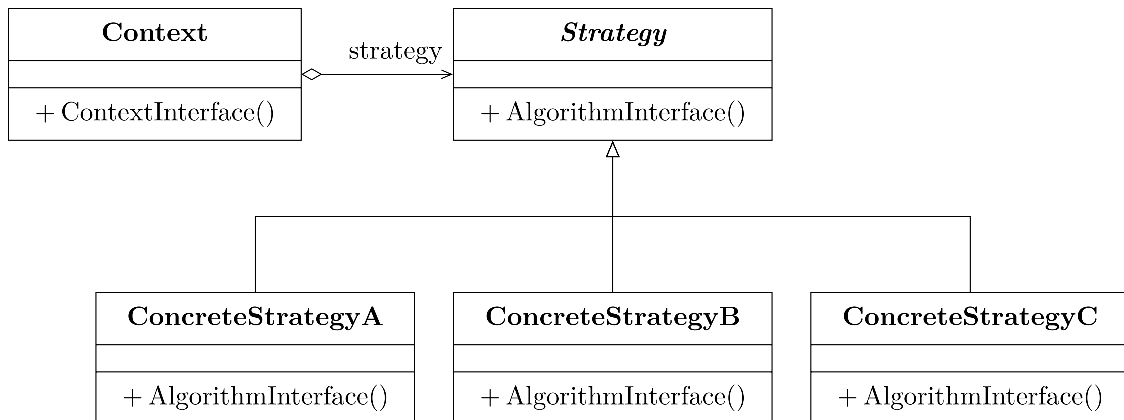
```
ConcreteStateB
ConcreteStateA
ConcreteStateB
ConcreteStateA
```

In un'applicazione reale, invece, le operazioni implementate sarebbero più complesse e utili (e, in generale, non è obbligatorio che si verifichi ogni volta una transizione di stato).

6 Strategy

Spesso, è utile passare come argomento di un metodo un algoritmo che ne influenza il comportamento. Tale algoritmo è solitamente implementato, a sua volta, da un metodo, ma alcuni linguaggi (come, ad esempio, le versioni di Java precedenti alla 8) non permettono di trattare i metodi come oggetti, e quindi di passarli come argomenti ad altri metodi. Allora, bisogna per forza definire un'interfaccia/classe molto piccola, il cui unico scopo è contenere un'implementazione di una determinata operazione. Questa soluzione prende il nome di **strategy** pattern.

6.1 UML



- La classe astratta Strategy (che potrebbe, equivalentemente, essere un'interfaccia) definisce l'operazione richiesta.
- Ogni implementazione concreta dell'operazione viene realizzata come sottoclasse di Strategy.
- La classe Context è parametrizzata da un oggetto Strategy, permettendo così di selezionare quale implementazione dell'operazione verrà usata.

6.2 Esempio: `Comparator`

Un esempio di strategy è l'interfaccia `java.util.Comparator`, che definisce un algoritmo di confronto tra oggetti:

```
interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Essa viene usata, ad esempio, come parametro di alcuni metodi di ordinamento, per specificare il modo in cui confrontare gli elementi da ordinare. Uno di questi metodi è `java.util.Arrays.sort`:

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

Nella sua implementazione, questo metodo utilizza l'algoritmo di confronto passato come argomento mediante chiamate del tipo `c.compare(a[i], a[j])`.

Un esempio d'uso, per l'ordinamento alfabetico (case-insensitive) di un array di stringhe, è:

```
import java.util.*;

public class AlphabeticComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
        return s1.toLowerCase().compareTo(s2.toLowerCase());
    }
}

public class MainApp {
    public static void main(String[] args) {
        // ...
        String[] s = new String[30];
        // ...
        Arrays.sort(s, new AlphabeticComparator());
        // ...
    }
}
```

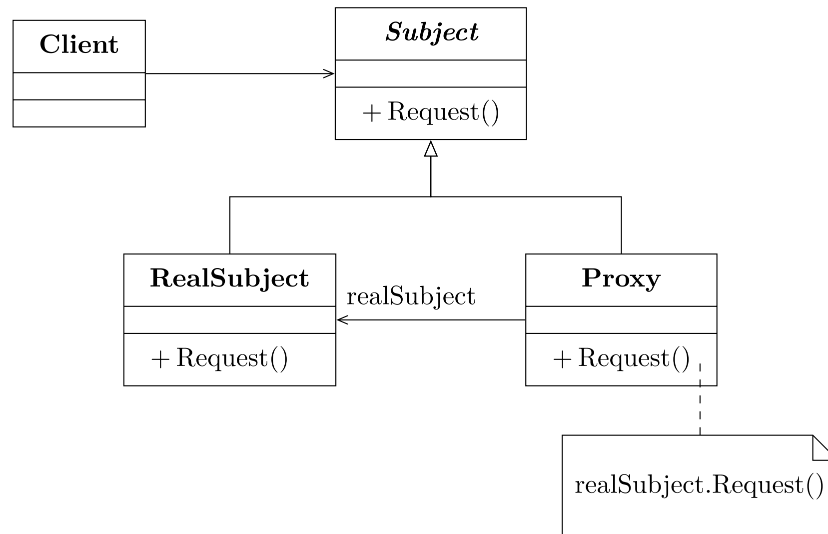
7 Proxy

Un **proxy** è un oggetto che si interpone tra un altro oggetto e i suoi client, presentando *esattamente la stessa interfaccia* di tale oggetto, ma permettendo di controllare meglio gli accessi a esso.

Questo pattern può essere utile, ad esempio, per:

- postporre l'istanziazione di un oggetto "pesante" finché non è effettivamente necessaria;
- aggiungere controllo dei permessi, caching, ecc. ai servizi offerti da un oggetto;
- rendere trasparente la comunicazione con un oggetto remoto.

7.1 UML



- Il Client interagisce con un Subject, che però non è un “vero” oggetto RealSubject, bensì un’istanza di Proxy, che inoltra poi le richieste di operazioni a RealSubject.
- RealSubject e Proxy sono entrambe sottoclassi di Subject, dato che implementano la stessa interfaccia (altrimenti, non si tratterebbe del proxy pattern).

7.2 Esempio di codice

```
public abstract class Subject {
    public abstract void request();
}

public class RealSubject extends Subject {
    public void request() {
        System.out.println("Called RealSubject.request()");
    }
}

public class Proxy extends Subject {
    private RealSubject realSubject;

    public void request() {
        if (realSubject == null) {
            realSubject = new RealSubject();
        }
        realSubject.request();
```

```
    }
}

public class MainApp {
    public static void main(String[] args) {
        Proxy proxy = new Proxy();
        proxy.request();
    }
}
```

In questo esempio, il proxy è usato solo per posticipare l'istanziazione di RealSubject: l'istanza non viene creata finché non è necessario eseguire il metodo `RealSubject.request`. L'output prodotto dal programma è:

```
Called RealSubject.request()
```

8 Adapter

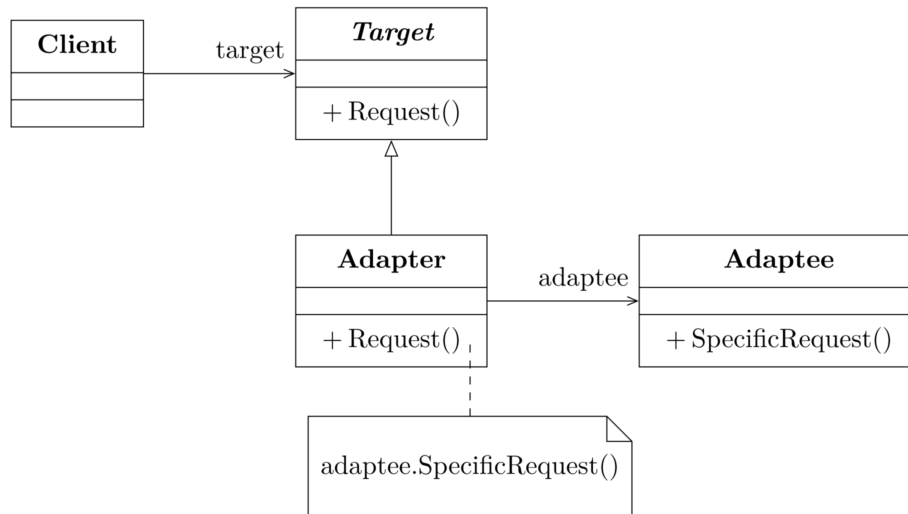
Spesso, un'interfaccia offerta da un oggetto che offre un determinato servizio deve essere adattata per i propri scopi. A tale scopo, si aggiunge uno strato intermedio di traduzione:

- l'oggetto traduttore si chiama **adapter**;
- l'oggetto originale, che offre il servizio, prende il nome di **adaptee**.

Questo pattern è utile, ad esempio, quando il codice deve essere compatibile con più librerie che espongono interfacce diverse per lo stesso servizio: usando gli adapter, si può effettuare la traduzione dall'interfaccia specifica di ciascuna libreria a un'interfaccia uniforme, che è quella con cui interagiscono poi i client. Così, se si cambia la libreria usata, è sufficiente sostituire l'adapter, mentre il resto del codice può rimanere invariato.

La differenza fondamentale rispetto al proxy pattern è che l'interfaccia esposta dall'adapter *non è uguale* a quella dell'oggetto sottostante.

8.1 UML



- Gli adapter sono sottoclassi della classe astratta Target, che definisce l'interfaccia uniforme implementata da essi. In questo diagramma ne è mostrato uno solo, ma ce ne potrebbero essere tanti (uno per ogni adaptee).
- Gli adaptee *non* implementano la stessa interfaccia, quindi non sono sottoclassi di Target.

8.2 Esempio di codice

```java
public abstract class Target {
    public abstract void request();
}

public class Adapter extends Target {
    private Adaptee adaptee = new Adaptee();

    public void request() {
        adaptee.specificRequest();
    }
}

public class Adaptee {
    public void specificRequest() {
        System.out.println("Called specificRequest()");
    }
}
```

```java
public class MainApp {
    public static void main(String[] args) {
        Target target = new Adapter();
        target.request();
    }
}
```

In questo esempio, l'istanza di `Adaptee` è creata direttamente dall'adapter, ma, in alternativa, potrebbe essere passata dall'esterno (ad esempio, come parametro del costruttore).

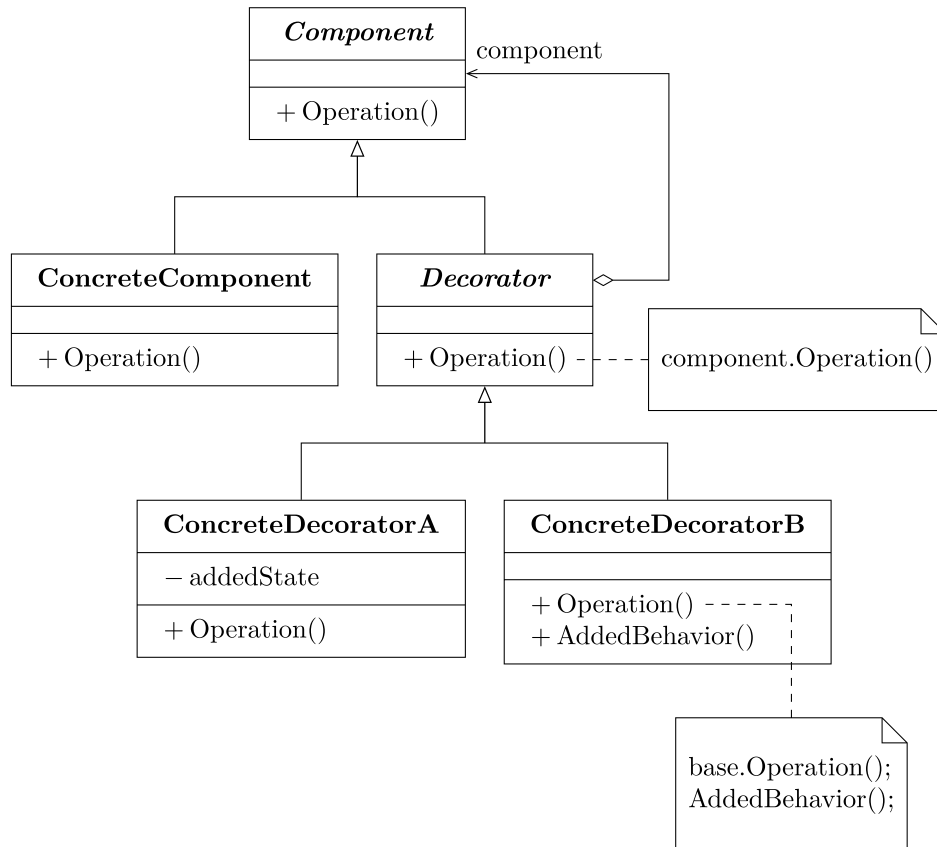
Il programma stampa:

```
Called specificRequest()
```

9 Decorator

Un **decorator** "decora" un oggetto interno, implementando la stessa interfaccia, ma fornendo anche delle funzionalità aggiuntive.

9.1 UML



- La classe astratta Component definisce l'interfaccia di base. Un oggetto che la implementa può essere o un componente concreto, oppure un decoratore, che "contiene" al suo interno un altro componente, al quale aggiunge delle funzionalità.
- Il componente contenuto da un decoratore può essere a sua volta un'istanza di Decorator: ciò permette di applicare successivamente più decoratori a uno stesso oggetto, aggiungendo sempre più funzionalità.
- Ciascun decoratore concreto, sottoclasse di Decorator, aggiunge funzionalità diverse all'oggetto decorato.

9.2 Esempio di codice

```
public abstract class Component {
    public abstract void operation();
}

public class ConcreteComponent extends Component {
```

```java
    public void operation() {
        System.out.println("ConcreteComponent.operation()");
    }
}

public class abstract Decorator extends Component {
    protected Component component;

    public void setComponent(Component component) {
        this.component = component;
    }

    public void operation() {
        if (component != null) {
            component.operation();
        }
    }
}

public class ConcreteDecoratorA extends Decorator {
    public void operation() {
        super.operation();
        System.out.println("ConcreteDecoratorA.operation()");
    }
}
public class ConcreteDecoratorB extends Decorator {
    public void operation() {
        super.operation();
        System.out.println("ConcreteDecoratorB.operation()");
    }
}

public class MainApp {
    public static void main(String[] args) {
        ConcreteComponent c = new ConcreteComponent();
        ConcreteDecoratorA d1 = new ConcreteDecoratorA();
        ConcreteDecoratorB d2 = new ConcreteDecoratorB();

        d1.setComponent(c);
        d2.setComponent(d1);
        d2.operation();
    }
}
```

- Mediante le chiamate `setComponent`, il decoratore `d1` viene applicato al componente concreto `c`, poi il decoratore `d2` si applica all'oggetto già decorato da `d1`.
- Il metodo `Decorator.operation` esegue solo l'operazione del componente decorato. I decoratori concreti fanno poi l'override di questo metodo per aggiungere delle funzionalità, che in questo caso sono delle semplici stampe, effettuate dopo l'operazione dell'oggetto interno.
- I decoratori potrebbero anche aggiungere codice da eseguire prima dell'operazione dell'oggetto interno: non è obbligatorio che la chiamata `super.operation()` sia la prima istruzione.

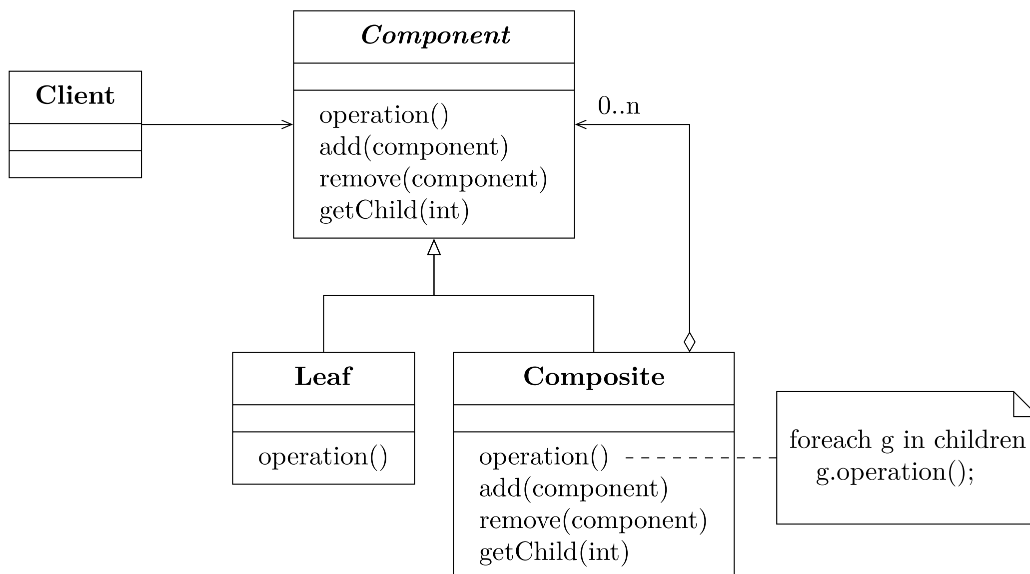
L'output stampato dal programma è:

```
ConcreteComponent.operation()
ConcreteDecoratorA.operation()
ConcreteDecoratorB.operation()
```

10 Composite

Il **composite** pattern permette di comporre oggetti in strutture ad albero, che rappresentano gerarchie intero-parte, e consente ai client di trattare oggetti singoli e compositi in modo uniforme.

10.1 UML



- Il client vuole eseguire un'operazione su un componente.
- Il componente può essere una foglia (Leaf), oppure un gruppo di altri componenti (Composite).
- Quando si richiede a un Composite di eseguire un'operazione, esso "inoltra" la richiesta a tutti i componenti che raggruppa. Se ci sono più livelli di Composite, ciò avviene in modo ricorsivo, fino a giungere alle foglie.
- Potrebbero esserci più tipi diversi di oggetti foglia e/o di oggetti compositi.

10.2 Esempio di codice

```
import java.util.*;

public abstract class Component {
    protected String name;

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

    public abstract void add(Component c);
    public abstract void remove(Component c);
    public abstract void display(int depth);
}

public class Composite extends Component {
    private ArrayList<Component> children = new ArrayList<>();

    public Composite(String name) {
        super(name);
    }

    public void add(Component c) {
        children.add(c);
    }

    public void remove(Component c) {
        children.remove(c);
    }

    public void display(int depth) {
        for (int i = 0; i < depth; i++) {
```

```java
            System.out.print('-');
        }
        System.out.println(name);
        for (Component child : children) {
            child.display(depth + 2);
        }
    }
}

public class Leaf extends Component {
    public Leaf(String name) {
        super(name);
    }

    public void add(Component c) {
        throw new UnsupportedOperationException("Cannot add to a leaf");
    }

    public void remove(Component c) {
        throw new UnsupportedOperationException(
            "Cannot remove from a leaf"
        );
    }

    public void display(int depth) {
        for (int i = 0; i < depth; i++) {
            System.out.print('-');
        }
        System.out.println(name);
    }
}

public class MainApp {
    public static void main(String[] args) {
        Composite root = new Composite("root");

        root.add(new Leaf("Leaf A"));
        root.add(new Leaf("Leaf B"));

        Composite comp = new Composite("Composite X");
        comp.add(new Leaf("Leaf XA"));
        comp.add(new Leaf("Leaf XB"));
        root.add(comp);
```

```
        root.add(new Leaf("Leaf C"));

        Leaf leaf = new Leaf("Leaf D");
        root.add(leaf);
        root.remove(leaf);

        root.display(1);
    }
}
```

Questo programma stampa:

```
-root
---Leaf A
---Leaf B
---Composite X
-----Leaf XA
-----Leaf XB
---Leaf C
```

10.3 Considerazioni

Il composite pattern ha il vantaggio di rendere più semplice l'aggiunta di nuove tipologie di componenti: è sufficiente aggiungere nuove sottoclassi foglia o composite, senza bisogno di modificare il codice dei clienti, che infatti trattano tutti i componenti in modo uniforme.

In compenso, l'uso del composite può rendere il progetto troppo generico, perché è difficile imporre restrizioni ai tipi di componenti che fanno parte di una struttura composita: se essa deve poter contenere soltanto determinate tipologie di componenti, bisogna inserire nel codice degli appositi controlli, che verranno effettuati a run-time.

11 Abstract Factory

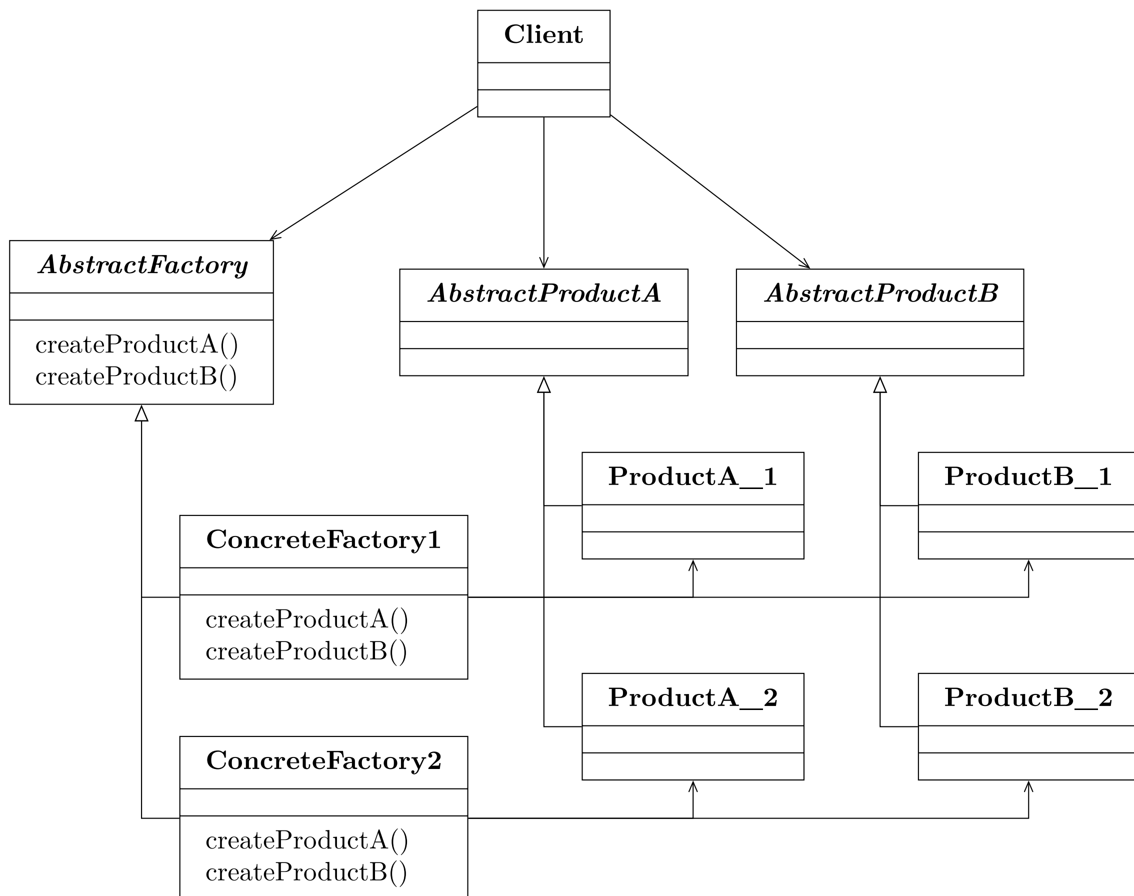
L'**abstract factory** pattern si applica quando si hanno diverse famiglie di oggetti correlati, bisogna creare oggetti appartenenti a una di queste famiglie, e si vuole scegliere la famiglia da utilizzare in un solo punto del codice, senza doverla specificare a ogni operazione di creazione.

La soluzione consiste nel definire diverse classi factory, una per ogni famiglia, che implementano un'interfaccia uniforme. Il codice client crea gli oggetti esclusivamente attraverso una di queste factory: per cambiare la famiglia di oggetti usata, è allora sufficiente sostituire la factory, senza bisogno di altre modifiche al codice.

Un esempio tipico di applicazione di questo pattern sono le interfacce grafiche: ogni sistema operativo definisce una sua famiglia di componenti grafici, e l'applicazione deve creare componenti di una famiglia diversa a seconda del sistema operativo su cui viene eseguita.

La differenza fondamentale rispetto al factory method pattern è che esso riguarda la creazione di un singolo tipo di oggetto, mentre l'abstract factory si occupa di creare una famiglia di oggetti.

11.1 UML



- Le classi AbstractProduct dichiarano le interfacce per ciascun tipo di prodotto.
- Ogni classe Product fornisce un'implementazione concreta per uno specifico tipo di prodotto all'interno di una determinata famiglia:

	Tipo A	Tipo B
Famiglia 1	ProductA_1	ProductB_1
Famiglia 2	ProductA_2	ProductB_2

- AbstractFactory dichiara un'interfaccia uniforme per la creazione dei prodotti di una famiglia.
- Per ogni famiglia è definita una ConcreteFactory, che implementa le operazioni di creazione dei prodotti di ciascun tipo appartenenti a tale famiglia.
- Il Client:
  - crea i prodotti chiamando una ConcreteFactory, attraverso l'interfaccia AbstractFactory;
  - usa i prodotti attraverso le interfacce AbstractProduct.

Osservazione: Aggiungere una nuova famiglia è abbastanza facile (basta definire i suoi prodotti e la ConcreteFactory che li crea), ma, invece, aggiungere nuovi tipi di prodotti è difficile:

- può richiedere cambiamenti all'interfaccia dell'AbstractFactory e alle sue sottoclassi;
- richiede l'aggiunta di nuovi AbstractProduct e Product.

11.2 Esempio di codice

```
public abstract class AbstractProductA {}
public class ProductA1 extends AbstractProductA {}
public class ProductA2 extends AbstractProductA {}

public abstract class AbstractProductB {}
public class ProductB1 extends AbstractProductB {}
public class ProductB2 extends AbstractProductB {}

public abstract class AbstractFactory {
    public abstract AbstractProductA createProductA();
    public abstract AbstractProductB createProductB();
}
public class ConcreteFactory1 extends AbstractFactory {
    public AbstractProductA createProductA() {
        return new ProductA1();
    }
    public AbstractProductB createProductB() {
        return new ProductB1();
    }
}
public class ConcreteFactory2 extends AbstractFactory {
    public AbstractProductA createProductA() {
```

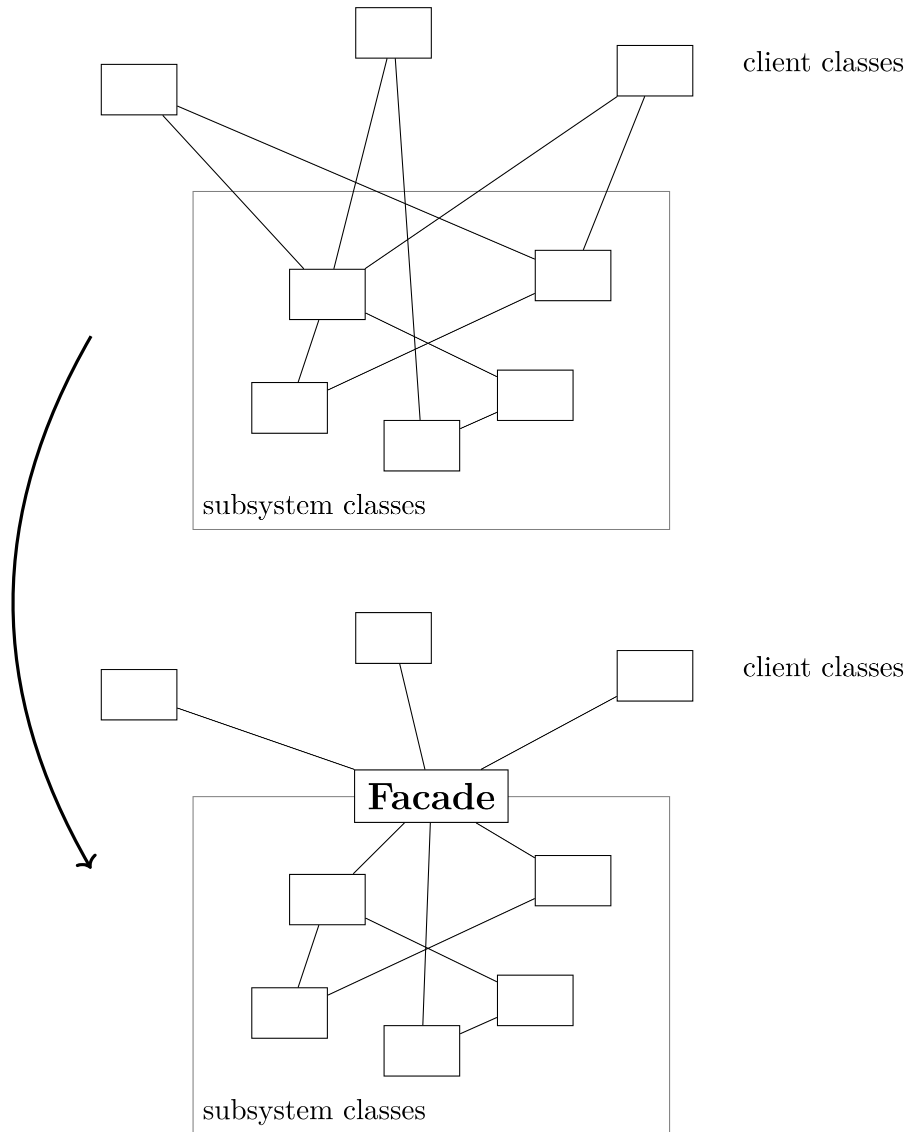
```
        return new ProductA2();
    }
    public AbstractProductB createProductB() {
        return new ProductB2();
    }
}

public class Client {
    private AbstractProductA abstractProductA;
    private AbstractProductB abstractProductB;

    public Client(AbstractFactory factory) {
        abstractProductA = factory.createProductA();
        abstractProductB = factory.createProductB();
    }

    public void run() {
        abstractProductB.interact(abstractProductA);
    }
}

public class MainApp {
    public static void main(String[] args) {
        AbstractFactory factory1 = new ConcreteFactory1();
        Client client1 = new Client(factory1);
        client1.run();

        AbstractFactory factory2 = new ConcreteFactory2();
        Client client2 = new Client(factory2);
        client2.run();
    }
}
```

12 Façade

Il **façade** pattern fornisce un'interfaccia unificata e a più alto livello verso un sottosistema complesso (costituito da un insieme di classi/oggetti), rendendolo più semplice da utilizzare. Inoltre, questo pattern:

- nasconde ai client la struttura interna del sottosistema, riducendo quindi l'accoppiamento tra client e sottosistema;

- rende topologicamente più semplice il sistema, perché ogni client che dovrebbe interagire con più componenti del sottosistema comunica invece solo con la classe façade;
- evita la ripetizione del codice che servirebbe ai client per ottenere un servizio mettendo insieme i servizi esportati da più classi del sottosistema.



12.1 Esempio di codice

```
public class SubSystemOne {
    public void methodOne() {
```

```java
        System.out.println(" SubSystemOne Method");
    }
}
public class SubSystemTwo {
    public void methodTwo() {
        System.out.println(" SubSystemTwo Method");
    }
}
public class SubSystemThree {
    public void methodThree() {
        System.out.println(" SubSystemThree Method");
    }
}
public class SubSystemFour {
    public void methodFour() {
        System.out.println(" SubSystemFour Method");
    }
}

public class Facade {
    private SubSystemOne one = new SubSystemOne();
    private SubSystemTwo two = new SubSystemTwo();
    private SubSystemThree three = new SubSystemThree();
    private SubSystemFour four = new SubSystemFour();

    public void methodA() {
        System.out.println("methodA() ----");
        one.methodOne();
        two.methodTwo();
        four.methodFour();
    }

    public void methodB() {
        System.out.println("methodB() ----");
        two.methodTwo();
        three.methodThree();
    }
}

public class MainApp {
    public static void main(String[] args) {
        Facade facade = new Facade();
        facade.methodA();
        facade.methodB();
```

```
    }
}
```

L'output di questo programma è:

```
methodA() ----
 SubSystemOne Method
 SubSystemTwo Method
 SubSystemFour Method
methodB() ----
 SubSystemTwo Method
 SubSystemThree Method
```

13 Observer

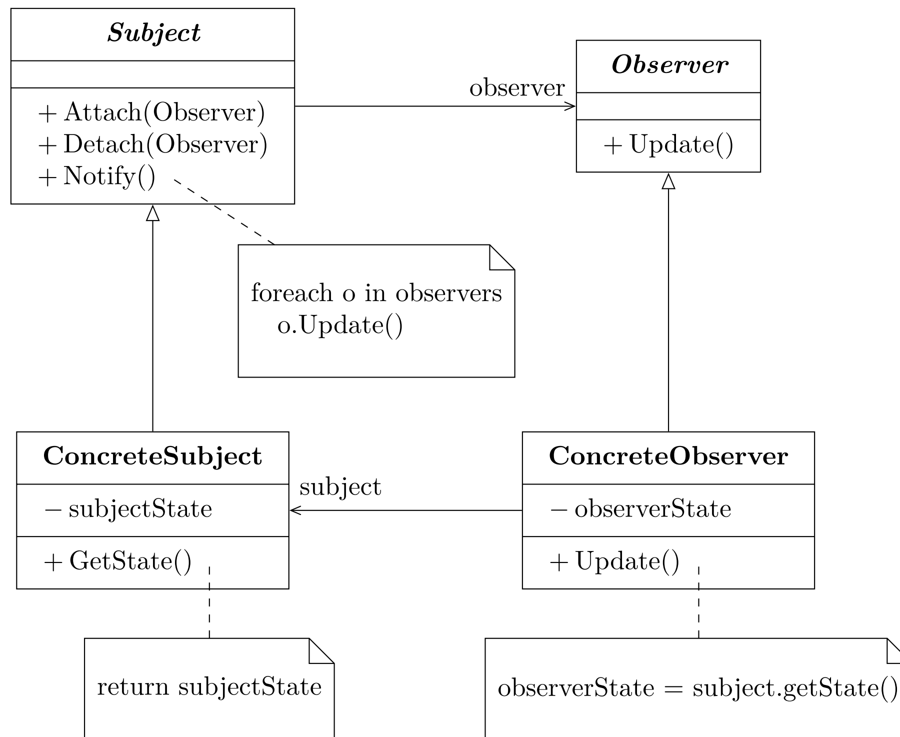
L'**observer** è il pattern dei sistemi ad eventi: un observer è un oggetto che è interessato a osservare gli eventi / cambiamenti di stato di un soggetto.

Per rendere il soggetto indipendente dal numero e dal tipo degli osservatori, e consentire l'aggiunta di nuovi osservatori durante l'esecuzione dell'applicazione, si inseriscono nel soggetto delle operazioni che consentono all'osservatore di dichiarare il proprio interesse per gli eventi / cambiamenti di stato di tale soggetto.

Questo pattern è utile quando:

- i cambiamenti di un soggetto devono essere inviati a tanti oggetti;
- i soggetti, indipendenti dall'applicazione (ad esempio, perché fanno parte di una libreria riutilizzabile), devono notificare cambiamenti del loro stato a oggetti implementati specificamente per l'applicazione, dei quali non possono conoscere niente (altrimenti non sarebbero indipendenti dall'applicazione).

13.1 UML



14 Tipologie di pattern

Esistono diverse tipologie di pattern, che si differenziano principalmente per la scala e il livello di astrazione:

architectural pattern: utili per strutturare un sistema in sottosistemi;

design pattern: operano a livello di un sottosistema, evidenziando le caratteristiche delle classi coinvolte e delle associazioni tra le classi;

idiom: descrivono, a basso livello, come implementare particolari aspetti di singoli componenti utilizzando le funzionalità di uno specifico linguaggio di programmazione.

15 MVC

MVC è un pattern architetturale che struttura l'applicazione mediante tre tipi principali di componenti:

Model: rappresenta i dati veri e propri, funzionali all'applicazione, in modo indipendente da come vengono visualizzati sullo schermo e dalle modalità di input da parte dell'utente.

View: è l'interfaccia utente (o con altri sistemi), che visualizza sullo schermo i dati rappresentati dal model.

Controller: gestisce la comunicazione tra view e model, ricevendo gli input dall'utente (tipicamente sotto forma di eventi) e traducendoli in richieste di servizio per il modello o la vista a cui è associato.

In questo modo, si realizza un sistema a strati, ciascuno dei quali comunica solo con quelli adiacenti. Grazie al disaccoppiamento di viste e modelli, la presentazione dei dati all'utente può essere modificata indipendentemente dalla logica applicativa.

È possibile (e comune) avere più rappresentazioni grafiche per gli stessi dati, mediante view e controller diversi che fanno riferimento allo stesso model.

15.1 Design pattern in un'applicazione MVC

In un'applicazione basata sul pattern architetturale MVC, si ritrovano solitamente numerosi design pattern. Ad esempio:

- il disaccoppiamento di viste e modelli si ottiene mediante l'*observer* pattern;
- il *composite* pattern permette di realizzare viste annidate;
- il controller associato a una vista incapsula un algoritmo di risposta all'utente, e può essere sostituito anche a run-time, quindi è un esempio di *strategy* pattern.