Ripasso Pattern Ingegneria del software

Anonimo

25 settembre 2022

Indice

1 Strutturali

Obbiettivo. Risolvere problemi a livello della struttura delle classi, semplificando la gestione di esse.

- ☛ **Proxy.**
  - **Descrizione:** simile a quello che accade nelle reti per un vero proxy. L'obbiettivo è di definire la struttura di una classe. La definizione avviene facendo rappresentare ad essa le funzionalità di un'altra classe. Quindi, vengono creati oggetti basati su un oggetto originale, ma che consentono di interfacciarsi con il mondo esterno.
  - **Caratteristiche:** viene creata un'interfaccia che definisce i metodi che devono essere implementati dalle classi specifiche.
    Il cliente, per esempio il `main`, esegue un "colloquio" **solamente** con le classi `Proxy`. Quest'ultima è una classe che al suo interno ha dei metodi che utilizzano la classe originale, la quale definisce l'effettivo comportamento.
    In parole molto povere, è come se una persona estranea richiedesse ad un ragazzo estraneo (figlio) di eseguire determinati compiti. Quest'ultimi verranno eseguiti da un soggetto (padre del figlio) che non è il figlio. Dunque, la persona estranea non sa realmente che i comportamenti verranno effettuati dal padre.
    Credo che sia un pattern che protegga determinate parti di codice.

☛ **Facade.**

- **Descrizione:** la complessità del sistema viene completamente nascosta al cliente. A quest'ultimo viene fornita un'interfaccia con cui comunicare con il sistema.
  Il motivo per cui questo pattern è strutturale è banale: l'aggiunta di un'interfaccia per nascondere la complessità del sistema.

- **Caratteristiche:** viene creata una classe che si interfaccerà con il cliente. Lo scopo di questa classe è **semplificare** i metodi richiesti dal cliente e **delegare** le chiamate ai metodi a classi già esistenti nel sistema.
  Per esempio, viene creata una classe `ShapeMaker` che ha l'obbiettivo di semplificare i metodi esistenti in altre classi. Questo avviene richiamando e "manipolando" classi esistenti:

```
public class ShapeMaker {
        private Shape circle;
        private Shape rectangle;
        private Shape square;
        public ShapeMaker() {
                circle = new Circle();
                rectangle = new Rectangle();
                square = new Square();
        }
        public void drawCircle(){
                circle.draw();
        }
        public void drawRectangle(){
                rectangle.draw();
        }
        public void drawSquare(){
                square.draw();
        }
}
```

  Quindi, il cliente, per esempio il `main`, eseguirà delle operazioni molto semplici:

```
public class FacadePatternDemo {
        public static void main(String[] args){
                ShapeMaker shapeMaker = new ShapeMaker();
                shapeMaker.drawCircle();
                shapeMaker.drawRectangle();
                shapeMaker.drawSquare();
}}
```

☛ **Decorator.**

- **Descrizione:** l'utente aggiunge nuove funzionalità ad un oggetto senza alterarne la sua struttura. Esegue un *wrapper* di una classe esistente (motivo per cui è di tipo strutturale).
- **Caratteristiche:** come avviene per il Facade pattern, anche qui si crea solamente una classe con il compito di **semplificare** i metodi richiesti dal cliente e **delegare** le chiamate ai metodi a classi già esistenti nel sistema.
  Per esempio, si creano due oggetti che sono una forma `Shape`, ma hanno delle proprietà particolari, come la colorazione rossa del bordo `RedShapeDecorator`. Tuttavia, l'oggetto `Shape` è rimasto invariato, sono state solo aggiunte delle caratteristiche. Ovviamente, `RedShapeDecorator` è il figlio della classe `Shape`:

```
public class DecoratorPatternDemo {
        public static void main(String[] args) {

                Shape circle = new Circle();

                Shape redCircle =
                        new RedShapeDecorator(new Circle());

                Shape redRectangle =
                        new RedShapeDecorator(new Rectangle());
                System.out.println("Circle with normal border");
                circle.draw();

                System.out.println("\nCircle of red border");
                redCircle.draw();

                System.out.println("\nRectangle of red border");
                redRectangle.draw();
        }
}
```

2 Creazionali

Obbiettivo. Risolvere problemi a livello di creazione di oggetti con l'intento di semplificare "l'istanziazione" di essi.

- ☛ **[Migliore] Singleton.**
  - **Descrizione:** uno dei migliori pattern creazionali. Consente di creare un *unico* oggetto al quale ci si può solo accedere indirettamente, cioè tramite un metodo.
  - **Caratteristiche:** la classe singleton ha un costruttore privato e un tentativo di creazione creerebbe un'eccezione. Inoltre, l'oggetto *unico* deve essere di tipo `static` per ovvi motivi.
- ☛ **[Più utilizzato] Factory.**
  - **Descrizione:** uno dei pattern creazionali più utilizzati all'interno di Java. La grande utilità è la possibilità di **nascondere tutta la logica** presente dietro la creazione e la gestione di un determinato oggetto. Si comunica con quest'ultimo attraverso una banale interfaccia.
  - **Caratteristiche:** l'interfaccia viene creata con lo scopo di comunicare con le classi all'interno del pattern. Il cliente esterno, o `main`, utilizza come parametro una convenzione, scrivendo una keyword simbolica che sarà utilizzata dall'interfaccia all'interno del pattern per capire quale classe specifica utilizzare.
    Le classi più specifiche implementano l'interfaccia e definiscono (`override`) i suoi metodi.

☛ **Abstract Factory.**

- **Descrizione:** molto simile al pattern Factory. La differenza sostanziale è che in questo caso la logica viene nascosta ancora di più grazie ad una *super-factory*. Quest'ultima non è altro che una Factory di factories, cioè crea altre factory.
- **Caratteristiche:** un cliente, o una classe `main`, non specifica esplicitamente una classe durante la creazione. L'esplicitazione avveniva nella Factory normale, quando alla creazione dell'oggetto veniva dichiarata la classe specifica interessata.
  In questo caso, il cliente dichiara un oggetto riferendosi alla classe Factory (`abstract`) e dicendo ad una classe `FactoryProducer`, cioè produttrice di Factory (super-factory), di creare un determinato oggetto. La creazione avviene come il pattern Factory. Per capire meglio, si osservi il seguente codice:

```
public static void main(String[] args) {
        // Ottenimento di una figura tramite le factory
        AbstractFactory shapeFactory =
                FactoryProducer.getFactory("SHAPE");
        Shape shape = shapeFactory.getShape("CIRCLE");

        // Possono esistere anche piu' factory!
        AbstractFactory colorFactory =
                FactoryProducer.getFactory("COLOR");
        Color color = colorFactory.getColor("RED");
}
```

  Come si vede dal codice, la `AbstractFactory` si interfaccia con il cliente. Invece, la `FactoryProducer` è la classe adibita alla creazione delle Factory. Nell'esempio, viene creata una Factory per le forme e una per i colori.
  Una volta creata una Factory specifica, il funzionamento è simile a una Factory. Ovviamente non viene specificata la classe specifica, ma viene usato al suo posto `nome-oggetto.getOggetto("PARAMETRO")`.

3 Comportamentali

Obbiettivo. Risolvere problemi a livello della struttura delle classi, semplificando la gestione di esse.

- ☛ **Observer.**
  - **Descrizione:** utilizzato quando c'è una relazione "uno-a-molti" tra gli oggetti. Per esempio, se viene modificato un oggetto, tutti gli oggetti dipendenti vengono modificati a loro volta.
  - **Caratteristiche:** viene creato un soggetto `subject` che ha la possibilità di aggiungere nuovi oggetti alla lista di elementi da notificare in caso di cambiamento, di modificare il valore dell'oggetto o di visualizzarne lo stato. Una volta creato il soggetto, una classe esterna, per esempio il `main`, può modificare il valore dell'oggetto; la modifica verrà presa in carico dal `subject` che si occuperà di notificare tutti i membri del gruppo.

☛ **[Il più richiesto] Template.**

- **Descrizione:** una classe astratta espone i suoi metodi che possono essere sovrascritti (**override**) in delle classi più specifiche che estendono tale classe astratta.
  Attenzione! Questo pattern viene utilizzato per eseguire una serie di operazioni (scritte nei metodi) in un ordine ben preciso deciso dal programmatore.
- **Caratteristiche:** viene creata una classe astratta contenente almeno un metodo che ha al suo interno una serie di metodi non definiti. Per esempio, la classe astratta avrà al suo interno un metodo `play` contenente i metodi:

```
public final void play() {
        // initialize the game
        intialize();

        // start game
        startPlay();

        // end game
        endPlay();
}
```

  Questi metodi sono scritti seguendo un certo ordine (per esempio il metodo `play()`, ha al suo interno i metodi `initialize()`, `startPlay()` e `endPlay()` in ordine cronologico).
  Successivamente, vengono create delle classi più specifiche che implementano la classe astratta e definiscono il comportamento dei vari metodi non specificati (nell'esempio i metodi non specificati sono `initialize()`, `startPlay()` e `endPlay()`).
  Infine, la classe `main` creerà un oggetto di tipo corrispondente alla classe astratta, ma dopo la keyword `new` utilizzerà la classe specifica interessata. Così facendo, potrà richiamare il metodo della classe astratta, il quale avrà una serie di metodi messi con un ordine ben specifico.
  Quindi, se la classe astratta si chiama `Game` e classi specifiche si chiamano `Football` e `Cricket`, la dichiarazione dell'oggetto per eseguire una partita di calcio sarà:

```
public static void main(String[] args) {
        Game game = new Football();

        // play football
        game.play();
}
```

☛ **[Il più utilizzato] Iterator.**

- **Descrizione:** utilizzato spesso nella programmazione in Java e .NET. L'obbiettivo è quello di **ottenere un accesso** agli elementi di una **collezione di oggetti** in maniera sequenziale, **senza conoscere la rappresentazione sottostante**.
- **Caratteristiche:** vengono create due interfacce, un iteratore e un contenitore.
  La prima è necessaria per scorrere tutti gli elementi di una lista, quindi i classici `next()` e `hasNext()`.
  La seconda viene utilizzata per contenere l'iteratore e ritornare semplicemente l'oggetto.
  Quindi, viene creata una classe che ha l'obbiettivo di essere una *repository* in cui si ha l'estensione di `Container`, perché deve restituire l'oggetto creato, e in cui si ha una classe privata che estende `Iterator` perché si deve ciclare su tutti gli elementi della lista.
  Un esempio di classe *repository* (si omette la banale creazione delle interfacce):

```
public class NameRepository implements Container {
        public String names[] = {"Robert", "John",
                "Julie", "Lora"};
        @Override
        public Iterator getIterator() {
                return new NameIterator();
        }
        private class NameIterator implements Iterator {
                int index;
                @Override
                public boolean hasNext() {
                        if(index < names.length){
                                return true;
                        }
                        return false;
                }
                @Override
                public Object next() {
                        if(this.hasNext()){
                                return names[index++];
                        }
                        return null;
                }
        }
}
```

Mentre una classe `Main` con l'obbiettivo di scorrere la lista:

```
public class IteratorPatternDemo {
        public static void main(String[] args) {
                NameRepository nameRepository =
                new NameRepository();
                for(Iterator iter = nameRepository.getIterator();
                iter.hasNext();){
                        String name = (String)iter.next();
                        System.out.println("Name : " + name);
                }
        }
}
```

4 Architetturali

Obbiettivo. Risolvere problemi comuni riguardo la progettazione di software aziendali.

- ☛ **Data Access Object (DAO).**
  - **Descrizione:** viene utilizzato per separare l'accesso alle operazioni sui dati a basso livello dalla parte di accesso alle operazioni sui dati ad alto livello.
    Esistono tre soggetti principali in questo pattern:
    - ☆ **Data Access Object Interface**, interfaccia che definisce le operazioni che possono essere eseguite;
    - ☆ **Data Access Object Concrete Class**, classe che implementa l'interfaccia definita al punto precedente. Il compito di questa classe è quello di recuperare dei dati da una risorsa, come un file XML, un database o altro;
    - ☆ **Model Object or Value Object**, oggetto di tipo POJO contenente i metodi per impostare o ottenere i dati recuperati usando la classe definita al punto precedente.
  - **Caratteristiche:** Il cliente si interfaccia sempre con la classe DAO che gli consente di eseguire varie operazioni di alto livello sui dati. Quindi, le operazioni di basso livello, se ne occupa la `Concrete Class` che insieme agli altri due soggetti soddisfa le richieste del cliente.