Metodologie di programmazione
Relazione progetto

Autore: Felici Simone
AA: 2022/2023
Matricola: 7013439
Versione: Java 8

Funzionalità del sistema

Il progetto implementa un gestionale di un negozio di scarpe, il negozio possiede un catalogo che a sua volta può contenere sotto cataloghi o scarpe, su entrambi sono definite delle operazioni al fine di ottenere informazioni sugli elementi ed il prezzo, quest'ultimo fornisce il prezzo della scarpa, se applicato su un oggetto di tipo scarpa, oppure fornisce il prezzo totale (dato dalla somma) di ogni scarpa contenuta nel catalogo, se applicato su un oggetto di tipo Catalogo.

Le scarpe inoltre possono essere decorate, aggiungendo quindi funzionalità quali:

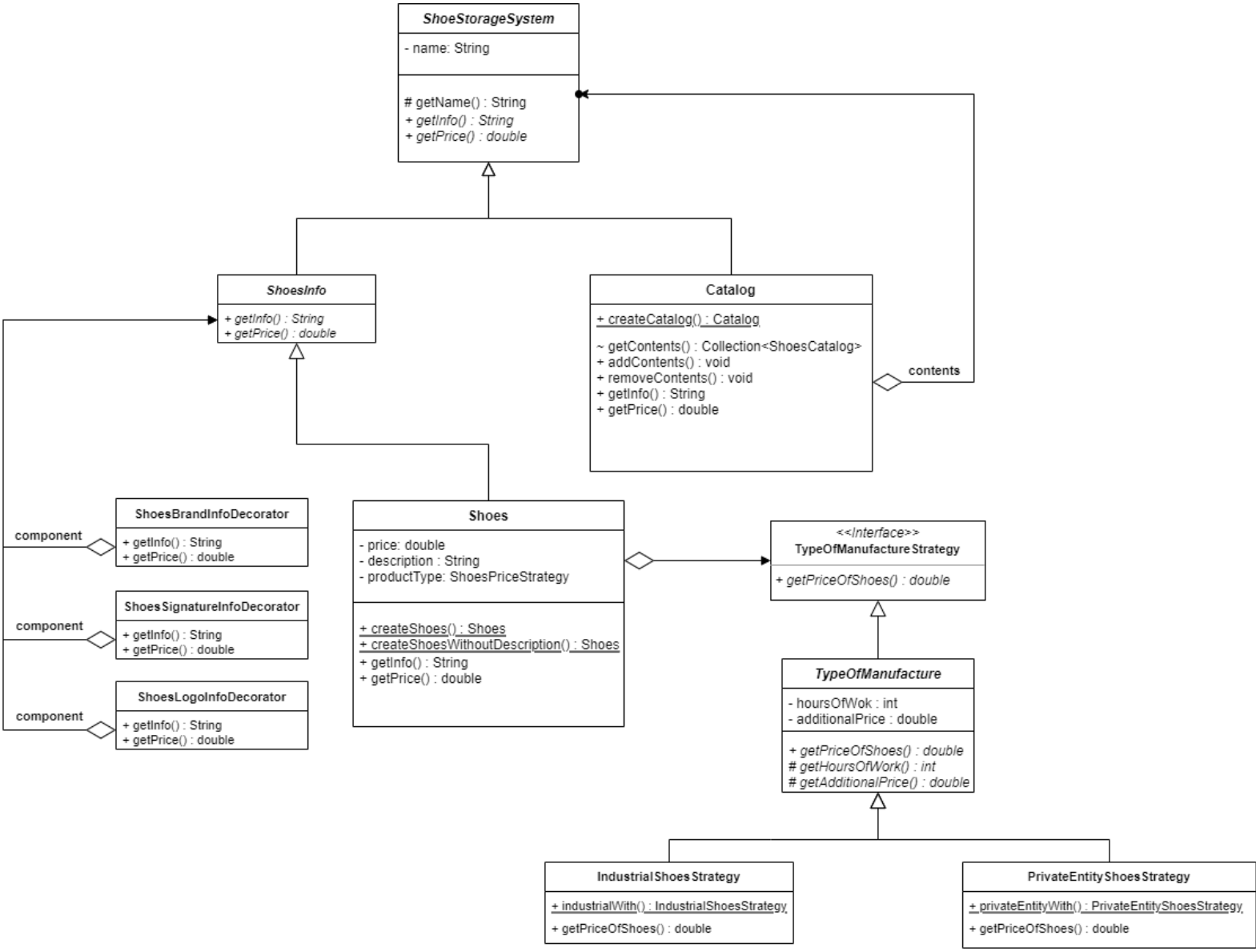
- il nome del brand;
- la firma della casa produttrice;
- il logo;

l'utilizzo di queste decorazioni modifica quindi le informazioni degli oggetti, aggiungendo l'eventuale nome del brand, logo, o la firma della casa produttrice, ed aumenta il prezzo in base a quale di queste decorazioni viene utilizzata sulle scarpe.

Il sistema fornisce inoltre, per le scarpe, un metodo di calcolo del prezzo, diverso in base al tipo di fabbricazione della scarpa, questa cambierà il prezzo iniziale della scarpa in base alle ore di lavoro ed in base ad un prezzo addizionale, quest'ultimo cambia in base al tipo di fabbricazione (manodopera o tassa industriale).

Per la creazione degli oggetti vengono forniti dei metodi statici al fine di semplificare la creazione di questi. Al momento della creazione le scarpe possono avere una descrizione oppure farne a meno.

DIAGRAMMA UML



PATTERN APPLICATI

- Composite
- Decorator
- Strategy
- Static Factory Methods

SCELTE DI DESIGN & IMPLEMENTATIVE

➢ ***Composite:*** il pattern, implementato nella sua variante type safe, è stato usato per definire la struttura del sistema, infatti il sistema progettato permette di lavorare con un catalogo, che a sua volta può contenere sotto cataloghi (*Composite*) oppure scarpe (*Leaf*). Il catalogo e le scarpe hanno in comune soltanto il campo del nome poi, il catalogo si mantiene una collezione di oggetti astratti *ShoeStorageSystem*, le scarpe invece hanno i campi riguardanti il prezzo, un'eventuale descrizione (vedi pattern *Static Factory Method*) ed un campo per il tipo di prodotto (vedi pattern *Strategy*). Nel catalogo il metodo *getPrice()* restituisce il prezzo complessivo di tutte le scarpe contenute dei cataloghi (e nei sotto cataloghi), mentre il metodo *getInfo()* fornisce il nome del catalogo (e dei sotto cataloghi) e per le scarpe fornisce il nome e la sua eventuale descrizione.
Nota: il metodo *getContents()* è stato reso package private dato che il suo utilizzo era necessario solo all'interno dei test.

```
Collection<ShoeStorageSystem> getContents() {
    return contents;
}
```

- ***Decorator:*** il decorator è stato usato per decorare solamente le foglie (*Shoes*) del pattern *Composite*, nel sistema è stata implementata la versione del decorator senza la classe comune ad i decorator concreti, quindi ogni decorator concreto ha un riferimento alla classe astratta *ShoesInfo*, questa è "l'interfaccia" (nel progetto è una classe astratta) comune all'oggetto decorato (*Shoes*) ed ai decoratori concreti. Siccome non vogliamo che il componente passato al decoratore sia *null*, facciamo un controllo con il metodo *requireNonNull* dellla classe *Objects*:

```
public ShoesBrandInfoDecorator(String name, ShoesInfo component) {
    super(name);
    requireNonNull(component, "Component cannot be null");
    this.shoesInfo = component;
}
```

possiamo inoltre notare come ogni decorator sfrutta il costruttore del Composite per istanziare il campo che poi aggiungerà informazioni al metodo *getInfo()*.
Nel sistema abbiamo 3 decoratori concreti:

- **ShoesBrandInfoDecorator**
- **ShoesSignatureInfoDecorator**
- **ShoesLogoInfoDecorator**

I decoratori, in base a quale viene applicato, modificano le informazioni del metodo *getInfo()* aggiungendo quindi eventualmente il nome del Brand, la firma della casa produttrice o il logo del brand, inoltre modificando anche il prezzo come mostrato nell'immagine qui sotto:

```
@Override
public double getPrice() {
    return shoesInfo.getPrice() + 10.0;
}

@Override
public String getInfo() {
    return shoesInfo.getInfo() + " of brand: " + this.getName();
}
```

- ***Strategy:*** il pattern viene utilizzato per astrarre il tipo di prodotto (campo della classe *Shoes*). L'interfaccia funzionale *TypeOfManufactureStrategy* fornisce il singolo metodo astratto *getPriceOfShoes()* che riceve un double (il prezzo senza le lavorazioni). La classe astratta *TypeOfManufacture* implementa l'interfaccia sopra citata ed aggiunge due campi *hoursOfWork* e *additionalPrice*, il costruttore poi controlla che nessuno dei due campi sia **non positivo**:

```
public TypeOfManufacture(int hoursOfWork, double additionalPrice) {
    if(hoursOfWork <= 0 || additionalPrice <= 0.0) {
        throw new IllegalArgumentException("Cannot insert negative input, "
                                           + "your input is: "
                                           + hoursOfWork
                                           +", "
                                           + additionalPrice);
    }
    this.hoursOfWork = hoursOfWork;
    this.additionalPrice = additionalPrice;
}
```

I due getter sono protected al fine di essere usati dalle sottoclassi solamente per ottenere i valori dei due campi.
Abbiamo due strategy concreti che cambiano in base al calcolo del prezzo, nello specifico:

- ***PrivateEntityShoesStrategy:*** al prezzo originale viene aggiunta la moltiplicazione delle ore di lavoro per il prezzo della manodopera.

```
@Override
public double getPriceOfShoes(double price) {
    return price + this.getAdditionalPrice() * this.getHoursOfWork();
}
```

- ***IndustrialShoesStrategy:*** al prezzo originale viene aggiunta una tassa industriale che fornirà da percentuale calcolata sul prezzo e la moltiplicazione delle ore di lavoro per 8 (prezzo fisso per 1 ora di lavoro).

```
@Override
public double getPriceOfShoes(double price) {
    return price + (price * this.getAdditionalPrice()/100) + (this.getHoursOfWork() * 8);
}
```

➢ ***Static Factory Method:*** questo pattern è stato utilizzato all'interno delle seguenti classi:

- ***Catalog:***

```
public static Catalog createCatalog(String name) {
    return new Catalog(name);
}
```

- ***Shoes:*** nella quale sono stati forniti due possibili scelte in modo da poter costruire un oggetto Shoes con oppure senza descrizione.

```
public static Shoes createShoes(String name, double price, String description, TypeOfManufactureStrategy productType) {
    return new Shoes(name, price, description, productType);
}

public static Shoes createShoesWithoutDescription(String name, double price, TypeOfManufactureStrategy productType) {
    return new Shoes(name, price, "", productType);
}
```

- ***PrivateEntityShoesStrategy:***

```
public static PrivateEntityShoesStrategy privateEntityWith(int hoursOfWork, double labor) {
    return new PrivateEntityShoesStrategy(hoursOfWork, labor);
}
```

- ***IndustrialShoesStrategy:***

```
public static IndustrialShoesStrategy industrialWith(int hoursOfWork, double industrialTax) {
    return new IndustrialShoesStrategy(hoursOfWork, industrialTax);
}
```

I costruttori delle calassi concrete sono stati resi privati mentre, nelle classi astratte (*ShoeStorageSystem* e *TypeOfManufacure)* questi sono protected in modo da essere utilizzati dalle sottoclassi e non private dato che non è possibile istanziare classi astratte quindi l'unico modo quello di istanziare classi concrete con i propri static factory method.

➢ ***Test*****:** i test, presenti nella source-folder *testing*, sono stati suddivisi in package diversi con nome relativo al package della classe testata al suo interno.

- ***storagesystem.structure:*** all'interno di questo pacchetto abbiamo due classi, una per testare la classe Catalog ed una per testare la classe Shoes.

All’interno della classe CatalogTesting vengono testati i metodi *addContents()* e *removeContents()*, proprio in questi metodi utilizziamo il metodo package-private *getContents()*. Inoltre vengono testati i metodi *getInfo()* nel caso in cui il catalogo sia vuoto oppure abbiamo elementi:

```
@Test
public void testGetInfoWhenIsEmpty() {
    assertThat(catalog.getInfo()).isEqualTo("Catalog contains: \n");
}

@Test
public void testGetInfo() {
    TypeOfManufactureStrategy industrialType = industrialWith(3, 5.0);
    ShoeStorageSystem airForce1 = createShoesWithoutDescription("AirForce1", 40.0, industrialType);
    ShoeStorageSystem nikeSportwear = createShoesWithoutDescription("B550", 40.0, industrialType);
    Catalog subCatalogue = createCatalog("Sub catalogue");

    catalog.addContents(airForce1);
    subCatalogue.addContents(nikeSportwear);
    catalog.addContents(subCatalogue);

    assertThat(catalog.getInfo())
                    .isEqualTo("Catalog contains: \n"
                                + "Shoes name: AirForce1"
                                + "\n"
                                + "Sub catalogue contains: \n"
                                + "Shoes name: B550"
                                + "\n");
}
```

allo stesso modo viene testato anche il metodo *getPrice()*:

```
@Test
public void testGetPriceWhenIsEmpty() {
    assertThat(catalog.getPrice()).isEqualTo(0.0);
}

@Test
public void testGetPrice() {
    TypeOfManufactureStrategy industrialType = industrialWith(3, 5.0);
    ShoeStorageSystem airForce1 = createShoesWithoutDescription("AirForce1", 40.0, industrialType);
    ShoeStorageSystem nikeSportwear = createShoesWithoutDescription("B550", 40.0, industrialType);
    Catalog subCatalogue = createCatalog("Sub catalogue");

    catalog.addContents(airForce1);
    subCatalogue.addContents(nikeSportwear);
    catalog.addContents(subCatalogue);

    assertThat(catalog.getPrice()).isEqualTo(132.0);
}
```

Nella classe Shoes viene testato che, in base alla strategy, il prezzo effettivamente cambi:

```
@Test
public void testsGetPriceIndustrial() {
    TypeOfManufactureStrategy industrialType = industrialWith(9, 15.0);
    Shoes shoes = createShoesWithoutDescription("Puma 1980", 70.0, industrialType);
    double expectedPrice = 70.0 + (70*15.0/100) + (9*8);

    assertThat(shoes.getPrice()).isEqualTo(expectedPrice);
}

@Test
public void testsGetPricePrivateEntity() {
    TypeOfManufactureStrategy privateType = privateEntityWith(10, 6.0);
    Shoes shoes = createShoesWithoutDescription("Puma 1980", 70.0, privateType);
    double expectedPrice = 70.0 + (6.0 * 10);

    assertThat(shoes.getPrice()).isEqualTo(expectedPrice);
}
```

Inoltre viene testato anche il metodo *getInfo()* nel caso in cui si abbia la descrizione e nel caso in cui non si abbia:

```
@Test
public void testsGetInfo() {
    TypeOfManufactureStrategy industrialType = industrialWith(3, 5.0);
    Shoes shoes = createShoes("NewBalance",
                              55.0,
                              "Tip:round; Closure:laces; Material:fabric;",
                              industrialType);

    assertThat(shoes.getInfo()).isEqualTo("Shoes name: NewBalance"
                                          + "Tip:round; Closure:laces; Material:fabric;"
                                          + "\n");
}

@Test
public void testsGetInfoWithoutDescription() {
    TypeOfManufactureStrategy industrialType = industrialWith(3, 5.0);
    Shoes shoes = createShoesWithoutDescription("NewBalance", 55.0, industrialType);

    assertThat(shoes.getInfo()).isEqualTo("Shoes name: "
                                          + "NewBalance"
                                          +"\n");
}
```

***storagesystem.shoesdecorator*:** all'interno di questo pacchetto abbiamo la classe per testare il decorator, vengono testati sia *getInfo()* che *getPrice()* combinando i vari decorator.

Inoltre viene testato il caso in cui il componente passato al decorator sia null:

```
@Test
public void testComponentNull() {
    assertThatThrownBy(() -> new ShoesBrandInfoDecorator("Puma",null))
    .isInstanceOf(NullPointerException.class)
    .hasMessage("Component cannot be null");
}
```

inoltre vengono testati i metodi *getInfo()* e *getPrice()* nei casi in cui si abbia una struttua aggregata di sotto cataloghi, scarpe e scarpe decorate:

```
@Test
public void testAggregateCatalog() {
    Catalog catalog = createCatalog("Catalog");
    TypeOfManufactureStrategy industrialType = industrialWith(3, 5.0);
    ShoeStorageSystem airForce1 = createShoesWithoutDescription("AirForce1", 40.0, industrialType);
    ShoesInfo shoesDecorated = new ShoesLogoInfoDecorator("Nike logo",
                                    new ShoesBrandInfoDecorator("Nike",
                                            createShoesWithoutDescription("B550",
                                                                          40.0,
                                                                          industrialType)));
    Catalog subCatalogue = createCatalog("Sub catalogue");

    catalog.addContents(airForce1);
    catalog.addContents(shoes);
    subCatalogue.addContents(shoesDecorated);
    catalog.addContents(subCatalogue);

    assertThat(catalog.getPrice()).isEqualTo(297.0);
    assertThat(catalog.getInfo()).isEqualTo("Catalog contains: \n"
                                        + "Shoes name: AirForce1\n"
                                        + "Shoes name: Puma 1980\n"
                                        + "Sub catalogue contains: \n"
                                        + "Shoes name: B550\n"
                                        + " of brand: Nike with logo: Nike logo");
```

- ***storagesystem.price.strategy:*** all'interno di questo pacchetto abbiamo la classe per testare lo strategy utilizzato. Testiamo il caso in cui gli input siano entrambi non positivi (e nel caso non ne sia uno dei due):

```
@Test
public void testBothNegativeInput() {
    assertThatThrownBy(() -> privateEntityWith(0, 0.0))
    .isInstanceOf(IllegalArgumentException.class)
    .hasMessage("Cannot insert negative input, your input is: 0, 0.0");
}
```

inoltre testiamo il metodo *getPriceOfShoes()* di entrambi gli strategy concreti in base al valore che ci aspettiamo:

```
@Test
public void testPrivateGetPriceOfShoes() {
    TypeOfManufactureStrategy privateTypeManufacture = privateEntityWith(10, 6.0);
    double expectedValue = 30.0 + (6.0 * 10);

    assertThat(privateTypeManufacture.getPriceOfShoes(30.0)).isEqualTo(expectedValue);
}

@Test
public void testIndustrialGetPriceOfShoes() {
    TypeOfManufactureStrategy industrialTypeManufacture = industrialWith(9, 15.0);
    double expectedValue = 50.0 + (50.0 * 15.0/100.) + (9 * 8);

    assertThat(industrialTypeManufacture.getPriceOfShoes(50.0)).isEqualTo(expectedValue);
}
```