**Metodologie di Programmazione**

Pierpaolo D’Angelo

**Introduzione**

Questo progetto si propone di modellare la base di uno shop online, all’interno del quale sono rappresentati dei prodotti, che possono essere o degli articoli o dei raggruppamenti eventualmente scontati di altri prodotti.

Sono presenti, inoltre, una serie di meccanismi che offrono la possibilità di manipolare e organizzare insiemi di prodotti, ad esempio:

* Monitorare il prezzo di un prodotto nel tempo.
* Monitorare il numero di prodotti di un certo tipo nel tempo.
* Applicare uno sconto aggiuntivo se il numero di articoli di un certo tipo è superiore a una certa soglia.
* Estrarre il prezzo maggiore e minore dei prodotti di un determinato tipo.

Pattern utilizzati all’interno del progetto:

* Composite
* Observer
* Visitor
* Strategy
* Factory

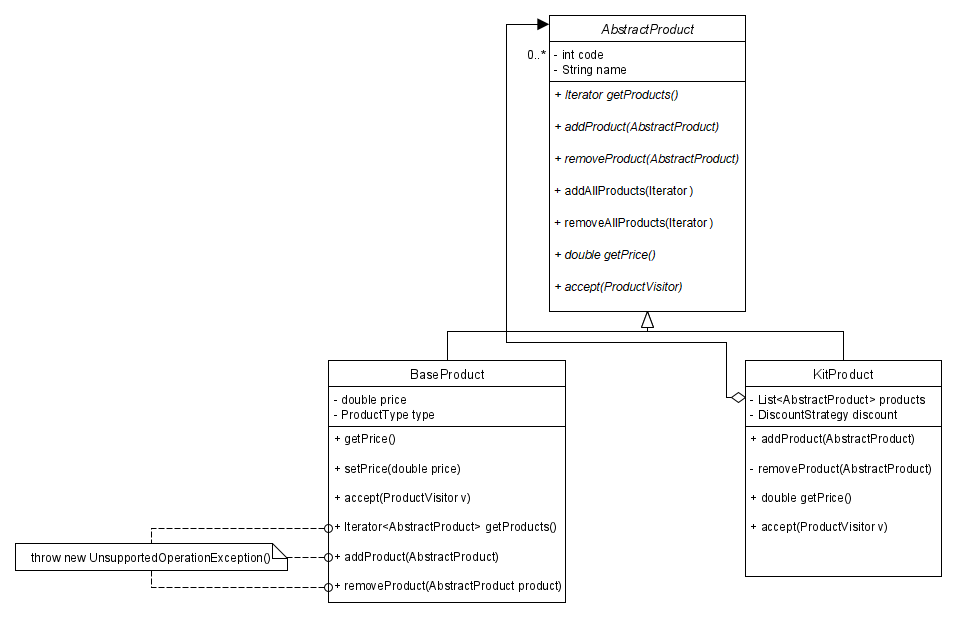
**Scelte implementative**

**Composite**

Per la modellazione della gerarchia dei prodotti è stato scelto di utilizzare un Composite, nella sua variante Design for Uniformity. In particolare, la classe astratta *AbstractProduct* presenta un campo codice e un campo nome ed espone, oltre ai getter, dei metodi per manipolare una lista di *AbstractProduct*. I metodi *addAllProducts* e *removeAllProducts* sono dei template methods, poiché si basano sull’aggiunta e la rimozione singola che invece vengono lasciate da implementare alle sottoclassi.

La classe *BaseProduct* consiste in una classe leaf, aggiunge i campi relativi a prezzo e tipo, e ridefinisce i metodi *addProduct*, *removeProduct* e *getProducts*, i quali, rappresentando operazioni non supportate dal BaseProduct sollevano un eccezione di tipo *UnsupportedOperationException*. Viene inoltre ridefinito il metodo *getPrice*, che qui restituisce il valore mantenuto dal campo price, e aggiunto il relativo setter.

La classe *KitProduct* consiste in una classe composite, aggiunge due campi, una *DiscountStrategy* e una lista di *AbstractProduct*. Vengono ridefiniti i metodi *addProduct* e *removeProduct*, i quali aggiungono e rimuovono un prodotto dalla lista e *getProducts* che restituisce l’iterator di quest’ultima. Il prezzo in questa classe non è un campo, ma viene calcolato dal metodo *getPrice* sommando i prezzi di tutti gli *AbstractProduct* presenti nella lista dei prodotti e applicando lo sconto specificato dalla *DiscountStrategy*.



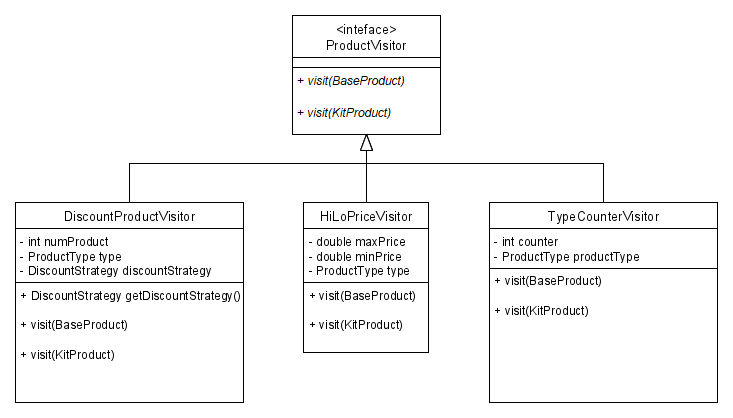
**Visitor**

Per poter manipolare un insieme di prodotti senza preoccuparsi della differenza tra *BaseProduct* e *KitProduct*, viene utilizzato un Visitor.

L’interfaccia *ProductVisitor* descrive due metodi visit in overload, i quali accettano come parametro rispettivamente un *BaseProduct* o un *KitProduct*.

Le implementazioni concrete di *ProductVisitor* sono le seguenti:

* **DiscountProductVisitor**: questo Visitor si occupa di decidere se uno sconto deve essere applicato o meno a seconda di quanti prodotti di uno stesso tipo sono contenuti all’interno di un *AbstractProduct*. Se al metodo visit viene passato un *BaseProduct* e questo è del tipo interessato, il contatore inizializzato in costruzione viene decrementato, altrimenti, viene invocato l’*accept* su ogni elemento del *KitProduct*. Il metodo *getDiscountStrategy* restituisce la *DiscountStrategy* passata in costruzione se si è raggiunto il numero di prodotti specificato, altrimenti la *DiscountStrategy*.*NO\_DISCOUNT* (che non applica nessuno sconto).
* **HiLoPriceVisitor**: questo Visitor si occupa, dato un determinato tipo, di determinare il prezzo più alto e più basso dei prodotti contenuti in un *AbstractProduct*. Se al metodo *visit* viene passato un *BaseProduct* del tipo interessato e il prezzo corrisponde al massimo o al minimo incontrato, la rispettiva variabile di istanza viene aggiornata, altrimenti, viene invocato l’*accept* su ogni elemento del *KitProduct*.
* **TypeCounterVisitor**: questo Visitor si occupa di contare il numero di prodotti di un determinato tipo all’interno di un *AbstractProduct*. Se al metodo *visit* viene passato un *BaseProduct* del tipo interessato il contatore viene incrementato, altrimenti, viene invocato l’*accept* su ogni elemento del *KitProduct*.



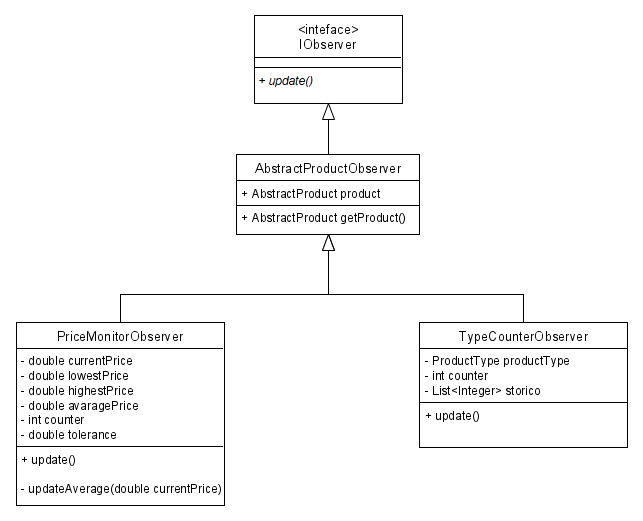
**Observer**

Per poter monitorare le variazioni subite nel tempo dagli *AbstractProduct* viene usato un Observer.

La classe astratta *AbstractSubject* ha come unico campo una lista di observer e i metodi ad essa utili. L’interfaccia *IObserver* descrive un unico metodo *update*. È stata introdotta una classe astratta intermedia *AbstractProductObserver* che centralizza la sottoscrizione dell’observer su un *AbstractProduct*.

Le implementazioni concrete di *IObserver* sono le seguenti:

* **PriceMonitorObserver**: questo Observer monitora nel tempo le variazioni del prezzo di un *AbstractProduct*. Quando viene chiamato il metodo update, se il prezzo è cambiato rispetto a quello precedentemente salvato, aggiorna il prezzo medio e se necessario aggiorna anche il prezzo più alto e più basso.
* **TypeCounterObserver**: questo Observer mantiene uno storico di quanti prodotti di un determinato tipo sono contenuti all’interno di un *AbstractProduct*. Quando viene chiamato il metodo *update*, un oggetto di tipo *TypeCounterVisitor* viene fatto accettare all’*AbstractProduct*, se il numero dei prodotti del tipo monitorato è cambiato, viene aggiornato lo storico.

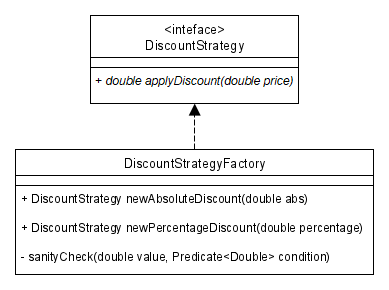


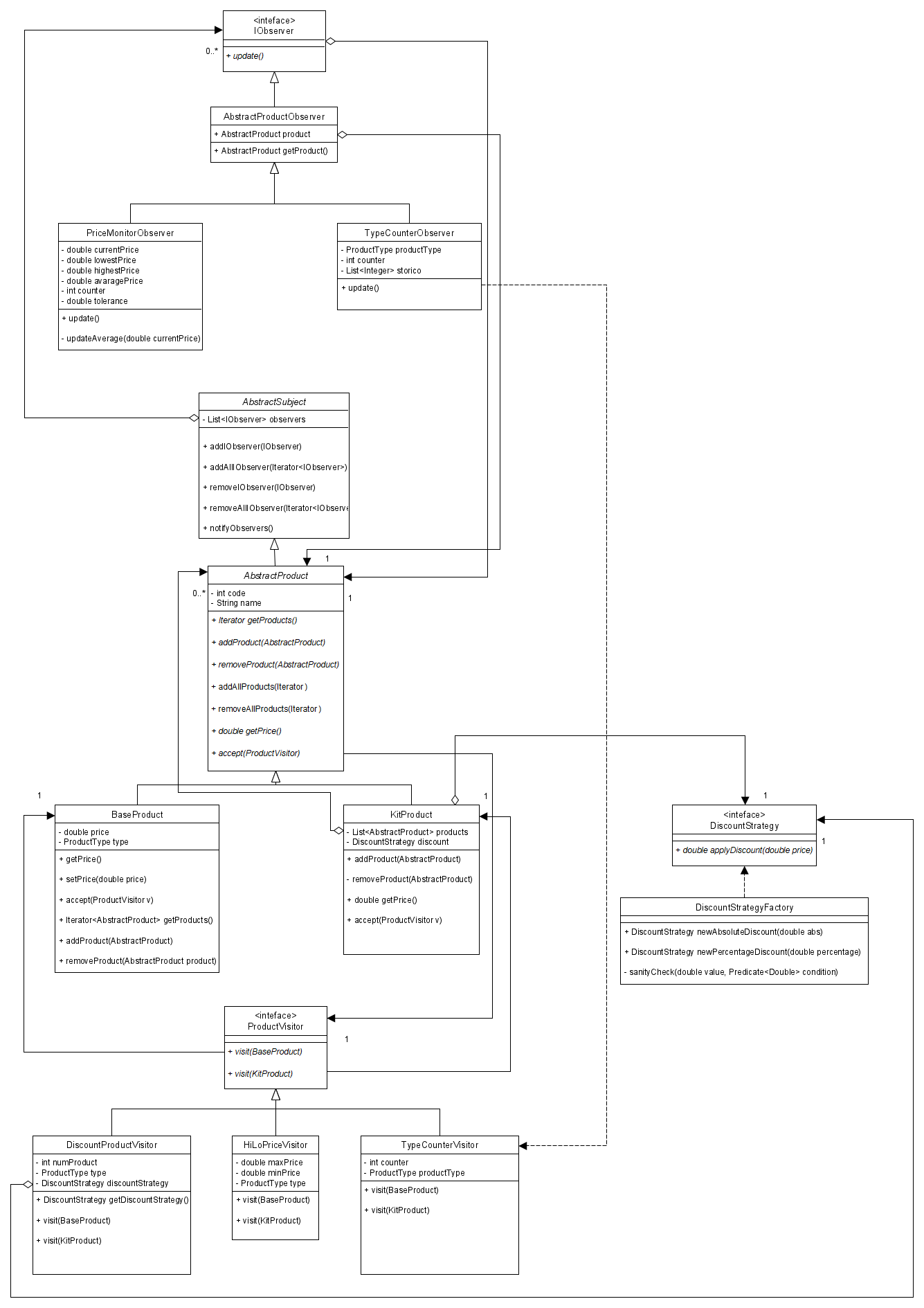
**Strategy e Factory**

Per implementare diverse strategie di sconto è stato usato uno Strategy.

L’interfaccia funzionale *DiscountStrategy* descrive un metodo *applyDiscount* che prende in input un prezzo e ne restituisce un altro.

Gli oggetti di tipo *DiscountStrategy* vengono creati da un’apposita Factory. In *DiscountStrategyFactory*, il metodo *newAbsoluteDiscount* costruisce una *DiscountStrategy* che riduce il prezzo di un importo pari al parametro specificato, mentre il metodo *newPercentageDiscount* riduce il prezzo di una percentuale passata come parametro. Infine, è anche presente il campo statico e pubblico *NO\_DISCOUNT* che lascia inalterato il prezzo.





**Test**

**Composite**

**BaseProduct**:

* *getProducts*, *addProduct*, *removeProduct*: si controlla che l’invocazione di questi metodi su un *BaseProduct* lanci un’eccezione di tipo *UnsupportedOperationException*.

**KitProduct**:

* Si controlla che costruendo un *KitProduct* senza una *DiscountStrategy* venga utilizzato NO\_DISCOUNT.
* Si controlla che costruendo un *KitProduct* con una *DiscountStrategy* il corrispondente sconto venga applicato.
* Vengono testati tutti i metodi di aggiunta e rimozione di un *AbstractProduct* dalla lista products.

**Observer**

**PriceMonitorObserver**:

* Si controlla che un *PriceMonitorObserver* appena instanziato abbia i valori corretti.
* Si controlla che chiamando il metodo *setPrice* passando come parametro un prezzo uguale al prezzo attuale la media non venga aggiornata.
* Si controlla che dopo aver chiamato tre volte il metodo *setPrice* con parametri significativi, la media, *lowestPrice*, *highestPrice* e counter siano aggiornati di conseguenza.

**TypeCounterObserver**:

* Si controlla che un *TypeCounterObserver* appena instanziato abbia i valori corretti.
* Si controlla che aggiungendo un prodotto del tipo monitorato lo storico venga aggiornato.
* Si controlla che aggiungendo un prodotto non del tipo monitorato lo storico non venga aggiornato.

**Strategy**

**DiscountStrategy**:

* Si controlla che tutti e tre i metodi forniti dalla Factory restituiscano valori corretti.
* Si controlla che passando a *newAbsoluteDiscount* come parametro un valore più grande del prezzo, il valore restituito da *applyDiscount* sia 0.
* Si controlla che passando a *newAbsoluteDiscount* come parametro un valore negativo venga lanciata l’eccezione di tipo *IllegalArgumentException*.
* Si controlla che passando a *newPercentageDiscount* come parametro un valore più grande di 100 o minore di 0 venga lanciata l’eccezione di tipo *IllegalArgumentException*.

**Visitor**

**DiscountProductVisitor**:

* Si controlla che il visitor restituista la *DiscountStrategy* passata in costruzione solo se all’interno dell’*AbstractProduct* sono presenti il numero minimo di prodotti di tipo scelto.

**HiLoPriceVisitor**:

* Si controlla che il visitor salvi il prezzo massimo e il prezzo minimo dei prodotti di tipo scelto presenti nell’*AbstractProduct*.

**TypeCounterVisitor**:

* Si controlla che il visitor conteggi il numero corretto di prodotti del tipo scelto all’interno dell’*AbstractProduct*.