Candidati:

Tommaso Capecchi:
-matricola: 5943118
-email: tommaso.capecchi@stud.unifi.it
-prova in itinere non superata

Lorenzo Mungai:
-matricola: 5962693
-email: lorenzo.mungai@stud.unifi.it
-prova in itinere non superata

Relazione

Scelta dell'esercizio esercizio:

L'esercizio da noi implementato riguarda lo Shop Online. Le specifiche chiedevano di realizzare una sorta di negozio composto da diversi oggetti acquistabili. Il nostro schema riguarda un negozio di prodotti tecnologici (telefoni, computer, pad ecc…). Il cliente ha a disposizione un carrello dove può inserire e rimuovere oggetti acquistati; inoltre può avere informazioni sugli oggetti contenuti nello stesso (come ad esempio il prezzo o il nome dell'oggetto). Il negozio è fornito di speciali offerte composte da pacchetti di oggetti. La fattura del cliente viene aggiornata all'aggiunta di nuovi oggetti o pacchetti all'interno del carrello. Il negozio offre uno sconto del 6% a clienti studenti e dell'8% a clienti anziani. E' possibile effettuare dei report per verificare quale siano gli oggetti più costosi o più economici del negozio; tale funzione può essere utilizzata anche per monitorare gli oggetti contenuti nel carrello. Al momento del pagamento il cliente può decidere se accettare o meno di pagare il

totale; ciò è permesso grazie all'utilizzo della classe Scanner di Java: basterà digitare [YES / NO]. Se la risposta è positiva il cliente può effettuare il pagamento utilizzando tre opzioni:

- carta di credito
- bonifico
- PayPal

Scelta dei Pattern

I pattern da noi scelti ed implementati sono i seguenti: Composite, Strategy, Observer e Template.

Pattern Composite

Nel nostro esercizio abbiamo realizzato questo pattern definendo una classe astratta "ShopOnline" che rappresenta il tipo comune sia per oggetti singoli (definiti dalla classe "Item") che per le composizioni di oggetti (definiti dalle classi "Pack","Shop" e "Cart"); questo fatto ci permette di trattarli uniformemente, potendo cosi invocare gli stessi metodi sia sulle "leaves" che sui vari "composite". Uno dei vantaggi principali nell'uso di questo pattern è quello di permettere di descrivere una composizione ricorsiva cosi che il "Client" non si preoccupi del fatto che l'oggetto che sta trattando sia un oggetto singolo o una composizione di oggetti: infatti il Client usa l'interfaccia "ShopOnline" per interagire con elementi della struttura, se tale elemento è un "Item", cioè una foglia, la richiesta viene elaborata immediatamente, se invece l'elemento è un composite, cioè un nodo, questo invia tale richiesta ai sui figli (*uso della delegation cioè viene delegata un'operazione a un altro oggetto*), fino a che non giunge ad una foglia, la quale finalmente potrà eseguire l'operazione. La sottoclasse "Shop" rappresenta il negozio concreto, ed è dunque composto da un nome e da una lista alla quale vengono aggiunti tramite il metodo ereditato da ShopOnline "addItem" tutti i prodotti e i pacchetti che può vendere il negozio. Le varie

composizioni nella gerarchia sono rappresentate dalla classe “Pack” e “Cart”. Entrambe hanno una lista parametrica di tipo <ShopOnline> in quanto possono trattare sia oggetti singoli che composizioni poiché estendono ShopOnline. Da segnalare il fatto che la classe astratta viene estesa sia dalle foglie che dai vari composti, anche se questi all’interno della gerarchia, hanno ruoli diversi e dunque alcuni metodi che vengono ereditati per alcune classi non hanno senso(ad esempio il metodo “addItem” per la classe Item non ha alcun significato); per poter risolvere questo problema i metodi di ShopOnline devono poter lanciare un “Exception” che verrà poi gestita nelle varie sottoclassi.

Pattern Strategy

Abbiamo scelto questo pattern per risolvere il problema di applicare sconti a prodotti singoli ma anche a pacchetti speciali. Il pattern Strategy permette all’algoritmo di variare al run-time a seconda delle richieste del client. Una delle caratteristiche principali di questo pattern è l’intercambiabilità; in altre parole due algoritmi appartenenti alla stessa “famiglia” cioè con la stessa funzionalità, devono implementare la medesima interfaccia cosi che il client sia dipendente solo dall’ interfaccia e non dalle varie implementazioni. Nel nostro caso abbiamo deciso che il negozio può applicare sconti solo a studenti o ad anziani, rispettivamente del 6% e dell’8%, quindi abbiamo astratto un’ interfaccia comune dal nome “DiscountStrategy” avente un solo metodo “applyDiscount” (*Single Resposability Principle*: *una classe deve avere un singolo motivo per poter essere modificata*); tale metodo è stato implementato in una classe “StudentDiscount” (che applica un 6% di sconto), ed in una classe “ElderDiscount” (che applica un 8% di sconto). Il client quindi non deve far altro che scegliere che tipo di sconto applicare tramite il metodo “setDiscountStrategy” disponibile nella classe astratta ShopOnline e questo verrà applicato senza modifiche alcune nel codice.

Pattern Observer

La scelta dell'uso di questo pattern è dovuta alla risoluzione del problema di visualizzare i vari oggetti che vengono inseriti nel carrello e conseguentemente nella fattura "provvisoria". Poiché il pattern Observer si basa su una relazione tra oggetti del tipo *one-to-many*, quando lo stato di un oggetto osservato cambia, tutti i suoi oggetti dipendenti cambiano; nel nostro caso questo modello è stato implementato con un'interfaccia "Observer" implementata a sua volta da un' observer concreto di nome "Bill" che rappresenta la fattura. Il Soggetto, cioè l'osservato, è definito dall'interfaccia "Subject" che permette di registrare, rimuovere e sopratutto notificare tutti i suoi osservatori. L'interfaccia "Subject" viene implementata da "ShopOnline" in quanto l'osservato è si la classe "Cart", ma questa estende la classe astratta "ShopOnline". Cosi tutte le volte che in cart viene aggiunto un item, questo chiama il metodo "notifyObserver" che aggiorna a sua volta la "Bill" aggiungendo l'elemento nella fattura. Questo meccanismo si basa sul ***principio di Hollywood,*** cioè non sono le sottoclassi a invocare direttamente il metodo concreto della superclasse, ma sarà la superclasse ad invocarlo all'occorrenza (***"non chiamarci, ti chiameremo noi"***). Questo pattern permette inoltre di costruire un codice flessibile grazie al fenomeno del "loosely coopled", cioè due oggetti possono interagire, ma "conoscendosi" relativamente poco; infatti l'unica cosa che il soggetto conosce dell' osservatore è proprio che questo implementa l'interfaccia "Observer".

Pattern Template

Infine, per risolvere il problema del pagamento del carrello, abbiamo scelto di utilizzare il pattern Template che permette di "incapsulare" un algoritmo; questo pattern infatti definisce la struttura di un algoritmo all'interno di un metodo (generalmente dichiarato final per

evitare che le sottoclassi ne facciano l'override) e fa si che alcuni passi di tale algoritmo siano implementati da sottoclassi a seconda delle esigenze, senza dover modificare il codice. Nel nostro caso lasciamo scegliere (tramite l'utilizzo dello Scanner di Java) al Client se accettare il pagamento tramite il metodo "acceptPayment": se la risposta è positiva, allora si avvia il processo per pagare gli elementi nel carrello, altrimenti la richiesta viene ignorata. Generalmente i metodi che vengono specificati nelle sottoclassi concrete sono dichiarati abstract, come nel nostro caso lo è il metodo "pay" che viene implementato nei 3 metodi di pagamento concreti che sono "PayPal","CreditCard" e "Bonifico".

Comparare i prezzi

Per effettuare i report del negozio abbiamo dovuto creare una vera e propria strategia, gestita dalla classe "CompareByPrice". Tale classe implementa l'interfaccia Compare<T> di Java, dove il parametro T è sostituito dalla nostra classe astratta ShopOnline. A questo punto la classe "CompareByPrice" eredita il metodo "compare", che una volta sovrascritto ci permette di confrontare secondo il criterio dei prezzi, tutti gli elementi del negozio, sia oggetti singoli che pacchetti. Il metodo compare prende in input due parametri di tipo ShopOnline che rappresentano i due oggetti da comparare. Tali oggetti sono forniti dalla linkedList della classe Shop che contiene tutti gli oggetti del negozio. Usando il metodo *sort* della classe *Collection*, possiamo a questo punto creare un'istanza della classe CompareByPrice e ordinare cosi la lista in ordine crescente: per vedere qual è l'articolo con il prezzo più basso basterà quindi restituire il primo elemento della lista, viceversa restituiremo l'ultimo elemento se vogliamo vedere l'oggetto più costoso. I report inoltre sono disponibili anche sulla classe Cart e Pack.