Relazione progetto

MP A.A. 2017-18

Autori:

- Federico Moncini, 5936828, federico.moncini@stud.unifi.it
- Alessio Danesi, 5967354, alessio.danesi@stud.unifi.it

Entrambi studenti del terzo anno. Parziali superati nel precedente anno.

Self-Bar

Contesto:

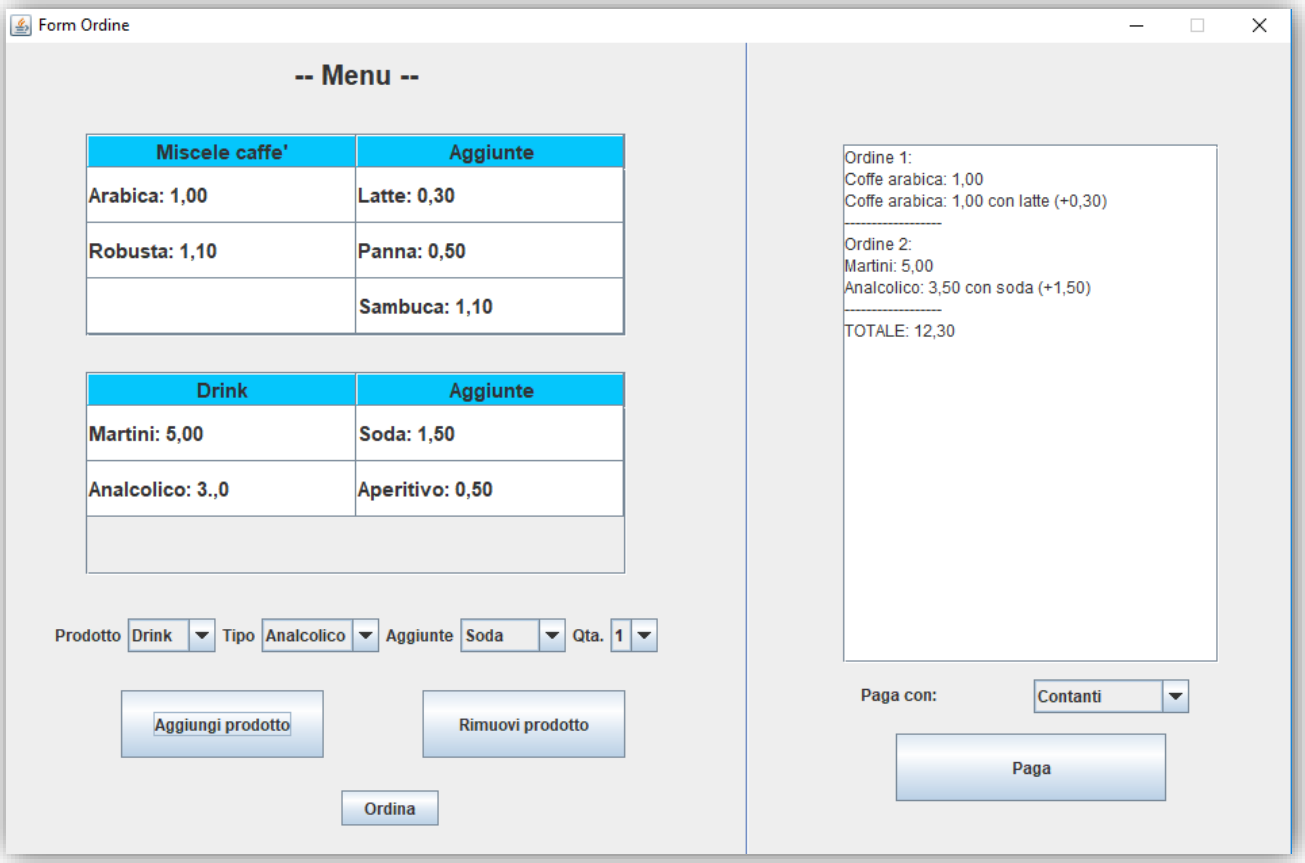
Il problema dell'esercizio self-bar riguarda la gestione di diversi ordini per ogni tavolo di un ipotetico bar con ordinazione automatizzata, attraverso un form che permette di completare gli ordini.
Il cliente quindi ordina una bevanda, ossia un caffè o un cocktail, con tutte le aggiunte desiderate, correlate di un costo aggiuntivo specifico.

Specifiche:

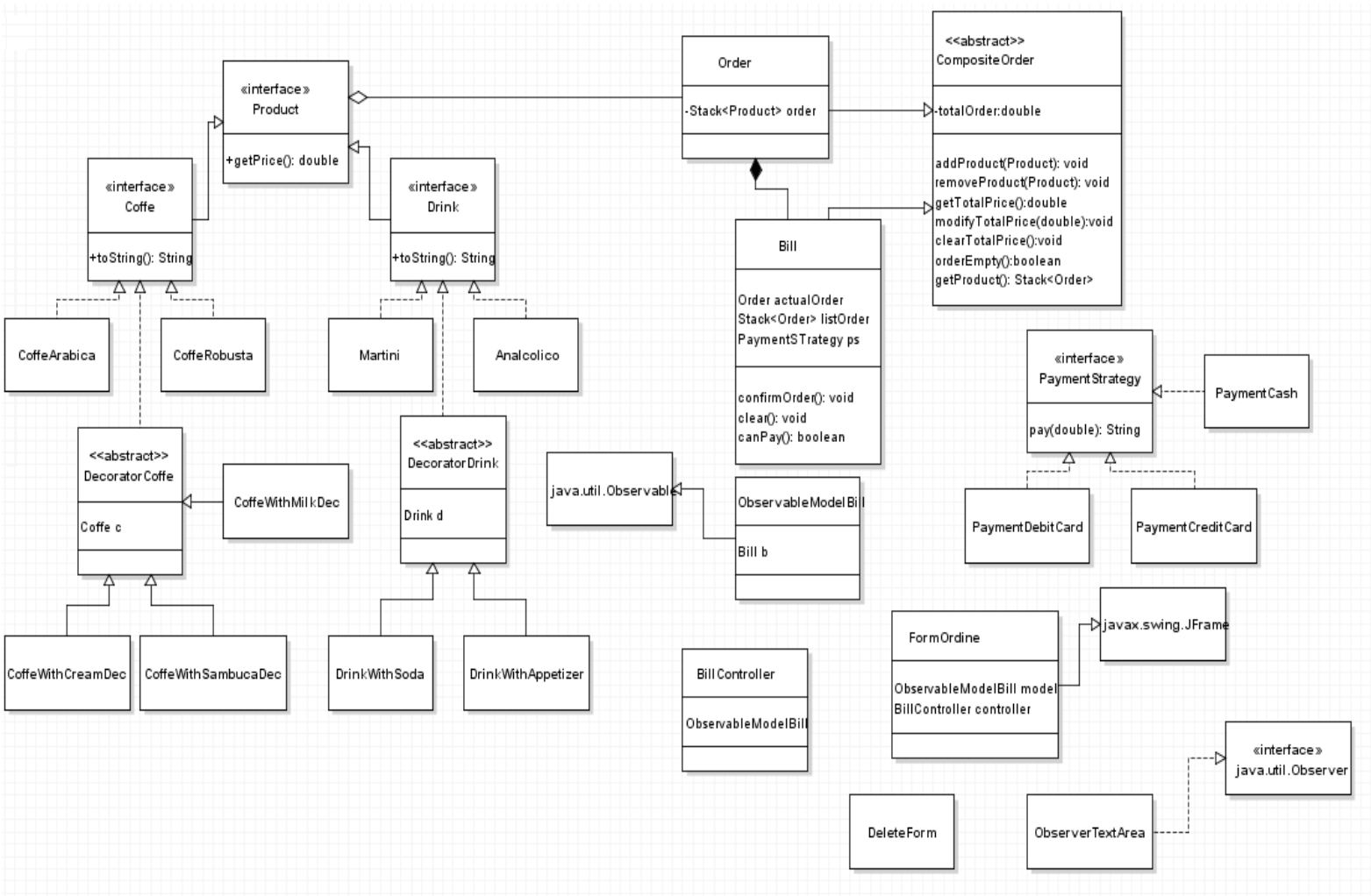
All'utente è data la possibilità di effettuare più ordini, con il solo vincolo che prima di iniziare un nuovo ordine deve completare quello precedente cliccando sull'apposito tasto "Ordina". Si ricorda a tal proposito che un tavolo è un insieme di ordini.
Al proprio ordine attuale è invece possibile aggiungere qualsiasi prodotto desiderato tra quelli disponibili: caffè o drink. Nel nostro caso è possibile scegliere tra due miscele di caffè (arabica e robusta) e due tipi di cocktail di base (Martini e analcolico). Sono presenti anche libere aggiunte per il caffè come latte, panna o sambuca oppure delle aggiunte ai drink come la soda o un aperitivo.

Ogni prodotto aggiunto all'ordine attuale è possibile rimuoverlo, anche compiendo una selezione multipla nel caso in cui si volesse eliminare più prodotti contemporaneamente.
E' possibile verificare i propri ordini, tra cui quello attuale e il costo totale di tutti gli ordini compiuti (aggiornato in tempo reale nel caso di aggiunte o rimozioni) nell'apposita sezione a destra del form.
Quando l'utente decide di pagare può farlo scegliendo uno dei possibili metodi di pagamento: contanti, carta di debito o carta di credito. In base al pagamento scelto vengono applicate commissioni diverse: 0,1% con carta di debito e 0,2% con carta di credito. Dopo aver effettuato il pagamento, il sistema di ordini si resetta ed è possibile crearne di nuovi.

Screenshot dell'applicazione:



UML


«interface»
Product
+getPrice(): double
Order
-Stack<Product> order
<<abstract>>
CompositeOrder
-totalOrder:double
addProduct(Product): void
removeProduct(Product): void
getTotalPrice():double
modifyTotalPrice(double):void
clearTotalPrice():void
orderEmpty():boolean
getProduct(): Stack<Order>
«interface»
Coffe
+toString(): String
«interface»
Drink
+toString(): String
Bill
Order actualOrder
Stack<Order> listOrder
PaymentSTrategy ps
confirmOrder(): void
clear(): void
canPay(): boolean
CoffeArabica
CoffeRobusta
Martini
Analcolico
«interface»
PaymentStrategy
pay(double): String
PaymentCash
PaymentDebitCard
PaymentCreditCard
<<abstract>>
DecoratorCoffe
Coffe c
CoffeWithMilkDec
<<abstract>>
DecoratorDrink
Drink d
java.util.Observable
ObservableModelBill
Bill b
CoffeWithCreamDec
CoffeWithSambucaDec
DrinkWithSoda
DrinkWithAppetizer
BillController
ObservableModelBill
FormOrdine
ObservableModelBill model
BillController controller
javax.swing.JFrame
«interface»
java.util.Observer
DeleteForm
ObserverTextArea

Motivazioni sulle scelte di design implementate

Per sviluppare il nostro progetto abbiamo utilizzato 4 Design Pattern:

Decorator:

E' un pattern strutturale, che viene usato per alterare (aggiungere o specializzare) il comportamento di un oggetto a run-time, avvolgendolo in un oggetto (wrapper) di una classe Decorator. Nel nostro caso è stato utile per poter decorare i prodotti di base con le varie aggiunte scelte dall'utente.

Composite:

Abbiamo usato il Composite per mantenere gli ordini effettuati che a loro volta sono composti da singoli prodotti decorati e non. Attraverso questo pattern è possibile operare su un conto o su un ordine attuale con i soliti metodi. La classe Bill mantiene gli N ordini effettuati al momento, mentre la classe Order mantiene una lista di M prodotti della sessione.

Strategy:

Utilizzato per differenziare le varie tipologie di pagamento: contanti, carta di debito o carta di credito.
La classe Bill mantiene un reference dell'interfaccia PaymentStrategy.

Observer:

Utilizzato per notificare il cambiamento di stato di Bill ad altri componenti.

Principi di progettazione applicati:

- Principio di Liskov (LSP): il quale afferma che in un codice, oggetti di sottotipo devono essere sostituibili a oggetti di supertipo, mantenendo un comportamento appropriato del codice. Un esempio è il seguente: un drink decorato con le sue aggiunte è sempre sostituibile a un drink di base.

-Principio di SINGLE-RESPONSABILITY PRINCIPLE (SRP):
una classe deve avere una sola ragione per essere modificata e alla stessa maniera un metodo deve avere un solo compito (dato che sarà anche più facile testarlo e riutilizzarlo).

-DEPENDENCY INVERSION PRINCIPLE(DIP):
Si elimina il forte accoppiamento fra classi concrete che le renderebbe non riusabili, non facilmente modificabili e difficili da testare. Si usano per cui astrazioni. Un esempio riguarda i Product, dove tutti i tipi di prodotto (Coffee o Drink nel nostro caso) devono estendere la classe Product.

-LAW OF DEMESTRA(LoD):
Ogni unità di codice parla solo a pochi amici diretti, non parla con gli stranieri (conosce solo poche unità). Infatti il metodo di una classe dovrà invocare solo metodi dei suoi parametri, dei suoi componenti e degli oggetti che lui crea direttamente.

Descrizione dei test effettuati:

Sono stati effettuati i singoli test case per le singole classi in modo da testarne la correttezza dei compiti svolti dai singoli metodi. I test case sono serviti, oltre a verificare la correttezza dei metodi, anche la logica applicativa e ci hanno permesso di compiere il refactoring in sicurezza. Abbiamo realizzato i test “Suite” che racchiudono i test di classe correlate tra loro, per esempio: tutti i vari test sulle classi riguardante i pagamenti.