UNIVERSITÀ
DEGLI STUDI
FIRENZE

Progetto di Metodologie di Programmazione

Studente: Ubaldo Puocci

Matricola: 5580871

Email: ubaldo.puocci@stud.unifi.it

Data: xx/12/2015

Informazioni: Consegno il progetto per sostenere l'esame completo

Esercizio scelto e specifiche del software

L'esercizio scelto per la realizzazione del progetto è quello dello shop online. Il programma permette la navigazione, tramite una barra di selezione laterale, del catalogo offerto per i clienti, così da risultare più ordinato e pulito.

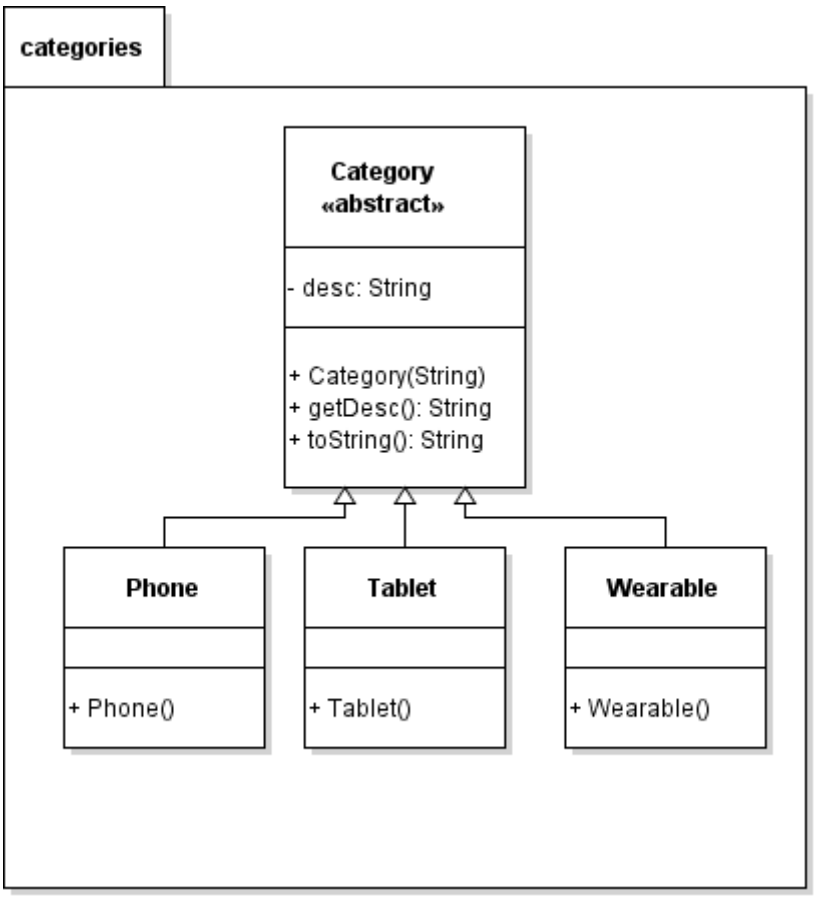
Ogni cliente ha la possibilità di acquistare dei prodotti oppure un pacchetto di prodotti, visualizzandone il prezzo. Di ogni prodotto, compresi i pacchetti, è possibile conoscere i dettagli tramite un pulsante apposito e, nel caso in cui il nostro prodotto sia un pacchetto, sapere inoltre quale sia il prodotto con il prezzo più alto nel pacchetto.

Una volta deciso quali prodotti acquistare, il cliente è invitato a selezionarne la quantità. Selezionata la quantità per ognuno dei prodotti scelti, questi possono essere inseriti nel carrello. Il carrello può essere visionato in ogni momento e può essere modificato rimuovendo dei singoli elementi oppure annullando completamente l'ordine.

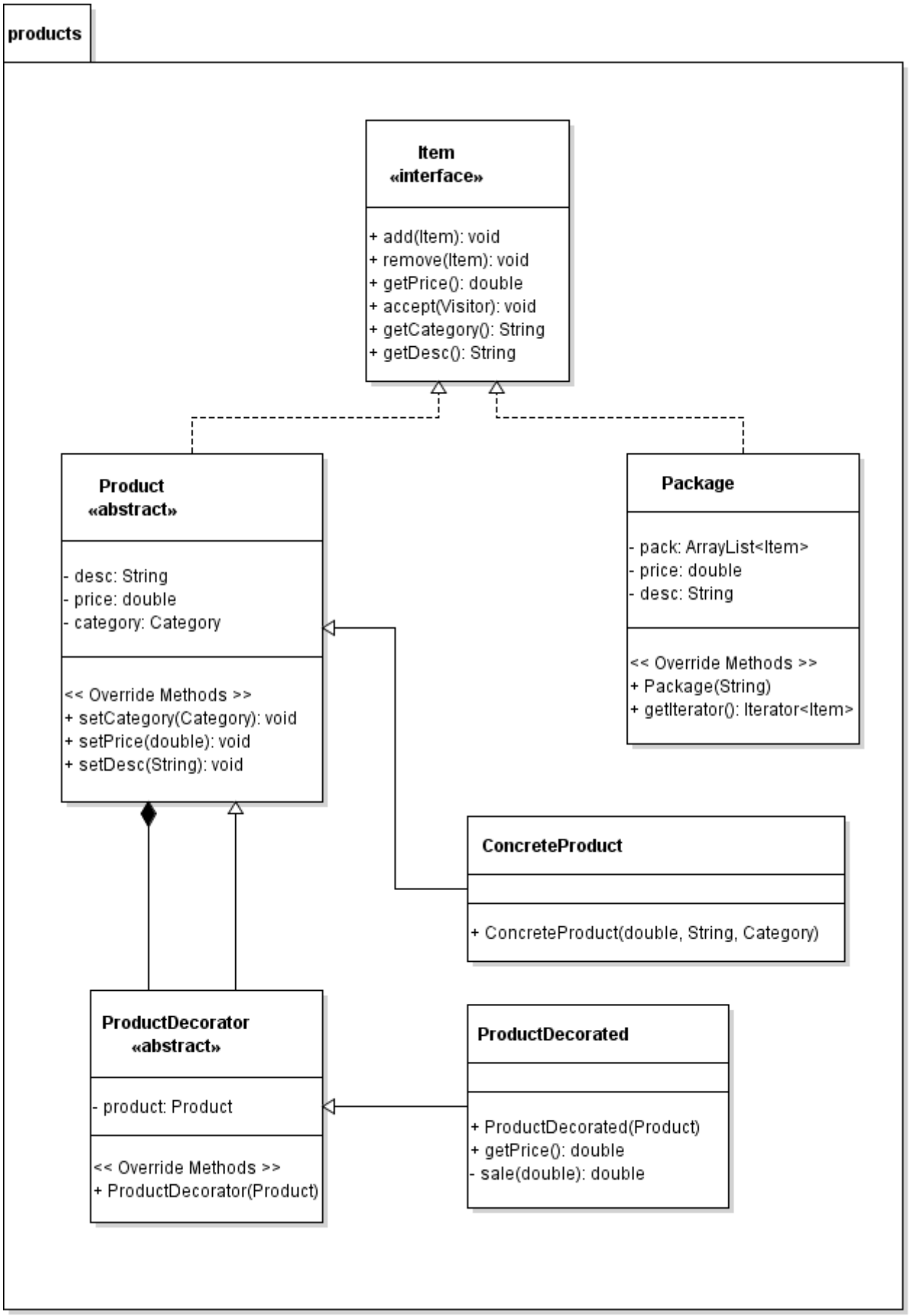
Una volta conclusi gli acquisti, il cliente potrà pagare inserendo gli estremi di una carta di credito. Al termine di questa procedura verrà mostrato a schermo un file di testo con la fattura dell'ordine appena effettuato, dopodiché si potrà tornare agli acquisti.

Diagrammi UML delle classi

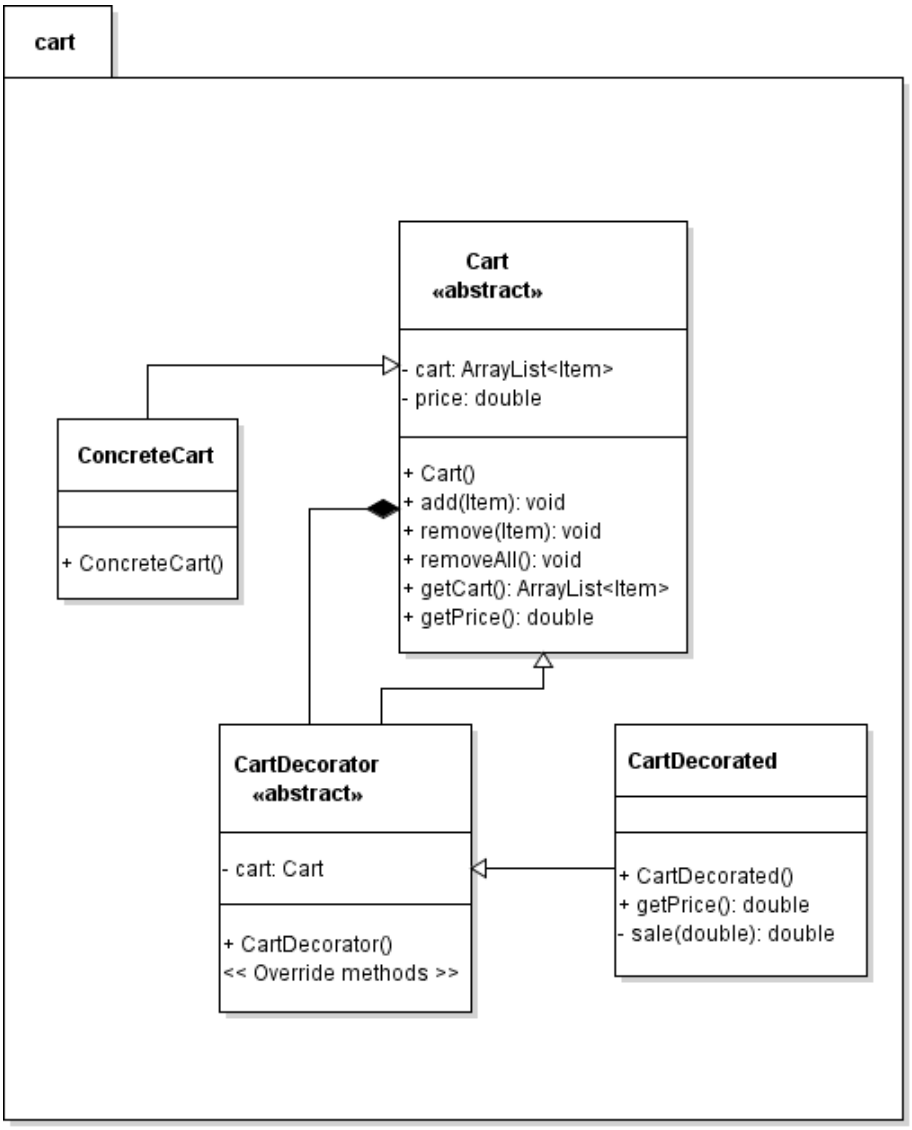
Categories



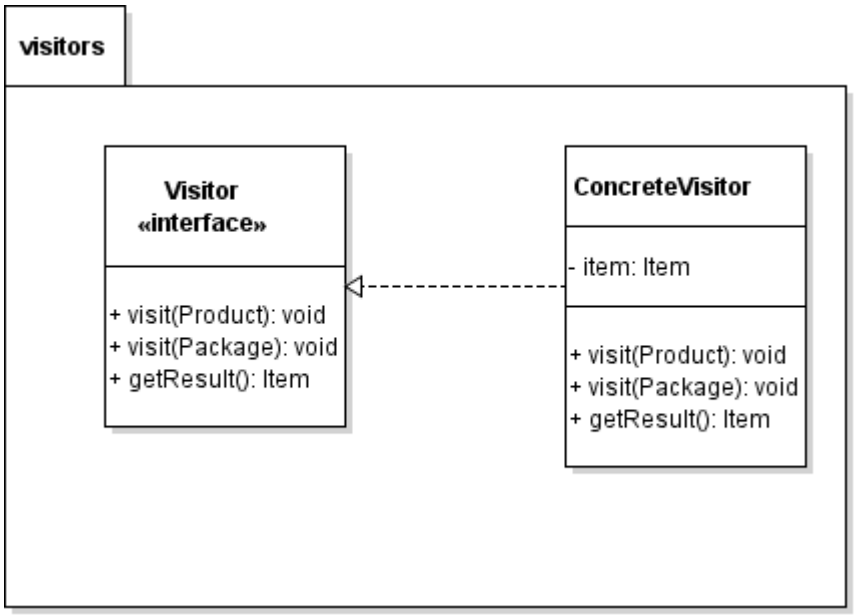
Products


products
Item
«interface»
+ add(Item): void
+ remove(Item): void
+ getPrice(): double
+ accept(Visitor): void
+ getCategory(): String
+ getDesc(): String
Product
«abstract»
- desc: String
- price: double
- category: Category
<< Override Methods >>
+ setCategory(Category): void
+ setPrice(double): void
+ setDesc(String): void
Package
- pack: ArrayList<Item>
- price: double
- desc: String
<< Override Methods >>
+ Package(String)
+ getIterator(): Iterator<Item>
ConcreteProduct
+ ConcreteProduct(double, String, Category)
ProductDecorator
«abstract»
- product: Product
<< Override Methods >>
+ ProductDecorator(Product)
ProductDecorated
+ ProductDecorated(Product)
+ getPrice(): double
- sale(double): double

Cart



Visitors



Design del software

Per scrivere al meglio il codice del mio programma mi sono servito di diversi design pattern: ho utilizzato il Composite, il Decorator ed il Visitor.

Design Pattern Composite

Il pattern Composite è il pattern fondamentale utilizzato nel mio progetto. Esso infatti permette di gestire oggetti di tipi diversi secondo una struttura ad albero, nella quale le foglie sono gli oggetti semplici ed i nodi sono gli oggetti composti, senza distinzione fra essi. Per raggiungere questa implementazione è stata creata un'interfaccia comune sia alle foglie che ai nodi. L'interfaccia definisce tutte le operazioni comuni che possono essere effettuate così che il client, interagendo solo con l'interfaccia, non debba preoccuparsi del tipo degli oggetti che sta utilizzando.

Il pattern Composite nel mio progetto è usato per rappresentare i prodotti singoli ed i pacchetti venduti dal negozio.

Design Pattern Decorator

Il pattern Decorator è il pattern che nel mio progetto mi permette di applicare gli sconti. Il pattern infatti consente di gestire ed aggiungere nuove funzionalità agli oggetti a run-time. Questo viene realizzato creando una classe decoratore che avvolge l'oggetto originale. La classe decoratore ha al suo interno un parametro che è l'oggetto decorato, quindi si può decorare anche un oggetto già decorato. Inoltre, si possono aggiungere anche altre decorazioni in quanto il pattern è facilmente modificabile: basta infatti creare una classe astratta padre di tutte le classi decoratrici che si vogliono creare, così da non dover modificare interamente il codice già esistente.

Il pattern Decorator nel mio progetto è utilizzato per applicare gli sconti ai singoli prodotti oppure a tutto il carrello. Per scontare un prodotto lo si crea direttamente come oggetto decorato, invece lo sconto sul carrello si applica automaticamente quando il costo totale della spesa supera un certo ammontare.

Design Pattern Visitor

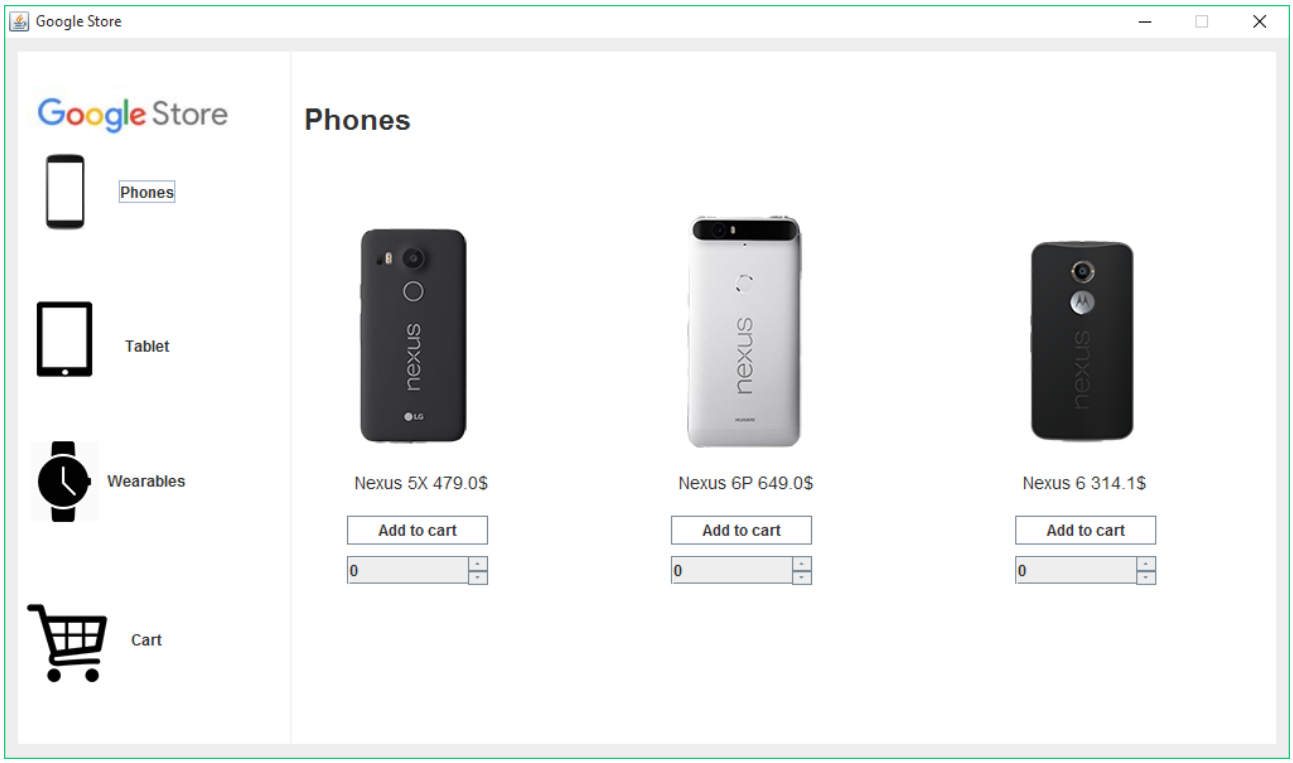
Il pattern Visitor permette di definire nuove operazioni su una gerarchia di oggetti, separandone l'implementazione. Il pattern è utile quando gli oggetti con cui si lavora sono poco soggetti a modifiche, ma vengono continuamente aggiunte nuove funzionalità ed operazioni: in questo modo non devo modificare ogni classe coinvolta nella modifica. Il pattern mette in azione il *Single Responsability Principle*, isolando nella gerarchia di oggetti solo le operazioni di base e delegando all'esterno le altre operazioni. Il pattern utilizza inoltre il double dispatch: infatti la chiamata del metodo dipende sia dal tipo dell'oggetto che esegue `accept(Visitor)` sia dal tipo effettivo di Visitor che esegue `visit()`.

Il pattern Visitor nel mio progetto è utilizzato per ricevere informazioni dettagliate sui prodotti e, nel caso in cui questi siano pacchetti, permette di sapere qual è il prodotto più costoso di tutto il pacchetto.

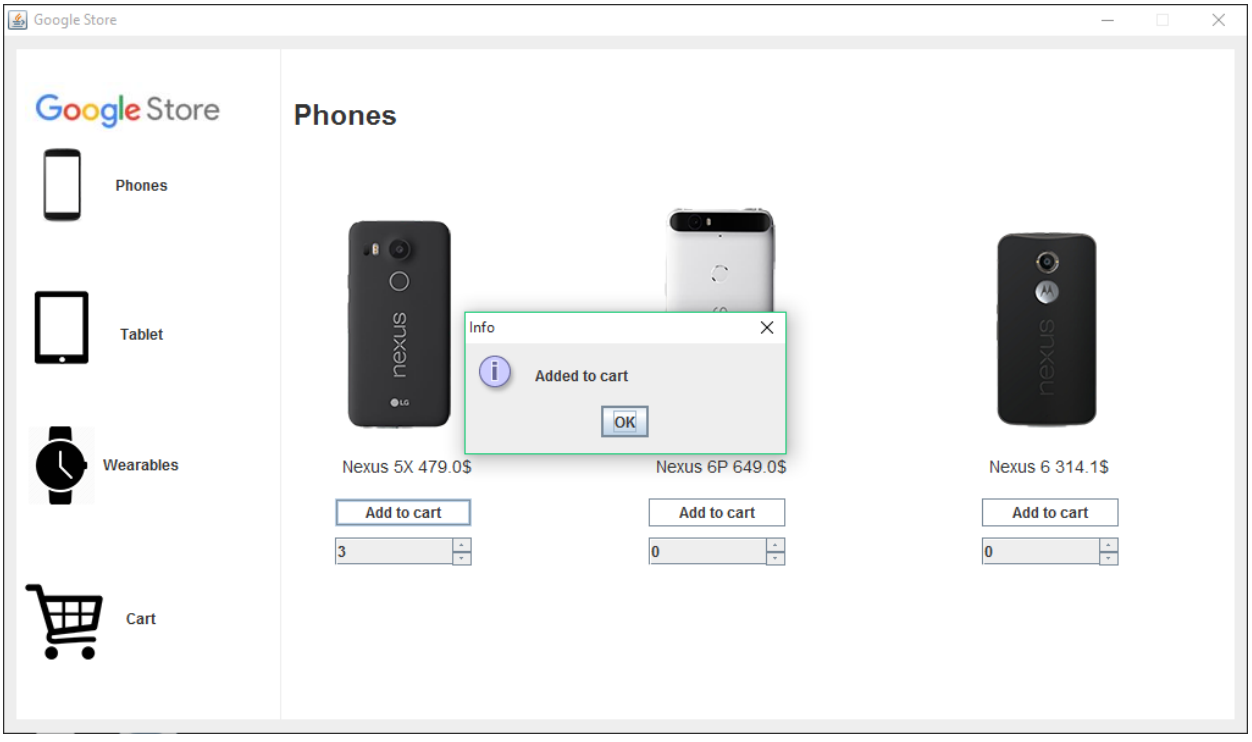
Implementazione e test

Implementazione

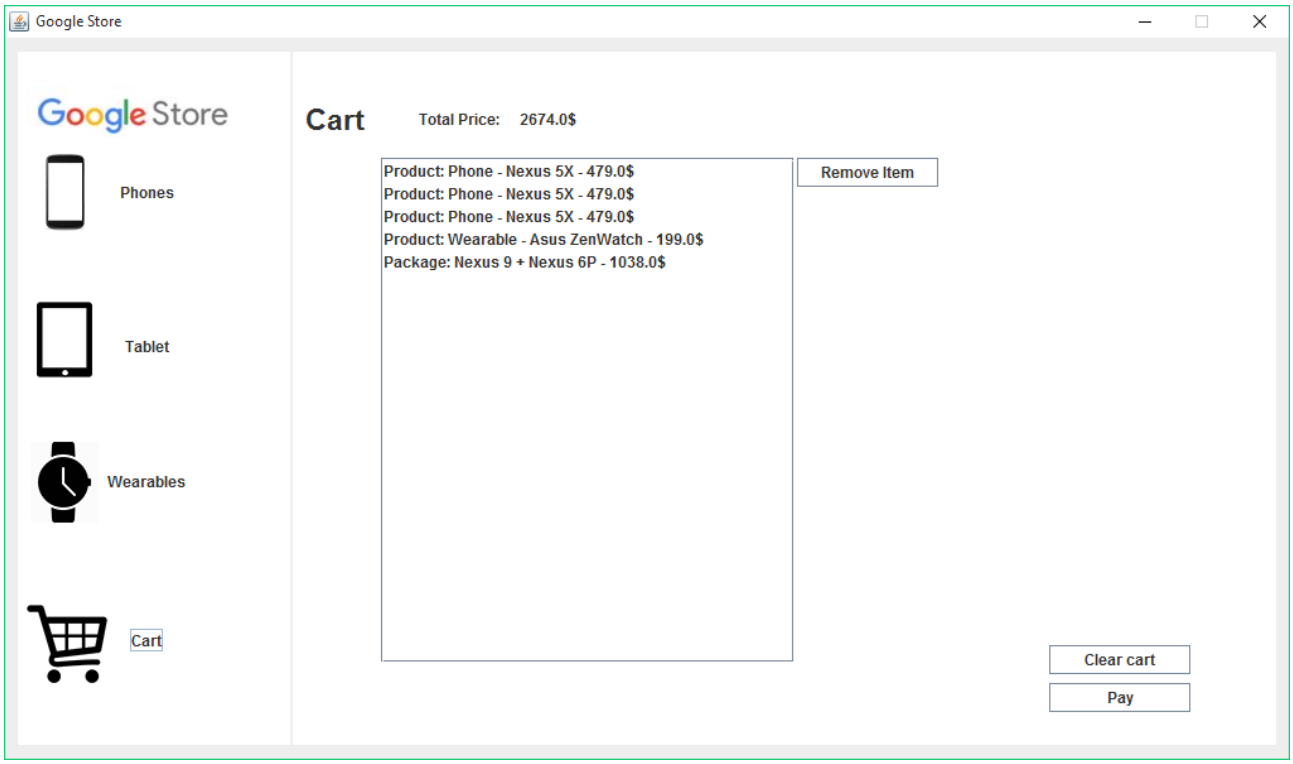
Tutte le classi sono state racchiuse ed implementate in una interfaccia grafica chiara e pulita che fa da tramite fra il cliente e le operazioni a lui nascoste.



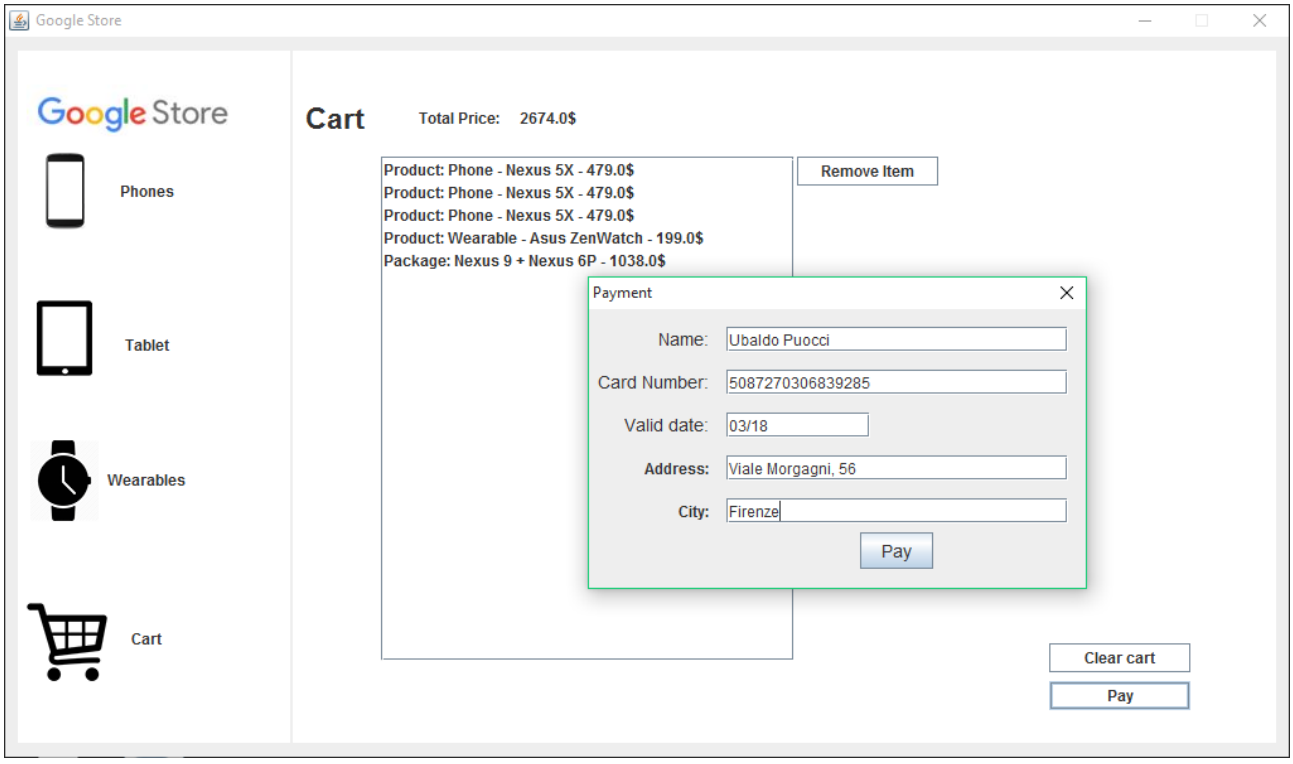
Dopo aver scelto quali prodotti acquistare potremo aggiungerli al carrello.



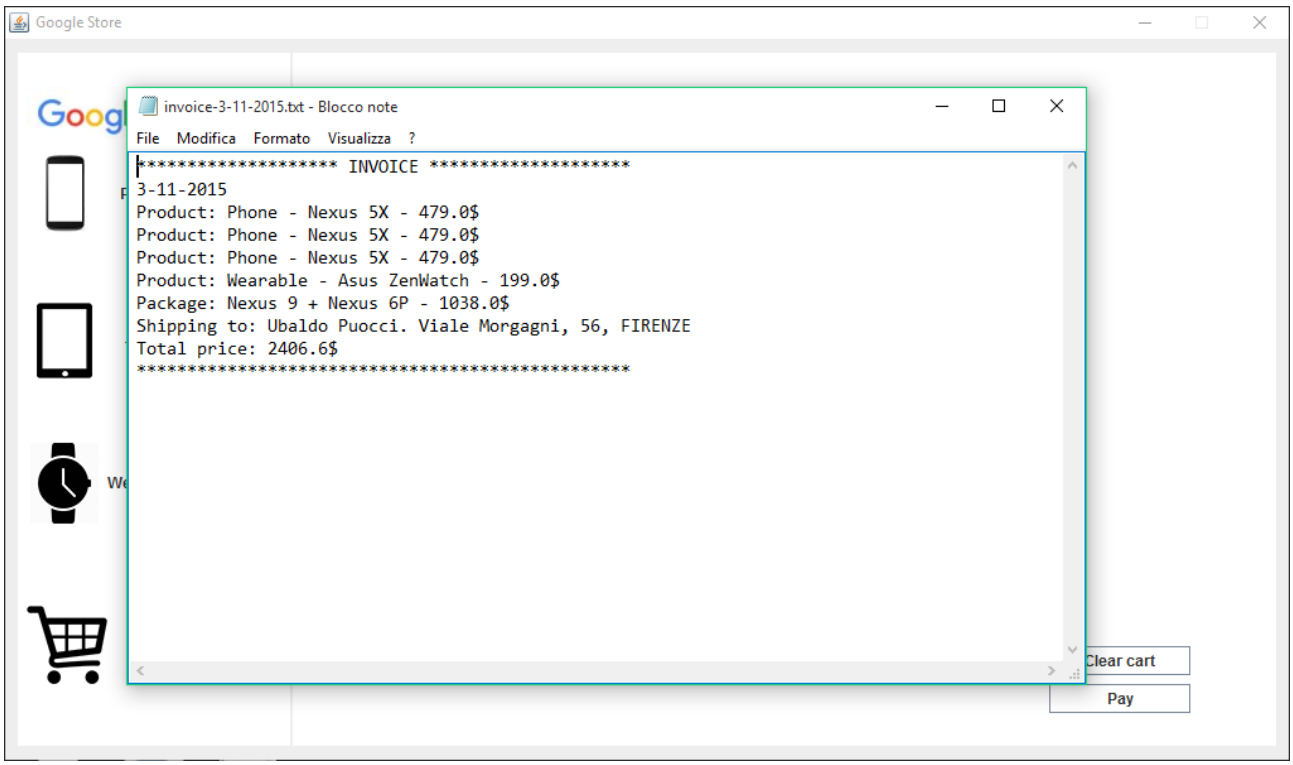
I prodotti scelti vengono visualizzati nel carrello in una lista. Tramite il pulsante apposito è possibile rimuovere gli elementi non più desiderati.



Una volta terminati gli acquisti è possibile inserire gli estremi per il pagamento.



Inseriti i dati, viene visualizzata la fattura.



Test

Tutte le classi concrete del software sono state testate con JUnit e superano tutti i test: vengono testate tutte le categorie di prodotti, i prodotti ed i carrelli normali e quelli decorati e infine, anche i prodotti ed i pacchetti. Sono stati testati anche i metodi che lanciano eccezioni assicurandosi che quest'ultime vengano lanciate per gli opportuni casi.