PROGETTO METODOLOGIE DI PROGRAMMAZIONE

A.A. 2019-2020

INFO SU AUTORE E DATA DI CONSEGNA

Mattia Matucci: mattia.matucci@stud.unifi.it

22/01/2020

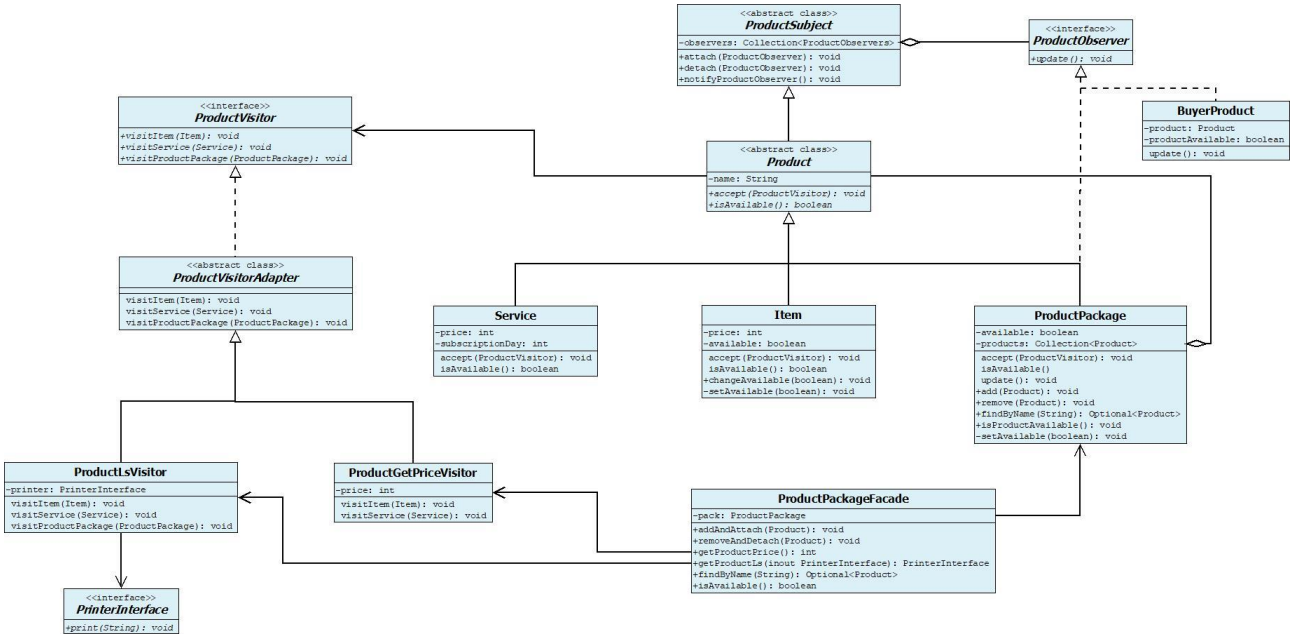
SISTEMA IMPLEMENTATO

Il sistema sviluppato cerca di simulare un software per la gestione di alcune tipologie di prodotti venduti all'interno di uno shop online. I prodotti che si possono trovare in vendita spaziano da articoli a servizi su abbonamento. Sono inoltre presenti dei pacchetti che offrono al loro interno alcuni prodotti, che si trovano anche venduti singolarmente nello shop.

È stata concentrata molta attenzione durante lo sviluppo del sistema alla gestione della disponibilità dei prodotti, cercando di creare una differenza tra prodotti sempre disponibili, quali i servizi su abbonamento, e prodotti con disponibilità limitata, come gli articoli e i pacchetti di prodotti.

Intorno a ciò sono state sviluppate anche altre funzionalità utili, come un sistema per ottenere una breve descrizione dei prodotti. È stata poi realizzata una piccola interfaccia con cui è possibile gestire i pacchetti più semplicemente.

DIAGRAMMA UML



PATTERN UTILIZZATI

Il primo problema ad essere stato affrontato durante lo sviluppo di questo software è stato la rappresentazione, in modo uniforme, di tutti i prodotti all'interno dello shop. L'utilizzo del pattern Composite è sembrato quindi ottimo, infatti la sua applicazione permette di gestire con uniformità sia oggetti composti, in questo contesto i pacchetti, sia oggetti semplici, per esempio gli articoli. È così possibile nascondere al client la reale natura del prodotto con cui sta interagendo.

Una volta realizzata la struttura dei prodotti dello shop, l'intenzione è stata quella di concentrare molte risorse nell'implementazione della gestione di disponibilità dei prodotti: cioè la notifica dello stato di disponibilità, in quel momento, del prodotto a cui il client è interessato. Ciò ha provocato un rinvio, ad un secondo momento nello sviluppo del sistema, delle altre funzionalità delle entità. Non volendo ogni volta dover riaprire la struttura dei prodotti, è sembrata un'ottima idea l'utilizzo del pattern Visitor, che oltre a permettere di aggiungere funzionalità in un secondo momento mantiene anche le classi del Composite leggere.

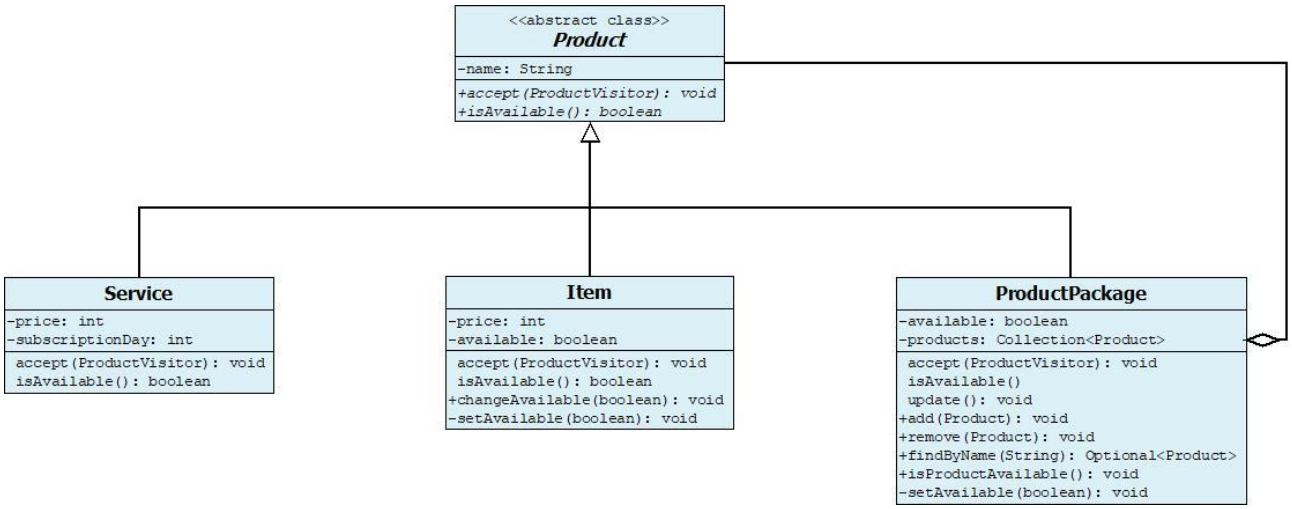
In prospettiva futura è stato pensato che potrebbe essere utile poter ampliare la gamma dei prodotti offerti nello shop, tuttavia ciò comporterebbe la necessità di rompere l'interfaccia del Visitor esistente e tutte le classi che la implementano, dunque applicare il pattern Adapter per fornire un'implementazione di default a tutti i metodi è sembrata una buona idea. Inoltre questa soluzione permette nei Visitor concreti di ridefinire solo i metodi d'interesse per quella classe, lasciando gli altri con l'implementazione di default fornita dalla classe Adapter.

Dopo aver predisposto il sistema al Visitor e averlo implementato, l'attenzione è passata allo sviluppo della gestione delle notifiche della disponibilità dei prodotti. Dovendo notificare qualcosa la scelta più ovvia da adottare è stata l'impiego del pattern Observer. La notifica deve essere inviata ogni qual volta che lo stato di disponibilità varia, per far ciò nei prodotti semplici a disponibilità limitata è stata introdotta una variabile booleana mentre per i pacchetti è stata introdotta sempre la medesima variabile solo che questa risulta essere vera se e solo se ogni componente del pacchetto è disponibile, dunque il pacchetto è sia un osservatore di tutte le sue componenti che soggetto in quanto estende la classe Product. I prodotti sono dunque tutti possibili soggetti di osservazione da parte di client interessati a loro.

Una volta completata l'implementazione del pattern Observer, è sembrata una buona idea applicare il pattern Facade per semplificare il lavoro del client interessato a gestire i pacchetti. È stata dunque realizzata una classe in cui il compito di aggiungere e seguire un prodotto è realizzato all'interno di un unico metodo, così da garantire un buon funzionamento del codice.

DETTAGLI DELL'IMPLEMENTAZIONE

PATTERN COMPOSITE:



shop.composite.Item;

La classe Item rappresenta una possibile entità con una disponibilità limitata. All'interno della classe è presente il metodo privato setAvailable (boolean) che può essere chiamato esclusivamente mediante il metodo pubblico changeAvailable (boolean) che prima di chiamare il setter effettua un controllo sul valore passato come parametro e il campo available per vedere se sono diversi. Se questo controllo passa, allora dopo la chiamata del setter, viene invocato il metodo notyfyProductObserver () della super classe ProductSubject per avvertire gli observer di quell'item.

shop.composite.Service;

La classe Service rappresenta invece un'entità sempre disponibile, dunque è stato deciso di implementare il metodo della super classe isAvailable (), facendogli restituire il valore true.

shop.composite.ProductPackage;

La classe ProductPackage vuole rappresentare, all'interno dei prodotti dello shop, i pacchetti che contengono un insieme di prodotti. È stato pensato che un possibile client interessato ad usare un prodotto non possa non essere interessato alla vera natura di questo. Dunque è parsa una buona idea implementare il pattern Composite nella versione type-safe, così che la gestione dei figli sia presente esclusivamente all'interno di questa classe. Sono dunque stati forniti i metodi di aggiunta e rimozione di prodotto e la ricerca per nome di un prodotto all'interno del pacchetto.

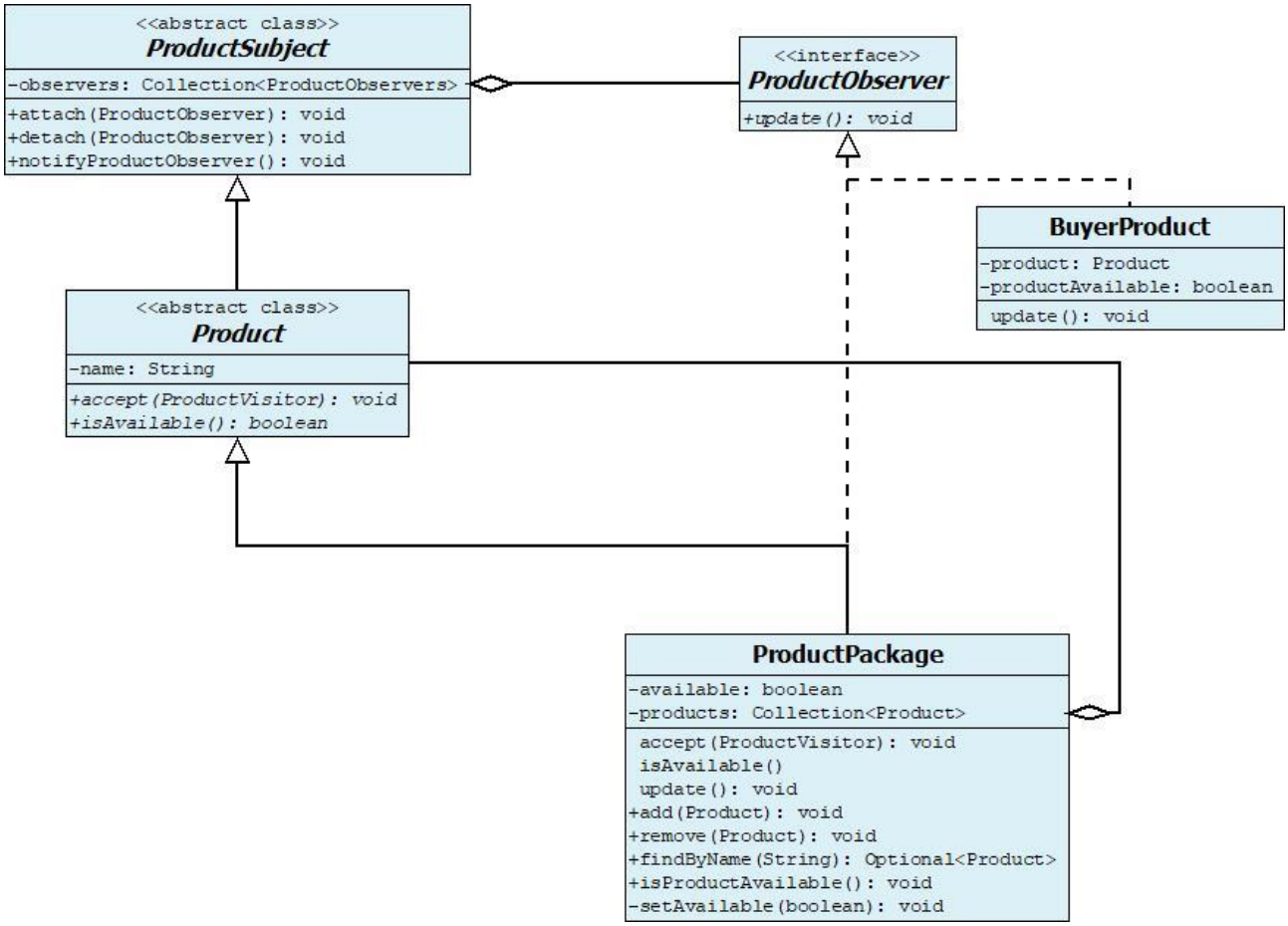
Altri metodi sono presenti dentro questa classe ma la trattazione è rimandata in seguito durante la spiegazione della classe nel pattern Observer.

shop.composite.Product;

La classe astratta Product ha il compito di nascondere la reale natura delle entità ai client che hanno bisogno esclusivamente di un prodotto. La classe possiede un campo name utile per distinguere e ricercare un prodotto, come per esempio nel metodo findByName () della sottoclasse ProductPackage, e contiene la firma dei metodi:

- accept (ProductVisitor) che serve per predisporre le classi dell'entità al Visitor.
- isAvailable () utile per la gestione della disponibilità del prodotto.

PATTERN OBSERVER:



shop.observer.ProductSubject;

La classe ProductSubject è l'interfaccia che accomuna tutti i prodotti che devono essere osservati. Contiene i metodi per gestire la collezione degli osservatori, inoltre si occupa di notificarli, mediante il metodo notifyProductObserver () che deve essere chiamato dai metodi che nell'entità permettono di modificare lo stato della propria disponibilità.

shop.observer.ProductObserver;

È l'interfaccia che contiene il metodo update (). Tale interfaccia deve essere implementata, ridefinendo il metodo update (), da tutte le classi che hanno bisogno di osservare lo stato di disponibilità dei prodotti, così che il subject possa notificarli in caso di un cambiamento di un proprio stato.

shop.composite.Product;

Riprendo qua, alcune considerazioni fatte in precedenza per questa classe.

La classe rappresenta in modo uniforme tutti i prodotti che possono essere presenti all'interno di uno shop. In questo caso per permettere a tutti gli osservatori di seguire lo stato di un prodotto indistintamente dalla reale natura dell'oggetto è presente il metodo isAvailable (), che permette di ricevere la sua disponibilità del prodotto a cui il client è interessato.

shop.observer.BuyerProduct;

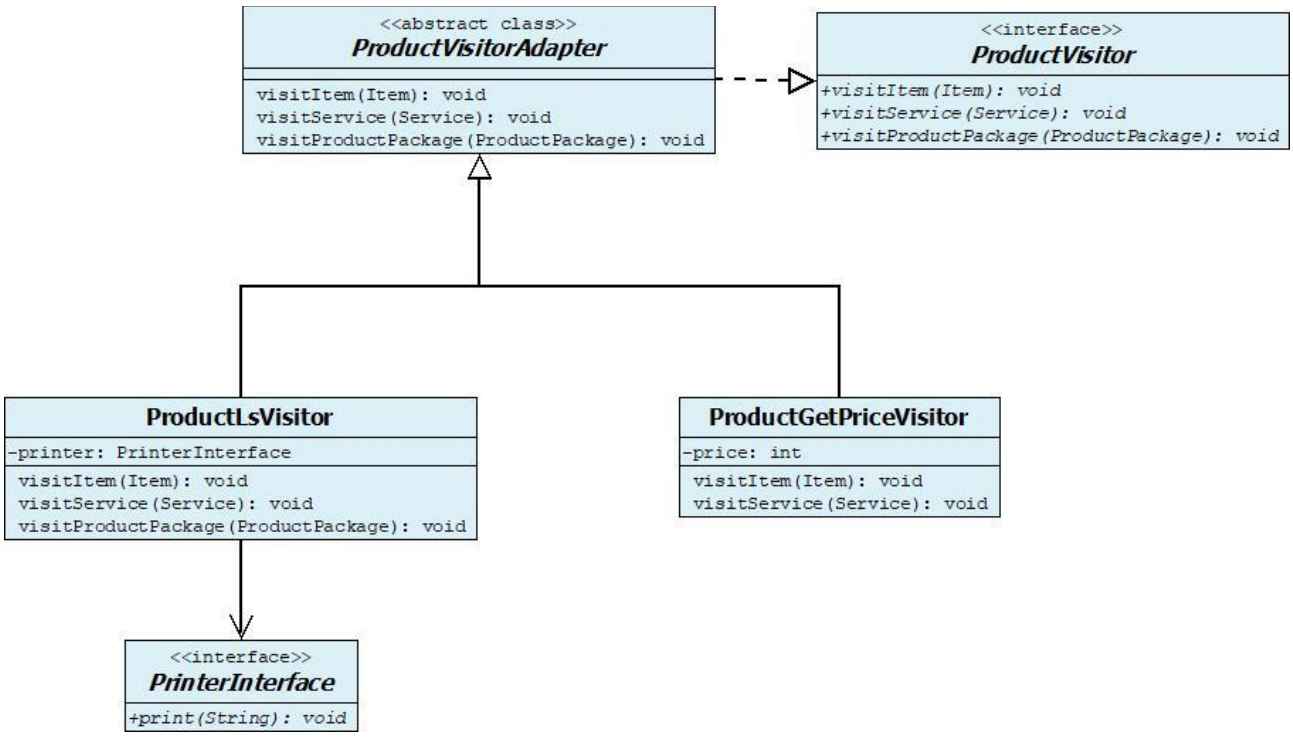
La classe rappresenta un possibile cliente del negozio interessato a seguire lo stato di un prodotto, a questo scopo tiene una variabile productAvailable, inizializzata con la disponibilità del prodotto nel costruttore, riassegnabile dal metodo update () ogni volta che viene ricevuta una notifica dal prodotto seguito.

shop.composite.ProductPackage;

La classe in questione è dunque sia un subject**,** in quanto estende la classe Product, sia un observer, poiché il pacchetto è interessato a conoscere la disponibilità di ogni sua componente, quindi implementa l'interfaccia ProductObserver. I metodi che possono lanciare una notifica sono l'update () chiamato da una sua componente, l'add (Product) di un nuovo prodotto e la remove (Product) di un prodotto presente nel pacchetto

Questi tre i metodi delegano l'operazione di notifica al metodo privato isProductAvailable () che si occupa di fare una visita su tutte le componenti del pacchetto, ritornando true se e solo se tutte le componenti sono in quel momento disponibili. A questo punto se la visita ha assunto un valore diverso rispetto al valore del campo available, viene aggiornato il campo e successivamente è eseguita la notifica a tutti gli observer.

PATTERN VISITOR e ADAPTER:



shop.visitorAndAdapter.ProductVisitor

L'interfaccia ProductVisitor viene usata da tutti coloro che hanno bisogno di accedere alle funzionalità delegate ai visitor concreti dei prodotti, poiché aggiunte in un secondo momento al codice. Contiene un metodo che effettua la visita per ogni tipologia di prodotto concreto rappresentato nel Composite, sarà dunque necessario modificare questa interfaccia se verrà modificata la struttura del Composite.

shop.visitorAndAdapter.ProductVisitorAdapter

Questa classe dovrebbe servire come intermediario tra l'interfaccia del Visitor, aprendola così a possibili modifiche che non dovrebbero più rompere tutti i client ma solo questa classe astratta, e i visitor concreti che dovrebbero estendere questa classe piuttosto che implementare direttamente l'interfaccia, così facendo è possibile permette ai visitor concreti di ridefinire esclusivamente i metodi che sono utili a svolgere la funzione che ha reso necessario creare quella classe.

La classe fornisce un'implementazione vuota alle foglie del Composite mentre per i composti, usando un iteratore, delega a tutte le componenti l'operazione da effettuare.

shop.visitorAndAdapter.ProductGetPriceVisitor

Grazie a questo visitor concreto è possibile restituire il prezzo del prodotto selezionato, accumulando in una variabile price l'importo di ogni componente. Mentre per gli oggetti semplici si tratterà di restituire il valore della variabile interna alla classe delle entità, per i pacchetti dovrà essere restituita la somma dei prezzi di tutti gli oggetti che compongo il pacchetto. Non è stato dunque necessario ridefinire il metodo visitProductPackage (ProductPackage) in quanto l'operazione di incremento del prezzo dipende esclusivamente dagli oggetti foglia del Composite.

shop.visitorAndAdapter.ProductLsVisitor

Questo visitor concreto permette di utilizzare un oggetto di classe PrinterInterface, che dovrà essere passato come parametro nel costruttore del ProductLsVisitor, per ottenere la descrizione del prodotto.

Ogni entità deve possedere una propria descrizione, è stato dunque necessario ridefinire tutti i metodi dell'Adapter, nel caso particolare del ProductPackage dopo aver scritto la propria descrizione, il metodo richiama l'implementazione della super classe per delegare ai suoi composti.

shop.utilies.PrinterInterface

È l'interfaccia usata dal ProductLsVisitor per produrre la descrizione del prodotto selezionato, qualunque client che abbia intenzione di usare quel visitor concreto deve necessariamente implementarla dal momento che nel codice non lo è mai stato fatto.

PATTERN FACADE:

shop.composite.ProductPackageFacade;

Questa classe vuole offrire ai client la possibilità di gestire più semplicemente i pacchetti, soprattutto adesso che hanno assunto sia il ruolo di entità da osservare, poiché la classe estende Product, che quello di osservatore, in quanto la classe implementa ProductObserver. I metodi di questa classe sono:

- addAndAttach (Product) permette di aggiungere e seguire un prodotto con un'unica chiamata.
- removeAndDetach (Product) permette di rimuove e smettere di seguire un prodotto atomicamente.

Questi due metodi sono stati pensati per prevenire possibili dimenticanze.

- finByName (String) restituisce l'oggetto ricercato se è uno delle componenti del pacchetto
- isAvailable () restituisce la disponibilità del pacchetto
- getProductPrice () restituisce il prezzo del pacchetto
- getProductLs (PrinterInterface) restituisce il parametro passato in ingresso al metodo una volta effettuata la visita

TESTS EFFETUATI

Durante lo sviluppo del codice del progetto, in una source folder separata, sono stati scritti i test per controllare il corretto funzionamento del codice. I test sono stati scritti attraverso l'uso di JUnit4 e in aggiunta la libreria di asserzioni AssertJ. Data la presenza di alcuni metodi che fanno uso di un oggetto di un'interfaccia mai implementata nel codice, nel folder dei test è presente un'implementazione fittizia di tale interfaccia così da poter testare anche tali metodi.

La classe in questione è la MockPrinterInterface, che usa un'istanza di StringBuilder, per salvare tutte le stringe passate al metodo print (String), in aggiunta a questo metodo è stato ridefinito il metodo toString () per restituire la visita effettuata.

Durante il periodo di scrittura dei test, è stato cercato di osservare il principio secondo cui in una classe di test non dovrebbero essere usati i metodi testati per testare altri metodi dentro la solita classe.

Esempio:

Per testare la classe ProductPackage è stato necessario testare sia l'add (Product) che il remove (Product). C'è stato dunque bisogno, nella classe test, di avere accesso alla collezione per poter inserire direttamente al suo interno senza dover chiamare il metodo add () di ProductPackage, durante il test del remove (). È stato così aggiunto nella classe da testate il metodo getProducts () con visibilità private-package per aver accesso al metodo, ma nascondere il metodo fuori dal pacchetto.

A questo scopo è stata prestata molta attenzione ad inserire le classi e le rispettive classi test all'interno dei soliti pacchetti, ma in folder distinti, facendo così accedere i test ai metodi con visibilità private-package pensati per semplificare la scrittura dei test e rispettare il principio prima presentato.

Sempre per ottenere test facilmente comprensibili alcuni metodi, come i getter e i setter, sono stati esulati dal rispettare tale principio.