RELAZIONE PROGETTO ALBERTO BILIOTTI

Numero di matricola: 7026939

Email: alberto.biliotti@stud.unifi.it

Descrizione funzionalità

Lo scopo di questo progetto è quello di implementare un gestore di spedizioni per un negozio di prodotti elettronici: esso dovrà gestire una collezione potenzialmente infinita di scatole che al loro interno possono contenere: apparecchi elettronici, materiale da imballaggio anti caduta oppure altre scatole che a loro volta posso contenere le stesse cose potenzialmente all'infinito (meccanismo che ho implementato grazie al pattern Visitor).

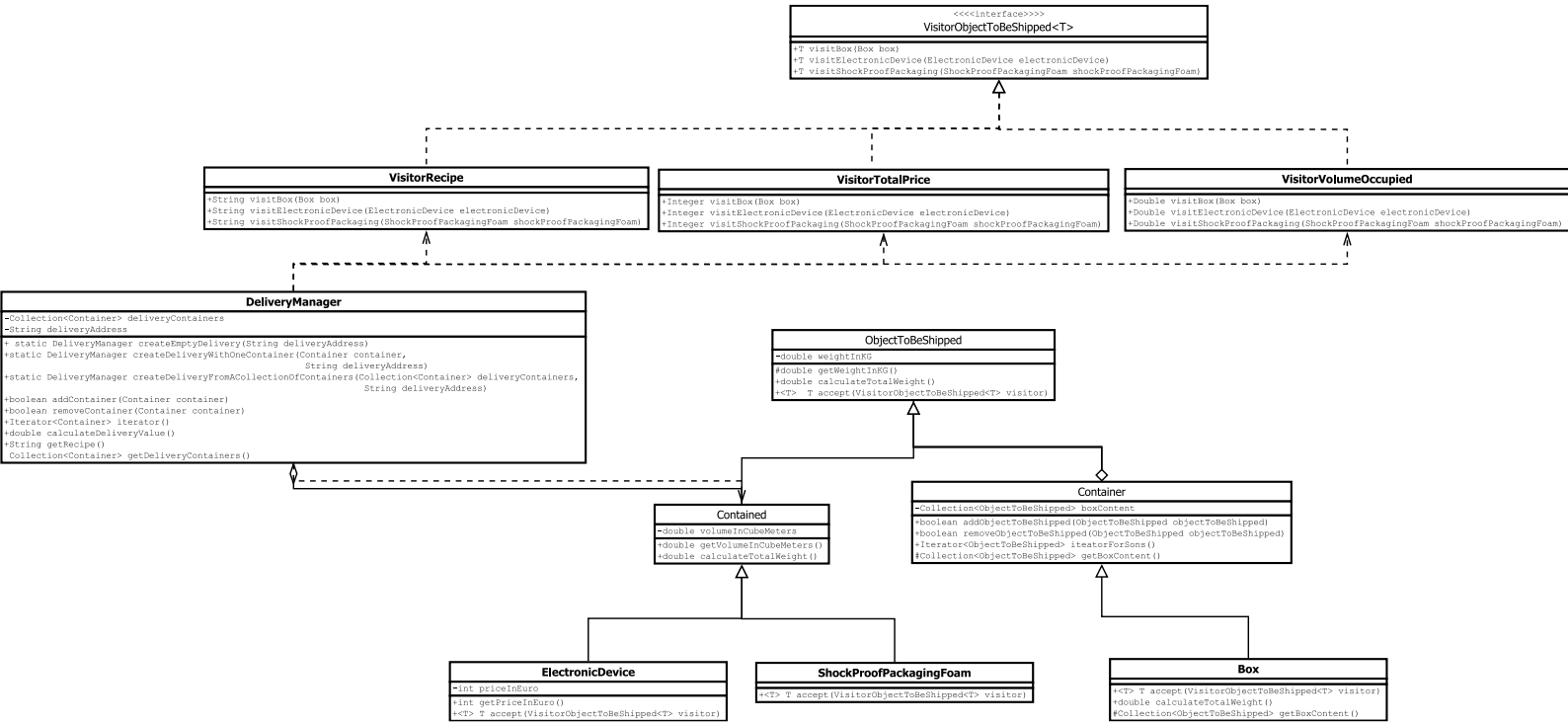
Il gestore dovrà essere in grado di: aggiungere o rimuovere scatole a piacimento (controllando però di non aggiungere null), restituire un iteratore contente le scatole presenti nella spedizione (senza contare le scatole contenute dentro altre scatole), calcolare il peso totale della spedizione, il volume occupato, il valore totale della merce (quindi considerando le scatole e l'imballaggio senza valore) e dovrà inoltre emettere una ricevuta contente tutte le caratteristiche degli oggetti contenuti nella spedizione.

Il gestore non potrà essere istanziato direttamente (ovvero con attraverso new) ma dovrà essere istanziato attraverso uno dei tre diversi "static factory method" che permettono di istanziare o un gestore vuoto, o un gestore a partire da una sola scatola oppure da una collezione di esse controllando però sempre di non passare una scatola od un valore null (andrà sempre passato comunque l'indirizzo del destinatario della spedizione).

Le mie entità semplici (le scatole, i prodotti elettronici e i materiali da imballaggio) possono essere "visitati" grazie al pattern Visitor che mi permette di aggiungere nuove funzionalità senza modificarne il codice ed è stato usati per tre scopi: restituire la ricevuta di un singolo oggetto da spedire, calcolarne il prezzo ed il volume occupato da ciascuno di essi.

Inoltre le mie entità semplici hanno un metodo che restituisce il peso di ciascuna entità considerando però come peso di una scatola come la somma del peso di tutti gli oggetti presenti al suo interno più il peso della scatola stessa.

Diagramma delle classi UML



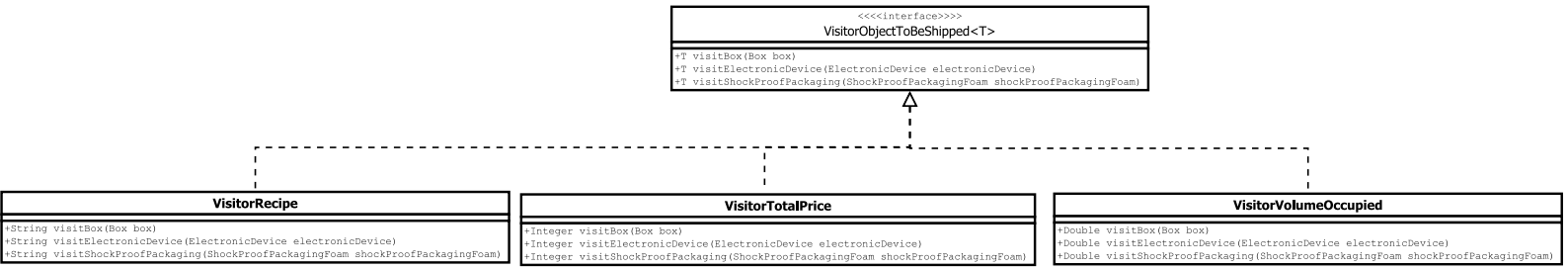
Realizzato con il Software DIA

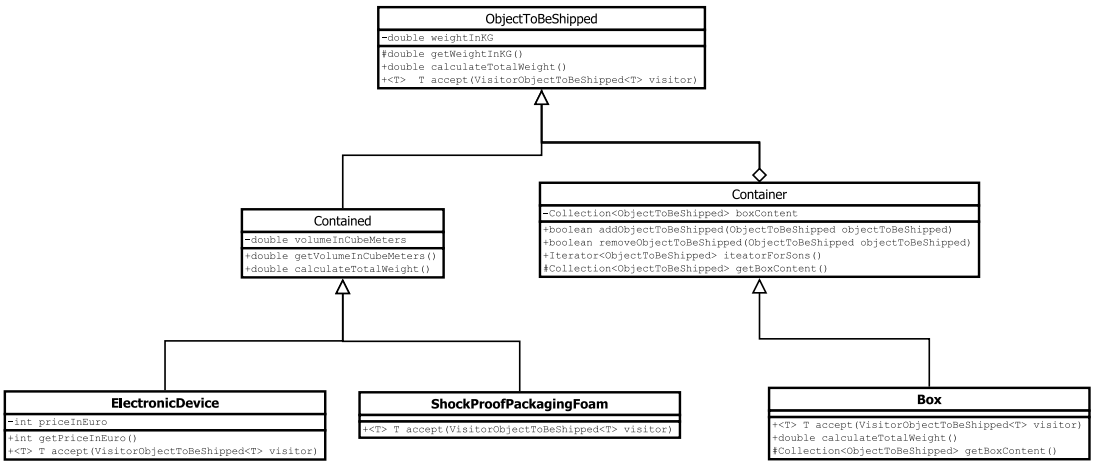
Scelte di Design e strategie per testare

Come accennato in precedenza i pattern che ho utilizzato nel progetto sono:

Il **Visitor** per gli oggetti da spedire (apparecchi elettronici, materiale da imballaggio anti caduta o scatole) che mi permette di aggiungere in futuro altre operazioni su questi oggetti senza dover modificare il codice ma solo estenderlo(rispettando così l' "Open Close Principle" che raccomanda di preferire l'estensione del codice alla sua modifica diretta), in particolare utilizzo la versione "generica" del Visitor, ovvero non "impongo" un tipo di ritorno specifico ai metodi della classe Visitor ma uso i tipi generici per lasciare libertà di scelta del tipo di ritorno al Client: questo permetterà di avere un'unica interfaccia per tutti i tipi di ritorno possibili invece di dover creare una nuova interfaccia (e di conseguenza anche un nuovo metodo accept)per ogni tipo di ritorno necessario. Purtroppo l'utilizzo del pattern Visitor rompe obbligatoriamente l'incapsulamento delle variabili di istanza delle mie entità semplici (pur riuscendo a mantenere l'incapsulamento della collezione contenente il contenuto di una scatola grazie al metodo che restituisce un iteratore) ma ho ritenuto che i vantaggi offerti dal Visitor fossero superiori agli svantaggi generati dalla rottura dell'incapsulamento.

Diagramma UML dei Visitor

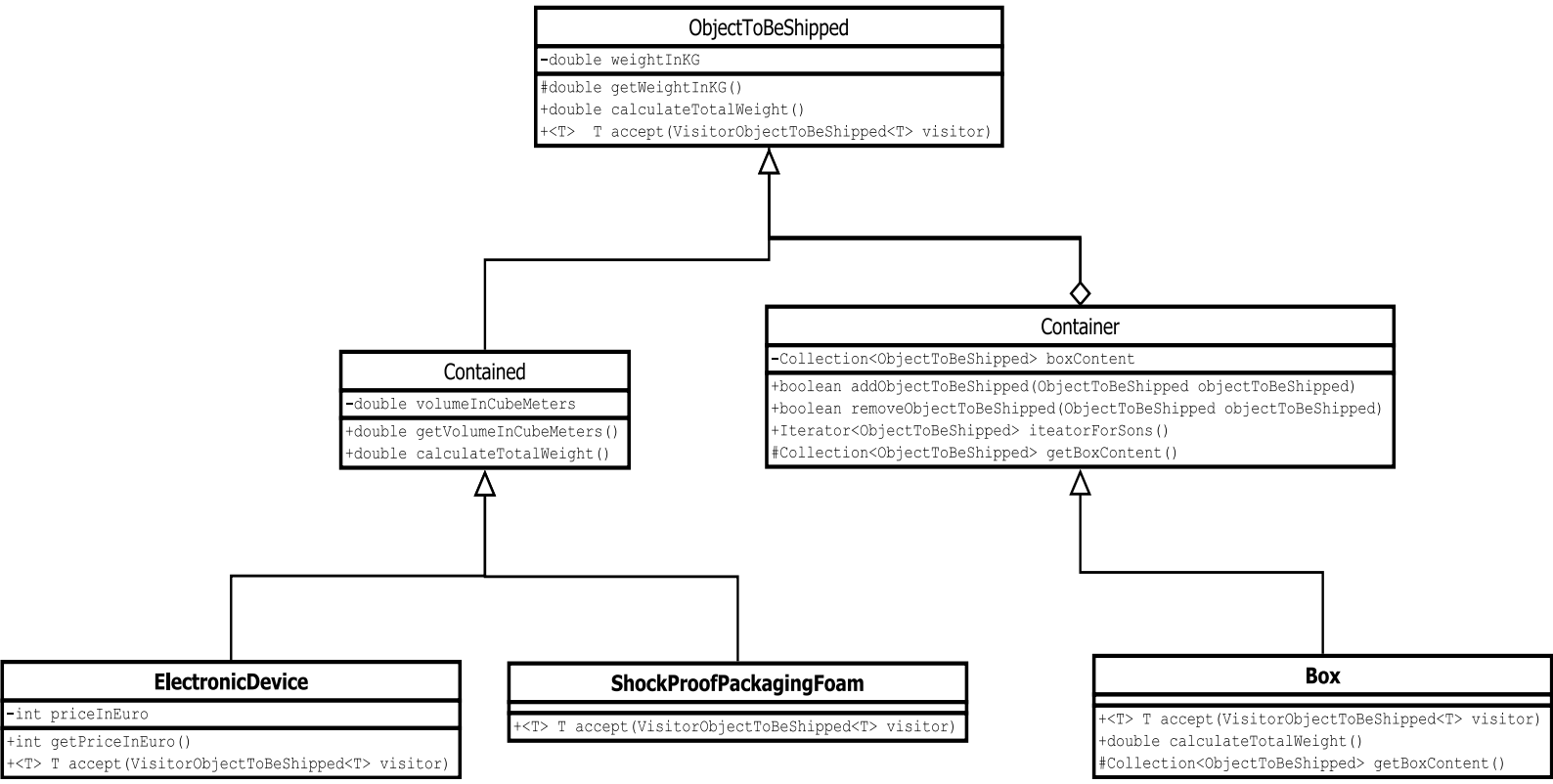




- Il **Composite** che ho sfruttato per poter dare origine ad una struttura "ad albero" che risulta molto utile nella rappresentazione del contenuto di una scatola che, nel mio caso, possono essere: apparecchi elettronici, materiali da imballaggio anti caduta oppure, ricorsivamente, altre scatole che a loro volta potranno contenere gli stessi elementi e cosi fino, potenzialmente, all'infinito. I vantaggi derivati dall'utilizzo di questo pattern sono, oltre alla già citata struttura ad albero nella quale le foglie sono rappresentate sia dagli apparecchi elettronici che dai materiali da imballaggio mentre i nodi da scatole: il fatto di poter implementare il metodo per il calcolo del peso totale di una scatola con il suo contenuto semplicemente sommando al proprio peso il risultato dell'applicazione dello stesso metodo o su una foglia o ricorsivamente su un'altra scatola sfruttando così il meccanismo di delega. La variante di Composite da me scelta è quella denominata "Design for type safety" in cui le operazioni sui figli non fanno parte della classe astratta per gli oggetti da spedire ma fanno parte della mia classe "Composite" ovvero della classe che rappresenta

il contenitore della spedizione evitando così che anche le mie foglie fossero obbligate a implementare questi metodi.

Diagramma UML del pattern Composite



- Lo **Static Factory Method** invece l'ho sfruttato per rendere la creazione di istanze della classe manager più semplice, chiara e soprattutto per poter instanziare un manager che può essere vuoto, contenente un solo contenitore od una collezione di essi senza dover stare a creare diversi costruttori in overloading che avrebbero rischiato di minare la chiarezza e la semplicità del codice oltre al fatto che, dovendo anche controllare di non star passando un valore null al posto di un contenitore o di una collezioni di essi avrei dovuto testare anche il costruttore, pratica che è preferibile evitare.

Classe del gestore con i tre Static Factory Method

DeliveryManager
-Collection<Container> deliveryContainers -String deliveryAddress
+ static DeliveryManager createEmptyDelivery(String deliveryAddress) +static DeliveryManager createDeliveryWithOneContainer(Container container, String deliveryAddress) +static DeliveryManager createDeliveryFromACollectionOfContainers(Collection<Container> deliveryContainers, String deliveryAddress) +boolean addContainer(Container container) +boolean removeContainer(Container container) +Iterator<Container> iterator() +double calculateDeliveryValue() +String getRecipe() Collection<Container> getDeliveryContainers()

Strategie per testare

Per testare le mie classi concrete ho usato la libreria "AssertJ" in quanto grazie alla sua interfaccia fluente risulta di più facile comprensione ed utilizzo rispetto ai metodi standard di Junit che al contrario risultano, in alcuni casi, di difficile interpretazione e comprensione.

Nei test per il gestore non ho testatolo lo Static Factory Method per l'istanziazione di un gestore a partire da una collezione di contenitori (tranne il controllo dei valori null) in quanto non faccio altro che passare direttamente i valori al costruttore e sarebbe risultato equivalente a testare un costruttore (questa caratteristica mi consente inoltre di instanziare nei test la classe del gestore senza usare altri metodi di cui non sono certo del funzionamento e che quindi vanno testati a loro volta con il rischio di fallire il test e non capire da dove provenga l'errore).

Infine ho anche aggiunto dei getter con visibilità di pacchetto per poter testare con più facilità il codice come ad esempio il getter per la lista dei contenitori da spedire che ho usato al posto di usare l'iterator che è inteso per essere utilizzato dai Client (tranne per il getter della collezione del contenuto di una scatola che ho dovuto dichiarare come protected in quanto è un metodo in ovveride dalla superclasse dei contenitori e quindi non ho potuto ridurgli la visibilità, ma avendo dichiarato la classe della scatola come final ho di fatto impostato la visibilità come se fosse effettivamente di pacchetto).

Lista Design Pattern usati nel progetto

I Pattern che usato nel progetto sono:

- **Visitor**
- **Static Factory Method**
- **Composite**