PROGETTO METODOLOGIE DI PROGRAMMAZIONE

Studente: Sofia Dami
Matricola: 6127313
E-mail: sofia.dami@stud.unifi.it

SPECIFICHE DEL SISTEMA

Il progetto consiste nella creazione di un sistema software per una **Libreria** nella quale sono messi in vendita fumetti, libri, cofanetti, gadget oppure pacchetti più grandi composti da fumetti, cofanetti, libri o altri pacchetti.
L'idea alla base del sistema è incentrata sulla creazione di oggetti [Carrello] posizionati all'interno della cassa, che consentono ai clienti del negozio di effettuare diverse operazioni.
Di seguito sono elencate le differenti funzionalità offerte dal suddetto software:

- Selezionare acquisti singoli o acquisti composti (pacchetti) messi a disposizione dal catalogo dalla libreria;
- aggiornare il conto di ogni carrello ad ogni aggiunta rimozione degli acquisti dal catalogo;
- selezionare varianti o gadget da aggiungere ai propri acquisti;
- Usufruire di eventuali sconti e omaggi a seconda degli acquisti selezionati (cofanetti);
- Effettuare il pagamento finale in contanti oppure bancomat, con successiva conferma tramite scontrino.

STRUTTURA

Analizziamo i vari pacchetti.

1. *catalogo*

Il primo pacchetto da prendere in esame è il package *catalogo*, che include l'interfaccia Acquisto, e le quattro classi di seguito descritte.

- *Acquisto*. Dichiara l'interfaccia comune a tutti gli oggetti della composizione e, tramite le operazioni *getNome* e *getPrezzo*, implementa il comportamento di default comune a tutte le sottoclassi: fra di esse ci sono i singoli ordini (Edizione, Edizione Variant e Cofanetto) e la classe Pacchetto, che rispettivamente rappresentano gli oggetti foglia e raggruppamenti dei primi.
- *Edizione*. Rappresenta l'oggetto base (ovvero ciò che può essere un libro, un fumetto, etc.), nonché uno dei singoli acquisti che è possibile aggiungere alla lista di acquisti del carrello, o ad un'eventuale composizione di acquisti. La classe è caratterizzata dalle Stringe titolo e autore, oltre che dal prezzo, inizializzati nel costruttore, e rispettivamente restituiti tramite le operazioni *getNome* e *getPrezzo* ereditati dall'interfaccia *Acquisto*.

- *EdizioneVariant*. È la classe astratta che rappresenta la base per la costruzione di un oggetto variabile, ovvero un ulteriore prodotto offerto dalla Libreria. Tale classe possiede dunque un riferimento all'interfaccia Acquisto, nel costruttore, e le implementazioni delle operazioni *getNome* e *getPrezzo* che rimandano rispettivamente al nome ed al prezzo dell'oggetto a cui fanno riferimento.
- *Cofanetto*. Simile alla classe Edizione, il Cofanetto rappresenta un ulteriore oggetto foglia a cui è possibile aggiungere o rimuovere sconti o promozioni (di cui parleremo in seguito). Tale classe è, infatti, un singolo acquisto selezionabile (che può, eventualmente, far parte di una composizione) su cui la Libreria può decidere di applicare opportuni sconti o omaggi. In questa classe la funzione *getNome* restituisce semplicemente il nome del cofanetto e – se presenti – i nomi delle promozioni ad esso applicate, mentre *getPrezzo* restituisce il prezzo con eventuali sottrazioni dovute alle promozioni applicate.
- *Pacchetto*. Il pacchetto rappresenta, invece, l'oggetto composto della composizione. Ha una Stringa che rappresenta il nome del pacchetto ed una lista di acquisti alla quale è possibile aggiungere e rimuovere singoli acquisti o, ricorsivamente, composizioni di ulteriori acquisti. Il prezzo del pacchetto è restituito – tramite uno stream – dalla sommatoria dei valori dei prodotti contenuti nel pacchetto.

2. *catalogo.extra*

Il secondo pacchetto è *catalogo.extra*, che contiene cinque classi, strettamente dipendenti dalle classi presenti in *catalogo*.

- *Variant*. Grazie al pattern Decorator, *Variant* è uno degli oggetti concreti di decorazione che possono essere aggiunti all'*EdizioneVariant*; è caratterizzato da una Stringa che descrive la variazione e che viene aggiunta alla stringa dell'*Edizione* di riferimento tramite l'operazione *getNome*. Il prezzo invece richiama quello della superclasse.
- *Gadget*. Similmente a *Variant*, *Gadget* rappresenta un altro oggetto concreto di decorazione. A differenza della suddetta classe, tuttavia, è caratterizzato da una descrizione (del Gadget) e da un prezzo extra che viene aggiunto al prezzo dell'*EdizioneVariant* di riferimento.
- *Promozione*. Interfaccia che definisce le due operazioni comuni alle sottoclassi *Omaggio* e *Sconto*, grazie al pattern Strategy; queste due rappresentano rispettivamente le due tipologie di promozione di cui è possibile usufruire grazie alla classe *Cofanetto*, e si differenziano nell'implementazione dei metodi relativi all'interfaccia estesa:
- *Omaggio*, per l'operazione *getNomePromozione*, restituisce "+" e una Stringa identificativa con il nome dell'omaggio in questione, mentre il prezzo rimane invariato;
- *Sconto*, similmente, restituisce una Stringa con l'annotazione del valore dello sconto per la prima operazione, mentre per *getPrezzoPromozione*, restituisce il valore dello sconto che verrà sottratto al prezzo del Cofanetto nella suddetta classe.

3. *gestione.carrello*

Il terzo pacchetto preso in esame è il package *gestione.carrello*, nel quale è possibile notare l'applicazione del pattern Observer. Si compone di tre classi.

- *Carrello*. La classe rappresenta l'oggetto base del Carrello per mezzo del quale la Libreria gestisce gli acquisti dei clienti e il successivo pagamento finale. Tale classe è costituita da un numero identificativo del carrello corrente, dal prezzo totale – aggiornato via via –, dalla lista degli acquisti e da una lista di osservatori. Le operazioni aggiungi osservatore e rimuovi osservatore vengono rispettivamente usate per aggiungere e rimuovere osservatori dalla lista degli osservatori, mentre il metodo notifica osservatore viene sfruttato per notificare tutti gli osservatori tramite una funzione *aggiorna*, ogni qualvolta viene richiamato. Inoltre possono

essere aggiunti e rimossi ordini dalla lista degli acquisti relativi al carrello in questione tramite le operazioni *aggiungiAcquistoAlCarrello* e *rimuoviAcquistoDalCarrello*; ogni volta che viene effettuata una di queste operazioni viene aggiornato il prezzo totale del carrello, tramite la funzione notifica.

- *CarrelloOsservatore*. Classe astratta che mantiene il riferimento al carrello corrispondente passato tramite costruttore. All'interno della classe viene richiamata la funzione di aggiunta o rimozione degli osservatori; è inoltre presente il metodo *aggiorna*, la cui implementazione viene lasciata alle classi che la estendono.
- *Conto*. Rappresenta l'oggetto concreto del conto, relativo al carrello passato come parametro nel costruttore. Esso è caratterizzato da una stringa che rappresenta lo Status del conto e che viene aggiornata, rimanendo consistente, ogni volta che vengono aggiunti o rimossi acquisti al carrello.

4. *pagamento*

Nel quarto e ultimo package *pagamento*, sono presenti tre classi.

- *Pagamento*. Si tratta di una classe astratta inizializza con la quantità di denaro in possesso per effettuare il pagamento finale, e dalla stringa *ricevuta*, utilizzata per simulare la stampa di uno scontrino. Il metodo *stampaRicevuta* scorre la lista degli acquisti del carrello a cui fa riferimento e per ciascuno restituisce nome e prezzo corrispondente; inoltre stampa il totale dovuto. Il metodo *pagamento* possiede due condizioni: se la quantità di denaro è inferiore al prezzo da pagare viene sollevata l'eccezione *CreditoInsufficienteEccezione*, se – invece – il denaro è maggiore di zero rimanda all'ultima funzione di conferma, che viene implementata dalle sue sottoclassi.
- *Contante*. La prima sottoclasse di Pagamento, rappresenta il pagamento tramite contanti. La sua implementazione concreta della funzione conferma richiama il metodo *stampaRicevuta* e restituisce la stringa del pagamento confermato con l'eventuale resto dovuto.
- *Bancomat*. La seconda sottoclasse di Pagamento, rappresenta l'opzione di pagamento tramite Bancomat o carta di credito. A differenza di *Contante*, implementa il metodo di conferma richiamando la funzione *stampaRicevuta* e restituisce direttamente la ricevuta del pagamento con il totale pagato.

ANALISI PATTERN APPLICATI

Il seguente schema elenca i pattern utilizzati all'interno del progetto e le classi coinvolte in ciascuno di essi:

- **Composite**: Acquisto, Edizione, EdizioneVariant, Cofanetto e Pacchetto;
- **Decorator**: Acquisto, Edizione, EdizioneVariant, Variant e Gadget;
- **Strategy**: Promozione, Sconto e Omaggio;
- **Observer**: Carrello, CarrelloOsservatore e Conto;
- **Template Method**: Pagamento, Bancomat e Contante.

Come struttura alla base del progetto è stato utilizzato il *pattern strutturale* **Composite**, in modo da poter realizzare una gerarchia di oggetti nella quale l'oggetto composto (il *Pacchetto*), è utilizzabile per creare un "contenitore" in grado di detenere sia oggetti elementari, sia – ricorsivamente – altri oggetti contenitori.

Il pattern si concentra dunque sulla creazione di insiemi (e non) di acquisti da parte della Libreria, in modo tale da offrire ai clienti rapide scelte di selezione dal catalogo, garantendo il funzionamento

del sistema e di trattare oggetti semplici e composti allo stesso modo, oltre a consentire in maniera semplificata l'aggiunta e la rimozione degli stessi.
Il ruolo del pattern sarebbe quindi quello di permettere al Client di navigare la gerarchia di oggetti e di comportarsi sempre nello stesso modo sia nei confronti degli oggetti semplici sia in quelli degli oggetti contenitori.
La gerarchia è caratterizzata dal component *Acquisto*, che dichiara l'interfaccia comune a tutte le sottoclassi e, oltre ad implementare il comportamento di default, definisce un modo per accedere ai componenti figli.
Tra gli oggetti leaf possiamo annoverare le classi *Cofanetto*, *Edizione* e *EdizioneVariant*; esse non hanno figli e definiscono il comportamento standard per gli oggetti primitivi della composizione.
Come menzionato sopra, il composite, l'oggetto contenitore, è rappresentato dalla classe *Pacchetto*, che detiene il riferimento ai componenti figli, definisce il comportamento degli oggetti contenitori ed implementa le operazioni relative alla gestione dei figli definite nell'interfaccia component.

Per testare questa parte del software è stato controllato che le foglie stampassero correttamente i propri contenuti e successivamente è stata testata la classe Pacchetto, testando che le stampe avvenissero correttamente anche per un pacchetto vuoto e infine sono stati effettuati diversi test tra cui: aggiunta/rimozione efficiente di vari ordini alla/dalla composizione, corretta restituzione del nome e del prezzo corrispondenti e accuratezza nella ricorsività delle composizioni.

A seguire, per la creazione delle edizioni "variabili" è stato è stato applicato il pattern strutturale **Decorator**, che ha permesso di realizzare Edizioni da "decorare" ovvero Edizioni a cui è possibile aggiungere *Gadget* oppure delle *Variant* (es. copertine speciali, cartoline con autografi, etc.), ovvero decorazioni aggiuntive scelte dal cliente con componenti appropriate. Essendo che il pattern permette di facilitare tale sviluppo, fornendo un'alternativa flessibile all'ereditarietà, non sarebbe un problema estendere ulteriormente la funzionalità degli oggetti anche in futuro, potendo impilare uno o più decorator l'uno sopra l'altro, aggiungendo nuove caratteristiche a run-time.

La struttura del pattern si compone di quattro elementi:
- il componente *Acquisto*, che rappresenta l'interfaccia dell'oggetto che dovrà esser decorato dinamicamente;
- il componente concreto *Edizione*, che rappresenta l'oggetto al quale andranno aggiunte tali decorazioni;
- il decoratore *EdizioneVariant*, che rappresenta il riferimento tra componente e decoratori concreti, mantenendo un riferimento ad Acquisto;
- i decoratori concreti *Variant* e *Gadget*, che rappresentano gli oggetti scelti dal cliente che aggiungono nuove funzionalità all'Edizione o, ricorsivamente, ad altre EdizioniVariant.

I test mostrano la correttezza e l'efficienza di tale pattern sia con l'aggiunta di semplici componenti ad un edizione di base, sia con l'aggiunta ricorsiva di decorazioni ai edizioni variabili.

Per quanto concerne, invece, il *Cofanetto*, sono state applicate delle promozioni aggiuntive di cui un cliente può approfittare: per invogliare all'acquisto, la Libreria può aggiungere sconti o omaggi che possono essere selezionati solo acquistando un cofanetto.
Per fare ciò ci si è serviti del pattern comportamentale **Strategy**, in modo da utilizzare due diverse implementazioni a seconda del contesto scelto. Tale pattern consiste nell'incapsulare un algoritmo all'interno delle classi *Omaggio* e *Sconto*, lasciando all'interfaccia generica *Promozione* il compito di dichiarare l'interfaccia utilizzata da Cofanetto per invocarei metodi concreti *getNomePromozione* e *getPrezzoPromozione*. Le due strategie concrete *Omaggio* e *Sconto*, si differenziano – infatti – per l'implementazione dei metodi ereditati, come abbiamo spiegato nei paragrafi precedenti.

I test riguardanti il funzionamento di tale pattern sono presenti nella classe *CofanettoTest*.
È stata testata la correttezza dei nomi e dei prezzi del cofanetto, con e senza promozioni, oltre che a sfruttare le operazioni di aggiunta e rimozione degli stessi.

Ogni Carrello dispone di un conto a sé stante che tiene traccia degli ordini selezionati e fornisce la somma totale delle spese interessate. Per realizzare ciò, è stato utilizzato il pattern comportamentale **Observer**, al fine di implementare una dipendenza uno-a-molti in cui il cambiamento di stato del soggetto Carrello viene notificato agli osservatori che si sono mostrati interessati (Conto).
Il Subject del pattern in questione è – appunto – l'oggetto Carrello, che consente agli osservatori di essere aggiunti o rimossi mantenendo comunque un riferimento alla lista con tutti gli osservatori, conformi al tipo astratto *CarrelloOsservatore*. Il Carrello ovviamente non conosce il tipo concreto dei suoi osservatori e crea, così, un accoppiamento minimo ed astratto. Inoltre, quando cambia il suo stato – ovvero quando vengono aggiunti e rimossi gli acquisti dal carrello – automaticamente tutti i suoi osservatori vengono informati ed aggiornati. *CarrelloOsservatore* rappresenta l'Observer del pattern e l'interfaccia per gli oggetti che devono essere notificati riguardo ai cambiamenti del Subject. Il Conto, infine, è l'osservatore concreto. Basato sull'idea della creazione di un conto del carrello, mantiene un riferimento al Subject ed un proprio stato che deve rimanere consistente con lo stato del Carrello; esso implementa l'interfaccia CarrelloOsservatore modificando il proprio stato Status ogni qualvolta che cambia lo stato del Subject o, più nello specifico, che vengono aggiunti e rimossi acquisti dal carrello.

Nella cartella dei tests sono evidenti due classi che hanno permesso di testare il funzionamento delle classi appartenenti al package *gestione.carrello*, nonché la correttezza del pattern Observer.
ContoTest testa gli effettivi conti relativi a composizioni di ordini semplici e ricorsivi (e nel caso il conto fosse vuoto o inesistente), in modo tale da verificare l'accoppiamento di ciascun osservatore con il Subject corrispondente.

Per ultimo è stato utilizzato il **Template Method**, un pattern comportamentale basato su classi, qui impiegato per definire la struttura di un algoritmo delegando alcuni passi alle sue sottoclassi.
Il suo utilizzo all'interno del progetto è volto a specificare l'ordine delle operazioni da effettuare per concludere l'operazione di pagamento, lasciando a Bancomat e Contante, l'implementazione di tale operazione. Come si può ben vedere, il metodo pagamento – che definisce l'algoritmo – viene implementato nell'omonima superclasse, mentre il metodo *conferma* specifica il comportamento in dettaglio e viene dichiarato astratto, per essere così implementato nelle sottoclassi.

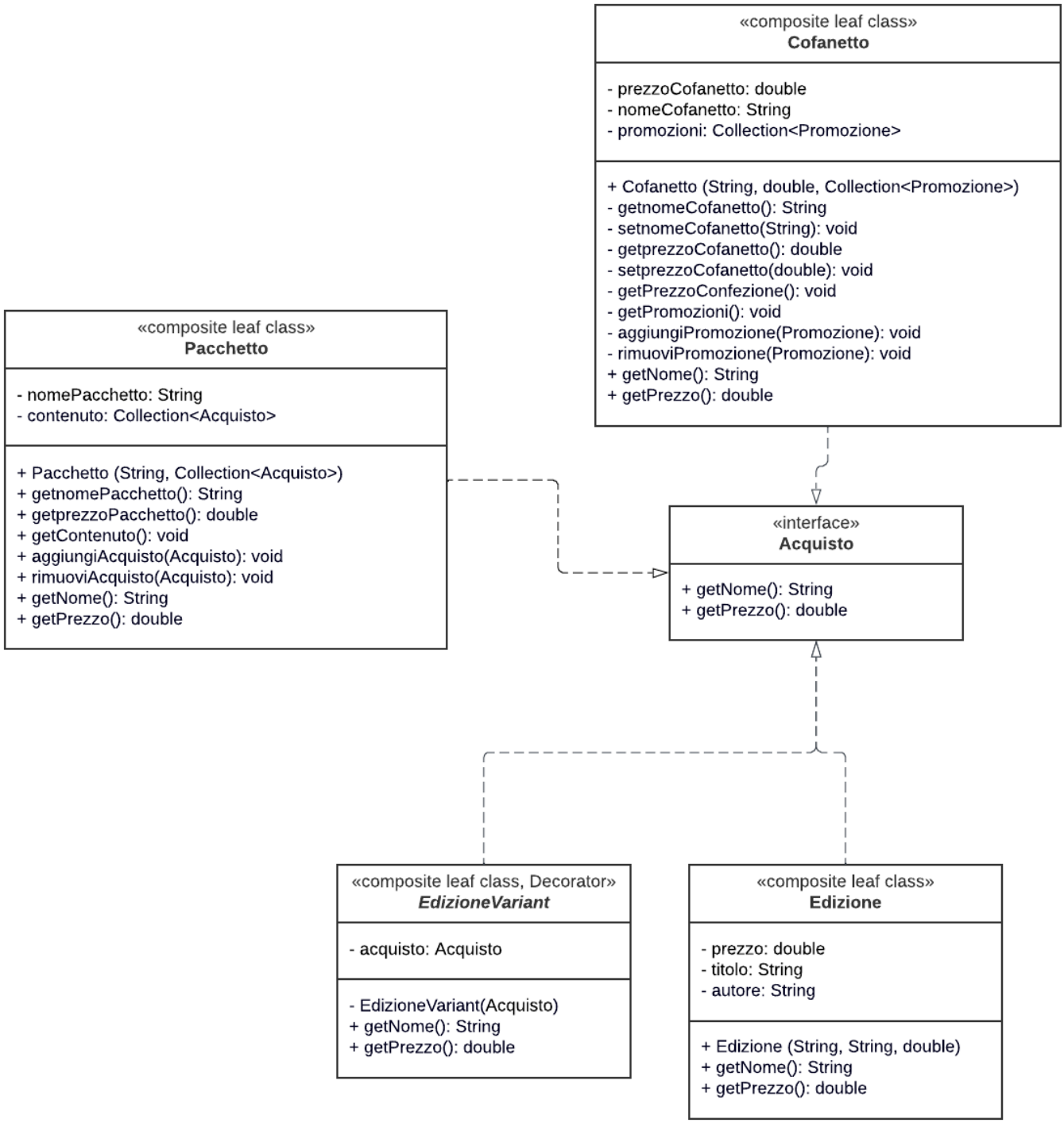
La funzione di pagamento è infatti caratterizzato da un ramo *if-else* che comporta di verificare due condizioni: se il cliente non possiede denaro sufficiente per effettuare il pagamento viene sollevata un'eccezione, in caso contrario si verifica se sono presenti acquisti nel carrello (per mezzo del prezzo totale) e si procedere col pagamento vero e proprio. Nel caso in cui le condizioni non siano entrambe verificate, non verrebbe generata alcuna ricevuta.
Il metodo conferma è astratto e viene implementato dalle sottoclassi Contante e Bancomat: la loro implementazione è simile e si differenziano unicamente per la modalità di pagamento effettuata: nel primo caso viene versata l'intera somma di denaro posseduta e, in caso, riportato il resto, mentre nel caso di pagamento tramite l'oggetto Bancomat viene versata l'esatta somma di denaro richiesta.
In ogni caso viene generata la ricevuta.

Per quanto riguarda le classi di test, sono state implementate le classi *ContanteTest* e *BancomatTest*; in particolare, sono stati effettuati diversi test fra i quali: conferma di pagamento, ricezione corretta della quantità di resto in contanti, mancata ricevuta causata dall'assenza di acquisti, assenza di fondi sufficienti per effettuare l'effettivo pagamento.

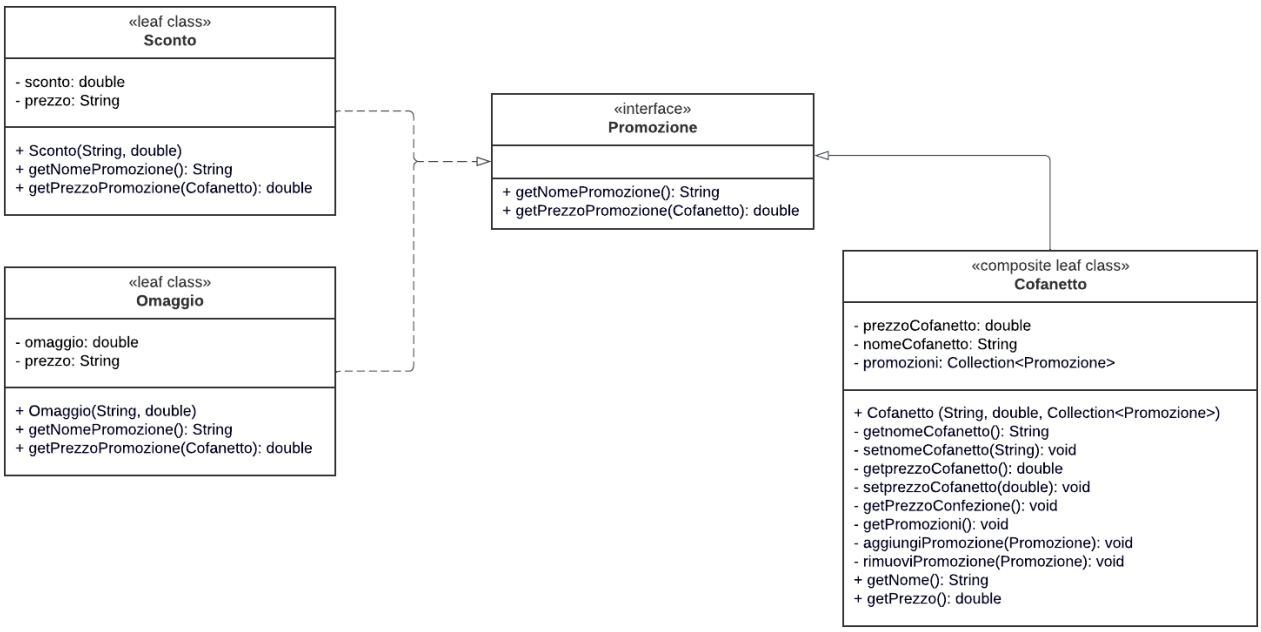
DIAGRAMMI UML DELLE CLASSI

Le seguenti immagini illustrano i diagrammi UML delle interazioni fra le classi del software.

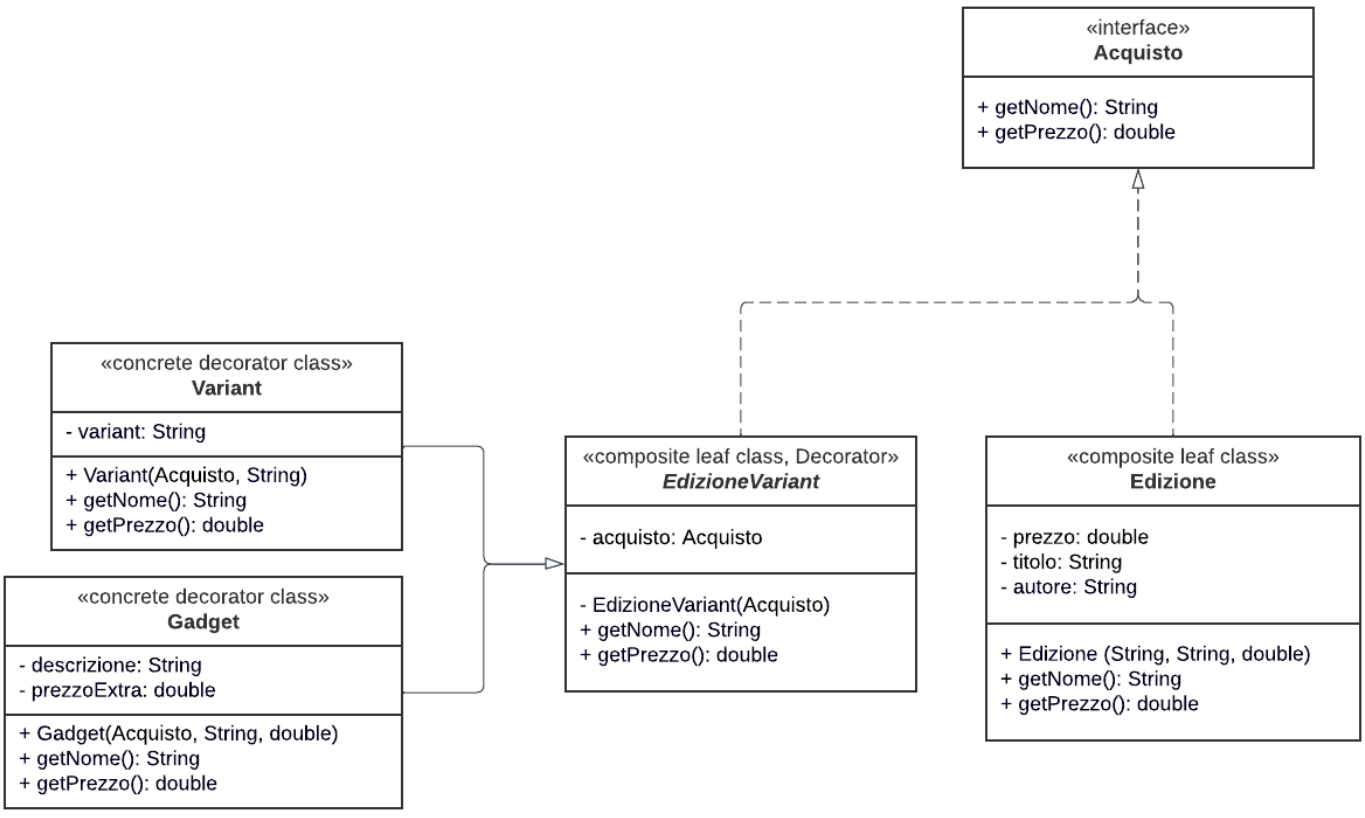
1. Diagrammi dell'interfaccia Acquisto e delle sottoclassi Cofanetto, Pacchetto e le varie Edizioni, che implementano il suo comportamento di default.



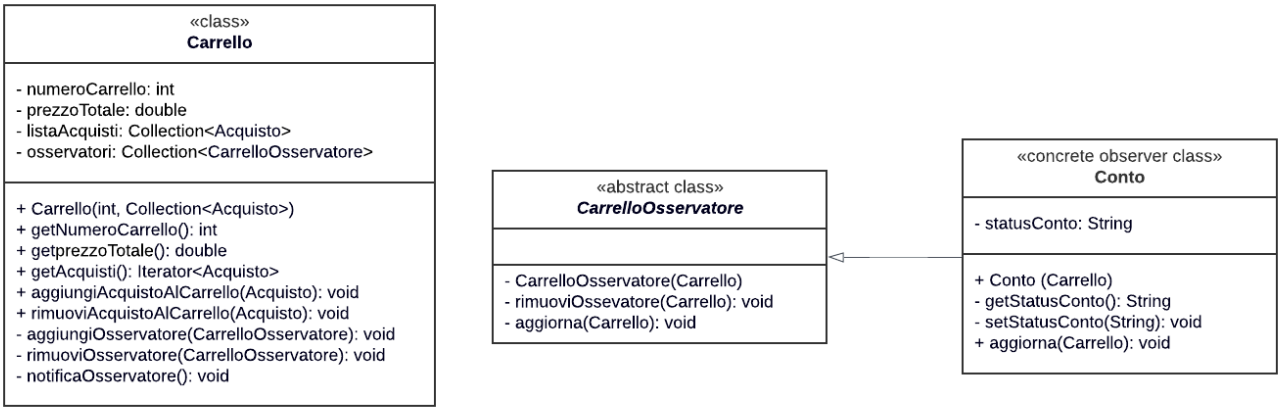
2. Interfaccia Promozione (implementata da Cofanetto) e le due sottoclassi Sconto e Omaggio.



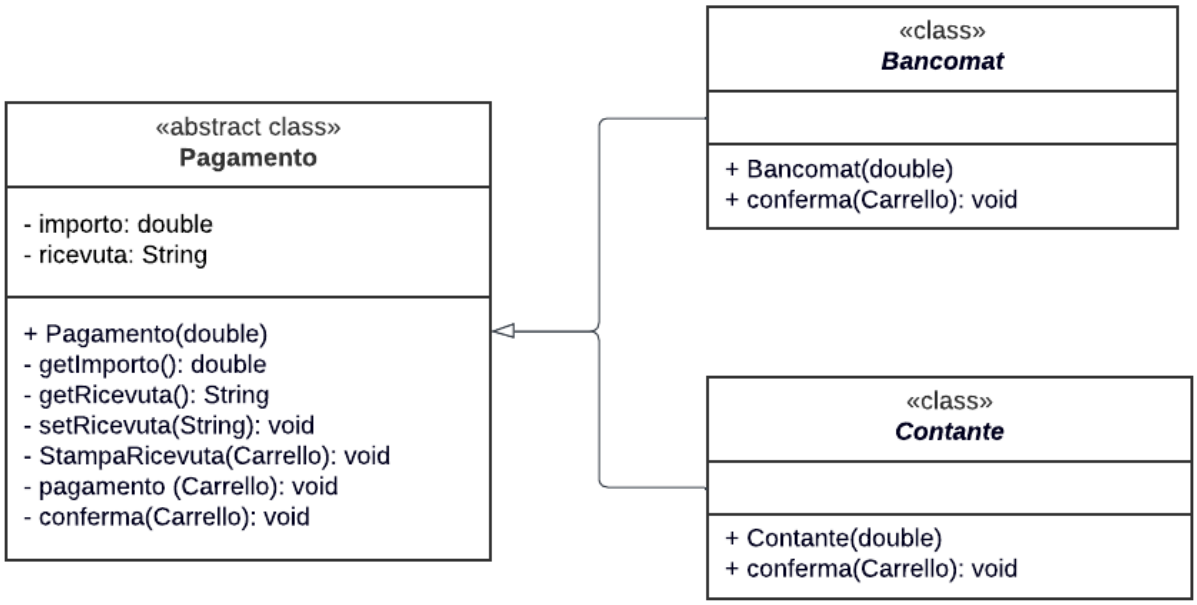
3. Utilizzo del Pattern Decorator.

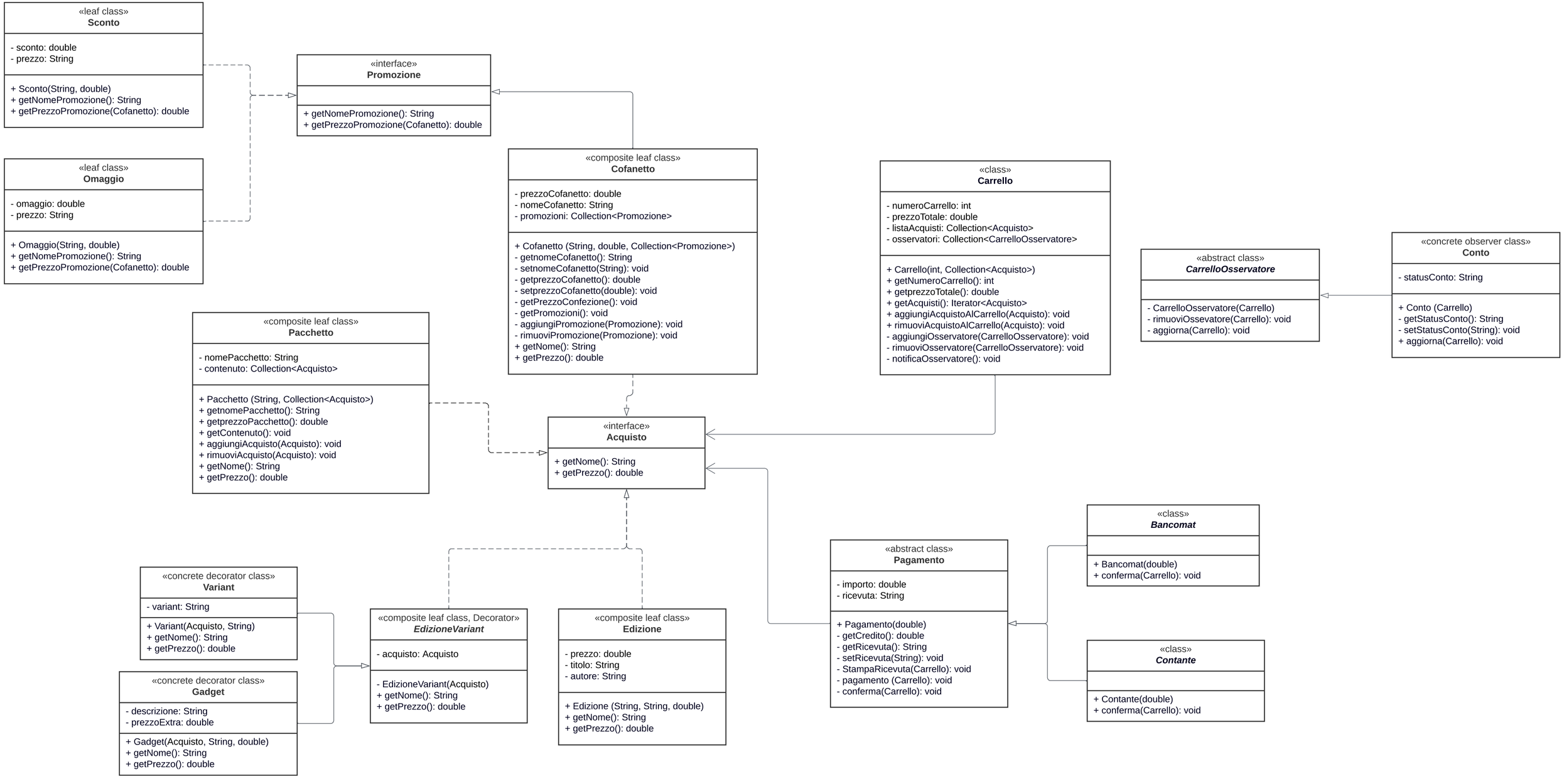


4. Relazione fra la classe astratta Carrello Osservatore e la sottoclasse Conto.



5. La classe astratta Pagamento e le due sottoclassi Contante e Bancomat.




«leaf class»
Sconto
- sconto: double
- prezzo: String
+ Sconto(String, double)
+ getNomePromozione(): String
+ getPrezzoPromozione(Cofanetto): double
«interface»
Promozione
+ getNomePromozione(): String
+ getPrezzoPromozione(Cofanetto): double
«leaf class»
Omaggio
- omaggio: double
- prezzo: String
+ Omaggio(String, double)
+ getNomePromozione(): String
+ getPrezzoPromozione(Cofanetto): double
«composite leaf class»
Cofanetto
- prezzoCofanetto: double
- nomeCofanetto: String
- promozioni: Collection<Promozione>
+ Cofanetto (String, double, Collection<Promozione>)
- getnomeCofanetto(): String
- setnomeCofanetto(String): void
- getprezzoCofanetto(): double
- setprezzoCofanetto(double): void
- getPrezzoConfezione(): void
- getPromozioni(): void
- aggiungiPromozione(Promozione): void
- rimuoviPromozione(Promozione): void
+ getNome(): String
+ getPrezzo(): double
«class»
Carrello
- numeroCarrello: int
- prezzoTotale: double
- listaAcquisti: Collection<Acquisto>
- osservatori: Collection<CarrelloOsservatore>
+ Carrello(int, Collection<Acquisto>)
+ getNumeroCarrello(): int
+ getprezzoTotale(): double
+ getAcquisti(): Iterator<Acquisto>
+ aggiungiAcquistoAlCarrello(Acquisto): void
+ rimuoviAcquistoAlCarrello(Acquisto): void
- aggiungiOsservatore(CarrelloOsservatore): void
- rimuoviOsservatore(CarrelloOsservatore): void
- notificaOsservatore(): void
«abstract class»
CarrelloOsservatore
- CarrelloOsservatore(Carrello)
- rimuoviOssevatore(Carrello): void
- aggiorna(Carrello): void
«concrete observer class»
Conto
- statusConto: String
+ Conto (Carrello)
- getStatusConto(): String
- setStatusConto(String): void
+ aggiorna(Carrello): void
«composite leaf class»
Pacchetto
- nomePacchetto: String
- contenuto: Collection<Acquisto>
+ Pacchetto (String, Collection<Acquisto>)
+ getnomePacchetto(): String
+ getprezzoPacchetto(): double
+ getContenuto(): void
+ aggiungiAcquisto(Acquisto): void
+ rimuoviAcquisto(Acquisto): void
+ getNome(): String
+ getPrezzo(): double
«interface»
Acquisto
+ getNome(): String
+ getPrezzo(): double
«concrete decorator class»
Variant
- variant: String
+ Variant(Acquisto, String)
+ getNome(): String
+ getPrezzo(): double
«concrete decorator class»
Gadget
- descrizione: String
- prezzoExtra: double
+ Gadget(Acquisto, String, double)
+ getNome(): String
+ getPrezzo(): double
«composite leaf class, Decorator»
EdizioneVariant
- acquisto: Acquisto
- EdizioneVariant(Acquisto)
+ getNome(): String
+ getPrezzo(): double
«composite leaf class»
Edizione
- prezzo: double
- titolo: String
- autore: String
+ Edizione (String, String, double)
+ getNome(): String
+ getPrezzo(): double
«abstract class»
Pagamento
- importo: double
- ricevuta: String
+ Pagamento(double)
- getCredito(): double
- getRicevuta(): String
- setRicevuta(String): void
- StampaRicevuta(Carrello): void
- pagamento (Carrello): void
- conferma(Carrello): void
«class»
Bancomat
+ Bancomat(double)
+ conferma(Carrello): void
«class»
Contante
+ Contante(double)
+ conferma(Carrello): void