README

Indice

1. PATTERN MODEL-VIEW-CONTROL (MVC)

La scelta del pattern MVC è stata presa dalle richieste del progetto. Si richiede un'interfaccia utente che si collegasse a un qualsiasi linguaggio di programmazione. Grazie a questo pattern si può andare a sostituire velocemente qualsiasi singola parte del MVC per andarla a sostituire in maniera semplice.

Le parti nel progetto sono distinte nel seguente modo:

- MODEL: package "calc", classi: Importa, ListaMerci, Merce, CalcolaSconto e Stampa
- VIEW: package "interfaccia", classe: Interfaccia
- CONTROL: package "controlli", classe: Controllo

L'immagine 1.1 mostra come sono collegati tra loro e come circolano i dati al loro interno.

Esempi di cambiamenti facili dovuti a MVC:

- Si può andare a cambiare l'interfaccia grafica per poterla vedere diversamente oppure per poterla scrivere in un altro linguaggio.
- Si può cambiare il controller o aggiungerne altri qual ora dovessero sorgere nuove precondizioni.
- Si può sostituire il model, se per esempio, ci si deve interfacciare ad un altro database e analizzare dati diversi.

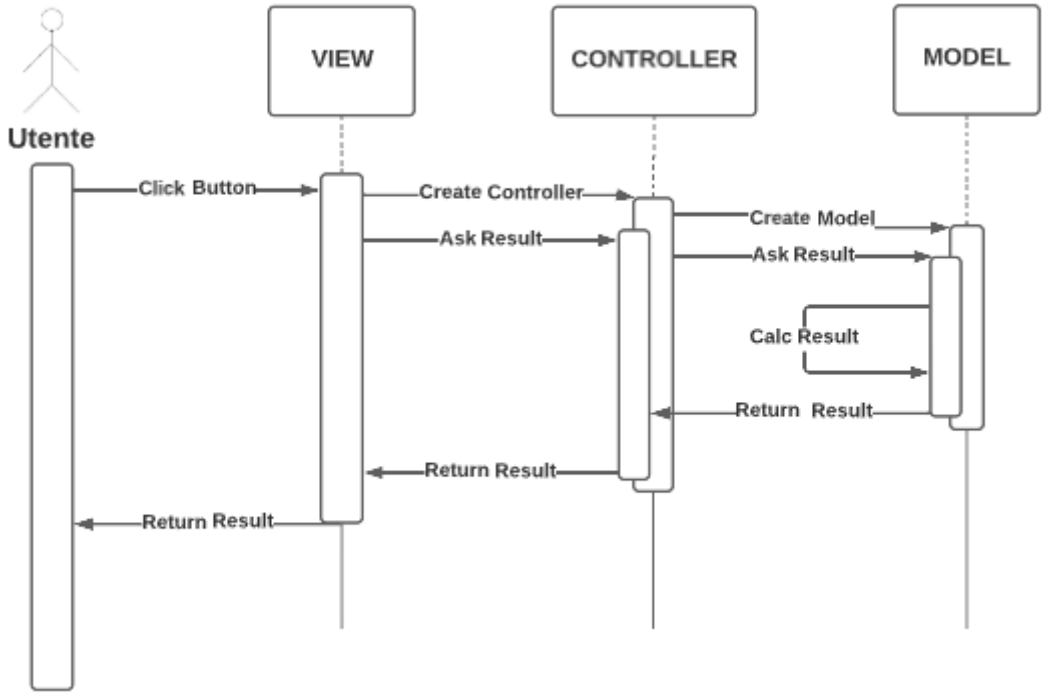


Figura ***Errore. Nel documento non esiste testo dello stile specificato.****.1*

2. VIEW

OBIETTIVO VIEW: Essere semplice e intuitiva per l'utente che la deve usare.

Il test richiede di usare un'interfaccia utente programmata con tecnologia web. Purtroppo, non avendo mai fatto tale linguaggio di programmazione non sapevo dove andare a mettere le mani, per questo ho continuato utilizzando java swing in maniera tale da avere un'idea di come sarebbe stata l'interfaccia utente. Teoricamente utilizzando il pattern MVC, la parte di controller e di model sarebbe stata identica, al massimo ci sarebbe stato un controller grafico in più che si sarebbe occupato di collegarsi col controller già esistente.

La scelta di usare label, separator, text box e pulsanti è dovuta all'obbiettivo della VIEW.

I risultati potevano essere visualizzati possibilmente in due modi:

1. Tramite una label.
2. Tramite una finestra di dialogo.

Ho scelto la seconda possibilità rispetto alla prima per un motivo di codice. La finestra dialogo rispetto alla label rispetta le spaziature e i "a capo" ("\n") presenti nella stringa che si vuole visualizzare a video. Costruita la stringa che si vuole visualizzare è bastato passarla come input al metodo che permette di visualizzare la finestra. Se avessi scelto la label sarei dovuto andare a creare una label e una stringa per ogni "a capo" che avrei voluto inserire.

3. CONTROLLER

OBIETTIVO CONTROLLER: assicurarsi di passare input pertinenti alla parte model.

Per risolvere l'obiettivo potevo procedere in due modi:

1. Creare un controller e nel momento in cui si chiedono i risultati analizzare gli input.
2. Creare un controller, analizzare i dati e nel momento in cui si chiedono i risultati controllare gli input.

Entrambi i casi sono validi, ma per pura scelta stilistica sono andato a scegliere il secondo modo. Trovo che il secondo metodo sia di più di facile comprensione nel caso in cui si vada a cambiare gli input, mentre nel caso in cui si vadano ad aggiungere degli input la differenza sia quasi nulla.

Per valutare gli input ho scelto i seguenti limiti:

- Nome:
  - Non può essere vuoto.
  - Non può contenere solo spazi.
- Num:
  - Non può essere vuoto.
  - Non può contenere solo spazi.
  - Non può contenere caratteri diversi dallo spazio e dai numeri.
  - Deve essere un numero maggiore o uguale a 1.

Se non si rispettano questi limiti si viene ad assegnare una stringa d'avviso alla variabile nome e un valore impossibile da avere al num (-1).

Quando la VIEW chiederà i risultati, questo oggetto andrà a creare un oggetto "Importa" e delegherà il compito esattamente come MVC richiede.

4.MODEL

OBIETTIVO MODEL: ottenere i dati delle merci da un Database, filtrarli e ottenere il risultato.

A differenza delle altre parti ho usato più classi. Per questo motivo ho deciso di usare altri pattern:

- Single Responsibility Principle (SRP)
- Information Expert

4.1 CLASSE IMPORTA

OBIETTIVO IMPORTA: importare da un Database le informazioni e filtrarle.

Ho scelto questa classe come collegamento tra i livelli CONTROLLER e MODEL in quanto penso che la prima azione utile da fare sia verificare se ci sono dei fornitori che soddisfino i requisiti di input.

La scelta di usare una terza parte (UCanAccess) per connettersi al Database è stata obbligatoria in quanto non esistono comandi in Eclipse e in Access per poter effettuare la connessione.

Ho scelto di inserire in questa classe un riferimento alla classe ListaMerci in quanto si occupa di andare a cercare gli elementi utili di questa lista. Questa scelta da sia le informazioni per ritornare i risultati alla VIEW che per ottenere i dati dal Database. In linea con i due pattern scelti ho dovuto andare a creare una nuova classe a cui passare la ListaMerci in maniera tale da dividere le due responsabilità.

Per l'inserimento dello sconto nella Merce avevo due possibilità:

1. Gestire subito la stringa in input, creare una classe per ogni tipologia di sconto e usare un'interfaccia da implementare in tutte le classi sconto.
2. Gestire a posteriori la stringa sconto.

Ho scelto la seconda per due motivi:

1. Volevo che la Merce rispecchiasse simmetricamente il record di una tabella del Database. Siccome lo sconto nel Database è una stringa ho mantenuto lo stesso formato nel codice.
2. La prima opzione può essere migliore in altri casi. Per esempio, se oltre agli sconti si volessero gestire anche delle offerte del tipo "3X2", magari avere una classe per ogni offerta e sconto che implementano un'interfaccia appropriata potrebbe risultare migliore a livello di progettazione. Per la realizzazione del test risulta migliore la seconda opzione in quanto parlando di soli sconti, con un semplice switch il tutto risulta più intuitivo.

4.2 CLASSE STAMPA

OBIETTIVO STAMPA: creare la stringa da visualizzare come risultato.

Questa funzione è stata messa in una classe apposita per due motivi:

1. Come citato in precedenza, per rispettare i pattern.
2. Si possono avere diversi modi di visualizzazione dei risultati. Cambiando questa classe è possibile alterare il modo di come si visualizzano i risultati senza andare a mettere mano nel resto del codice.

La scelta di usare uno StringBuilder invece che altre soluzioni come lo StringBuffer è per diminuire gli accessi in lettura in memoria.

4.3 CLASSI LISTAMERCI e MERCE

OBIETTIVO: Salvare e tenere a disposizione le merci da cui estrapolare i risultati.

Su queste due classi c'è poco da dire; essenzialmente si occupano della parte di gestione degli oggetti relativi a sé stesse.

Ho preferito usare un ArrayList piuttosto che altro in quanto è possibile andare ad aggiungere e togliere record dalle tabelle rendendo impossibile sapere un numero definito in un determinato istante. Usando questa struttura di collezione d'oggetti posso pensare a collezionare record del database senza avere un limite definito.

4.4 CLASSE CALCOLASCONTO

Per lo sconto, come citato in precedenza, ho due possibili alternative:

1. Gestire in una classe diversa da "Merce" lo sconto.
2. Gestire al di fuori della classe "Merce" lo sconto, creando una classe apposita per ogni tipo di sconto e implementando un'interfaccia comune.

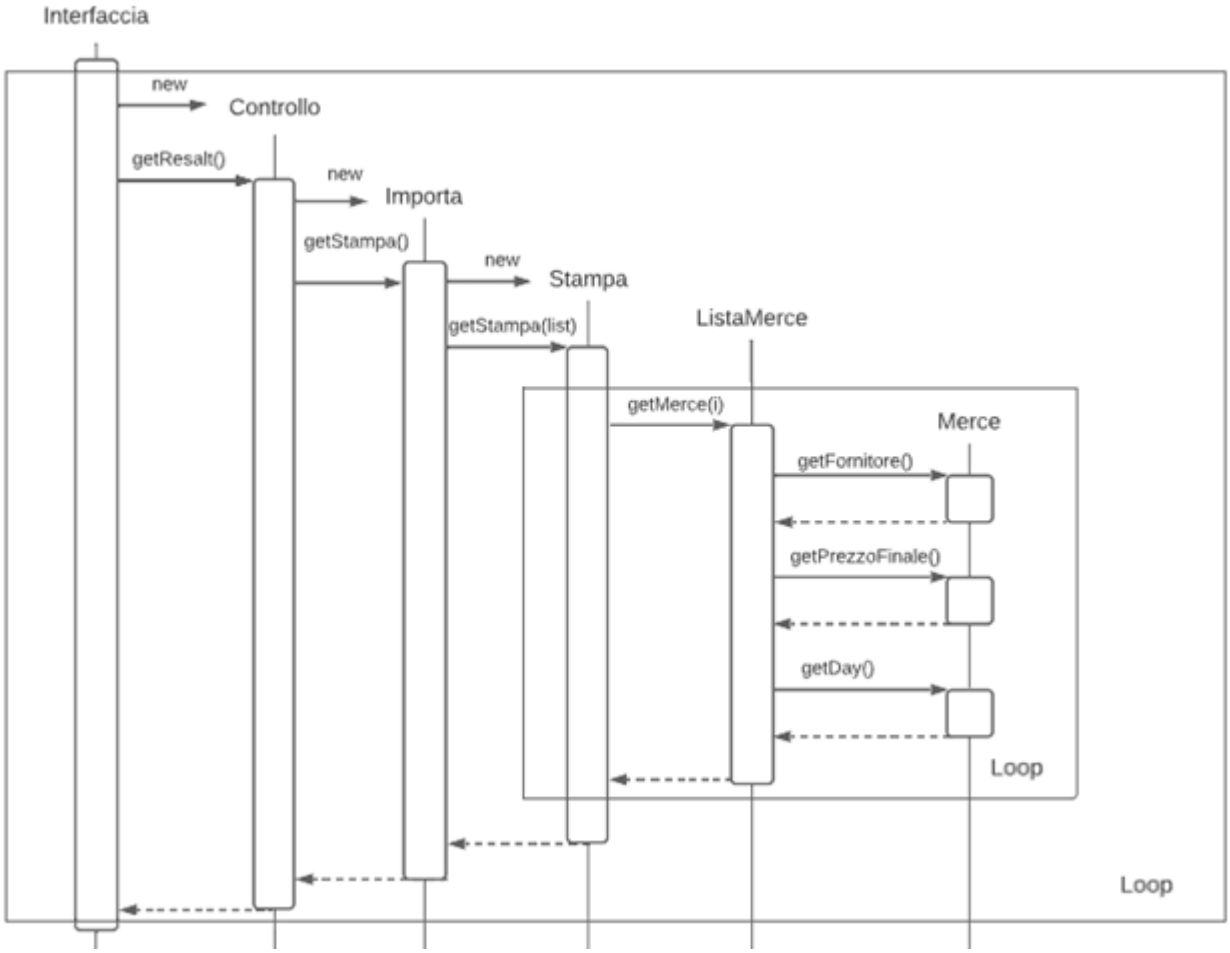
Ho scelto la prima opzione non solo per avere il parallelismo citato nel paragrafo 4.2, ma anche perché finché si parla di singolo sconto mi sembra migliore l'opzione 1. Quando sorgeranno altri metodi di risparmio al di fuori dello sconto si potrà decidere se gestirli esattamente come fatto per lo sconto o portare il tutto in una progettazione che sfrutta la prima alternativa.

Ho dovuto scegliere un determinato modo di inserire gli sconti nel Database in maniera tale da poter calcolare correttamente il prezzo finale. Per sapere l'intera sintassi andare nel "Manuale uso e installazione", qua riporterò solo il significato dei simboli proposti:

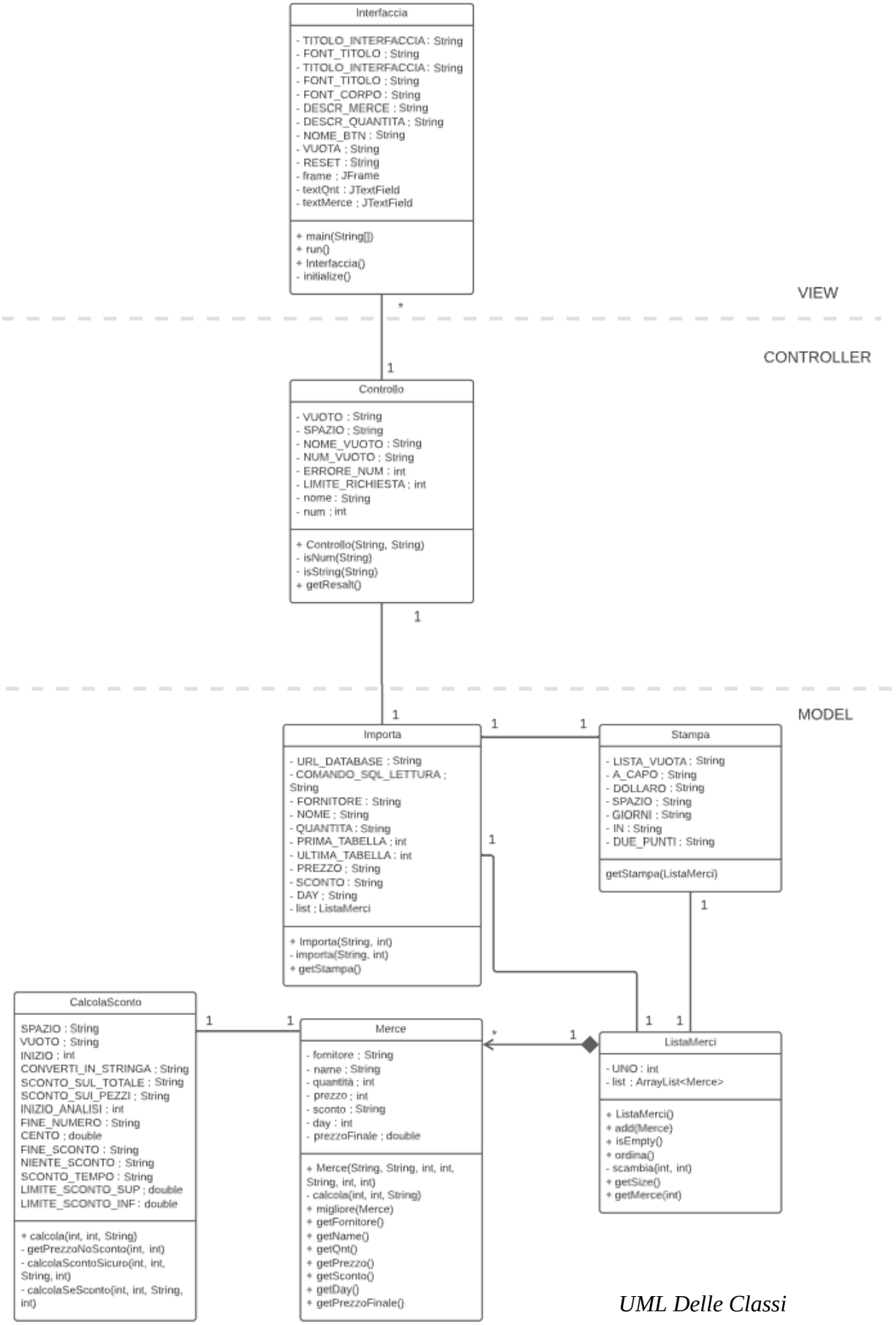
- "/": Nessuno sconto.
- "+": Sconto se il totale è almeno un totale.
- "-": Sconto se si acquistano almeno una determinata quantità.
- "%": Fine parametro di sconto e inizio valore di sconto.
- "$": Sconto fisso a un determinato valore (può rappresentare lo sconto fisso in un determinato periodo di tempo in un fornitore).
- "!": Fine stringa che descrive lo sconto.

L'utilizzo delle strutture try, catch è stato scelto in quanto se si inseriscono delle stringhe nel campo sconto in maniera diversa da quella scelta, è possibile che si vengano a generare delle eccezioni. Queste eccezioni vengono a essere gestite considerando la merce a prezzo pieno.

5.UML



UML Delle Sequenze



UML Delle Classi