Relazione progetto Metodologie di Programmazione (Monicolini Matteo A.S 2021/2022)

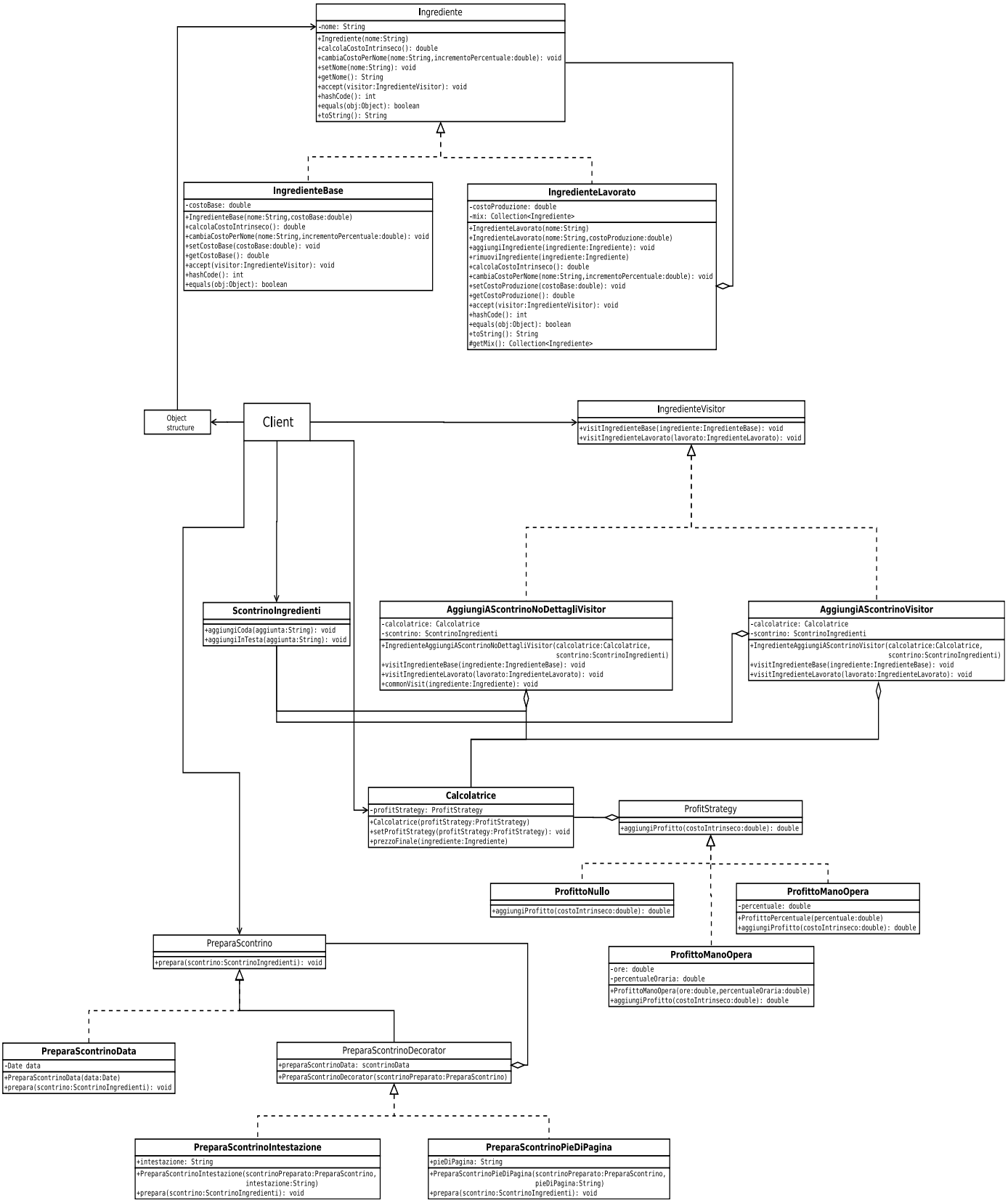
Funzionalità del sistema:

L'idea alla base del sistema è quella di implementare un programma per la gestione della vendita di ingredienti all'interno di un punto caldo (ad es. un forno o una pizzeria). In particolare, a riguardo sono state implementate le seguenti funzionalità:

- Gestione degli ingredienti con distinzione fra ingredienti base e ingredienti lavorati. Un ingrediente base è un ingrediente che viene acquistato dall'esterno. Un ingrediente lavorato è un ingrediente ottenuto dall'unione di più ingredienti base e/o dalla lavorazione degli stessi. I cibi pronti per essere mangiati sono di conseguenza visti come degli ingredienti lavorati. Tutti gli ingredienti hanno un nome. Gli ingredienti base hanno un costo base che è da decidere in fase di creazione dell'ingrediente. Gli ingredienti lavorati al contrario possono essere creati senza impostare alcun costo o impostando un costo di produzione.
- Un metodo per il calcolo del costo intrinseco di un ingrediente. Questo metodo ritorna un valore che rappresenta quello che l'ingrediente in questione è costato al venditore. Di conseguenza un ingrediente base avrà costo intrinseco equivalente al costo base. Un ingrediente lavorato avrà costo intrinseco equivalente alla somma del costo base degli ingredienti che lo compongono più il suo costo di produzione (se presente). In questo modo il venditore può determinare in fase di vendita il costo finale a cui vendere l'ingrediente decidendo il profitto che intende farne e sommando questo valore al costo intrinseco.
- Un metodo che permette di modificare il costo base degli ingredienti base in relazione al proprio nome. In particolare, è possibile definire una percentuale di cui incrementare il costo base, e una stringa da cercare all'interno del nome degli ingredienti base. Questo metodo eseguito su un ingrediente base andrà ad incrementare il costo base dell'ingrediente base se il suo nome contiene quella stringa. Il metodo è applicabile anche sugli ingredienti lavorati, e in quel caso verrà eseguita una visita ricorsiva di tutti gli ingredienti che lo compongono e verrà incrementato il costo di tutti gli ingredienti base il cui nome contiene la stringa. In questo modo è possibile essere specifici (ad es. "Farina di grano tenero di tipo 00") oppure più generici (ad es. "Farina") a seconda della necessità.
- Una calcolatrice, che utilizza diversi algoritmi per calcolare un valore finale partendo da un valore di base. I due algoritmi implementati per questo sistema in particolare sono: uno basato sul numero di ore impiegate per la produzione e l'incremento di costo percentuale per ogni ora, ed uno basato su un rincaro in percentuale sul costo totale. Ogni volta che il venditore vuole calcolare il costo finale di un ingrediente, questo dovrà scegliere la strategia da usare per calcolare il profitto e quanto profitto generare. In un'ipotetica implementazione nella realtà l'idea è quella di avere la strategia con percentuale sul totale, con una percentuale decisa in partenza, come strategia di default. A quel punto il venditore potrà decidere di cambiare la percentuale o di usare la strategia con mano d'opera per ingredienti particolari che lo richiedono.
- La generazione di uno scontrino basilare, formato dai soli ingredienti che un compratore ha deciso di acquistare, con possibilità di aggiunta di un ingrediente in cima o in fondo allo scontrino. Sono stati pensati due metodi di aggiunta degli ingredienti allo scontrino, uno dettagliato ed uno non dettagliato. Con la modalità dettagliata nel caso in cui un ingrediente lavorato sia acquistato vengono elencati anche tutti gli ingredienti che lo compongono con il relativo prezzo, ricorsivamente, mentre con la modalità non dettagliata questi dettagli vengono omessi.

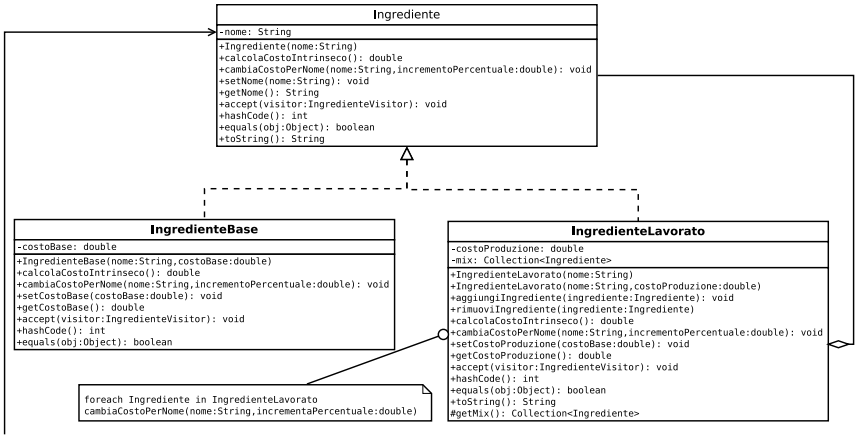
- La personalizzazione dello scontrino prima che questo sia mandato in fase di stampa, con aggiunta automatica di ora e data correnti e possibilità di aggiungere intestazione e/o pie di pagina personalizzabili.

Diagramma UML delle classi:

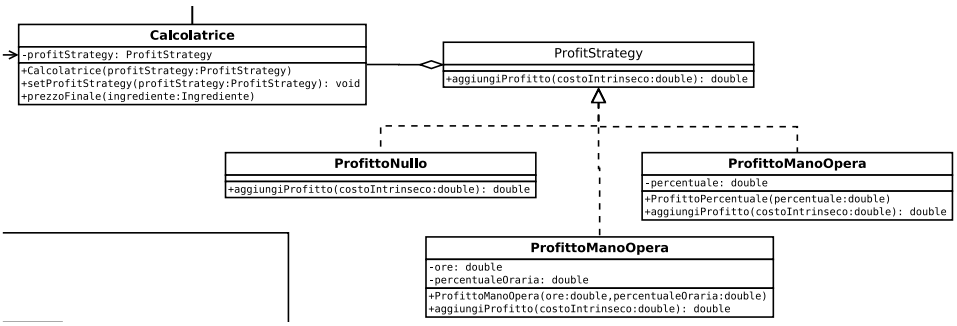


Pattern applicati:

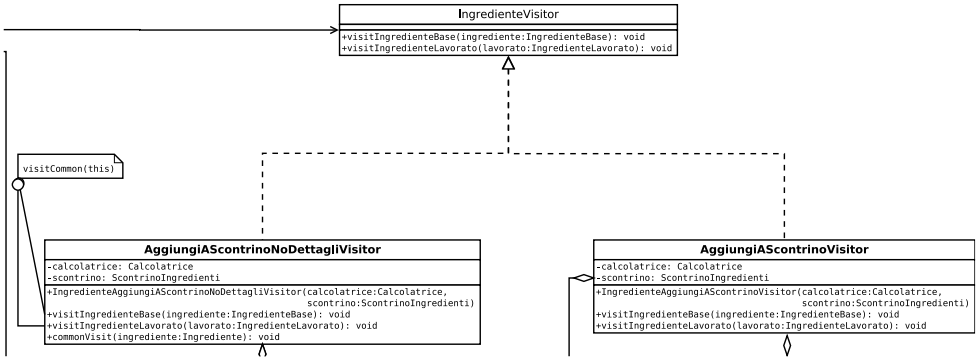
Composite(Design per Type Safety):



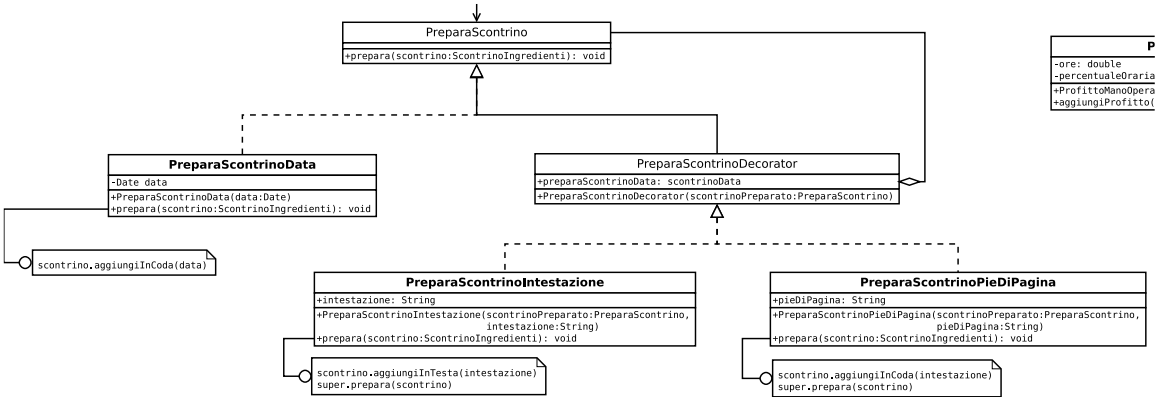
Strategy:



Visitor (Visitor void):



Decorator (variante con classe base)



Scelte di design per il progetto:

Test:

Per i test ho deciso di utilizzare la libreria “AssertJ” in modo da avere una migliore leggibilità, e una gestione migliore dei messaggi di errore in fase di testing. Una spiegazione più approfondita per i singoli test sarà esposta nei successivi paragrafi.

Idea di base:

l’idea di base del progetto era quella di avere una dispensa di ingredienti che un venditore possiede e che intende catalogare per poi venderli. Pensando al processo con cui si lavorano gli ingredienti, l’associazione al pattern composite è stata immediata, questo perché nella produzione di un cibo sono sempre utilizzati degli ingredienti di base di cui si dispone, ed alcune preparazioni intermedie, a loro volta formate da altri ingredienti. Di conseguenza è facile pensare ad un ingrediente base come ad una foglia del pattern e ad una preparazione intermedia o un prodotto finito, come ad un composto del pattern. La scelta di utilizzare la tipologia di composite con design per type safety è stata fatta perché in questa implementazione non vedevo nessun motivo per introdurre il bisogno di gestire eventuali errori a runtime per garantire l’uniformità. Inoltre, non ci sono mai riferimenti da parte dei figli al loro ipotetico padre.

Per quanto riguarda i campi da inserire nella classe astratta Component, inizialmente avevo pensato di inserire oltre al campo Nome, anche il campo costoBase, inteso come costo intrinseco, tuttavia questo avrebbe reso problematico fare sì che un composto potesse avere staticamente il valore del suo costo intrinseco corretto, soprattutto nel caso in cui il composto contenesse altri composti annidati. In più questa scelta avrebbe portato il costo base di un ingrediente lavorato a poter essere impostato da un eventuale setter, cosa che è in contrasto con la logica che volevo dare al campo costoBase in quel caso. Di conseguenza ho deciso di inserire questo campo solo nella classe IngredienteBase, e lasciare che il costo intrinseco fosse calcolato nel momento del bisogno dal metodo astratto calcolaCostoIntrinseco, anche perché il pattern composite rendeva molto semplice un’implementazione di questo tipo.

L’altro metodo astratto che ho inserito, cambiaCostoPerNome, è pensato per permettere di cambiare il costo di un ingrediente di cui si conosce il nome, senza dover necessariamente conoscere quale è l’oggetto che lo rappresenta, oltre che per permettere di cambiare il costo a più ingredienti che hanno in comune una parte del nome, in una sola operazione.

Per il test di IngredienteBase ho verificato il corretto funzionamento delle ridefinizioni dei metodi toString, hashCode ed equals, oltre che la implementazione del metodo cambiaCostoPerNome. Per hashCode ed equals, i test sono molto semplici, e sono fatti confrontando fra di loro degli oggetti di tipo IngredienteBase di cui si conoscono le caratteristiche, per toString ho eseguito il confronto con una stringa. Per la classe IncrementaCostoPerNome ho verificato che l'incremento fosse eseguito sia se la stringa corrispondeva esattamente con il nome dell’Ingrediente, sia se la stringa era contenuta nel nome, mentre non fosse eseguito alcun aumento se anche solo parte della stringa non era contenuta nel nome. Non ho testato il metodo calcolaCostoIntrinseco perché in questo caso è un semplice return del valore del campo costoBase.

Per il test di IngredienteLavorato la strategia usata è stata molto simile a quella usata per IngredienteBase. Ho usato un getter package private che ritorna il parametro mix, e permette di aggiungere e rimuovere degli Ingredienti ad un IngredienteLavorato senza utilizzare i metodi pubblici di aggiunta e rimozione. Inoltre, in questo caso ho dovuto testare anche il metodo calcolaCostoIntrinseco in quanto è un metodo che esegue un’operazione, ho quindi testato che il calcolo fosse corretto sia usando oggetti istanziati usando il costruttore con costoProduzione, sia usando oggetti istanziati con il costruttore che lascia il valore di questa

variabile a 0. Per tutti i successivi test delle altre classi ho poi deciso di usare il costruttore che non modifica questo campo per semplicità.

Implementazione di strategy:

Successivamente ho pensato di inserire un modo per permettere al venditore di calcolare il costo finale di un ingrediente nel caso intendesse venderlo. Come prima implementazione avevo pensato ad un metodo che calcolasse il costo finale usando una percentuale di profitto che intendeva fare da quell'oggetto; tuttavia, ho poi realizzato che un venditore potesse aver bisogno a volte di vendere un oggetto senza fare alcun profitto, oppure di calcolare il profitto in modo diverso, di conseguenza ho introdotto il pattern Strategy.

Questo pattern permette di implementare facilmente diverse strategie per il calcolo del costo finale partendo da un costo intrinseco. Ho quindi definito un'interfaccia che presenta il solo metodo aggiungiProfitto che prende in input un valore double, e ritorna un valore double. Ho dato tre implementazioni di questa interfaccia, una che non applica alcun profitto, una che applica un profitto percentuale, ed un'altra che applica una certa percentuale di profitto moltiplicata per un numero di ore. Per queste tre implementazioni ho deciso di arrotondare il valore del ritorno, questo perché le implementazioni dovrebbero servire per calcolare un costo che dovrà fisicamente essere pagato dai clienti con una valuta, e di conseguenza il valore non può avere più di due cifre decimali.

Il motivo per cui ho scelto il pattern strategy è che permette di definire eventuali nuove strategie in base alle esigenze dell'utilizzatore del programma, e con questo metodo è facile, ad esempio, modificare il codice inserendo una nuova strategia che calcola il profitto utilizzando dei dati provenienti da classi che in questo momento non sono ancora state implementate.

Come classe Context ho inserito una classe chiamata Calcolatrice che ha come parametro un ProfitStrategy che viene inizializzato con un ProfitStrategy che deve essere passato al costruttore della classe. La classe presenta inoltre due altri metodi, uno che permette di cambiare strategia, ed uno che ritorna il costo finale di un Ingrediente passato come parametro, calcolato applicando la strategia scelta.

Per i test ho scelto di istanziare un oggetto della classe Calcolatrice ed un oggetto della classe IngredienteBase nel setup. A questo punto per eseguire i test vado ad impostare la calcolatrice con una determinata strategia e verifico che il valore di ritorno del metodo prezzoFinale sia corretto. Un'eccezione è il primo test, questo test serve per verificare che non ci siano errori nel calcolo del costo finale di un Ingrediente con costo zero, essendo che l'unico tipo di ingrediente a poter avere questo costo è un IngredienteLavorato, non ho utilizzato l'Ingrediente istanziato in fase di setup. Per tutti gli altri test ho utilizzato un IngredienteBase, dato che il calcolo si basa sul metodo calcolaCostoIntrinseco, che è verificato essere corretto grazie ad altri test, e non rende necessaria quindi una differenziazione fra IngredienteLavorato e IngredienteBase nei test di questa classe. Ogni strategia di profitto ha due test, uno per un valore di ritorno che non necessita arrotondamento ed uno per un valore di ritorno che deve essere arrotondato.

Creazione dello scontrino:

A questo punto quello che mancava nel programma era un modo per riuscire ad avere un elenco che idealmente doveva aggiornarsi ogni volta che un utente decideva di acquistare un Ingrediente. Inizialmente avevo pensato ad una classe concreta che definisse un metodo di aggiunta dell'ingrediente ad una un elenco ben specifico. Avevo quindi provato ad implementare una classe ScontrinoIngredienti che era ideata per produrre solo uno scontrino fisico. Tuttavia, mi sono reso conto che non avevo a disposizione alcuna informazione sul funzionamento di un'ipotetica stampante di scontrini, e soprattutto che questo "scontrino" potesse essere visto anche come uno scontrino virtuale e quindi stampabile a schermo. Ho quindi pensato ad una semplice interfaccia con due metodi, aggiungiInCoda e aggiungiInTesta, che dovrebbero servire per creare degli elenchi di stringhe con possibilità di aggiungere una stringa in cima o in fondo all'elenco.

A questo punto per poter stampare uno scontrino con l'informazione minima che uno scontrino dovrebbe possedere, ovvero il costo dell'oggetto acquistato, ho deciso di implementare un visitor. In particolare, pensando alle operazioni che i visitor concreti avrebbero dovuto eseguire ho deciso di usare la variante del visitor void. Ho implementato l'interfaccia IngredienteVisitor con i due metodi void visitIngredienteBase che prende in input un IngredienteBase, e visitIngredienteLavorato che prende in input un IngredienteLavorato, ed ho inserito come da specifica del pattern il metodo accept in Ingrediente, per poi ridefinirlo nelle sotto classi.

Per poter aggiungere correttamente allo scontrino il costo di un Ingrediente si è reso necessario utilizzare un oggetto della classe Calcolatrice, già implementata in precedenza, questo perché come spiegato nei paragrafi precedenti il costo finale di un ingrediente non è un'informazione contenuta nell'oggetto che lo rappresenta.

Le due implementazioni concrete della classe IngredienteVisitor hanno quindi bisogno di avere due attributi, uno di tipo ScontrinoIngredienti e l'altro di tipo Calcolatrice, che devono essere inizializzati con il costruttore. I due visitor concreti che ho deciso di implementare sono: un visitor che aggiunge allo scontrino il nome dell'ingrediente che visita e il costo finale, e un visitor che aggiunge allo scontrino il nome dell'ingrediente che visita e il costo finale, e ricorsivamente se l'Ingrediente visitato è un IngredienteLavorato, visita anche tutti gli oggetti contenuti all'interno di mix.

Per quanto riguarda il visitor IngredienteAggiungiAScontrinoNoDettagliVisitor ho deciso di inserire anche il metodo commonVisit, dato che l'operazione da eseguire nel caso in cui l'ingrediente sia un IngredienteBase o un IngredienteLavorato è esattamente la stessa.

Per quanto riguarda invece IngredienteAggiungiAScontrinoVisitor ho deciso di utilizzare un Iterator per poter iterare tutti gli oggetti presenti in mix. Ho quindi proceduto a modificare la classe IngredienteBase, aggiungendo un metodo iterator che ritorna semplicemente mix sotto forma di Iterator. Questo è stato fatto per far sì che il visitor non dipenda dal modo in cui mix è gestito dalla classe IngredienteLavorato, e anche per evitare che in futuro vengano create delle classi che possono modificare mix dall'esterno.

Per quanto riguarda i test dei visitor ho creato una semplice implementazione di ScontrinoIngredienti, MockCreaScontrino, che utilizza uno StringBuilder per gestire le operazioni aggiungiInTesta e aggiungiInCoda. Inoltre, ho creato due classi di test e in entrambe ho inserito nella fase di setup l'inizializzazione di:

- una Calcolatrice con la strategy ProfittoNullo,
- uno Scontrino con MockCreaScontrino
- un visitor appropriato a seconda della classe di test.

La scelta di usare una calcolatrice che utilizza la strategia con profitto nullo è stata fatta per avere più semplicità e leggibilità dei test.

Per quanto riguarda il test di AggiungiAScontrinoNoDettagliVisitor, ho eseguito dei test su IngredienteBase, aggiungendo un singolo ingrediente o più ingredienti uno dopo l'altro, e dei test su IngredienteLavorato, aggiungendo un singolo ingrediente, e degli ingredienti annidati. In ogni test sono andato a controllare se la stringa ritornata dal mock di ScontrinoIngredienti corrispondeva a quella corretta.

Per quanto riguarda il test di AggiungiAScontrinoVisitor, i test sono stati fatti in modo identico a quelli fatti per la versione del visitor senza dettagli, andando a cambiare solamente le stringhe di confronto.

Preparazione dello scontrino alla stampa:

Come ultima funzione implementata, ho pensato di inserire un modo per rendere lo scontrino effettivamente pronto alla stampa, con l'aggiunga di data ed ora, e, eventualmente uno o più messaggi di intestazione e/o uno o più messaggi come Piè di pagina.

Per farlo ho subito pensato al pattern Decorator, e in particolare ho deciso di utilizzare la variante con una classe decorator base. Ho quindi creato l'interfaccia PreparaScontrino con il metodo void prepara, che prende in input lo scontrino da decorare. Successivamente ho realizzato la prima implementazione concreta di questa classe, PreparaScontrinoData. Ho pensato di utilizzare la variante con la classe base proprio perché mi sembrava fondamentale inserire almeno la data prima di mandare in stampa lo scontrino, dato che questa deve essere sempre presente in uno scontrino.

Ho poi realizzato la classe astratta PreparaScontrinoDecorator seguendo la specifica del pattern, e le sue due implementazioni concrete PreparaScontrinoIntestazione e PreparaScontrinoPieDiPagina. Queste due implementazioni sono molto simili fra di loro, in quanto entrambe, nel costruttore, oltre che ad un oggetto di tipo PreparaScontrino, richiedono di passare anche una stringa. La differenza sta chiaramente nel fatto che nel primo caso la stringa sarà inserita in testa mentre nel secondo caso sarà inserita in coda.

La scelta di usare proprio il pattern decorator è derivata dal fatto che in questo modo è possibile aggiungere più stringhe singole in testa o in coda prima di eseguire la stampa, invece di dover usare per forza una sola stringa, cosa che, a seconda della gestione concreta della classe ScontrinoIngredienti, potrebbe essere rilevante. Inoltre, rimane possibile in futuro aggiungere facilmente ulteriori decorator che permettono di aggiungere altri tipi di informazioni, anche in questo caso potenzialmente provenienti da classi non ancora implementate, come era per il pattern Strategy.

Per quanto riguarda i test ho dovuto portare particolare attenzione nel test della data, questo perché non volevo che i test potessero fallire in relazione al fatto che ovviamente l'orario cambiava continuamente durante l'esecuzione dei test. Per essere sicuro di non avere problemi, nella fase di setup ho quindi inizializzato, oltre che uno scontrino con una stringa di test, anche un oggetto della classe Data con un orologio impostato ad un orario preciso. In questo modo eseguendo i test, avrei potuto passare questo oggetto di tipo Data, sapendo che non avrei avuto problemi dovuti al cambiamento di orario. Per il test ho utilizzato uno scontrino con una stringa di test già presente, così da evitare di dover usare altri metodi di altre classi in questa classe di test. Ho testato il corretto funzionamento della decorazione singola o di tutte le possibili combinazioni di decorazioni applicabili contemporaneamente.