Studente: Andrei Jonathan Ghergut
Matricola: 7029448

---

# Relazione del Progetto di Metodologie di Programmazione

Descrizione delle funzionalità del progetto implementato

Il progetto implementato si basa su uno shop di alimenti, legata alla vendita di bevande e cibi.
Questo shop si basa esclusivamente sulla vendita di quattro prodotti: gelati, crepes, té e limonate.
Esse sono rappresentate dalle rispettive classi: **IceCream**, **Crepes**, **Tea** e **Lemonade**.
Le quattro classi in questione hanno in comune alcuni campi, come il nome e il costo, e per questo motivo tutte e quante estendono una classe astratta **Product**.

Facendo parte di due categorie di prodotti diversi, come bevande e cibi, ogni classe implementa la rispettiva interfaccia di prodotto:

- **Beverage,** per Tea e Lemonade;
- **Food,** per IceCream e Crepes.

Lo shop, occupandosi della vendita esclusiva di questi soli prodotti, che possono essere dunque prodotti *caldi* o *freddi*, e avendo anche prodotti di famiglie diverse, come Beverage e Food, affidata la loro creazione a due factory concrete: **HotProductFactory** e **ColdProductFactory.**

Queste due classi implementano le operazioni per creare prodotti della stessa famiglia, dichiarate nella classe astratta da loro estesa, cioè **ProductFactory** .
ProductFactory dichiara due metodi per creare i prodotti, *orderBeverage()* e *orderFood()*.
Nella classe concreta HotProductFactory, orderBeverage() crea una bevanda calda, cioè un Tea, e in orderFood() crea un cibo caldo, cioè una Crepes.
Nella classe concreta ColdProductFactory, orderBeverage() crea una bevanda fredda, cioè una Lemonade, e in orderFood() crea un cibo freddo, cioè un IceCream.

Lo shop permette al cliente di lasciare una recensione sui cibi, il metodo che permette di recensire, *review(ProductPrinter product)*, è dichiarato nell'interfaccia **FoodReview**, che viene implementata da **ClientReview**, che rappresenta la recensione, che può essere *decorata* da **DataReviewDecorator** e **StarsReviewDecorator**, entrambi che ereditano i metodi della classe astratta **FoodReviewDecorator**.

I prodotti dello shop possono inoltre essere *visitati* da un **ProductVisitor**, ed attraverso un **ProductPrinter**, vengono aggiunte delle operazioni ai vari prodotti.
I due visitor concreti, **DeliveryTipPrinter** e **ProductInfoPrinter**, potendo accedere allo stato dell'oggetto, aggiungono due operazioni:

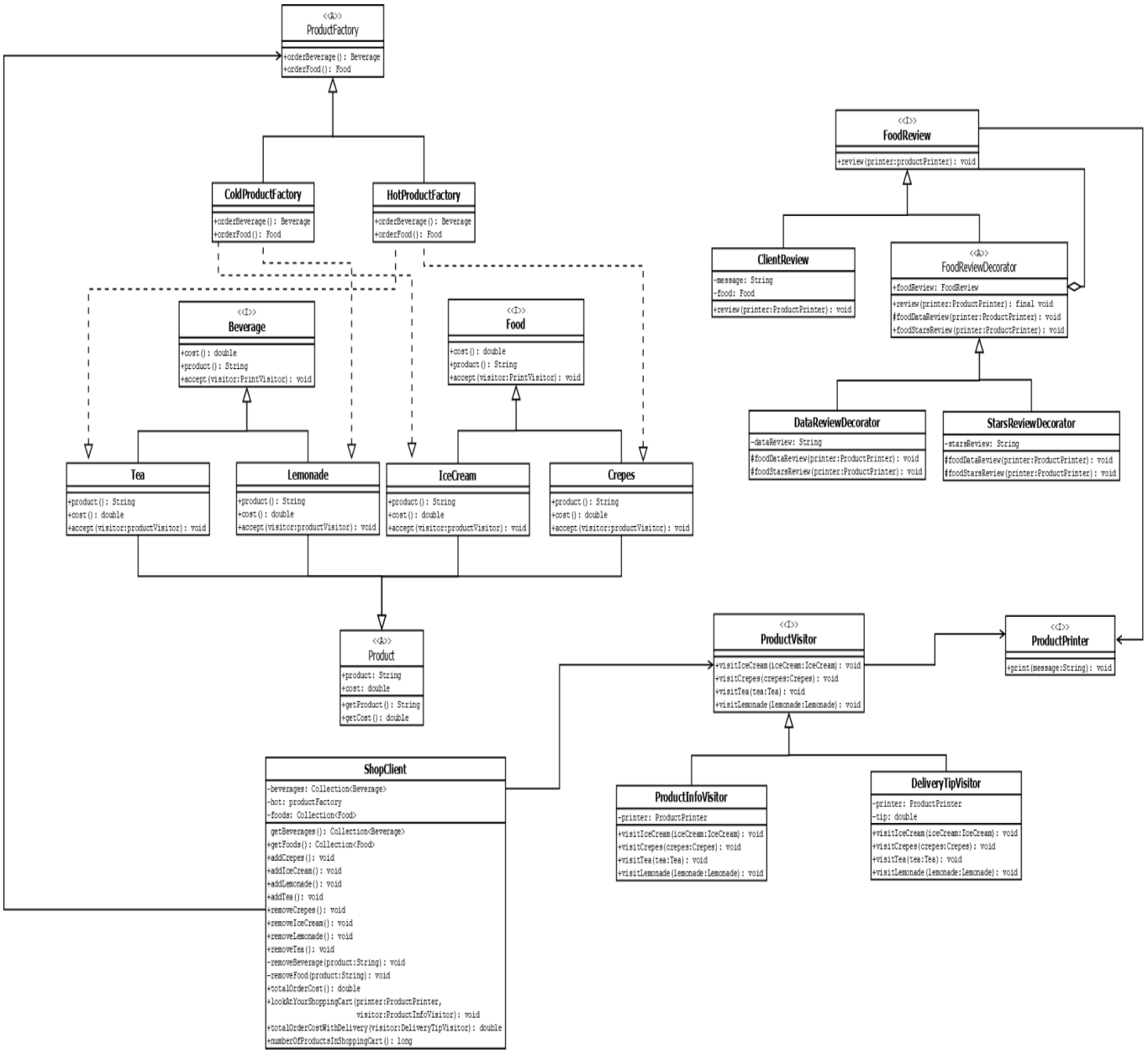
Studente: Andrei Jonathan Ghergut
Matricola: 7029448

- DeliveryTipPrinter: chiede obbilgatoriamente una mancia nel caso si usufruisca di un servizio di delivery;
- ProductInfoPrinter: stampa le informazioni riguardo il tipo e il costo del prodotto.

Infine è presente una classe **ShopClient** nella quale è possibile ordinare i vari prodotti e aggiungerli o rimuoverli dal proprio *carrello della spesa*.
La classe dispone di metodi con i quali è possibile vedere il costo totale della spesa effettuata (anche nel caso si usufruisca del servizio di delivery) , del numero di prodotti comprati, e una visione di tutti i prodotti nel proprio *carrello* attraverso il metodo *lookAtYourShoppingCart().*

Studente: Andrei Jonathan Ghergut
Matricola: 7029448

## Progettazione diagramma UML delle classi e interfacce.



Studente: Andrei Jonathan Ghergut
Matricola: 7029448

Descrizione delle scelte implementative e di design

Nell'implementazione del sistema software appena descritto, ho fatto uso di quattro design pattern: **AbstractFactory**, **Visitor**, **Decorator** e **Template method**.

**AbstractFactory**

Vedendo che gli oggetti da creare riguardavano famiglie di prodotti diversi, ho deciso di sfruttare questo pattern perché mi sembrava il più adeguato.
La scelta è data anche dal fatto che la struttura riguarda prodotti stabili che cambiano difficilmente, come richiede il pattern, visto che lo shop si basa sulla vendita di questi solo quattro prodotti.

L'AbstractFactory in questo caso è rappresentato dalla classe astratta **ProductFactory** che dichiare il metodi con i quali creare gli oggetti delle varie famiglie, attraverso dei **ConcreteFactory** rappresentati da **ColdProductFactory** e **HotProductFactory**.

Le interfacce e le implementazioni dei generi di prodotti sono rappresentate rispettivamente da: **Beverage**, **Food**, **IceCream**, **Lemonade**, **Tea** e **Crepes**.

**Visitor**

Avendo una struttura ad oggetti stabile, come già detto, e volendo aggiungere nuove operazioni, ho implementato il pattern Visitor.
Per utilizzarlo ho usato i metodi visitXXX di tipo void cosicché l'implementazione mi risultasse più chiara.
Così come i metodi visitXXX, anche gli accept con i quali ho *sporcato* le varie classi a cui volevo aggiungere le operazioni, sono void.

La dichiarazione dei vari metodi visitXXX viene fatta nell'interfaccia **ProductVisitor**, mentre i visitor concreti sono rappresentati da **DeliveryTipVisitor** e **ProductInfoProduct**.

**Decorator**

Nel progetto ho fatto uso del pattern Decorator, visto che lo shop consentiva di dare una recensione al cibo.
Ho usufruito della versione con Decorator di base, rappresentata da **FoodReviewDecorator**.
Essa presenta un riferimento all'interfaccia **FoodReview**, che mette a disposizione il metodo review() per recensire.
Tale interfaccia è implementata anche da **ClientReview**, che rappresenta la classe decorata che definisce il comportamento di base.

Studente: Andrei Jonathan Ghergut
Matricola: 7029448

Le due classi concrete che aggiungono responsabilità ad un oggetto FoodReview sono **DataReviewDecorator** e **StarsReviewDecorator**, che rappresentano la data nella quale è stata data la recensione, e le stelle recensive che un cliente vuole dare.

**Template Method**

Implementando il pattern Decorator ho fatto uso anche del Template Method rendendo final il metodo review nella classe **FoodReviewDecorator**, e aggiungendo due metodi astratti protetti, per permettere la decorazione prima e dopo, che sono stati poi ridefiniti dalle sottoclassi **DataReviewDecorator** e **StarsReviewDecorator**.