**Ministerul Educaţiei, Culturii și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Departamentul Informatică și Ingineria Sistemelor**

**RAPORT**

**Proiect de an la TMPS**

A efectuat:

st. gr. TI-203 Cuzub Maxim TI-203

A verificat:

l. univ. Mihai Gaidau

Chişinău – 2023

INTRODUCERE

Pe lângă beneficiile evidente ale utilizării patternurilor de proiectare în dezvoltarea software, acestea oferă și avantaje pe termen lung. Unul dintre aceste avantaje este îmbunătățirea scalabilității și flexibilității aplicațiilor. Prin aplicarea patternurilor de proiectare, putem crea structuri modulare și extensibile, care permit adăugarea și modificarea funcționalităților într-un mod simplu și controlat. Aceasta înseamnă că aplicația poate crește și se poate adapta mai ușor la cerințele și schimbările viitoare.

De asemenea, patternurile de proiectare promovează principii importante de proiectare software, cum ar fi separarea preocupărilor și principiul deschis-închis. Aceste principii ne ajută să construim aplicații care sunt mai ușor de înțeles, de testat și de întreținut. Ele contribuie la crearea unui cod modular și coerent, care este mai rezistent la erori și mai ușor de reutilizat în alte proiecte.

Un alt avantaj al utilizării patternurilor de proiectare este îmbunătățirea colaborării în cadrul echipei de dezvoltare. Atunci când membrii echipei folosesc un vocabular comun și urmează aceleași abordări de proiectare, comunicarea și colaborarea devin mai eficiente. Acest lucru poate duce la o productivitate crescută și la dezvoltarea unor aplicații de calitate superioară.

Pe măsură ce un dezvoltator devine familiar cu diversele patternuri de proiectare, acesta își dezvoltă și abilități de proiectare mai avansate și o înțelegere mai profundă a principiilor de proiectare software. Acest lucru îi permite să abordeze problemele complexe cu încredere și să ofere soluții bine structurate și eficiente.

Cu toate aceste avantaje, este important să menționăm că nu orice problemă sau proiect necesită aplicarea unui pattern de proiectare. Este important să evaluăm în mod critic situația și să ne asigurăm că utilizarea unui pattern este justificată și aduce valoare reală proiectului. Uneori, soluțiile simple și directe pot fi mai potrivite decât soluțiile mai complexe implicate de un pattern de proiectare.

În concluzie, utilizarea patternurilor de proiectare în dezvoltarea software aduce multiple beneficii, cum ar fi îmbunătățirea scalabilității și flexibilității aplicațiilor, respectarea principiilor de proiectare software și îmbunătățirea colaborării în echipă. Prin învățarea și aplicarea adecvată a acestor patternuri, dezvoltatorii pot crea aplicații mai eficiente, mai ușor de întreținut și de extins, contribuind la succesul proiectelor lor pe termen lung.

# DESCRIEREA PRICIPIULUI DE LUCRU AL TEMATICII ALESE

Codul dat reprezinta un "Sistem de comenzi pentru un restaurant" sau "Restaurant Order Management System".

Prezentarea patternurilor utilizate

1. Singleton Pattern

Explicarea conceptului de Singleton Pattern.

Rolul Singleton Pattern în clasa Order pentru a asigura o singură instanță a clasei și acces global la aceasta.

2. Observer Pattern

Explicarea conceptului de Observer Pattern.

Rolul Observer Pattern în clasa Order pentru a notifica utilizatorii despre modificările stării comenzii.

Descrierea relației între clasa Order (subiect observabil) și clasa User (observator).

3. Factory Pattern

Explicarea conceptului de Factory Pattern.

Descrierea interfeței FoodProductFactory și a claselor PizzaFactory și BurgerFactory ca fabrici pentru crearea de produse alimentare.

Rolul Factory Pattern în crearea obiectelor de tip FoodProduct în funcție de opțiunea selectată de utilizator.

4. Builder Pattern

Explicarea conceptului de Builder Pattern.

Descrierea claselor PizzaBuilder și BurgerBuilder pentru construirea flexibilă și extensibilă a obiectelor Pizza și Burger.

Utilizarea metodelor de setare a atributelor și a metodei build() pentru construirea finală a obiectelor.

5. Strategy Pattern

Explicarea conceptului de Strategy Pattern.

Descrierea interfeței PricingStrategy și a claselor NormalPricingStrategy și PremiumPricingStrategy ca strategii de preț și costuri de livrare.

Rolul Strategy Pattern în implementarea diferitelor strategii de preț și costuri de livrare pentru produsele alimentare.

6. Decorator Pattern

Explicarea conceptului de Decorator Pattern.

Descrierea clasei abstracte FoodProductDecorator și a clasei ExtraCheeseDecorator ca decoratoare pentru adăugarea extra-urilor la produsele alimentare.

Utilizarea Decorator Pattern pentru extinderea funcționalității produselor alimentare existente cu extra-uri.

Utilizarea codului în aplicații practice

Prezentarea diferitelor scenarii de utilizare a sistemului de comenzi pentru un restaurant.

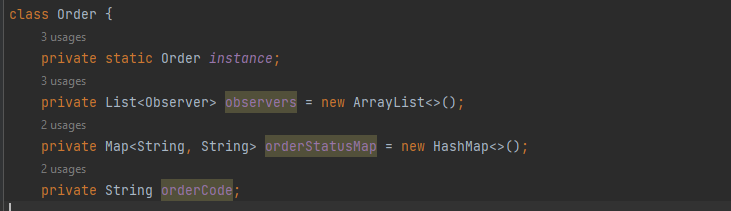
Exemple de aplicații reale în care acest cod poate fi implementat și adaptat.

Sugestii pentru extinderea și personalizarea codului în funcție de nevoile specifice ale aplicațiilor.

1. Singleton Pattern

Singleton Pattern este un design pattern care permite crearea unei singure instanțe a unei clase și asigură că există un singur punct global de acces la acea instanță. Acesta este utilizat pentru a controla crearea și accesul la o anumită clasă, astfel încât să nu existe mai multe instanțe ale sale.

În codul dat, Singleton Pattern este implementat în clasa Order. Aceasta are un constructor privat și o metodă statică getInstance() care returnează instanța clasei Order. Implementarea Singleton Pattern asigură că există o singură instanță a clasei Order și că aceasta poate fi obținută prin apelul metodei getInstance(). Astfel, orice parte a codului care dorește să lucreze cu instanța clasei Order poate obține întotdeauna aceeași instanță folosind metoda getInstance().



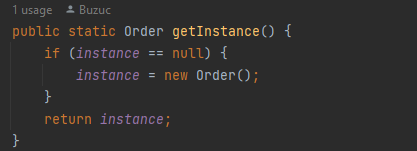
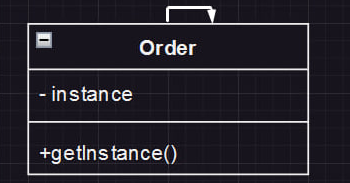


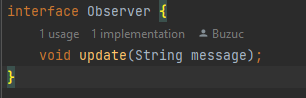
Diagrama UML

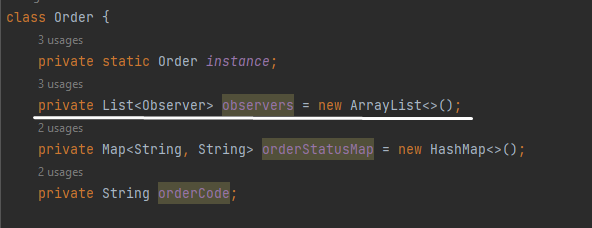


1. Observer Pattern

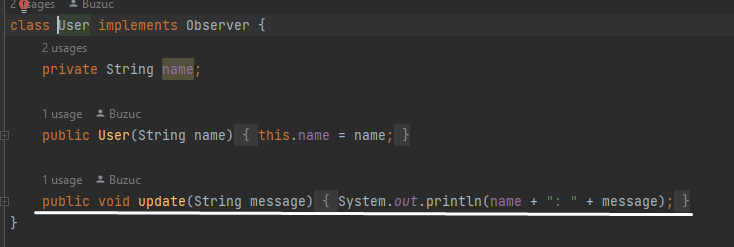
Observer Pattern este un pattern de proiectare comportamental care permite unor obiecte, numite observatori, să fie notificate automat de către un alt obiect, numit subiect observabil, atunci când acesta din urmă suferă modificări de stare.

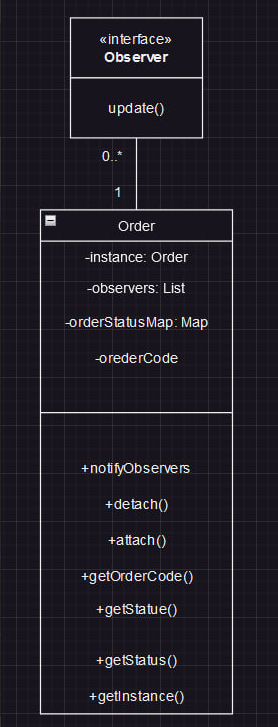
În acest bloc de cod, Observer Pattern este implementat prin intermediul interfeței Observer, clasei Order și clasei User. Clasa Order acționează ca subiect observabil și are metode pentru a atașa (attach()) și detasa (detach()) observatori, precum și pentru a notifica observatorii (notifyObservers()) despre modificările stării comenzii. Clasa User implementează interfața Observer și reprezintă un observator care primește actualizări despre starea comenzilor. În metoda update(), observatorul User afișează un mesaj cu informații despre starea comenzii.







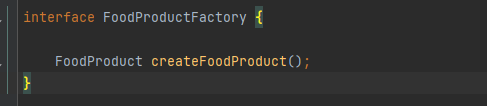


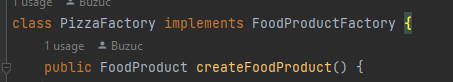
Schema UML:  


1. Factory Pattern

Factory Pattern este un pattern de proiectare creational care oferă o interfață comună pentru crearea obiectelor, dar permite subclaselor să decidă ce tip de obiect să creeze.

În acest bloc de cod, Factory Pattern este implementat prin intermediul interfeței FoodProductFactory, clasei PizzaFactory și clasei BurgerFactory. Interfața FoodProductFactory definește metoda createFoodProduct() pentru a crea obiecte de tip FoodProduct. Clasele PizzaFactory și BurgerFactory implementează această interfață și oferă implementări specifice pentru crearea de pizza și burger, respectiv.





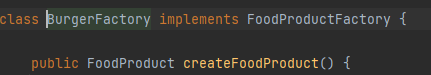
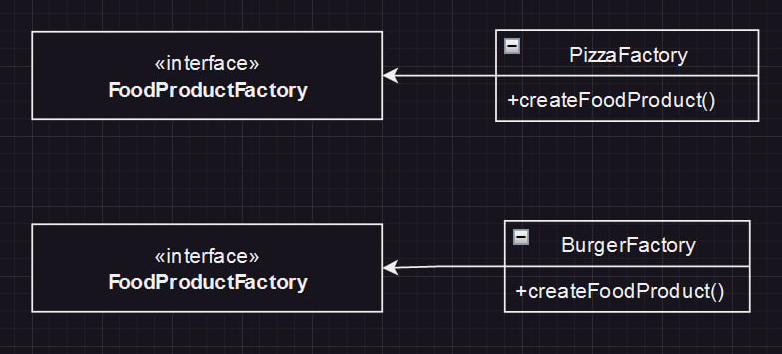


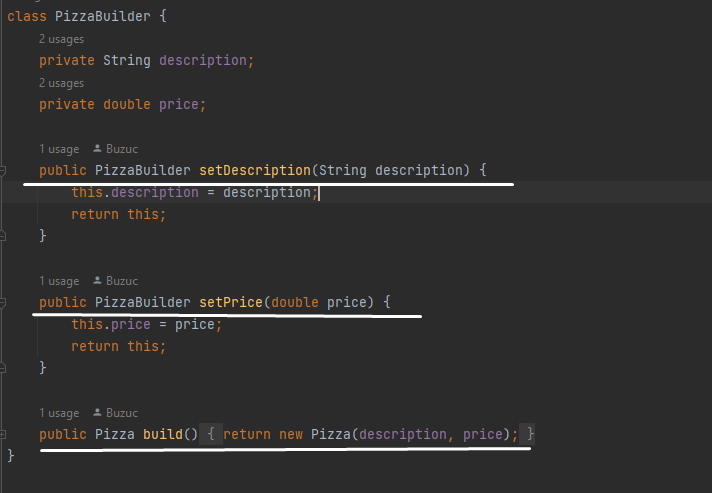
Diagrama UML

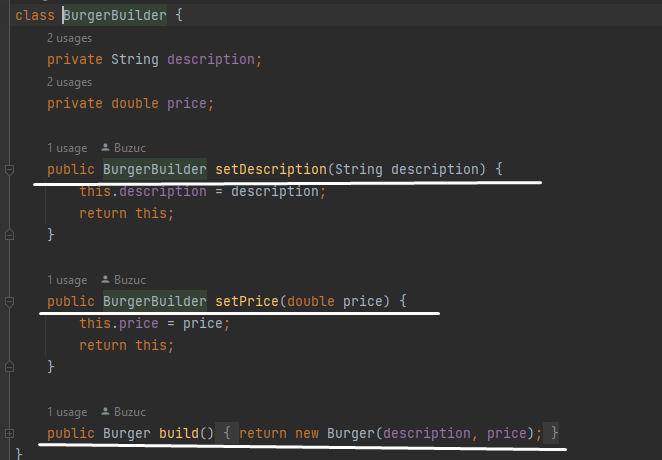


1. Builder Pattern

Builder Pattern este un pattern de proiectare creational care permite construirea unui obiect complex pas cu pas. El separă procesul de construcție a obiectului de reprezentarea sa finală și permite aceeași construcție să creeze reprezentări diferite.

În acest bloc de cod, Builder Pattern este implementat prin intermediul claselor PizzaBuilder și BurgerBuilder. Acestea oferă metode de setare a diferitelor atribute ale unui obiect (setDescription(), setPrice()) și o metodă build() pentru a construi obiectele finisate de tip Pizza și Burger cu valorile specificate.

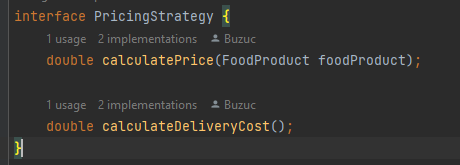


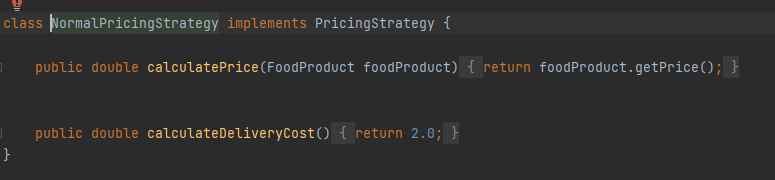


1. Strategy Pattern

Strategy Pattern este un pattern de proiectare comportamental care permite definirea unei familii de algoritmi, împachetarea fiecăruia într-o clasă separată și schimbarea lor în timpul executării. Acest pattern permite selectarea algoritmului dorit în mod dinamic, în funcție de necesități.

În acest bloc de cod, Strategy Pattern este implementat prin intermediul interfeței PricingStrategy, clasei NormalPricingStrategy și clasei PremiumPricingStrategy. Interfața PricingStrategy definește metode pentru calculul prețului unui produs alimentar (calculatePrice()) și a costului de livrare (calculateDeliveryCost()). Clasele NormalPricingStrategy și PremiumPricingStrategy implementează această interfață și oferă implementări specifice pentru calculul prețurilor și costurilor de livrare.





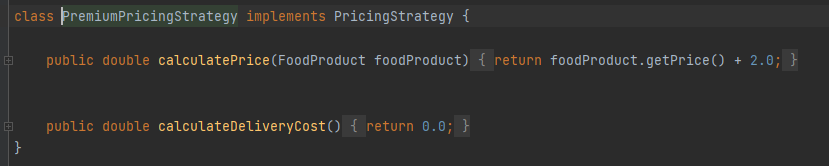
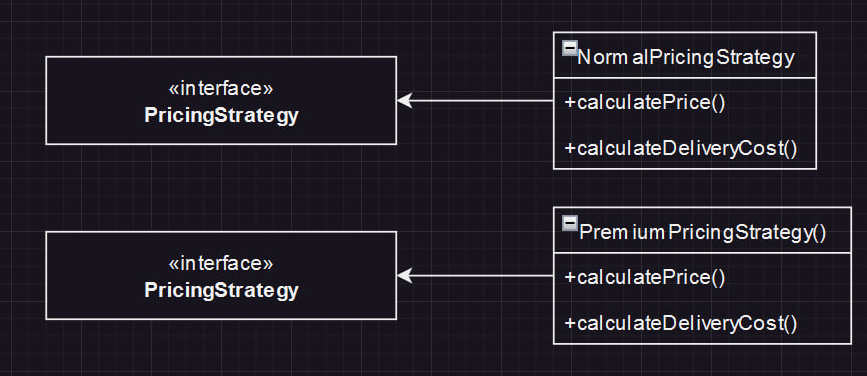


Diagrama UML



1. Decorator Pattern

Decorator Pattern este un pattern de proiectare structural care permite adăugarea de comportamente suplimentare (decoratori) la un obiect existent, fără a modifica clasa de bază. Acesta oferă o alternativă flexibilă la moștenirea multiplă pentru extinderea funcționalității unui obiect.

În codul dat, Decorator Pattern este utilizat în implementarea opțiunilor suplimentare (extra-uri) pentru produsele alimentare. Clasa abstractă FoodProductDecorator acționează ca decorator de bază, implementând interfața FoodProduct și având o referință către un obiect de tip FoodProduct decorat. Aceasta permite adăugarea de comportamente suplimentare prin intermediul claselor concrete care extind FoodProductDecorator, cum ar fi ExtraCheeseDecorator.

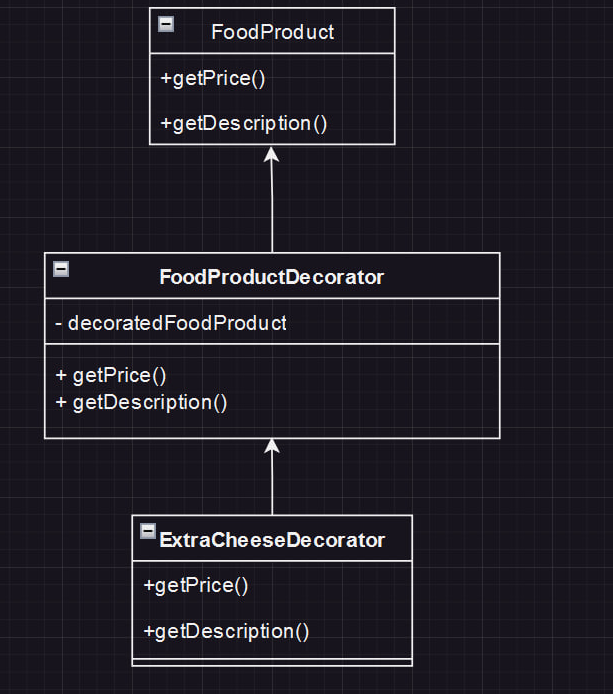




Astfel, Decorator Pattern permite decorarea unui produs alimentar cu opțiuni suplimentare, cum ar fi adăugarea de brânză extra (ExtraCheeseDecorator). Decoratorul îmbunătățește funcționalitatea produsului alimentar de bază (Pizza sau Burger) prin adăugarea de comportamente suplimentare fără a modifica codul acestuia. Acest lucru oferă flexibilitate și extensibilitate în adăugarea sau înlăturarea de opțiuni suplimentare într-un mod modular și dinamic.

În general, Decorator Pattern este util atunci când dorim să adăugăm funcționalități suplimentare la un obiect într-un mod flexibil și modular, fără a afecta structura de bază a obiectului. Acesta oferă o alternativă mai bună decât moștenirea multiplă în situațiile în care clasa de bază are multiple combinații posibile de comportamente suplimentare, permițând adăugarea sau înlăturarea acestora la runtime.

Schema UML



Concluzie :

Codul furnizat utilizează mai multe pattern-uri de proiectare pentru a rezolva diferite probleme și a îmbunătăți flexibilitatea și extensibilitatea sistemului. Iată o recapitulare a pattern-urilor de proiectare utilizate și a modului în care sunt aplicate în cod:

Singleton Pattern: Este utilizat pentru a asigura existența unei singure instanțe a clasei Order. Astfel, se controlează accesul la resursa reprezentată de comenzi și se evită crearea mai multor instanțe ale clasei.

Observer Pattern: Este folosit pentru a implementa sistemul de notificare a utilizatorilor cu privire la actualizările stării comenzilor. Clasa Order acționează ca subiect observabil, iar clasa User acționează ca observator care primește actualizări.

Factory Pattern: Se utilizează pentru crearea obiectelor de tip FoodProduct prin intermediul claselor de fabrică PizzaFactory și BurgerFactory. Astfel, se abstractizează procesul de creare a produselor alimentare și se permite extensibilitatea pentru crearea altor tipuri de produse.

Strategy Pattern: Este utilizat pentru calculul prețului total al unei comenzi și a costului de livrare, în funcție de strategia de preț selectată. Interfața PricingStrategy definește metodele de calcul al prețului și costului de livrare, iar clasele NormalPricingStrategy și PremiumPricingStrategy implementează această interfață, oferind strategii diferite de preț.

Decorator Pattern: Este folosit pentru a adăuga opțiuni suplimentare (extra-uri) la produsele alimentare. Clasa abstractă FoodProductDecorator acționează ca decorator de bază, iar clasa ExtraCheeseDecorator extinde acest decorator adăugând opțiunea de brânză extra la produsele alimentare.

Prin utilizarea acestor pattern-uri de proiectare, codul devine mai modular, mai ușor de extins și de întreținut. Ele oferă soluții bine definite și flexibile pentru problemele comune în proiectarea software-ului și promovează principiile de design solid și dezvoltare orientată pe obiecte.