Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**Proiect de Curs**

**Tema:** Gestionarea unui magazin alimentar

A efectuat: Rusu Felicia, TI-202

A verificat: asistent universitar, Gaidau Mihai,

profesor univ, Poștaru A.

Chișinău 2023

**Сuprins**

1. Introducere.........................................................................................................................3

1.1 Contextul si obiectivele proiectului......................................................................3

1.2 Descrierea tehnologiilor si design pattern-urilor utilizate.................................3-4

2. Design Pattern-uri utilizate.............................................................................................4-6

3. Descrierea arhitecturii proiectului.................................................................................6-11

4. Descrierea modelelor de design utilizate.....................................................................11-16

5Diagrame UML……………………………………………………………………….16-21

Concluzie………………………………………………………………….................... .21-22

Bibliografie………………………………………………………………......................…..23

1. **Introducere**
   1. **Contextul și obiectivele proiectului**

Proiectul de gestionare a unui magazin alimentar în JavaScript a fost dezvoltat pentru a crea o aplicație interactivă și eficientă care să permită utilizatorilor să adauge și să gestioneze produse într-un magazin virtual. Scopul acestui proiect este de a demonstra abilitățile de programare și utilizarea design pattern-urilor pentru a crea o arhitectură solidă și modulară.

Alegerea temei de gestionare a unui magazin alimentar și reprezentarea design patterns în cadrul acestei teme poate fi motivată de relevanța practică, complexitatea funcționalității, modularizare și extensibilitatea, precum și oportunitatea de învățare și înțelegere a design patterns.

Gestionarea unui magazin alimentar implică o multitudine de operațiuni, cum ar fi adăugarea și ștergerea produselor, calcularea prețurilor, gestionarea stocului, efectuarea plăților etc. Aceste funcționalități complexe pot beneficia de utilizarea design patterns pentru a împărți și organiza codul într-un mod clar și eficient.

Principalele obiective ale proiectului sunt:

* Să permită utilizatorilor să adauge produse noi în magazin prin intermediul unei interfețe intuitive.
* Să ofere utilizatorilor posibilitatea de a adăuga produse în coșul de cumpărături și de a calcula totalul de plată în funcție de cantitatea și prețul produselor.
* Să ofere opțiuni de plată flexibile, inclusiv prin card bancar sau numerar.
* Să demonstreze utilizarea design pattern-urilor creational, structural și comportamental pentru a crea o arhitectură robustă și ușor de întreținut.
  1. **Descrierea tehnologiilor si design pattern-urilor utilizate**

Design Patterns numite si modele de design sunt soluții de design pentru rezolvarea problemelor in cadrul proiectului. Acestea arată interacțiunile si relațiile dintre clase și obiecte. Design Patterns sunt utilizate in limbaje de programare OOP, precum Java, C#, JavaScript, C si altele. În cadrul acestui proiect s-a utilizat limbajul JavaScript în care s-a reprezentat gestiunea unui magazin alimentar. Exista peste 2000 de modele de design, iar unul dintre principalele moduri de clasificare a lor este urmatorul: Creational Patterns, Structural Patterns si Behavioral Patterns.

Design pattern-uri creational:

a. Factory Pattern: utilizat pentru a crea obiecte de produs într-un mod flexibil și modular.

b. Singleton Pattern: utilizat pentru a asigura că există o singură instanță a coșului de cumpărături în întreaga aplicație.

Design pattern-uri structurale:

a. Decorator Pattern: utilizat pentru a adăuga funcționalități suplimentare produselor, cum ar fi adăugarea unui adaos pentru produsele organice.

b. Proxy Pattern: utilizat pentru a controla accesul la prețurile produselor în funcție de drepturile de utilizator.

Design pattern-uri comportamentale:

a. Observer Pattern: utilizat pentru a notifica observatorii despre modificările produselor și prețurilor acestora.

b. Strategy Pattern: utilizat pentru a permite utilizatorilor să selecteze strategia de plată preferată, fie card bancar, fie numerar.

Utilizarea acestor tehnologii și design pattern-uri permite crearea unei aplicații robuste, extensibile și ușor de întreținut, în conformitate cu principiile de proiectare software și bunele practici de dezvoltare.

1. **Design Patterns**

Design patterns reprezintă soluții generice și reutilizabile pentru probleme comune în proiectarea software-ului. Acestea sunt considerate bune practici și au fost dezvoltate și documentate de-a lungul anilor de către experți în domeniul ingineriei software. Design patterns facilitează dezvoltarea software-ului într-un mod modular, ușor de întreținut și extensibil.

Există mai multe categorii principale de design patterns, fiecare având un rol specific și o modalitate unică de funcționare. Iată o descriere detaliată a câtorva dintre aceste categorii:

Design Patterns de Creare:

Acestea se concentrează pe procesul de creare a obiectelor. Printre cele mai cunoscute design patterns de creare se numără:

Singleton - este un design pattern de creare care se concentrează pe asigurarea că o clasă are o singură instanță și furnizează un punct global de acces la aceasta. Scopul său este de a limita crearea de obiecte și de a oferi o cale simplă de acces la acea instanță unui obiect în întreaga aplicație.

În mod tradițional, implementarea unui Singleton implică definirea unei metode statice în clasa respectivă, care returnează întotdeauna aceeași instanță a clasei. Această metodă verifică dacă instanța există deja și, dacă nu, o creează și o stochează într-o variabilă statică. Astfel, la fiecare apel al metodei de acces, se va returna aceeași instanță a clasei.

Singleton este util în situațiile în care avem nevoie de o singură instanță a unei clase și dorim să avem un punct centralizat de acces la acea instanță, cum ar fi managerul de configurare sau conexiunea la baza de date.

Builder - este un model de design creațional care vă permite să construiți obiecte complexe pas cu pas. Modelul vă permite să produceți diferite tipuri și reprezentări ale unui obiect folosind același cod de construcție.

Prototype - este un model de design creațional care vă permite să copiați obiectele existente fără a face codul să depindă de clasele lor.

Factory Method - este un design pattern de creare care oferă o interfață pentru crearea de obiecte, dar lasă subclasele să decidă clasa concretă a obiectului care va fi creată. Scopul său este de a permite o flexibilitate mai mare în crearea de obiecte, ascunzând logica specifică de creare în subclase.

Pentru a implementa Factory Method, definim o interfață sau o clasă abstractă care conține metoda de fabricare. Subclasele vor extinde această clasă și vor implementa metoda de fabricare în funcție de nevoile lor specifice. Astfel, putem crea multiple implementări ale aceluiași obiect folosind interfața comună a clasei de fabrică.

Factory Method este util în situațiile în care dorim să abstractizăm procesul de creare a obiectelor și să permitem extensibilitatea și personalizarea acestora de către subclase.

Abstract Factory - este un design pattern de creare care furnizează o interfață pentru crearea unei familii de obiecte conexe, fără a specifica clasele concrete. Scopul său este de a permite crearea de obiecte asociate care funcționează împreună într-o manieră compatibilă.

Pentru a implementa Abstract Factory, definim o interfață sau o clasă abstractă pentru fabrica abstractă, care definește metode pentru crearea fiecărui tip de obiect din familia respectivă. Apoi, implementăm clase concrete de fabrică care extind fabrica abstractă și furnizează implementări concrete pentru fiecare metodă de creare.

Abstract Factory este util în situațiile în care avem nevoie de o familie de obiecte interconectate, cum ar fi crearea de interfețe grafice cu aspect consistent pe diferite platforme sau gestionarea diferitelor baze de date.

Design Patterns de Structură:

Acestea se concentrează pe relațiile și structurile dintre obiecte. Iată câteva exemple de design patterns de structură:

Adapterul - este un design pattern structural care permite cooperarea între două clase incompatibile prin convertirea interfeței uneia într-o altă interfață așteptată de client. Scopul său este de a face posibilă comunicarea între obiecte care altfel nu ar putea interacționa.

Adapterul are două componente principale: adaptorul propriu-zis și componenta adaptată. Adaptorul acționează ca o punte între client și componenta adaptată, transformând apelurile către metodele componente adaptate într-un format acceptat de client. Astfel, clientul nu trebuie să cunoască direct implementarea sau interfața componentei adaptate.

Adapterul este util în situațiile în care dorim să integrăm componente deja existente într-un sistem și să evităm modificarea codului existent sau atunci când lucrăm cu biblioteci sau servicii terțe care folosesc interfețe incompatibile cu cele ale aplicației noastre.

Decoratorul - este un design pattern structural care permite atașarea de funcționalități suplimentare la un obiect într-un mod dinamic. Scopul său este de a extinde comportamentul unui obiect fără a modifica clasa de bază.

Decoratorul utilizează compoziția și moștenirea pentru a atașa funcționalități suplimentare la un obiect. Astfel, obiectul de bază rămâne neatins, iar funcționalitățile adăugate pot fi adăugate sau îndepărtate la runtime. Decoratorul implementează aceeași interfață ca și obiectul de bază, permițându-i să fie tratat în mod transparent ca și obiectul de bază de către client.

Decoratorul este util în situațiile în care dorim să adăugăm funcționalități suplimentare unui obiect fără a crea o clasă nouă pentru fiecare combinație posibilă de funcționalități.

Composite - este un design pattern structural care definește o structură ierarhică a obiectelor într-un arbore, permițând tratamentul uniform al acestora și a compozițiilor lor. Scopul său este de a trata obiectele individuale și compozițiile acestora în mod uniform.

În cadrul modelului Composite, există două tipuri principale de obiecte: obiecte frunză (leaf) și obiecte compozit (composite). Obiectele frunză reprezintă elementele individuale ale structurii, în timp ce obiectele compozit conțin una sau mai multe instanțe ale altor obiecte (atât frunză, cât și compozit). Astfel, putem trata fiecare obiect în mod recursiv și uniform, indiferent dacă este o frunză sau o compoziție.

Modelul Composite este util în situațiile în care dorim să tratăm obiectele individuale și compozițiile acestora într-un mod uniform și să construim structuri complexe prin compunerea obiectelor simple.

Design Patterns de Comportament:

Acestea se concentrează pe interacțiunea și comunicarea dintre obiecte. Exemple de design patterns de comportament includ:

Observer - este un design pattern comportamental care permite obiectelor dependente să fie notificate automat atunci când starea obiectului observat se schimbă. Scopul său este de a realiza o comunicare slab cuplată între obiecte și de a asigura sincronizarea automată a acestora.

În cadrul acestui model, există două entități principale: subiectul și observatorii. Subiectul este obiectul observat, iar observatorii sunt obiectele care urmăresc starea acestuia. Atunci când subiectul se schimbă, toți observatorii înregistrați vor fi notificați automat și vor primi informații despre schimbare.

Observer este util în situațiile în care există o relație de dependență între obiecte, iar schimbarea stării unui obiect necesită actualizarea automată a altor obiecte.

Strategy - este un design pattern comportamental care permite definirea unei familii de algoritmi, încapsulându-le și făcându-le interschimbabile. Scopul său este de a permite schimbarea algoritmului utilizat într-un obiect în timpul execuției fără a afecta codul client.

În acest model, avem trei componente principale: contextul, strategiile și interfața comună. Contextul reprezintă obiectul care utilizează algoritmul și este configurat cu o strategie specifică. Strategiile reprezintă implementările concrete ale algoritmilor și implementează interfața comună. Prin intermediul interfeței comune, contextul poate utiliza orice strategie și poate schimba strategia în timpul execuției.

Strategy este util în situațiile în care dorim să separăm algoritmul de implementarea sa și să permitem clienților să aleagă și să schimbe algoritmul utilizat fără a afecta restul codului.

State - este un design pattern comportamental care permite unui obiect să-și modifice comportamentul în funcție de starea internă, aparent schimbându-se clasa. Scopul său este de a gestiona tranzitia între stări și de a evita logica complexă a structurii condiționale.

În cadrul acestui model, avem două componente principale: contextul și stările. Contextul reprezintă obiectul care are o anumită stare internă și reacționează în funcție de aceasta. Stările reprezintă implementările concrete ale diferitelor stări posibile și definesc comportamentul asociat fiecărei stări. Contextul utilizează o instanță de stare curentă și deleghează cererile către aceasta.

State este util în situațiile în care un obiect trebuie să prezinte comportamente diferite în funcție de starea sa internă și dorim să evităm construirea unor structuri complexe de condiționale.

Acestea sunt doar câteva exemple de design patterns, dar există și altele, fiecare având un scop specific și avantaje în diverse contexte de dezvoltare software. Utilizarea design patterns-urilor poate îmbunătăți calitatea software-ului, modularitatea și flexibilitatea acestuia, permițând dezvoltatorilor să construiască aplicații robuste și ușor de întreținut.

1. **Descrierea arhitecturii proiectului**

Proiectul ce urmează sa fie descris aici va fi despre gestionarea unui magazin alimentar. Un proiect simplu dar funcțional, in care dupa rulate utiliatorul va putea vedea interfața web cu care va putea interacționa.



Figura1 – Interfața de utilizator

În figura1 este reprezentat interfața web ce conține anumite funcționalități. Utilizatorul poate adauga produse in lista de cumpărături și să finalizeze comanda după ce alege tipul de achitare a cumpărăturii online.

Pentru a adauga un produs a fost creată funcția *addProduct().* Se inițializează trei variabile locale (name, price și quantity) cu valorile extrase din câmpurile de introducere ale formularului HTML (name, price și quantity). Se verifică dacă name nu este gol și atât price cât și quantity sunt numere valide (nu sunt NaN).

Dacă condiția este adevărată, se creează un nou produs utilizând productFactory.createProduct(name, price, quantity) și se adaugă în coșul de cumpărături folosind metoda cart.addItem(newProduct). Apoi, se apelează funcțiile displayProducts() și displayTotal() pentru a actualiza afișarea produselor și totalul în interfața utilizatorului. Astfel, funcția addProduct() extrage valorile introduce în formularul HTML, validează și adaugă un produs nou în coșul de cumpărături, iar apoi actualizează afișarea produselor și totalul.

// Funcția de adăugare a produsului

function addProduct() {

const name = document.getElementById("name").value;

const price = parseFloat(document.getElementById("price").value);

const quantity = parseInt(document.getElementById("quantity").value);

if (name && !isNaN(price) && !isNaN(quantity)) {

const newProduct = productFactory.createProduct(name, price, quantity);

cart.addItem(newProduct);

displayProducts();

displayTotal();

}

}



Figura2 – Exemplu de comandă

În această imagine vedem că in coșul de cumpărături a fost adăugat un produs, mere în care s-a selectat prețul și cantitatea produsului. Drept urmare se calculează automat prețul total care este și preț final. În imaginea care urmează va fi reprezentat rezultatul in consolă pentru comanda efectuata in figura2.

function displayTotal() {

const total = cart.getTotal();

const totalElement = document.getElementById("total");

totalElement.textContent = `Total: ${total} MDL`;

}

În această secvență de cod se calculează valoarea totală a coșului de cumpărături utilizând metoda cart.getTotal(), iar rezultatul este atribuit în variabila total. Se selectează elementul HTML cu id-ul "total" utilizând metoda document.getElementById("total"), iar rezultatul este atribuit în variabila totalElement. Se actualizează conținutul text al elementului totalElement utilizând proprietatea textContent. Se utilizează șablon de șir (template literal) pentru a formata șirul afișat, adăugând valoarea totală calculată în șirul "Total: " și sufixul " MDL".

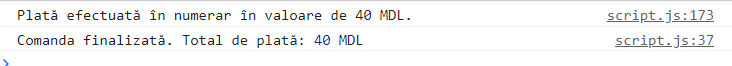


Figura3 – Rezultat consolă

Pentru cazul în care se selectează produsul, cantitatea si prețul, iar respectiv produsul nu a fost adăugat în coș si utilizatorul încearcă sa finalizeze comanda, acesta va primi următorul mesaj in consolă;



Figura4 – Coș gol

Rezultatul din figura4 este redat de funția *checkout()*, care este responsabilă de finalizarea comenzii.

function checkout() {

const total = cart.getTotal();

if (total > 0) {

const paymentMethod = document.querySelector('input[name="payment"]:checked').value;

switch (paymentMethod) {

case "card":

makePayment(total, cardPaymentStrategy);

break;

case "cash":

makePayment(total, cashPaymentStrategy);

break;

default:

console.log("Metoda de plată invalidă.");

}

cart.checkout();

displayProducts();

displayTotal();

} else {

console.log("Coșul de cumpărături este gol. Nu se poate finaliza comanda.");

}

}

Funcția începe prin calcularea valorii totale a coșului de cumpărături utilizând metoda cart.getTotal(), iar rezultatul este atribuit în variabila total. Se verifică dacă valoarea totală este mai mare decât zero, ceea ce înseamnă că există produse în coșul de cumpărături și comanda poate fi finalizată.

Se obține metoda de plată selectată de utilizator din interfață, utilizând metoda document.querySelector('input[name="payment"]:checked'). Aceasta selectează elementul de tip input de tip radio cu numele "payment" care este bifat, și se obține valoarea acestuia utilizând .value. Valoarea metodei de plată este atribuită în variabila paymentMethod. Se utilizează o declarație switch pentru a verifica valoarea paymentMethod și a apela funcția makePayment(amount, paymentStrategy) cu valoarea totală și strategia de plată corespunzătoare în funcție de metoda de plată selectată. În cazul în care valoarea paymentMethod corespunde "card" sau "cash", se va apela funcția makePayment cu strategia de plată adecvată (cardPaymentStrategy sau cashPaymentStrategy).

După efectuarea plății, se apelează metoda cart.checkout() pentru a finaliza comanda și a goli coșul de cumpărături. Apoi, se apelează funcțiile displayProducts() și displayTotal() pentru a actualiza afișarea produselor și totalul în interfața utilizatorului. În cazul în care valoarea totală a coșului de cumpărături este zero sau mai mică, se afișează un mesaj în consolă că coșul de cumpărături este gol și comanda nu poate fi finalizată. Astfel, funcția checkout() verifică valoarea totală a coșului de cumpărături și metoda de plată selectată, efectuează plata utilizând strategia corespunzătoare și finalizează comanda prin golirea coșului de cumpărături. Apoi, se actualizează afișarea produselor și totalul în interfața utilizatorului sau se afișează un mesaj de eroare în cazul în care coșul de cumpărături este gol.

Acest proiect s-a realizat in Visual Studio Code și în cadrul acestui proiect există trei fișiere: index.html, style.css și script.js.

Fișierul index.html a fost utilizat pentru a structura și afișa conținutul paginii web. Este utilzat pentru a defini structura logică, titluri, paragrafe, liste, imagini si altele. Fișierul html din acest proiect este reprezentat de următorul cod:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Document</title>

<link rel="stylesheet" type="text/css" href="style.css">

</head>

<body>

<h1>Gestionare Magazin Alimentar</h1>

<div class="product-form">

<h2>Adaugă produs</h2>

<input type="text" id="name" placeholder="Nume">

<input type="number" id="price" placeholder="Preț">

<input type="number" id="quantity" placeholder="Cantitate">

<button onclick="addProduct()">Adaugă</button>

</div>

<div class="product-list">

<h2>Produse</h2>

<ul id="products"></ul>

</div>

<div class="cart">

<h2>Coș de cumpărături</h2>

<ul id="cart"></ul>

<p id="total"></p>

<div class="payment-options">

<input type="radio" id="card" name="payment" value="card" checked>

<label for="card">Card bancar</label>

<input type="radio" id="cash" name="payment" value="cash">

<label for="cash">Numerar</label>

</div>

<button onclick="checkout()">Finalizează comanda</button>

</div>

<script src="script.js"></script>

</body>

</html>

Pentru definirea aspectului și prezentarea elementelor din pagină s-a folosit limbajul CSS. Acest limbaj ne permite sa aplicăm stiluri, culori, dimensiuni, fonturi, spațiere și altele. În continuare afișăm fișierul css creat:

body {

font-family: Arial, sans-serif;

margin: 20px;

}

h1 {

text-align: center;

}

.product-form {

margin-bottom: 20px;

}

.product-form input {

margin-right: 10px;

}

.product-list,

.cart {

width: 50%;

float: left;

}

.product-list ul,

.cart ul {

list-style-type: none;

padding: 0;

}

.cart {

clear: both;

}

button {

display: block;

margin-top: 10px;

}

JavaScript (JS) este un limbaj de programare de nivel înalt, utilizat pentru a adăuga interactivitate și funcționalitate paginilor web. JavaScript permite să se creeze și să se manipuleze elemente HTML, să se gestioneze evenimente (cum ar fi clicuri sau trageri), să se efectueze validări de formular, să se efectueze cereri asincrone către server și multe altele. JavaScript este un limbaj de scripting versatil și puternic, care oferă interactivitate și dinamism paginilor web.

1. **Descrierea modelelor de design utilizate**

În proiectul dat au fost implementare urmatoarele modele de design: Factory Pattern, Singleton Pattern, Decorator Pattern, Proxy Pattern, Observer Pattern și Strategy Pattern.

Implementarea Factory Pattern permite crearea obiectelor de tip Product printr-o fabrică specializată, abstractizând procesul de creare și oferind o modalitate flexibilă de inițializare a instanțelor.

Clasa *ProductFactory* este o fabrică de produse și implementează Factory Pattern. Metoda *createProduct* primește informații despre un produs (nume, preț, cantitate) și creează o instanță a clasei Product cu aceste informații.Acest pattern este utilizat pentru a abstractiza procesul de creare a obiectelor *Product* și pentru a oferi un mod flexibil de creare a instanțelor.

class ProductFactory {

createProduct(name, price, quantity) {

return new Product(name, price, quantity);

}

}

Utilizarea Singleton Pattern pentru obiectul Cart asigură că există o singură instanță a coșului de cumpărături în întreaga aplicație, evitând duplicarea și gestionând accesul concurent.

Clasa *Cart* implementează Singleton Pattern. Ea asigură că există o singură instanță a coșului de cumpărături în întreaga aplicație. Metoda *getInstance* returnează această instanță, iar în interiorul clasei se definesc metodele pentru adăugarea, ștergerea și afișarea produselor din coș, calcularea totalului și finalizarea comenzii. Singleton-ul *Cart* este utilizat pentru a gestiona coșul de cumpărături și pentru a oferi o singură instanță a acestuia în întreaga aplicație.

// 2. Singleton Pattern, cos de cumparaturi

class Cart {

constructor() {

if (!Cart.instance) {

this.cartItems = [];

Cart.instance = this;

}

return Cart.instance;

}

addItem(product) {

this.cartItems.push(product);

}

removeItem(product) {

const index = this.cartItems.indexOf(product);

if (index !== -1) {

this.cartItems.splice(index, 1);

}

}

getItems() {

return this.cartItems;

}

getTotal() {

let total = 0;

this.cartItems.forEach(function(item) {

total += item.getPrice() \* item.getQuantity();

});

return total;

}

checkout() {

console.log("Comanda finalizată. Total de plată:", this.getTotal(), "MDL");

this.cartItems = [];

}

}

Clasa *ProductDecorator* implementează Decorator Pattern. Aceasta primește un obiect de tip *Product* și adaugă funcționalități suplimentare prin intermediul metodelor *getPrice* și *getQuantity*, care pot fi suprascrise în clasele derivate.

Clasa *OrganicProduct* este o clasă derivată a *ProductDecorator* și reprezintă un decorator specific pentru produse organice. Aceasta suprascrie metoda *getPrice* pentru a adăuga un adaos de 10% la prețul produsului.Acest pattern este utilizat pentru a extinde funcționalitatea unui obiect *Product* fără a modifica clasa de bază.

class ProductDecorator {

constructor(product) {

this.product = product;

}

getPrice() {

return this.product.getPrice();

}

getQuantity() {

return this.product.getQuantity();

}

}

class OrganicProduct extends ProductDecorator {

constructor(product) {

super(product); //trasnmite obiectul constructorului clsei de baza

}

getPrice() { // suprascrie metoda

return this.product.getPrice() \* 1.1; // Adaugă un adaos de 10% la pretul pentru produse organice

}

}

Clasa *ProductProxy* implementează Proxy Pattern. Aceasta furnizează o interfață intermediară pentru accesarea produselor și controlează drepturile de acces. Verifică dacă utilizatorul are drepturi de administrator și returnează prețul produsului sau un mesaj de acces restricționat, în funcție de acest lucru.Acest pattern este utilizat pentru a controla și limita accesul la anumite funcționalități ale obiectelor *Product*, în funcție de drepturile utilizatorului.

// 2. Proxy Pattern

class ProductProxy {

constructor(name, price, quantity) {

this.product = new Product(name, price, quantity);

}

getPrice() {

// Verifică dacă utilizatorul are drepturi speciale pentru a afișa prețul

if (isAdmin()) {

return this.product.getPrice();

} else {

return "Acces restricționat la preț";

}

}

getQuantity() {

return this.product.getQuantity();

}

}

function isAdmin() {

// Simulăm verificarea drepturilor de administrator

return Math.random() < 0.5; // 50% șansa de a avea drepturi de administrator

}

Clasa *Product* implementează Observer Pattern. Aceasta reprezintă un produs cu un nume, preț și cantitate. Are metode pentru adăugarea, eliminarea și notificarea observatorilor atunci când se schimbă prețul sau cantitatea. Observatorii, precum clasa *PriceObserver*, pot fi adăugați pentru a fi notificați despre aceste schimbări.

Clasa *PriceObserver* este un observator specific pentru prețul produselor. Aceasta primește un produs și, atunci când este notificată, afișează un mesaj cu noul preț al produsului.

Acest pattern este utilizat pentru a realiza o comunicare între obiectele *Product* și observatorii care doresc să fie notificați atunci când se produc anumite evenimente.

// 1. Observer Pattern

class Product {

constructor(name, price, quantity) {

this.name = name;

this.price = price;

this.quantity = quantity;

this.observers = [];

}

addObserver(observer) {

this.observers.push(observer);

}

removeObserver(observer) {

const index = this.observers.indexOf(observer);

if (index !== -1) {

this.observers.splice(index, 1);

}

}

notifyObservers() { // apelează metoda update() pentru fiecare observator din vectorul observers prin utilizarea metodei forEach.

this.observers.forEach(function(observer) {

observer.update();

});

}

getPrice() {

return this.price;

}

getQuantity() {

return this.quantity;

}

setPrice(newPrice) {

this.price = newPrice;

this.notifyObservers();

}

setQuantity(newQuantity) {

this.quantity = newQuantity;

this.notifyObservers();

}

}

class PriceObserver {

constructor(product) {

this.product = product;

}

update() {

console.log(`Prețul produsului ${this.product.name} a fost actualizat: ${this.product.getPrice()} MDL`);

}

}

Clasa abstractă *PaymentStrategy* implementează Strategy Pattern pentru metoda de plată. Aceasta definește o metodă abstractă pay, care trebuie suprascrisă în subclasele sale.

// 2. Strategy Pattern

class PaymentStrategy {

pay(amount) {

throw new Error("Metoda trebuie suprascrisă în subclasă.");

}

}

class CardPaymentStrategy extends PaymentStrategy {

pay(amount) {

console.log(`Plată efectuată prin card bancar în valoare de ${amount} MDL.`);

}

}

class CashPaymentStrategy extends PaymentStrategy {

pay(amount) {

console.log(`Plată efectuată în numerar în valoare de ${amount} MDL.`);

}

}

Clasa *CardPaymentStrategy* este o subclasă a *PaymentStrategy* și reprezintă o strategie de plată prin card bancar. Suprascrie metoda *pay* pentru a afișa un mesaj corespunzător de plată prin card.

Clasa *CashPaymentStrategy* este o altă subclasă a *PaymentStrategy* și reprezintă o strategie de plată în numerar. Suprascrie metoda *pay* pentru a afișa un mesaj corespunzător de plată în numerar.

Funcția *makePayment* primește o sumă și o strategie de plată și efectuează plata prin intermediul strategiei respective. Aceasta apelează metoda *pay* a strategiei.

Funcția *addProduct* este apelată atunci când se dorește adăugarea unui produs nou. Extrage informațiile despre produs din elementele HTML și apelează metodele corespunzătoare ale fabricii de produse și coșului pentru a adăuga produsul și a afișa lista actualizată de produse și totalul.

Funcția *removeProduct* este apelată pentru a șterge un produs din listă. Primește indexul produsului în listă, utilizează metodele coșului pentru a înlătura produsul și apoi actualizează lista și totalul afișate.

Funcția *displayProducts* este apelată pentru a afișa lista de produse în elementul HTML corespunzător. Parcurge lista de produse din coș și creează elementele HTML corespunzătoare pentru fiecare produs, inclusiv un buton pentru ștergere. Apelează funcția *removeProduct* la apăsarea butonului de ștergere.

Funcția *displayTotal* calculează și afișează valoarea totală a coșului de cumpărături în elementul HTML corespunzător.

Funcția *checkout* este apelată pentru a finaliza comanda. Verifică dacă există produse în coș, extrage metoda de plată selectată din elementele HTML și efectuează plata prin intermediul strategiei corespunzătoare. Apoi finalizează comanda prin apelarea metodei *checkout* a coșului și actualizează lista de produse și totalul afișate.

1. **Diagrame UML**

În figura5 am reprezentat relatia de asociere dintre actor si use case. In exemplul dat actorul reprezinta utilizatorul, iar use case-urile ne arata actiunile pe care le poate indeplini orice utilizator in cadrul interfeței web implementate a magazinul alimentar online.

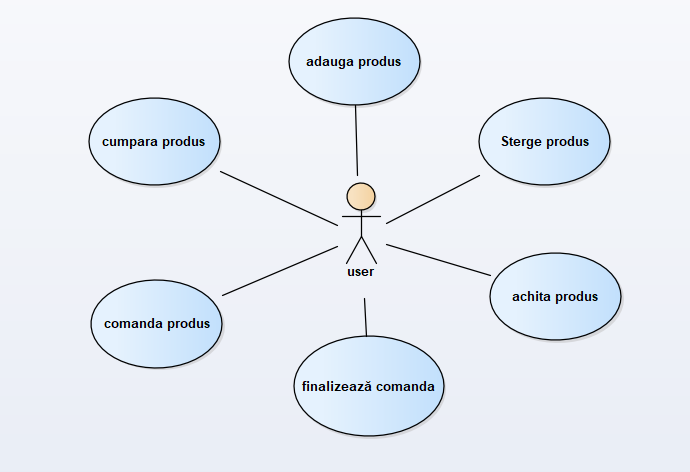


Figura5 – Funcțiile utilizatorului

Factory Pattern:

Această diagramă prezintă relația între clasa ProductFactory și clasa Product. ProductFactory este responsabilă de crearea și returnarea obiectelor de tip Product. Ea are o metodă createProduct care primește parametrii necesari pentru a crea un nou obiect Product și returnează obiectul creat. Product reprezintă clasa produsului și conține informații despre nume, preț și cantitate.

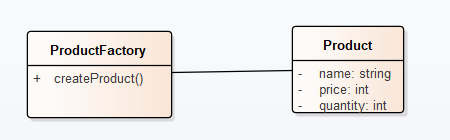


Figura6 – Factory Pattern

ProductFactory are o relație de asociere cu clasa Product, deoarece este responsabilă de crearea obiectelor de tip Product utilizând constructorul acelei clase.

ProductFactory utilizează Factory Pattern pentru a abstractiza procesul de creare a obiectelor și a furniza o interfață simplificată pentru crearea acestora.

Singleton Pattern:

Diagrama prezintă clasa Cart care implementează Singleton Pattern. Clasa Cart are o instanță statică instance și o metodă statică getInstance() care returnează această instanță. Astfel, este asigurat faptul că există o singură instanță a clasei Cart în întreaga aplicație. Clasa Cart conține metode pentru adăugarea și ștergerea de produse din coș, obținerea produselor din coș, calcularea totalului și finalizarea comenzii.

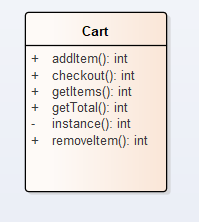


Figura7 – Singleton Pattern

Cart este clasa care implementează Singleton Pattern. Clasa Cart are o relație de asociere cu clasa Product, deoarece utilizează obiecte de tip Product pentru a le adăuga sau elimina din coș.

Apelul metodei statice getInstance() a clasei Cart asigură faptul că este creată o singură instanță a clasei și că aceasta este returnată ori de câte ori este necesară.

Decorator Pattern:

Diagrama prezintă modul în care Decorator Pattern este implementat în cadrul claselor ProductDecorator și OrganicProduct. Clasa ProductDecorator este clasa de bază pentru decoratori și conține o referință la un obiect de tip Product. Ea are metode pentru a obține prețul și cantitatea produsului. Clasa OrganicProduct este un decorator specific care extinde ProductDecorator și adaugă un adaos de 10% la prețul produsului de bază.

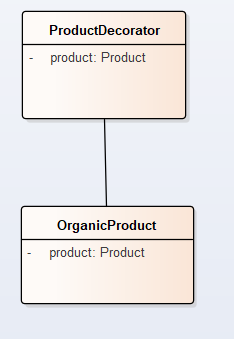


Figura8 – Decorator Pattern

Clasa ProductDecorator este clasa de bază pentru decoratori și are o relație de asociere cu clasa Product, deoarece deține o referință la un obiect Product pe care îl decorează. Clasa OrganicProduct este un decorator specific care extinde clasa ProductDecorator. Ea are o relație de asociere cu clasa Product și apelează metodele acesteia pentru a obține prețul și cantitatea, adăugând ulterior un adaos de 10% la prețul produsului.

Proxy Pattern:

Diagrama prezintă relația între clasa ProductProxy și clasa Product. Clasa ProductProxy este un proxy pentru obiectele de tip Product. Ea conține o referință la un obiect Product și implementează aceleași metode ca și Product, dar poate să controleze accesul la aceste metode. În acest exemplu, metoda getPrice() din ProductProxy verifică dacă utilizatorul are drepturi speciale pentru a accesa prețul și returnează rezultatul corespunzător.

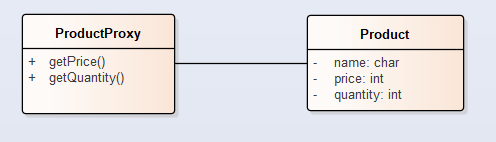


Figura9 – Proxy Pattern

Clasa ProductProxy are o relație de asociere cu clasa Product, deoarece deține o referință la un obiect Product pe care îl înlocuiește. Clasa ProductProxy utilizează Proxy Pattern pentru a controla accesul la metodele obiectului Product. În acest caz, metoda getPrice() din ProductProxy verifică drepturile utilizatorului înainte de a returna prețul.

Observer Pattern:

Diagrama prezintă relația între clasa Product și clasa Observer. Clasa Product este subiectul observat, în timp ce clasa Observer este observatorul care primește actualizări de la subiect. Clasa Product conține o listă de observatori și metode pentru adăugarea, eliminarea și notificarea observatorilor. Observatorii, cum ar fi clasa PriceObserver, implementează metoda update() care este apelată atunci când subiectul (Product) este actualizat.

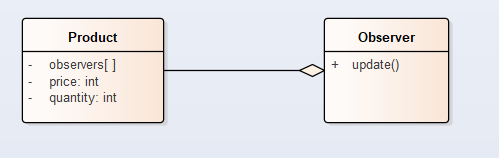


Figura10 – Observer Pattern

Clasa Product are o relație de agregare cu clasa Observer, deoarece deține o listă de observatori. Această relație permite adăugarea și eliminarea dinamice a observatorilor.

Clasa Product utilizează Observer Pattern pentru a notifica observatorii atunci când are loc o modificare. Aceasta apelează metoda update() a fiecărui observator în momentul notificării.

Strategy Pattern:

Diagrama prezintă relația între clasa abstractă PaymentStrategy și clasele concrete CardPaymentStrategy și CashPaymentStrategy. Clasa PaymentStrategy reprezintă strategia de plată și are o metodă pay(amount) pe care clasele derivate o implementează. Astfel, se pot defini diferite strategii de plată, cum ar fi plata cu cardul sau plata în numerar. Clasele concrete CardPaymentStrategy și CashPaymentStrategy extind clasa PaymentStrategy și implementează metoda pay(amount) în funcție de strategia specifică.

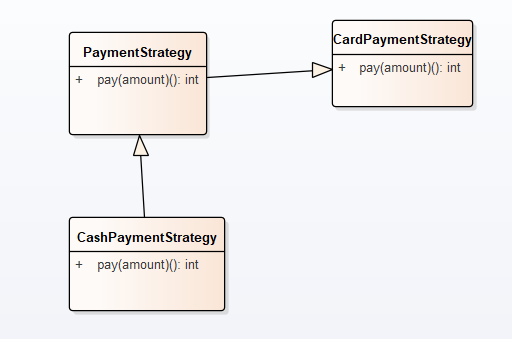


Figura11 – Strategy Pattern

Clasa PaymentStrategy este clasa abstractă pentru strategiile de plată și are o relație de generalizare cu clasele CardPaymentStrategy și CashPaymentStrategy. Apelul metodei pay(amount) este realizat prin intermediul obiectelor de tip PaymentStrategy, permițând utilizarea diferitelor strategii de plată în funcție de context. Clasele concrete CardPaymentStrategy și CashPaymentStrategy implementează metoda pay(amount) în mod specific pentru a efectua plata cu cardul sau în numerar.

Acestea sunt relațiile și interacțiunile principale între diagramele prezentate, evidențiind modul în care clasele și interfețele colaborează pentru a implementa diferite pattern-uri de design.

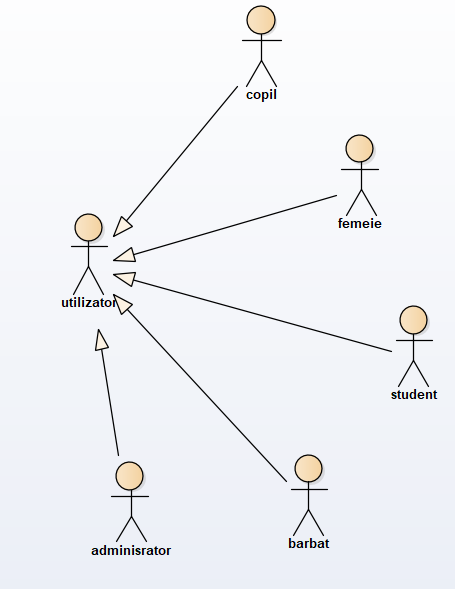


Figura12 – Tipuri de utilizatori

În figura12 am reprezentat tipuri de utilizatori care pot accesa platforma prin elementul actor. Relația utilizată în această diagramă este moștenirea, numită și generalizare. Ea indică faptul că o entitate este părinte, iar altele doua sau mai multe entitați o pot înlocui.

1. **Concluzie**

Motive pentru alegerea temei de gestionare a unui magazin alimentar și reprezentarea modelelor de design in cadrul acestei teme sunt: magazinele alimentare sunt afaceri comune și au nevoie de un sistem eficient de gestionare a produselor, coșurilor de cumpărături și a plăților. Implementarea unui sistem de acest gen poate avea beneficii practice și poate fi aplicată într-o gamă largă de aplicații reale.

Gestionarea unui magazin alimentar implică o multitudine de operațiuni, cum ar fi adăugarea și ștergerea produselor, calcularea prețurilor, gestionarea stocului, efectuarea plăților etc. Aceste funcționalități complexe pot beneficia de utilizarea design patterns pentru a împărți și organiza codul într-un mod clar și eficient. Implementarea design patterns permite separarea responsabilităților și modularizarea componentelor sistemului. Acest lucru facilitează întreținerea și extinderea ulterioară a codului. În contextul unui magazin alimentar, în care pot apărea noi cerințe și funcționalități în timp, design patterns pot ajuta la gestionarea acestor schimbări fără a afecta întregul sistem.

Temele practice, cum ar fi gestionarea unui magazin alimentar, oferă o oportunitate excelentă de a învăța și de a înțelege design patterns prin aplicarea lor într-un scenariu real. Prin reprezentarea și implementarea acestor pattern-uri în cod, se poate dobândi o înțelegere practică a modului în care funcționează și cum pot fi aplicate într-un context real.

În final putem spune că au fost implementate mai multe pattern-uri de design în diferite funcționalități ale unei aplicații de gestionare a coșului de cumpărături. Fiecare pattern a fost aplicat într-un context specific pentru a rezolva anumite probleme și a oferi flexibilitate și extensibilitate în dezvoltarea aplicației.

În ceea ce privește pattern-urile creational, am implementat Factory Pattern pentru a crea obiecte de produse și Singleton Pattern pentru a asigura că există o singură instanță a coșului de cumpărături în întreaga aplicație. Pattern-urile structurale au fost utilizate pentru a adăuga funcționalități suplimentare la obiectele existente. Decorator Pattern a fost folosit pentru a adăuga un adaos de preț pentru produsele organice, iar Proxy Pattern a fost utilizat pentru a restricționa accesul la prețul produselor în funcție de drepturile utilizatorului.

În ceea ce privește pattern-urile comportamentale, am implementat Observer Pattern pentru a permite observarea modificărilor prețurilor produselor și Strategy Pattern pentru a permite selecția unei strategii de plată. Aceste pattern-uri de design au adus beneficii semnificative în implementarea aplicației. Ele au permis separarea responsabilităților și modularizarea codului, astfel încât să fie ușor de înțeles, de întreținut și de extins în viitor. De asemenea, au promovat flexibilitatea și reutilizarea codului prin intermediul obiectelor și interacțiunilor lor.

În concluzie, implementarea acestor pattern-uri de design demonstrează abilitatea de a aplica principiile de proiectare software pentru a crea soluții robuste și scalabile. Utilizarea lor într-un context specific poate îmbunătăți calitatea, flexibilitatea și extensibilitatea unei aplicații.

1. **Bibliografie**
2. Modele de design JavaScript – explicate cu exemple [ citat 07.06.23 ] – Disponibil:

<https://www.freecodecamp.org/news/javascript-design-patterns-explained/>

1. Design patterns [ citat 07.06.23 ] – Disponibil:

<http://elf.cs.pub.ro/poo/laboratoare/design-patterns>

1. Javascript Design Patterns [ citat 07.06.23 ] – Disponibil:

<https://www.youtube.com/results?search_query=Javascript+Design+Patterns>

1. Design-Patterns [ citat 07.06.23 ] – Disponibil:

<https://github.com/ZoranPandovski/design-patterns>

1. Diagrame UML [ citat 07.06.23 ] – Disponibil:

<http://inf.ucv.ro/~mirel/courses/I213/cursuri/Curs-05-UML-Clase-Relatii.pdf>