Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**Raport**

**Proiect de an la TMPS**

Tema : Sistem de vanzari electronic

**Realizat de : st. gr. TI-204 Dubenco Nicolae**

Chișinău 2023

Cuprins

[1.Introducere: 3](#_Toc137079925)

[2.Sistemul de cumpărături online 4](#_Toc137079926)

[2.1 Functionaliatile sistemului 4](#_Toc137079927)

[2.2 Design pattern-urile utilizate 5](#_Toc137079928)

[3. Implementare 6](#_Toc137079929)

[Concluzie 13](#_Toc137079930)

# 1.Introducere:

Proiectul propus are ca scop implementarea unui sistem simplu de coș de cumpărături online, utilizând limbajul de programare TypeScript și incorporând șapte tipuri diferite de pattern-uri de design. Pattern-urile de design sunt soluții reutilizabile pentru problemele comune de proiectare, oferind o structură clară și flexibilitate în dezvoltarea aplicațiilor. Proiectul se concentrează pe trei categorii de pattern-uri de design: pattern-uri de creare, pattern-uri structurale și pattern-uri comportamentale.

În prima parte a proiectului, am implementat două pattern-uri de creare: Factory Method și Singleton. Pattern-ul Factory Method ne permite să creăm diferite tipuri de produse în cadrul sistemului de coș de cumpărături, asigurând flexibilitate și extensibilitate. Am creat clasele Product, Book și Electronics, în care clasa Product este clasa de bază pentru celelalte două clase. Am implementat și clasa ProductFactory, care utilizează Factory Method pentru a crea obiecte de produse în funcție de tipul specificat. Pattern-ul Singleton este folosit pentru a asigura că există o singură instanță a obiectului de coș de cumpărături. Astfel, am implementat clasa ShoppingCart utilizând un model Singleton, care ne permite să lucrăm cu o singură instanță a coșului de cumpărături în întreaga aplicație.

În partea a doua a proiectului, ne-am concentrat pe pattern-uri structurale: Decorator și Adapter. Pattern-ul Decorator ne permite să adăugăm funcționalități sau comportamente suplimentare produselor individuale din coșul de cumpărături. Am creat clasa ProductDecorator și am extins-o cu clasa DiscountedProduct, care oferă un preț redus pentru produse. Acest pattern ne permite să adăugăm dinamic comportamente speciale produselor fără a le modifica structura de bază. Pattern-ul Adapter este utilizat pentru a converti interfața coșului de cumpărături într-un format sau interfață compatibilă cu sistemele externe de plată. Am creat clasa ShoppingCartAdapter, care se adaptează la interfața sistemului de plată extern și permite procesarea plăților prin intermediul acestuia.

În partea a treia a proiectului, am implementat două pattern-uri comportamentale: Observer, Iterator și Command. Pattern-ul Observer ne permite să abonăm diferite componente ale sistemului de coș de cumpărături pentru a primi actualizări atunci când se schimbă conținutul coșului. Am creat clasa ShoppingCart, care implementează funcționalitățile coșului de cumpărături și permite adăugarea sau eliminarea de produse. Am implementat și clasa CartTotalDisplay ca un observator, care primește actualizări cu privire la modificările din coș și afișează prețul total actualizat. Pattern-ul Command este utilizat pentru a încapsula comenzile ca obiecte și permite sistemului de coș de cumpărături să parametrizeze clienții cu diferite comenzi, să le înregistreze sau să le anuleze. Astfel, am implementat clasa AddItemCommand, care adaugă un produs în coșul de cumpărături și poate fi anulată prin eliminarea produsului respectiv.

Prin implementarea acestor șapte pattern-uri de design în cadrul proiectului, am demonstrat abordarea modulară, flexibilă și reutilizabilă în dezvoltarea aplicațiilor TypeScript. Utilizarea pattern-urilor de design ne permite să separăm responsabilitățile, să creăm cod mai ușor de întreținut și să extindem funcționalitățile fără a afecta alte părți ale aplicației. Proiectul sistemului de coș de cumpărături online demonstrează aplicabilitatea și avantajele pattern-urilor de design în dezvoltarea de aplicații realiste și scalabile.

# 2.Sistemul de cumpărături online

Proiectul propus are ca scop implementarea unui sistem simplu de coș de cumpărături online, utilizând limbajul de programare TypeScript și integrând diferite pattern-uri de design. Acest sistem permite utilizatorilor să adauge produse în coș, să le elimine și să afle prețul total al produselor din coș. Prin utilizarea pattern-urilor de design, se urmărește îmbunătățirea modularității, flexibilității și extensibilității sistemului.

## 2.1 Functionaliatile sistemului

Sistemul de cumpărături creat în această aplicație de consolă oferă utilizatorilor o serie de funcționalități interactive și ușor de utilizat pentru a simula experiența unui coș de cumpărături online. Cu ajutorul acestei aplicații, utilizatorii pot explora lista de produse disponibile, adăuga și elimina produse din coș, verifica conținutul coșului, aplica coduri promoționale și finaliza cumpărăturile prin procesul de checkout.

Una dintre funcționalitățile cheie ale sistemului este posibilitatea de a verifica lista de produse disponibile. Utilizatorii pot accesa această listă și pot vedea informații despre fiecare produs, cum ar fi numele și prețul acestuia. Această funcționalitate permite utilizatorilor să exploreze gama de produse și să facă alegeri informate înainte de a adăuga produse în coșul de cumpărături.

După ce utilizatorii au identificat produsele dorite, aceștia pot adăuga produse în coș folosind opțiunea corespunzătoare. Utilizatorii pot introduce numele sau codul produsului și aplicația va căuta produsul în lista disponibilă. Dacă produsul este găsit, acesta este adăugat în coș și utilizatorul primește o confirmare a adăugării.

Pentru a oferi flexibilitate și opțiuni de personalizare, utilizatorii pot elimina produse din coș folosind funcția de eliminare. Aceasta permite utilizatorilor să revizuiască și să ajusteze conținutul coșului lor în funcție de preferințele și necesitățile individuale.

Pentru a verifica conținutul coșului în orice moment, utilizatorii pot utiliza funcția de verificare a produselor din coș. Aceasta afișează o listă detaliată a produselor din coș, inclusiv numele și prețul acestora. Utilizatorii pot verifica astfel produsele adăugate și să obțină o imagine de ansamblu a valorii coșului lor de cumpărături.

O altă funcționalitate importantă a sistemului este capacitatea de a aplica coduri promoționale. Utilizatorii pot introduce un cod promoțional valabil și sistemul va verifica eligibilitatea acestuia. În cazul în care codul promoțional este valid, se aplică reducerea corespunzătoare și utilizatorul primește un mesaj de confirmare.

Pentru finalizarea procesului de cumpărături, utilizatorii pot utiliza funcția de checkout. Aceasta finalizează comanda și furnizează o sumarizare detaliată a produselor achiziționate, prețul total și informațiile necesare pentru procesul de plată.

În concluzie, aplicația de consolă pentru sistemul de cumpărături dezvoltată oferă utilizatorilor o gamă largă de funcționalități pentru a simula și gestiona experiența de cumpărături online. Prin intermediul acestei aplicații, utilizatorii pot explora, adăuga, elimina, verifica, aplica promoții și finaliza cumpărăturile, oferindu-le o experiență completă și interactivă în procesul de cumpărături.

## 2.2 Design pattern-urile utilizate

1. Singleton: Acest pattern este utilizat pentru a asigura existența unei singure instanțe a clasei ShoppingCart în întreaga aplicație. Astfel, coșul de cumpărături poate fi accesat și gestionat în mod centralizat. Singleton permite și partajarea ușoară a stării coșului între diferitele funcționalități ale aplicației.
2. Factory Method: Acest pattern este utilizat pentru a crea diferite tipuri de produse în cadrul sistemului de cumpărături. Clasa ProductFactory acționează ca un creator de produse, oferind o interfață simplă pentru crearea acestora. Astfel, noi tipuri de produse pot fi adăugate în viitor fără a afecta logica de creare existentă.
3. Decorator: Acest pattern este utilizat pentru a adăuga funcționalități suplimentare produselor din coșul de cumpărături. Clasa ProductDecorator și clasa derivată DiscountedProduct permit adăugarea de comportamente speciale pentru produse, cum ar fi reduceri de preț sau opțiuni suplimentare. Aceasta oferă flexibilitate în gestionarea modificărilor și a personalizărilor produselor fără a modifica codul existent al acestora.
4. Adapter: Acest pattern este utilizat pentru a asigura interoperabilitatea între coșul de cumpărături și un sistem extern de plată. Clasa ShoppingCartAdapter acționează ca un adaptor între interfața coșului de cumpărături și interfața sistemului de plată extern. Astfel, coșul de cumpărături poate fi integrat cu ușurință într-un sistem de plată existent, fără a modifica codul coșului de cumpărături.
5. Observer: Acest pattern este utilizat pentru a notifica diverse componente ale sistemului despre modificările în coșul de cumpărături. Clasa ShoppingCart permite adăugarea de observatori, precum CartTotalDisplay, care sunt actualizați atunci când conținutul coșului se modifică, afișând prețul total actualizat. Acest pattern facilitează sincronizarea automată a componentelor și reacționarea la evenimente în timp real.
6. Command: Acest pattern este utilizat pentru a încapsula comenzile ca obiecte și pentru a permite înregistrarea și anularea acestora. Clasa AddItemCommand reprezintă o comandă care adaugă un produs în coșul de cumpărături și poate fi anulată prin eliminarea produsului respectiv. Pattern-ul Command oferă flexibilitate în gestionarea comenzilor și permite implementarea unei funcționalități de revenire la starea anterioară a sistemului.
7. Iterator: Acest pattern este utilizat pentru a itera prin produsele din coșul de cumpărături și a verifica conținutul acestuia. Clasa ShoppingCartIterator permite parcurgerea secvențială a produselor din coș, oferind metode precum hasNext() și next(). Acest pattern facilitează parcurgerea și manipularea coșului de cumpărături într-un mod modular și eficient.

Prin utilizarea acestor design pattern-uri, sistemul de cumpărături beneficiază de o arhitectură flexibilă, modulară și ușor de întreținut. Fiecare pattern își îndeplinește rolul specific, contribuind la structura și funcționalitatea generală a sistemului.

# 3. Implementare

În cadrul procesului de implementare al sistemului electronic de cumparaturi, am aplicat o abordare bazată pe utilizarea diverselor design pattern-uri pentru a asigura o arhitectură scalabilă, modulară și ușor de întreținut. În această secțiune a raportului, vom explora în detaliu implementarea design pattern-urilor creational, structural și behavioral și modul în care acestea au contribuit la funcționalitatea și eficiența sistemului.

1. Singleton:

class ShoppingCart {

// ...

private constructor(){

//...

}

static getInstance() {

if (!this.instance) {

this.instance = new ShoppingCart();

}

return this.instance;

}

// ...

}

// Utilizare:

const shoppingCart = ShoppingCart.getInstance();

În acest fragment, metoda statică getInstance() permite obținerea unei instanțe unice a clasei ShoppingCart. Dacă o instanță nu există deja, se creează una nouă și se returnează. Astfel, asigurăm că întreaga aplicație folosește o singură instanță a coșului de cumpărături.

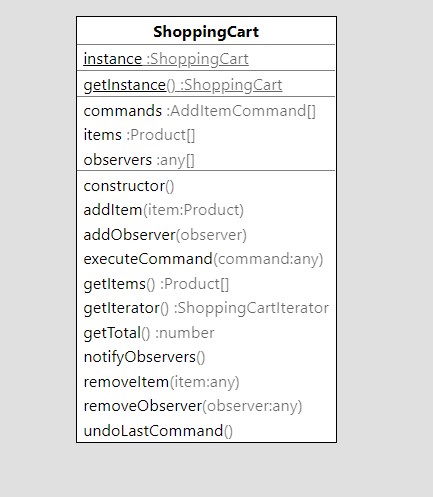


Fig.1 ShoppingCart Singleton

1. Factory

class ProductFactory {

static createProduct(type: string, name: string, price: number): Product {

if (type === "book") {

return new Book(name, price);

} else if (type === "electronics") {

return new Electronics(name, price);

}

throw new Error("Invalid product type");

}

}

// Utilizare:

const product = ProductFactory.createProduct('book', ‘Abecedar’ 10.99);

Clasa ProductFactory oferă o metodă createProduct() care returnează o instanță a clasei Product cu numele și prețul specificate. Aceasta oferă o abstracție pentru crearea produselor, permițând extinderea facilă cu noi tipuri de produse în viitor.

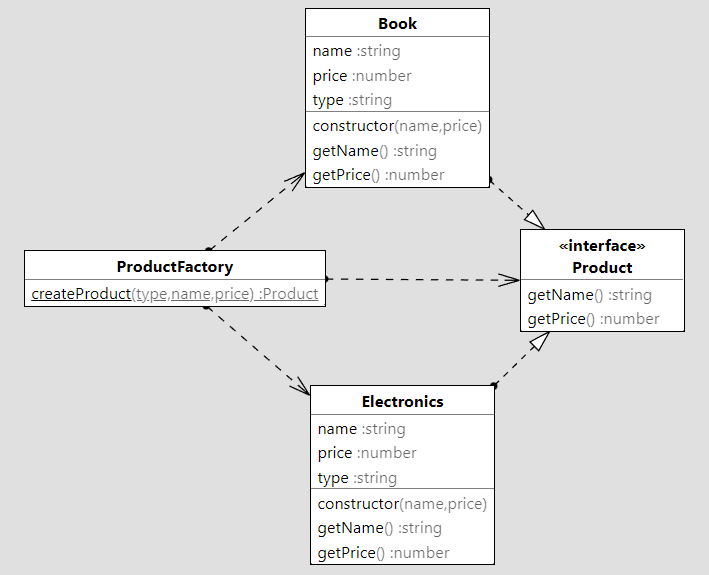


Fig.2 Diagrama patternului Factory

1. Decorator

export abstract class ProductDecorator {

abstract getName(): string ;

abstract getPrice() : number

}

export class DiscountedProduct implements ProductDecorator {

private discountedProduct: Product;

private discountPercentage: number;

constructor(product: Product, discountPercentage: number ) {

this.discountedProduct = product;

this.discountPercentage = discountPercentage;

}

getName() {

return this.discountedProduct.getName();

}

getPrice() {

const totalPrice = this.discountedProduct.getPrice();

const discountedPrice = totalPrice - totalPrice \* ( this.discountPercentage/100);

return discountedPrice;

}

}

// Utilizare:

const baseProduct = new Product('Book', 10.99);

const discountedProduct = new DiscountedProduct(baseProduct, 20);

console.log(discountedProduct.getPrice());

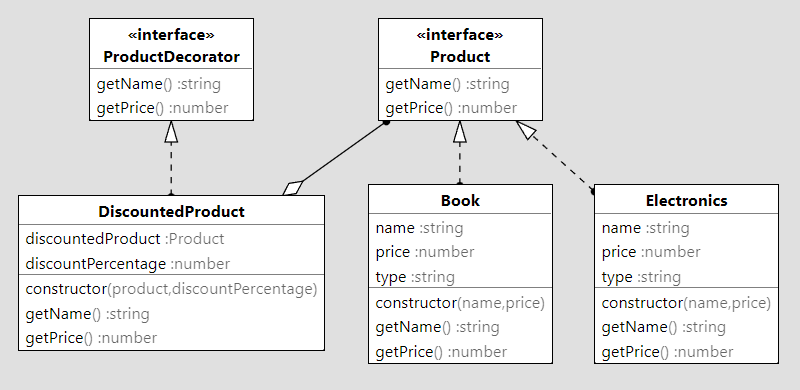


Fig.3 Diagrama patternului Decorator

Clasa ProductDecorator servește ca o clasă de bază pentru decoratori. Decoratorii, precum DiscountedProduct, extind clasa de bază și adaugă comportamente suplimentare. În exemplu, DiscountedProduct oferă un preț redus pentru un produs, aplicând o reducere specificată în constructor.

1. Adapter:

class ShoppingCartAdapter {

adapt(product: Product) {

return {

name: product.getName(),

price: product.getPrice(),

};

}

revert(adaptedProduct: any) {

return ProductFactory.createProduct(

adaptedProduct.type,

adaptedProduct.name,

adaptedProduct.price,

);

}

// Utilizare:

const produs = ProductFactory.createProduct(“book”,”Abecedar”,12.50);

const shoppingCart = new ShoppingCart();

const cartAdapter = new ShoppingCartAdapter(shoppingCart);

cartAdapter.adapt(produs);

Clasa ShoppingCartAdapter acționează ca un adaptor între interfața coșului de cumpărături (ShoppingCart) și interfața sistemului de plată extern. Prin intermediul metodei adapt(), se converteaza produsele din cos intr-un format acceptat de sistemul de plata (JSON).

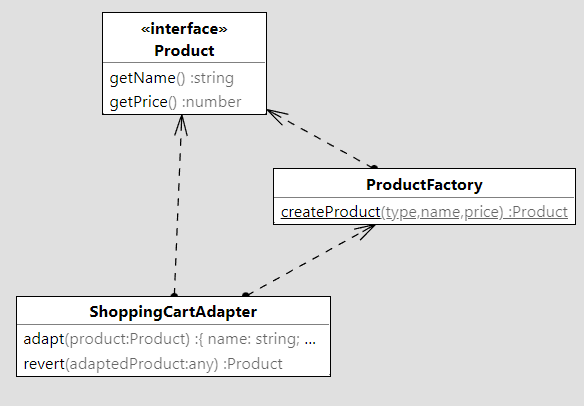


Fig.3 Diagrama patternului Adapter

1. Observer:

class ShoppingCart {

constructor() {

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() {

for (const observer of this.observers) {

observer.update(this);

}

}

// Restul codului coșului de cumpărături

}

class CartTotalDisplay {

update(cart) {

const totalAmount = cart.calculateTotal();

console.log(`Total amount: ${totalAmount}`);

}

}

// Utilizare:

const shoppingCart = new ShoppingCart();

const cartTotalDisplay = new CartTotalDisplay();

shoppingCart.addObserver(cartTotalDisplay);

shoppingCart.addProduct(new Product('Book', 10.99));

Clasa ShoppingCart conține metode pentru adăugarea, eliminarea și notificarea observatorilor. Observatorul CartTotalDisplay implementează metoda update(), care este apelată de către coșul de cumpărături atunci când acesta se modifică. În exemplu, atunci când se adaugă un produs în coș, observatorul CartTotalDisplay este notificat și afișează suma totală a coșului.

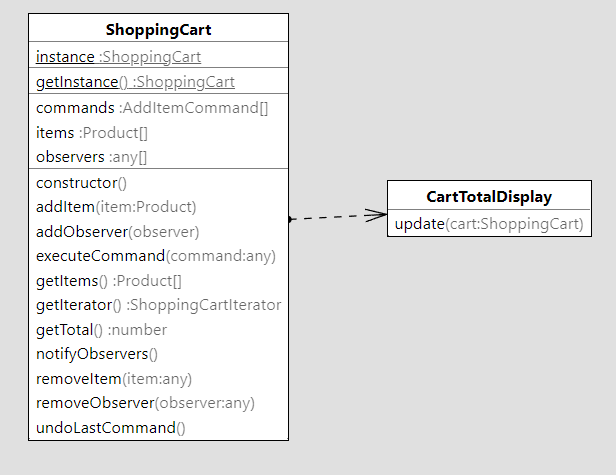


Fig.3 Diagrama patternului Observer

1. Command:

class AddItemCommand {

constructor(cart, product) {

this.cart = cart;

this.product = product;

}

execute() {

this.cart.addProduct(this.product);

}

undo() {

this.cart.removeProduct(this.product);

}

}

// Utilizare:

const shoppingCart = new ShoppingCart();

const product = new Product('Book', 10.99);

const addItemCommand = new AddItemCommand(shoppingCart, product);

addItemCommand.execute();

Clasa AddItemCommand reprezintă o comandă care adaugă un produs în coșul de cumpărături. Metoda execute() adaugă produsul în coș, în timp ce metoda undo() elimină produsul adăugat. Aceasta permite efectuarea și anularea comenzilor în sistemul de cumpărături.

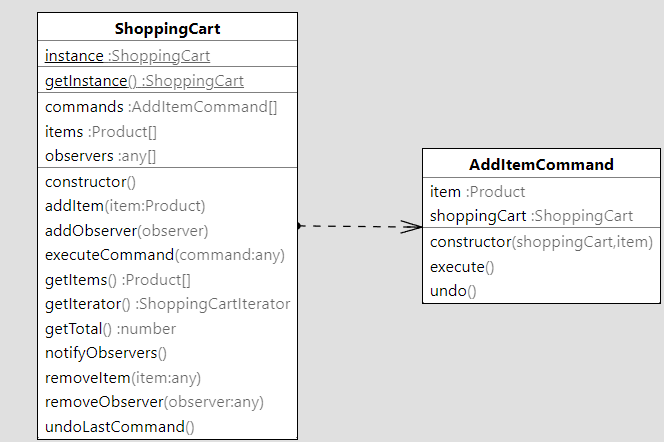


Fig.3 Diagrama patternului Command

1. Iterator

class ShoppingCartIterator {

currentIndex: number;

items: any;

constructor(items) {

this.items = items;

this.currentIndex = 0;

}

hasNext() {

return this.currentIndex < this.items.length;

}

next() {

const item = this.items[this.currentIndex];

this.currentIndex++;

return item;

}

}

// Utilizare:

const cartIterator =shoppingCart.getIterator();

while (cartIterator.hasNext()) {

const item = cartIterator.next();

console.log(`${item.getName()}: ${item.getPrice()}`);

}

Clasa ShoppingCartIterator reprezintă un iterator pentru a parcurge și verifica produsele din coșul de cumpărături. Prin utilizarea iteratorului, putem itera prin produsele din coșul de cumpărături într-un mod simplu și consistent. Acest pattern ne permite să parcurgem colecția de produse fără a fi nevoie să accesăm direct detaliile de implementare ale acesteia.

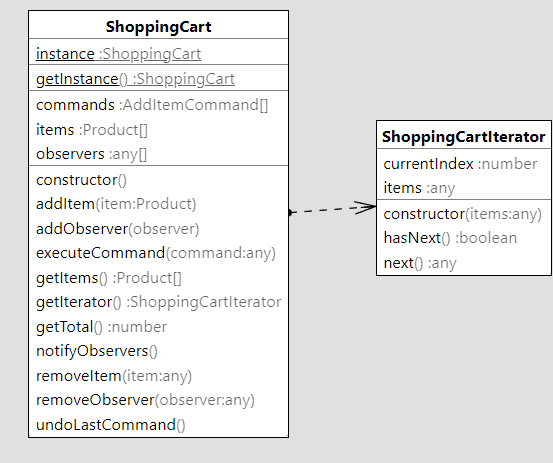


Fig.3 Diagrama patternului Iterator

# Concluzie

În concluzie, proiectul de sistem de cumpărături implementat în TypeScript a avut la bază utilizarea mai multor design pattern-uri, respectiv Singleton, Factory Method, Decorator, Adapter, Observer, Command și Iterator. Aceste design pattern-uri au oferit o structură modulară și flexibilă, îmbunătățind claritatea și extensibilitatea codului. Implementarea acestora a permis separarea responsabilităților și a permis adăugarea și modificarea funcționalităților într-un mod elegant și eficient.

Prin utilizarea pattern-urilor, am promovat principiile SOLID (Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation și Dependency Inversion), care au condus la un cod mai ușor de întreținut și de extins. Fiecare design pattern a rezolvat un set specific de probleme, permițând modularitate, reutilizabilitate și adaptabilitate.

În timpul implementării proiectului, am dobândit o experiență valoroasă în dezvoltarea aplicațiilor TypeScript, lucrând cu concepte avansate de design și arhitectură software. Am învățat cum să aplicăm design pattern-uri pentru a rezolva probleme comune și pentru a îmbunătăți calitatea codului și flexibilitatea sistemului.

De asemenea, am avut ocazia să lucrăm cu diverse concepte și funcționalități, cum ar fi gestionarea stării într-o aplicație, integrarea cu sisteme externe, actualizarea automată a componentelor, gestionarea comenzilor și iterația prin colecții de obiecte.

Experiența obținută în timpul acestui proiect ne va ajuta să abordăm cu încredere dezvoltarea de aplicații TypeScript complexe, aplicând principii și pattern-uri de design pentru a crea soluții software robuste și scalabile.