Ministerul Educației, Culturii și Cercetării

Universitatea Tehnică a Moldovei

**Raport**

Lucrarea de an

*la TMPS*



A elaborat: Dragan Dorin

Grupa: TI-203

Semnătura:\_\_\_\_\_\_\_\_\_\_\_\_

Chișinău, 2023

Cuprins

[Introducere 3](#_Toc136894991)

[Principiul SOLID 4](#_Toc136894992)

[Pattern-uri folosite 5](#_Toc136894993)

[Singleton 6](#_Toc136894994)

[Prototype 6](#_Toc136894995)

[Adapter 7](#_Toc136894996)

[Bridge 8](#_Toc136894997)

[Memento 8](#_Toc136894998)

[Observer 9](#_Toc136894999)

# **Introducere**

Această lucrare este axată pe analiza și înțelegerea unui cod de exemplu pentru o aplicație de cumpărături online. Ne propunem să ne aprofundăm în analiza codului, să deconstruim structura și funcționalitatea sa și să oferim o claritate și o perspectivă asupra modului în care diversele componente interacționează pentru a forma ansamblul sistemului. Pe măsură ce ne adâncim în cod, obiectivul nostru nu este doar să înțelegem funcționalitatea de bază, ci și să identificăm relațiile dintre părțile componente, oferind astfel o imagine a întregului sistem.

Codul prezentat este structurat meticulos în jurul unui set de clase, fiecare având un rol distinct și important în cadrul funcționalității globale a aplicației. Organizarea în clase nu doar facilitează un cod curat și ușor de înțeles, dar ajută și la crearea unei structuri modulare care încurajează reutilizarea codului și reduce complexitatea sistemului în ansamblu. Prin examinarea fiecărei clase în parte, putem dezvălui rolul pe care îl joacă în cadrul sistemului mai larg și modul în care contribuie la obiectivele globale ale aplicației.

În ceea ce privește funcționalitatea, codul prezentat oferă o gamă largă de capacități. Dincolo de facilitățile de bază, precum adăugarea de produse în coșul de cumpărături și clonarea produselor, codul prevede și procesarea plăților prin diverse sisteme. Aceasta asigură o flexibilitate maximă în acomodarea preferințelor variate ale clienților. De asemenea, codul permite afișarea detaliilor produselor în mai multe moduri, adaptându-se la nevoile specifice ale fiecărui scenariu. Salvarea și restaurarea stării coșului de cumpărături reprezintă o altă caracteristică importantă, permițând utilizatorilor să revină la o versiune anterioară a coșului de cumpărături. În plus, codul este echipat pentru a notifica utilizatorii despre evenimente specifice, îmbunătățind astfel experiența de cumpărare a clientului prin comunicarea activă și eficientă a informațiilor relevante.

# **Principiul SOLID**

* **Principiul responsabilității unice (Single Responsibility Principle):**

Fiecare clasă din acest cod are o singură responsabilitate.

Clasa ShoppingCart este responsabilă pentru gestionarea produselor din coșul de cumpărături. Se ocupă de adăugarea elementelor în coș și de recuperarea acestora.

class ShoppingCart {

addItem(item) {...}

getItems() {...}

}

* **Principiul deschis-închis (Open-Closed Principle):**

Clasa PaymentAdapter este un exemplu bun de acest principiu. Este deschisă pentru extindere, adică poate fi folosită cu orice alt sistem de plată care are o metodă processPayment, dar este închisă pentru modificare, deoarece nu trebuie să modificăm clasa în sine atunci când adăugăm un nou sistem de plată.

class PaymentAdapter {

pay(amount) {...}

}

* **Principiul substituției Liskov (Liskov Substitution Principle):**

Acest principiu nu este aplicat direct în codul dat, deoarece JavaScript este un limbaj de programare bazat pe prototipuri, nu pe clase, și nu suportă moștenirea în mod direct. Dar, în teorie, dacă am avea o subclasă a ShoppingCart sau Computer, ar trebui să poată fi folosită oriunde se folosește clasa de bază.

* **Principiul segregării interfețelor (Interface Segregation Principle):**

Deși JavaScript nu suportă interfețe în mod nativ, putem vedea un exemplu al acestui principiu în modul în care PaymentAdapter este folosit. Plățile pot fi procesate de sisteme diferite, fiecare cu propria lor metodă. În loc să se creeze o interfață masivă care să permită toate acestea, PaymentAdapter oferă o interfață simplificată (pay) care poate fi folosită indiferent de sistemul de plată.

```

class PaymentAdapter {

pay(amount) {...}

}

```

* **Principiul inversării dependențelor (Dependency Inversion Principle):**

Acest principiu este implementat prin folosirea modelului de proiectare Adapter. Clasa PaymentAdapter depinde de o abstracțiune, nu de detalii concrete. Mai degrabă decât să depindă de un anumit sistem de plată, ea depinde de existența unei metode processPayment, permițând utilizarea a orice sistem de plată care îndeplinește acest criteriu.

```

class PaymentAdapter {

pay(amount) {...}

}

```

# **Pattern-uri folosite**

Design Patterns, sunt soluții reutilizabile la problemele frecvente întâlnite în proiectarea software. Acestea nu sunt coduri sau scripturi finalizate care pot fi copiate și lipite în aplicație, ci mai degrabă ghiduri sau șabloane care arată cum să abordăm și să rezolvăm anumite probleme de proiectare.

Ele sunt bazate pe principiile de proiectare ale software-ului și pe cele mai bune practici ale industriei și sunt menite să simplifice procesul de proiectare, oferind dezvoltatorilor un vocabular comun pentru a comunica ideile mai eficient.

Pattern-urile de proiectare pot fi clasificate în trei categorii principale:

* Creational Patterns (Pattern-uri de Creare): Aceste pattern-uri se referă la modul de creare a obiectelor sau a claselor. Ele ajută la abstractizarea procesului de instantiere a obiectelor. Exemple de astfel de pattern-uri includ Singleton, Factory, Abstract Factory, Builder și Prototype.
* Structural Patterns (Pattern-uri Structurale): Aceste pattern-uri se referă la modul în care clasele și obiectele sunt compuse pentru a forma structuri mai mari. Ele asigură că, atunci când o parte a unui sistem se schimbă, întregul sistem nu trebuie să fie restructurat. Exemple de astfel de pattern-uri includ Adapter, Bridge, Composite, Decorator, Facade, Flyweight și Proxy.
* Behavioral Patterns (Pattern-uri de Comportament): Aceste pattern-uri se referă la algoritmi și la asignarea responsabilităților între obiecte. Ele nu descriu doar modele de obiecte sau de clase, ci și modele de comunicare între obiecte. Exemple de astfel de pattern-uri includ Observer, Mediator, Strategy, Chain of Responsibility, Command, State, Visitor, Template Method, Iterator și Memento.

In aceasta lucrare au fost folosite urmatorele pattern-uri:

* Singleton: Acest pattern este folosit atunci când vrem să ne asigurăm că o clasă are o singură instanță în întreaga aplicație. Singleton oferă un mecanism de acces global la acea instanță unică. Este util când un obiect necesită coordonarea activităților în întreaga aplicație.
* Prototype: Acest pattern este folosit pentru a crea obiecte prin clonarea unui prototip în loc să le creeze prin apelarea constructorului. Acesta este folosit atunci când procesul de creare a unei instanțe noi este costisitor din punct de vedere al resurselor sau al timpului.
* Adapter: Pattern-ul Adapter este utilizat pentru a asigura interoperabilitatea între două interfețe care altfel nu ar fi compatibile. Acesta este utilizat pentru a transforma interfața unei clase într-o altă interfață pe care clientul o așteaptă, permițându-le astfel să colaboreze.
* Bridge: Pattern-ul Bridge este folosit pentru a separa o abstracție de implementarea ei, astfel încât cele două pot fi modificate independent una de cealaltă. Acest pattern este util atunci când abstracțiile și implementările nu trebuie să fie legate permanent.
* Memento: Acest pattern oferă o modalitate de a salva și restaura starea unui obiect fără a expune detalii interne ale obiectului. Acesta este folosit când este necesar să se facă o copie de siguranță a stării unui obiect și să se poată reveni la acea stare la un moment ulterior.
* Observer: Acest pattern definește o dependență de tipul unu-la-multi între obiecte, astfel încât atunci când un obiect își schimbă starea, toți dependenții săi sunt notificați și actualizați. Acesta este folosit atunci când o schimbare într-un obiect necesită schimbări în alte obiecte, și nu se cunoaște numărul sau setul exact de aceste obiecte.

## **Singleton**

Pattern-ul Singleton se asigură că o clasă are doar o singură instanță și oferă un punct de acces global la aceasta. Clasa ShoppingCart este un exemplu de Singleton, deoarece asigură existența unei singure instanțe prin verificarea dacă instanța a fost deja creată înainte de a crea una nouă.



Figura 1. Schema UML pentru Singleton

class ShoppingCart {

constructor() {

if (!ShoppingCart.instance) {

this.items = [];

ShoppingCart.instance = this;

}

return ShoppingCart.instance;

}

...

}

* "ShoppingCart" este clasa noastră Singleton.
* Aceasta are două variabile private, "instance" care reține instanța Singleton, și "items", care este o listă de elemente.
* Avem un constructor public, care se asigură că există o singură instanță a clasei "ShoppingCart".
* Există două metode publice, "addItem" care adaugă un produs în coș și "getItems" care returnează toate produsele din coș.

## **Prototype**

Pattern-ul Prototype este folosit pentru a clona sau a copia obiecte existente fără a face obiectul cunoscător de ce obiect este clonat. Clasa Computer include o metodă clone() care creează o copie a instanței existente.

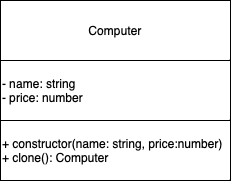


Figura 2. Schema UML pentru Prototype

class Computer {

...

clone() {

return new Computer(this.name, this.price);

}

}

* "Computer" este clasa care implementează pattern-ul Prototype.
* Aceasta are două variabile private, "name" și "price".
* Există un constructor public, care inițializează numele și prețul unui computer.
* Există și o metodă "clone", care creează și returnează o copie a obiectului "Computer". Aceasta este metoda cheie pentru pattern-ul Prototype.

## **Adapter**

Pattern-ul Adapter este folosit pentru a traduce interfața unei clase într-o interfață așteptată de client. Clasa PaymentAdapter funcționează ca un adaptor între NewPaymentSystem și OldPaymentSystem, permițându-le să lucreze împreună în ciuda diferențelor de interfață.

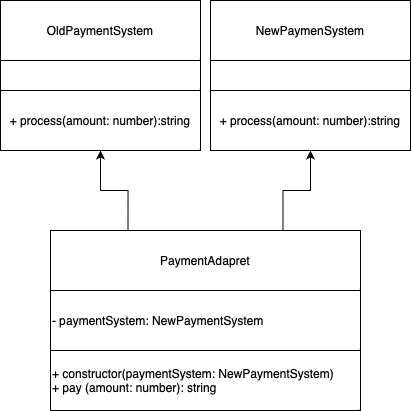


Figura 3. Schema UML pentru Adapter

class PaymentAdapter {

constructor(paymentSystem) {

this.paymentSystem = paymentSystem;

}

pay(amount) {

return this.paymentSystem.processPayment(amount);

}

}

* Clasele OldPaymentSystem și NewPaymentSystem sunt clasele existente care au metode de procesare a plăților, dar cu interfețe diferite.
* Clasa PaymentAdapter este clasa adapter care este utilizată pentru a face ca interfața NewPaymentSystem să fie compatibilă cu interfața OldPaymentSystem. Are o metodă pay() care apelează metoda processPayment() a sistemului de plată pe care îl deține (paymentSystem).

## **Bridge**

Pattern-ul Bridge desparte o abstracție de implementarea sa, astfel încât ambele pot fi variate independent. Clasele ComputerAbstraction și ComputerImplementation exemplifică acest pattern. ComputerAbstraction reprezintă abstracția, iar ComputerImplementation este implementarea specifică.

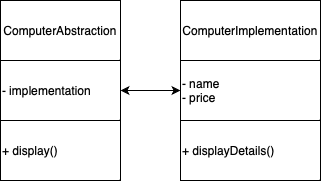


Figura 4. Schema UML pentru Bridge

class ComputerAbstraction {

constructor(implementation) {

this.implementation = implementation;

}

display() {

return this.implementation.displayDetails();

}

}

class ComputerImplementation {

...

}

* Clasa ComputerAbstraction este "abstracția" în modelul Bridge. Aceasta are o referință la o instanță a clasei ComputerImplementation (reprezentată prin linia cu diamant, semnificând o relație de agregare) și o metodă display() care apelează metoda displayDetails() a implementării.
* Clasa ComputerImplementation este "implementarea" în modelul Bridge. Aceasta conține detaliile care trebuie afișate (numele și prețul computerului) și metoda displayDetails() care returnează aceste detalii într-un format specific.

Modelul Bridge permite separarea "abstracției" (clasa ComputerAbstraction) de "implementarea" (clasa ComputerImplementation), permițându-le să varieze independent una de cealaltă.

## **Memento**

Pattern-ul Memento oferă o modalitate de a captura și a restabili starea internă a unui obiect fără a dezvălui detalii de implementare. Clasele ShoppingCartMemento și ShoppingCartCaretaker reprezintă acest pattern, salvând și restabilind starea coșului de cumpărături.

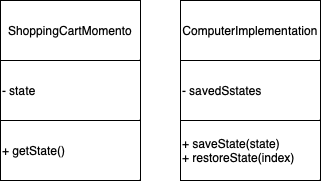


Figura 5. Schema UML pentru Momento

class ShoppingCartMemento {

constructor(state) {

this.state = JSON.parse(JSON.stringify(state));

}

getState() {

return this.state;

}

}

class ShoppingCartCaretaker {

...

}

* Clasa ShoppingCartMemento este "Memento" în acest model. Aceasta are o proprietate state care conține starea salvată a coșului de cumpărături și o metodă getState() care returnează această stare.
* Clasa ShoppingCartCaretaker este "Caretaker" în acest model. Aceasta are o proprietate savedStates care conține toate stările salvate ale coșului de cumpărături și metodele saveState(state) și restoreState(index) pentru salvarea unei noi stări și restaurarea unei stări salvate, respectiv.

Modelul Memento permite salvarea și restaurarea stărilor unui obiect fără a încălca încapsularea acestuia.

## **Observer**

Pattern-ul Observer definește o dependență unu-la-mulți între obiecte, astfel încât când un obiect se schimbă de stare, toți dependenții săi sunt notificați și actualizați automat. Clasa Shop funcționează ca un subiect care notifică observatorii săi (clasa User) atunci când se adaugă un element în coșul de cumpărături.

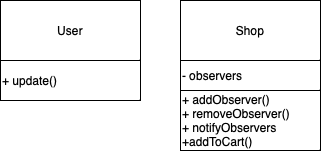


Figura 6. Schema UML pentru Observer

class Shop {

...

addObserver(observer) {

this.observers.push(observer);

}

...

notifyObservers(message) {

for (const observer of this.observers) {

observer.update(message);

}

}

...

}

class User {

update(message) {

console.log(`User notified: ${message}`);

}

}

* Clasa User este "Observer" în acest model. Aceasta are o metodă update() care este apelată atunci când un eveniment la care s-a abonat se produce.
* Clasa Shop este "Subject" în acest model. Aceasta are o listă de observers și metodele addObserver(), removeObserver(), și notifyObservers() pentru gestionarea abonaților și notificarea acestora atunci când se produce un eveniment (în acest caz, când un produs este adăugat în coș).

Modelul Observer permite unui obiect să notifice alte obiecte atunci când starea sa se schimbă.

**Concluzie**

În concluzie, codul analizat în cadrul acestei lucrări reprezintă un exemplu practic și concret de utilizare a diferitelor pattern-uri de proiectare în cadrul unei aplicații de cumpărături online. Acesta ilustrează modul în care pattern-urile de proiectare pot îmbunătăți structura, eficiența și reutilizabilitatea codului.

Analiza detaliată a fiecărui pattern de proiectare aplicat, a relevat beneficiile utilizării acestora, precum și modul în care contribuie la un design mai robust și mai flexibil. Am văzut cum pattern-urile de Singleton, Prototype, Adapter, Bridge, Memento și Observer pot fi folosite pentru a gestiona eficient coșul de cumpărături, procesarea plăților, afișarea detaliilor produselor, gestionarea stării coșului de cumpărături și notificarea utilizatorilor.

Apreciem că, în ciuda complexității pe care o presupune utilizarea acestor pattern-uri, ele aduc un aport semnificativ în ceea ce privește claritatea și modularitatea codului, facilitând astfel dezvoltarea și întreținerea acestuia pe termen lung.

Această analiză subliniază relevanța și valoarea pattern-urilor de proiectare în dezvoltarea software, evidențiind modul în care ele pot fi utilizate pentru a rezolva problemele comune cu care se confruntă dezvoltatorii în activitatea lor.