Universitatea Tehnică a Moldovei
Facultatea Calculatoare Informatică și Microelectronică
Departamentul Ingineria Software și Automatică

Proiect de an

Disciplina : Tehnici și mecanisme de proiectare a produselor program

A efectuat: Russu Iulia, TI-224

A verificat: Bîtca Ernest

Chişinău, 2025

Cuprins

Introducere

În contextul actual al dezvoltării software, cerințele pentru aplicații devin din ce în ce mai complexe și dinamice, necesitând soluții eficiente, flexibile și ușor de întreținut. O provocare majoră pentru dezvoltatori este să creeze sisteme care să poată fi extinse sau modificate cu minim efort, fără a compromite integritatea și stabilitatea codului existent. În acest sens, șabloanele de proiectare (design patterns) se dovedesc a fi instrumente esențiale pentru abordarea unor probleme recurente în arhitectura software.

Șabloanele de proiectare sunt soluții standardizate, reutilizabile și bine documentate, care oferă un set de principii și practici pentru structurarea codului și organizarea componentelor. Ele facilitează crearea unor sisteme modulare, reducând complexitatea internă prin separarea clară a responsabilităților și promovând reutilizarea componentelor. Astfel, ele permit dezvoltatorilor să evite reinventarea roții și să aplice cele mai bune practici recunoscute în industrie.

În plus, utilizarea șabloanelor asigură o comunicare mai eficientă între membrii echipei de dezvoltare, prin oferirea unui vocabular comun și a unor concepte standard. Aceasta conduce la o înțelegere mai rapidă a arhitecturii sistemului și la o colaborare mai fluidă, reducând riscul apariției erorilor și a inconsecvențelor în implementare.

Acest proiect se concentrează pe implementarea unei aplicații pentru o cafenea, în care sunt integrate cinci șabloane de proiectare: Factory, Singleton, Decorator, Observer și Strategy. Prin acest exemplu practic, se evidențiază rolul fiecărui șablon în soluționarea unor aspecte specifice ale aplicației, precum crearea obiectelor, gestionarea instanțelor unice, extinderea dinamică a funcționalităților, notificarea evenimentelor și adaptarea comportamentului în funcție de context.

Astfel, proiectul demonstrează necesitatea utilizării șabloanelor de proiectare ca metodă fundamentală pentru dezvoltarea unor aplicații software scalabile, robuste și ușor de întreținut, oferind totodată un cadru didactic valoros pentru aprofundarea cunoștințelor în arhitectura software.

Scopurile / obiectivele proiectului

Scopul acestui proiect este de a realiza o aplicație practică care să demonstreze utilizarea efectivă a unor șabloane de proiectare fundamentale în dezvoltarea software. Prin implementarea unei aplicații pentru gestionarea comenzilor într-o cafenea, se urmărește evidențierea modului în care aceste șabloane contribuie la crearea unui cod modular, flexibil și ușor de întreținut. Proiectul oferă o bază solidă pentru înțelegerea importanței design-ului orientat pe obiect și aplicarea celor mai bune practici în programare.

Obiectivele proiectului:

- Implementarea șablonului Factory pentru crearea instanțelor de băuturi, astfel încât să se izoleze logica de instanțiere de restul aplicației.

- Utilizarea șablonului Singleton pentru a asigura o gestionare unică și globală a comenzilor plasate, evitând problemele de sincronizare și duplicare a datelor.

- Aplicarea șablonului Decorator pentru extinderea funcționalității obiectelor băuturilor prin adăugarea dinamică a toppingurilor (lapte, zahăr), fără modificarea claselor de bază.

- Includerea șablonului Observer pentru notificarea automată a părților interesate în momentul plasării unei comenzi noi, facilitând decuplarea și extinderea funcționalităților.

- Implementarea șablonului Strategy pentru adaptarea calculului prețului în funcție de diferite contexte (ex. preț normal sau happy hour), permițând schimbarea dinamică a comportamentului aplicației.

- Dezvoltarea unei interfețe CLI simple și intuitive care să permită utilizatorului final să plaseze comenzi și să interacționeze cu sistemul fără dificultăți.

- Documentarea clară a arhitecturii și a modului de utilizare a fiecărui șablon, pentru a sprijini învățarea și reutilizarea în proiecte viitoare.

Proiectarea sistemului

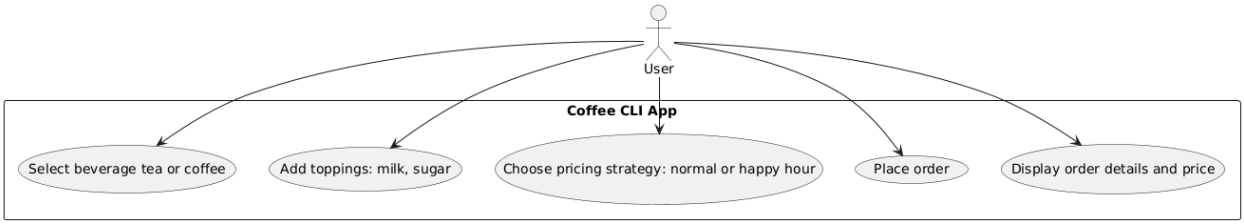

Figura 1 – Diagrama Use Case

Actor: Utilizator – interacționează cu aplicația în linia de comandă.

Use case-uri:

Selectează băutura – Apelare prin BeverageFactory (Factory Pattern)

Adaugă toppinguri – Se aplică decoratori (Decorator Pattern)

Alege strategia de preț – Se aplică o clasă de strategie (Strategy Pattern)

Plasează comanda – Se salvează în OrderManager (Singleton Pattern)

Notifică observatorii – Prin logNewOrder (Observer Pattern)

Afișează totalul – Se calculează și se afișează cu descriere + preț

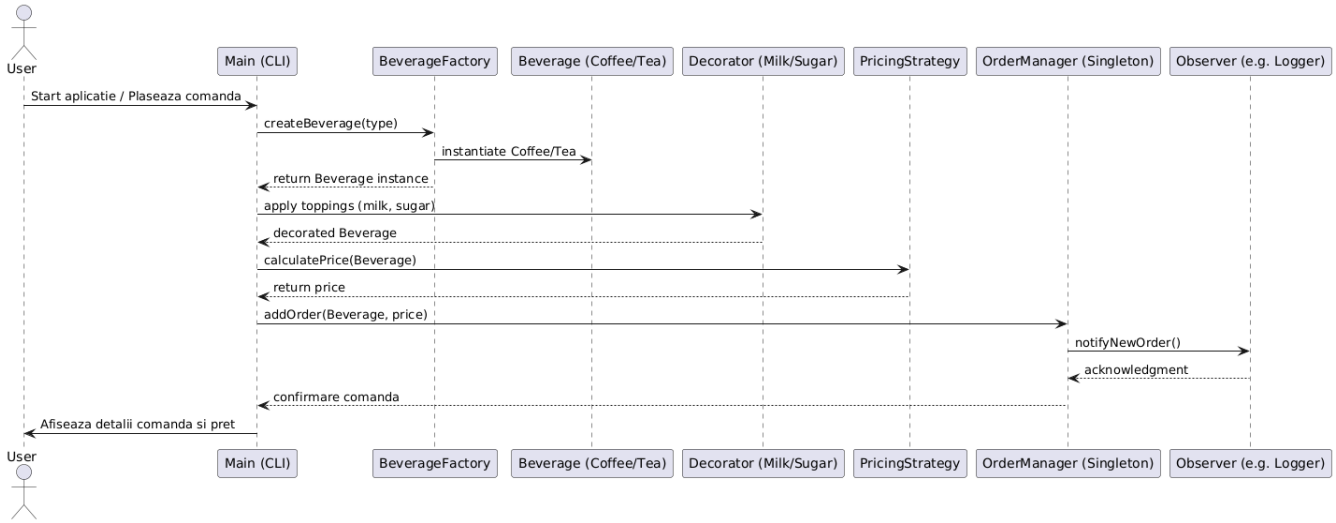

Figura 2 – Diagrama secvențială

User interacționează cu aplicația în CLI (Main).

Main folosește Factory pentru a crea băutura cerută.

Băutura este decorată cu toppinguri folosind Decorator.

Prețul este calculat prin strategia aleasă (Strategy).

Comanda este adăugată în managerul unic de comenzi (Singleton).

La adăugarea comenzii, observatorii sunt notificați (Observer).

Aplicația confirmă comanda și afișează detalii utilizatorului.

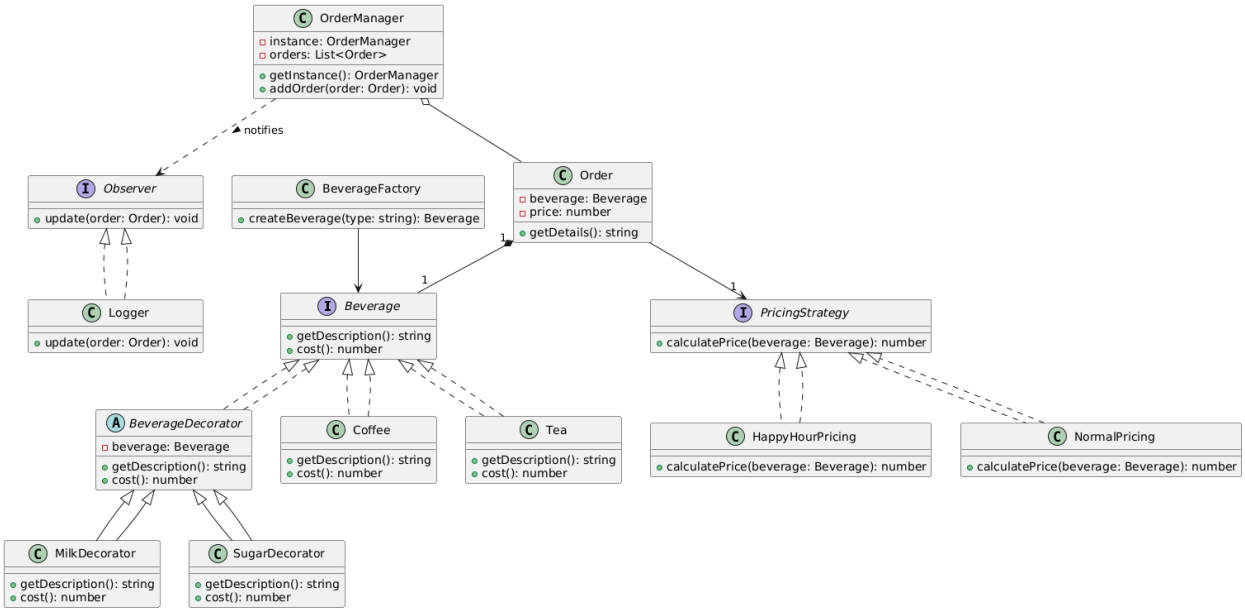

Figura 3 – Diagrama de clase

Beverage este interfața de bază pentru băuturi.
Coffee și Tea sunt implementări concrete.
BeverageDecorator este clasa abstractă pentru decoratori, iar MilkDecorator și SugarDecorator extind funcționalitatea.
BeverageFactory creează băuturi fără a expune logica de instanțiere.
PricingStrategy este interfața pentru strategii de preț, cu două implementări (normal și happy hour).
OrderManager este un Singleton care gestionează comenzile și notifică observatorii.

Observer și Logger reprezintă mecanismul de notificare.

Order leagă băutura și prețul.

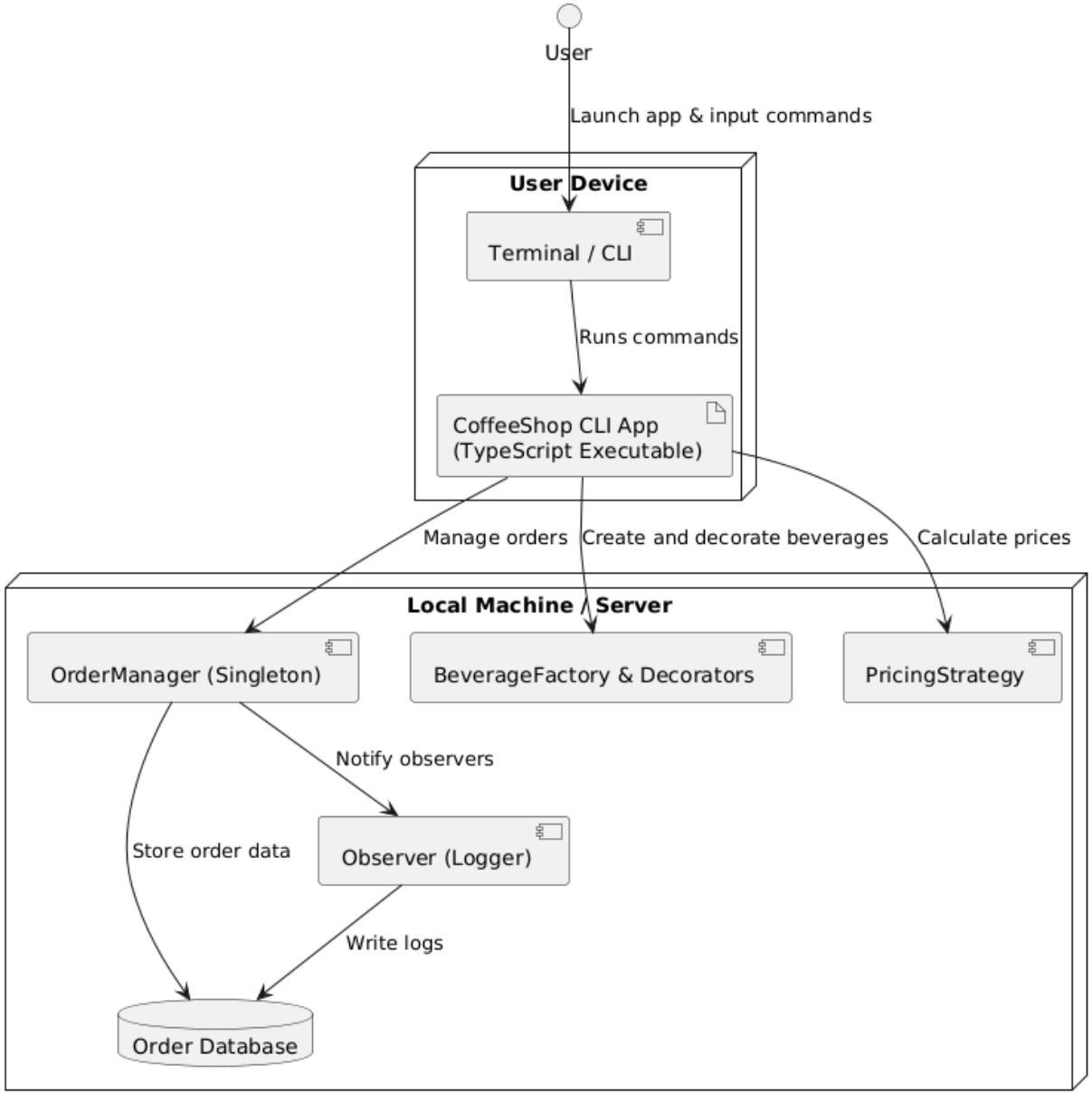

Figura 4 – Diagrama de clase

User Device: terminalul/CLI de unde utilizatorul interacționează.

CoffeeShop CLI App: codul executabil al aplicației.

Local Machine / Server: unde rulează componentele principale (logică aplicație, gestionare comenzi, strategii).

Order Database: sistemul de stocare a comenzilor. (opțional, în aplicația prezentată nu este prezent).

Observer: exemplu de componentă notificată (de ex. logger).

Implementarea

Aplicația a fost implementată folosind limbajul TypeScript, fiind structurată modular și orientată pe obiect. Scopul principal a fost integrarea a cinci șabloane de proiectare: Factory, Singleton, Decorator, Strategy și Observer, pentru a evidenția rolul fiecăruia în arhitectura generală a aplicației.

Aplicația rulează în linie de comandă (CLI), permițând utilizatorului să selecteze tipul de băutură, să adauge toppinguri, să aplice o strategie de preț și să plaseze comanda. După ce o comandă este înregistrată, observatorii sunt notificați automat (ex. un logger intern).

Structura generală:
Aplicația este organizată pe module, fiecare responsabil pentru un aspect al funcționalității:
src/models – conține clasele și interfețele pentru băuturi.
src/decorators – conține toppingurile implementate ca decoratori.
src/factory – conține logica de instanțiere a băuturilor.
src/strategy – conține strategiile de calcul al prețului.
src/singleton – conține managerul comenzilor, implementat ca Singleton.
src/observer – conține interfața și clasele observatorilor.
src/main.ts – conține interfața principală cu utilizatorul și fluxul principal al aplicației.

Utilizarea șabloanelor de proiectare

Factory Pattern
Pentru a crea instanțe de băuturi fără a depinde de clasele concrete, a fost implementată o clasă BeverageFactory. Aceasta primește un tip de băutură sub formă de string și returnează o instanță a clasei corespunzătoare (Coffee, Tea, etc.). Astfel, logica de instanțiere este centralizată și extensibilă.
Avantaje:
Separarea responsabilităților: Codul client nu trebuie să cunoască clasele concrete.
Extensibilitate: Se pot adăuga noi tipuri de băuturi fără a modifica codul client.
Centralizare: Toată logica de instanțiere este într-un singur loc.

Dezavantaje:
Dificultate în menținere: Adăugarea unui nou tip de băutură implică modificarea fabricii.
Mai puțină flexibilitate: Bazarea pe stringuri (nume de clase) poate duce la erori la runtime.
Crește complexitatea: Necesită o clasă suplimentară pentru instanțiere.

Singleton Pattern

Clasa OrderManager este responsabilă cu gestionarea comenzilor plasate. Ea este implementată ca Singleton pentru a asigura că există o singură instanță a managerului pe toată durata de rulare a aplicației. Aceasta permite gestionarea unificată a comenzilor din orice punct al aplicației.

Avantaje:
Acces global controlat: Permite accesul controlat la o singură instanță.
Consistență: Asigură o singură sursă de adevăr pentru comenzile plasate.
Economie de resurse: Evită crearea multiplă a obiectelor costisitoare.
Dezavantaje:
Testare dificilă: Singleton-urile sunt greu de mock-uit în teste unitare.
Ascundere a dependențelor: Codul poate deveni dependent de Singleton fără a o evidenția explicit.
Risc de „God object”: Singleton-ul poate ajunge să gestioneze prea multe responsabilități.

Decorator Pattern

Pentru a permite personalizarea băuturilor fără a crea subclase pentru fiecare combinație posibilă de toppinguri, s-a utilizat șablonul Decorator. Clase precum MilkDecorator și SugarDecorator adaugă comportament suplimentar unei băuturi deja create, păstrând în același timp interfața comună Beverage.

Avantaje:
Flexibilitate: Permite combinarea comportamentelor în mod dinamic.
Extensibilitate: Nu necesită modificarea claselor existente.
Respectă OCP: Se adaugă comportamente noi fără a modifica codul existent.

Dezavantaje:

Complexitate crescută: Poate duce la o structură de obiecte dificil de urmărit.

Debugging complicat: Mai multe straturi de decoratori pot îngreuna depanarea.

Configurație manuală: Trebuie compusă manual combinația dorită de decoratori.

Strategy Pattern

Calculul prețului unei comenzi se face printr-o strategie care poate fi schimbată dinamic. Aplicația include două strategii: NormalPricing și HappyHourPricing. Astfel, logica de calcul a prețului este separată de restul aplicației și poate fi modificată fără a afecta clasele existente.

Avantaje:

Separarea comportamentului: Logica de calcul este izolată de restul aplicației.

Flexibilitate: Poți schimba strategia în timpul execuției.

Testare ușoară: Strategiile pot fi testate independent.

Dezavantaje:

Creșterea numărului de clase: Fiecare strategie necesită o clasă separată.

Configurare suplimentară: Trebuie gestionată manual alegerea strategiei potrivite.

Poate părea excesivă: Pentru cazuri simple, introduce complexitate inutilă.

Observer Pattern

Pentru a notifica alte componente în momentul în care este plasată o comandă, s-a utilizat șablonul Observer. Clasa Logger este un exemplu de observator care primește notificări și înregistrează comenzile în consolă sau într-un fișier. Această abordare permite extinderea ușoară cu alți observatori (ex. notificări prin email, sincronizare cu o bază de date externă etc.).

Avantaje:

Decuplare: Emitentul nu cunoaște detalii despre observatori.

Scalabilitate: Poți adăuga observatori noi fără a modifica sursa evenimentului.

Extensibilitate: Ușor de integrat cu alte sisteme (email, baze de date, notificări).

Dezavantaje:
Ordinea notificărilor necontrolabilă: Nu este garantat ordinea în care observatorii sunt notificați.
Debugging dificil: E greu de urmărit ce se întâmplă când există mulți observatori.
Risc de scurgeri de memorie: Dacă observatorii nu sunt dezabonați corect, pot apărea probleme.

Importanța utilizării șabloanelor de proiectare în cod

În dezvoltarea software-ului modern, complexitatea sistemelor crește constant, ceea ce face ca menținerea clarității, reutilizabilității și scalabilității codului să devină o prioritate. În acest context, șabloanele de proiectare (design patterns) oferă soluții testate și general acceptate pentru probleme frecvent întâlnite în procesul de dezvoltare. Acestea reprezintă un set de bune practici care nu doar că facilitează dezvoltarea, dar și îmbunătățesc semnificativ calitatea codului și colaborarea în echipă.

Un prim beneficiu esențial al utilizării șabloanelor este reutilizarea logicii consacrate. În loc să reinventăm roata pentru probleme precum instanțierea obiectelor, gestionarea stărilor sau comunicarea între componente, șabloanele oferă structuri clare și eficiente care pot fi aplicate imediat. De exemplu, folosirea unui Factory Pattern pentru a crea obiecte permite dezvoltatorilor să decupleze codul client de clasele concrete, facilitând astfel extinderea și întreținerea sistemului.

În al doilea rând, șabloanele contribuie la îmbunătățirea arhitecturii aplicației. Prin aplicarea unor șabloane precum Strategy, Observer sau Decorator, logica aplicației devine modulară, flexibilă și ușor de extins. De exemplu, atunci când logica de calcul a prețului unei comenzi trebuie să se adapteze în funcție de context (cum ar fi orele de vârf), Strategy Pattern permite înlocuirea sau schimbarea acestei logici fără a afecta alte componente ale sistemului. Astfel, sistemul devine mai robust la schimbări, un aspect esențial în ciclul de viață al unei aplicații moderne.

Un alt aspect important este facilitarea colaborării în echipă. Atunci când o echipă

utilizează șabloane standard, membrii pot înțelege rapid structura și logica aplicației, chiar dacă nu au fost implicați în dezvoltarea inițială. Termeni precum singleton, factory, decorator sau observer devin un limbaj comun, reducând timpul de onboarding și îmbunătățind comunicarea tehnică între colegi. Mai mult, acest lucru se traduce și într-o documentație implicită: arhitectura codului este recunoscută și înțeleasă mai ușor.

Utilizarea șabloanelor are și un impact major asupra testabilității și mentenanței codului. Deoarece șabloanele încurajează separarea responsabilităților și interfețele clare, devine mai ușor să testăm unitățile individuale de cod. De exemplu, un Observer sau o strategie de pricing pot fi testate izolat, fără a fi nevoie să rulăm întreaga aplicație. Acest lucru nu doar că accelerează procesul de dezvoltare, dar și reduce numărul de erori apărute în producție.

Aplicarea unui șablon într-un context nepotrivit poate complica inutil codul și poate îngreuna înțelegerea acestuia. De aceea, un bun dezvoltator trebuie să cunoască nu doar ce face un șablon, ci și când și de ce să-l folosească. Alegerea corectă vine din experiență, dar și dintr-o înțelegere profundă a problemei de rezolvat.

Interacțiunea cu utilizatorul

Aplicația folosește biblioteca *readline-sync* pentru a primi input de la utilizator în terminal. Acesta poate selecta băutura dorită, toppingurile, tipul de preț și poate vizualiza rezultatul comenzii.

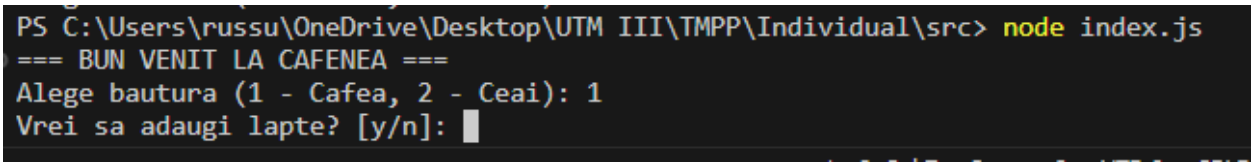

Figura 5 – Input de la utilizator

Secvențe de cod

```
// factory/BeverageFactory.ts
import { Coffee } from "../models/Coffee";
import { Tea } from "../models/Tea";
import { Beverage } from "../models/Beverage";

export class BeverageFactory {
  static createBeverage(type: string): Beverage {
    switch (type.toLowerCase()) {
      case "coffee":
        return new Coffee();
      case "tea":
        return new Tea();
      default:
        throw new Error("Invalid beverage type");
    }
  }
}
```

Secvența 1 – Factory pattern

Acest șablon abstractizează procesul de creare a băuturilor. Codul client nu trebuie să cunoască clasele concrete precum Coffee sau Tea, ci doar să solicite un tip de băutură. Astfel, se îmbunătățește scalabilitatea aplicației.

```
// singleton/OrderManager.ts
import { Order } from "./Order";
import { Observer } from "../observer/Observer";

export class OrderManager {
  private static instance: OrderManager;
  private orders: Order[] = [];
  private observers: Observer[] = [];

  private constructor() {}
```

```
  static getInstance(): OrderManager {
    if (!OrderManager.instance) {
      OrderManager.instance = new OrderManager();
    }
    return OrderManager.instance;
  }

  addObserver(observer: Observer) {
    this.observers.push(observer);
  }

  addOrder(order: Order) {
    this.orders.push(order);
    this.notifyObservers(order);
  }

  private notifyObservers(order: Order) {
    this.observers.forEach((obs) => obs.update(order));
  }
}
```

Secvența 2 – Singleton pattern

OrderManager se asigură că toate comenzile sunt gestionate centralizat și în mod sincronizat, păstrând o singură instanță în toată aplicația. Astfel, toate modulele partajează același context de lucru.

```
// decorators/MilkDecorator.ts
import { Beverage } from "../models/Beverage";

export class MilkDecorator implements Beverage {
  constructor(private beverage: Beverage) {}

  getDescription(): string {
    return this.beverage.getDescription() + ", Milk";
  }

  cost(): number {
    return this.beverage.cost() + 2.0;
  }
}
```

Secvența 3 – Decorator pattern

Prin utilizarea decorării, putem adăuga funcționalitate suplimentară (toppinguri) fără a modifica clasele existente. MilkDecorator adaugă 2 lei și actualizează descrierea băuturii.

```
// strategy/NormalPricing.ts
import { PricingStrategy } from "./PricingStrategy";
import { Beverage } from "../models/Beverage";

export class NormalPricing implements PricingStrategy {
  calculatePrice(beverage: Beverage): number {
    return beverage.cost();
```

```
  }
}

// strategy/HappyHourPricing.ts
export class HappyHourPricing implements PricingStrategy {
  calculatePrice(beverage: Beverage): number {
    return beverage.cost() * 0.8; // 20% discount
  }
}
```

Secvența 4 – Strategy pattern

Strategia de preț este aplicată dinamic în funcție de context. Se pot adăuga ușor alte strategii precum „VIPPricing”, „Reduceri sezoniere”, fără a modifica codul client.

```
// observer/Logger.ts
import { Observer } from "./Observer";
import { Order } from "../singleton/Order";

export class Logger implements Observer {
  update(order: Order): void {
    console.log("[LOG] Comanda înregistrată: ", order.getDetails());
  }
}
```

Secvența 5 – Observer pattern

Logger este un observator care este notificat când se adaugă o comandă. Șablonul Observer decuplează emiterea evenimentelor de componentele care reacționează la ele.

```
// index.ts
const beverage = BeverageFactory.createBeverage("coffee");
const withMilk = new MilkDecorator(beverage);
const withSugar = new SugarDecorator(withMilk);

const strategy = new HappyHourPricing();
const price = strategy.calculatePrice(withSugar);

const order = new Order(withSugar, price);
const manager = OrderManager.getInstance();
manager.addObserver(new Logger());
manager.addOrder(order);
```

Secvența 6 – Utilizarea în index.ts

Acest cod exemplifică integrarea tuturor șabloanelor: băutura este creată prin Factory, personalizată cu Decorators, prețul este calculat prin Strategy, comanda este gestionată de Singleton, iar observatorul (Logger) este notificat.

Documentarea Produsului

```
PS C:\Users\russu\OneDrive\Desktop\UTM III\TMPP\Individual\src> node index.js
=== BUN VENIT LA CAFENEA ===
Alege bautura (1 - Cafea, 2 - Ceai):
```

Figura 6 – Meniul Inițial

Aplicația începe cu alegerea băuturii dorite, avem două opțiuni pentru cafea sau ceai.

```
PS C:\Users\russu\OneDrive\Desktop\UTM III\TMPP\Individual\src> node index.js
=== BUN VENIT LA CAFENEA ===
Alege bautura (1 - Cafea, 2 - Ceai): 2
Vrei sa adaugi lapte? [y/n]: y
Vrei sa adaugi zahar? [y/n]: n
```

Figura 7 – Opțiuni suplimentare

Pentru fiecare băutură avem posibilitatea de a adăuga suplimente ca zahăr sau lapte, cu ajutorul decoratorului, aceste ingrediente suplimentare influențează prețul.

```
PS C:\Users\russu\OneDrive\Desktop\UTM III\TMPP\Individual\src> node index.js
=== BUN VENIT LA CAFENEA ===
Alege bautura (1 - Cafea, 2 - Ceai): 2
Vrei sa adaugi lapte? [y/n]: y
Vrei sa adaugi zahar? [y/n]: n
Alege tipul de pret (1 - Normal, 2 - Happy Hour): 2
Comanda: Ceai, lapte | Total: 8.00 lei
Comandă nouă: Ceai, lapte - 10 lei
Vrei sa mai comanzi? [y/n]:
```

Figura 8 – Aplicarea reducerii

Aplicația suportă aplicarea reducerilor, cu timpul vom putea adăuga diferite reduceri care depind de sezon, sărbători sau diferite promoții.

```
Comandă nouă: Ceai, lapte - 10 lei
Vrei sa mai comanzi? [y/n]: y
Alege bautura (1 - Cafea, 2 - Ceai): 2
Vrei sa adaugi lapte? [y/n]: y
Vrei sa adaugi zahar? [y/n]: y
Alege tipul de pret (1 - Normal, 2 - Happy Hour): 1
Comanda: Ceai, lapte, zahăr | Total: 11.00 lei
Comandă nouă: Ceai, lapte, zahăr - 11 lei
Vrei sa mai comanzi? [y/n]:
```

Figura 9 – Comanda nouă

După o comandă, putem imediat să inițializăm comanda următoare, observăm în figura 9 că prețul produsului depinde de adaosuri și de reducerea aplicată.

```
Comandă nouă: Ceai, lapte, zahăr - 11 lei
Vrei sa mai comanzi? [y/n]: n

=== Comenzile tale ===
1. Ceai, lapte - 10 lei
2. Ceai, lapte, zahăr - 11 lei

Multumim pentru vizita!
PS C:\Users\russu\OneDrive\Desktop\UTM III\TMPP\Individual\src>
```

Figura 10 – Comenzile efectuate

Programul încetează funcțiunea când alegem opțiunea nu la alegerea unei comenzi noi, astfel aplicația ne prezintă o listă cu toate comenzile efectuate, prezentând adaosurile și prețul final, aceasta poate fi dezvoltată cu alte funcții utile: un rezumat cu produsele totale folosite pe zi, venitul total din toate comenzile. Aplicația poate fi folositoare dacă vom implementa memorarea produselor în unități fixe (ex. kg de zahăr), iar la sfârșit de zi vom avea o concluzie pentru eventuala restocare.

Concluzie

Proiectul realizat demonstrează importanța și utilitatea șabloanelor de proiectare în dezvoltarea aplicațiilor software moderne. Prin aplicarea practică a cinci șabloane – Factory, Singleton, Decorator, Strategy și Observer – aplicația pentru gestionarea comenzilor într-o cafenea a fost construită într-un mod modular, extensibil și ușor de întreținut.

Folosirea acestor șabloane a permis separarea clară a responsabilităților și evitarea duplicării codului, facilitând astfel adăugarea de noi funcționalități fără a afecta componentele existente. De exemplu, noi tipuri de băuturi sau toppinguri pot fi integrate fără a modifica logica de bază, iar strategii alternative de preț pot fi aplicate dinamic, în funcție de context.

În plus, șablonul Observer a oferit o soluție elegantă pentru notificarea automată a componentelor interesate de acțiuni interne (precum plasarea unei comenzi), demonstrând beneficiile unei arhitecturi reactive.

Aplicația poate fi extinsă cu ușurință pentru a include funcționalități suplimentare precum integrarea cu o bază de date reală, interfață grafică sau funcționalități de raportare, păstrând în același timp claritatea și flexibilitatea arhitecturii existente.

În concluzie, proiectul nu doar că îndeplinește cerințele tehnice ale temei propuse, dar oferă și un exemplu concret al modului în care șabloanele de proiectare pot îmbunătăți semnificativ calitatea codului și eficiența procesului de dezvoltare software. Aplicația poate fi folosită în cadrul cantinei Universității Tehnice a Moldovei pentru a ușura munca lucrătorilor care manual calculează suma comenzilor studenților.

Anexa 1

```
export interface Beverage {
  getDescription(): string;
  getCost(): number;
}

import { Beverage } from './Beverage';

export class Coffee implements Beverage {
  getDescription(): string {
    return "Cafea";
  }

  getCost(): number {
    return 10;
  }
}

import { Beverage } from './Beverage';

export class Tea implements Beverage {
  getDescription(): string {
    return "Ceai";
  }

  getCost(): number {
    return 8;
  }
}

import { Beverage } from '../beverages/Beverage';

export class MilkDecorator implements Beverage {
  constructor(private beverage: Beverage) { }

  getDescription(): string {
    return this.beverage.getDescription() + ", lapte";
  }

  getCost(): number {
    return this.beverage.getCost() + 2;
  }
}

import { Beverage } from '../beverages/Beverage';

export class SugarDecorator implements Beverage {
  constructor(private beverage: Beverage) { }

  getDescription(): string {
    return this.beverage.getDescription() + ", zahăr";
  }

  getCost(): number {
    return this.beverage.getCost() + 1;
  }
}

import { Coffee } from '../beverages/Coffee';
import { Tea } from '../beverages/Tea';
```

```
import { Beverage } from '../beverages/Beverage';

export class BeverageFactory {
  static createBeverage(type: 'coffee' | 'tea'): Beverage {
    if (type === 'coffee') return new Coffee();
    else return new Tea();
  }
}

import { Beverage } from '../beverages/Beverage';

export function logNewOrder(order: Beverage) {
  console.log(`Comandă nouă: ${order.getDescription()} - ${order.getCost()} lei`);
}
import { Beverage } from '../beverages/Beverage';

type Observer = (order: Beverage) => void;

export class OrderManager {
  private static instance: OrderManager;
  private orders: Beverage[] = [];
  private observers: Observer[] = [];

  private constructor() { }

  static getInstance(): OrderManager {
    if (!OrderManager.instance) {
      OrderManager.instance = new OrderManager();
    }
    return OrderManager.instance;
  }

  addObserver(observer: Observer) {
    this.observers.push(observer);
  }

  notify(order: Beverage) {
    this.observers.forEach(obs => obs(order));
  }

  addOrder(order: Beverage) {
    this.orders.push(order);
    this.notify(order);
  }

  getOrders(): Beverage[] {
    return this.orders;
  }
}

import { Beverage } from '../beverages/Beverage';

export interface PricingStrategy {
  calculatePrice(beverage: Beverage): number;
}

export class NormalPricing implements PricingStrategy {
  calculatePrice(beverage: Beverage): number {
    return beverage.getCost();
  }
}
```

```
export class HappyHourPricing implements PricingStrategy {
  calculatePrice(beverage: Beverage): number {
    return beverage.getCost() * 0.8; // 20% reducere
  }
}
import { BeverageFactory } from './factory/BeverageFactory';
import { MilkDecorator } from './decorators/MilkDecorator';
import { SugarDecorator } from './decorators/SugarDecorator';
import { OrderManager } from './singleton/OrderManager';
import { logNewOrder } from './observer/OrderObserver';
import { HappyHourPricing, NormalPricing, PricingStrategy } from './strategy/PricingStrategy';
import readlineSync from 'readline-sync';
import { Beverage } from './beverages/Beverage';

const orderManager = OrderManager.getInstance();
orderManager.addObserver(logNewOrder);

console.log("=== BUN VENIT LA CAFENEA ===");

let running = true;

while (running) {
  const beverageChoice = readlineSync.question('Alege bautura (1 - Cafea, 2 - Ceai): ');
  let beverage: Beverage;

  if (beverageChoice === '1') {
    beverage = BeverageFactory.createBeverage('coffee');
  } else if (beverageChoice === '2') {
    beverage = BeverageFactory.createBeverage('tea');
  } else {
    console.log('Optiune invalida!');
    continue;
  }

  const addMilk = readlineSync.keyInYNStrict('Vrei sa adaugi lapte?');
  if (addMilk) beverage = new MilkDecorator(beverage);

  const addSugar = readlineSync.keyInYNStrict('Vrei sa adaugi zahar?');
  if (addSugar) beverage = new SugarDecorator(beverage);

  const priceChoice = readlineSync.question('Alege tipul de pret (1 - Normal, 2 - Happy Hour):
');
  let pricingStrategy: PricingStrategy;

  if (priceChoice === '2') {
    pricingStrategy = new HappyHourPricing();
  } else {
    pricingStrategy = new NormalPricing();
  }

  const finalPrice = pricingStrategy.calculatePrice(beverage);
  console.log(`Comanda: ${beverage.getDescription()} | Total: ${finalPrice.toFixed(2)} lei`);

  orderManager.addOrder(beverage);

  running = readlineSync.keyInYNStrict('Vrei sa mai comanzi?');
}

console.log('\n=== Comenzile tale ===');
orderManager.getOrders().forEach((order, index) => {
```

```
  console.log(`${index + 1}. ${order.getDescription()} - ${order.getCost()} lei`);
});

console.log('\nMultumim pentru vizita!');
```

Anexa 2

https://github.com/russuiulia/TMPP_Russu_Iulia_TI-224/tree/master/Proiect-de-an - repozitoriul cu aplicația creată însoțită de un ReadME file.