**Ministerul Educației, Culturii și Cercetări**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**Departamentul Ingineria Software și Automatică**

**Proiect de an**

Disciplina: Tehnici și mecanisme de proiectare software

Tema: „Proiectarea unei aplicații pentru eliberarea ticketelor”

A efectuat st.gr. TI-204 Bujenița Grigore

A verificat asis.univ. Gaidau Mihai

Chișinău 2023

Cuprins

[**Introducere** 2](#_Toc137427813)

[**1.Proiectarea aplicației** 6](#_Toc137427814)

[**1.1 Descrierea problemei** 6](#_Toc137427815)

[**1.2 Șabloane Creaționale** 7](#_Toc137427816)

[**1.3 Șabloane Structurale** 11](#_Toc137427817)

[**1.4 Șabloane comportamentale** 12](#_Toc137427818)

[Figura 1.7 Șablonul Memento 14](#_Toc137427819)

[**2. Descrierea principiului de lucru** 14](#_Toc137427820)

[**Concluzii** 16](#_Toc137427821)

[**Bibliografie** 17](#_Toc137427822)

# **Introducere**

Modelele de proiectare sunt soluții tipice pentru problemele care apar frecvent în proiectareasoftware.Sunt ca niște planuri pre-făcute pe care le puteți personaliza pentru a rezolva o problemă dedesign recurentă din codul dvs.Nu poate fi găsit doar un model și să fie copiat în programul dvs., așa cum se poate face cu funcțiile sau bibliotecile de-pe-raft. Modelul nu este o bucată de cod specific, ci un concept general pentru rezolvarea unei anumite probleme. Se pot urmări detaliile modelului și se poate implementa o soluție care se potrivește realităților propriului program. Modelele sunt deseori confundate cu algoritmii, deoarece ambele concepte descriu soluții tipice pentru unele probleme cunoscute .În timp ce un algoritm definește întotdeauna un set clar de acțiuni care pot atinge un anumit obiectiv, un model este o descriere mai bună a unei soluții. Codul aceluiași model aplicat la două programe diferite poate fi diferit. O analogie cu un algoritm este o rețetă de gătit: ambii au pași clari pentru atingerea unui obiectiv. Pe de altă parte ,un model seamănă mai mult cu un model: puteți vedea care este rezultatul și caracteristicile acestuia.

ÎN CE CONSTĂ MODELUL?

Majoritatea șabloanelor sunt descrise foarte formal , astfel încât oamenii să le poată reproduce în multe contexte. Iată secțiunile care sunt de obicei prezente într-o descriere a modelului:

* Intenția tiparului de a descrie pe scurt atât problema,cât și soluția.
* Motivația explică în continuare problema și soluția pe care modelul o face posibilă.
* Structura claselor arată fiecare parte a modelului și modul în care acestea sunt legate.
* Exemplul de cod într-unul dintre limbajele de programare populare face mai ușor să înțelegeți ideea din spatele modelului. Unele cataloage de șabloane enumeră alte detalii utile, cum ar fi aplicabilitatea modelului, etapele de implementare și relațiile cu alte tipare.

Un proiectant fara experienta este de multe ori coplesit de multitudinea optiunilor disponibile si are tendinta de a se intoarce la tehnicile non-obiectuale pe care le-a folosit in trecut. Un proiectant experimentat stie ca NU TREBUIE sa rezolve fiecare problema incepand de la zero, ci reutilizand solutii din proiecte anterioare. Atunci cand descopera o solutie buna o va folosi mereu. Acest tip de experienta este o parte din ceea ce confera unui proiectant statutul de expert.

In cadrul sistemelor OO realizate in mod profesionist se pot distinge sabloane de clase si de obiecte in comunicare care rezolva probleme specifice si fac ca sistemele respective sa fie mai flexibile, mai elegante si reutilizabile. Un proiectant familiar cu asemnea sabloane le va putea aplica repede, fara a trebui sa le redescopere.

Sabloanele de proiectare sunt de fapt o memorare pentru posteritate a experientei in domeniul proiectarii sistemelor OO.

***Criterii de clasificare:***

**Dupa scop**: sabloanele pot fi creationale, structurale sau comportamentale.

* + Sabloanele **creationale** (creational patterns) privesc modul de creare a obiectelor.
  + Sabloanele **structurale** (structural patterns) se refera la compozitia claselor sau a obiectelor.
  + Sabloanele **comportamentale** (behavioral patterns) caracterizeaza modul in care obiectele si clasele interactioneaza si isi distribuie responsabilitatile.

**Dupa domeniul de aplicare**: sabloanele se pot aplica obiectelor sau claselor.

* + Sabloanele **obiectelor** se refera la relatiile dintre obiecte, relatii care au un caracter dinamic.

Sabloanele creationale ale obiectelor acopera situatiile in care o parte din procesul crearii unui obiect cade in sarcina unui alt obiect. Sabloanele structurale ale obiectelor descriu caile prin care se asambleaza obiecte. Sabloanele comportamentale ale obiectelor descriu modul in care un grup de obiecte coopereaza pentru a indeplini o sarcina ce nu ar putea fi efectuata de un singur obiect.

* + Sabloanele **claselor** se refera la relatii dintre clase, relatii stabilite prin mostenire si care sunt statice (fixate la compilare).

Sabloanele creationale ale claselor acopera situatiile in care o parte din procesul crearii unui obiect cade in sarcina subclaselor. Sabloanele structurale ale claselor descriu modul de utilizare a mostenirii in scopul compunerii claselor. Sabloanele comportamentale ale claselor utilizeaza mostenirea pentru descrierea unor algoritmi si fluxuri de control.

|  |  |
| --- | --- |
| **Șabloane creationale** | |
| [Abstract Factory](https://www.dofactory.com/Patterns/PatternAbstract.aspx) | Furnizeaza o interfata pentru crearea unor familii de obiecte |
| [Builder](https://www.dofactory.com/Patterns/PatternBuilder.aspx) | Separa obiectele construite de reprezentarea lor |
| [Factory Method](https://www.dofactory.com/Patterns/PatternFactory.aspx) | Creaza o instanta a diferitelor clase derivate |
| [Prototype](https://www.dofactory.com/Patterns/PatternPrototype.aspx) | O instanta initializata poate fi copiata sau clonata |
| [Singleton](https://www.dofactory.com/Patterns/PatternSingleton.aspx) | O clasa din care doar o singura instanta poate exista |
| **Șabloane structurale** | |
| [Adapter](https://www.dofactory.com/Patterns/PatternAdapter.aspx) | Face ca interfetele diferitelor clase sa corespunda |
| [Bridge](https://www.dofactory.com/Patterns/PatternBridge.aspx) | Separa interfata unui obiect de implementarea acestuia |
| [Composite](https://www.dofactory.com/Patterns/PatternComposite.aspx) | Structura arborescenta de obiecte simple si compuse |
| [Decorator](https://www.dofactory.com/Patterns/PatternDecorator.aspx) | Adauga responsabilitati obiectelor in mod dinamic |
| [Faade](https://www.dofactory.com/Patterns/PatternFacade.aspx) | O singura clasa ce reprezinta un intreg sistem |
| [Flyweight](https://www.dofactory.com/Patterns/PatternFlyweight.aspx) | O instanta folosita pentru un sharing eficient |
| [Proxy](https://www.dofactory.com/Patterns/PatternProxy.aspx) | Un obiect reprezentand un un alt obiect |

Table 1.1 Șabloane creaționale și Șabloane structurale

|  |  |
| --- | --- |
| **Behavioral Patterns** | |
| [Chain of Resp.](https://www.dofactory.com/Patterns/PatternChain.aspx) | O cale de a transmite cereri intre obiecte |
| [Command](https://www.dofactory.com/Patterns/PatternCommand.aspx) | Incapsuleaza o cerere ca un obiect |
| [Interpreter](https://www.dofactory.com/Patterns/PatternInterpreter.aspx) | O cale de a include elemente lingvistice intr-un program |
| [Iterator](https://www.dofactory.com/Patterns/PatternIterator.aspx) | Acces secvential la elementele unei colectii |
| [Mediator](https://www.dofactory.com/Patterns/PatternMediator.aspx) | Defineste comunicarea simplificata intre clase |
| [Memento](https://www.dofactory.com/Patterns/PatternMemento.aspx) | Capteaza si reconstitui starea interna a unui obiect |
| [Observer](https://www.dofactory.com/Patterns/PatternObserver.aspx) | O cale de a anunta schimbarea unui numar de clase |
| [State](https://www.dofactory.com/Patterns/PatternState.aspx) | Modificarea comportamentului unui obiect atunci cand isi schimba starea |
| [Strategy](https://www.dofactory.com/Patterns/PatternStrategy.aspx) | Incapsuleaza un algoritm intr-o clasa |
| [Template Method](https://www.dofactory.com/Patterns/PatternTemplate.aspx) | Transfera pasii exacti ai unui algoritm unei subclase |
| [Visitor](https://www.dofactory.com/Patterns/PatternVisitor.aspx) | Defineste o noua operatie pentru o clasa fara a face schimbari |

Tabel 1.2 Șabloane de comportament

# **1.Proiectarea aplicației**

## **1.1 Descrierea problemei**

Aplicația este un sistem simplu de vânzare a biletelor cu interfață de linie de comandă (CLI). Aceasta utilizează diverse șabloane de proiectare pentru a oferi funcționalități precum cumpărarea de bilete, afișarea biletelor disponibile și gestionarea acțiunilor utilizatorului. Utilizatorul poate interacționa cu aplicația prin intermediul CLI și poate efectua următoarele acțiuni: Cumpărarea de bilete: Utilizatorul poate introduce numărul de loc, numărul de rând și informații despre locul VIP. De asemenea, utilizatorul poate indica dacă beneficiază de un discount pentru studenți. Aplicația creează un obiect Ticket corespunzător și îl vinde utilizând un obiect TicketFacade. Informații despre biletul vândut și prețul acestuia sunt afișate. Afișarea biletelor disponibile: Utilizatorul poate solicita afișarea tuturor biletelor disponibile. Aplicația utilizează un iterator pentru a itera prin colecția de bilete și le afișează unul câte unul. Gestionarea acțiunilor utilizatorului: După fiecare acțiune, utilizatorul primește opțiunea de a alege o altă acțiune. Aceasta poate implica cumpărarea unui alt bilet, afișarea biletelor disponibile sau încheierea interacțiunii cu aplicația. Aplicația utilizează șabloane de proiectare precum Facade, Builder și Iterator pentru a organiza și simplifica logica vânzării de bilete. Aceasta permite separarea responsabilităților și oferă o interfață coerentă pentru utilizator.Șabloanele de proiectare abordează diferite scenarii de creare a obiectelor și facilitează adaptabilitatea și extensibilitatea sistemului. Iată câteva exemple de șabloane care ar putea fi utilizate în proiectul nostru:

* Factory Method
* Builder
* Prototype
* Adapter
* Decorator
* Memento
* Chain of Responsability

## **1.2 Șabloane Creaționale**

Am implementat următoarele 3 șabloane creaționale: Factory Method,Builder, Prototype,

Pentru început va fi descris Factory Method

**Intent –** Factory Method este un model de design creațional care oferă o interfață pentru crearea de obiecte într-o superclasă, dar permite subclaselor să modifice tipul de obiecte care vor fi create.

-

Figura 1.1 Diagrama UML pentru Factory Method

**Implementare –**

export class TicketBuilder {

    constructor() {

        this.ticket = new Ticket();

    }

    setSeat(seat) {

        this.ticket.seat = seat;

        return this;

    }

    setRow(row) {

        this.ticket.row = row;

        return this;

    }

    setVIP(isVIP) {

        this.ticket.isVIP = isVIP;

        return this;

    }

    setDiscount(hasDiscount) {

        this.ticket.hasDiscount = hasDiscount;

        return this;

    }

    setEmail(email) {

        this.ticket.email = email;

        return this;

    }

    build() {

        const basePrice = 10;

        const vipPriceMultiplier = 1.5;

        const seatPriceMultiplier = 1.2;

        let price = basePrice;

        if (this.ticket.isVIP) {

            price \*= vipPriceMultiplier;

        }

        if (this.ticket.seat === 'A1' || this.ticket.seat === 'B2' || this.ticket.seat === 'C3') {

            price \*= seatPriceMultiplier;

        }

        this.ticket.price = price;

        return this.ticket;

    }

}

Împreună cu acest șablon am implementat și **builder**,care este la fel un șablon creational.

**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**.**

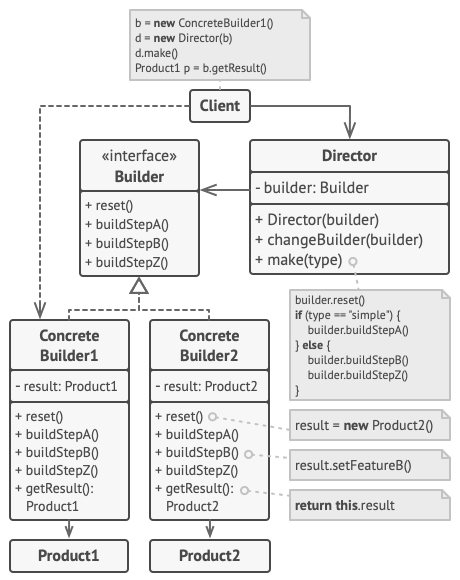


Figura 1.2 Șablonul Builder

class Builder(ABC):  
 class TicketOrderBuilder {

    constructor() {

        this.movie = null;

        this.theater = null;

        this.tickets = [];

    }

    setMovie(movie) {

        this.movie = movie;

        return this;

    }

    setTheater(theater) {

        this.theater = theater;

        return this;

    }

    addTicket(ticket) {

        this.tickets.push(ticket);

        return this;

    }

    build() {

        return new TicketOrder(this.movie, this.theater, this.tickets);

    }

}

**Prototype.**

**Prototype** este un model de design creațional care permite copierea obiectelor existente fără a face codul să depindă de clasele lor.

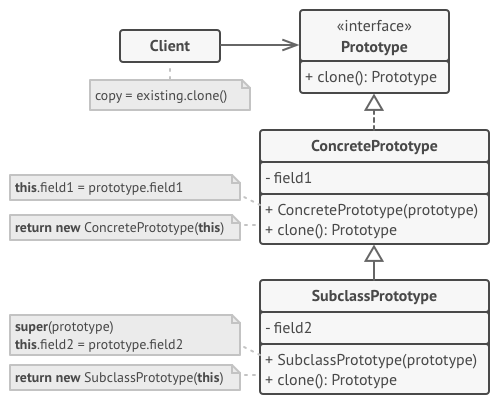


Figura 1.3 Șablonul Prototype

Pentru implementarea acestuia a fost implementată interfața Prototype cu metoda clone, iar apoi pentru fiecare subclasă a fost realizată prototipul concret

// Prototype for creating ticket objects

class Ticket {

    constructor(movie, theater, row, seat) {

        this.movie = movie;

        this.theater = theater;

        this.row = row;

        this.seat = seat;

    }

    // Implement this method in the child classes

    getPrice() { }

}

// Concrete prototype for creating regular tickets

class RegularTicket extends Ticket {

    getPrice() {

        return 10;

    }

}

// Concrete prototype for creating VIP tickets

class VIPTicket extends Ticket {

    getPrice() {

        return 20;

    }

}

## **1.3 Șabloane Structurale**

Modelele de proiectare structurală explică modul de asamblare a obiectelor și claselor în structuri mai mari, păstrând în același timp aceste structuri flexibile și eficiente.

În realizarea proiectului am folosit 2 șabloane structurale:Adapter și Decorator.

**Adapter** este un model de design structural care permite obiectelor cu interfețe incompatibile să colaboreze.

Convertează interfața unei clase într-o altă interfață și așteptată de la clienți. Adaptorul permite claselor să lucreze împreună, ceea ce altfel nu ar putea din cauza interfețelor incompatibile.

Conectează o clasă existentă cu o interfață nouă.

Potrivirea impedanței unei componente vechi cu un sistem nou.

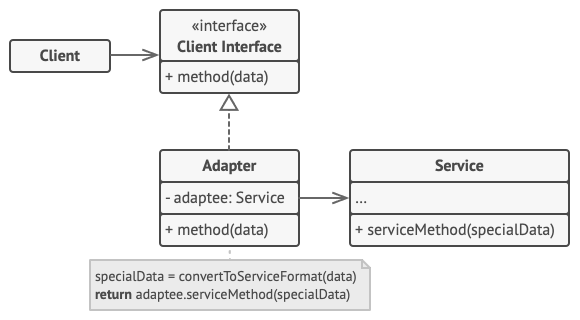


Figura 1.4 Șablonul Adapter

export class OTSAdapter extends TheaterTicketSeller {

    constructor(onlineTicketSeller) {

        super();

        this.onlineTicketSeller = onlineTicketSeller;

    }

    sellTicket(show, row, seat) {

        const ticket = this.onlineTicketSeller.buyTicket(show, row, seat);

        return new Ticket(show, ticket.price, row, seat);

    }

}

## **1.4 Șabloane comportamentale**

În ingineria software, modelele de proiectare comportamentală sunt modele de proiectare care identifică modele comune de comunicare între obiecte și realizează aceste modele. Procedând astfel, aceste modele cresc flexibilitatea în realizarea acestei comunicări.

La proiectarea acestei aplicații au fost folosite 2 șabloane de comportament: Memento și Chain of responsability.

**Chain of Responsability** este un model de design comportamental care vă permite să transmiteți cereri de-a lungul unui lanț de manipulatori. La primirea unei cereri, fiecare handler decide fie să proceseze cererea, fie să o transmită următorului handler din lanț.

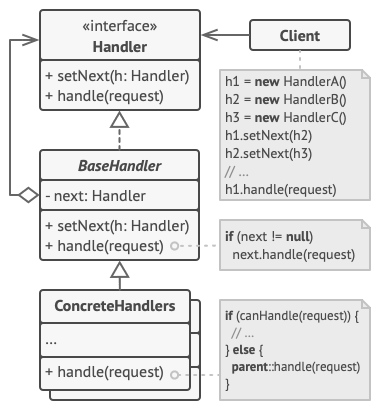


Figura 1.6 Șablonul Chain of Responsability

**Memento** este un model de design comportamental care vă permite să salvați și să restabiliți starea anterioară a unui obiect fără a dezvălui detaliile implementării acestuia.

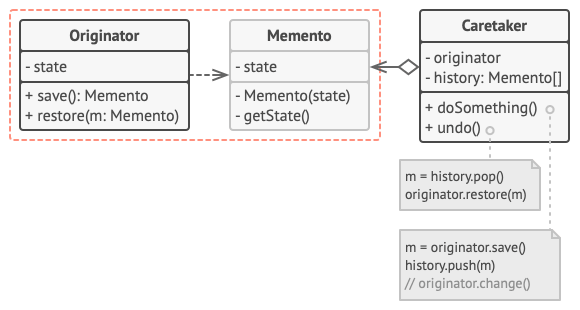


Figura 1.7 Șablonul Memento

# **2. Descrierea principiului de lucru**

Principiile de muncă ale sabloanelor creatională facilitează crearea și utilizarea obiectelor într-un mod modular, flexibil și eficient. Aceste sabloane ne permit să separăm responsabilitățile legate de crearea și configurarea obiectelor de alte aspecte ale sistemului, precum logica de afaceri sau interacțiunea cu utilizatorul. Astfel, putem obține un cod mai modular, mai ușor de întreținut și cu o arhitectură mai flexibilă.

Aplicația are un funcțional simplu,dar suficient pentru realizarea unei comenzi depline.

La început este prezent un menu

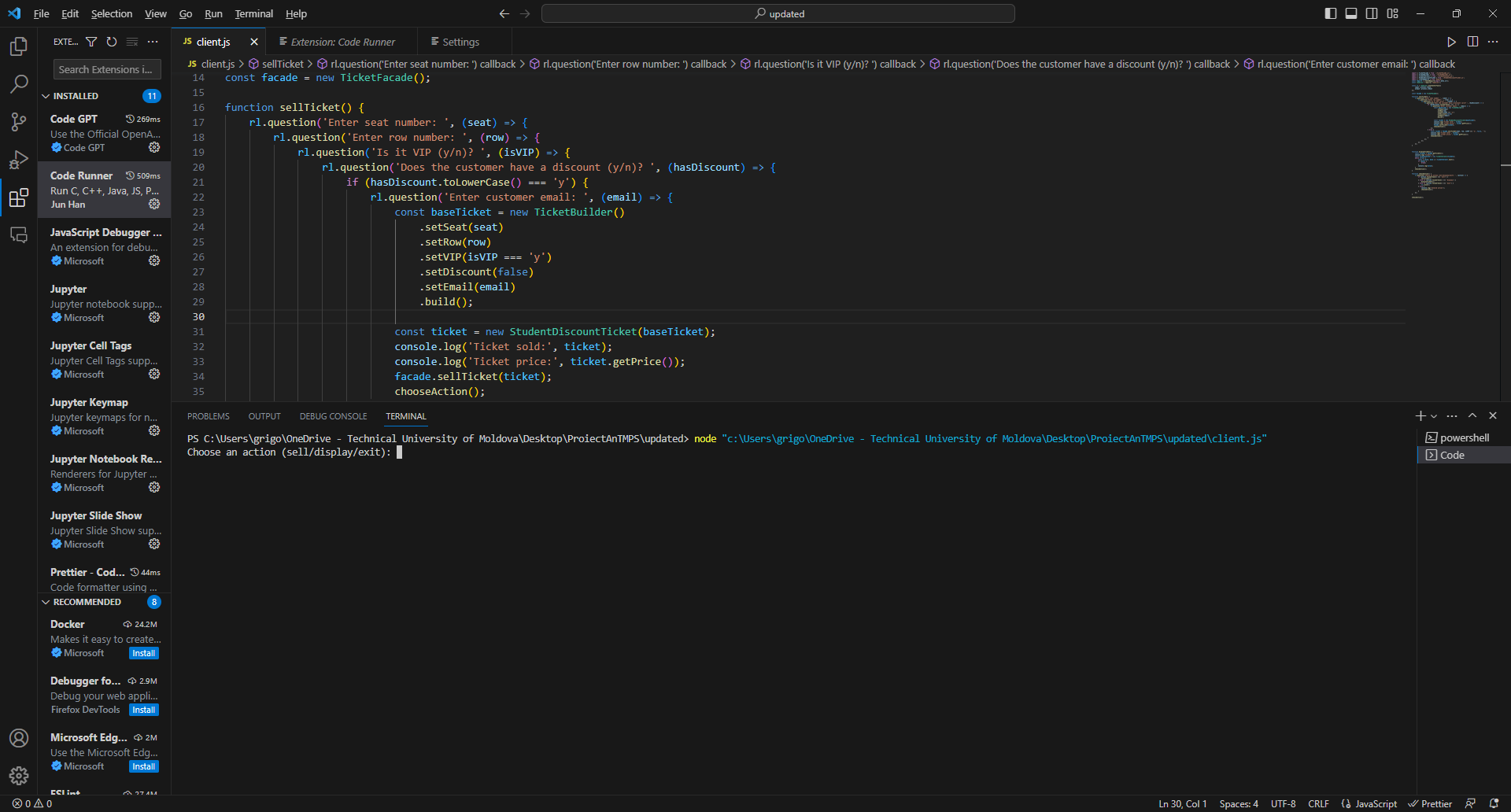


Figura 2.1 Meniul aplicației

Figura 2.2 Alegerea optiunilor

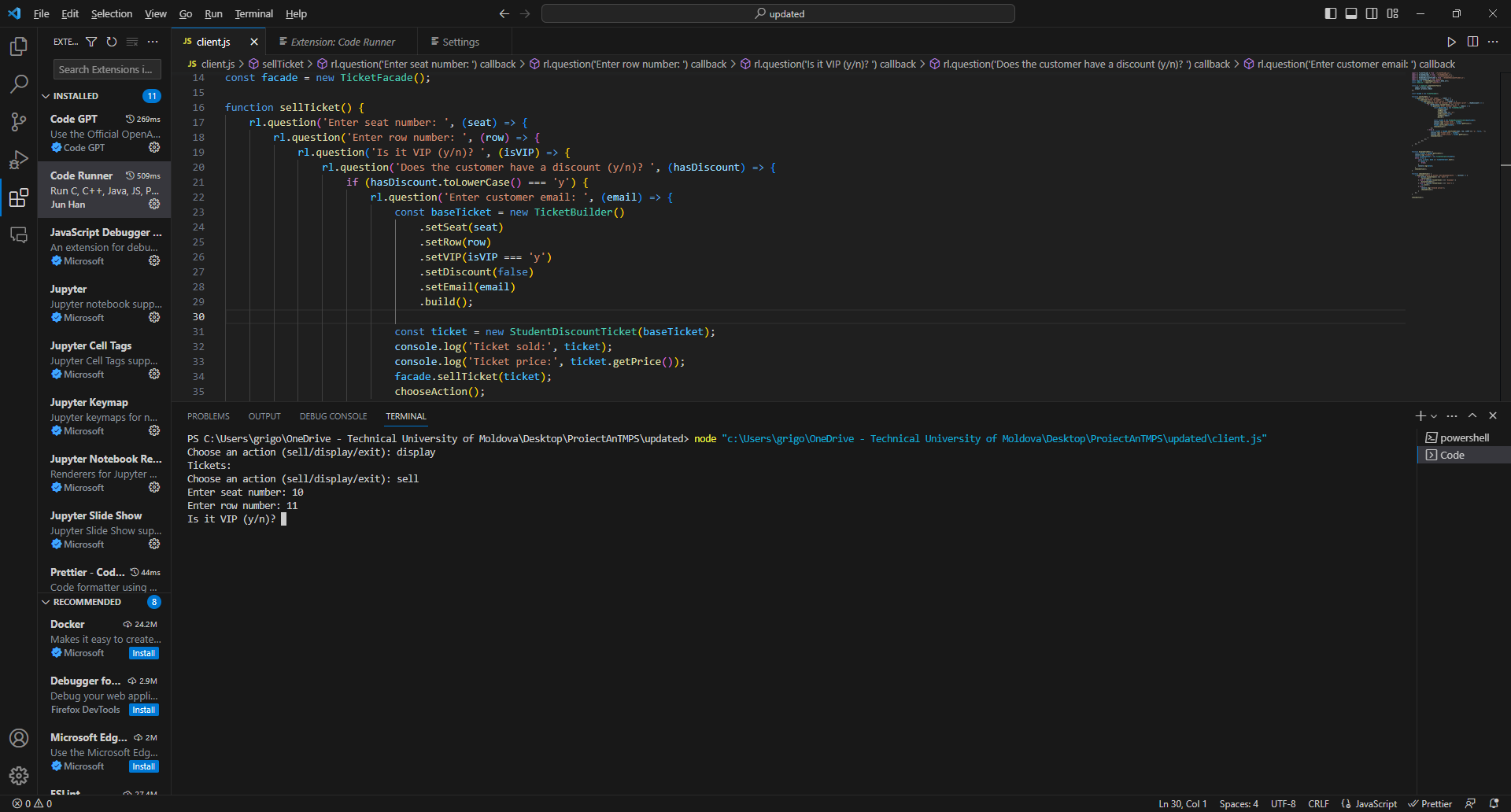
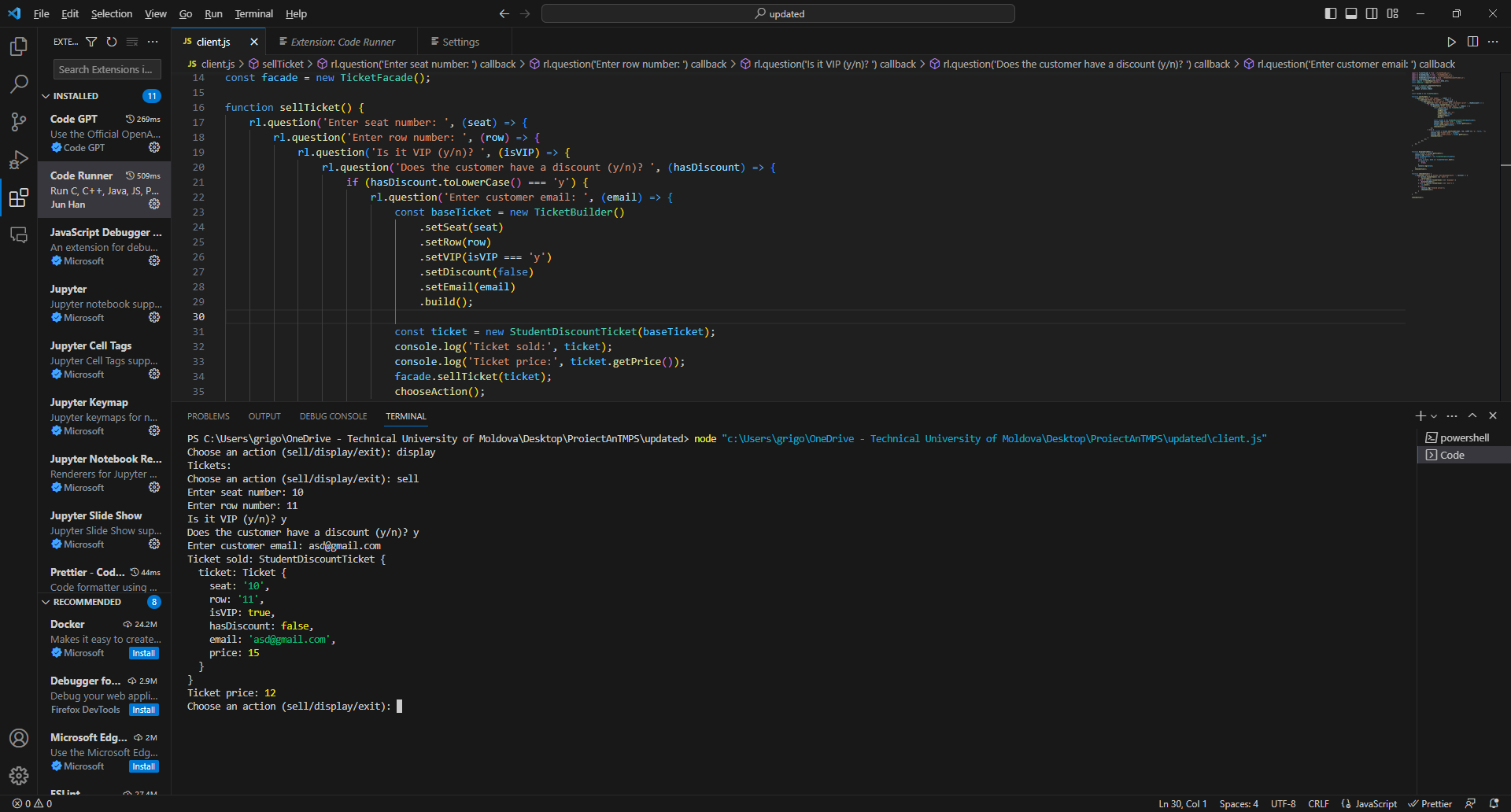


Figura 2.4 Rezultatul



# **Concluzii**

În concluzie, proiectul de curs la disciplina "Tehnici și Mecanisme de Proiectare Software" pentru proiectarea unei aplicații pentru un magazin de pizzerie o oportunitate de aplicare și aprofundare a cunoștințelor teoretice în proiectarea software. Prin abordarea acestui proiect, s-a avut posibilitatea de a explora a diverselor aspecte și principii importante în procesul de proiectare a unei aplicații.

În continuare, s-au aplicat diverse sabloane de proiectare, inclusiv sabloane creaționale, pentru optimizarea procesului de creare și de gestionare a obiectelor în aplicare. S-au folosit sabloane precum Singleton, Factory și Builder pentru asigurarea unei creări eficiente și flexibile a obiectelor, facilitând astfel extensibilitatea și adaptabilitatea sistemului în viitor. Pe proiect, s-au explorat și alte principii și concepte importante în proiectarea software, cum ar fi separat responsabilitățile, modularitatea, extensibilitatea și flexibilitatea. S-a avut ocazia de a lucra cu arhitecturi de software și s-a înțeles importanța aplicării acestor principii pentru obținerea unui cod coerent, ușor de întreținut și scalabil.

În final, realizarea acestui proiect a condus la dobândirea unei mai bune dezvoltări a tehnologiilor și tehnicilor de proiectare software, precum și a importanței aplicării principiilor și a sabloanelor de proiectare în contextul dezvoltării unei aplicații complexe. Acest proiect a oferit oportunitatea de aplicare a cunoștințelor teoretice într-un proiect practic, dezvoltând abilități de proiectare și gândire critică în domeniul software.

# **Bibliografie**

* + 1. Șabloane Creaționale , Citat 05.06.20203 disponibil: <https://refactoring.guru/design-patterns/creational-patterns>
    2. Șabloane Structurale , Citat 05.06.20203 disponibil: <https://refactoring.guru/design-patterns/structural-patterns>
    3. Șabloane de Comportament , Citat 05.06.20203 disponibil: <https://refactoring.guru/design-patterns/behavioral-patterns>

# 