**Ministerul Educaţiei, Culturii și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Departamentul Informatică și Ingineria Sistemelor**

**RAPORT**

**Proiect de an la TMPS**

A efectuat:

st. gr. TI-203 Arapu Anatol

A verificat:

l. univ. Mihai Gaidau

Chişinău – 2023

Introducerea

Proiectul este o aplicație Java pentru un sistem de comenzi de mâncare care încorporează diferite patternuri de proiectare. Aplicația permite utilizatorilor să plaseze comenzi de mâncare, să-și personalizeze comenzile cu instrucțiuni speciale și să aleagă diferite strategii de plată.

Proiectul implementează următoarele patternuri de proiectare:

**Singleton Pattern:** Patternul Singleton asigură că există o singură instanță a clasei FoodOrderService, care gestionează plasarea și procesarea comenzilor. Acest lucru garantează că toate operațiunile legate de comenzi sunt centralizate și consistente în întreaga aplicație.

**Builder Pattern:** Patternul Builder este utilizat pentru a construi obiecte de tip FoodOrder. Clasa FoodOrderBuilder oferă o abordare pas cu pas pentru crearea de obiecte complexe FoodOrder prin stabilirea diferitelor atribute precum alimente, cantități și instrucțiuni speciale.

**Prototype Pattern:** Patternul Prototype permite clonarea obiectelor FoodOrder pentru a crea noi instanțe cu aceleași valori de atribut. Acest lucru este util atunci când utilizatorii doresc să dubleze o comandă existentă cu modificări minore.

**Decorator Pattern:** Patternul Decorator este utilizat pentru a adăuga comportament suplimentar obiectelor FoodOrder. Clasa FoodOrderDecorator acționează ca o încapsulare în jurul unui obiect FoodOrder și îmbunătățește funcționalitatea acestuia, oferind metode suplimentare sau modificând cele existente.

**Proxy Pattern:** Patternul Proxy este implementat folosind clasa FoodOrderServiceProxy. Aceasta acționează ca un proxy pentru serviciul real FoodOrderService și oferă funcționalități suplimentare sau verificări înainte de a redirecționa cererile către serviciul real.

**Observer Pattern:** Patternul Observer este utilizat pentru a implementa un sistem de notificare a stării comenzilor. Clasa OrderStatusSubject menține o listă de observatori (obiecte OrderStatusObserver) care sunt notificați atunci când starea comenzii se schimbă.

**Strategy Pattern:** Patternul Strategy este utilizat pentru implementarea diferitelor strategii de plată. Interfața PaymentStrategy definește metodele comune pentru plată, în timp ce implementările concrete precum CreditCardPaymentStrategy și PayPalPaymentStrategy oferă mecanisme specifice de plată.

**Command Pattern:** Patternul Command este utilizat pentru a încapsula operațiunile legate de comenzi în obiecte de tip comandă separate (OrderCommand, AddItemCommand, RemoveItemCommand). Acest lucru permite aplicației să separare invocatorul (clientul) de destinatar (obiectul FoodOrder) și oferă o modalitate de executare și anulare a operațiunilor.

Structura dosarelor proiectului organizează codul în categorii distincte, precum client, domeniu, utilitare și date. Această structură ajută la menținerea unui cod modular și organizat.

În ansamblu, proiectul își propune să demonstreze implementarea practică a diferitelor tipare de proiectare într-un sistem de comenzi de mâncare, prezentând avantajele lor în ceea ce privește modularitatea, extensibilitatea și reutilizabilitatea codului.

**Consideratii teoretice**

1. În proiect, structura dosarelor organizează codul într-un mod modular și bine structurat, asigurând separarea clară a diferitelor componente și funcționalități ale sistemului de comenzi. Aceasta facilitează dezvoltarea, testarea și menținerea proiectului pe termen lung. Prin organizarea componentelor în dosare specifice, avem o abordare mai coerentă și ușoară de gestionare a codului sursă și a resurselor asociate proiectului. Considerațiile teoretice pentru proiectul unui sistem de comenzi de mâncare implementând diverse tipare de proiectare sunt următoarele:

* Singleton Pattern: Utilizarea patternului Singleton pentru clasa FoodOrderService asigură că există o singură instanță a acesteia în întreaga aplicație. Aceasta permite centralizarea operațiunilor de gestionare a comenzilor și menține consistența datelor pe parcursul aplicației. O singură instanță a clasei FoodOrderService asigură că informațiile legate de comenzile de mâncare sunt gestionate coerent și că nu există conflicte sau inconsistente între diferitele componente ale sistemului.
* Builder Pattern: Implementarea patternului Builder pentru obiectele FoodOrder permite crearea ușoară și flexibilă a unor comenzi complexe. Prin intermediul clasei FoodOrderBuilder, putem adăuga treptat diferite atribute ale comenzii, oferind astfel un control mai mare asupra procesului de construire. Acest tipar ne ajută să gestionăm cu ușurință comenzile care pot avea mai multe atribute personalizabile și să oferim o experiență flexibilă utilizatorilor care pot personaliza comenzile lor în funcție de preferințe individuale.
* Prototype Pattern: Utilizarea patternului Prototype facilitează crearea de noi instanțe ale comenzilor pe baza unor modele existente. Clonarea obiectelor FoodOrder permite replicarea comenzilor cu ușurință, iar apoi se pot face modificări minore la acestea pentru a se adapta cerințelor utilizatorului. Acest tipar ne oferă o modalitate eficientă de a crea și gestiona comenzile personalizate, pornind de la un model existent și făcând ajustări specifice.
* Decorator Pattern: Implementarea patternului Decorator pentru obiectele FoodOrder permite adăugarea de funcționalități suplimentare sau modificarea comportamentului acestora fără a afecta structura de bază a comenzilor. Aceasta oferă o flexibilitate crescută în extinderea funcționalității sistemului de comenzi. Prin intermediul decoratorilor, putem adăuga caracteristici suplimentare la comenzile de mâncare, cum ar fi opțiuni de ambalare specială, servicii de livrare prioritară sau alte modificări personalizate, fără a modifica direct clasa de bază a comenzii.
* Proxy Pattern: Utilizarea patternului Proxy prin intermediul clasei FoodOrderServiceProxy adaugă un strat suplimentar de control și securitate asupra operațiunilor de gestionare a comenzilor. Proxy-ul poate valida cererile, adăuga logica suplimentară și oferi o interfață unificată către serviciul real de gestionare a comenzilor. Acest pattern ne permite să implementăm reguli de securitate sau verificări suplimentare înainte de a procesa comenzile și asigură că resursele și operațiunile sistemului sunt utilizate în mod corespunzător.
* Observer Pattern: Implementarea patternului Observer pentru sistemul de notificare a stării comenzilor permite componentelor interesate să fie informate în timp real despre modificările stării comenzilor. Acest pattern sporește modularitatea și separarea preocupărilor, permițând o extensibilitate mai mare a sistemului. Observatorii pot fi înregistrati pentru a primi notificări atunci când starea comenzii se schimbă, ceea ce permite componentelor sau serviciilor conexe să reacționeze în mod corespunzător la aceste evenimente.
* Strategy Pattern: Utilizarea patternului Strategy pentru implementarea diferitelor strategii de plată oferă posibilitatea de a adăuga și modifica ușor modalitățile de plată disponibile în sistem. Acest pattern separă logica de plată de obiectele de comandă și permite adăugarea de strategii de plată personalizate fără modificarea structurii existente a codului. Utilizarea acestui tipar ne permite să extindem sistemul pentru a accepta noi metode de plată sau pentru a implementa reguli specifice pentru fiecare strategie de plată.
* Command Pattern: Implementarea patternului Command pentru operațiunile legate de comenzile de mâncare permite separarea clară a invocatorului și a receptorului operațiunilor. Astfel, se asigură o decuplare eficientă între componentele implicate și se oferă flexibilitate în gestionarea comenzilor și a acțiunilor asociate acestora. Acest tipar ne permite să encapsulăm operațiunile de comandă în obiecte separate, ceea ce ne oferă posibilitatea de a executa sau de a anula operațiuni în funcție de necesități.

1. **Modularitate și extensibilitate**

Proiectul sistemului de comenzi de mâncare implementând diverse patterne de proiectare se bazează pe principii solide de modularitate și extensibilitate. Aceste aspecte sunt esențiale pentru a permite dezvoltarea și evoluția sistemului pe termen lung, adăugând noi funcționalități sau adaptându-l la cerințele viitoare.

Prin utilizarea patternurilor de proiectare menționate, se obține o separare clară și coerentă a responsabilităților între diferitele componente ale sistemului. Fiecare tipar de proiectare abordează o anumită problemă specifică și oferă o soluție bine definită și independentă. Aceasta facilitează dezvoltarea modulară a sistemului, unde fiecare componentă poate fi dezvoltată și testată individual, fără a afecta restul sistemului.

Modularitatea în proiect este obținută prin împărțirea codului în dosare și pachete specifice, organizate în funcție de funcționalitatea lor. Astfel, avem dosare separate pentru componente precum clienți, domeniu, utilități și date. Această structură ajută la menținerea unui cod ordonat și ușor de navigat, unde fiecare componentă are un scop clar și bine definit. Această abordare modulară facilitează dezvoltarea echipei, permite paralelizarea lucrului și face mai ușoară înțelegerea și extinderea codului.

1. **Flexibilitatea în implementare**

Flexibilitatea în implementarea acestui proiect este susținută de utilizarea patternurilor de proiectare, care oferă soluții modulare, extensibile și personalizabile pentru diferitele componente și funcționalități ale sistemului de comenzi de mâncare. Aceasta permite adaptarea facilă a proiectului la cerințele specifice și evoluția ulterioară a aplicației, asigurând un grad ridicat de flexibilitate și adaptabilitate.

**Descrierea Principiului de lucru al tematicii alese**

Tema aleasă pentru proiect este un sistem de comenzi de mâncare care integrează diverse patterne de proiectare. Principiul de funcționare al sistemului de comenzi de mâncare poate fi descris astfel:

Interacțiunea cu utilizatorul: Sistemul permite utilizatorilor să interacționeze cu acesta furnizând date de intrare pentru plasarea comenzilor de mâncare. Acest lucru poate fi realizat printr-o interfață de utilizator, cum ar fi o interfață de linie de comandă sau o interfață grafică.

Crearea comenzilor: Atunci când un utilizator plasează o comandă, sistemul utilizează tiparul Builder pentru a construi un obiect FoodOrder. Utilizatorul specifică produsele alimentare, cantitățile și orice instrucțiuni speciale pentru comandă.

Procesarea comenzilor: Sistemul utilizează tiparul Singleton pentru a asigura că există o singură instanță a clasei FoodOrderService. Această clasă se ocupă de procesarea comenzilor, inclusiv de sarcini precum validarea comenzii, calcularea costului total și coordonarea plății.

Strategia de plată: Se utilizează tiparul Strategy pentru a oferi flexibilitate în alegerea diferitelor strategii de plată. Utilizatorul poate selecta metoda de plată dorită, cum ar fi cardul de credit sau PayPal, iar sistemul comută dinamic strategia de plată în funcție de aceasta.

Serviciu Proxy: Sistemul utilizează tiparul Proxy cu clasa FoodOrderServiceProxy. Proxy-ul acționează ca intermediar între client și serviciul real FoodOrderService. Acesta poate efectua operații suplimentare, cum ar fi autentificarea sau înregistrarea, înainte de a delega cererea către serviciul real.

Notificarea stării comenzilor: Se implementează tiparul Observer pentru a notifica utilizatorii cu privire la starea comenzilor lor. Sistemul menține o listă de observatori (observatori ai stării comenzii) care sunt notificați atunci când există actualizări sau modificări ale stării comenzii. Acest lucru permite utilizatorilor să fie informați cu privire la progresul comenzilor lor.

Operații Undo/Redo: Se utilizează tiparul Command pentru a încapsula operațiunile legate de comenzi în obiecte de comandă. Acest lucru permite sistemului să suporte funcționalitatea de anulare și refacere, permițând utilizatorilor să revină sau să refacă acțiunile efectuate asupra comenzilor lor.

Persistența datelor: Sistemul poate utiliza un strat de date, reprezentat de clasele FoodOrderRepository și Database, pentru a persista și recupera datele comenzilor. Acest lucru permite stocarea și recuperarea datelor în diferite sesiuni.

Prin încorporarea acestor tipare de proiectare, sistemul de comenzi de mâncare atinge modularitatea, extensibilitatea și ușurința în menținere. Acestea separă preocupările, oferă flexibilitate în alegerea metodelor de plată, susțin notificările privind starea comenzilor și permit operații de anulare/refacere pentru modificările comenzilor. Principiul general de funcționare asigură o experiență fluidă și prietenoasă utilizatorilor în ceea ce privește plasarea și procesarea comenzilor de mâncare.

# DEMONSTRAREA PE UN EXEMPLU CONCRET PRINCIPIUL DE LUCRU

**1. Interacțiunea cu utilizatorul:**

- Utilizator: "Aș dori să plasez o comandă pentru o pizza cu extra brânză și un burger fără ceapă."

- Sistem: Utilizatorul furnizează produsele alimentare, cantitățile și orice instrucțiuni speciale prin intermediul interfeței cu utilizatorul.

**2. Crearea comenzilor:**

- Sistem: Sistemul utilizează tiparul Builder pentru a construi un obiect `FoodOrder`.

- Clasa `FoodOrderBuilder` setează produsele alimentare ca "Pizza" și "Burger", cantitățile ca 1 și 1, respectiv, și instrucțiunile speciale ca "Extra brânză" și "Fără ceapă".

**3. Procesarea comenzilor:**

- Sistem: Tiparul Singleton asigură existența unei singure instanțe a clasei `FoodOrderService`.

- Clasa `FoodOrderService` primește obiectul `FoodOrder` și începe procesarea comenzii.

**4. Strategia de plată:**

- Utilizator: "Voi plăti prin PayPal."

- Sistem: Utilizatorul selectează metoda de plată ca PayPal.

- `FoodOrderService` comută dinamic la implementarea `PayPalPaymentStrategy` pentru procesarea plății.

**5. Serviciu Proxy:**

- Sistem: Sistemul utilizează tiparul Proxy cu clasa `FoodOrderServiceProxy`.

- `FoodOrderServiceProxy` efectuează operații suplimentare, cum ar fi autentificarea sau înregistrarea, înainte de a delega cererea către serviciul real `FoodOrderService`.

**6. Notificarea stării comenzilor:**

- Sistem: Tiparul Observer este implementat pentru notificarea stării comenzilor.

- Clasa `OrderStatusSubject` menține o listă de observatori ai stării comenzilor.

- `FoodOrderService` actualizează starea comenzii, iar `OrderStatusSubject` notifică observatorii.

- Utilizatorul primește o notificare: "Comanda ta este în curs de pregătire."

În timpul acestui exemplu, sistemul de comenzi de mâncare se bazează pe diverse tipare de proiectare pentru a oferi o experiență de utilizare fluidă. Sistemul construiește comenzile utilizând tiparul Builder, procesează comenzile cu tiparul Singleton, gestionează strategiile de plată cu tiparul Strategy, adaugă funcționalități cu tiparul Proxy, trimite notificări privind starea comenzilor cu tiparul Observer, susține operațiile de anulare/refacere cu tiparul Command. Aceste tipare de proiectare contribuie în mod colectiv la modularitatea, flexibilitatea și ușurința de întreținere a sistemului de comenzi de mâncare.

**Modelarea sistemului UML**

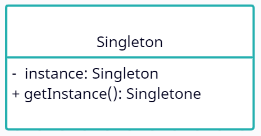


Fig.1 Diagrama UML pentru Singletone Pattern

În aceasta diagramă se afla un membru static privat numit instance, care stochează singura instanță a clasei FoodOrder. Metoda getInstance() este o metodă publică statică care oferă acces la instanța clasei FoodOrder. Aceasta se asigură că este creată și returnată doar o singură instanță a clasei FoodOrder către apelanți.

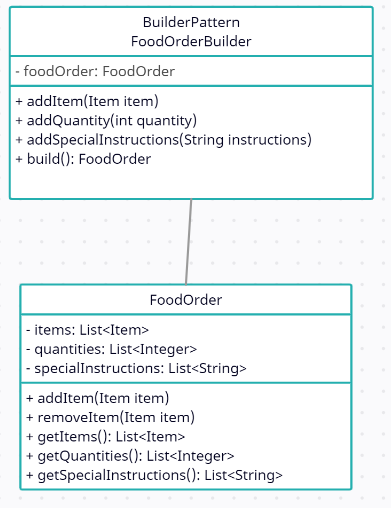


Fig.2 Diagrama UML pentru Builder Pattern (association)

În această diagramă, clasa FoodOrderBuilder reprezintă implementarea tiparului Builder. Aceasta are metode precum addItem(), addQuntity() și build() pentru a seta diferitele atribute ale unui obiect FoodOrder. Metoda getResult() returnează obiectul FoodOrder construit.Clasa FoodOrder reprezintă produsul care este construit. Aceasta are atribute precum Item, quantity și specialInstructions, accesa aceste atribute.

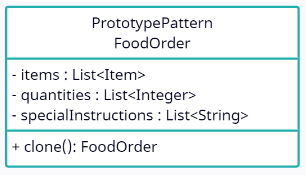


Fig.3 Diagrama UML pentru Prototype Pattern

În aceasta diagrama, clasa FoodOrder acționează ca un prototip sau o schiță pentru crearea de noi instanțe. Metoda clone() ne permite să creați copii ale obiectelor FoodOrder, păstrându-le starea în timp ce evitam costurile inutile asociate creării de obiecte noi. Prin utilizarea tiparului Prototype, putem crea eficient noi instanțe de FoodOrder prin clonarea unui obiect FoodOrder existent și modificându-i atributele specifice după necesitate, în loc să cream obiecte noi sau să copiem manual valorile atributelor.

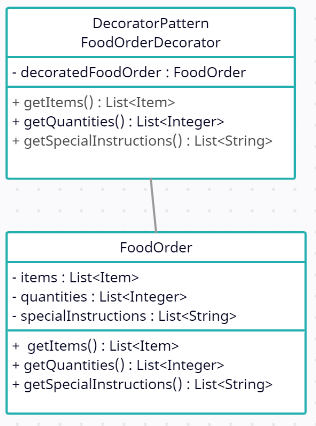


Fig.4 Diagrama UML pentru Decorator Pattern (association)

Decorator Pattern este utilizat pentru a extinde funcționalitatea clasei FoodOrder prin învelirea acesteia cu clasa FoodOrderDecorator. FoodOrderDecorator menține o referință către obiectul FoodOrder decorat și îmbunătățește comportamentul acestuia prin adăugarea de caracteristici sau modificări suplimentare. Acest tipar permite personalizarea flexibilă a funcționalității comenzilor de mâncare fără a afecta clasa FoodOrder originală sau clienții săi.

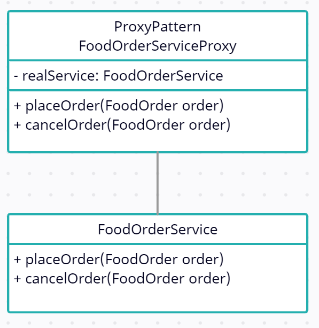


Fig.5 Diagrama UML pentru Proxy Pattern (association)

Proxy Pattern este utilizat pentru a controla accesul la serviciul FoodOrderService prin introducerea unui intermediar, denumit FoodOrderServiceProxy. Proxy-ul oferă funcționalități suplimentare sau control asupra serviciului, cum ar fi verificări de securitate sau memorare cache, fără a modifica interfața serviciului în sine.

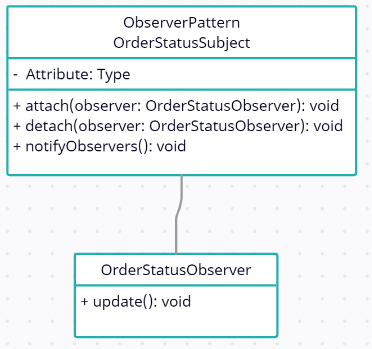


Fig.6 Diagrama UML pentru Observer Pattern (association)

Cu ajutorul Observer Pattern , mai multe obiecte OrderStatusObserver pot să se aboneze la OrderStatusSubject și să primească notificări ori de câte ori starea comenzii se schimbă. Acest lucru permite decuplarea între subiect și observatori, asigurând că modificările într-unul nu sunt strâns cuplate la ceilalți.

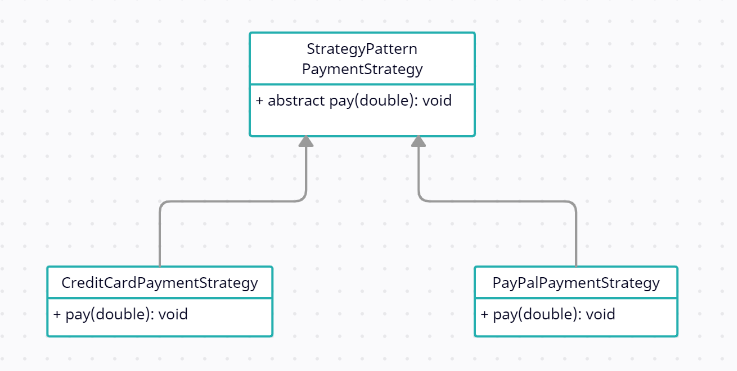


Fig. 7 Diagrama UML pentru Strategy Pattern (Composition nu era in compiler)

Strategy Pattern permite încapsularea diferitelor strategii de plată (CreditCardPaymentStrategy și PayPalPaymentStrategy) sub o interfață comună (PaymentStrategy). Acest lucru permite clienților să selecteze și să schimbe dinamic între diferite strategii de plată în funcție de cerințele lor, fără a fi strâns cuplați la implementările specifice.

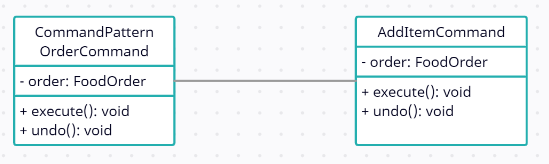


Fig. 8 Diagrama UML pentru Command Pattern (association)

Command Pattern încapsulează solicitările (comenzile) ca obiecte, permițând clienților să le parametrizeze și să le execute fără a cunoaște detaliile receptorului (obiectul care execută comanda). Interfața OrderCommand furnizează un contract comun pentru comenzile legate de o comandă, iar AddItemCommand reprezintă o comandă specifică pentru adăugarea unui element în comandă, oferind propria implementare a execuției comenzii și a operațiilor de anulare.

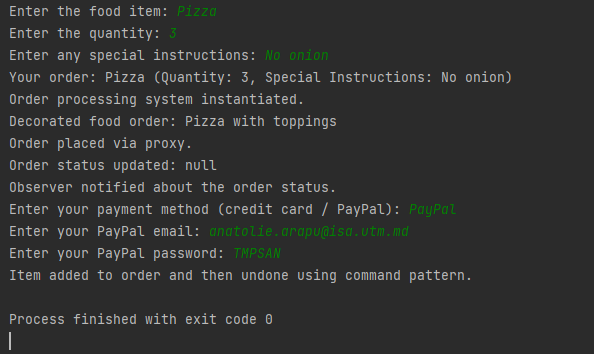


Fig. 9 Principiul de lucru al acestui program(OrderInput)

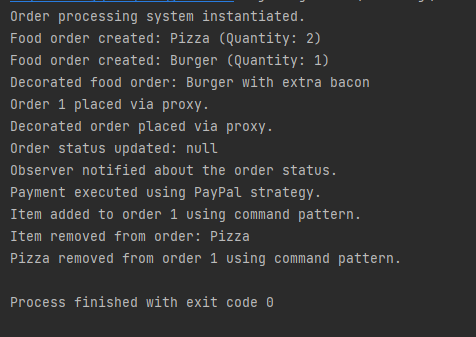


Fig. 10 Principiul de lucru al acestui program(Main)

**Concluzie**

În concluzie, proiectul de aplicație Java a implementat diverse patternuri de proiectare pentru a crea un sistem de comenzi de mâncare flexibil și extensibil. Proiectul a utilizat patternul Singleton pentru a asigura o singură instanță a clasei `FoodOrderService`, patternul Builder pentru a construi obiecte complexe de tip `Item`, patternul Prototype pentru a clona obiecte de tip `FoodOrder`, patternul Decorator pentru a adăuga funcționalități suplimentare clasei `FoodOrder`, patternul Proxy pentru a controla accesul la `FoodOrderService`, patternul Observer pentru a notifica clienții cu privire la actualizările comenzilor, patternul Strategy pentru a implementa diferite strategii de plată și patternul Command pentru a încapsula și executa comenzile legate de comandă.

Prin utilizarea acestor patternuri de proiectare, proiectul a obținut mai multe beneficii. S-a promovat reutilizabilitatea codului, modularitatea și separarea preocupărilor prin încapsularea funcționalităților specifice în clase individuale. Utilizarea patternurilor de proiectare a îmbunătățit întreținerea și extensibilitatea aplicației, permițând adăugarea ușoară a unor funcționalități noi sau modificări fără a afecta codul existent. În plus, proiectul a demonstrat importanța patternurilor de proiectare în obținerea unei arhitecturi de cod curate și bine structurate, facilitând înțelegerea, testarea și evoluția sistemului.