**,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 Cibotaru Vitalie

A verificat:

asist. univ. Mihai Gaidau

Chişinău – 2023

Cuprins :

[INTRODUCEREA 3](#_Toc136588721)

[DESCRIEREA PRICIPIULUI DE LUCRU AL TEMATICII ALESE 4](#_Toc136588723)

[DEMONSTRAREA PE UN EXEMPLU CONCRET PRINCIPIUL DE LUCRU 5](#_Toc136588724)

[MODELAREA SISTEMULUI UML 7](#_Toc136588725)

[CONCLUZIE 15](#_Toc136588726)

[BIBLIOGRAFIE 16](#_Toc136588727)

# INTRODUCEREA

Design Patterns (Modelul de Proiectare) reprezintă soluții recurente la probleme comune de proiectare de software. Acestea sunt concepte și abordări testate și recunoscute care ajută dezvoltatorii să creeze aplicații cu o arhitectură solidă și modulară. Design Patterns oferă un set de reguli și principii care permit dezvoltatorilor să abordeze problemele comune într-un mod eficient și robust. Ele se concentrează pe structura, comportamentul și relațiile dintre obiectele software-ului, oferind soluții practice pentru problemele de proiectare și implementare. În cele ce urmează, vom explora în detaliu fiecare design pattern menționat: Singleton, Builder, Adapter, Chain of Responsibility, Bridge si Command.

**Singleton** este unul dintre cele mai simple și utilizate tipuri de design pattern-uri. Scopul său este de a permite crearea unei singure instanțe a unei clase și de a oferi un punct global de acces la această instanță. Acest design pattern este util în cazurile în care dorim să ne asigurăm că există o singură instanță a unei clase în întreaga aplicație și să putem accesa această instanță din orice parte a codului.

Ideea de bază a Singleton este că clasa respectivă deține o metodă statică care returnează întotdeauna aceeași instanță a clasei. Pentru a asigura unicitatea instanței, constructorul clasei este privat, astfel încât să nu poată fi apelat direct din alte părți ale codului. În schimb, metoda statică de acces la instanță verifică dacă există deja o instanță și o returnează, altfel creează o nouă instanță și o stochează pentru utilizări ulterioare.

Utilizarea Singleton poate fi întâlnită în situații precum gestionarea resurselor partajate (de exemplu, conexiuni la baze de date), configurarea aplicației sau implementarea unui sistem de jurnalizare.

**Builder** este un design pattern creational care ne permite să construim obiecte complexe pas cu pas. Acesta este util atunci când avem nevoie să creăm obiecte cu mai multe proprietăți și opțiuni, dar dorim să evităm o metodă de construire lungă cu parametri mulți sau redundanți.

Într-un design pattern Builder, avem o clasă principală (Builder) care definește metode pentru a seta diferite proprietăți ale obiectului pe care dorim să-l construim. Apoi, avem o clasă de construcție (ConcreteBuilder) care implementează metodele de setare și stochează obiectul în construcție. În cele din urmă, avem o clasă de director (Director) care controlează secvența și logica de construcție, folosind metodele definite în clasa Builder.

Prin utilizarea acestui design pattern, putem crea obiecte complexe, pas cu pas, prin apelarea metodelor relevante din clasa Builder. Aceasta ne permite să evităm construirea unui constructor lung sau a unui constructor cu parametrii redundanți și să oferim o abordare mai flexibilă pentru construirea obiectelor.

**Adapter** este un design pattern structural care permite obiectelor incompatibile să lucreze împreună prin intermediul unui intermediar. Scopul său este de a converti interfața unei clase într-o altă interfață așteptată de clientul care o utilizează.

Imaginează-ți următorul scenariu: ai o clasă existentă (ClasaA) cu o interfață și funcționalități specifice, iar ai nevoie să o utilizezi într-un context în care se așteaptă o altă interfață (InterfațaB). În acest caz, poți folosi un Adapter care să implementeze InterfațaB și să acționeze ca un intermediar între ClasaA și codul care așteaptă InterfațaB. Adapterul va traduce apelurile către InterfațaB în apeluri către ClasaA și va gestiona adaptarea logică între cele două.

**Chain of Responsibility** este un design pattern comportamental care permite unui set de obiecte să prelucreze o cerere într-o manieră secvențială. Fiecare obiect în lanțul de responsabilitate poate decide să prelucreze cererea sau să o transmită înainte către următorul obiect din lanț.

Ideea principală a Chain of Responsibility este de a evita dependența strânsă între obiectul care inițiază cererea și obiectul care o prelucrează. În schimb, obiectele sunt conectate într-un lanț, iar prelucrarea cererii trece prin fiecare obiect în ordine până când este găsit un obiect care o poate gestiona. Acest design pattern este util atunci când dorim să obținem o flexibilitate și o extensibilitate sporită în prelucrarea cererilor.

De exemplu, putem avea un lanț de obiecte care prelucrează cereri HTTP și fiecare obiect din lanț se ocupă de un anumit tip de cerere sau de o anumită metodă HTTP.

**Bridge** este un design pattern structural care separă o ierarhie de clase din două dimensiuni diferite în două ierarhii separate, permițându-le să evolueze independent una de cealaltă. Acesta este util atunci când dorim să evităm o ierarhie extinsă de clase prin cuplarea puternică și să permitem modificări separate pentru două aspecte diferite ale unei clase.

În design pattern-ul Bridge, avem două ierarhii separate: o ierarhie de clase abstracte sau interfețe care definește funcționalitatea principală, și o ierarhie de implementări care definește detaliile concrete ale acelei funcționalități. Prin intermediul unei abstracții (Bridge), obiectele din prima ierarhie interacționează cu obiectele din a doua ierarhie.

Prin separarea acestor două dimensiuni, putem obține o flexibilitate mai mare în extinderea și modificarea funcționalității, evitând creșterea complexității într-o singură ierarhie.

**Command** este un design pattern comportamental care încapsulează o cerere sub formă de obiect, permițând astfel parametrizarea clienților cu diferite cereri și gestionarea acestora în mod flexibil. Scopul său este de a separa emitentul unei cereri de obiectul care o prelucrează.

În acest design pattern, avem un obiect Command care reprezintă o anumită cerere și care este responsabil pentru prelucrarea acesteia. Emitentul cererii (Clientul) creează un obiect Command și îl transmite către un obiect Invoker, care este responsabil pentru gestionarea și execuția cererii. Obiectul Invoker poate păstra o listă de obiecte

Command și le poate executa într-o anumită ordine sau în funcție de anumite criterii. Command oferă un nivel înalt de flexibilitate, deoarece putem schimba sau extinde comportamentul aplicației prin simpla creare și înlocuire a obiectelor Command fără a afecta restul sistemului.

# DESCRIEREA PRICIPIULUI DE LUCRU AL TEMATICII ALESE

Food Shop Delivery este o tematică care poate beneficia de aplicarea mai multor design pattern-uri. Voi descrie principiul de lucru al tematicii și modul în care pot fi utilizate design pattern-urile menționate pentru a rezolva problemele specifice acesteia.

Principiul de lucru al tematicii Food Delivery pentru modelele Singleton, Builder, Adapter, Chain of Responsibility, Bridge și Command poate fi descris astfel:

1. Singleton Pattern:
   * Se folosește pentru a asigura existența unei singure instanțe a unei clase în întreaga aplicație.
   * În contextul Food Delivery, acest model ar putea fi utilizat pentru a gestiona un singur obiect de tip Meniu sau un obiect de tip Cos de Cumpărături, care trebuie să fie accesibil și actualizat de mai multe componente ale sistemului.
2. Builder Pattern:
   * Este utilizat pentru a crea obiecte complexe pas cu pas, oferind flexibilitate și separare a logicii de construcție
   * În cadrul Food Delivery, acest model poate fi aplicat pentru a construi comenzi personalizate, unde se pot adăuga treptat produsele, adresele de livrare și alte detalii specifice comenzii.
3. Adapter Pattern:
   * Ajută la conectarea două interfețe incompatibile între ele, oferind o abordare comună pentru a le utiliza împreună.
   * În contextul Food Delivery, un adaptor ar putea fi folosit pentru a permite integrarea cu diferite servicii de plăți sau de livrare, care au interfețe diferite, dar trebuie să comunice cu sistemul Food Delivery.
4. Chain of Responsibility Pattern:
   * Acest model permite definirea unei serii de obiecte care vor prelua și trata o solicitare în ordine, până când una dintre ele o va gestiona.
   * În cazul Food Delivery, Chain of Responsibility poate fi aplicat pentru a gestiona etapele de prelucrare a comenzilor, precum verificarea stocului, validarea adresei de livrare și plasarea comenzii către restaurant.
5. Bridge Pattern:
   * Este utilizat pentru a separa o abstractizare de implementarea sa, permițându-le să evolueze independent.
   * În domeniul Food Delivery, Bridge poate fi aplicat pentru a separa procesul de livrare de cel de plasare a comenzii, permițând sistemului să funcționeze cu diferite implementări de servicii de livrare.
6. Command Pattern:
   * Acest model permite încapsularea unei acțiuni într-un obiect, permițând astfel parametrizarea și executarea ulterioară a acelei acțiuni.
   * În cazul Food Delivery, Command poate fi utilizat pentru a reprezenta diferitele acțiuni disponibile, cum ar fi adăugarea unui produs în coșul de cumpărături sau plasarea comenzii, permitând astfel gestionarea ușoară și flexibilă a acestor acțiuni.

Aceste modele pot fi folosite în combinație pentru a crea o arhitectură robustă și flexibilă pentru un sistem de livrare de mâncare (Food Delivery), oferind o structură clară și ușor de întreținut, precum și extensibilitate pentru a adăuga noi funcționalități în viitor.

# DEMONSTRAREA PE UN EXEMPLU CONCRET PRINCIPIUL DE LUCRU

Imaginați-vă că sunteți un utilizator care dorește să folosească aplicația pentru a cumpăra produse alimentare online. În primul rând, deschideți aplicația și vedeți interfața principală a magazinului de produse alimentare.

În interfața principală, aveți câteva câmpuri de intrare pentru a adăuga un produs nou în coșul de cumpărături. Introduceți numele produsului, descrierea și selectați cantitatea și opțiunile disponibile (cum ar fi dimensiunea sau gustul) din listele afișate. După ce ați completat toate informațiile, apăsați butonul "Adaugă în coș". Aici intervine principiul de lucru al design pattern-ului Builder. Clasa ProductBuilder preia datele introduse de utilizator și le utilizează pentru a construi un obiect de tip Product. Cu ajutorul metodelor din clasa ProductBuilder, precum setName, setDescription și setOptions, specificați atributele dorite pentru obiectul Product.

La final, apelați metoda build pentru a obține obiectul Product finalizat. Acum, obiectul Product este creat și este adăugat în coșul de cumpărături, care implementează principiul de lucru al design pattern-ului Singleton. Aplicația utilizează instanța unică a clasei CartManager, obținută prin intermediul metodei statice getInstance, pentru a adăuga produsul nou-creat în lista de produse existente din coș. După adăugarea produsului, câmpurile de intrare pentru nume, descriere și opțiuni se golesc automat, pregătindu-vă pentru adăugarea unui nou produs.

În același timp, coșul de cumpărături din interfață se actualizează pentru a afișa toate produsele existente, inclusiv cel nou adăugat. Pentru a elimina un produs din coș, puteți utiliza funcționalitatea de eliminare oferită de aplicație. După ce apăsați butonul "Elimină" lângă un produs, principiul de lucru al design pattern-ului Adapter intră în joc. Clasa ProductAdapter asigură adaptarea interfeței obiectului Product la o formă mai convenabilă pentru gestionarea în cadrul aplicației.

De exemplu, metoda getName a clasei ProductAdapter vă permite să obțineți numele produsului pentru afișare sau alte scopuri. Aplicația noastră de cumpărături online utilizează și alte design pattern-uri, cum ar fi Chain of Responsibility pentru gestionarea fluxului de plasare a comenzilor, Bridge pentru separarea interfeței utilizatorului de implementarea efectivă și Command pentru gestionarea comenzilor utilizatorului. Aceste pattern-uri oferă flexibilitate, extensibilitate și modularitate în implementarea și gestionarea cumpărăturilor.

Pentru a exemplifica principiul de lucru al acestei aplicații, vom folosi un scenariu simplu:

1. Utilizatorul deschide aplicația pentru cumpărături alimentare online.
2. Pe ecranul principal, utilizatorul poate vedea un titlu mare "Magazin de produse alimentare" și un formular pentru adăugarea de produse în coș.
3. Utilizatorul completează câmpurile "Nume", "Descriere" și selectează o "Cantitate" pentru produs.
4. Utilizatorul apasă butonul "Adaugă în coș" pentru a adăuga produsul.
5. Produsul nou este creat utilizând Builder Pattern și este adăugat în coșul de cumpărături folosind Singleton Pattern.
6. Formularul se curăță, iar lista de produse din coș este actualizată automat.
7. Utilizatorul poate vedea lista de produse din coș în partea de jos a ecranului, cu informații precum numele, descrierea și cantitatea fiecărui produs.
8. Utilizatorul poate derula în sus și în jos pentru a vizualiza toate produsele din coș
9. Pentru a elimina un produs, utilizatorul apasă pe butonul "Elimină" lângă produsul respectiv.
10. Produsul este eliminat din coș și lista de produse se actualizează automat în interfață.
11. Utilizatorul poate adăuga, vizualiza și elimina produse în mod repetat, iar lista de produse din coș se actualizează în timp real.

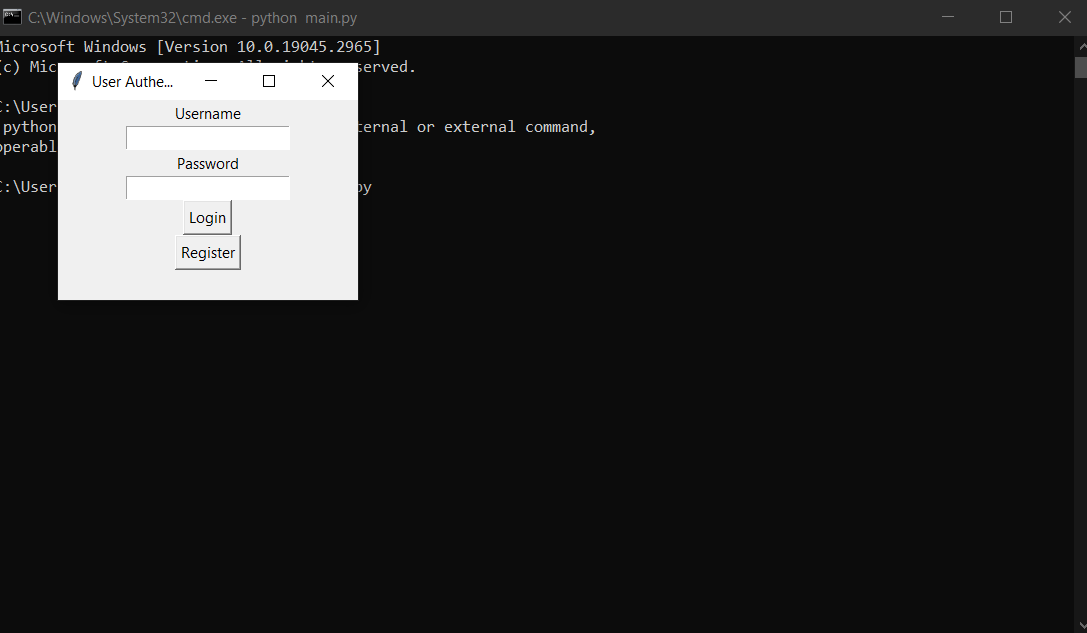


Figura 1.1 Interfata User Authentification

Fereastra în cauză furnizează utilizatorului trei opțiuni esențiale pentru gestionarea contului său. Utilizatorul poate alege să se autentifice folosind informațiile de autentificare existente, să își creeze o parolă nouă sau să acceseze contul în mod direct. Această interfață intuitivă și flexibilă permite utilizatorului să aleagă opțiunea potrivită în funcție de nevoile și preferințele sale, facilitând astfel experiența de utilizare.

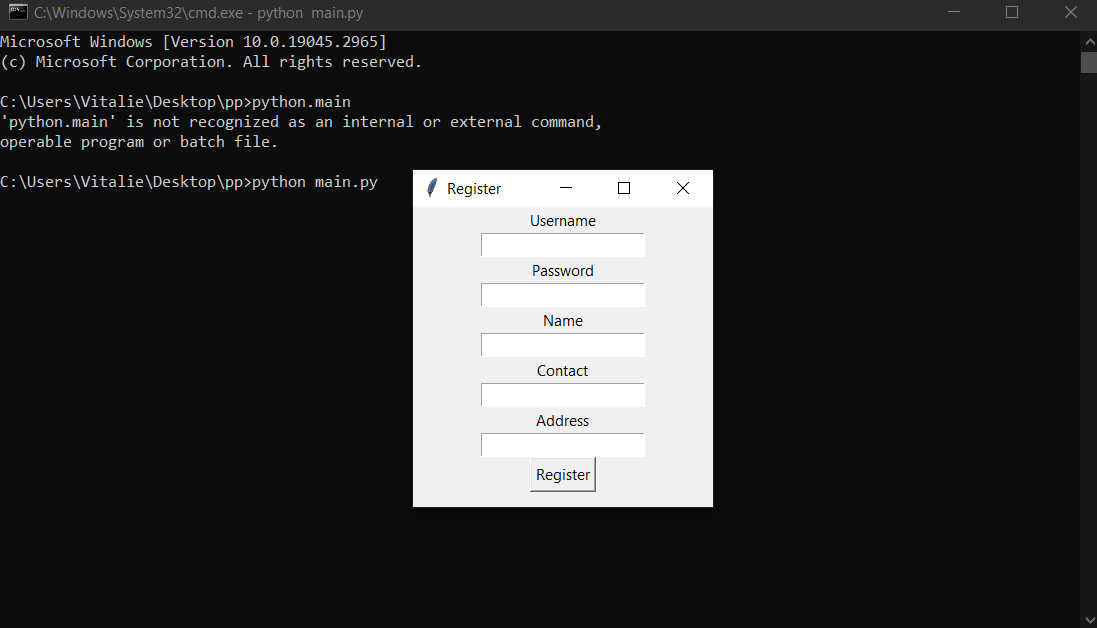


Figura 1.2 Fereastra Register

Fereastra respectivă oferă utilizatorului posibilitatea de a-și crea un cont personalizat. Prin intermediul interfeței, utilizatorul poate introduce informațiile necesare, cum ar fi numele de utilizator, parola, numele său, detalii de contact și adresa. Această funcționalitate permite utilizatorului să-și configureze contul în mod adecvat, asigurând în același timp securitatea și personalizarea. Prin furnizarea acestor opțiuni, fereastra facilitează procesul de înregistrare și oferă o experiență utilizator plăcută și adaptată nevoilor individuale.



Figura 1.3 Inregistrarea conturile

În această secțiune, sunt afișate conturile înregistrate în platforma noastră. Utilizatorii care s-au înregistrat pot fi identificați în această listă, oferind o vedere de ansamblu asupra numărului și detaliilor conturilor existente. Această funcționalitate permite o gestionare eficientă a utilizatorilor și furnizează informații relevante pentru administrarea programului nostru. Prin urmărirea și vizualizarea acestor conturi înregistrate, putem evalua succesul și creșterea platformei noastre în funcție de numărul de utilizatori activi.

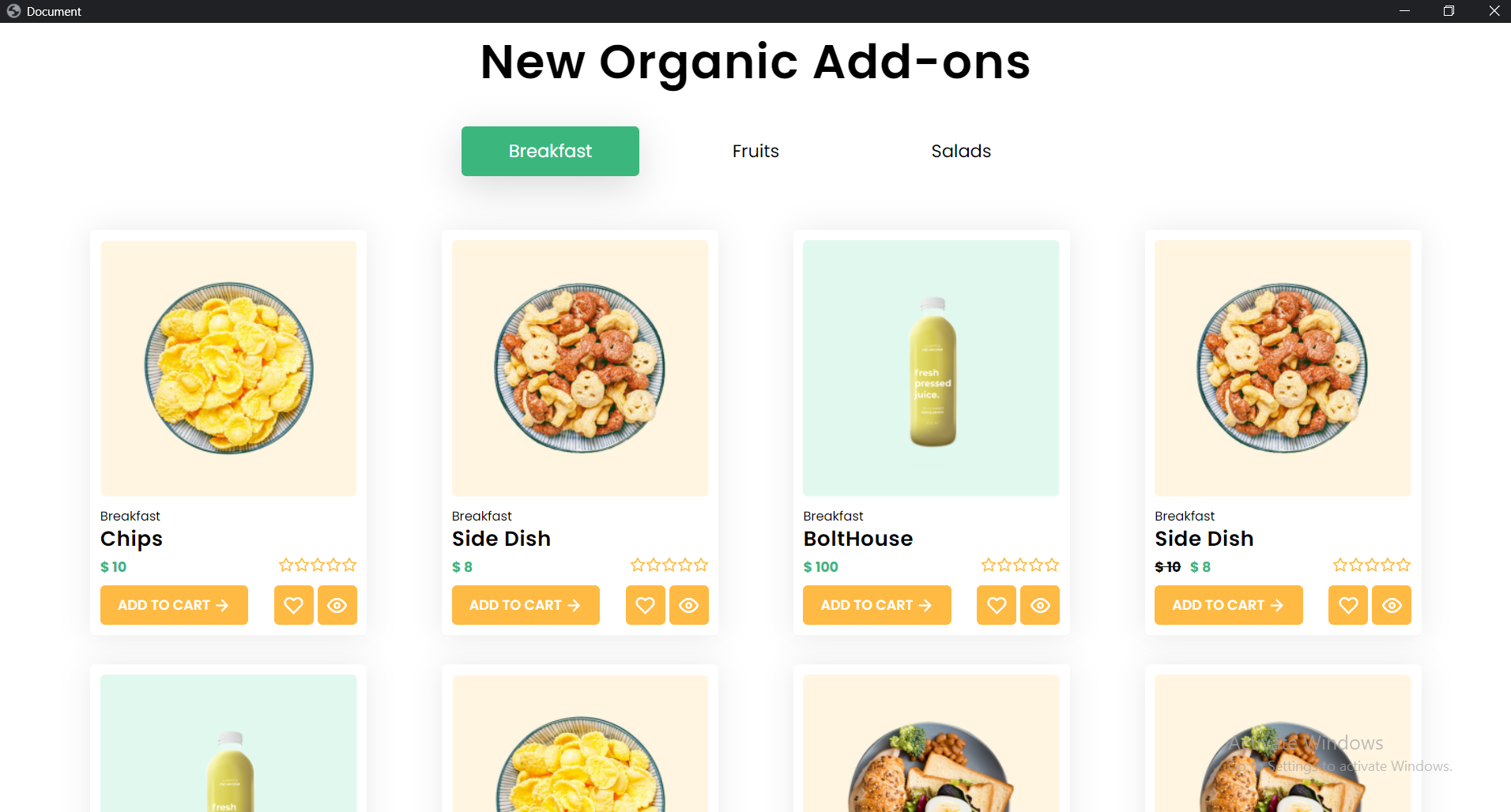


Figura 1.3 Interfata aplicatiei

După ce utilizatorul introduce username-ul și parola, va fi direcționat către platformă, unde îi vor fi disponibile opțiunile de comandă pentru mâncare. Pe această platformă, utilizatorul poate explora meniul, adăuga produse în coșul de cumpărături și finaliza comanda. Această funcționalitate îi oferă utilizatorului posibilitatea de a comanda rapid și comod mâncarea preferată prin intermediul platformei noastre.

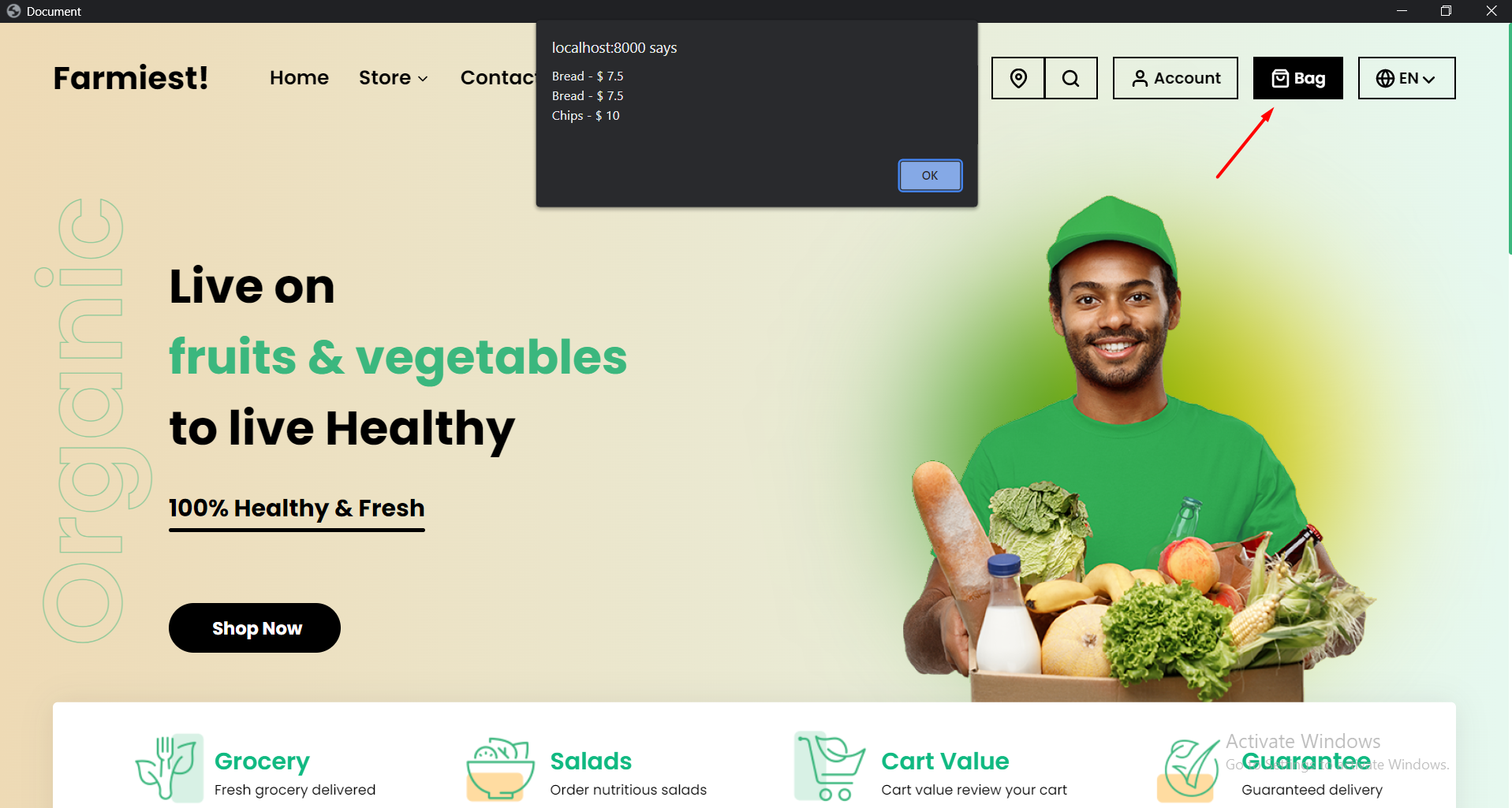


Figura 1.3 Selectarea produselor

Pe platforma noastră, utilizatorii pot selecta produsele dorite pentru comandă, iar apoi pot vizualiza lista cu produsele alese apăsând pe butonul "Coș". Această funcționalitate permite utilizatorilor să vadă în mod clar mancarea pe care au selectat-o înainte de a finaliza comanda.

# MODELAREA SISTEMULUI UML

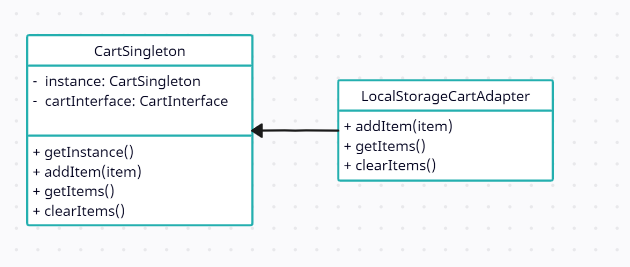


Figura 2.1 - Singleton Pattern

Singleton pattern am aplicat pentru a evidenția legătura dintre două clase, și anume CartSingleton, care se bazează pe o instanță și o interfață, și este asociată cu sarcinile de adăugare a unui element (addItem), obținerea elementelor (getItems) și eliminarea elementelor (clearItems).

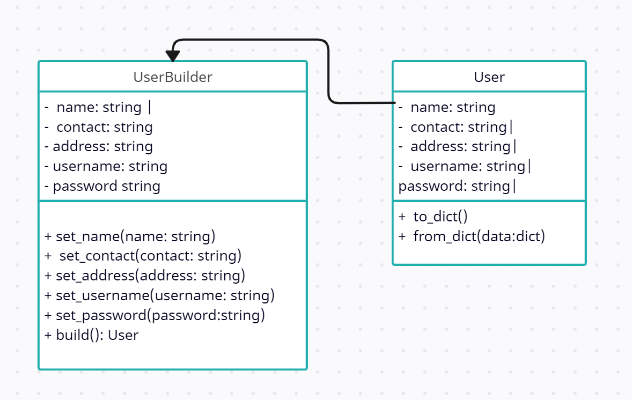


Figura 2.2 Builder Pattern

Builder pattern este utilizat În această situație între două clase, UserBuilder și User, care sunt conectate prin intermediul parametrilor comuni, cum ar fi numele (name), informațiile de contact (contact), adresa (address), numele de utilizator (username) și parola (password). Clasa UserBuilder este responsabilă de construirea și inițializarea obiectelor de tip User. Aceasta furnizează o abordare mai flexibilă și mai modulară în procesul de creare a obiectelor User, în comparație cu o simplă utilizare a constructorului obiectului User..

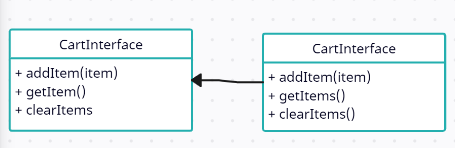


Figura 2.3 Bridge

În acest caz, utilizăm pattern-ul Bridge, care este reprezentat prin intermediul legăturii între două interfețe CartInterface. Acest pattern are la bază principiile de bază ale pattern-ului Bridge, care implică separarea unei abstracții de implementare pentru a permite modificarea independentă a acestora. Prin implementarea pattern-ului Bridge în contextul nostru, avem două interfețe, CartInterface1 și CartInterface2, care sunt legate între ele și împărtășesc aceleași funcționalități de bază, cum ar fi addItem, getItems și clearItems..

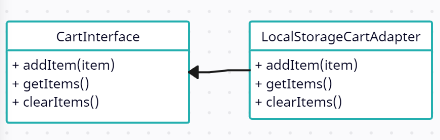


Figura 2.4 Adapter Pattern

În acest context, am aplicat pattern-ul Adapter pentru a crea o legătură între CartInterface și clasa LocalStorageCartAdapter. CartInterface definește metodele addItem, getItems și clearItems, iar LocalStorageCartAdapter implementează aceleași metode. Pattern-ul Adapter este folosit atunci când avem două componente cu interfețe incompatibile, dar dorim să le conectăm. În acest caz, CartInterface și LocalStorageCartAdapter sunt componente separate, dar avem nevoie ca acestea să comunice și să ofere aceleași funcționalități de bază..

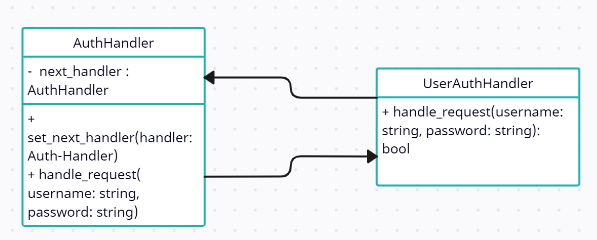


Figura 2.5 Chain of Responsibility

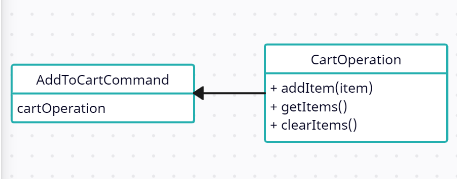
În această situație, am aplicat pattern-ul Chain of Responsibility pentru a crea o legătură între două clase, AuthHandler și UserAuthHandler. Pattern-ul Chain of Responsibility este folosit atunci când avem mai multe obiecte care pot manipula o anumită solicitare, iar decizia privind manipularea solicitării este luată la momentul rulării. Clasa AuthHandler reprezintă clasa de bază pentru manipularea autentificării și conține metodele comune necesare pentru acest proces. 

Figura 2.6 Command

În acest caz, am aplicat pattern-ul Command pentru a crea o legătură între clasa AddToCartCommand și clasa CartOperation, care împărtășește trei task-uri comune: addItem, getItems și clearItems. Pattern-ul Command este utilizat atunci când dorim să encapsulăm o acțiune sau o operațiune într-un obiect, permițând astfel manipularea, anularea sau refacerea acelei acțiuni într-un mod flexibil și modular..

# CONCLUZIE

În concluzie, pattern-urile de proiectare Singleton, Builder, Adapter, Chain of Responsibility, Bridge și Command sunt instrumente puternice și versatile utilizate în dezvoltarea software-ului pentru a rezolva diverse probleme și a îmbunătăți structura și modularitatea aplicațiilor.

Singleton oferă asigurarea unei singure instanțe a unei clase și un punct global de acces la aceasta, util în situațiile în care trebuie să existe o singură instanță a unei clase și să se limiteze crearea de obiecte.

Builder permite crearea și configurarea flexibilă a obiectelor complexe, separând procesul de construcție de reprezentarea finală a obiectului și oferind abordări extensibile și flexibile în gestionarea construcției obiectelor.

Adapter facilitează comunicarea între clase cu interfețe incompatibile, adaptând interfața unei clase existente pentru a fi compatibilă cu o altă interfață, asigurând interoperabilitatea și reutilizarea componentelor software.

Chain of Responsibility permite gestionarea flexibilă și extensibilă a cererilor prin intermediul unui lanț de obiecte, fiecare obiect având posibilitatea de a manipula cererea sau de a o pasa mai departe către următorul obiect în lanț, facilitând decuplarea și modularitatea în tratarea cererilor.

Bridge separă abstractizarea de implementare, permițându-le să evolueze independent și facilitând schimbarea și adăugarea de implementări fără a afecta codul abstractizării, promovând astfel flexibilitatea, modularitatea și extensibilitatea în proiectare.

Command permite încapsularea unei cereri sub formă de obiect, permițând astfel parametrizarea și programarea cererii, oferind flexibilitate și extensibilitate în gestionarea și manipularea operațiunilor, precum și implementarea funcționalităților de anulare și refacere.

Utilizarea corectă și judicioasă a acestor pattern-uri de proiectare poate contribui la dezvoltarea unui software modular, flexibil, ușor de întreținut și extensibil, aducând beneficii semnificative în dezvoltarea și evoluția aplicațiilor software.

# BIBLIOGRAFIE

1. "Design Patterns: Elements of Reusable Object-Oriented Software" de Erich Gamma, Richard Helm, Ralph Johnson și John Vlissides.
2. "Head First Design Patterns" de Eric Freeman și Elisabeth Robson.
3. "Design Patterns in Ruby" de Russ Olsen.
4. "Design Patterns: Elements of Reusable Object-Oriented Software" (cartea clasică a design pattern-urilor).
5. "Refactoring: Improving the Design of Existing Code" de Martin Fowler.
6. Site-ul oficial al Gang of Four (GOF): <http://www.oodesign.com/>
7. Site-ul oficial al Design Patterns: <https://www.designpatterns.com/>
8. Tutorialspoint: <https://www.tutorialspoint.com/design_pattern/index.htm>