**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 un sistem de băuturi”

A efectuat st.gr. TI-204 Calugher Ion

A verificat asis.univ. Gaidau Mihai

Chișinău 2023

Cuprins

[**Introducere** 3](#_Toc137074640)

[**1.Proiectarea aplicației** 6](#_Toc137074641)

[**1.1 Descrierea problemei** 6](#_Toc137074642)

[**1.2 Șabloane Creaționale** 7](#_Toc137074643)

[**1.3 Șabloane Structurale** 12](#_Toc137074644)

[**1.4 Șabloane comportamentale** 16](#_Toc137074645)

[**2. Descrierea principiului de lucru** 18](#_Toc137074646)

[**Concluzii:** 20](#_Toc137074647)

[**Bibliografie** 21](#_Toc137074648)

# **Introducere**

Șabloanele de proiectare sunt soluții tipice pentru problemele frecvente întâlnite în proiectarea de software. Acestea reprezintă planuri pre-făcute și personalizabile, care pot fi utilizate pentru a rezolva probleme recurente de design în codul dvs. Nu puteți pur și simplu copia un model și aplicați-l în programul dvs., așa cum ați putea face cu funcții sau biblioteci. Un model nu este o bucată specifică de cod, ci un concept general pentru a rezolva o anumită problemă. Detaliile specifice ale modelului pot fi urmărite și implementate pentru a se potrivi cu necesitățile propriului program. Uneori, modelele sunt confundate cu algoritmii, deoarece ambele descriu soluții tipice pentru anumite probleme cunoscute. Cu toate acestea, în timp ce un algoritm definește întotdeauna un set clar de acțiuni pentru a atinge un anumit obiectiv, un model este o descriere mai abstractă a unei soluții. Codul aplicat aceluiași model în două programe diferite poate fi diferit. O analogie ar fi că un algoritm este similar cu o rețetă de gătit, în timp ce un model seamănă mai mult cu un șablon sau o machetă, unde puteți vedea rezultatul și caracteristicile sale.

Un model de proiectare este de obicei descris în mod formal, pentru a permite oamenilor să-l reproducă în diferite contexte.

De obicei, o descriere a unui model conține următoarele secțiuni:

* Intenția modelului, care descrie pe scurt problema și soluția pe care modelul o adresează.
* Motivația, care explică în detaliu problema și soluția pe care modelul le facilitează.
* Structura claselor, care arată cum sunt organizate și cum interacționează diferitele părți ale modelului.

Un exemplu de cod, de obicei într-un limbaj de programare cunoscut, care ajută la înțelegerea mai ușoară a ideii din spatele modelului. Unele cataloage de modele pot enumera și alte detalii utile, cum ar fi aplicabilitatea modelului, etapele de implementare și relațiile cu alte modele.

Un designer fără experiență poate fi copleșit de multitudinea de opțiuni disponibile și poate avea tendința de a se întoarce la tehnicile non-obiectuale pe care le-a utilizat în trecut. În schimb, un designer experimentat știe că nu trebuie să rezolve fiecare problemă de la zero, ci poate reutiliza soluții din proiecte anterioare. Atunci când găsește o soluție bună, o va utiliza în mod constant. Această experiență face parte din ceea ce conferă unui designer statutul de expert.

În cadrul sistemelor orientate pe obiecte realizate profesional, pot fi identificate modele de clase și modele de obiecte care rezolvă probleme specifice și fac sistemele mai flexibile, mai elegante și mai reutilizabile. Un designer familiarizat cu aceste modele va putea să le aplice rapid, fără a le reinventa de fiecare dată. Modelele de proiectare sunt, de fapt, o înregistrare a experienței în domeniul proiectării sistemelor orientate pe obiecte.

Modelele de proiectare pot fi clasificate în funcție de scopul lor:

* Modele creationale se referă la modul de creare a obiectelor.
* Modelele structurale se referă la compoziția claselor sau a obiectelor.
* Modelele comportamentale caracterizează modul în care obiectele și clasele interacționează și își distribuie responsabilitățile.

De asemenea, modelele pot fi clasificate în funcție de domeniul lor de aplicare:

* Modelele obiectelor se referă la relațiile dintre obiecte, care au un caracter dinamic.
* Modelele creationale ale obiectelor acoperă situațiile în care o parte a procesului de creare a unui obiect este delegată unui alt obiect.
* Modelele structurale ale obiectelor descriu modul în care obiectele sunt asamblate.
* Modelele comportamentale ale obiectelor descriu modul în care un grup de obiecte colaborează pentru a îndeplini o sarcină care nu poate fi realizată de un singur obiect.
* Modelele claselor se referă la relațiile dintre clase, care sunt stabilite prin moștenire și sunt statice (fixate la compilare).
* Modelele creationale ale claselor acoperă situațiile în care o parte a procesului de creare a unui obiect este delegată subclaselor.
* Modelele structurale ale claselor descriu modul de utilizare a moștenirii în compunerea claselor.

|  |  |
| --- | --- |
| **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 |

|  |  |
| --- | --- |
| **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 |

Tabelul 1.1 Șabloane creaționale, Șabloane structurale și Șabloane de comportament

# **1.Proiectarea aplicației**

## **1.1 Descrierea problemei**

Aplicația mea oferă o platformă de gestionare a băuturilor, care permite adăugarea, ștergerea și sortarea acestora în funcție de categorie. Aceasta utilizează mai multe modele de proiectare pentru a asigura o funcționalitate eficientă și o experiență coerentă pentru utilizatori.

Scopul aplicației mele este de a oferi utilizatorilor un instrument simplu și intuitiv pentru gestionarea băuturilor. Prin intermediul acesteia, utilizatorii pot adăuga noi băuturi în sistem, le pot șterge și pot sorta băuturile existente în funcție de categorie.

Acest proiect urmărește să ofere o soluție eficientă și ușor de utilizat pentru gestionarea băuturilor. Prin utilizarea modelelor de proiectare:

* **Singleton**
* **Prototype**
* **Builder**
* **Facade**
* **Decorator**
* **Iterator**

aplicația mea oferă o abordare modulară și extensibilă pentru gestionarea băuturilor. Aceasta permite adăugarea și eliminarea ușoară a funcționalităților, precum și extinderea sistemului pentru a suporta alte operații sau caracteristici legate de gestionarea băuturilor.

În ansamblu, aplicația mea își propune să ofere o experiență simplă, coerentă și eficientă pentru gestionarea băuturilor, permițând utilizatorilor să își organizeze și să își gestioneze colecția de băuturi într-un mod comod și personalizat.

## **1.2 Șabloane Creaționale**

Am implementat următoarele 3 șabloane creaționale: Singleton, Prototype și Builder.

Pentru început va fi descris Singleton:

**Singleton** este un model de design creațional care se concentrează pe garantarea faptului că o clasă are o singură instanță și oferă un punct global de acces la acea instanță.



Figura 1.1 Diagrama UML pentru Singleton

**Implementare –** Sablonul Singleton a fost implementat în clasa DrinkSystem prin intermediul următoarelor aspecte:

Constructorul clasei DrinkSystem utilizează o verificare a existenței instanței prin intermediul variabilei statice DrinkSystem.instance. Dacă deja există o instanță, constructorul returnează acea instanță în loc să creeze o nouă instanță.

Dacă nu există o instanță a clasei DrinkSystem, constructorul continuă prin crearea noii instanțe a clasei și inițializarea variabilei drinks ca un array gol.

Apoi, instanța nou-creată este asignată la variabila statică DrinkSystem.instance, asigurându-se astfel că această instanță unică este disponibilă în întregul cod.

În cele din urmă, constructorul returnează instanța creată sau instanța existentă, asigurându-se că se utilizează întotdeauna aceeași instanță a clasei DrinkSystem.

Astfel, în implementarea mea, Singleton Pattern permite obținerea unei singure instanțe a clasei DrinkSystem, indiferent de câte încercări de creare a instanței se fac în cod. Aceasta asigură că se lucrează mereu cu aceeași instanță și oferă acces controlat și consistent la metodele și proprietățile clasei DrinkSystem în întregul sistem.

// Singleton Pattern

class DrinkSystem {

constructor() {

if (DrinkSystem.instance) {

return DrinkSystem.instance;

}

this.drinks = [];

DrinkSystem.instance = this;

return this;

}

addDrink(drink) {

this.drinks.push(drink);

}

deleteDrink(drinkName) {

this.drinks = this.drinks.filter(drink => drink.name !== drinkName);

}

sortByCategory(category) {

const sortedDrinks = this.drinks.filter(drink => drink.category === category);

sortedDrinks.sort((a, b) => a.name.localeCompare(b.name));

return sortedDrinks;

}

}

După 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**.**

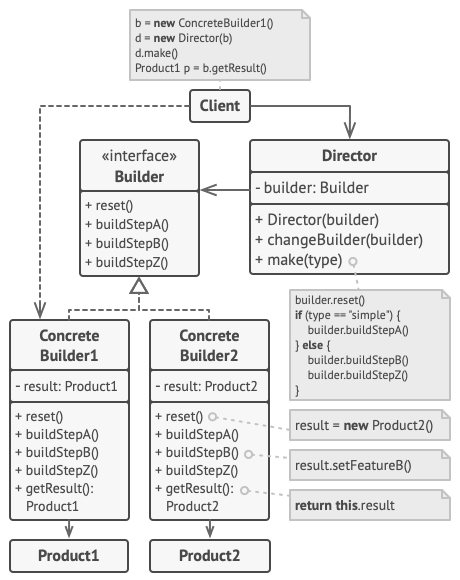


Figura 1.2 Șablonul Builder

Sablonul **Builder** a fost implementat în clasa DrinkBuilder pentru a construi obiecte de tip Drink cu mai multe proprietăți. Implementarea acestui sablon implică următoarele aspecte:

Clasa DrinkBuilder conține o serie de metode de setare (setName, setCategory) care permit specificarea valorilor pentru proprietățile obiectului Drink. Aceste metode returnează întotdeauna instanța curentă a DrinkBuilder, ceea ce permite apeluri în lanț (chaining) ale metodelor.

Clasa DrinkBuilder conține metoda build care creează și returnează un obiect Drink cu valorile specificate anterior prin metodele de setare. Aceasta utilizează valorile stocate în proprietățile name și category ale instanței curente a DrinkBuilder.

Utilizarea sablonului Builder în cod se face prin intermediul metodelor de setare ale DrinkBuilder, care permit specificarea detaliilor obiectului Drink, și prin apelul metodei build pentru a obține obiectul final.

În implementarea mea, sablonul Builder permite crearea ușoară a obiectelor de tip Drink cu mai multe proprietăți. Prin utilizarea metodelor de setare și apelul metodei build, poți crea obiecte Drink cu proprietățile dorite fără a fi nevoie să utilizezi un constructor cu mai mulți parametri sau să setezi manual fiecare proprietate. Aceasta facilitează construirea obiectelor și îmbunătățește claritatea și flexibilitatea codului.

// Builder Pattern

class DrinkBuilder {

constructor() {

this.name = '';

this.category = '';

}

setName(name) {

this.name = name;

return this;

}

setCategory(category) {

this.category = category;

return this;

}

build() {

return new Drink(this.name, this.category);

}

}

**Prototype.**

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

// Prototype Pattern

class Drink {

constructor(name, category) {

this.name = name;

this.category = category;

}

clone() {

return new Drink(this.name, this.category);

}

}

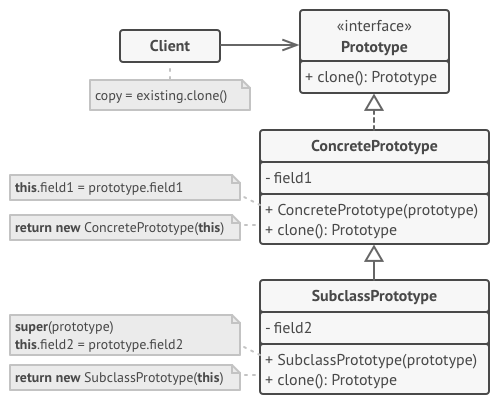


Figura 1.3 Șablonul Prototype

Clasa Drink conține un constructor care primește valorile pentru proprietățile name și category ale unui obiect Drink.

Clasa Drink conține metoda clone care returnează o copie a obiectului curent. În interiorul acestei metode, se creează un nou obiect Drink și se atribuie valorile din obiectul curent pentru proprietățile name și category noului obiect.

Utilizarea sablonului Prototype se face prin apelul metodei clone pe un obiect existent de tip Drink, obținând astfel o copie identică a acestuia.

În implementarea ta, sablonul Prototype permite crearea de copii ale obiectelor de tip Drink, oferind o modalitate eficientă de a obține obiecte similare. Prin apelul metodei clone pe un obiect Drink, poți crea rapid copii ale acestuia cu aceleași valori pentru proprietățile name și category. Acest lucru este util atunci când dorești să creezi noi obiecte bazate pe unul deja existent, evitând astfel construcția unui obiect complet nou de fiecare dată.

## **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: Facade și Decorator.

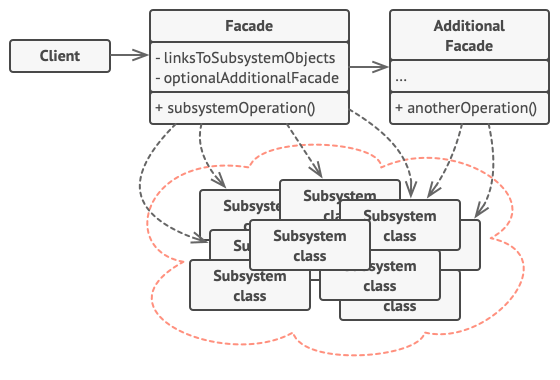
**Facade** este un model de design structural care oferă o interfață simplificată și unificată către un sistem complex sau un set de clase, ascunzând detaliile complexității din spatele unei singure interfețe. Scopul principal al modelului de design Facade este de a facilita utilizarea și interacțiunea cu un sistem, oferind o abstracție simplă și ușor de utilizat. 

Figura 1.4 Șablonul Facade

Sablonul Facade a fost implementat în clasa DrinkFacade pentru a oferi o interfață simplificată și unificată către sistemul de gestionare a băuturilor (DrinkSystem). Implementarea acestui sablon implică următoarele aspecte:

Clasa DrinkFacade conține o instanță a clasei DrinkSystem, pe care o utilizează pentru a accesa și manipula băuturile.

Metodele publice din clasa DrinkFacade oferă funcționalități specifice pentru operațiile de adăugare, ștergere și sortare a băuturilor. Aceste metode abstractizează detaliile complexe ale interacțiunii cu DrinkSystem și oferă o interfață simplă și ușor de utilizat pentru utilizatorii clasei DrinkFacade.

Atunci când se apelează metodele din DrinkFacade, acestea folosesc obiectul DrinkSystem intern pentru a efectua operațiile corespunzătoare. Utilizatorii clasei DrinkFacade nu trebuie să cunoască detaliile interne ale DrinkSystem și pot folosi interfața simplificată oferită de DrinkFacade.

Implementarea sablonului Facade permite ascunderea complexității sistemului de gestionare a băuturilor și oferă o interfață mai prietenoasă și mai simplă pentru utilizatori. Prin intermediul clasei DrinkFacade, utilizatorii pot adăuga, șterge și sorta băuturi fără a cunoaște detaliile interne ale implementării DrinkSystem. Aceasta îmbunătățește modularitatea și ușurința de utilizare a codului.

// Facade Pattern

class DrinkFacade {

constructor() {

this.drinkSystem = new DrinkSystem();

}

addDrink(name, category) {

const drink = new DrinkBuilder()

.setName(name)

.setCategory(category)

.build();

this.drinkSystem.addDrink(drink);

console.log(`Drink "${name}" added to the system.`);

}

deleteDrink(name) {

this.drinkSystem.deleteDrink(name);

console.log(`Drink "${name}" deleted from the system.`);

}

sortDrinksByCategory(category) {

const sortedDrinks = this.drinkSystem.sortByCategory(category);

console.log(`Sorted Drinks (Category: ${category}):`);

sortedDrinks.forEach(drink => console.log(drink.name));}}

**Decorator** permite unui utilizator să adauge o nouă funcționalitate unui obiect existent fără a-i modifica structura. Acest tip de model de design face parte din modelul structural, deoarece acest model acționează ca un înveliș pentru clasa existentă.

Acest model creează o clasă de decorator care înglobează clasa originală și oferă funcționalitate suplimentară, păstrând semnătura metodelor de clasă intactă.

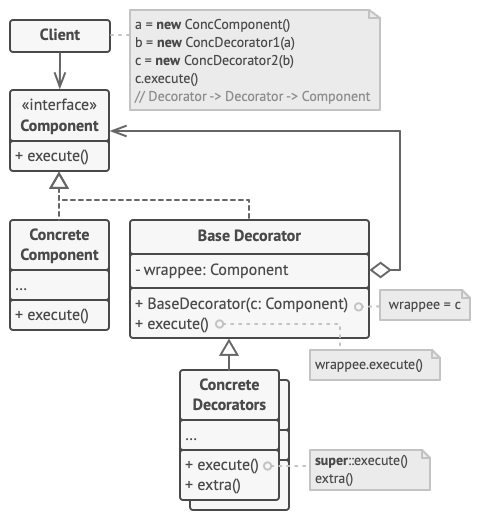


Figura 1.5 Șablon Decorator

Sablonul Decorator a fost implementat în clasa DrinkWithPrice pentru a adăuga comportamente suplimentare la obiectele de tip Drink. Implementarea acestui sablon implică următoarele aspecte:

Clasa DrinkWithPrice extinde clasa de bază Drink și conține o referință către obiectul Drink inițial.

Clasa DrinkWithPrice adaugă o nouă proprietate price care reprezintă prețul băuturii.

Metodele din clasa DrinkWithPrice redefinesc metodele existente din clasa Drink pentru a furniza și prețul împreună cu celelalte informații despre băutură.

Clasa DrinkWithPrice permite extinderea și adăugarea ulterioară a altor funcționalități la obiectele Drink, fără a modifica codul existent al clasei Drink sau a crea subclase separate pentru fiecare combinație de băutură cu preț.

Implementarea sablonului Decorator permite adăugarea de caracteristici suplimentare (în acest caz, prețul) la obiectele de bază (în acest caz, băuturile) fără a modifica structura lor de bază sau a crea clase separate pentru fiecare combinație posibilă. Acesta oferă flexibilitate și extensibilitate în gestionarea băuturilor, permițând adăugarea și eliminarea dinamică a funcționalităților suplimentare.

// Decorator Pattern

class DrinkWithPrice {

constructor(drink, price) {

this.drink = drink;

this.price = price;

}

getName() {

return this.drink.name;

}

getCategory() {

return this.drink.category;

}

getPrice() {

return this.price;

}

}

## **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 am fost folosit un șablon de comportament: Iterator.

**Iterator** este un model de design comportamental care permite parcurgerea elementelor unei colecții sau a unei structuri de date fără a expune detaliile interne ale acesteia. Prin intermediul iteratorului, putem accesa în mod secvențial și eficient elementele unei colecții, fără a fi nevoie să cunoaștem structura internă a colecției sau să depindem de implementarea specifică.

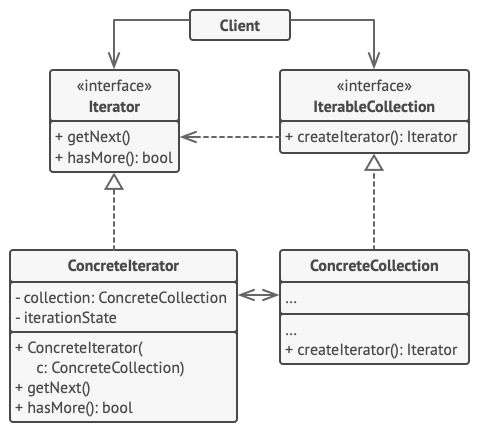


Figura 1.6 Șablonul Iterator

// Iterator Pattern

class DrinkIterator {

constructor(drinks) {

this.drinks = drinks;

this.index = 0;

}

hasNext() {

return this.index < this.drinks.length;

}

next() {

if (this.hasNext()) {

const drink = this.drinks[this.index];

this.index++;

return drink;

}

return null;

}

}

Sablonul Iterator a fost implementat în clasa DrinkIterator pentru a permite parcurgerea într-un mod secvențial a unei colecții de obiecte Drink. Implementarea acestui sablon implică următoarele aspecte: Clasa DrinkIterator conține o referință la colecția de obiecte drinks pe care se dorește să o parcurgem.

Clasa DrinkIterator utilizează o variabilă index pentru a urmări poziția curentă în colecție.

Metoda hasNext() verifică dacă mai există elemente disponibile în colecție prin compararea valorii variabilei index cu lungimea colecției.

Metoda next() returnează următorul element din colecție și avansează variabila index pentru a pregăti următoarea valoare.

Clasa DrinkIterator oferă o abstracție simplă și unificată pentru parcurgerea elementelor colecției de obiecte Drink, ascunzând detalii despre structura și implementarea colecției în sine.

Implementarea sablonului Iterator permite parcurgerea unei colecții de obiecte într-un mod secvențial, fără a dezvălui detalii interne despre cum sunt stocate și organizate aceste obiecte. Acesta oferă o interfață simplă și consistentă pentru a itera prin elementele colecției și permite gestionarea eficientă a iterației prin intermediul metodelor hasNext() și next().

# **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

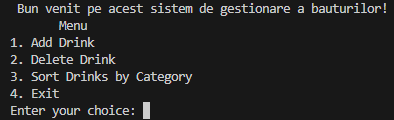


Figura 2.1 Meniul aplicației

În acest menu putem adauga băuturi, sa stergem unele bauturi și să le sortăm. În figura 2.2 este reprezentat cum are loc adăugarea unei băuturi. Pentru a face această operație este nevoie să dăm nume bauturii și categoria din care face parte. Prototype Pattern: Clasa Drink implementează Prototype Pattern și servește ca prototip pentru crearea de obiecte de tipul Drink. Aceasta are o metodă clone() care permite crearea de clone ale obiectului Drink.

Builder Pattern: Clasa DrinkBuilder implementează Builder Pattern și oferă o modalitate convenabilă de a construi obiecte de tipul Drink. Aceasta permite setarea treptată a proprietăților name și category ale unui obiect Drink și, la final, construirea efectivă a obiectului prin apelul metodei build()

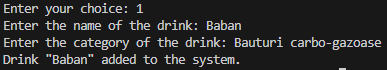


Figura 2.2 Adaugarea unei băuturi

Pentru a șterge băuturile, în cadrul aplicației prezentate, se utilizează modelul de design "Facade". Clasa DrinkFacade acționează ca o interfață simplificată către sistemul de băuturi (DrinkSystem) și ascunde complexitatea operațiunii de ștergere a unei băuturi în spatele unei singure metode deleteDrink(). Aceasta oferă o abstracție simplă și ușor de utilizat pentru utilizatorii aplicației.

În metoda deleteDrink(name) din clasa DrinkFacade, se face apel la metoda corespunzătoare deleteDrink(drinkName) din clasa DrinkSystem, care se ocupă efectiv de ștergerea băuturii. Astfel, utilizatorul nu trebuie să interacționeze direct cu clasa DrinkSystem și să cunoască detaliile de implementare specifice. Prin intermediul clasei DrinkFacade, operația de ștergere a băuturii devine mai ușor de utilizat și mai puțin expusă la complexitatea internă a sistemului.

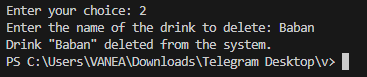


Figura 2.3 Ștergerea băuturilor

Pentru a realiza sortarea băuturilor, în cadrul aplicației prezentate, se utilizează modelul de design "Iterator" în combinație cu "Facade".

Clasa DrinkFacade acționează ca o interfață simplificată către sistemul de băuturi (DrinkSystem). Metoda sortDrinksByCategory(category) din clasa DrinkFacade se ocupă de inițierea operației de sortare a băuturilor după categorie.

În interiorul metodei sortDrinksByCategory(category), se face apel la metoda sortByCategory(category) din clasa DrinkSystem, care returnează o listă de băuturi sortate după categoria specificată. Apoi, lista sortată este afișată în consolă prin intermediul unei bucle forEach.

Modelul de design "Iterator" este implicat în implementarea clasei DrinkIterator, care este utilizată în interiorul clasei DrinkSystem. Aceasta permite parcurgerea secvențială a listei de băuturi pentru a le sorta și a returna rezultatul sortării.

Astfel, prin intermediul modelului de design "Facade" și a combinației cu "Iterator", utilizatorul poate sorta băuturile în funcție de categoria specificată fără a cunoaște detaliile interne de implementare ale claselor implicate în sistemul de băuturi.



Figura 2.4 Sortarea băuturilor după categorie

# **Concluzii:**

În concluzie, aplicația dezvoltată în cadrul proiectului de curs "Tehnici și Mecanisme de Proiectare Software" demonstrează utilizarea eficientă a mai multor modele de design în contextul sistemului de gestionare a băuturilor.

Prin implementarea modelului "Singleton" în clasa DrinkSystem, asigurăm faptul că există o singură instanță a sistemului de băuturi în întreaga aplicație, garantând astfel coerența și consistența datelor.

Modelul "Builder" este utilizat prin intermediul clasei DrinkBuilder, permițând construirea ușoară și flexibilă a obiectelor de tip Drink, oferind posibilitatea de a seta diferite atribute ale băuturilor într-un mod incremental și modular.

Prin implementarea modelului "Prototype" în clasa Drink, putem clona băuturi existente, evitând astfel crearea repetitivă de obiecte și economisind resurse.

Modelul "Facade" este folosit prin intermediul clasei DrinkFacade, oferind o interfață simplificată și unificată pentru operațiile de adăugare, ștergere și sortare a băuturilor. Astfel, se ascunde complexitatea sistemului de gestionare a băuturilor și se facilitează interacțiunea cu acesta.

Prin utilizarea modelului "Decorator" în clasa DrinkWithPrice, putem adăuga informații suplimentare despre prețul băuturilor, fără a modifica structura de bază a acestora.

Modelul "Iterator" este implicat în implementarea clasei DrinkIterator, permițând parcurgerea secvențială a listei de băuturi și oferind acces la elementele acesteia într-un mod eficient.

În ansamblu, prin aplicarea acestor modele de design, aplicația devine mai modulară, flexibilă și ușor de extins în viitor. Aceste modele ajută la separarea responsabilităților, crearea unor interfețe clare și reducerea cuplării dintre componentele aplicației.

Astfel, proiectul de curs "Tehnici și Mecanisme de Proiectare Software" a oferit oportunitatea de a învăța și aplica diverse modele de design într-un context real, demonstrând beneficiile și utilitatea acestora în dezvoltarea software-ului. 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 07.06.20203 disponibil: <https://refactoring.guru/design-patterns/creational-patterns>
    2. Șabloane Structurale , Citat 07.06.20203 disponibil: <https://refactoring.guru/design-patterns/structural-patterns>
    3. Șabloane de Comportament , Citat 07.06.20203 disponibil: <https://refactoring.guru/design-patterns/behavioral-patterns>

# 