Lucrare de laborator Chiriac Dan TI-172

la disciplina TMPS Tema – Coffee Shop

Design Pattern-urile folosite:

Creational Pattern:

* Factory Method
* Prototype

Structural Pattern:

* Decorator Pattern
* Facade Pattern

Behavioral Pattern:

* Chain of Responsibility
* Iterator Pattern

Ca temă principală am ales un Coffee Shop, care are ca idee principală nu numai confecționarea băururilor de cafea pentru public larg, dar și posibilitatea de a procura cafea în cantități mari, începând de la un kilogram. Coffee Shop-ul dat este primul în felul său, căci importează cafea la direct din țările exotice sau care au o formă de pregătire mai specifică. Fiecare pattern are un rol anume în întregul sistem al cafenelei.

**Creational Pattern:**

Sunt preocupate de modul de creare a obiectelor. Aceste modele de proiectare sunt utilizate atunci când trebuie luată o decizie în momentul instantării unei clase (adică crearea unui obiect al unei clase).

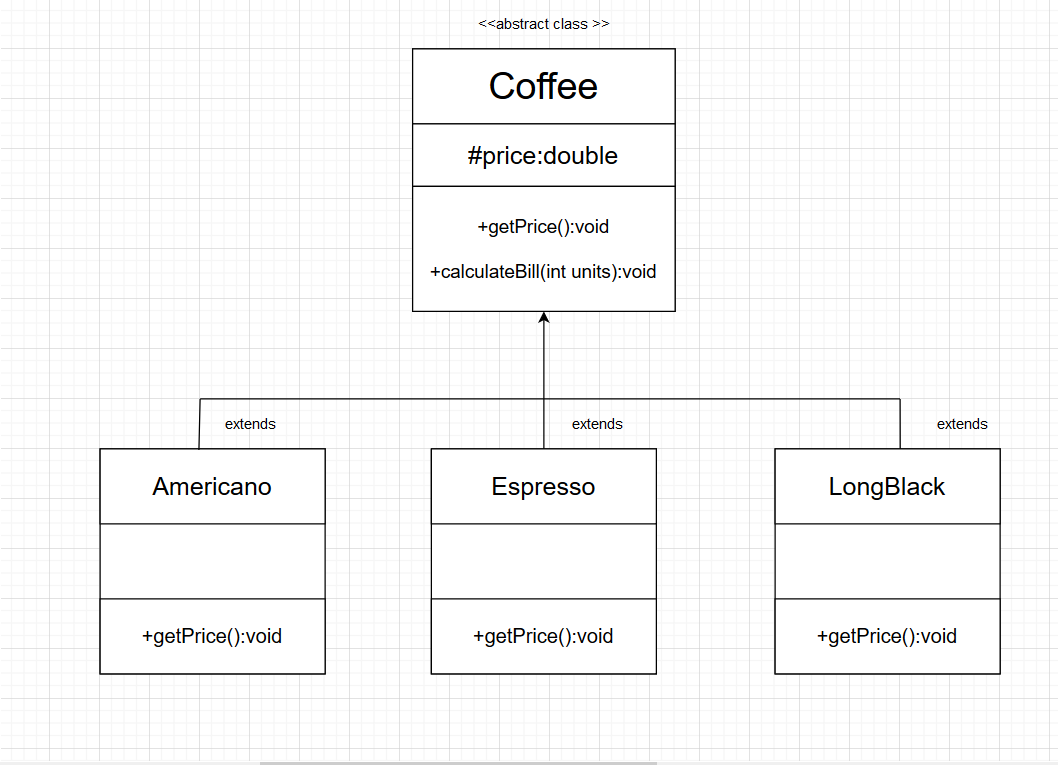
**Factory Method** - definește o interfață sau o clasă abstractă pentru crearea unui obiect, lăsând subclasele să decidă ce clasă să se instanteze. Cu alte cuvinte, subclasele sunt responsabile pentru crearea instanței clasei.

**Avantajele:**

* Permite sub-claselor să aleagă tipul de obiecte de creat.
* Promovează cuplajul prin eliminarea nevoii de a lega clasele specifice aplicației în cod. Aceasta înseamnă că codul interacționează exclusiv cu interfața rezultată sau cu clasa abstractă, astfel încât va funcționa cu orice clase care implementează acea interfață sau care extinde acea clasă abstractă.

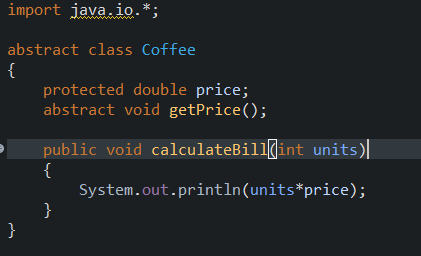
**Utilizarea:**

* Când o clasă nu știe ce sub-clase vor fi necesare pentru a crea.
* Când o clasă dorește ca sub-clasele sale să specifice obiectele care vor fi create.
* Când clasele părinte aleg crearea de obiecte în sub-clasele sale.

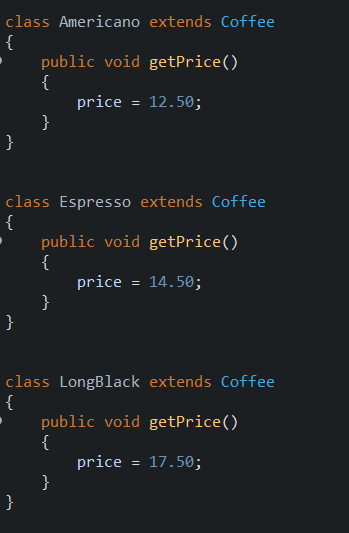


Problema in cause deține alegerea clientului unul din cele 3 tipuri de cafea populare. Prețul este prestabilit de cafenea. Cumparătorul alege ce tip de cafea dorește și evident cantitatea, după care îi va fi prestabilit prețul cecului.

Creăm un fișier denumit Coffee.java. Inițiem o clasă abstractă pentru determinarea prețului și calcularea cecului după cumparea produsului.



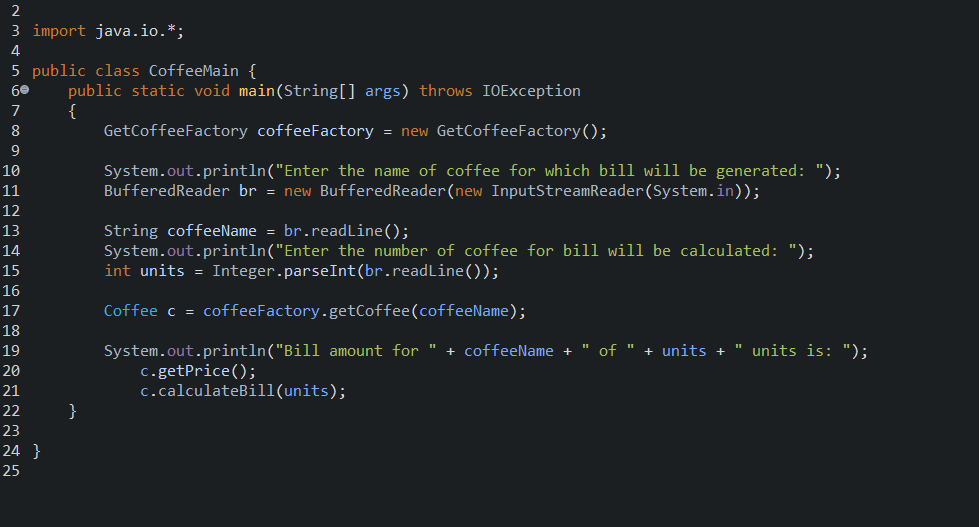
Într-un fișier aparte, facem 3 clase cu diferite tipuri de marfă, indicând prețul acestora. Folosim metoda “extends” la clasa abstractă Coffee.



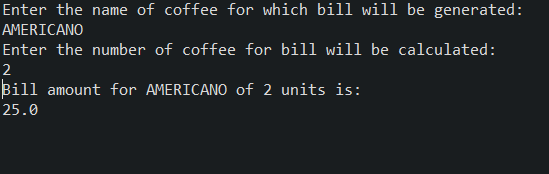
Inițiem fișierul CoffeeFactory.java, ce returnează una din cele 3 alegeri a cafelei.



La final, creăm fișierul CoffeeMain.java, ce reprezintă afișarea și implementarea a Factory Method



OutPut:



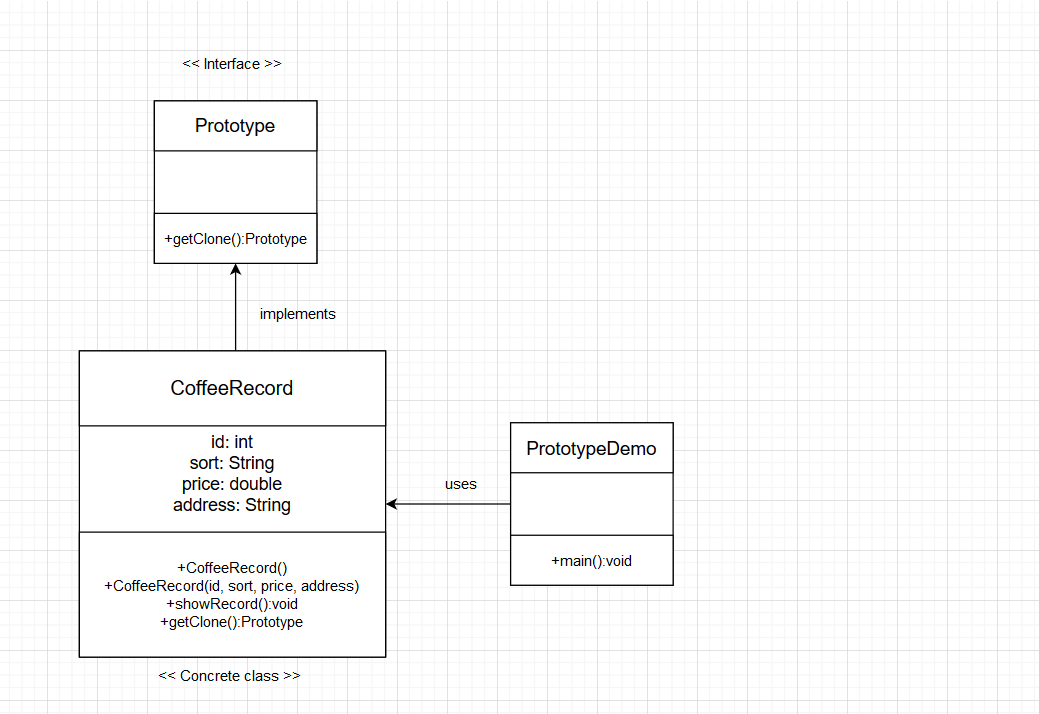
**Prototype -** spune că clonarea unui obiect existent în loc să creeze unul nou și poate fi, de asemenea, personalizată conform cerințelor. Acest model trebuie să fie respectat, dacă costul creării unui obiect nou este costisitor și necesită resurse.

**Avantaje:**

* Reduce nevoia de sub-clasificare.
* Ascunde complexități ale creării de obiecte.
* Clienții pot obține obiecte noi fără să știe ce tip de obiect va fi.
* Permite adăugarea sau eliminarea obiectelor în timp de execuție.

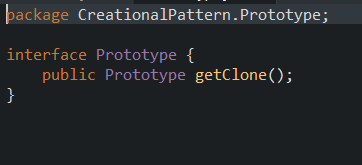
**Utilizarea:**

* Când clasele sunt inițiate la runtime.
* Când costul creării unui obiect este costisitor sau complicat.
* Când doriți să păstrați numărul de clase într-o aplicație la minim.
* Când aplicația client trebuie să nu cunoască crearea și reprezentarea obiectelor.

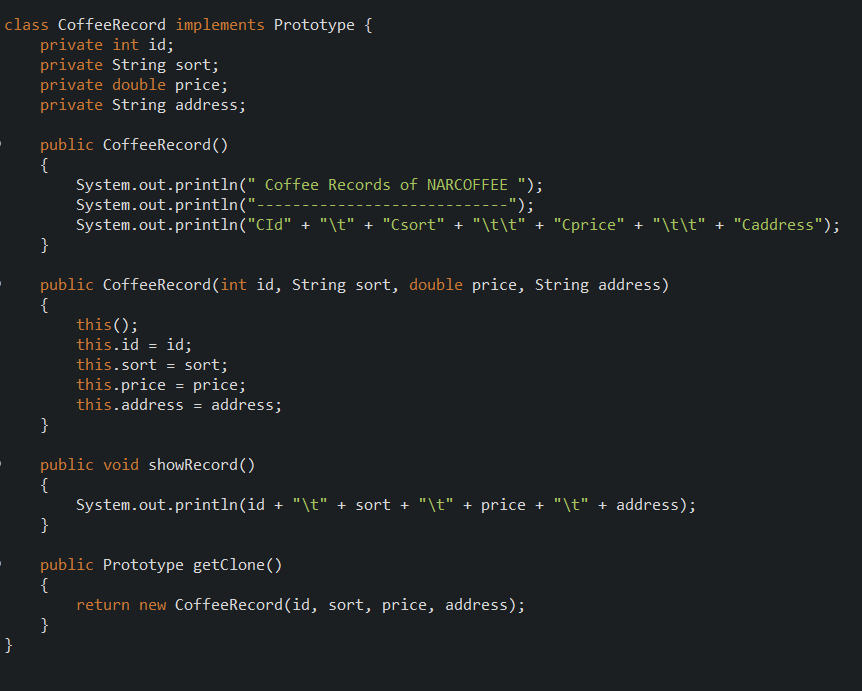


Tipul dat ajută la adăugarea cafelei importate in baza de date. Codul implică adaugarea dupa id, sort, prețul și țara de unde a fost importată cafeaua. Completarea bazei de date în așa formă, reduce timpul și ajută la ducerea de cunoștință a clientului despre informația produselor noastre. Ceea ce garantează transparența unor date publice.

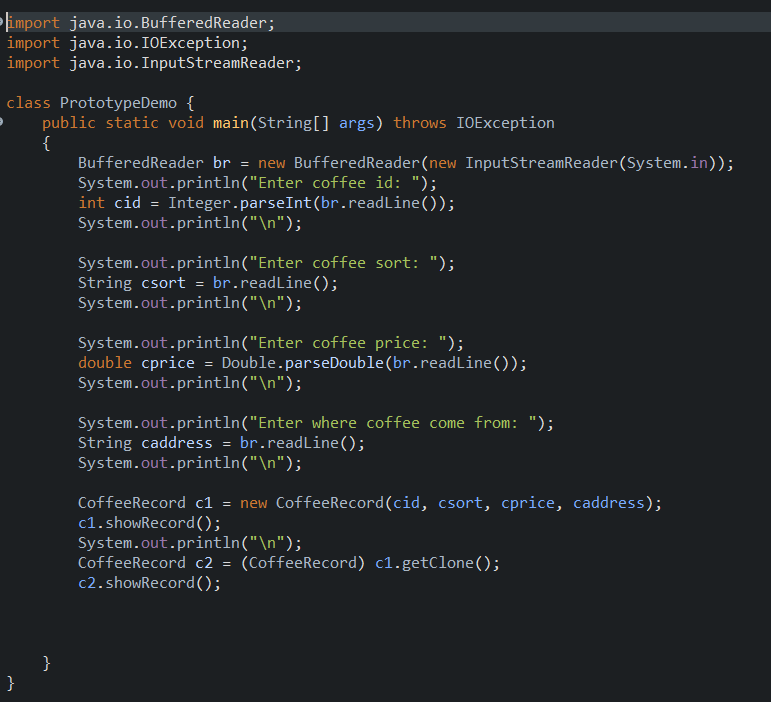
Creăm un fișier Prototype.java



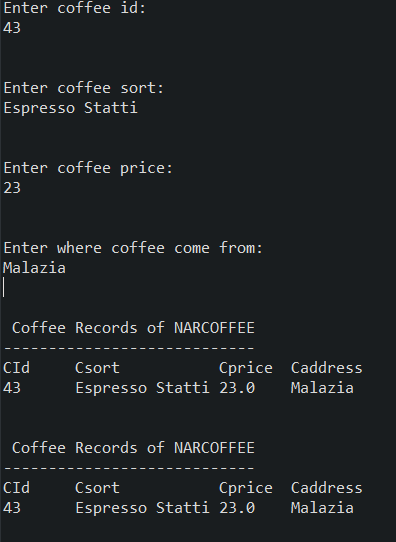
Creăm fișierul CoffeeRecord.java ce implementează clasa Prototype.



Fișierul PrototypeDemo.java va folosi la direct clasa CoffeeRecord



OutPut:



**Structural Pattern**

Sunt preocupate de modul în care clasele și obiectele pot fi compuse, pentru a forma structuri mai mari. Ele simplifică structura prin identificarea relațiilor. Aceste modele se concentrează asupra modului în care clasele se moștenesc una de la alta și cum sunt compuse din alte clase.

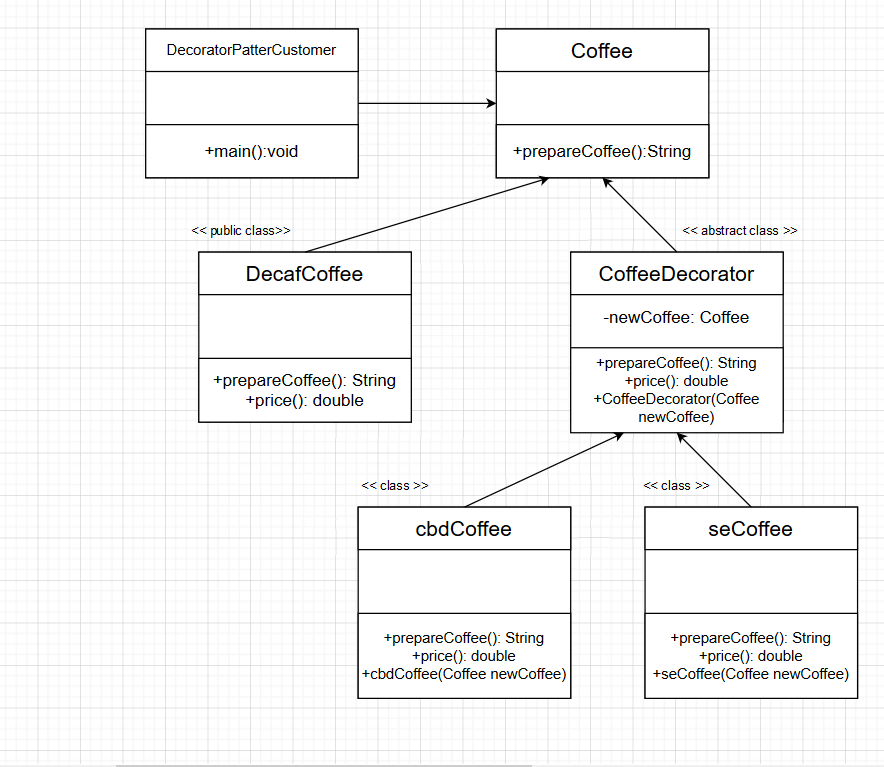
**Decorator Pattern -** atașează dinamic o responsabilitate suplimentară flexibilă unui obiect. Cu alte cuvinte, el folosește compoziția în loc de moștenire pentru a extinde funcționalitatea unui obiect în timpul executării. Decorator Pattern este cunoscut și sub denumirea de Wrapper.

**Avantaje:**

* Oferă flexibilitate mai mare decât moștenirea statică.
* Îmbunătățește extensibilitatea obiectului, deoarece modificările sunt făcute prin codarea claselor noi.
* Simplifică codare permițând de dezvolt o serie de funcționalități din clase direcționate în loc de codat tot comportamentul în obiect.

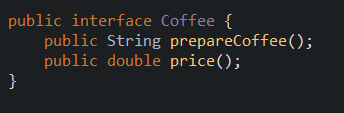
**Utilizarea:**

* Când adăugăm în mod transparent și dinamic responsabilități obiectelor, fără a afecta alte obiecte.
* Când adăugăm responsabilități la un obiect pe care poate doriți să îl schimbați în viitor.
* Extinderea funcționalității prin sub-clasificare nu mai este practică.

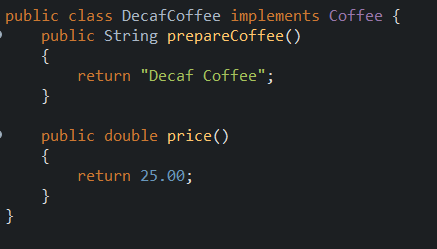


Cafeneaua are update la meniu. Clientul are alegere dintre tipurile de cafea noi apărute: cu și fără cofeină. În departamentul cu cofeină am implementat 2 exemple de cafea: cafea cu CBD și cafea unică de la barista (sau cafea specială).

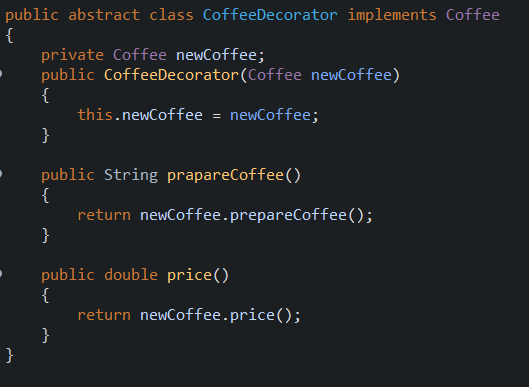
Creăm o interfață pentru tipul de cafea preparată și prețui acesteia.



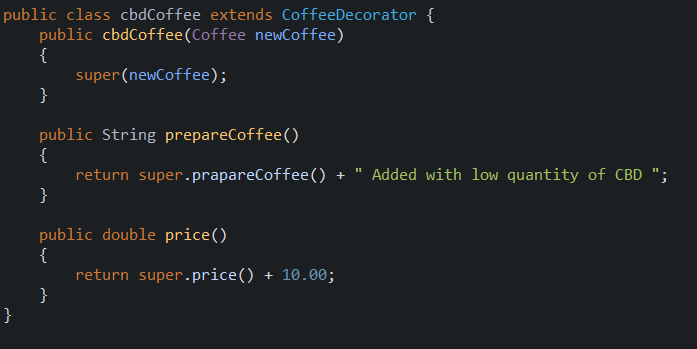
Facem clasa pentru cafea fără cofeină, plasând denumirea cafelei și prețul acestuia. Cu implementarea interfeței “Coffee”.



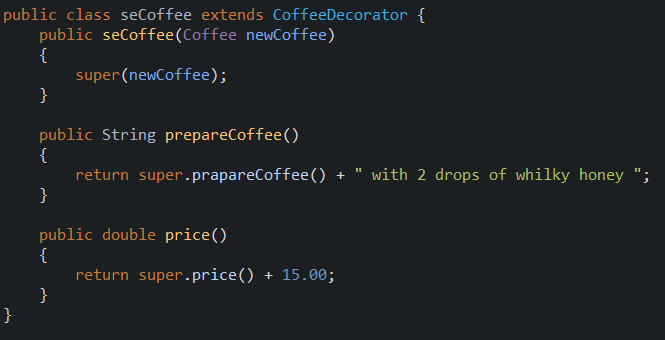
Implementăm clasa CoffeeDecorator ce implică descripția cafelei noi și prețul lor.



Creăm clasa pentru cafea cu CBD, extinsă cu CoffeeDecorator, cu descripția acesteia și prețul lui.

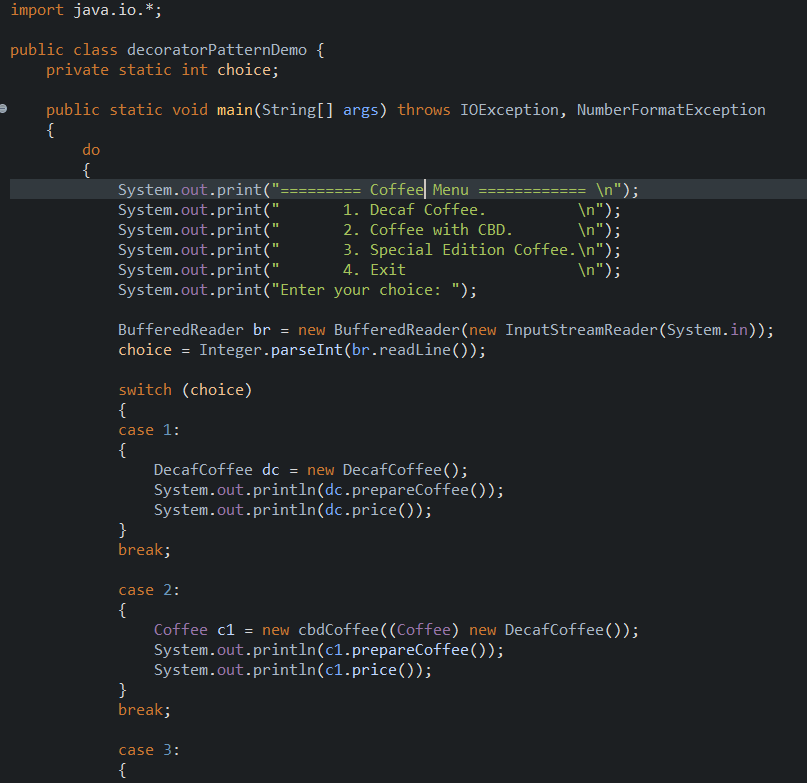


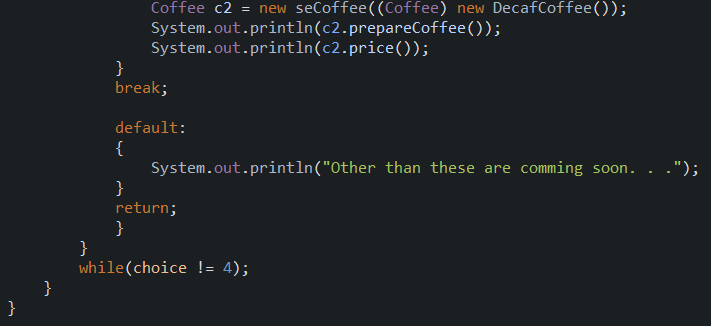
Creăm clasa pentru cafea specială, extinsă cu CoffeeDecorator, cu descripția acesteia și prețul lui.



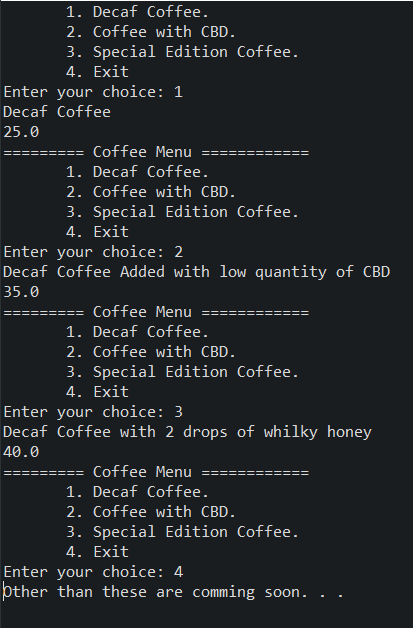
Creăm un fișier main pentru implementarea demo a metodei. Clasa include un meniu in consolă

pentru alegere a clientului





OutPut:



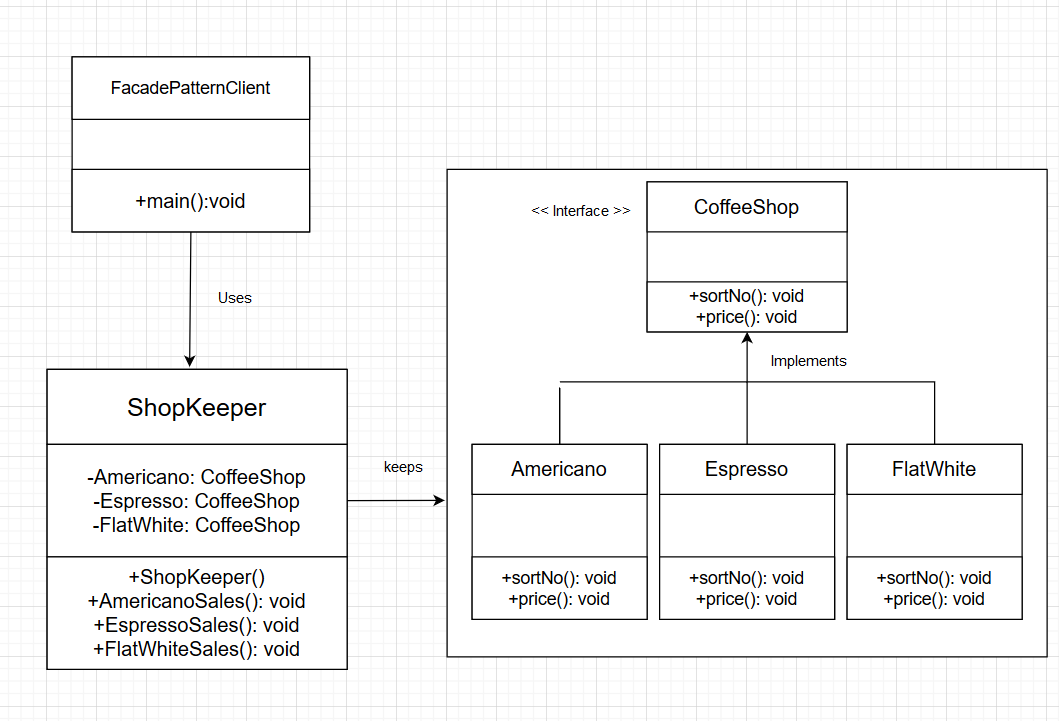
**Facade Pattern -** oferă o interfață unificată și simplificată unui set de interfețe dintr-un subsistem, prin urmare ascunde complexitățile subsistemului de client. Cu alte cuvinte, descrie o interfață de nivel superior care face mai ușor de utilizat subsistemul.

**Avantaje:**

* Protejează clienții de complexitatea componentelor subsistemului.
* Promovează cuplarea liberă între subsisteme și clienții săi.

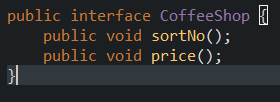
**Utilizarea:**

* Când doriți să furnizați o interfață simplă unui sub-sistem complex.
* Când există mai multe dependențe între clienți și clasele de implementare ale unei abstractizări.

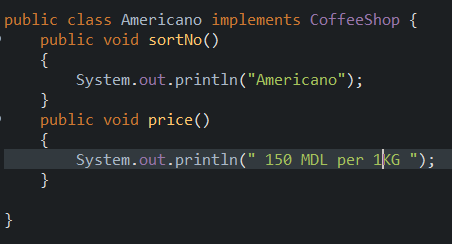


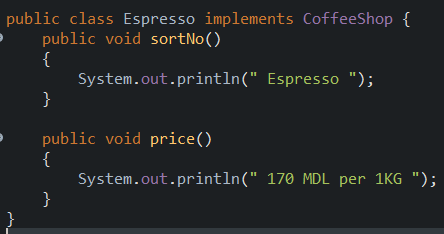
Cafeneua oferă posibilitatea cumpărării cafelei în cantități mari. Clientul poate liber să cumpere cafea începănd de la un kg. Depinde de ce tip de cafea va comanda. În exemplu am introdus 3 tipuri de cafea.

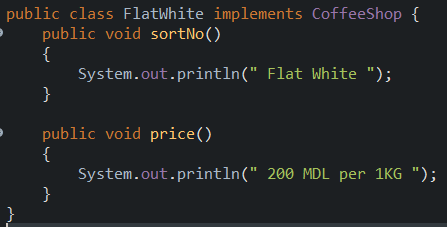
Implementăm o interfață pentru sort și preț



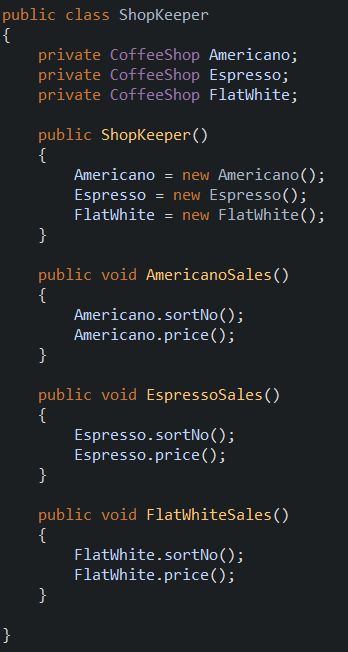
Creăm clase pentru fiecare sort, scriind denumirea și prețul lor.

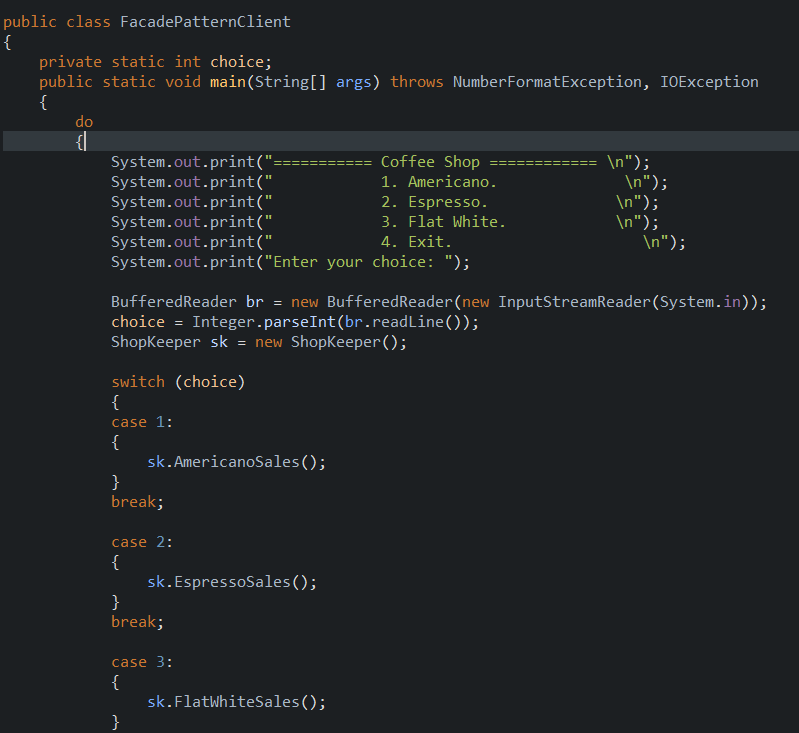


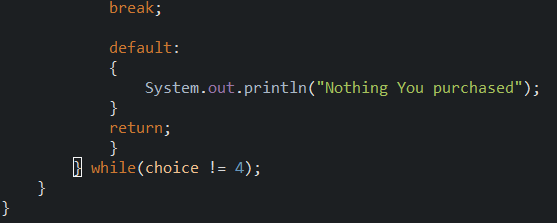




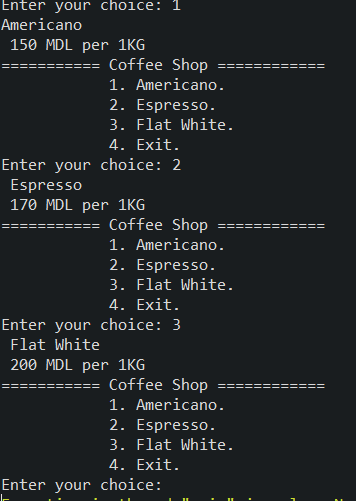
Clasa “ShopKeeper” întreprinde toate tipurile de cafea



Afișăm rezultatul într-un meniu de consolă 



Output:



**Behavioral Pattern**

Sunt preocupate de interacțiunea și responsabilitatea obiectelor. În aceste modele de design, interacțiunea dintre obiecte ar trebui să fie astfel încât să poată vorbi cu ușurință între ele și să fie totuși ușor cuplate. Aceasta înseamnă că implementarea și clientul ar trebui să fie ușor cuplate pentru a evita codificarea și dependențele.

**Chain of Responsibility –** evită cuplarea expeditorului unei cereri la receptorul său, oferind mai multor obiecte o șansă de a gestiona cererea. Cu alte cuvinte, putem spune că, în mod normal, fiecare receptor conține referința unui alt receptor. Dacă un obiect nu poate gestiona cererea, atunci aceasta trece la următorul receptor și așa mai departe.

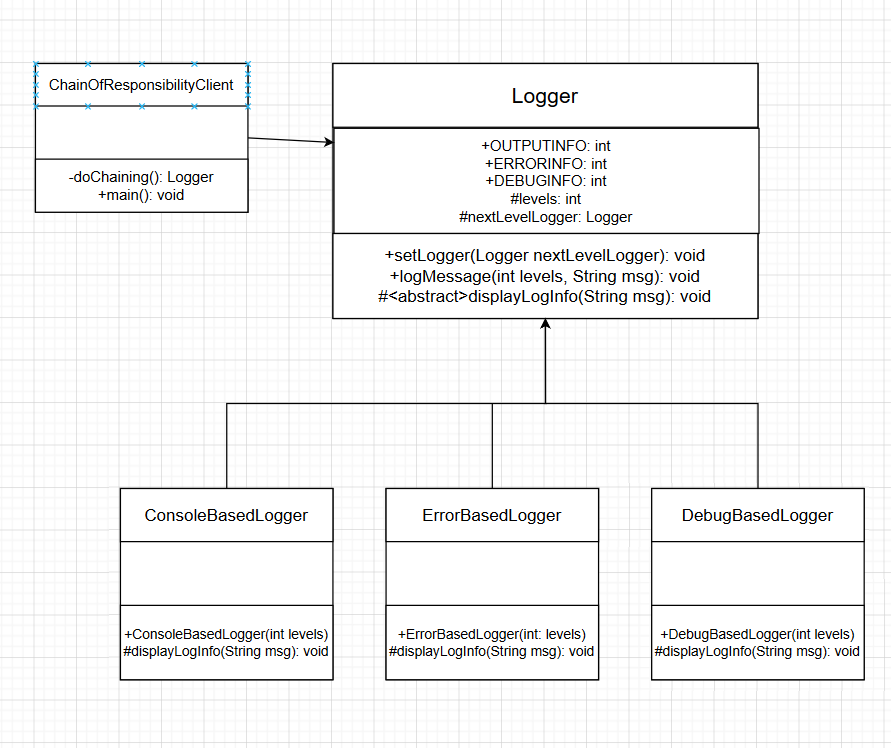
**Avantaje:**

* Acesta reduce cuplajul.
* Adăuga flexibilitate în timp ce atribuie responsabilități obiectelor.
* Permite unui set de clase să acționeze ca unul; evenimentele produse într-o clasă pot fi trimise la alte clase de handler cu ajutorul compoziției.

**Utilizarea:**

* Când mai multe obiecte pot gestiona o solicitare și manipulatorul este necunoscut.
* Când grupul de obiecte care pot gestiona cererea trebuie specificat în mod dinamic.

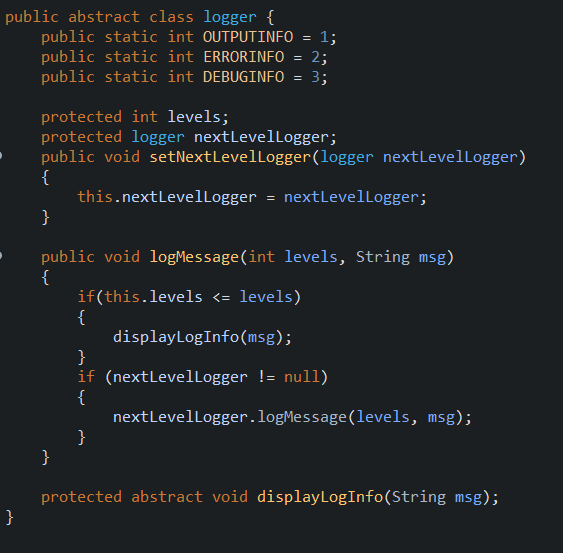
Pattern-ul dat va fi folosit mai mult pentru Sistema interna a backend-ului Coffee-Shop-ului.



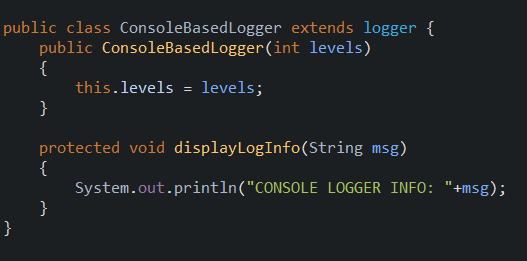
Exprimăm un sistem de creare a comenzii și împărțirea lui în OutPut, Error și Debug. Descriem petrecerea de optimizare a comenzilor. Fiecare comandă trece printr-un ciclu e verificare pentru

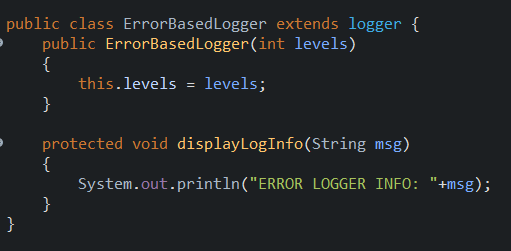
determinarea erorilor sau predicția acestora. Sistemul dat ajută la optimizarea oricărui request care vine spre baza de date a cafenelei. Și, evident, adduce un ajutor pentru comenzile făcute online.

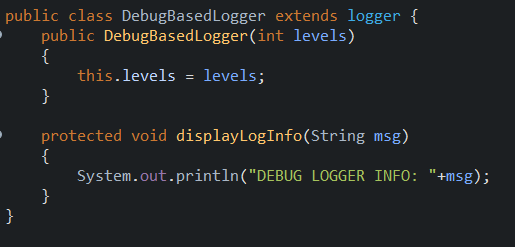
Creăm clasa pentru introducerea nivelelor și mesajelor impuse.



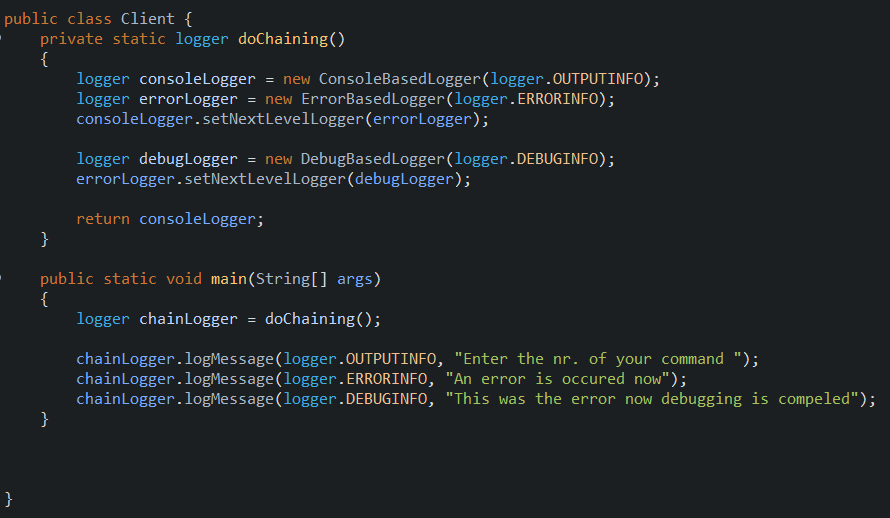
La fiecare nivel implementăm câte o clasă:



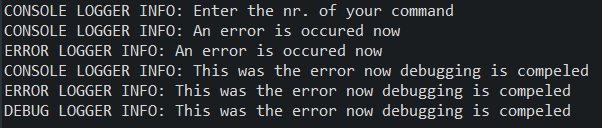




Obținem rezultatul dorit prin implementarea clasei principale de client



OutPut



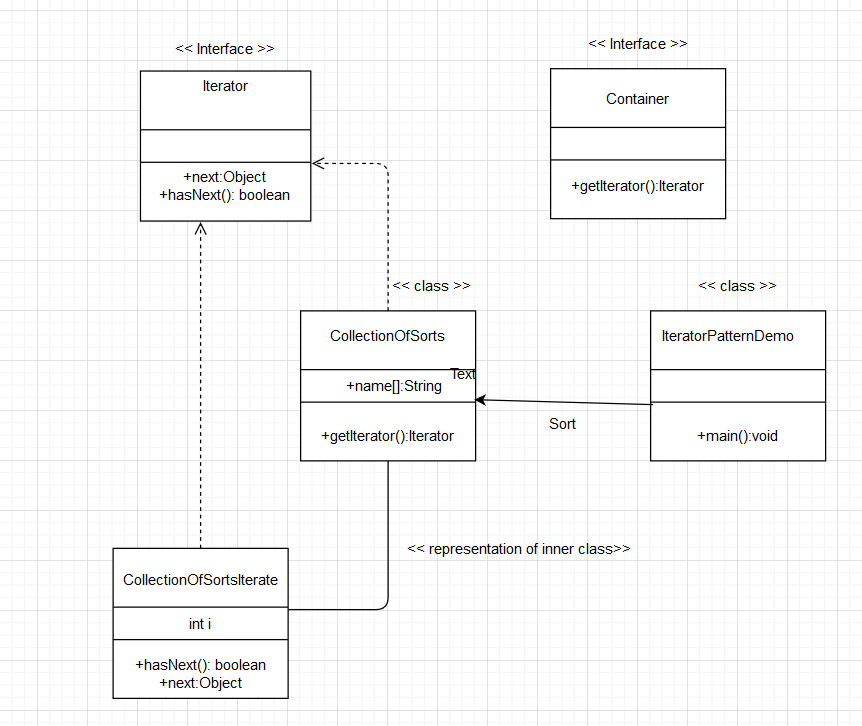
**Iterator Pattern -** este folosit pentru a accesa secvențial elementele unui obiect agregat fără a expune implementarea sa de bază. Iterator Pattern este cunoscut și sub numele de Cursor.

**Avantaje:**

* Acceptă variații în traversarea unei colecții.
* Simplifică interfața la colecție.

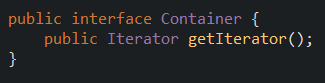
**Utilizarea:**

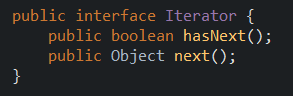
* Când accesăm o colecție de obiecte fără a expune reprezentarea sa internă.
* Când există mai multe traversări de obiecte, trebuie să fie acceptate în colecție.



Prin metoda dată ajutăm sistemul la sortarea produselor și organizarea acestora.

Creăm 2 interfețe pentru iterator și container





Facem clasa a colecției anumitor sorturi



Creăm clasa pentru implementarea metodei

