

**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII**  
**Universitatea Tehnică a Moldovei**  
**Facultatea Calculatoare Informatică și Microelectronică**  
**Departamentul Ingineria Software și Automatică**

## **Proiect de curs**

**Disciplina:** Tehnici și Mecanisme de Proiectare a Programelor

**Tema:** Sistem de Închiriere a Automobilelor  
Car Rental System

**Student:** \_\_\_\_\_ **Vlasitchi Stefan, TI-216**

**Coordonator:** \_\_\_\_\_ **Valeriu Popa, asis. univ.**

**Chișinău 2024**

## Cuprins

Introducere.....	2
1    Analiza domeniului de studiu .....	3
1.1    Scopul, obiectivele și cerințele sistemului.....	4
1.2 Analiza sistemelor deja existente .....	5
2.1 Proiectarea aplicației.....	6
2.2 Descrierea tehnologiilor pentru sistem.....	14
2.3 Descrierea la nivel de cod pe module .....	16
3. Documentarea produsul realizat.....	34
Concluzii .....	41
Bibliografie.....	42

## Introducere

Într-o lume în care mobilitatea și transportul sunt în continuă schimbare, există o creștere a nevoii de soluții eficiente în domeniul serviciilor de închiriere auto. Cu avansarea tehnologiei, metodele tradiționale pentru închirierea de mașini au devenit mai eficiente în satisfacerea nevoilor consumatorilor de astăzi. Acest lucru pune bazele pentru introducerea Sistemului de Închiriere Auto, o inițiativă care își propune să schimbe modul în care oamenii accesează și folosesc automobilele.

Sistemul de Închiriere Auto îmbină tehnologia de vârf cu practicitatea, fiind conceput pentru a oferi o experiență de închiriere atât pentru clienți, cât și pentru furnizorii de servicii. Prin dezvoltarea software-ului și automatizarea, acest proiect își propune să simplifice procesul de închiriere auto eliminând documentele și timpul îndelungat de așteptare asociat în mod obișnuit cu companiile de închiriere.

În esență, Sistemul de Închiriere Auto se bazează pe concepte precum fiabilitatea, eficiența și accesibilitatea. Cu o interfață prietenoasă pentru utilizator și caracteristici solide în spatele sistemului, acesta oferă utilizatorilor libertatea de a rezerva și de a supraveghea vehiculele pe care le-au închiriat pentru o varietate de scopuri, cum ar fi afacerile sau călătoriile. În plus, prin analiza datelor și utilizarea uneltelor de modelare predictivă, propunerea este de a îmbunătăți practicile de gestionare a flotelor prin optimizarea distribuției resurselor pentru o călătorie eficientă în termeni de costuri pentru toate părțile implicate.

Cu accentul său principal pe practicitate, Sistemul de Închiriere Auto contribuie la o gamă mai largă de inovații și avansuri în industria serviciilor de transport. Atât timp cât societatea continuă să îmbrățișeze transformarea digitală și economia participativă, acesta va servi ca dovadă a potențialului tehnologiei de a redefini industriile și modelele de afaceri stabilite. Astfel, Sistemul de Închiriere Auto servește și ca un predictor al avansurilor viitoare în domeniul serviciilor de mobilitate, pe lângă faptul că este o soluție funcțională pentru o nevoie urgentă.

În secțiunile ulterioare ale acestui proiect, ne vom adânci în designul, dezvoltarea și implementarea intricate a Sistemului de Închiriere a Vehiculelor Autonome, examinându-i caracteristicile cheie, specificațiile tehnice și influența potențială asupra industriei de închirieri auto. Prin analiza comprehensivă și evaluarea critică, vom clarifica importanța acestui proiect în ceea ce privește modelarea transportului viitor și reimaginarea modului în care accesăm și utilizăm vehiculele închiriate.

## 1 Analiza domeniului de studiu

Sistemul de închiriere auto reprezintă un aspect crucial al industriei transporturilor, permițând atât persoanelor fizice, cât și întreprinderilor să aibă acces la vehicule pentru utilizare pe termen scurt. Acest domeniu de studiu cuprinde o gamă largă de subiecte, inclusiv tehnologie, servicii pentru clienți, operațiuni de afaceri și reglementări legale. Prin analizarea domeniului de studiu pentru sistemele de închiriere auto, emerg câteva insight-uri cheie.

În primul rând, tehnologia joacă un rol critic în sistemul de închiriere auto. Aceasta include platforme de rezervare, sisteme de urmărire a vehiculelor și software de gestionare a relațiilor cu clienții. Aceste avansuri tehnologice permit companiilor de închiriere să-și optimizeze operațiunile, să îmbunătățească experiența clienților și să-și optimizeze gestionarea flotei. În plus, tehnologiile emergente precum aplicațiile mobile și automatele de autoservire își rescriu modul în care clienții interacționează cu serviciile de închiriere, ducând la o mai mare comoditate și eficiență.

În al doilea rând, serviciul pentru clienți este un aspect fundamental al sistemului de închiriere auto. De la momentul în care un client face o rezervare până la returnarea vehiculului, companiile de închiriere trebuie să asigure o experiență fluentă și pozitivă. Acest lucru implică furnizarea unei comunicări clare, a unor procese eficiente de check-in/check-out și soluționarea promptă a oricăror probleme sau îngrijorări. În plus, înțelegerea preferințelor și comportamentului clienților poate ajuta companiile de închiriere să-și adapteze serviciile și să îmbunătățească satisfacția clienților.

În plus, operațiunile de afaceri sunt fundamentale pentru succesul companiilor de închiriere auto. Acestea includ gestionarea flotei, strategiile de prețuri, inițiativele de marketing și parteneriatele cu alte afaceri. Operațiunile de afaceri eficiente sunt esențiale pentru menținerea unui avantaj competitiv pe piață, maximizarea utilizării vehiculelor și asigurarea rentabilității.

În cele din urmă, reglementările legale și conformitatea sunt cruciale în sistemul de închiriere auto. Companiile de închiriere trebuie să respecte legile locale, naționale și internaționale legate de siguranța vehiculelor, asigurare, responsabilitate și licențiere. Rămânerea actualizată cu reglementările și gestionarea proactivă a riscurilor legale sunt esențiale pentru sustenabilitatea pe termen lung a afacerilor de închiriere.

În concluzie, domeniul de studiu pentru sistemele de închiriere auto este complex, cuprinzând tehnologie, servicii pentru clienți, operațiuni de afaceri și reglementări legale. O analiză a acestui domeniu relevă interconexiunea acestor aspecte și impactul lor colectiv asupra succesului companiilor de închiriere auto. Cercetările viitoare în acest domeniu ar trebui să continue să exploreze tehnologiile emergente, preferințele evolutive ale clienților și mediul regulatory dinamic pentru a îmbunătăți în continuare eficiența și sustenabilitatea sistemelor de închiriere auto.

## 1.1 Scopul, obiectivele și cerințele sistemului

Scopul analizării domeniului de studiu pentru sistemele de închirieri auto este de a obține o înțelegere cuprinzătoare a diferitelor aspecte care contribuie la succesul companiilor de închiriere. Această analiză își propune să identifice punctele de vedere cheie și interconectarea tehnologiei, serviciului pentru clienți, operațiunilor de afaceri și reglementărilor legale în industria de închirieri auto. Prin aceasta, acesta încearcă să ofere informații valoroase pentru îmbunătățirea eficienței și durabilității sistemelor de închiriere de mașini.

Obiectivele proiectului includ:(de redactat)

- Să înțeleagă rolul tehnologiei în sistemul de închiriere de mașini, inclusiv platforme de rezervare, sisteme de urmărire a vehiculelor și software-ul de gestionare a relațiilor cu clienții.
- Să recunoască importanța serviciului pentru clienți pentru a oferi o experiență fără probleme și pozitivă clienților de închiriere.
- Să analizeze rolul central al operațiunilor de afaceri, inclusiv managementul flotei, strategiile de stabilire a prețurilor, inițiativele de marketing și parteneriatele cu alte întreprinderi, în succesul companiilor de închirieri auto.
- Să sublinieze importanța reglementărilor legale și a conformității în asigurarea durabilității pe termen lung a întreprinderilor de închirieri auto.
- Explorarea tehnologiilor emergente, a preferințelor în evoluție ale clienților și a mediului dinamic de reglementare pentru a identifica oportunitățile de îmbunătățire a eficienței și durabilității sistemelor de închirieri auto.

Cerințele sistemului(de redactat)

- Analiza în profunzime a rolului tehnologiei în sistemul de închirieri auto, inclusiv impactul platformelor de rezervare, sistemelor de urmărire a vehiculelor și software-ului de gestionare a relațiilor cu clienții asupra operațiunilor și experienței clienților.
- Examinarea celor mai bune practici în serviciul clienților pentru a asigura o experiență fără probleme și pozitivă pentru clienții de închiriere, inclusiv procese eficiente de check-in/check-out și soluționarea promptă a preocupărilor clientului.
- Evaluarea strategiilor eficiente de operare a afacerii, cum ar fi managementul flotei, strategiile de stabilire a prețurilor, inițiativele de marketing și parteneriatele cu alte întreprinderi, pentru a menține un avantaj competitiv și pentru a maximiza profitabilitatea.
- Înțelegerea aprofundată a reglementărilor legale și a cerințelor de conformitate în industria de închirieri auto pentru a asigura respectarea legilor locale, naționale și internaționale referitoare la siguranța vehiculelor, asigurări, răspundere și licențiere.

- Explorarea continuă a tehnologiilor emergente, a preferințelor în evoluție ale clienților și a mediului dinamic de reglementare pentru a identifica oportunitățile de îmbunătățire a eficienței și durabilității sistemelor de închirieri auto.

Prin îndeplinirea acestor cerințe, analiza domeniului de studiu pentru sistemele de închirieri auto va oferi perspective valoroase pentru îmbunătățirea eficienței și durabilității industriei de închiriere auto, beneficiind în cele din urmă companiile de închiriere, clienții și industria de transport în ansamblu.

## **1.2 Analiza sistemelor deja existente**

Atunci când comparați aplicația noastră de închiriere de mașini cu alte sisteme existente pe piață, este important să rețineți că aplicația nu poate fi la fel de sofisticată ca marii jucători din industrie. Cu toate acestea, aplicația noastră oferă încă mai multe caracteristici și avantaje cheie care o fac o opțiune competitivă pentru utilizatori.

În primul rând, aplicația noastră oferă o interfață ușor de utilizat, care este ușor de navigat, făcându-l accesibil pentru toate tipurile de utilizatori. În timp ce unele dintre aplicațiile de închiriere de mașini mai mari pot avea caracteristici mai avansate și un design mai elegant, aplicația noastră se concentrează pe simplitate și ușurință de utilizare, ceea ce poate fi atrăgător pentru utilizatorii care preferă un proces simplu de rezervare.

În plus, aplicația noastră oferă o gamă diversă de opțiuni de mașini și pachete de închiriere, care se potrivesc nevoilor diferitelor clienți. În timp ce aplicațiile mai mari de închiriere de mașini pot avea o rețea mai largă și opțiuni de flotă mai extinse, aplicația noastră se străduiește să ofere soluții personalizate și personalizate pentru utilizatorii noștri, creând o experiență mai specializată.

Pe langa acestea, aplicația noastră acordă prioritate serviciului pentru clienți și suportului, asigurându-se că utilizatorii au acces la asistență și îndrumare pe tot parcursul procesului de închiriere. Acest accent pe suport personalizat ne diferențiază de unele dintre cele mai mari aplicații de închiriere de mașini, care pot avea sisteme de servicii pentru clienți mai automatizate și standardizate.

În concluzie, deși aplicația noastră poate să nu aibă același nivel de sofisticare și resurse ca unii dintre marii jucători din industria de închirieri auto, aceasta oferă totuși avantaje și caracteristici unice care se adresează preferințelor specifice ale utilizatorilor. Prin concentrarea pe simplitate, personalizare și suport personalizat, aplicația noastră poate concura eficient cu sistemele mai mari de pe piață și oferă o alternativă valoroasă pentru utilizatori.

## 2.1 Proiectarea aplicației

În dezvoltarea de software, implementarea sabloanelor de proiectare la nivel de cod reprezintă un pilon esențial pentru crearea unui sistem eficient și ușor de întreținut. Aceste sabloane sunt soluții consacrate, testate în timp, pentru a rezolva problemele recurente din procesul de dezvoltare software.

Realizarea sistemului nostru a început prin structurarea aplicației și implementarea acestor sabloane de proiectare la nivel de cod. În total, am integrat în proiect 9 sabloane de proiectare, fiecare aducând o valoare distinctă și contribuind la coeziunea și flexibilitatea sistemului nostru. Acestea sunt:

1. Singleton
2. Factory
3. Builder
4. Decorator
5. Repository
6. Facade
7. State
8. Strategy
9. Observer

Fiecare dintre aceste sabloane de proiectare va fi examinat mai detaliat în continuarea referatului, evidențiind modul în care au fost implementate și contribuția lor la arhitectura și funcționalitățile sistemului nostru.

Această abordare strategică în implementarea sabloanelor de proiectare reprezintă o parte vitală a procesului nostru de dezvoltare software și a fost fundamentală pentru construirea unui sistem robust și ușor de întreținut.

Sabloanele de proiectare reprezintă un aspect crucial al dezvoltării software, oferind soluții testate și validate pentru problemele comune din domeniu. Printre acestea, Singleton-ul este unul dintre cele mai esențiale și frecvent utilizate sabloane.

**Singleton** este un concept fundamental în proiectarea software-ului, concentrându-se pe crearea unei singure instanțe a unei clase în cadrul unei aplicații. Acesta furnizează un punct global de acces către instanța sa, asigurând că aceasta este utilizată în mod coerent și eficient în întreaga aplicație.

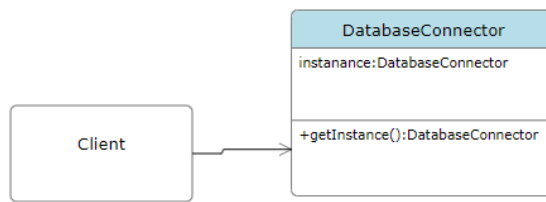


Figura 2.1 Diagrama de clasa a șablonului Singleton

Diagrama de sablon a Singleton-ului, exemplificată în Figura 2.1, ilustrează clar modul în care acesta garantează existența unei singure instanțe a unei clase și oferă un mecanism simplu de acces la aceasta. Singleton-ul este adesea utilizat în situații în care este necesară gestionarea unei resurse unice sau furnizarea unui punct central de acces către anumite servicii sau componente.

Implementarea corectă a Singleton-ului în cadrul proiectelor software aduce numeroase beneficii, inclusiv gestionarea eficientă a resurselor comune și asigurarea consistenței datelor și funcționalităților.

Sablonul de proiectare **Factory** este o abordare fundamentală în dezvoltarea software, oferind un mecanism flexibil pentru crearea obiectelor.

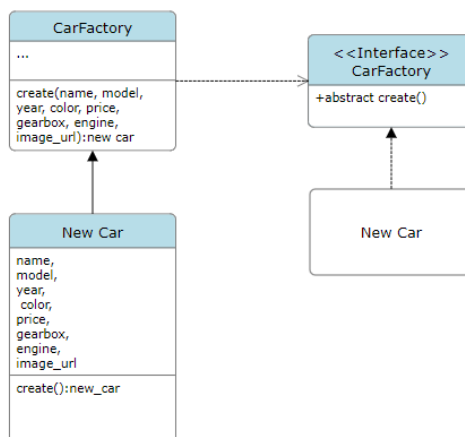


Figura 2.1 Diagrama de clasa a șablonului Factory Method

Figura 2.2 prezintă diagrama de sablon a Factory-ului, ilustrând modul în care acesta gestionează procesul de creare a obiectelor.

Factory-ul separă procesul de creare a obiectelor de modul în care acestea sunt utilizate în cadrul aplicației, permițând o mai mare flexibilitate și extensibilitate. Acesta oferă un punct centralizat de creare a obiectelor, ascunzând detaliile specifice ale implementării și permițând schimbări în modul în care obiectele sunt create fără a afecta restul codului.



Implementarea corectă a sablonului Factory poate duce la o creștere semnificativă a modularității și reutilizabilității codului, permițând dezvoltatorilor să se concentreze pe logica aplicației fără a fi nevoie să se ocupe de detalii tehnice legate de crearea obiectelor.

Sablonul de proiectare Builder este o unealtă esențială în dezvoltarea software, folosită pentru construirea obiectelor complexe.

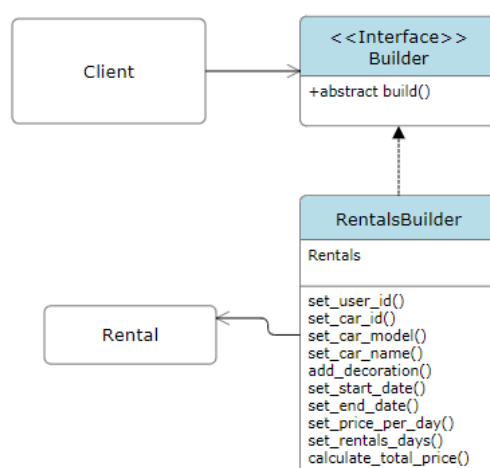


Figura 2.3 Diagrama de clasa a șablonului Builder

Diagrama de șablon a Builder-ului, prezentată în Figura 2.3, ilustrează modul în care acesta permite construirea unui obiect pas cu pas, oferind control și flexibilitate deplină asupra procesului de construcție.

Builder-ul este util în situațiile în care dorim să creăm obiecte complexe care pot avea mai multe configurații sau setări diferite. Acesta ne permite să separăm procesul de construcție a obiectului de reprezentarea acestuia, permițându-ne să construim obiectul pas cu pas în funcție de nevoile noastre specifice.

Prin utilizarea șablonului Builder, putem obține cod mai curat și mai ușor de înțeles, deoarece separăm logica de construcție a obiectului de restul aplicației. Acest lucru ne oferă flexibilitate și extensibilitate în ceea ce privește procesul de construcție a obiectului, permițându-ne să adaptăm și să extindem funcționalitatea fără a afecta codul existent.

Sablonul de proiectare Decorator este o unealtă puternică în dezvoltarea software, utilizată pentru extinderea funcționalității obiectelor existente.

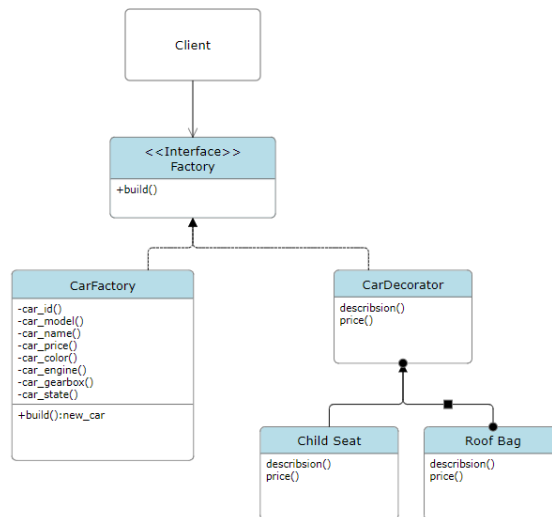


Figura 2.4 Diagrama de clasa a șablonului Decorator

Figura 2.4 prezintă diagrama de sablon a Decorator-ului, ilustrând modul în care acesta permite adăugarea de comportamente suplimentare la un obiect existent, fără modificarea structurii acestuia.

Decorator-ul este util în situațiile în care dorim să adăugăm sau să modificăm comportamentul unui obiect fără a modifica codul existent sau fără a crea subclase noi. Acesta ne oferă flexibilitate și extensibilitate în ceea ce privește modificarea comportamentului obiectelor, permițându-ne să adăugăm funcționalități noi într-un mod modular și reutilizabil.

Prin utilizarea șablonului Decorator, putem obține un cod mai curat și mai ușor de întreținut, deoarece separăm modificările de comportament de restul aplicației. Acest lucru ne permite să extindem funcționalitatea obiectelor într-un mod flexibil și controlat, fără a compromite structura și coerența codului existent.

Sablonul de proiectare Repository este un instrument esențial în dezvoltarea software, utilizat pentru gestionarea accesului la date.

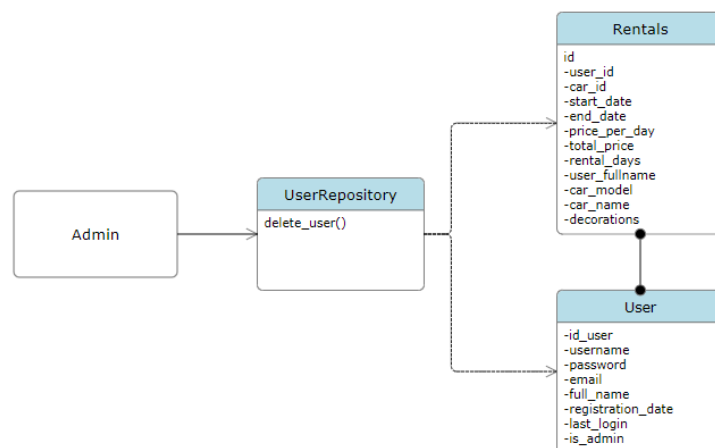


Figura 2.5 Diagrama de clasa a șablonului Repository

Figura 2.5 prezintă diagrama de sablon a Repository-ului, ilustrând modul în care acesta gestionează operațiile de stocare și recuperare a datelor în cadrul aplicației.

Repository-ul este util în situațiile în care dorim să abstractizăm și să encapsulăm accesul la date, permițându-ne să gestionăm operațiile CRUD (Create, Read, Update, Delete) într-un mod centralizat și eficient. Acesta ne oferă un punct central de acces către date, ascunzând detaliile specifice ale implementării și permițându-ne să schimbăm sursa de date fără a afecta restul aplicației.

Prin utilizarea sablonului Repository, putem obține un cod mai modular și mai ușor de întreținut, deoarece separăm logica de acces la date de restul aplicației. Acest lucru ne permite să gestionăm operațiile de stocare și recuperare a datelor într-un mod consistent și scalabil, facilitând dezvoltarea și întreținerea aplicației noastre pe termen lung.

Sablonul de proiectare Facade este o unealtă importantă în dezvoltarea software, folosită pentru a oferi o interfață simplificată către un set complex de funcționalități.

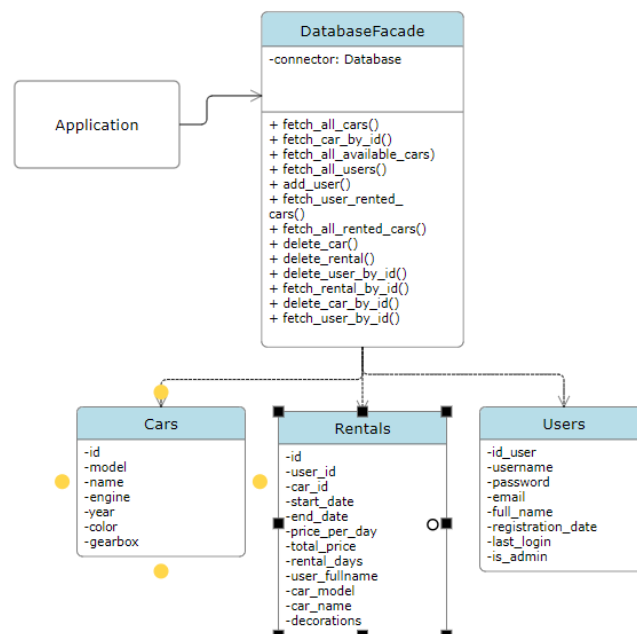


Figura 2.6 Diagrama de clasa a șablonului Facade

Figura 2.6 prezintă diagrama de sablon a Facade-ului, ilustrând modul în care acesta ascunde detaliile complexe ale unui sistem și oferă un punct centralizat de acces către acesta.

Facade-ul este util în situațiile în care avem de-a face cu sisteme complexe sau cu o interfață de programare a aplicațiilor (API) complicată. Acesta ne permite să encapsulăm logica complexă într-o interfață simplificată, permițându-ne să interacționăm cu sistemul fără a fi nevoie să cunoaștem detaliile interne.

Prin utilizarea sablonului Facade, putem obține un cod mai curat și mai ușor de înțeles, deoarece separăm interacțiunea cu sistemul de logica internă a acestuia. Acest lucru ne permite să reducem complexitatea aplicației noastre și să oferim o interfață coezivă și ușor de utilizat pentru clienții noștri.

Sablonul de proiectare State este o unealtă utilă în dezvoltarea software, folosită pentru a permite unui obiect să-și schimbe comportamentul în funcție de starea internă.

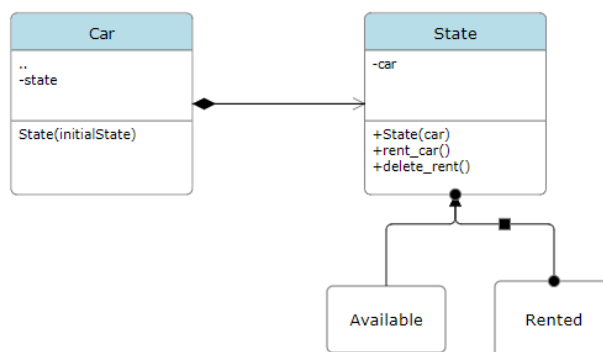


Figura 2.7 Diagrama de clasa a șablonului State

Figura 2.7 prezintă diagrama de sablon a State-ului, ilustrând modul în care acesta permite obiectului să treacă între diferite stări și să-și ajusteze comportamentul în funcție de acestea.

State-ul este util în situațiile în care avem de-a face cu obiecte care pot avea mai multe comportamente sau stări diferite în timpul execuției. Acesta ne permite să encapsulăm comportamentele specifice ale obiectului în clase separate, permițându-ne să gestionăm tranzițiile între stări într-un mod organizat și flexibil.

Prin utilizarea șablonului State, putem obține un cod mai modular și mai ușor de întreținut, deoarece separăm comportamentele specifice ale obiectului în clase separate. Acest lucru ne permite să gestionăm mai eficient comportamentul obiectelor în funcție de starea lor internă, facilitând dezvoltarea și întreținerea aplicației noastre pe termen lung.

Sablonul de proiectare Strategy este o unealtă esențială în dezvoltarea software, folosită pentru a defini o familie de algoritmi și a le încapsula, făcându-i interschimbabili.

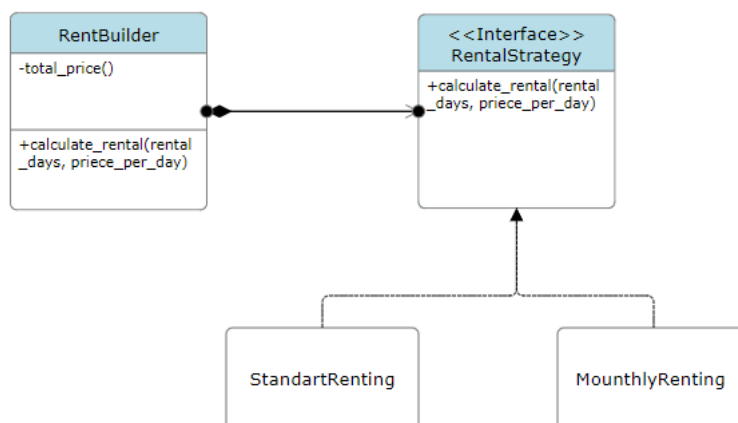


Figura 2.8 Diagrama de clasa a șablonului Strategy

Figura 2.8 prezintă diagrama de sablon a Strategy-ului, ilustrând modul în care acesta permite definirea mai multor strategii de lucru și schimbarea lor în timpul execuției.

Strategy-ul este util atunci când dorim să implementăm diferite variante ale unei funcționalități și să le alegem în mod dinamic în funcție de nevoile noastre. Acesta ne permite să encapsulăm algoritmi diferiți în clase separate, permițându-ne să selectăm și să schimbăm strategiile în mod dinamic fără a afecta codul clientului.

Prin utilizarea sablonului Strategy, putem obține un cod mai flexibil și mai ușor de întreținut, deoarece separăm algoritmi specifici de codul clientului. Acest lucru ne permite să adaptăm comportamentul aplicației noastre în funcție de cerințele specifice ale utilizatorului sau ale mediului de execuție.

Sablonul de proiectare Observer este o unealtă puternică în dezvoltarea software, folosită pentru a stabili o relație de tip unu-la-mulți între obiecte, astfel încât modificările într-un obiect să fie reflectate automat în alte obiecte.

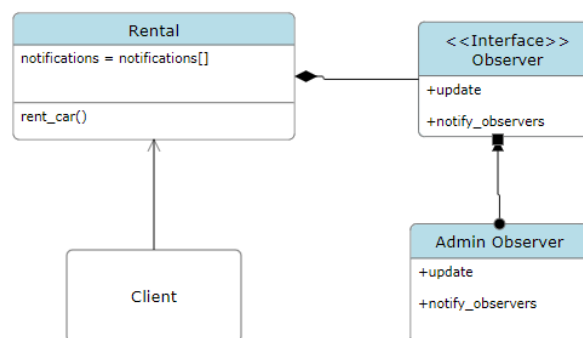


Figura 2.9 Diagrama de clasa a șablonului Observer

Figura 2.9 prezintă diagrama de sablon a Observer-ului, ilustrând modul în care acesta permite comunicarea între observabil și observatori.

Observer-ul este util atunci când dorim să implementăm un sistem în care modificările într-un obiect trebuie să fie propagate automat către alte obiecte interesate. Acesta ne permite să definim o structură de tip publisher-subscriber, în care obiectul observabil trimite notificări către toți observatorii săi atunci când apar modificări.

Prin utilizarea sablonului Observer, putem obține un cod mai modular și mai ușor de întreținut, deoarece separăm logica observării de logica observabilului. Acest lucru ne permite să gestionăm mai eficient comunicarea între obiecte și să reducem cuplajul între componentele aplicației noastre.

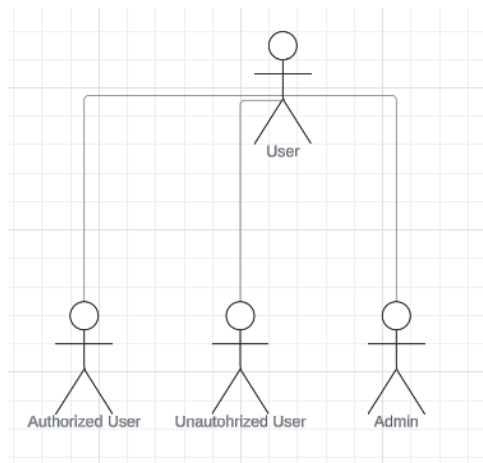


Figura 2.10 Tipurile de utilizatori

În aplicația proiectată, există trei tipuri de utilizatori reprezentați în Figura 2.10, fiecare cu propriile roluri și privilegii :

1. **Utilizatorul Neautentificat:** Utilizatorul neautentificat este un vizitator al aplicației care nu s-a autentificat încă în sistem. Cu toate acestea, acest utilizator poate accesa funcționalități limitate, cum ar fi crearea unui cont nou și vizualizarea mașinilor disponibile în sistem. Crearea unui cont nou îi permite să devină un utilizator autentificat și să beneficieze de funcționalitățile suplimentare ale aplicației.
2. **Utilizatorul Autentificat:** Utilizatorul autentificat este un utilizator care s-a conectat în sistem și are un cont activ. Acesta poate vizualiza mașinile disponibile în sistem și poate efectua acțiuni precum închirierea mașinilor disponibile. De asemenea, utilizatorul autentificat poate gestiona propriile închirieri, având opțiunea de a șterge închirierile încheiate. Beneficiază de funcționalități personalizate, cum ar fi vizualizarea istoricului închirierilor și gestionarea profilului personal.
3. **Adminul:** Adminul este utilizatorul cu cel mai înalt nivel de privilegii în sistem. Este responsabil pentru gestionarea întregii aplicații și are acces complet la toate funcționalitățile și datele disponibile. Adminul poate gestiona conturile utilizatorilor, inclusiv crearea, ștergerea și actualizarea acestora. De asemenea, poate monitoriza activitatea întregului sistem și poate interveni în cazul unor probleme sau incidente.

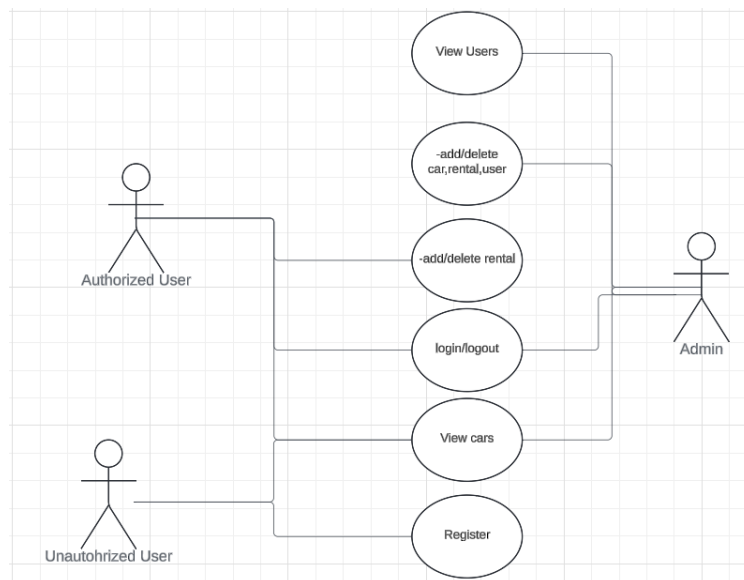


Figura 2.11 Diagrama Acțiunilor Utilizatorilor pe Platformă

Aceste roluri și privilegii sunt esențiale pentru buna funcționare a aplicației noastre și pentru a oferi o experiență optimă utilizatorilor figura 2.11. Ele asigură că fiecare utilizator are acces la funcționalitățile relevante și că adminul poate gestiona și menține sistemul în mod eficient.

## 2.2 Descrierea tehnologiilor pentru sistem

Pentru dezvoltarea aplicației noastre, am utilizat o serie de tehnologii, librării și framework-uri moderne pentru a ne asigura că obținem o soluție robustă și scalabilă. Printre acestea se numără:



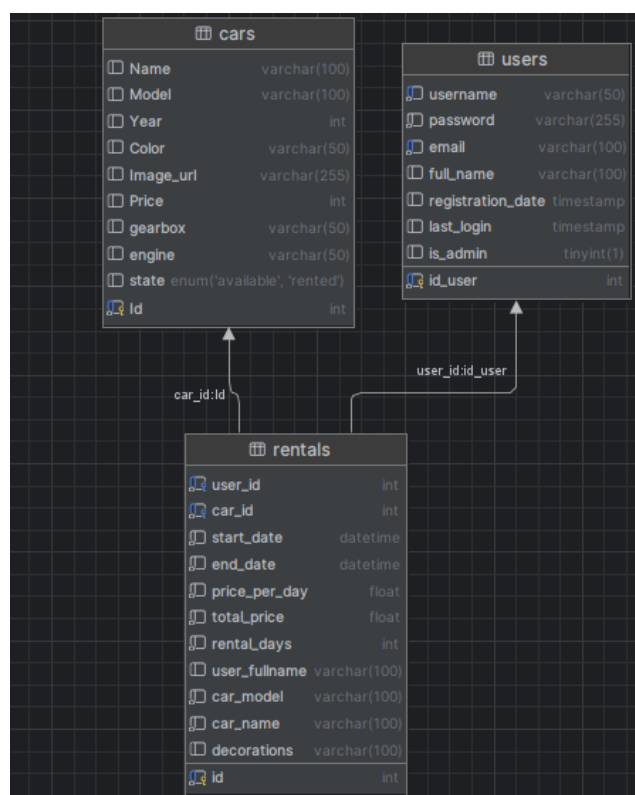
Figura 2.2.1 Logotipul Python

Limbajul de programare principal folosit în dezvoltarea întregului proiect. Pythonul este cunoscut pentru simplitatea sa și pentru capacitățile sale de dezvoltare rapidă, fiind o alegere populară pentru dezvoltarea aplicațiilor web.



**Figura 2.2.2 Logotipul Flask**

Am folosit Flask pentru a construi interfața web a aplicației noastre. Flask este un framework web ușor de utilizat și flexibil, care oferă un set de instrumente puternice pentru dezvoltarea rapidă a aplicațiilor web în Python.



**Figura 2.2.3 Logotipul Flask**

Am integrat SQLAlchemy pentru a gestiona interacțiunea cu baza noastră de date relațională. SQLAlchemy este un ORM puternic și flexibil care ne permite să lucrăm cu baze de date relaționale într-un mod orientat obiect, facilitând manipularea și interogarea datelor. Figura 2.2. oferă o privire de ansamblu asupra modului în care SQLAlchemy ne-a ajutat să gestionăm interacțiunile între tabelele din baza de date creată.

Werkzeug este o bibliotecă Python de bază utilizată în mod implicit de Flask pentru manipularea cererilor și a răspunsurilor HTTP. Această bibliotecă oferă funcționalități esențiale pentru gestionarea



rutelor, a sesiunilor și a multor altor aspecte legate de dezvoltarea aplicațiilor web. Werkzeug este un element de bază al framework-ului Flask și este folosit pentru a asigura funcționarea corectă a aplicației noastre web.

`concurrent.futures` este o bibliotecă Python integrată, utilizată pentru a executa operații concurente și asincrone în Python. Am folosit această bibliotecă pentru a gestiona sarcinile concurente și pentru a asigura performanța optimă a aplicației noastre în timpul execuției unor operații intensive.

Aceste tehnologii și librării ne-au permis să construim o aplicație web robustă și eficientă, care să ofere o experiență plăcută utilizatorilor noștri. Integrarea acestora în cadrul proiectului nostru a fost esențială pentru dezvoltarea și implementarea cu succes a aplicației noastre.

## 2.3 Descrierea la nivel de cod pe module

```
rental_builder.py

from abc import ABC, abstractmethod

from db_models import Rentals

class Builder(ABC):
    @abstractmethod
    def build(self):
        pass

class RentalsBuilder(Builder):
    def __init__(self, rentals=None):
        self.rentals = rentals if rentals else Rentals()
        self.decorations = []

    def set_user_id(self, user_id):
        self.rentals.user_id = user_id
        return self

    def set_car_id(self, car_id):
        self.rentals.car_id = car_id
        return self

    def set_car_model(self, car_model):
        self.rentals.car_model = car_model
        return self

    def set_car_name(self, car_name):
        self.rentals.car_name = car_name
        return self

    def add_decoration(self, decoration):
        self.decorations.append(decoration)
        return self

    def add_decorations(self, decorations):
        self.decorations.extend(decorations)
        return self

    def set_start_date(self, start_date):
```

```

        self.rentals.start_date = start_date
        return self

    def set_end_date(self, end_date):
        self.rentals.end_date = end_date
        return self

    def set_price_per_day(self, price_per_day):
        self.rentals.price_per_day = price_per_day
        return self

    def set_rental_days(self, rental_days):
        self.rentals.rental_days = rental_days
        return self

    def calculate_total_price(self):
        if self.rentals.price_per_day is None or self.rentals.rental_days is None:
            raise ValueError("Price per day and rental days must be set before calculating
total price")
        self.rentals.total_price = self.rentals.price_per_day * self.rentals.rental_days
        return self

    def build(self):
        if not all([self.rentals.user_id, self.rentals.car_id, self.rentals.start_date,
self.rentals.end_date,
                    self.rentals.price_per_day, self.rentals.total_price,
self.rentals.rental_days]):
            raise ValueError("All parameters must be set before building Rentals")

        # Join selected decorators into a string
        decorations = ', '.join(self.decorations)

        # Create a new Rentals object with all the set properties
        rentals = Rentals(
            user_id=self.rentals.user_id,
            user_fullname=self.rentals.user_fullname,
            car_id=self.rentals.car_id,
            car_model=self.rentals.car_model,
            car_name=self.rentals.car_name,
            decorations=decorations,
            start_date=self.rentals.start_date,
            end_date=self.rentals.end_date,
            price_per_day=self.rentals.price_per_day,
            total_price=self.rentals.total_price,
            rental_days=self.rentals.rental_days
        )

        return rentals

@app.route("/rent/<int:car_id>", methods=['GET', 'POST'])
def rent_car(car_id):
    if 'user_id' not in session:
        flash("You need to log in to rent a car", "error")
        return redirect(url_for('login'))

    # Get the user ID from the session
    user_id = session['user_id']

    user = db.session.get(User, user_id)
    user_fullname = user.full_name

    # Get the car object from the database
    car = db.session.get(Cars, car_id)

    # Check if the car exists
    if not car:
        flash("Car not found", "error")
        return redirect(url_for('main'))

    if request.method == 'POST':
        # Extract form data

```

```

start_date = request.form.get('start_date')
end_date = request.form.get('end_date')
selected_decorators = request.form.getlist('decorators') # Assuming decorators are
selected as checkboxes

# Retrieve car model and name from the database
car_model = car.model
car_name = car.name
price_per_day = car.price

rental_days = (datetime.strptime(end_date, '%Y-%m-%d') - datetime.strptime(start_date,
'%Y-%m-%d')).days

# Determine pricing strategy based on the number of days rented
if rental_days > 30:
    pricing_strategy = MonthlyPricingStrategy()
else:
    pricing_strategy = StandardPricingStrategy()

# Calculate total price using selected pricing strategy
total_price = pricing_strategy.calculate_rental(rental_days, price_per_day)

# Apply decorator costs
decorated_car = car # Initialize decorated car with the base car
for decorator_name in selected_decorators:
    if decorator_name == 'ChildSeat':
        decorated_car = ChildSeat(decorated_car)
    elif decorator_name == 'GPS':
        decorated_car = GPS(decorated_car)
    elif decorator_name == 'RoofBag':
        decorated_car = RoofBag(decorated_car)

# Create a RentalsBuilder instance and set its properties
builder = RentalsBuilder()
rental = builder.set_user_id(user_id) \
    .set_car_id(car_id) \
    .set_car_model(car_model) \
    .set_car_name(car_name) \
    .add_decorations(selected_decorators) \
    .set_start_date(start_date) \
    .set_end_date(end_date) \
    .set_price_per_day(price_per_day) \
    .set_rental_days(rental_days) \
    .calculate_total_price(total_price) \
    .build()

# Add the decorated car details to the rental object
rental.car_model = decorated_car.model
rental.car_name = decorated_car.name
rental.total_price += decorated_car.price # Update total price with decorator costs

# Save the rental object to the database
db.session.add(rental)
db.session.commit()

flash("Car rented successfully", "success")
return redirect(url_for('main'))

return render_template('rent_car.html', car=car)

```

Acest fragment de cod utilizează șablonul de proiectare Builder pentru a construi și personaliza obiecte de tip Rentals.

- Builder Pattern: Este un șablon de proiectare creativ care permite construirea unui obiect complex pas cu pas. Este util atunci când un obiect trebuie să fie creat în etape sau când configurarea obiectului este complexă.

1. Se definește o clasă de bază abstractă `Builder`, care conține o metodă abstractă `build()`, responsabilă de construirea obiectului.
2. Se implementează o clasă concretă `RentalsBuilder`, care moștenește `Builder`. Această clasă conține metode pentru setarea diferitelor proprietăți ale obiectului `Rentals`.
3. Metodele din `RentalsBuilder` permit setarea de date precum id-ul utilizatorului, id-ul mașinii, modelul mașinii, numele mașinii, data de început și de sfârșit a închirierii, prețul pe zi, numărul de zile închiriate și altele.
4. Metoda `calculate\_total\_price()` calculează prețul total al închirierii pe baza prețului pe zi și a numărului de zile închiriate.
5. Metoda `build()` finalizează construcția obiectului `Rentals` și returnează obiectul complet.

Acest șablon de proiectare este util pentru a gestiona crearea și personalizarea obiectelor `Rentals` într-un mod clar și structurat. Permite adăugarea și modificarea proprietăților obiectului într-o manieră flexibilă și modulară, fără a polua codul cu o mulțime de constructori și metode set. De asemenea, oferă un mod eficient de a valida și finaliza obiectul creat înainte de utilizare.

Acest fragment de cod construiește și salvează un obiect de tip `Rentals` în baza de date în funcție de datele primite dintr-un formular HTML. Utilizează, de asemenea, decoratori pentru a adăuga accesorii la mașină și calculează prețul total al închirierii în funcție de durata închirierii și de prețul mașinii.

```
Car_factory.py
# models/factories.py
from abc import abstractmethod, ABC

from db_models import Cars

class Factory(ABC):
    @abstractmethod
    def create(self, name, model, year, color, price, gearbox, engine, image_url):
        pass

class CarFactory(Factory):
    def create(self, name, model, year, color, price, gearbox, engine, image_url):
        if not all([name, model, year, color, price, gearbox, engine, image_url]):
            raise ValueError("Incomplete car information. Make sure all attributes are set.")
        new_car = Cars(
            name=name,
            model=model,
            year=year,
            color=color,
```

```

        price=price,
        gearbox=gearbox,
        engine=engine,
        image_url=image_url
    )
    # Create and return a new Cars object
    return new_car

app.py
@app.route('/submit-new-car', methods=['POST'])
def submit_new_car():
    if request.method == 'POST':
        # Extract form data
        name = request.form.get('name')
        model = request.form.get('model')
        year = request.form.get('year')
        color = request.form.get('color')
        price = request.form.get('price')
        gearbox = request.form.get('gearbox')
        engine = request.form.get('engine')

        # Handle the file upload
        if 'image_url' in request.files:
            photo = request.files['image_url']
            if photo.filename != '':
                filename = secure_filename(photo.filename)
                photo_path = os.path.join('photos', filename)
                photo.save(photo_path)
                image_url = f"photos/{filename}"
            else:
                image_url = None
        else:
            image_url = None

        # Create the car using the factory pattern
        new_car = car_factory.create(name, model, year, color, price, gearbox, engine,
image_url)

        # Add the new car to the database
        db.session.add(new_car)
        db.session.commit()

        # subject.notify_observers({"car_model": model})

        # Redirect to the main page or any other appropriate page
        return redirect('/')

    return "Method not allowed", 405

```

Acest fragment de cod utilizează șablonul de proiectare Factory pentru a crea obiecte de tip Car într-un mod modular și extensibil.

- Factory Pattern: Este un șablon de proiectare creational care abstrage procesul de creare a obiectelor. Acesta permite crearea de obiecte de tipul unei clase de bază prin intermediul unei clase Factory, care poate avea implementări diferite în funcție de nevoile aplicației.

1. Se definește o clasă abstractă `Factory`, care conține o metodă abstractă `create()`, responsabilă pentru crearea obiectului.

2. Se implementează o clasă concretă `CarFactory`, care moștenește `Factory`. Această clasă conține o implementare specifică pentru crearea obiectelor de tip Car.
3. Metoda `create()` din `CarFactory` primește argumente referitoare la atributele mașinii și creează un obiect de tipul `Cars`.
4. Obiectul `Cars` este creat utilizând datele primite și este returnat în funcție de specificațiile clasei `Factory`.

Acest șablon de proiectare este util pentru a gestiona crearea și inițializarea obiectelor de tip Car într-un mod organizat și modular. Separarea procesului de creare a obiectelor de implementarea lor permite o extensibilitate și o flexibilitate mai mare în codul aplicației. De asemenea, permite adăugarea ușoară a unor noi tipuri de obiecte sau a unor noi implementări de fabrici pentru a satisface nevoile viitoare ale aplicației.

Fragmentul de cod primește datele dintr-un formular HTML, creează un nou obiect de tip Car folosind Factory Pattern și salvează obiectul în baza de date. Este o modalitate eficientă și organizată de a gestiona crearea și inițializarea obiectelor complexe într-o aplicație web.

```
Db.py
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class MyCustomError(Exception):
    def __init__(self, message="An error occurred"):
        super().__init__(message)

class DatabaseConnector:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.db = db # Initialize db instance
        else:
            raise MyCustomError("DatabaseConnector instance already exists. Use
DatabaseConnector.get_instance() to access the instance.")
        return cls._instance

    @classmethod
    def get_instance(cls):
        if not cls._instance:
            cls._instance = cls()
        return cls._instance

app.py
db_connector = DatabaseConnector.get_instance()
db = db_connector.db
```

Acest fragment de cod implementează șablonul Singleton pentru a asigura că există o singură instanță a conexiunii la baza de date în întreaga aplicație.

- Singleton Pattern: Este un șablon de proiectare creational care garantează că o clasă are doar o singură instanță și oferă un punct global de acces la această instanță.

1. Clasa `DatabaseConnector` conține o variabilă statică privată `\_\_instance` care stochează instanța unică a clasei.

2. Metoda `\_\_new\_\_()` este suprascrisă pentru a controla crearea instanțelor clasei. Dacă nu există încă o instanță, se creează una și se atribuie la variabila `\_\_instance`.

3. Metoda `get\_instance()` este folosită pentru a accesa instanța existentă sau pentru a crea una nouă dacă nu există.

4. Variabila `db` este inițializată cu instanța singleton a conexiunii la baza de date, astfel încât să fie utilizată în întreaga aplicație.

Acest șablon de proiectare este util pentru a asigura că o singură instanță a conexiunii la baza de date este utilizată în întreaga aplicație. Acest lucru poate fi util pentru a evita crearea mai multor conexiuni simultane, care ar putea duce la probleme de performanță sau consistență a datelor în aplicație.

Fragmentul de cod oferă un mod sigur și eficient de a gestiona conexiunea la baza de date în cadrul aplicației Flask, permițând accesul la această conexiune din orice parte a codului.

```
car_decorator.py
from abc import abstractmethod, ABC

from models import Car

class CarDecorator(Car, ABC):
    def __init__(self, car):
        self.car = car

    @abstractmethod
    def description(self):
        pass

    @abstractmethod
    def price(self):
        pass

class ChildSeat(CarDecorator):
    def __init__(self, car):
        super().__init__(car)

    def description(self):
        return f"{super().description()}, Children Seat: Yes"

    def price(self):
        return self.car.price() + 15

class GPS(CarDecorator):
```

```

def __init__(self, car):
    super().__init__(car)

def description(self):
    return f"{super().description()}, GPS: Yes"

def price(self):
    return self.car.price() + 20

class RoofBag(CarDecorator):
    def __init__(self, car):
        super().__init__(car)

    def description(self):
        return f"{super().description()}, Roof Bag: Yes"

    def price(self):
        return self.car.price() + 30

app.py
@app.route("/rent/<int:car_id>", methods=['GET', 'POST'])
def rent_car(car_id):
    if 'user_id' not in session:
        flash("You need to log in to rent a car", "error")
        return redirect(url_for('login'))

    # Get the user ID from the session
    user_id = session['user_id']

    user = db.session.get(User, user_id)
    user_fullname = user.full_name

    # Get the car object from the database
    car = db.session.get(Cars, car_id)

    # Check if the car exists
    if not car:
        flash("Car not found", "error")
        return redirect(url_for('main'))

    if request.method == 'POST':
        # Extract form data
        start_date = request.form.get('start_date')
        end_date = request.form.get('end_date')
        selected_decorators = request.form.getlist('decorators') # Assuming decorators are
selected as checkboxes

        # Retrieve car model and name from the database
        car_model = car.model
        car_name = car.name
        price_per_day = car.price

        rental_days = (datetime.strptime(end_date, '%Y-%m-%d') - datetime.strptime(start_date,
'%Y-%m-%d')).days

        # Determine pricing strategy based on the number of days rented
        if rental_days > 30:
            pricing_strategy = MonthlyPricingStrategy()
        else:
            pricing_strategy = StandardPricingStrategy()

        # Calculate total price using selected pricing strategy
        total_price = pricing_strategy.calculate_rental(rental_days, price_per_day)

        # Apply decorator costs
        decorated_car = car # Initialize decorated car with the base car
        for decorator_name in selected_decorators:
            if decorator_name == 'ChildSeat':
                decorated_car = ChildSeat(decorated_car)

```



```

        elif decorator_name == 'GPS':
            decorated_car = GPS(decorated_car)
        elif decorator_name == 'RoofBag':
            decorated_car = RoofBag(decorated_car)

# Create a RentalsBuilder instance and set its properties
builder = RentalsBuilder()
rental = builder.set_user_id(user_id) \
                .set_car_id(car_id) \
                .set_car_model(car_model) \
                .set_car_name(car_name) \
                .add_decorations(selected_decorators) \
                .set_start_date(start_date) \
                .set_end_date(end_date) \
                .set_price_per_day(price_per_day) \
                .set_rental_days(rental_days) \
                .calculate_total_price(total_price) \
                .build()

# Add the decorated car details to the rental object
rental.car_model = decorated_car.model
rental.car_name = decorated_car.name
rental.total_price += decorated_car.price # Update total price with decorator costs

# Save the rental object to the database
db.session.add(rental)
db.session.commit()

flash("Car rented successfully", "success")
return redirect(url_for('main'))

return render_template('rent_car.html', car=car)

```

Acest fragment de cod implementează șablonul de proiectare Decorator pentru a adăuga funcționalități suplimentare la obiectele de tip Car în funcție de opțiunile selectate de utilizator în timpul închirierii unei mașini.

- Decorator Pattern: Este un șablon de proiectare structural care permite adăugarea de funcționalități suplimentare sau modificarea comportamentului unui obiect existent, fără a afecta alte instanțe ale aceleiași clase.

1. Clasa abstractă `CarDecorator` este definită pentru a servi drept bază pentru decoratori. Ea moștenește clasa `Car` și definește metodele abstracte `description()` și `price()` pe care toți decoratorii trebuie să le implementeze.

2. Decoratorii concreți, cum ar fi `ChildSeat`, `GPS` și `RoofBag`, moștenesc clasa `CarDecorator` și implementează metodele abstracte pentru a adăuga descrieri și prețuri suplimentare în funcție de opțiunile selectate de utilizator.

3. În `app.py`, în funcția `rent\_car()`, atunci când utilizatorul selectează accesorii pentru mașină în timpul închirierii, acestea sunt aplicate la mașină utilizând decoratorii concreți. De exemplu, dacă utilizatorul selectează un scaun pentru copii, obiectul `ChildSeat` este aplicat la mașină, adăugând o descriere și un preț suplimentar.

4. După aplicarea decorării, obiectul mașinii decorate este utilizat pentru a crea obiectul de închiriere și pentru a calcula prețul total al închirierii.

Acest șablon de proiectare este util pentru a extinde funcționalitatea obiectelor de tip Car într-un mod flexibil și modular. Permite adăugarea de opțiuni suplimentare sau accesorii la mașină fără a modifica structura acesteia, ceea ce facilitează gestionarea și extinderea codului în timp.

Fragmentul de cod demonstrează utilizarea șablonului Decorator în cadrul unei aplicații Flask pentru a oferi funcționalități suplimentare utilizatorilor care închiriază mașini, precum adăugarea de scaune pentru copii, GPS-uri sau saci pentru acoperiș.

```
DatabaseFacade.py
from sqlalchemy import select
from sqlalchemy.exc import SQLAlchemyError

from db import DatabaseConnector, db
from db_models import User, Cars, Rentals

class DatabaseFacade:
    def __init__(self):
        self.connector = DatabaseConnector()

    def fetch_all_cars(self):
        try:
            cars_query = select(Cars).filter()
            cars = db.session.execute(cars_query).scalars().all()
            return cars
        except SQLAlchemyError as e:
            print("Error fetching cars:", e)
            return []

    def fetch_car_by_id(self, car_id):
        try:
            car = db.session.query(Cars).get(car_id)
            return car
        except SQLAlchemyError as e:
            print("Error fetching car by ID:", e)
            return None

    def fetch_all_available_cars(self):
        try:
            cars_query = select(Cars).filter(Cars.state == 'available')
            cars = db.session.execute(cars_query).scalars().all()
            return cars
        except SQLAlchemyError as e:
            print("Error fetching available cars:", e)
            return []

    def fetch_all_users(self):
        try:
            return db.session.query(User).all()
        except SQLAlchemyError as e:
            print("Error fetching users:", e)
            return []

    def add_user(self, username, email, password, full_name):
        try:
            user = User(username=username, email=email, password=password,
full_name=full_name)
            db.session.add(user)
            db.session.commit()
            return user
        except SQLAlchemyError as e:
            print("Error adding user:", e)
            return None
```

```

def fetch_user_rented_cars(self, user_id):
    try:
        rented_cars = db.session.query(Rentals).filter_by(user_id=user_id).all()
        return rented_cars
    except SQLAlchemyError as e:
        print("Error fetching rented cars for user:", e)
        return []

def fetch_all_rented_cars(self):
    rented_cars = db.session.query(Rentals).all()
    return rented_cars

def delete_car(self, car_id):
    car = db.session.query(Cars).get(car_id)
    if car:
        db.session.delete(car)
        db.session.commit()

def delete_rental(self, rental_id):
    rental = db.session.query(Rentals).get(rental_id)
    if rental:
        car_id = rental.car_id

        # Update the status of the car to 'available'
        car = db.session.query(Cars).get(car_id)
        if car:
            car.state = 'available'
            db.session.commit()
        else:
            print(f"Car {car_id} not found")

        # Delete the rental after updating the car state
        db.session.delete(rental)
        db.session.commit()

def delete_user_by_id(self, user_id):
    user = db.session.query(User).get(user_id)
    if user:
        db.session.delete(user)
        db.session.commit()

def fetch_rental_by_id(self, rental_id):
    rental = db.session.query(Rentals).get(rental_id)
    return rental

def delete_car_by_id(self, car_id):
    try:
        car = db.session.query(Cars).get(car_id)
        if car:
            db.session.delete(car)
            db.session.commit()
    except SQLAlchemyError as e:
        print("Error deleting car by ID:", e)

def fetch_user_by_id(self, user_id):
    try:
        user = db.session.query(User).get(user_id)
        return user
    except SQLAlchemyError as e:
        print("Error fetching user by ID:", e)
        return None

```

Acest fragment de cod implementează un Facade Pattern pentru a ascunde complexitatea operațiilor cu baza de date și pentru a oferi o interfață simplificată pentru a accesa datele din diferite tabele.

- Facade Pattern: Este un șablon de proiectare structural care oferă o interfață simplificată către un set mai mare și mai complex de clase, module sau framework-uri. Acesta ascunde complexitatea operațiunilor și oferă un punct de acces unic către funcționalități diverse.

1. Clasa `DatabaseFacade` oferă o interfață simplificată pentru a accesa operațiile de bază cu baza de date.

2. Metodele din `DatabaseFacade` abstractizează operațiile de bază, cum ar fi adăugarea, ștergerea și interogarea datelor din diferite tabele.

3. În interiorul fiecărei metode, se utilizează `DatabaseConnector` pentru a obține o instanță a conexiunii la baza de date și `db.session` pentru a efectua operațiile efective cu baza de date.

4. Toate metodele din `DatabaseFacade` gestionează excepțiile `SQLAlchemyError` pentru a asigura o manipulare robustă a erorilor.

Acest șablon de proiectare este util pentru a izola complexitatea operațiilor cu baza de date și pentru a oferi o interfață simplificată către aceste operații. Acest lucru face codul mai ușor de înțeles și de gestionat, deoarece utilizatorii nu trebuie să interacționeze direct cu detalii tehnice ale bazei de date.

Fragmentul de cod oferă o abordare organizată și modulară pentru gestionarea operațiilor cu baza de date în cadrul unei aplicații Flask.

Repository.py

```
from db import db
from db_models import User, Rentals

class UserRepository:
    @staticmethod
    def delete_user(user_id):
        user = db.session.query(User).get(user_id)

        if user:
            # Check if the user has any associated rentals
            rentals = db.session.query(Rentals).filter_by(user_id=user_id).all()

            if rentals:
                # If there are associated rentals, delete them first
                for rental in rentals:
                    db.session.delete(rental)

            # Now delete the user
            db.session.delete(user)
            db.session.commit()
            return True
        else:
            return False
```

app.py

```
@app.route('/delete-user/<int:id_user>', methods=['GET', 'POST'])
def delete_user(id_user):
    if request.method == 'POST':
        # Check if the user is logged in and is an admin
        if 'user_id' not in session:
            flash("You need to log in to perform this action", "error")
            return redirect(url_for('login'))

        # Call the delete_user method from UserRepository
```

```

    if UserRepository.delete_user(id_user):
        flash("User deleted successfully", "success")
    else:
        flash("User not found", "error")

    return redirect(url_for('home'))

# If the request method is not POST, return a method not allowed error
return "Method not allowed", 405

```

Acest fragment de cod implementează un pattern numit Repository Pattern pentru a izola operațiile de acces și modificare a datelor legate de entitatea User într-o clasă separată.

- Repository Pattern: Este un șablon de proiectare care abstractizează logica de acces și manipulare a datelor într-o clasă numită repository. Acesta oferă o interfață simplificată pentru a lucra cu datele, izolând detaliile specifice ale bazei de date sau ale accesului la date.

1. Clasa `UserRepository` conține metode statice care definesc operațiile disponibile pentru entitatea User, în acest caz, doar operația de ștergere a unui utilizator.

2. Metoda `delete\_user(user\_id)` primește ID-ul utilizatorului care trebuie șters și verifică dacă utilizatorul există în baza de date.

3. Dacă utilizatorul există, repository-ul verifică dacă există înregistrări asociate cu utilizatorul în tabela Rentals. Dacă există, aceste înregistrări sunt șterse mai întâi pentru a evita conflicte de integritate referențială.

4. Apoi, utilizatorul este șters din baza de date și modificările sunt comise în sesiunea actuală a bazei de date.

Acest șablon de proiectare este util pentru a separa logic operațiile de acces și modificare a datelor de logica aplicației. Acest lucru face codul mai modular, mai ușor de întreținut și mai ușor de testat, deoarece operațiile legate de bază de date sunt izolate în repository-uri.

Fragmentul de cod oferă o abordare organizată pentru gestionarea ștergerii utilizatorilor în cadrul unei aplicații Flask, cu utilizarea pattern-ului Repository pentru a izola logica specifică bazei de date.

```

Rentalstrategy.py
from abc import ABC, abstractmethod

class RentalStrategy(ABC):
    @abstractmethod
    def calculate_rental(self, rental_days, price_per_day):
        pass

class StandardPricingStrategy(RentalStrategy):
    def calculate_rental(self, rental_days, price_per_day):
        return rental_days * price_per_day

class MonthlyPricingStrategy(RentalStrategy):
    DISCOUNT_RATE = 0.8

    def calculate_rental(self, rental_days, price_per_day):

```

```

        total_price = rental_days * price_per_day
        return total_price * self.DISCOUNT_RATE
app.py
@app.route("/rent/<int:car_id>", methods=['GET', 'POST'])
def rent_car(car_id):
    if 'user_id' not in session:
        flash("You need to log in to rent a car", "error")
        return redirect(url_for('login'))

    # Get the user ID from the session
    user_id = session['user_id']

    user = db.session.get(User, user_id)
    user_fullname = user.full_name

    # Get the car object from the database
    car = db.session.get(Cars, car_id)

    # Check if the car exists
    if not car:
        flash("Car not found", "error")
        return redirect(url_for('main'))

    if request.method == 'POST':
        # Extract form data
        start_date = request.form.get('start_date')
        end_date = request.form.get('end_date')
        selected_decorators = request.form.getlist('decorators') # Assuming decorators are
selected as checkboxes

        # Retrieve car model and name from the database
        car_model = car.model
        car_name = car.name
        price_per_day = car.price

        rental_days = (datetime.strptime(end_date, '%Y-%m-%d') - datetime.strptime(start_date,
'%Y-%m-%d')).days

        # Determine pricing strategy based on the number of days rented
        if rental_days > 30:
            pricing_strategy = MonthlyPricingStrategy()
        else:
            pricing_strategy = StandardPricingStrategy()

        # Calculate total price using selected pricing strategy
        total_price = pricing_strategy.calculate_rental(rental_days, price_per_day)

        # Apply decorator costs
        decorated_car = car # Initialize decorated car with the base car
        for decorator_name in selected_decorators:
            if decorator_name == 'ChildSeat':
                decorated_car = ChildSeat(decorated_car)
            elif decorator_name == 'GPS':
                decorated_car = GPS(decorated_car)
            elif decorator_name == 'RoofBag':
                decorated_car = RoofBag(decorated_car)

        # Create a RentalsBuilder instance and set its properties
        builder = RentalsBuilder()
        rental = builder.set_user_id(user_id) \
            .set_car_id(car_id) \
            .set_car_model(car_model) \
            .set_car_name(car_name) \
            .add_decorations(selected_decorators) \
            .set_start_date(start_date) \
            .set_end_date(end_date) \
            .set_price_per_day(price_per_day) \
            .set_rental_days(rental_days) \
            .calculate_total_price(total_price) \
            .build()

```

```

# Add the decorated car details to the rental object
rental.car_model = decorated_car.model
rental.car_name = decorated_car.name
rental.total_price += decorated_car.price # Update total price with decorator costs

# Save the rental object to the database
db.session.add(rental)
db.session.commit()

flash("Car rented successfully", "success")
return redirect(url_for('main'))

return render_template('rent_car.html', car=car)

```

Acest fragment de cod implementează un sistem de închiriere a mașinilor în cadrul unei aplicații Flask, folosind o strategie de tarification diferită în funcție de numărul de zile pentru care este închiriată mașina.

- RentalStrategy: Este o clasă abstractă care definește metoda `calculate\_rental`, care este implementată de clasele concrete `StandardPricingStrategy` și `MonthlyPricingStrategy`.
- StandardPricingStrategy: Implementează o strategie de tarification standard, în care prețul închirierii este calculat multiplicând numărul de zile cu prețul pe zi.
- MonthlyPricingStrategy: Implementează o strategie de tarification lunară, în care se aplică o reducere de 20% la prețul total al închirierii pentru închirieri de peste 30 de zile.

1. În funcția `rent\_car`, atunci când utilizatorul trimite o cerere de închiriere a unei mașini, se verifică dacă utilizatorul este autentificat în sesiune. Dacă nu este autentificat, este afișat un mesaj de eroare și utilizatorul este redirecționat către pagina de autentificare.

2. Se obține modelul și numele mașinii din baza de date pe baza ID-ului mașinii specificat în URL.

3. Se extrag datele din formular, cum ar fi data de început și data de sfârșit a închirierii, precum și opțiunile suplimentare selectate de utilizator (de ex. scaun pentru copii, GPS).

4. Se calculează numărul de zile de închiriere folosind datele extrase din formular.

5. Se selectează strategia de tarification adecvată pe baza numărului de zile de închiriere: `StandardPricingStrategy` pentru închirieri mai scurte de 30 de zile și `MonthlyPricingStrategy` pentru închirieri mai lungi de 30 de zile.

6. Se calculează prețul total al închirierii folosind strategia de tarification selectată.

7. Se aplică costurile suplimentare (dacă există) pentru opțiunile suplimentare selectate de utilizator, cum ar fi scaune pentru copii sau GPS-uri.

8. Se creează un obiect de închiriere utilizând `RentalsBuilder`, care conține toate detaliile închirierii, inclusiv informațiile despre utilizator, mașină și opțiunile suplimentare.

9. Detaliile mașinii decorate sunt adăugate la obiectul de închiriere.

10. Obiectul de închiriere este salvat în baza de date folosind ``db.session.add()`` și ``db.session.commit()``.

11. Un mesaj de succes este afișat utilizatorului, iar acesta este redirecționat către pagina principală.

Această abordare oferă o modalitate eficientă de gestionare a închirierii mașinilor în cadrul unei aplicații web, utilizând strategii de tarification diferite în funcție de durata închirierii.

```
Observer.py
from abc import abstractmethod

class Observer:
    @abstractmethod
    def update(self, data):
        pass
    @abstractmethod
    def format_notification(self, data):
        pass

class Subject:
    def __init__(self):
        self._observers = []
        self.notifications = []

    def register(self, observer):
        self._observers.append(observer)

    def unregister(self, observer):
        self._observers.remove(observer)

    def notify_observers(self, data=None):
        for observer in self._observers:
            observer.update(data)
            self.notifications.append(observer.format_notification(data))

class UserObserver(Observer):
    def update(self, data):
        # Logic to notify regular users about new cars added
        print(f"New car added: {data['car_model']}")

    def format_notification(self, data):
        return f"New car added: {data['car_model']}"

class AdminObserver:
    def update(self, data):
        # Logic to notify admins about car rentals
        print(f"Car rented: {data['car_model']} by user {data['user_id']}")

    def format_notification(self, data):
        return f"Car rented: {data['car_model']} by user {data['user_id']}"
```

Acest fragment de cod implementează un pattern de proiectare numit Observer Pattern, care este folosit pentru a notifica alte obiecte (observatori) despre modificările sau evenimentele care au loc într-un obiect (subiect) fără ca observatorii să fie conștienți de existența altor observatori.



- Observer Pattern: Este un pattern de proiectare în care un obiect numit "subiect" menține o listă de dependențe numită "observatori" și îi notifică automat pe observatori despre orice modificări de stare, de obicei prin apeluri de metode specifice pe care le implementează observatorii.

1. Clasa abstractă Observer: Definese două metode abstracte: ``update(data)`` și ``format_notification(data)``. Aceste metode sunt implementate de clasele concrete de observatori.

2. Clasa Subject: Reprezintă subiectul care este observat. Aceasta conține o listă de observatori și metode pentru a le gestiona. Atunci când apar modificări în subiect, acesta notifică automat observatorii înregistrați.

3. Clasa UserObserver: Este o implementare concretă a unui observator. Are două metode, ``update(data)`` și ``format_notification(data)``, care sunt apelate atunci când subiectul notifică un eveniment relevant.

4. Clasa AdminObserver: O altă implementare concretă a unui observator, care gestionează notificările destinate administratorilor.

5. `app.py`: Aici se creează un obiect de tipul ``Subject``, care va funcționa ca subiect. Apoi, sunt înregistrați observatorii: ``AdminObserver()`` și ``UserObserver()``.

Prin utilizarea acestui pattern, se realizează o descuplare între subiect și observatori. Subiectul nu trebuie să știe nimic despre observatori și viceversa. Atunci când apare un eveniment relevant pentru observatori (cum ar fi adăugarea unei mașini noi sau închirierea unei mașini), subiectul pur și simplu notifică observatorii, iar aceștia reacționează conform implementării lor specifice. Acest lucru face ca codul să fie mai modular și mai ușor de întreținut.

```
app.py
subject = Subject()
subject.register(AdminObserver())
subject.register(UserObserver())
```

```
db_models.py
def rent_out(self):
    self.state = 'rented'
```

```
DatabaseFacade.py
def fetch_all_available_cars(self):
    try:
        cars_query = select(Cars).filter(Cars.state == 'available')
        cars = db.session.execute(cars_query).scalars().all()
        return cars
    except SQLAlchemyError as e:
        print("Error fetching available cars:", e)
        return []
```

```
def delete_rental(self, rental_id):
    rental = db.session.query(Rentals).get(rental_id)
    if rental:
        car_id = rental.car_id
```

```
# Update the status of the car to 'available'
car = db.session.query(Cars).get(car_id)
if car:
    car.state = 'available'
    db.session.commit()
else:
    print(f"Car {car_id} not found")

# Delete the rental after updating the car state
db.session.delete(rental)
db.session.commit()
```

Aici avem două bucăți de cod care se încadrează în pattern-ul de proiectare "State":

#### 1. Metoda `rent\_out` din `db\_models.py`:

- Această metodă pare să fie o metodă a unei clase din `db\_models.py` care este responsabilă pentru a închiria o mașină.
- În mod specific, această metodă modifică starea unei mașini din 'available' în 'rented', ceea ce sugerează că mașina trece printr-o tranziție de stare atunci când este închiriată.

#### 2. Metoda `delete\_rental` din `DatabaseFacade.py`:

Această metodă face parte dintr-o clasă de fațadă care gestionează interacțiunile cu baza de date, inclusiv operațiile legate de închirierea și returnarea mașinilor.

Atunci când o închiriere este ștearsă (probabil atunci când un utilizator returnează o mașină închiriată), starea mașinii asociate este actualizată la 'available'.

- Acest lucru indică o altă tranziție de stare, de data aceasta din 'rented' în 'available'.

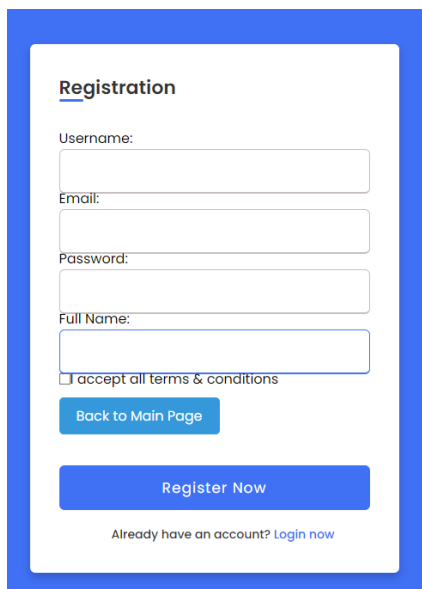
Cum lucrează pattern-ul "State":

- Pattern-ul "State" este un pattern de comportament care permite unui obiect să-și modifice comportamentul atunci când starea sa internă se schimbă.
- În acest caz, starea mașinii ('available' sau 'rented') determină comportamentul său.
- Atunci când o mașină este închiriată, starea acesteia este schimbată din 'available' în 'rented', ceea ce poate afecta modul în care alte părți ale sistemului interacționează cu acea mașină.
- Similar, atunci când o închiriere este ștearsă și mașina este returnată, starea mașinii este actualizată înapoi la 'available', modificând astfel comportamentul său în funcție de această stare.

În esență, pattern-ul "State" permite unui obiect să-și schimbe comportamentul pe baza stării sale interne, fără a fi nevoie de utilizarea unor instrucțiuni condiționale complicate în cod. Aceasta conduce la o implementare mai modulară și mai ușor de întreținut, deoarece logica specifică stării este izolată în clase dedicate și poate fi modificată independent.

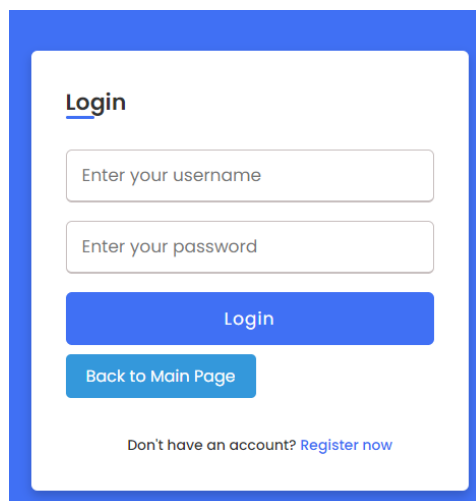
### 3. Documentarea produsul realizat

Aplicatia finala reprezinta o pagina web a unei platforme de inchiriere a masinilor Pentru a accesa funcționalitățile platformei noastre de închiriere auto, utilizatorii sunt întâmpinați cu o pagină de înregistrare și logare. Aici, ei au opțiunea de a-și crea un cont nou sau de a se autentifica dacă au deja un cont existent.

A registration form titled "Registration" with a blue border. It contains five input fields: "Username:", "Email:", "Password:", and "Full Name:". Below the "Full Name" field is a checkbox labeled "I accept all terms & conditions". There are two buttons: a blue "Back to Main Page" button and a larger blue "Register Now" button. At the bottom, it says "Already have an account? [Login now](#)".

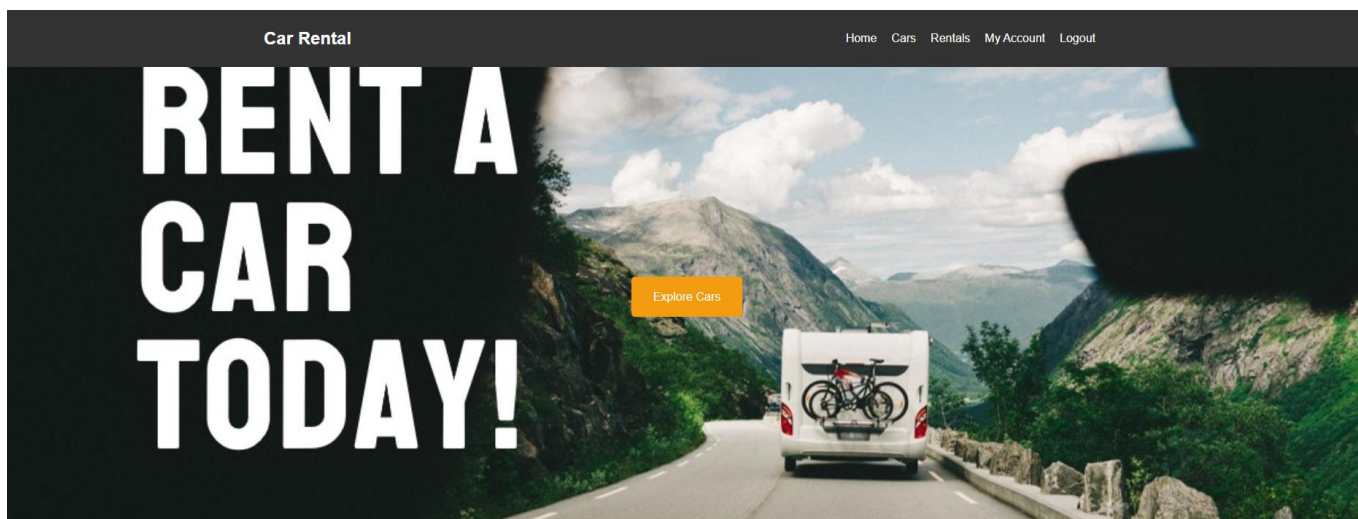
**Figura 3.1 Pagina de inregistrare**

- **Înregistrare:** Utilizatorii sunt ghidați printr-un proces simplu de înregistrare figura 3.1, unde sunt rugați să furnizeze detalii precum numele, adresa de email și o parolă sigură. Un formular intuitiv și interactiv facilitează introducerea acestor informații.

A login form titled "Login" with a blue border. It contains two input fields: "Enter your username" and "Enter your password". Below these fields are two buttons: a blue "Login" button and a smaller blue "Back to Main Page" button. At the bottom, it says "Don't have an account? [Register now](#)".

**Figura 3.2 Pagina de logare**

- **Logare:** Utilizatorii care au deja un cont pot folosi secțiunea de logare figura 3.2 pentru a accesa platforma. Ei trebuie să introducă adresa de email și parola asociată contului lor pentru a se autentifica.

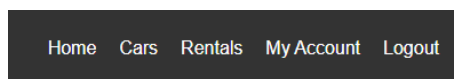


**Figura 3.3 Pagina de Home**

După autentificare sau înregistrare, utilizatorii sunt direcționați către pagina principală a platformei noastre. Această pagină servește ca punct central pentru toate funcționalitățile și informațiile relevante disponibile pentru utilizatori.

Pagina principală figura 3.3 oferă o prezentare generală a platformei noastre de închiriere auto. Utilizatorii pot găsi informații utile despre serviciile noastre, oferte speciale sau evenimente curente.

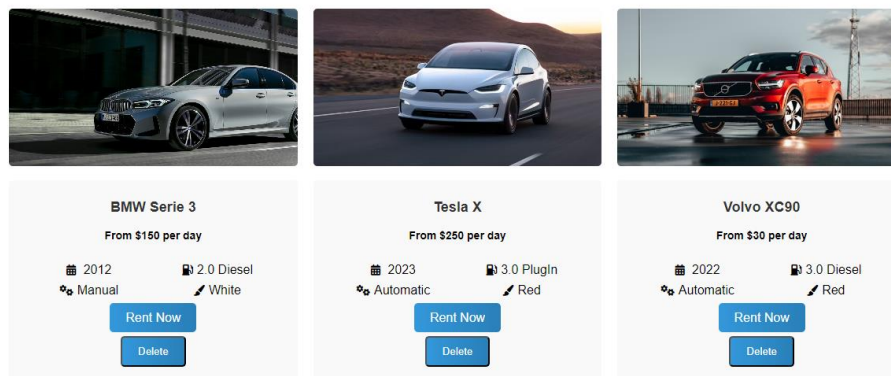
- **Meniu Navigare:** Pentru a oferi o navigare ușoară și accesibilă, am inclus un meniu de navigare care permite utilizatorilor să acceseze rapid și direct diferitele secțiuni ale platformei noastre. Meniul include următoarele opțiuni:



**Figura 3.4 Meniul de Navigare**

- **Home:** Acces rapid la pagina principală a platformei.
- **Cars:** Permite utilizatorilor să exploreze și să caute în întreaga gamă de mașini disponibile pentru închiriere.
- **Rentals:** Oferă utilizatorilor acces la istoricul și detaliile închirierilor lor curente și anterioare.
- **My Account:** Conduce utilizatorii către pagina lor personală de profil, unde pot gestiona informațiile contului și preferințele lor.
- **Logout:** Permite utilizatorilor să se deconecteze în siguranță din contul lor.

Pagina Cars oferă utilizatorilor o experiență interactivă și informativă pentru a explora și selecta mașina potrivită pentru nevoile lor de închiriere. Această pagină prezintă o gamă diversă de mașini disponibile pentru închiriere, fiecare cu detalii complete și imagini atractive.



**Figura 3.5 Pagina Cars**

### **Caracteristici Principale:**

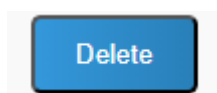
**Căutare și Filtrare:** Utilizatorii pot căuta mașinile disponibile.

**Vizualizare Detaliată:** Fiecare mașină listată include detalii complete, cum ar fi imaginea, marca, modelul, anul, prețul de închiriere, disponibilitatea și alte specificații relevante.

**Navigare Intuitivă:** Interfața utilizatorului este proiectată pentru o navigare ușoară și intuitivă, facilitând găsirea și compararea diferitelor mașini disponibile.

**Rezervare Rapidă:** Utilizatorii pot selecta rapid mașina dorită și pot avansa în procesul de rezervare fără a părăsi pagina. Un flux simplu și eficient asigură o experiență plăcută pentru utilizatori.

### **Funcționalități Adicionale pentru Admin:**



**Figura 3.6 Butonul Delete Car**

- **Buton pentru Ștergere Mașină (Delete Car)( figura 3.6):** Administratorii au acces la un buton special pentru a șterge o mașină din sistem. Această funcționalitate permite administrarea eficientă a mașinilor disponibile pentru închiriere.

# Add Car

Add Car

Figura 3.7 Butonul Add Car

- **Buton pentru Adăugare Mașină (Add Car)(figura 3.7):** Administratorii pot adăuga rapid mașini noi în sistem folosind un buton dedicat pentru adăugarea unei noi mașini în baza de date. Această opțiune simplifică procesul de actualizare și extindere a listei de mașini disponibile.

## BMW

### Serie 3

Year: 2012

Color: White

Price per day:

\$150

Engine: 2.0

Diesel

Gearbox:

Manual

#### Ready to Rent?

Start Date:

End Date:

☐ Child Seat

☐ GPS

☐ Roof Bag

Rent Car

Back to Cars

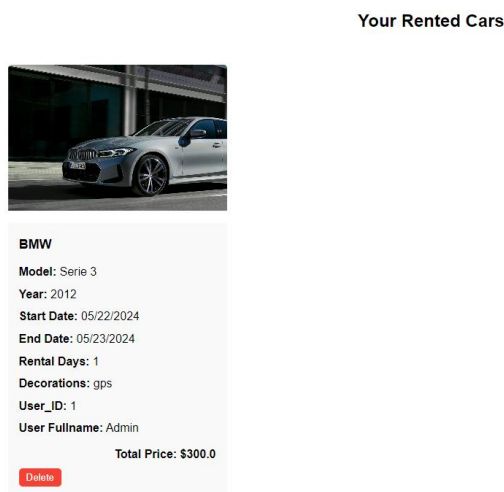
Figura 3.8 Pagina detaliata

Pagina Detaliată a Mașinii furnizează figura 3.8 o prezentare detaliată a unei mașini specifice, inclusiv informații extinse despre specificațiile tehnice, aspectul estetic și disponibilitatea acestora pentru închiriere.

#### Elemente Cheie ale Paginii:

- **Imagini și Descriere:** Pagina prezintă imagini multiple ale mașinii, permițând utilizatorilor să examineze detaliile din diferite unghiuri. În plus, o descriere succintă oferă informații esențiale despre caracteristicile și performanțele mașinii.
- **Selectare Dată Închiriere:** Utilizatorii pot selecta data de început și data de sfârșit a perioadei de închiriere utilizând un calendar interactiv. Această funcționalitate le permite să vizualizeze disponibilitatea mașinii și să facă programări precise.
- **Selectare Decoratiuni:** Utilizatorii pot personaliza experiența de închiriere a mașinii selectând opțiuni de decorare suplimentare. Acestea pot include accesorii cum ar fi scaun pentru copii, sistem de navigație GPS sau suport pentru bagaje pe acoperiș.

- **Buton pentru Rezervare:** După ce au efectuat toate selecțiile necesare, utilizatorii pot finaliza procesul de rezervare făcând clic pe un buton special pentru a avansa la etapa următoare. Aceasta îi va direcționa către pagina de confirmare a rezervării.

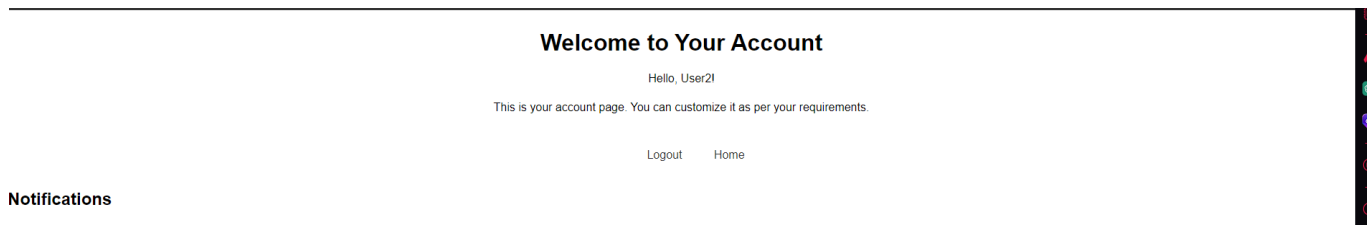


**Figura 3.8 Pagina rentals**

Pagina Rentals figura 3.8 furnizează o listă comprehensivă a mașinilor pe care utilizatorul le-a închiriat anterior, împreună cu informații relevante despre fiecare închiriere. Aici, utilizatorii pot vizualiza și gestiona cu ușurință rezervările lor existente.

#### **Elemente Cheie ale Paginii:**

- **Lista Mașinilor Închiriate:** Pagina prezintă o listă detaliată a mașinilor pe care utilizatorul le-a închiriat anterior. Pentru fiecare închiriere, sunt afișate informații cum ar fi modelul mașinii, perioada de închiriere și prețul total.
- **Detalii Suplimentare:** Utilizatorii pot vizualiza informații suplimentare despre fiecare închiriere, cum ar fi data de început și data de sfârșit a perioadei de închiriere, precum și opțiunile de decorare selectate.
- **Buton pentru Anulare:** Pentru fiecare închiriere, este disponibil un buton special pentru anulare. Utilizatorii pot face clic pe acest buton pentru a anula rezervarea și pentru a elibera mașina pentru alți potențiali clienți.



**Figura 3.9 Pagina My Account**

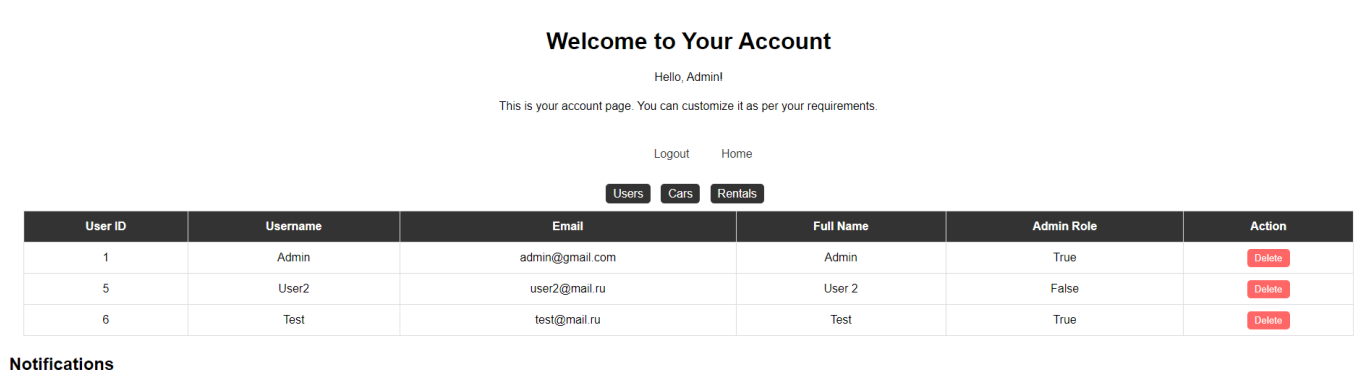
Pagina My Account figura 3.9 este destul de simplă și prezintă o salutare personalizată, urmată de două butoane importante pentru navigare.

**Elemente Cheie ale Paginii:**

- **Salut Personalizat:** Pagina începe cu un mesaj de bun venit personalizat, care să-i întâmpine pe utilizatori. Acest mesaj poate include numele utilizatorului și poate fi adaptat în funcție de tipul de utilizator (de exemplu, "Hello, Admin!" sau "Welcome back, John!").
- **Buton de Logout:** Utilizatorii pot face clic pe acest buton pentru a se deconecta din contul lor și pentru a încheia sesiunea curentă. Acest buton este esențial pentru a oferi utilizatorilor control asupra securității contului lor.
- **Buton Home:** Pentru a facilita navigarea înapoi la pagina principală a platformei, utilizatorii au acces la un buton Home. Acest buton îi va redirecționa pe utilizatori înapoi la pagina principală a aplicației.

**Mesaj Descriptiv:**

Pe lângă salutul personalizat, pagina poate include un scurt mesaj descriptiv care să informeze utilizatorii cu privire la funcționalitățile disponibile pe pagina My Account și la modul în care acestea pot personaliza experiența lor pe platformă.



**Figura 3.9 Pagina My Account (Admin)**



### **Secțiunea pentru Administratori:**

- **Gestionare Masini:** Un link către o pagină specială de gestionare a mașinilor, unde administratorii pot adăuga, edita sau șterge mașini din baza de date. Aici, ei pot accesa și butonul "Delete Car" menționat anterior.
- **Gestionare Inchirieri:** Un link către o pagină dedicată pentru a vizualiza toate închirierile în curs și pentru a gestiona aceste închirieri. Acesta este locul unde administratorii pot vedea ce mașini sunt închiriate în prezent și de cine.
- **Notificări:** O casetă de notificări în care administratorii pot primi alerte și notificări importante despre activitatea pe platformă. Aceasta ar putea include alerte pentru închirieri noi, actualizări ale mașinilor și alte informații esențiale pentru administrare.

Această secțiune specială pentru administratori oferă o interfață centralizată și ușor accesibilă pentru a gestiona toate aspectele cheie ale platformei de închirieri auto.

## Concluzii

În timpul dezvoltării acestui proiect, am întâmpinat diverse provocări și am aplicat mai multe pattern-uri de proiectare pentru a construi o soluție solidă și eficientă. Implementarea acestor pattern-uri a fost esențială pentru crearea unei arhitecturi coerente și ușor de întreținut. Deși unele aspecte ale proiectului au fost mai dificile decât altele, am reușit să învățăm multe lucruri valoroase din această experiență.

Prin aplicarea Factory Pattern, am putut gestiona crearea obiectelor de tip Car într-un mod flexibil, permitând extinderea și adaptarea ușoară a codului nostru la schimbările cerințelor. Builder Pattern ne-a ajutat să construim obiectele de tip Rentals într-un mod modular și structurat, facilitând gestionarea proprietăților acestora și oferind o modalitate elegantă de configurare a închirierilor.

Observer Pattern a fost esențial pentru implementarea sistemului de notificări, permițând comunicarea eficientă între diferitele componente ale aplicației noastre. Prin intermediul Strategy Pattern, am gestionat diferitele strategii de preț în funcție de numărul de zile de închiriere, oferind flexibilitate și scalabilitate în gestionarea politicii de prețuri.

Chiar dacă întâmpinarea anumitor dificultăți a fost inevitabilă, aceste provocări ne-au oferit oportunități de învățare și creștere. Am devenit mai conștienți de bunele practici de dezvoltare și de importanța aplicării pattern-urilor de proiectare în construirea unui cod robust și ușor de întreținut.

În concluzie, acest proiect ne-a oferit nu doar o experiență valoroasă în dezvoltarea unei aplicații web, ci și oportunitatea de a învăța și de a crește ca dezvoltatori. Am înțeles importanța aplicării conceptelor de proiectare și dezvoltare software și ne-am consolidat abilitățile în implementarea acestora într-un mediu real de dezvoltare de aplicații.

## Bibliografie

1. "Refactoring Guru - Design Patterns & Refactoring". Disponibil la: <https://refactoring.guru/>.
2. Freeman, Eric, Elisabeth Robson, Kathy Sierra, și Bert Bates. "Head First Design Patterns". O'Reilly Media, 2004.
3. Gamma, Erich, Richard Helm, Ralph Johnson, și John Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1994.
4. Martin, Robert C. "Clean Code: A Handbook of Agile Software Craftsmanship". Prentice Hall, 2008.
5. "Design Patterns - GeeksforGeeks". Disponibil la: <https://www.geeksforgeeks.org/design-patterns/>.
6. "Design Patterns - Tutorialspoint". Disponibil la: [https://www.tutorialspoint.com/design\\_pattern/index.htm](https://www.tutorialspoint.com/design_pattern/index.htm).
7. "Design Patterns in Python - Real Python". Disponibil la: <https://realpython.com/design-patterns-python/>.
8. Design Patterns - SourceMaking". Disponibil la: [https://sourcemaking.com/design\\_patterns](https://sourcemaking.com/design_patterns).
9. 15. "Design Patterns - Dofactory". Disponibil la: <https://www.dofactory.com/net/design-patterns>.
10. 10 Design Patterns Explained in 10 minutes. Disponibil la : <https://www.youtube.com/watch?v=tv-1er1mWI>

Minim 10 surse!

