**MINISTERUL EDUCAŢIEI, CULTURII ȘI CERCETĂRII**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare Informatică şi Microelectronică**

**Departamentul Ingineria Software și Automatică**

**Proiect de an**

**Disciplina:**Tehnici și Mecanisme de Proiectare a Produselor Program

**Tema:** Implementarea platformei online pentru închirierea de vehicule

(The implementation of the online platform for vehicle rental)

|  |  |  |
| --- | --- | --- |
| **Student:** | **\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Popa Cătălin, TI-211** |
| **Coordonator:** | **\_\_\_\_\_\_\_\_\_\_\_\_\_** | **Cebotari Daria, asis. univ.** |
|  |  |  |

**Chişinău, 2024**

**Cuprins**

[Introducere 2](#_Toc163412614)

[1 Analiza domeniului de studiu 3](#_Toc163412615)

[1.1 Scopul, obiectivele și cerințele sistemului 4](#_Toc163412616)

[1.2 Analiza sistemelor deja existente 28](#_Toc163412617)

[2. Realizarea sistemului 29](#_Toc163412618)

[2.1 Proiectarea aplicației 29](#_Toc163412619)

[2.2 Descrierea tehnologiilor pentru sistem 29](#_Toc163412620)

[2.3 Descrierea la nivel de cod pe module 30](#_Toc163412621)

[3. Documentarea produsul realizat 31](#_Toc163412622)

[Concluzii 32](#_Toc163412623)

[Bibliografie 33](#_Toc163412624)

[Anexa A 34](#_Toc163412625)

# Introducere

Lucrarea dată urmărește scopul de a elabora crearea unei platforme online pentru închirierea vehiculelor, ca răspuns la nevoile pieței în creștere pentru mobilitate și accesibilitate în transport. Lumea modernă este tot mai interconectată și dinamică, iar o platformă digitală de închiriere a vehiculelor devine o necesitate. Această soluție va fi convenabilă și eficientă pentru utilizatori.

Lucrarea este structurată în trei capitole, unde se va analiza concret modul de proiectare și se va explora în detaliu etapele de realizare a platformei. Vor fi evidențiate aspectele cheie ale domeniului, structura conceptuală a platformei și tehnologiile implicate.

Primul dintre aceste capitole vine să prezinte în profunzime domeniul ales, clarificând scopul, obiectivele și cerințele sistemului. Prin intermediul unei analize mai profunde, se examinează sistemul din persepctiva cerințelor utilizatorilor și a tendințelor pieței, având în vedere și existența altor soluții deja existente pe piață.

Al doilea capitol cuprinde partea de implementarea a codului pentru platforma de închiriere a vehiculelor. Se va detalia proiectarea aplicației, incluzând aspecte precum arhitectura sistemului, fluxurile de date și interfețele utilizatorului. De asemenea, se vor descrie tehnologiile utilizate pentru dezvoltarea sistemului, precum și modul în care acestea sunt integrate în proiect.

Ultimul capitol este dedicat documentării produsului final, evidențiind funcționalitățile implementate, structura și modul de utilizare a platformei de închiriere a vehiculelor. Se vor detalia aspecte practice ale implementării, oferind informații utile pentru utilizatori și dezvoltatori.

În concluzie, această lucrare urmărește să ofere o viziune detaliată asupra procesului de creare a unei platforme online de închiriere a vehiculelor, evidențiind atât aspectele teoretice, cât și cele practice implicate într-un proiect de acest tip. Utilizând șabloanele și tehnici moderne de dezvoltare software, precum și abordări eficiente de proiectare și implementare, se urmărește să se ofere o soluție eficientă și inovatoare pentru nevoile specifice ale utilizatorilor din acest domeniu în continuă evoluție.

In acest capitol se specifica ce anume va contine lucrarea in sine(domeniul ales, sabloanele folosite si mecanismele de implimentare TOTUL LA GENERAL)

# Analiza domeniului de studiu

Înainte de a se aborda proiectarea și implementarea unei platforme online de închiriere a vehiculelor, este nevoie să se efectueze o analiză detaliată a domeniului de studiu. Lucrarea prezintă analiza și explorarea domeniului închirierii de vehicule în contextul creării unei platforme online pentru acest scop. Într-o lume tot mai interconectată și dinamică, o platformă digitală de închiriere auto devine imperativă pentru a răspunde nevoilor pieței în creștere pentru mobilitate și accesibilitate în transport. Scopul acestei analize este de a evidenția aspectele cheie ale domeniului și de a oferi o înțelegere profundă a cerințelor utilizatorilor și a tendințelor pieței.

În primul rând, este important să evidențiem creșterea continuă a cererii pentru serviciile de închiriere auto într-o lume în care mobilitatea este din ce în ce mai importantă. Datele statistice relevante arată o creștere constantă a pieței de închiriere auto în ultimii ani, reflectând schimbările din comportamentul consumatorilor și nevoia acestora de flexibilitate în transport. Utilizatorii din ce în ce mai conectați la internet și mobili doresc soluții de transport convenabile și accesibile, ceea ce face o platformă online de închiriere auto o necesitate în era digitală.

Analiza concurenței în acest domeniu relevă prezența unor platforme deja existente de închiriere auto online. Este esențial să evaluăm punctele forte și slabe ale acestor competitori pentru a identifica oportunitățile de diferențiere și îmbunătățire a propriei platforme. Benchmarking-ul cu competitorii va ajuta la înțelegerea cerințelor pieței și a așteptărilor utilizatorilor, oferind astfel o bază solidă pentru dezvoltarea platformei noastre.

O analiză profundă a cerințelor și așteptărilor utilizatorilor este importantă pentru proiectarea unei platforme de închiriere auto online de succes. Feedback-ul de la potențiali utilizatori și cercetările de piață relevante vor oferi o înțelegere mai clară a funcționalităților și caracteristicilor dorite de către aceștia. Utilizatorii doresc o experiență simplă, convenabilă și sigură atunci când utilizează o platformă de închiriere auto online, iar satisfacerea acestor nevoi va fi o prioritate în dezvoltarea platformei noastre.

Integrarea tehnologiilor moderne și a inovațiilor în dezvoltarea platformei noastre va juca un rol esențial în asigurarea succesului acesteia pe piață. Inteligența artificială pentru recomandări personalizate, tehnologiile de securitate pentru protejarea datelor utilizatorilor și alte tehnologii de vârf vor contribui la crearea unei platforme avansate și eficiente din punct de vedere tehnic. În fine, respectarea reglementărilor și a cerințelor legale este esențială în domeniul închirierii auto online. Asigurarea conformității cu toate reglementările din domeniul transportului și al comerțului electronic va consolida încrederea utilizatorilor în platforma noastră și va evita posibilele probleme legale în viitor.

Prin analiza detaliată a acestor aspecte cheie ale domeniului, lucrarea își propune să ofere o viziune comprehensivă asupra procesului de creare a unei platforme online de închiriere auto. Evidențiind aspectele teoretice și practice implicate într-un proiect de acest tip, ne propunem să dezvoltăm o soluție eficientă și inovatoare pentru nevoile specifice ale utilizatorilor din acest domeniu în continuă evoluție.

# 1.1 Scopul, obiectivele și cerințele sistemului

Scopul acestui proiect constă în dezvoltarea unei platforme online pentru închirierea vehiculelor, care să ofere o soluție modernă și eficientă pentru nevoile de mobilitate și accesibilitate în transport ale utilizatorilor. Această platformă va fi construită pe tehnologia web (World Wide Web - WWW), asigurând astfel accesibilitatea și interoperabilitatea pe diverse dispozitive și platforme. În plus, sistemul va fi proiectat și implementat în conformitate cu principiile SOLID și va folosi sabloanele de proiectare (design patterns) pentru a asigura o arhitectură robustă, flexibilă și ușor de întreținut.

Platforma va respecta următoarele obiective principale, pentru a realiza o platformă sigură, scalabilă și performantă:

* Interfață utilizator simplificată;
* securitatea datelor și tranzacțiilor;
* funcționalități avansate;
* performanță și scalabilitate;
* personalizare și adaptabilitate.

Dezvoltarea unei interfețe intuitive și ușor de utilizat pentru utilizatorii platformei, facilitând procesul de căutare, rezervare și închiriere a vehiculelor. Va fi respectată asigurarea securității informațiilor utilizatorilor și a tranzacțiilor online prin implementarea unor măsuri adecvate de protecție a datelor și autentificare. Vor fi implementate funcționalități avansate, precum căutarea și filtrarea vehiculelor în funcție de preferințele utilizatorului, gestionarea rezervărilor și procesarea plăților online. Sistematic va avea loc optimizarea performanței și scalabilității platformei pentru a gestiona un volum crescut de utilizatori și tranzacții fără a compromite experiența utilizatorilor. Vor fi integrate tehnologiilor moderne pentru a oferi o experiență personalizată și adaptată nevoilor și preferințelor individuale ale utilizatorilor.

Cerințele funcționale definesc ce face sistemul, cerințele non-funcționale definesc cum se comportă și performează sistemul, iar cerințele tehnice definesc cu ce tehnologii și standarde este construit sistemul. Toate aceste cerințe trebuie luate în considerare în procesul de dezvoltare a unui sistem software pentru a asigura că acesta îndeplinește cu succes obiectivele și nevoile utilizatorilor săi. Pentru a defini clar și obiectiv cerințele sistemului, acestea sunt împărțite în trei categorii principale:

* Cerințe funcționale;
* cerințe non-funcționale;
* cerințe tehnice.

Cerințele funcționale sunt cerințele care definesc funcționalitățile sau comportamentele sistemului software. Ele se concentrează pe ceea ce sistemul trebuie să facă pentru a îndeplini obiectivele și nevoile utilizatorilor. Exemple de cerințe funcționale includ: autentificarea utilizatorilor, căutarea și afișarea informațiilor, procesul de închiriere a vehiculelor, sistemul de evaluare și feedback, integrarea cu servicii de plată. Aceste cerințe sunt esențiale pentru funcționarea corectă a sistemului și sunt adesea exprimate sub formă de scenarii de utilizare sau fluxuri de lucru. Platforma trebuie să ofere funcționalități complete de autentificare și autorizare, asigurând că doar utilizatorii înregistrați și autorizați au acces la funcționalitățile platformei. Utilizatorii trebuie să poată căuta și vizualiza vehiculele disponibile în funcție de diferite criterii, cum ar fi locația, tipul de vehicul, disponibilitatea și prețul. Platforma trebuie să ofere un proces complet și intuitiv de închiriere a vehiculelor, inclusiv rezervare, confirmare și gestionare a tranzacțiilor. Utilizatorii ar trebui să poată selecta vehiculul dorit, să stabilească perioada de închiriere și să finalizeze tranzacția în mod eficient. Pentru a îmbunătăți experiența utilizatorilor, platforma trebuie să ofere un sistem de evaluare și feedback pentru utilizatori și vehicule. Utilizatorii ar trebui să poată evalua și lăsa feedback despre experiența lor cu serviciul de închiriere și despre vehiculele închiriate. Pentru a facilita procesul de plată, platforma trebuie să fie integrată cu servicii de plată online, permițând utilizatorilor să efectueze tranzacții sigure și convenabile pentru închirierea vehiculelor.

Cerințele non-funcționale sunt cerințele care definesc caracteristicile sistemului care nu sunt legate direct de funcționalități specifice, ci mai degrabă de calități globale ale sistemului sau de performanță. Exemple de cerințe non-funcționale includ: securitatea datelor, performanța sistemului, scalabilitatea, interfața utilizatorului, fiabilitatea și disponibilitatea, eficiența energetică. Aceste cerințe sunt la fel de importante ca și cerințele funcționale, deoarece afectează experiența utilizatorului și succesul general al sistemului. Platforma trebuie să respecte cele mai înalte standarde de securitate pentru protejarea datelor utilizatorilor și a tranzacțiilor online. Aceasta include criptarea datelor sensibile, implementarea autentificării în doi factori și utilizarea protocolului HTTPS pentru comunicarea securizată. Platforma trebuie să ofere o performanță ridicată și un timp de răspuns rapid pentru a asigura o experiență fluidă utilizatorilor. Aceasta include optimizarea codului sursă, gestionarea eficientă a resurselor serverului și utilizarea cache-ului pentru a reduce timpul de încărcare al paginilor. Platforma trebuie să fie scalabilă pentru a gestiona creșterea volumului de utilizatori și tranzacții fără a afecta performanța sau disponibilitatea sistemului. Aceasta include utilizarea tehnologiilor și arhitecturilor scalabile, precum serviciile cloud și bazele de date distribuite. Interfața utilizator trebuie să fie intuitivă și ușor de navigat, satisfăcând nevoile utilizatorilor pe diferite dispozitive și platforme. Aceasta include proiectarea unei interfețe responsive, care se adaptează la diferite dimensiuni de ecrane și moduri de utilizare. Platforma trebuie să fie fiabilă și disponibilă non-stop, asigurând că utilizatorii pot accesa și utiliza serviciul în orice moment fără întreruperi planificate.

Cerințele tehnice sunt cerințele care definesc tehnologiile, standardele și metodologiile care trebuie utilizate în dezvoltarea și implementarea sistemului software. Exemple de cerințe tehnice includ: tehnologiile de dezvoltare (cum ar fi limbajele de programare, bazele de date), arhitectura sistemului (cum ar fi arhitectura client-server sau arhitectura bazată pe microservicii), standardele de securitate, cerințele de performanță. Aceste cerințe sunt esențiale pentru a asigura că sistemul este dezvoltat și implementat în conformitate cu cele mai bune practici și standardele industriale. Platforma va fi dezvoltată și implementată folosind tehnologia web, asigurând accesibilitatea și interoperabilitatea pe diferite dispozitive și platforme. Proiectarea și implementarea codului sursă vor respecta principiile SOLID garantând un cod modular, ușor de întreținut și de extins. Pentru a asigura o arhitectură robustă și ușor de întreținut, se vor utiliza diverse sabloane de proiectare. Acestea vor ajuta la structurarea și organizarea codului într-un mod clar și eficient, facilitând dezvoltarea și întreținerea sistemului.

Principiile SOLID[5] reprezintă un set de cinci principii de proiectare a codului, propuse de Robert C. Martin, care ajută la crearea unor aplicații software mai flexibile, mai ușor de întreținut și mai robuste.

Principiul Responsabilității Unice afirmă că o clasă ar trebui să aibă o singură responsabilitate și să fie responsabilă pentru un singur aspect al funcționalității aplicației. Acest lucru înseamnă că o clasă ar trebui să fie modificată doar atunci când există o schimbare în responsabilitatea pe care o deține. De exemplu, într-o aplicație de gestionare a bibliotecii, o clasă care gestionează înregistrările de utilizatori ar trebui să se concentreze doar pe operațiunile legate de utilizatori, cum ar fi adăugarea, ștergerea sau actualizarea informațiilor acestora. Orice logică referitoare la alte aspecte ale aplicației, cum ar fi gestionarea împrumuturilor de cărți, ar trebui să fie tratată în alte clase. Aplicând Principiul Responsabilității Unice, dezvoltatorii pot crea clase mai coezive, mai ușor de înțeles și de întreținut, reducând riscul de încălcări ale integrității și eliminând dependențele nedorite între diferitele părți ale aplicației.

Principiul Deschis/Închis afirmă că o entitate software, cum ar fi o clasă, ar trebui să fie deschisă pentru extindere, dar închisă pentru modificare. Cu alte cuvinte, odată ce o clasă este implementată și funcționează corect, ar trebui să fie posibil să se adauge funcționalități noi fără a modifica codul existent. De exemplu, într-o aplicație de comerț electronic, o clasă care gestionează metodele de plată ar trebui să fie proiectată astfel încât să poată fi extinsă pentru a accepta noi metode de plată, cum ar fi carduri de credit, portofele electronice sau criptomonede, fără a fi necesară modificarea codului existent. Principiul Deschis/Închis promovează modularitatea și flexibilitatea în cadrul aplicațiilor, permițând dezvoltatorilor să adauge noi funcționalități fără a risca să strice funcționalitățile existente și să creeze bug-uri nedorite.

Principiul Substituirii Liskov afirmă că obiectele unei clase derivate ar trebui să poată fi folosite ca și instanțe ale clasei de bază fără a afecta corectitudinea programului. Mai simplu spus, dacă o clasă B este o subclasă a clasei A, atunci obiectele de tip B ar trebui să poată fi folosite în locul obiectelor de tip A fără a schimba comportamentul programului. De exemplu, într-o aplicație de gestionare a formelor geometrice, dacă avem o clasă de bază "Formă" cu metoda "calculeazăAria()", atunci orice subclasă, cum ar fi "Pătrat" sau "Cerc", ar trebui să implementeze și să poată fi folosite în locul clasei de bază fără a afecta corectitudinea programului. Aplicând Principiul Substituirii Liskov, dezvoltatorii pot crea ierarhii de clase coerente și flexibile, în care subclaselor le este ușor să extindă și să modifice comportamentul clasei de bază, fără a provoca erori sau comportamente neașteptate în cadrul aplicației. Principiul Segregației Interfețelor afirmă că o interfață ar trebui să fie specifică pentru nevoile clientului și să nu impună implementarea de metode inutile sau neutilizate. În loc de a avea o singură interfață mare care conține toate metodele posibile, este preferabil să se creeze interfețe mai mici și mai specializate, care să se concentreze pe un singur aspect al funcționalității. De exemplu, într-o aplicație de editare foto, o interfață pentru un instrument de desen ar trebui să conțină doar metode legate de desen, cum ar fi "deseneazăLinie()" sau "deseneazăCerc()", fără a conține metode neutilizate, cum ar fi "salveazăImagine()" sau "încarcăImagine()". Principiul Segregației Interfețelor promovează coeziunea și modularitatea în cadrul aplicațiilor, facilitând reutilizarea și întreținerea codului și reducând dependențele între diferitele componente ale aplicației.

Principiul Inversării Dependințelor afirmă că clasele ar trebui să fie dependente de interfețe, nu de implementările acestora, iar modulele de nivel înalt nu ar trebui să depindă de modulele de nivel inferior, ci ambele ar trebui să depindă de abstracțiuni. De exemplu, într-o aplicație de gestionare a facturilor, clasa care generează facturile ar trebui să fie dependentă de o interfață "MetodăDePlată", în loc să fie dependentă direct de clase concrete precum "CardDeCredit" sau "PayPal". Acest lucru facilitează schimbul de implementări ale metodei de plată fără a afecta codul care generează facturile. Principiul Inversării Dependințelor promovează flexibilitatea și reutilizarea codului în cadrul aplicațiilor, facilitând testarea și întreținerea acestora și reducând cuplajul între diferitele componente ale aplicației.

Pattern-urile de proiectare reprezintă soluții comune la problemele întâlnite în dezvoltarea software și sunt esențiale pentru proiectarea și implementarea sistemelor eficiente și ușor de întreținut. Acestea oferă un set de practici și abordări testate care ajută dezvoltatorii să rezolve diverse provocări întâlnite în timpul dezvoltării software. Paternurile creationale sunt concepute pentru a facilita procesul de creare a obiectelor într-o aplicație. Printre acestea se numără Singleton-ul, care asigură că o clasă are o singură instanță și furnizează un punct global de acces la acea instanță. În figura 1.1, se afișează structura la Singleton[3].

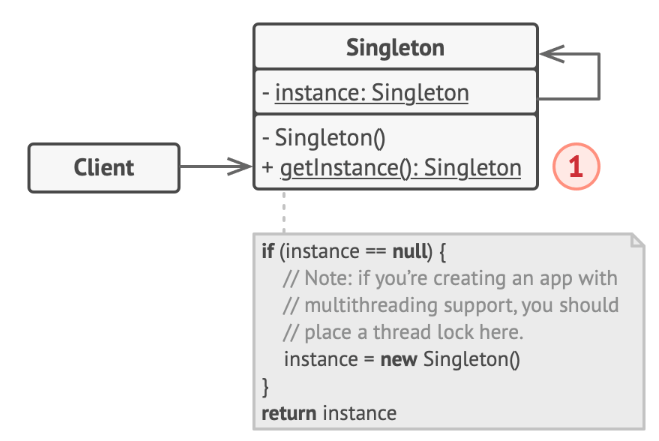


Figura 1.1 - Șablonul de proiectare Singleton

Acesta este util în situații în care trebuie să existe o singură instanță a unei clase în întreaga aplicație, cum ar fi managerul de fișiere sau obiectul de configurare. Factory Method-ul este un alt pattern creational, care definește o interfață pentru crearea unui obiect, dar lasă subclaselor să decidă ce clasă să instanțieze. Patternul Factory Method este unul dintre cele mai utilizate și fundamentale modele de proiectare în dezvoltarea software-ului. Este clasificat ca un pattern de creare, însemnând că se concentrează pe modul în care obiectele sunt create într-un sistem. Factory Method Pattern este un pattern de proiectare care permite crearea de obiecte fără a specifica clasa exactă a obiectului care va fi creat. În loc să creăm obiectele direct folosind constructorii acestora, folosim o metodă de fabrică (factory method) pentru a crea obiectele. Această metodă este definită într-o clasă separată sau este o metodă abstractă într-o clasă abstractă, iar clasele concrete care extind această clasă abstractă vor implementa această metodă pentru a crea instanțele specifice ale obiectelor. În figura 1.2, este reprezentată structura șablonului Factory Method.

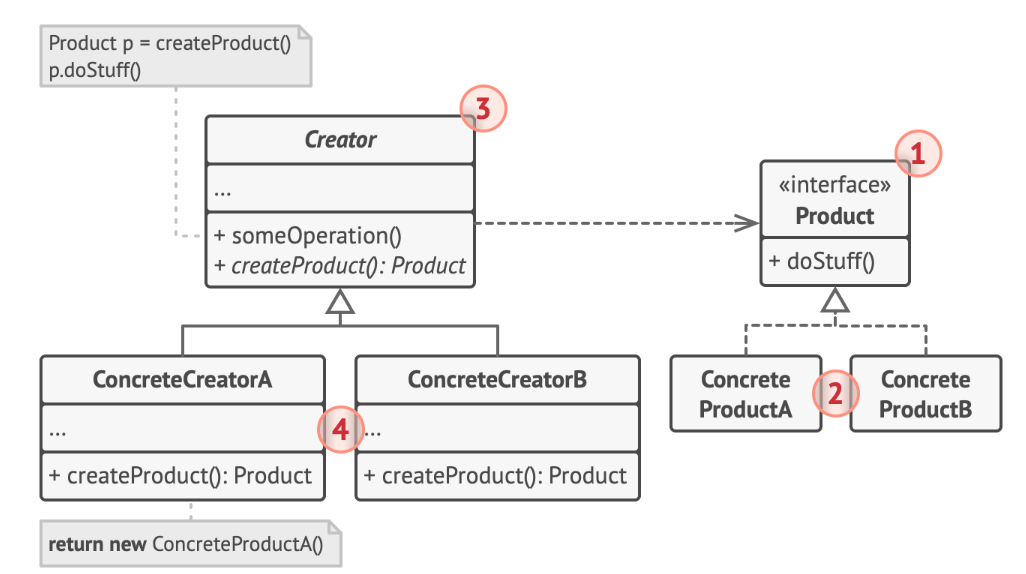


Figura 1.2 – Șablonul de proiectare Factory Method

Acesta este util atunci când procesul de creare a unui obiect este complex sau necesită logică suplimentară.

Abstract Factory-ul este ca un magician al codului, care nu doar creează obiecte, ci întrege familii de obiecte, cu o simplă mișcare a baghetei sale de programare. În lumea fascinantă a dezvoltării software, acest concept nu este doar o simplă metodă de creare a obiectelor, ci o adevărată artă a organizării și structurării codului. Unul dintre aspectele interesante și mai puțin cunoscute ale pattern-ului Abstract Factory este capacitatea sa de a asigura compatibilitatea între obiecte într-un mod elegant și eficient. Da, acest pattern nu doar creează obiecte, ci se asigură că acestea funcționează perfect împreună ca o echipă bine orchestrată. Imaginați-vă un orchestru în care fiecare instrument cântă o notă perfectă pentru a completa armonia generală - aceasta este esența Abstract Factory-ului în lumea dezvoltării software. Mai jos în figura 1.3 este afișată schema șablonului de proiectare Abstract Factory.

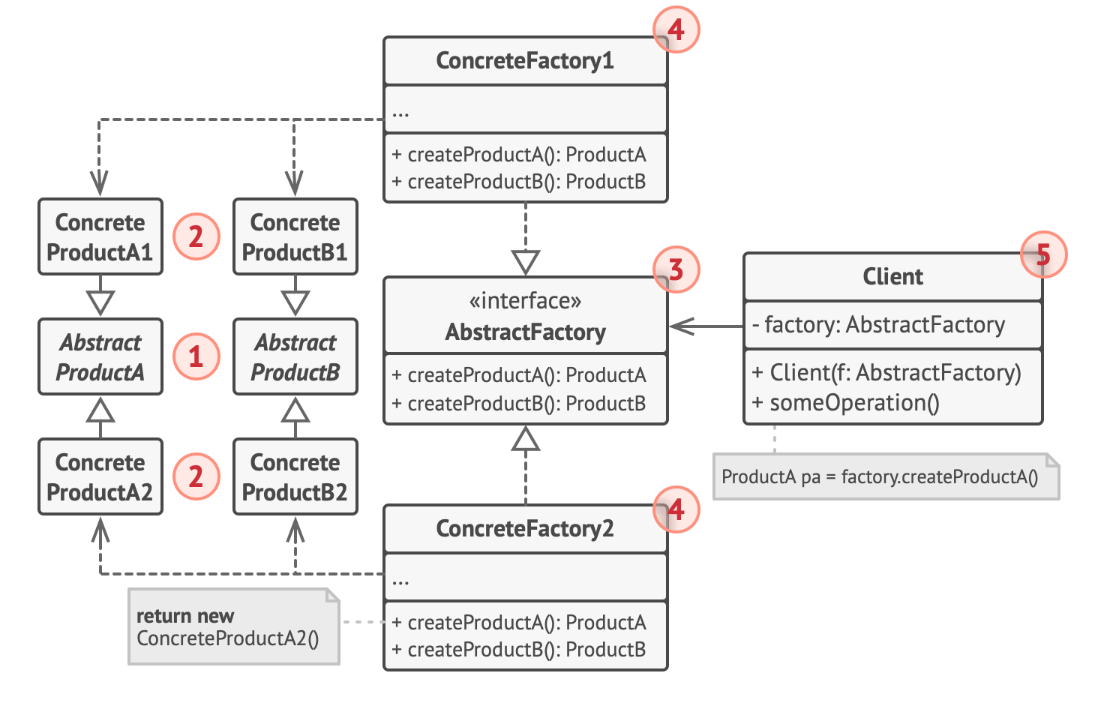


Figura 1.3 – Șablonul de proiectare Abstract Factory

Alte două pattern-uri creationale importante sunt Prototype-ul, care permite crearea de noi obiecte prin clonarea unui obiect existent, și Builder-ul, care ajută la construirea unui obiect complex pas cu pas, oferind o interfață pentru construirea pașilor și asamblarea finală a obiectului. Pattern-ul Builder este ca un arhitect priceput care nu doar construiește o clădire, ci o sculptează cu grijă din temelie până în vârf, transformând visurile în realitate fără compromisuri. În lumea dezvoltării software, Builder-ul nu este doar o unealtă pentru crearea obiectelor, ci o metodă ingenioasă de a construi și configura obiecte complexe într-un mod flexibil și elegant. Unul dintre aspectele remarcabile și mai puțin cunoscute ale pattern-ului Builder este capacitatea sa de a gestiona configurări diferite ale aceluiași obiect într-un mod eficient și clar. Imaginează-ți că ai nevoie de mai multe tipuri de mașini, fiecare cu propriile lor opțiuni și accesorii. Cu Builder-ul, poți crea și configura fiecare tip de mașină cu ușurință, fără a polua codul cu o mulțime de constructori diferiți sau parametri opționali. În plus, Builder-ul nu este doar o unealtă pentru dezvoltatorii pricepuți, ci și o soluție ingenioasă pentru echipele care lucrează în mod colaborativ. Echipa poate defini interfața comună a constructorilor, iar fiecare membru poate implementa propriile sale construcții care respectă această interfață. Astfel, colaborarea devine mai ușoară și mai eficientă, iar codul devine mai modular și mai ușor de întreținut. Mai jos în figura 1.4, este reprezentată structura șablonului de proiectare Builder.

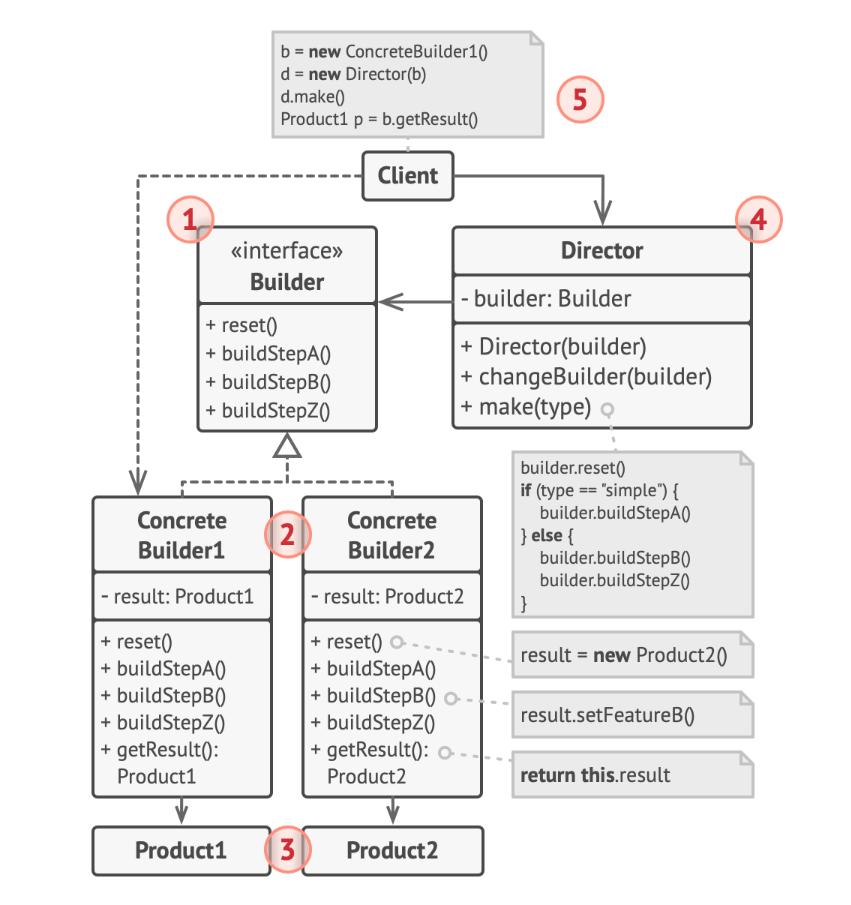


Figura 1.4 – Șablonul de proiectare Builder

Pattern-ul Prototype este precum un artist talentat care nu doar creează o singură operă de artă, ci deschide o ușă către o lume întreagă de posibilități, permitându-i să creeze și să re-creeze cu ușurință într-un mod infinit. În domeniul dezvoltării software, Prototype-ul nu este doar o metodă simplă de creare a obiectelor, ci o abordare ingenioasă care permite clonarea obiectelor existente pentru a crea noi instanțe într-un mod eficient și flexibil. Un aspect fascinant și mai puțin cunoscut al pattern-ului Prototype este capacitatea sa de a gestiona obiecte complexe și interconectate într-un mod elegant. Imaginați-vă că aveți nevoie să creați o serie de obiecte care sunt similare în multe aspecte, dar diferă în câteva detalii. Cu Prototype-ul, puteți clona un obiect existent și să modificați doar acele detalii diferite, evitând astfel duplicarea codului și menținând coerența și integritatea sistemului. În plus, Prototype-ul nu este doar o unealtă pentru dezvoltatorii experimentați, ci și o soluție ingenioasă pentru echipele care lucrează în mod colaborativ. Echipa poate defini un set comun de prototipuri și poate crea și adapta noi instanțe în funcție de nevoile proiectului, fără a afecta structura generală a codului. Astfel, colaborarea devine mai ușoară și mai eficientă, iar dezvoltarea devine mai agilă și mai adaptabilă la schimbări.

În categoria paternurilor structurale, găsim soluții pentru structurarea și compunerea obiectelor și claselor într-o aplicație. Printre acestea se numără Adapter-ul, care permite obiectelor cu interfețe incompatibile să lucreze împreună. Pattern-ul Adapter este ca un traducător priceput care permite două entități incompatibile să comunice între ele în mod transparent și eficient. Adapter-ul funcționează ca o componentă intermediară între două interfețe incompatibile. Poate fi utilizat atunci când aveți două componente sau biblioteci software care folosesc interfețe diferite și trebuie să le integrați într-un sistem comun. În loc să rescrieți codul pentru a face cele două interfețe să se potrivească, puteți crea un adaptor care să convertească cerințele unei interfețe în formatul acceptabil pentru cealaltă interfață. Acest lucru permite integrarea componentelor fără a modifica sau perturba codul existent. Un exemplu clar al utilizării pattern-ului Adapter este în domeniul dezvoltării aplicațiilor web. Să presupunem că aveți o aplicație care folosește două framework-uri diferite pentru gestionarea autentificării utilizatorilor: unul pentru autentificarea internă și unul pentru autentificarea externă prin servicii terțe. Aceste framework-uri pot avea interfețe diferite pentru autentificare. Utilizând un adaptor, puteți crea o interfață comună pentru autentificare care să se potrivească cu interfețele fiecărui framework. Astfel, puteți integra ambele metode de autentificare fără a rescrie codul existent. De asemenea, Adapter-ul poate fi utilizat pentru a adapta interfețele existente la noile cerințe ale aplicației. De exemplu, dacă o bibliotecă externă este actualizată și interfața sa este modificată, puteți crea un adaptor care să adapteze interfața veche la noua interfață, fără a modifica codul care folosește biblioteca. În figura 1.5, este afișată structura șablonului Adapter.

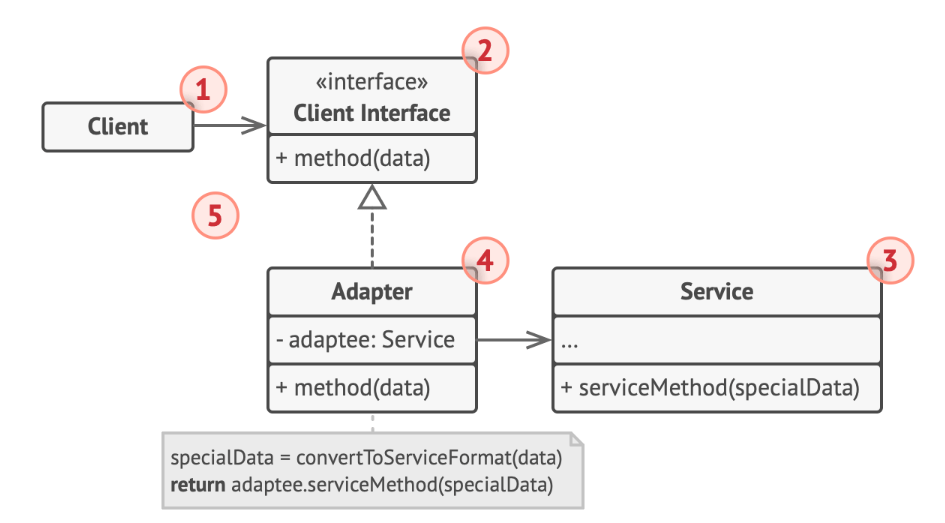


Figura 1.5 – Șablonul de proiectare Adapter

Decorator-ul, permite adăugarea de funcționalități suplimentare unei clase existente, fără a modifica structura acesteia. La o primă vedere, poate părea că Decorator-ul este doar o altă modalitate de a adăuga funcționalitate la un obiect. Cu toate acestea, ceea ce face acest pattern atât de special este capacitatea sa de a permite combinarea și compunerea liberă a funcționalității în timpul rulării programului. Astfel, un obiect poate fi decorat cu mai multe straturi de comportamente suplimentare, fiecare adăugând o anumită funcționalitate fără a afecta structura de bază a obiectului. Un exemplu concret al aplicării pattern-ului Decorator este în dezvoltarea unui sistem de gestionare a fișierelor. În loc să avem clase separate pentru fiecare tip de operațiune pe fișiere, precum citire, scriere sau criptare, putem utiliza Decorator-ul pentru a adăuga aceste funcționalități în mod dinamic. Astfel, un obiect care reprezintă un fișier poate fi decorat succesiv cu funcționalități suplimentare, cum ar fi citirea criptată sau scrierea comprimată, fără a schimba structura inițială a obiectului de bază. Un alt beneficiu al pattern-ului Decorator este că ne permite să evităm supraîncărcarea codului cu clase separate pentru fiecare combinație posibilă de funcționalități. În schimb, putem crea clase de decoratori separate, fiecare responsabil pentru adăugarea unei anumite funcționalități. Acest lucru face codul mai modular, mai ușor de înțeles și de întreținut pe termen lung. Pattern-ul Decorator este un instrument puternic în trusa dezvoltatorului, oferind flexibilitate și extensibilitate fără a sacrifica claritatea și simplitatea codului. Prin adăugarea și eliminarea dinamică a comportamentelor, acest pattern ne permite să construim aplicații mai ușor de gestionat și de adaptat la schimbările cerințelor utilizatorilor. În figura 1.6, se observă reprezentarea structurii șablonului Decorator.

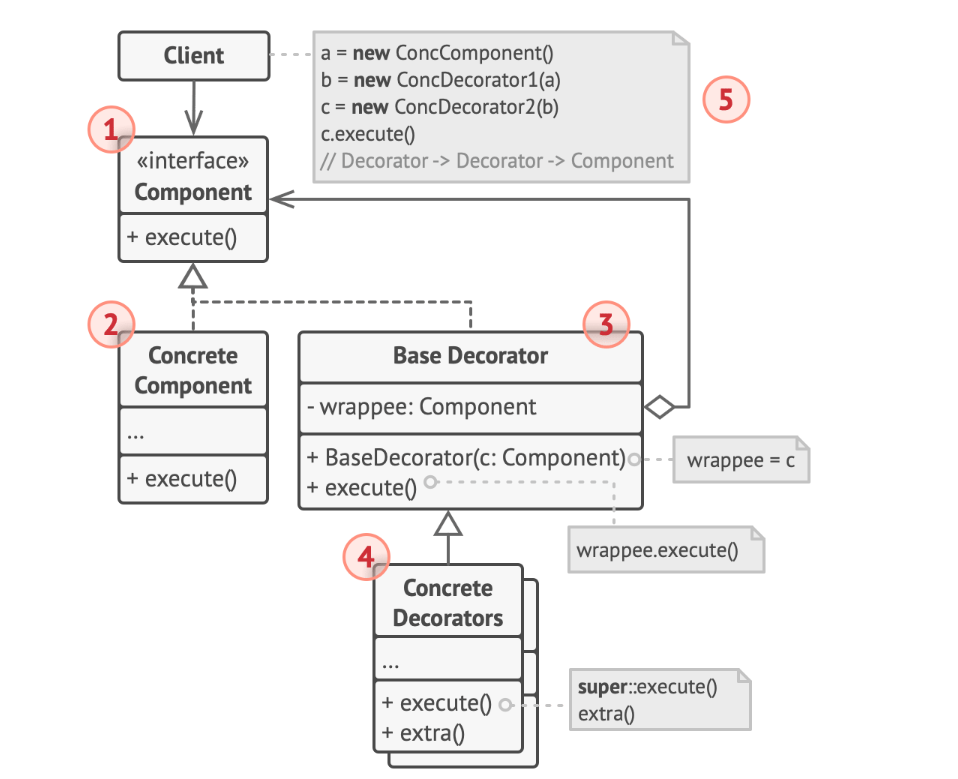


Figura 1.6 – Șablonul de proiectare Decorator

Bridge-ul separă o abstractizare de implementarea sa, astfel încât cele două să poată varia independent, fiind util atunci când există mai multe moduri de a implementa o funcționalitate și când este necesară separarea detaliilor de implementare de abstracțiunea sa. La prima vedere, pattern-ul Bridge poate părea simplu, dar este de fapt o metodă sofisticată pentru a separa diferitele aspecte ale unei aplicații și pentru a le organiza într-o structură coerentă și ușor de întreținut. Ideea fundamentală din spatele acestui pattern este să separăm abstractizarea de implementare, adică să izolăm interfața publică de detaliile de implementare a unei funcționalități. Într-un context practic, putem exemplifica acest pattern în dezvoltarea unui sistem de desenare grafică. Avem două entități majore: formele geometrice și modulul de desenare. Utilizând pattern-ul Bridge, putem defini o interfață pentru formele geometrice și o interfață separată pentru modulul de desenare. Aceste două interfețe sunt conectate printr-un "bridge", care permite formelor geometrice să fie desenate utilizând diferite metode de desenare, cum ar fi desenarea pe ecran sau pe hârtie, fără a modifica codul sursă al formelor geometrice. Beneficiile utilizării pattern-ului Bridge devin evidente într-un proiect de dezvoltare software pe termen lung. Separarea abstractului de implementare permite o mai mare flexibilitate în dezvoltarea și extinderea sistemului. De exemplu, putem adăuga cu ușurință noi forme geometrice sau noi metode de desenare fără a afecta codul existent. De asemenea, Bridge-ul facilitează testarea și depanarea, deoarece ne permite să izolăm și să testăm fiecare componentă în mod independent. În figura 1.8, este reprezentată structura șablonului de proiectare Bridge.

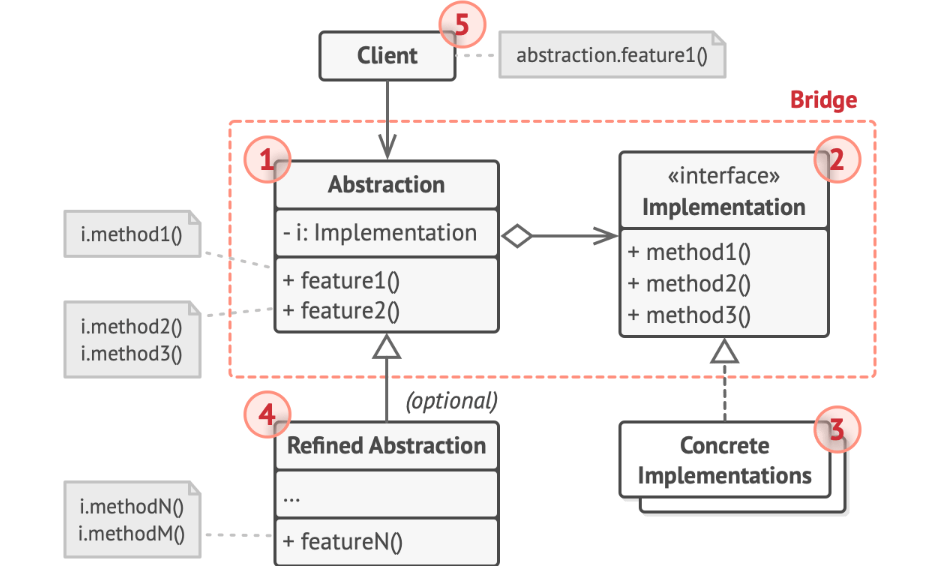


Figura 1.8 – Șablonul de proiectare Bridge

Facade-ul furnizează o interfață simplificată pentru a interacționa cu un sistem complex, ascunzând complexitatea unui subsistem și oferind un punct de acces simplificat. Facade-ul nu este doar o simplă interfață pentru a accesa un sistem mai complex, ci o soluție ingenioasă pentru a izola și a abstractiza componentele interne, facilitând utilizarea și întreținerea sistemului în ansamblu. Să ne imaginăm că dezvoltăm un sistem software mare și complex, care constă din multiple module și sub-sisteme interconectate. Fiecare dintre aceste componente are propria sa logică și funcționalitate, iar interacțiunile între ele pot deveni rapid complicate și dificil de gestionat. Aici intervine pattern-ul Facade, oferind o interfață simplă și unificată către acest sistem complex, ascunzând detaliile interne și complexitatea fiecărui modul sau sub-sistem. Practic, Facade-ul acționează ca un strat intermediar între utilizatorul final și componentele interne ale sistemului. Acest strat intermediar oferă o serie de metode și funcții simple și ușor de înțeles, care permit utilizatorului să interacționeze cu sistemul fără a fi nevoie să înțeleagă detaliile interne sau să gestioneze interacțiunile complexe între module. De exemplu, o interfață de utilizator grafică poate utiliza un Facade pentru a comunica cu mai multe module din spatele scenei, fără a fi necesar ca utilizatorul să fie conștient de toate aceste interacțiuni. Beneficiile pattern-ului Facade sunt numeroase. În primul rând, abstractizarea și izolarea componentelor interne fac sistemul mai ușor de utilizat și de înțeles pentru dezvoltatori și pentru utilizatorii finali. Acest lucru facilitează dezvoltarea și întreținerea sistemului, deoarece modificările în logica internă pot fi făcute fără a afecta interfața cu utilizatorul sau alte module externe. În plus, Facade-ul poate servi ca un punct centralizat de control și gestionare a întregului sistem, permițând implementarea unor politici de securitate sau de acces mai stricte. În figura 1.10, avem reprezentat șablonul de proiectare Facade.

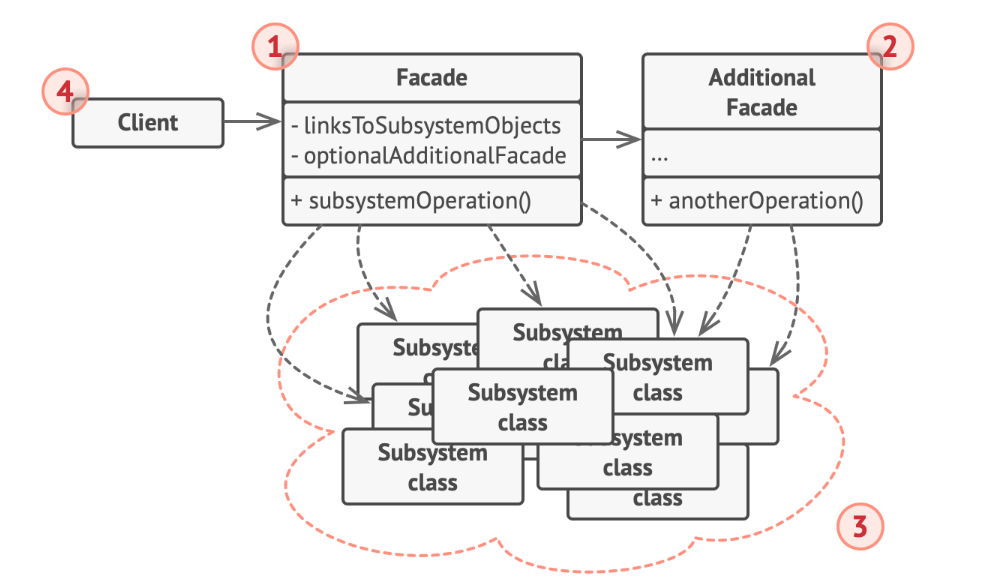


Figura 1.10 – Șablonul de proiectare Facade

Flyweight-ul reduce costul memoriei prin partajarea cât mai mult cât se poate de mult a stării între obiecte, fiind util atunci când trebuie să se gestioneze un număr mare de obiecte și să se minimizeze consumul de memorie. La baza pattern-ului Flyweight stă ideea de a separa atributele intrinseci ale unui obiect de cele extrinseci, astfel încât obiectele comune să poată fi partajate între mai multe instanțe. De exemplu, într-o aplicație de editare grafică, culorile și texturile pot fi considerate atribute intrinseci, în timp ce poziția și dimensiunea obiectului pot fi considerate atribute extrinseci. Prin partajarea culorilor și texturilor între mai multe forme, aplicația poate economisi memorie și timp de procesare. Un alt aspect important al pattern-ului Flyweight este utilizarea unei fabrici de obiecte pentru a gestiona obiectele partajate. Fabrica poate menține un registru al obiectelor deja create și poate returna o referință către obiectele existente în loc să creeze mereu noi instanțe. Această abordare reduce redundanța și crește eficiența în utilizarea resurselor. În plus, pattern-ul Flyweight poate fi implementat în mai multe moduri, în funcție de necesitățile specifice ale aplicației. De exemplu, într-un editor de text, caracterele comune precum literele și cifrele pot fi partajate folosind un Flyweight, în timp ce atributele specifice cum ar fi culoarea și dimensiunea pot fi păstrate ca atribute extrinseci. Flyweight este o unealtă valoroasă în cadrul dezvoltării software, oferind un mecanism eficient de gestionare a obiectelor partajate și reducând astfel costurile și complexitatea aplicațiilor. Prin separarea atributelor intrinseci de cele extrinseci și utilizarea unei fabrici de obiecte, acest pattern poate îmbunătăți semnificativ performanța și scalabilitatea aplicațiilor în care este implementat. În figura 1.11, se afișează structura șablonului de proiectare Flyweight.

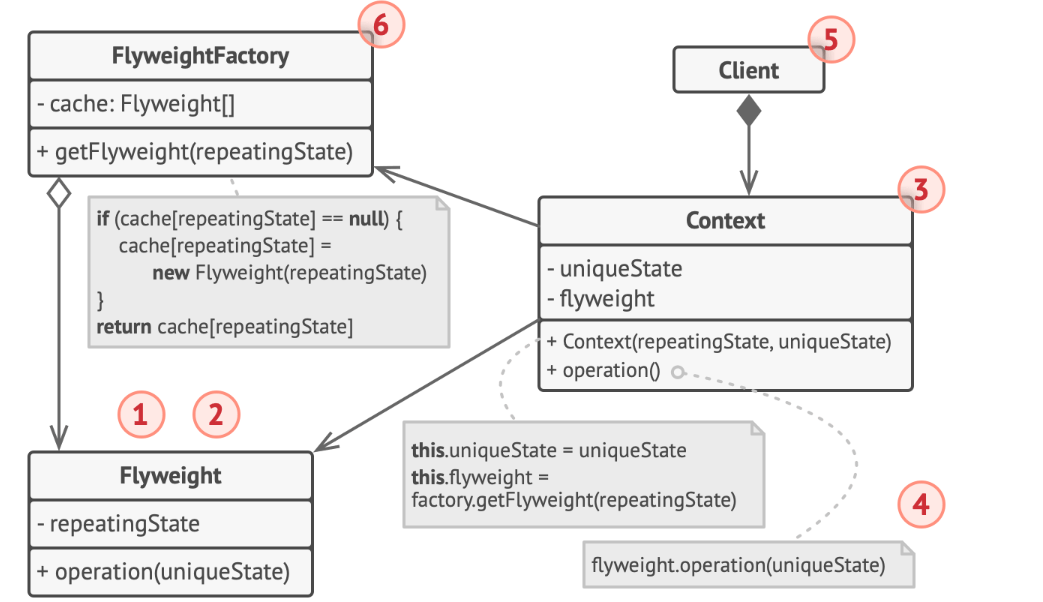


Figura 1.11 – Șablonul de proiectare Flyweight

În categoria paternurilor comportamentale, găsim soluții pentru cooperarea între obiecte și distribuția responsabilităților în cadrul unei aplicații. Sablonul de proiectare Chain of Responsibility este un concept esențial în ingineria software, furnizând o abordare elegantă pentru gestionarea cererilor și a comportamentului într-un mod flexibil și decuplat. Acest șablon permite unor obiecte să prelucreze cereri și să le paseze într-o serie de moduri, formând o "lanț" de responsabilitate. Odată ce o cerere este trimisă către acest lanț, fiecare obiect din acesta decide dacă o poate gestiona sau o poate transmite mai departe către un alt obiect din lanț. Această abordare oferă un grad mare de libertate în configurarea modului în care cererile sunt procesate și în definirea modului în care interacționează diversele componente ale sistemului. Unul dintre principalele avantaje ale acestui șablon este decuplarea completă între obiectul care trimite cererea și cel care o procesează. Acest lucru înseamnă că obiectele care tratează cererile nu sunt conștiente de obiectul care a trimis cererea sau de alte obiecte din lanțul de responsabilitate. Această separare a responsabilităților îmbunătățește coeziunea și modularitatea sistemului, permițând schimbarea și extinderea ușoară a comportamentului fără a afecta alte părți ale aplicației. În plus, Chain of Responsibility promovează principiile deschiderii și închiderii în proiectarea software-ului. Prin adăugarea sau eliminarea de noduri din lanțul de responsabilitate, dezvoltatorii pot extinde sau modifica comportamentul sistemului fără a afecta codul existent. Aceasta facilitează dezvoltarea și întreținerea aplicațiilor pe termen lung, reducând dependența de modificări costisitoare în codul deja scris. Un alt beneficiu al acestui șablon este capacitatea sa de a gestiona cereri complexe sau contextuale într-un mod clar și structurat. Deoarece fiecare obiect din lanțul de responsabilitate este specializat în tratarea unui anumit tip de cerere sau scenariu, acesta poate lua decizii în cunoștință de cauză și poate adapta comportamentul în funcție de necesități. Acest lucru conduce la un cod mai curat și mai ușor de înțeles, deoarece responsabilitățile sunt distribuite în mod clar între diferitele componente ale sistemului. În figura 12, este reprezentată structura șablonului de proiectare Chain of Responsibility.

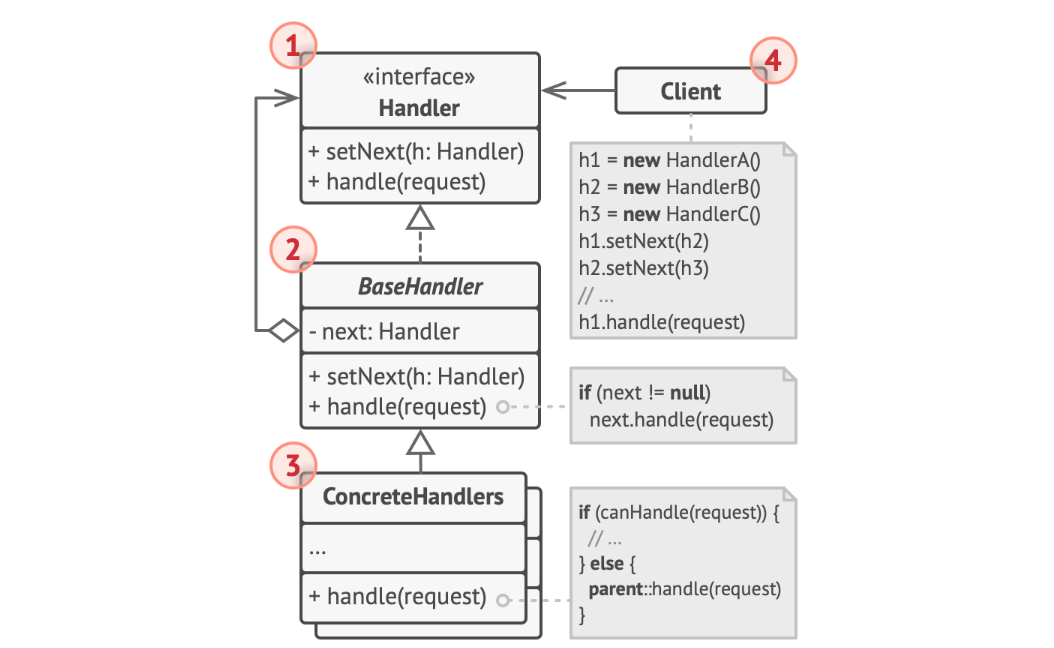


Figura 1.12 – Șablonul de proiectare Chain of Responsibility

Sablonul de proiectare Iterator este un concept esențial în dezvoltarea software-ului orientat pe obiecte. Acesta oferă o metodă de a accesa elementele unei colecții fără a dezvălui structura internă a acesteia. Într-un mod simplu, Iteratorul oferă un mecanism pentru a parcurge secvențial elementele unei colecții, fără a fi nevoie să știm cum sunt stocate sau organizate acele elemente. Un aspect important al sablonului Iterator este abstractizarea procesului de parcurgere a unei colecții. Acest lucru este esențial în dezvoltarea software-ului, deoarece permite o mai mare flexibilitate și modularitate. Prin utilizarea unui Iterator, un client poate parcurge o colecție fără a fi nevoie să cunoască detaliile interne ale implementării acesteia. Astfel, se promovează separarea preocupărilor și se facilitează refolosirea și întreținerea codului. Un alt beneficiu al sablonului Iterator este că permite parcurgerea unei colecții în funcție de nevoile specifice ale aplicației. De exemplu, un Iterator poate oferi o modalitate de a parcurge o colecție în ordine inversă sau într-o anumită ordine definită de utilizator. Aceasta oferă o flexibilitate considerabilă în manipularea datelor și în implementarea algoritmilor. De asemenea, Iteratorul poate fi implementat pentru a lucra cu diferite tipuri de colecții, fără a fi nevoie să modificăm codul clientului. Acest lucru este posibil datorită abstractizării procesului de parcurgere a unei colecții. Astfel, putem folosi același cod de parcurgere pentru diverse structuri de date, cum ar fi liste, arbori sau alte colecții, fără a fi nevoie să rescriem codul clientului. În figura 14, se observă structura șablonului de proiectare Iterator.

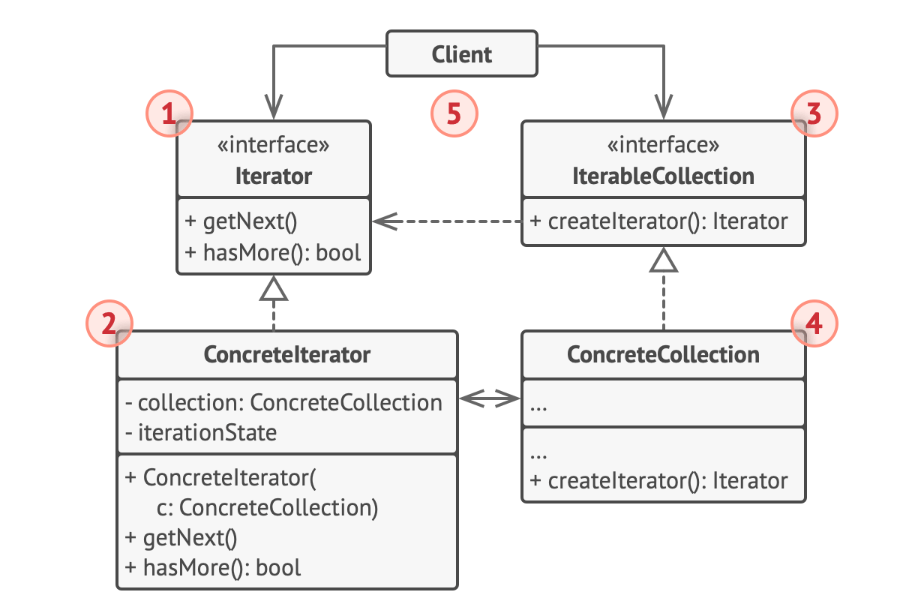


Figura 1.14 – Șablonul de proiectare Iterator

Sablonul de proiectare comportamental, cunoscut sub numele de Mediator, reprezintă una dintre cele mai utile și versatilie unelte în dezvoltarea software. Acesta se încadrează în categoria șabloanelor de proiectare comportamental, concentrându-se pe gestionarea relațiilor dintre obiectele unui sistem și reducerea dependenței acestora între ele. Mediatorul acționează ca un intermediar între diferitele componente ale sistemului, facilitând comunicarea și coordonarea între ele. Unul dintre principalele avantaje ale utilizării sablonului Mediator constă în reducerea cuplării dintre obiecte. Într-un sistem software complex, obiectele pot interacționa direct între ele, ceea ce duce la o dependență strânsă și fragilă între acestea. Prin implementarea unui Mediator, aceste obiecte comunică exclusiv prin intermediul acestuia, reducând astfel dependența directă și izolând modificările la nivelul mediatorului, fără a afecta restul sistemului. Această modularitate și izolare a funcționalității sunt esențiale în menținerea și extinderea unui sistem software complex în mod eficient și sustenabil. Un alt beneficiu major al sablonului Mediator este facilitarea extinderii și modificării sistemului. Datorită faptului că toate interacțiunile dintre obiecte sunt centralizate în Mediator, adăugarea sau schimbarea comportamentului sistemului devine mai ușoară și mai sigură. În loc să modificăm interfețele și logica fiecărui obiect implicat, putem interveni doar în Mediator, fără a afecta restul componentelor. Aceasta reduce riscul de apariție a erorilor și face codul mai ușor de întreținut și de înțeles. Un alt aspect important al sablonului Mediator este promovarea coeziunii și a clarității în cadrul sistemului. Prin centralizarea logicii de mediere într-un singur loc, este mai ușor de înțeles modul în care diferitele componente interacționează între ele. Acest lucru îmbunătățește înțelegerea generală a sistemului și face mai ușoară colaborarea între membrii echipei de dezvoltare. De asemenea, promovează coerența și consistența în cadrul codului, deoarece toate interacțiunile sunt gestionate într-un singur loc, în loc să fie răspândite în întregul sistem. În figura 15, este reprezentată structura șablonului de proiectare Mediator.

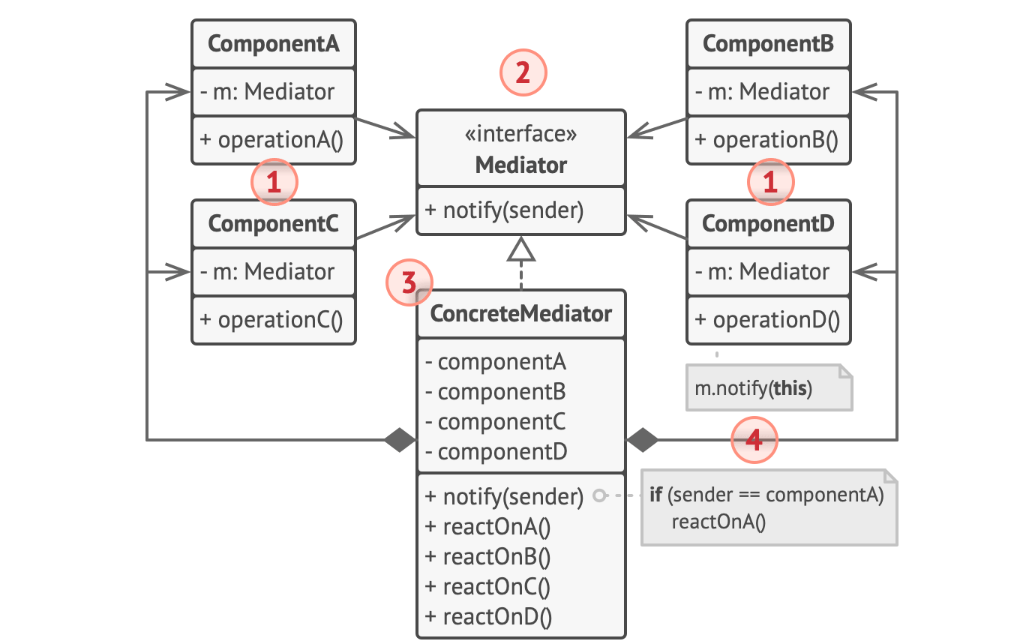


Figura 1.15 – Șablonul de proiectare Mediator

Sablonul de proiectare comportamental Observer este unul dintre cele mai utilizate modele în dezvoltarea software-ului, fiind esențial în crearea unor aplicații robuste și flexibile. Acest șablon este parte a familiei de șabloane de proiectare comportamentală și se concentrează pe gestionarea notificărilor între obiectele care au o relație de dependență. Cu alte cuvinte, Observer permite unor obiecte să observe și să reacționeze la schimbările de stare sau de date ale altor obiecte. Un argument solid în favoarea utilizării sablonului Observer este modularitatea pe care o aduce în proiectarea software-ului. Prin implementarea acestui model, obiectele sunt decuplate și nu sunt legate direct una de cealaltă. În loc să fie nevoie ca un obiect să monitorizeze direct starea altui obiect, acesta poate să se aboneze la notificări de la obiectul observabil. Această separare a responsabilităților face codul mai ușor de întreținut, deoarece modificările la un obiect nu necesită modificări extinse la alte obiecte. Un alt avantaj major al sablonului Observer este extensibilitatea. Prin adăugarea sau eliminarea de observatori, sistemul poate fi ușor adaptat la cerințele noi sau schimbate. Această flexibilitate face ca Observer să fie ideal pentru situațiile în care există nevoi de notificare către mai multe părți interesate în legătură cu schimbările de stare ale unui obiect. Mai mult decât atât, Observer promovează un design orientat spre obiecte în care fiecare componentă își îndeplinește rolul specific. Aceasta contribuie la creșterea coeziunii și a cuplării reduse în aplicații. Componentele sunt concentrate pe sarcinile lor specifice și nu trebuie să fie conștiente de modul în care alte componente își îndeplinesc funcțiile lor, ceea ce duce la o arhitectură mai modulară și mai ușor de înțeles. În figura 17, este afișată structura șablonului de proiectare Observer.

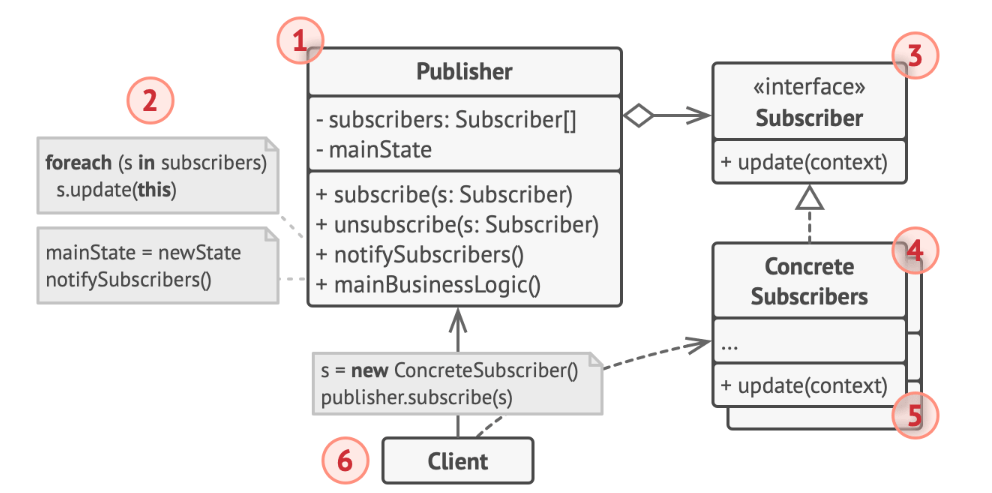


Figura 1.17 – Șablonul de proiectare Observer

Sablonul de proiectare comportamental, cunoscut și sub numele de sablonul de proiectare State, este un concept fundamental în ingineria software, folosit pentru a gestiona starea internă a unui obiect în funcție de schimbările în condițiile externe. Acest sablon oferă o modalitate elegantă de a organiza codul și de a gestiona comportamentul unui obiect în funcție de starea sa internă și a factorilor externi. Una dintre principalele beneficii ale sablonului de proiectare state constă în separarea responsabilităților și modularizarea codului. Prin împărțirea comportamentului obiectului în clase separate pentru fiecare stare posibilă, se facilitează menținerea și extinderea codului în viitor. Acest lucru duce la o arhitectură mai ușor de înțeles, de întreținut și de extins, ceea ce îmbunătățește calitatea și scalabilitatea software-ului. Un alt aspect important al sablonului de proiectare state este modularitatea și extensibilitatea sa. Prin separarea fiecărei stări într-o clasă distinctă, se facilitează adăugarea de noi stări și modificarea comportamentului obiectului fără a afecta restul codului. Acest lucru permite dezvoltatorilor să adauge funcționalități noi sau să modifice comportamentul existent cu ușurință, fără a perturba funcționarea generală a sistemului. Pe lângă beneficiile practice, sablonul de proiectare state promovează și o abordare clară și logică a proiectării software-ului. Prin definirea explicită a stărilor și a tranzițiilor între acestea, se îmbunătățește înțelegerea și comunicarea între membrii echipei de dezvoltare. Acest lucru facilitează colaborarea și reduce riscul de neînțelegeri sau erori în timpul dezvoltării software-ului. Șablonul de proiectare state reprezintă o unealtă valoroasă în cadrul ingineriei software, oferind o modalitate eficientă de gestionare a stării interne a obiectelor și de definire a comportamentului acestora în funcție de condițiile externe. Prin separarea responsabilităților și promovarea modularității și extensibilității, acest sablon contribuie la creșterea calității, scalabilității și ușurinței de întreținere a software-ului. În figura 18, se observă structura șablonului dat.

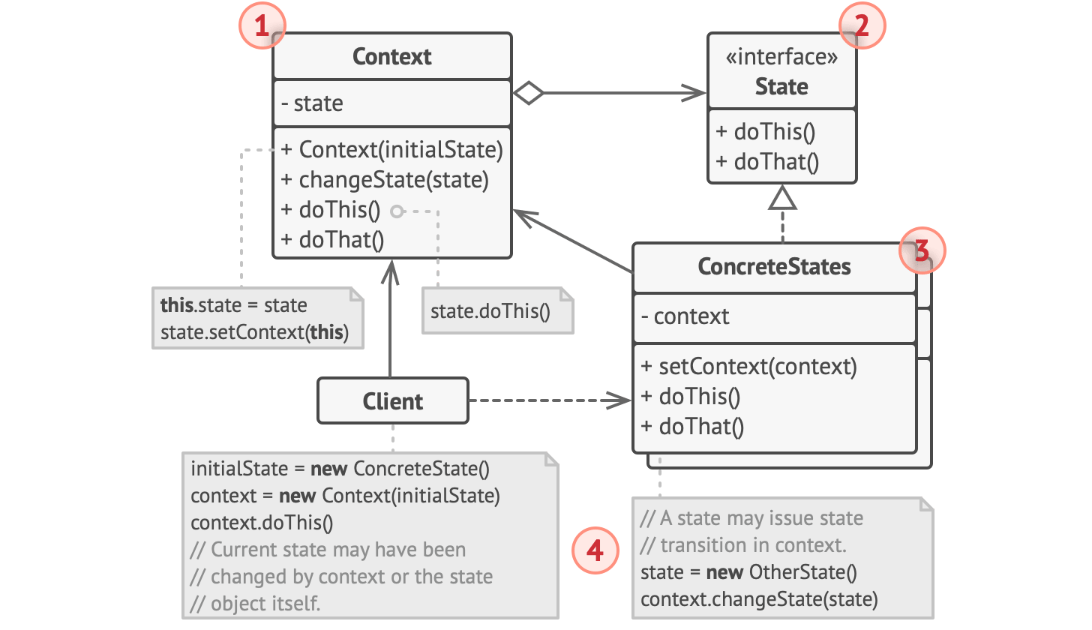


Figura 1.18 – Șablonul de proiectare State

Sablonul de proiectare comportamental, cunoscut și sub numele de Strategy în lumea dezvoltării software, este un concept fundamental în ingineria software, care permite separarea algoritmilor de implementarea lor concretă. Această abordare oferă o modalitate elegantă de a gestiona diverse implementări ale unei acțiuni sau a unui algoritm, permițând astfel flexibilitate, extensibilitate și reutilizare în cadrul unei aplicații. Unul dintre cele mai mari avantaje ale sablonului Strategy este că permite programatorilor să schimbe comportamentul unei clase fără a afecta alte clase care o folosesc. Acest lucru se realizează prin encapsularea diferitelor strategii în clase separate și prin utilizarea unei interfețe comune pentru a interacționa cu aceste strategii. Prin urmare, atunci când se dorește schimbarea comportamentului unei clase, este suficient să se înlocuiască strategia corespunzătoare, fără a fi nevoie să se facă modificări extinse în codul existent. Un alt beneficiu al sablonului Strategy este că încurajează compunerea și modularitatea. Prin definirea fiecărei strategii într-o clasă separată, dezvoltatorii pot construi comportamente complexe prin combinarea sau compunerea diferitelor strategii. Acest lucru face codul mai ușor de înțeles, întreținut și extins pe termen lung. Mai mult decât atât, sablonul Strategy promovează principiul deschis-închis: clasele sunt deschise pentru extindere, dar închise pentru modificare. Acest lucru favorizează dezvoltarea unei baze de cod stabilă și robuste, care poate fi adaptată la cerințele în continuă schimbare ale aplicației. Totodată, utilizarea sablonului Strategy îmbunătățește testabilitatea codului. Prin separarea logicii de business în clase separate, este mai ușor să se scrie teste unitare pentru fiecare strategie în parte, ceea ce facilitează identificarea și remedierea erorilor în timpul dezvoltării. În figura 19, este reprezentată structura șablonului de proiectare Strategy.

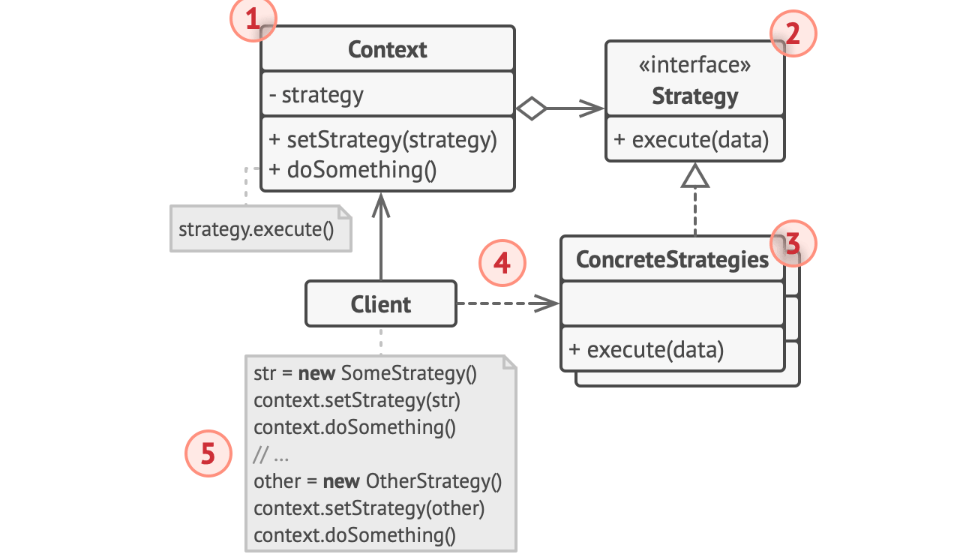


Figura 1.19 – Șablonul de proiectare Strategy

Template Method este un sablon de proiectare comportamental care oferă o metodă cadru pentru definirea algoritmului și permite subclaselor să suprascrie anumite etape ale acestuia fără a modifica structura generală a algoritmului. Este una dintre cele mai utile și mai des utilizate tehnici în programare orientată pe obiecte pentru a promova reutilizarea codului și pentru a facilita extensibilitatea și menținerea codului. Un aspect esențial al sablonului de proiectare Template Method este abilitatea sa de a izola diferitele părți ale unui algoritm în metode separate, permițând astfel implementarea specifică a fiecărei etape fără a afecta întreaga structură a algoritmului. Acest lucru face ca modificările ulterioare ale algoritmului să fie mai ușoare și mai sigure, deoarece se reduce riscul de introducere a erorilor în alte părți ale codului. Un alt beneficiu al sablonului Template Method este că oferă o modalitate clară și simplă de a organiza și de a structura codul. Prin împărțirea algoritmului în mai multe metode, fiecare responsabilă pentru o anumită etapă, se îmbunătățește înțelegerea și întreținerea codului. De asemenea, această abordare promovează principiul împărțirii responsabilităților și separării preocupărilor, ceea ce duce la o arhitectură mai coerentă și mai ușor de gestionat. Pe lângă beneficiile sale în ceea ce privește organizarea și întreținerea codului, sablonul Template Method poate îmbunătăți și flexibilitatea și extensibilitatea unei aplicații. Prin intermediul metodelor suprascrise în subclase, este posibil să se extindă și să se personalizeze comportamentul algoritmului în funcție de nevoile specifice ale aplicației sau cerințele utilizatorului, fără a fi necesară modificarea codului sursă al clasei de bază. Cu toate acestea, există și unele aspecte de luat în considerare atunci când se utilizează sablonul Template Method. De exemplu, supraîncărcarea cu metode potrivite pentru suprascriere poate duce la complexitate și dificultăți în înțelegerea și întreținerea codului. De asemenea, există riscul ca implementările specifice să încalce contractele definite de metoda cadru, ceea ce poate duce la erori și comportamente neașteptate în timpul rulării aplicației. În figura 1.20, este reprezentată structura șablonului dat.

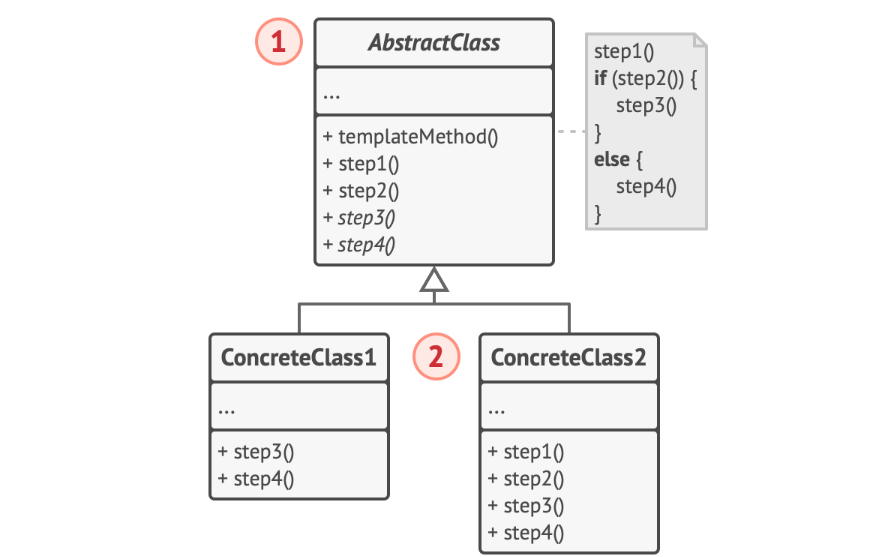


Figura 1.20 – Șablonul de proiectare Template Method

Sablonul de proiectare comportamental Visitor este o metodă puternică și versatilă în programarea orientată pe obiecte, care permite separarea logică a algoritmului de operațiile efectuate pe diverse tipuri de obiecte. Acest sablon este utilizat pentru a defini o nouă operație fără a modifica clasele obiectelor asupra cărora se efectuează operația. În esență, Visitor permite adăugarea de noi funcționalități fără a modifica structura obiectelor existente. Unul dintre cele mai importante aspecte ale sablonului de proiectare Visitor este că încurajează deschiderea claselor pentru extensii viitoare, fără a le modifica. Acest lucru promovează principiile de deschidere/închidere ale programării orientate pe obiecte, unde clasele sunt deschise pentru extensii, dar închise pentru modificări. Prin intermediul Visitor, putem defini noi operații fără a altera codul sursă al claselor existente, ceea ce duce la un cod mai curat și mai ușor de întreținut. O altă caracteristică importantă a sablonului Visitor este că permite separarea operațiilor de structura obiectelor pe care le manipulează. Astfel, putem defini noi funcționalități fără a polua clasele existente cu cod suplimentar. Această separare clară a responsabilităților duce la un design mai modular și mai flexibil al sistemului, facilitând extinderea și întreținerea acestuia pe termen lung. Un exemplu clar de aplicare a sablonului Visitor este în procesarea și analiza unui arbore de obiecte. Fie că este vorba despre un arbore de parsare a unui document XML sau despre o ierarhie complexă de obiecte într-o aplicație software, Visitor poate fi folosit pentru a aplica diverse operații asupra fiecărui nod din arbore, fără a modifica structura acestuia. Deși sablonul Visitor oferă numeroase avantaje, există și anumite inconveniente. Implementarea sa poate fi destul de complexă și poate duce la creșterea complexității codului, în special în aplicațiile cu ierarhii de obiecte extinse. De asemenea, adăugarea de noi funcționalități prin intermediul unor vizitatori noi poate necesita modificări multiple în codul existent. În figura 1.21, este reprezentată structura șablonului Visitor.

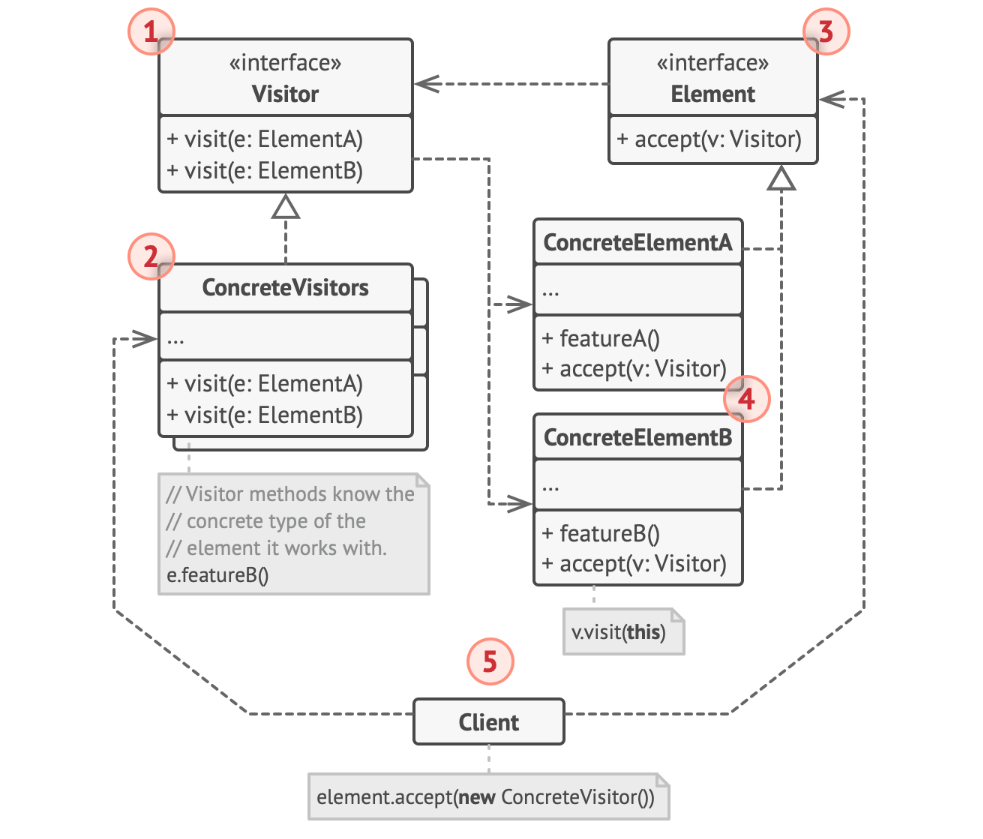


Figura 1.21 – Visitor

Sabloanele de proiectare reprezintă un instrument esențial în cadrul ingineriei software, oferind soluții recunoscute și testate pentru problemele comune în proiectarea și implementarea aplicațiilor. Acestea sunt abordări și strategii recunoscute pentru rezolvarea anumitor probleme de proiectare, încurajând practici de dezvoltare de înaltă calitate și promovând concepte precum reutilizarea codului, modularitatea și extensibilitatea.

# 1.2 Analiza sistemelor deja existente

Un exemplu de site web existent în domeniul închirierii de vehicule este **Turo**[7]. Turo este o platformă online care permite utilizatorilor să închirieze vehicule direct de la proprietarii acestora, oferind o alternativă la serviciile tradiționale de închiriere auto.

Turo oferă o interfață de utilizator curată și intuitivă, cu un proces de rezervare simplu și ușor de navigat. Interfața principală, este reprezentată în figura 1.22.



Figura 1.22 – Platforma de arendă a mașinilor Turo

Utilizatorii pot căuta vehicule în funcție de locație, dată și preferințe de vehicul, iar rezultatele căutării sunt afișate într-un mod clar și organizat. Platforma Turo oferă o gamă variată de vehicule, inclusiv mașini obișnuite, SUV-uri, vehicule sportive de lux și chiar RV-uri și vehicule recreative. Utilizatorii au posibilitatea de a selecta vehiculul care se potrivește cel mai bine nevoilor lor și preferințelor lor de călătorie. Turo oferă tarife competitive și transparente, care pot varia în funcție de modelul vehiculului, locație și durata închirierii. Platforma oferă, de asemenea, diverse opțiuni de asigurare și politici de anulare flexibile, ceea ce oferă utilizatorilor o mai mare libertate și control în procesul de închiriere. Pe lângă închirierea de vehicule, Turo oferă și opțiuni suplimentare, cum ar fi asigurarea suplimentară, servicii de livrare și preluare a vehiculului și chiar posibilitatea de a închiria accesorii, cum ar fi scaune pentru copii sau suporturi pentru biciclete. Platforma utilizează o varietate de canale de marketing și promovare, inclusiv publicitate online, parteneriate cu alte companii din industria călătoriilor și programe de recomandare pentru utilizatori existenți. Compania investește în tehnologie de ultimă generație pentru a asigura securitatea datelor utilizatorilor și a procesului de închiriere. Platforma utilizează sisteme avansate de verificare a identității și de securitate a plăților pentru a preveni fraudele, abuzurile și pentru a asigura securitatea și integritatea clienților.

Turo reprezintă un exemplu de succes în domeniul închirierii de vehicule, oferind o experiență de utilizare de înaltă calitate, o gamă variată de vehicule și opțiuni flexibile de închiriere, toate susținute de o strategie solidă de marketing și de tehnologie avansată.

Un alt exemplu de platformă dedicată închirierii de vehicule este **Getaround**. Getaround oferă utilizatorilor posibilitatea de a închiria mașini direct de la proprietarii acestora, similar cu Turo, aducând în același timp inovații și opțiuni personalizate.

Interfața Getaround este la fel de intuitivă și ușor de utilizat ca cea a lui Turo, permițând utilizatorilor să caute vehicule în funcție de locație, dată și preferințe specifice. Rezultatele căutării sunt prezentate într-un mod organizat, facilitând procesul de selecție a vehiculului potrivit. În figura 1.23, este reprezentată interfața platformei.

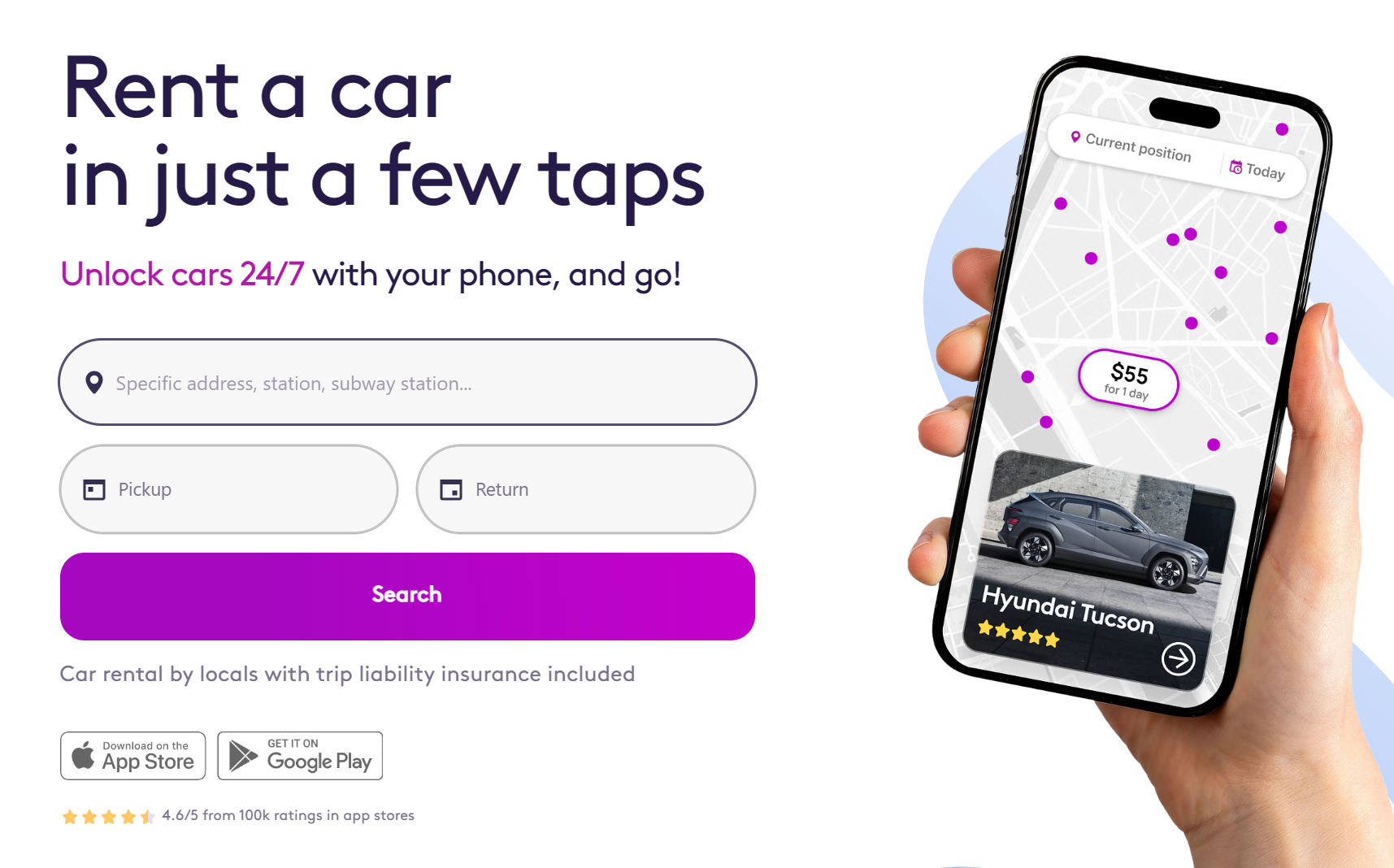


Figura 1.23 – Platforma Getaround

Platforma Getaround oferă o varietate largă de vehicule, inclusiv mașini convenționale, vehicule electrice și chiar biciclete electrice în unele zone. Utilizatorii au posibilitatea de a alege vehiculul care se potrivește cel mai bine necesităților lor și preferințelor lor de călătorie. Tarifele practicate de Getaround sunt competitive și transparente, variind în funcție de modelul vehiculului, locație și durata închirierii, similare cu modelul Turo. De asemenea, platforma oferă opțiuni de asigurare și politici de anulare flexibile, oferind utilizatorilor o mai mare libertate și control în procesul de închiriere. În plus față de închirierea de vehicule, Getaround oferă și alte servicii suplimentare, cum ar fi asigurarea suplimentară, livrarea și preluarea vehiculului, precum și posibilitatea de a închiria accesorii, inclusiv scaune pentru copii sau sisteme de navigație GPS.

Prin oferirea unei game variate de vehicule, tarife competitive și opțiuni flexibile, Getaround reprezintă o alternativă viabilă în domeniul închirierii de vehicule, aducând inovație și comoditate utilizatorilor săi.

# 2. Realizarea sistemului

Platforma de închiriere a vehiculelor este un sistem complex care necesită o abordare atentă și bine planificată pentru a asigura funcționalitatea corespunzătoare și experiența plăcută a utilizatorilor. În acest context, utilizarea tehnologiilor moderne precum ASP.NET Core API pentru dezvoltarea backend-ului și Angular pentru crearea interfeței de utilizator devine o alegere dintre cele mai bune. Aceste tehnologii oferă un cadru puternic și scalabil pentru implementarea funcționalităților complexe ale unei astfel de platforme.

ASP.NET Core API reprezintă o alegere bună pentru dezvoltarea backend-ului unei aplicații web datorită performanței ridicate, securității îmbunătățite și flexibilității pe care o oferă. În cadrul acestei platforme, vom utiliza ASP.NET Core API pentru a gestiona logica de afaceri și comunicarea cu baza de date. Un aspect important al platformei este securitatea autentificării și autorizării utilizatorilor. Vom implementa un sistem de autentificare bazat pe token-uri JWT pentru a autentifica utilizatorii și a le permite accesul la resursele platformei în funcție de rolurile și permisiunile lor. Acest sistem va asigura că doar utilizatorii autorizați au acces la funcționalitățile administrale și că datele utilizatorilor sunt protejate împotriva accesului neautorizat. Pentru administrarea utilizatorilor, se va crea un set de API-uri care permit adminului să efectueze operații precum adăugarea, ștergerea și actualizarea utilizatorilor. Aceste operații vor fi protejate prin autorizare, astfel încât doar adminul să aibă acces la aceste funcționalități. O altă componentă importantă a backend-ului este gestionarea vehiculelor disponibile pentru închiriere. Vor fi dezvoltate API-uri care permit adăugarea, ștergerea și actualizarea vehiculelor în baza de date. Aceste API-uri vor fi accesate atât de către admin, cât și de către utilizatorii obișnuiți pentru a căuta și a închiria vehiculele disponibile. Implementarea unui sistem eficient de gestionare a închirierilor este esențială pentru funcționarea corectă a platformei. Se va dezvolta API-uri care permit utilizatorilor să caute vehicule disponibile pentru închiriere, să facă cereri de închiriere și să gestioneze închirierile existente. Aceste API-uri vor gestiona aspecte precum verificarea disponibilității vehiculelor, calcularea tarifelor de închiriere și actualizarea statusului închirierilor. Pentru a îmbunătăți experiența utilizatorilor, se va implementa un sistem de notificări care să informeze utilizatorii despre evenimente importante precum confirmarea unei cereri de închiriere sau modificările aduse în profilul lor. Aceste notificări vor fi trimise utilizatorilor prin e-mail sau prin notificări în aplicație.

Interfața de utilizator reprezintă puntea de legătură între utilizatori și funcționalitățile oferite de platforma de închiriere a vehiculelor. Este important ca interfața să fie intuitivă, atrăgătoare și ușor de utilizat, pentru a oferi o experiență plăcută utilizatorilor. Prin utilizarea Angular, putem crea o interfață de utilizator modernă și reactivă, care să ofere toate funcționalitățile necesare pentru căutarea, închirierea și gestionarea vehiculelor.

Pagina de autentificare reprezintă primul punct de contact al utilizatorilor cu platforma. Aceasta va conține un formular de autentificare în care utilizatorii își pot introduce credențialele pentru a accesa contul lor. Vom utiliza servicii Angular pentru a gestiona logica de autentificare și pentru a trimite cererile către backend pentru autentificare și obținerea token-urilor JWT. În cazul în care autentificarea este reușită, utilizatorul va fi redirecționat către pagina principală a aplicației. În caz contrar, utilizatorul va primi un mesaj de eroare și va fi încurajat să încerce din nou sau să își creeze un cont nou în cazul în care nu are deja unul. Pagina de administrare este destinată exclusiv adminului și oferă funcționalități pentru gestionarea utilizatorilor, vehiculelor și închirierilor. Această pagină va conține tab-uri sau meniuri pentru accesul la diferitele funcționalități administrative și va utiliza servicii Angular pentru a comunica cu API-urile backend-ului și a realiza operațiile necesare. Adminul va putea să adauge, să șteargă și să actualizeze informațiile despre utilizatori și vehicule, să vizualizeze lista de închirieri și să gestioneze cererile primite. De asemenea, vor exista funcționalități pentru generarea de rapoarte și statistici pentru a oferi o imagine de ansamblu asupra activității platformei. Pagina de căutare a vehiculelor este destinată utilizatorilor obișnuiți și le permite să exploreze vehiculele disponibile pentru închiriere pe baza anumitor criterii precum tipul vehiculului, prețul sau disponibilitatea. Această pagină va conține un formular de căutare și o listă de rezultate care vor fi afișate utilizatorilor în timp real pe măsură ce introduc criteriile de căutare. Se va utiliza servicii Angular pentru a trimite cererile de căutare către backend și pentru a afișa rezultatele în mod dinamic în interfață. Utilizatorii vor putea să filtreze și să sorteze rezultatele pentru a găsi mai ușor vehiculele dorite și vor putea să acceseze pagina de detalii pentru fiecare vehicul în parte pentru a afla mai multe informații. Detalii a vehiculului va oferi utilizatorilor informații extinse despre vehiculele disponibile pentru închiriere, inclusiv imagini, specificații și disponibilitate. Acestea vor fi accesată atât din rezultatele căutării, cât și din alte pagini care afișează vehiculele disponibile.

Vom utiliza servicii Angular pentru a solicita informațiile detaliate despre un vehicul specific către backend și pentru a le afișa în mod dinamic în interfață. Utilizatorii vor putea să vadă imagini cu vehiculul, să citească specificațiile tehnice și să verifice disponibilitatea pentru perioadele dorite de închiriere. Pagina principală va include și procesul închiriere, destinată utilizatorilor care doresc să închirieze un vehicul disponibil. Această pagină va conține un formular de închiriere în care utilizatorii vor putea să selecteze perioada de închiriere. Se va utiliza servicii Angular pentru a valida datele introduse de utilizatori în formular și pentru a trimite cererea de închiriere către backend pentru procesare. Utilizatorii vor primi confirmări în timp real despre statusul cererii lor și vor fi informați despre pașii următori, inclusiv despre procedura de plată și despre modalitățile de preluare a vehiculului închiriat.

Implementarea unei interfețe de utilizator eficiente și atrăgătoare pentru platforma de închiriere a vehiculelor este necesară pentru a oferi o experiență plăcută utilizatorilor și pentru a încuraja utilizarea și fidelitatea. Prin utilizarea Angular și a unor principii moderne de design, putem crea o interfață de utilizator intuitivă și reactivă, care să satisfacă nevoile și așteptările utilizatorilor și să ofere toate funcționalitățile necesare pentru căutarea, închirierea și gestionarea vehiculelor. La fel, interfața este fața platformei, ea este cea care întâmpină utilizatorii. Respectiv, având o interfață atrăgătoare, clientul va fi mai satisfăcut pentru a rămâne pe interfața resăectivă și pentru a utiliza funcțiile ei.

# 2.1 Proiectarea aplicației

Pentru proiectarea aplicației, se va folosi câteva șabloane de proiectare. Dintre șabloanele creaționale, au fost folosite Singleton și Builder. Singleton se va aplica, ca atunci când apare o excepție, ea să se trimită pe email. Șablonul Singleton este afișat în figura 2.1.1.

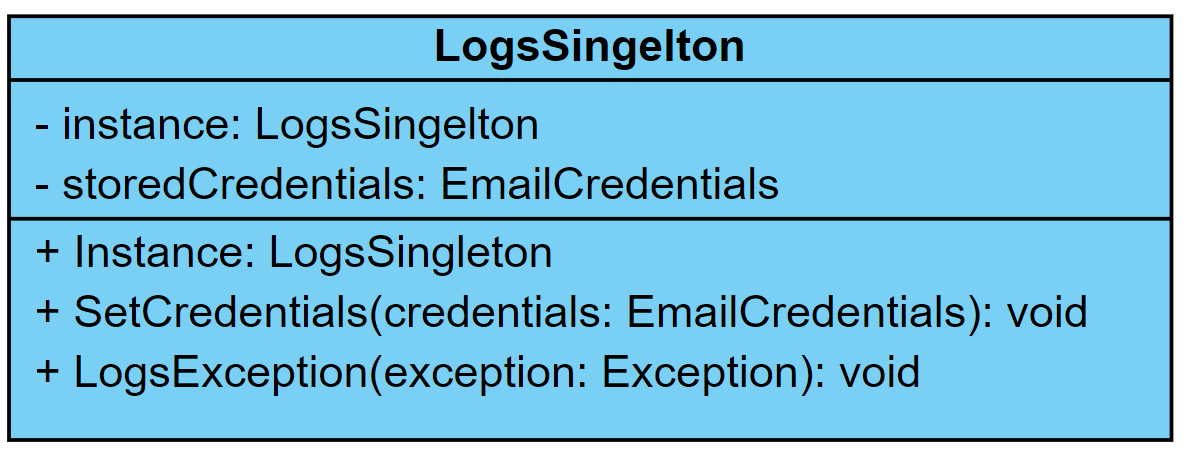


Figura 2.1.1 – Șablonul de proiectare Singleton

Patternul Singleton asigură că o clasă are o singură instanță și furnizează un punct de acces global la aceasta. Instanță asigură că există o singură instanță a clasei. Constructorul privat LogsSingleton() previne crearea instanțelor suplimentare din exteriorul clasei. Metoda Instance oferă acces global la instanța unică a clasei, creând-o dacă nu există deja. Variabila statică storedCredentials și metoda SetCredentials permit stocarea și setarea credentialelor de email. Metoda LogException permite trimiterea unui email cu detaliile excepției, folosind credențialele stocate.

Următorul șablon creaționel este Builder. Diagrama UML al șablonului este afișată în figura 2.1.2.



Figura 2.1.2 – Șablonul creațional Builder

Dintre șabloanele structurale, au fost folosite Facade și Decorator. În figura 2.1.3, este reprezentată diagrama UML al șablonului de proiectare Facade.

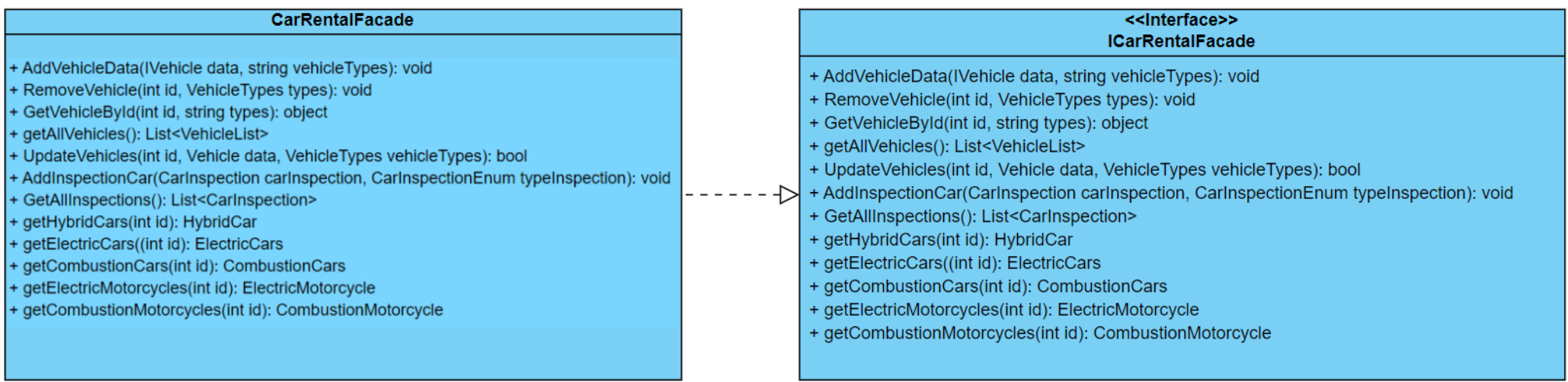


Figura 2.1.3 – Șablonul de proiectare Facade

Acest pattern se bazează pe ideea de a crea o interfață simplificată care maschează detaliile complexe ale subsistemului, facilitând astfel interacțiunea utilizatorilor cu sistemul. Interfața IcarRentalFacade definește un set de metode care permit realizarea operațiunilor esențiale, cum ar fi adăugarea și eliminarea vehiculelor, actualizarea datelor acestora, și gestionarea inspecțiilor. Această interfață servește ca un punct de acces unificat, oferind utilizatorilor posibilitatea de a efectua aceste operațiuni fără a se confrunta cu complexitatea internă a sistemului. Implementarea propriu-zisă a acestei interfețe este realizată de clasa CarRentalFacade. Aceasta clasează toate metodele specificate în interfață și se ocupă de detaliile de implementare, ascunzând astfel utilizatorilor complexitatea internă. De exemplu, adăugarea unui vehicul sau gestionarea inspecțiilor devin operațiuni simple și intuitive prin utilizarea acestei fațade, fără ca utilizatorul să fie nevoit să înțeleagă structura internă a subsistemului.

Următorul șablon de proiectare utilizat este Deecorator. Diagrama lui este afișata în figura 2.1.4

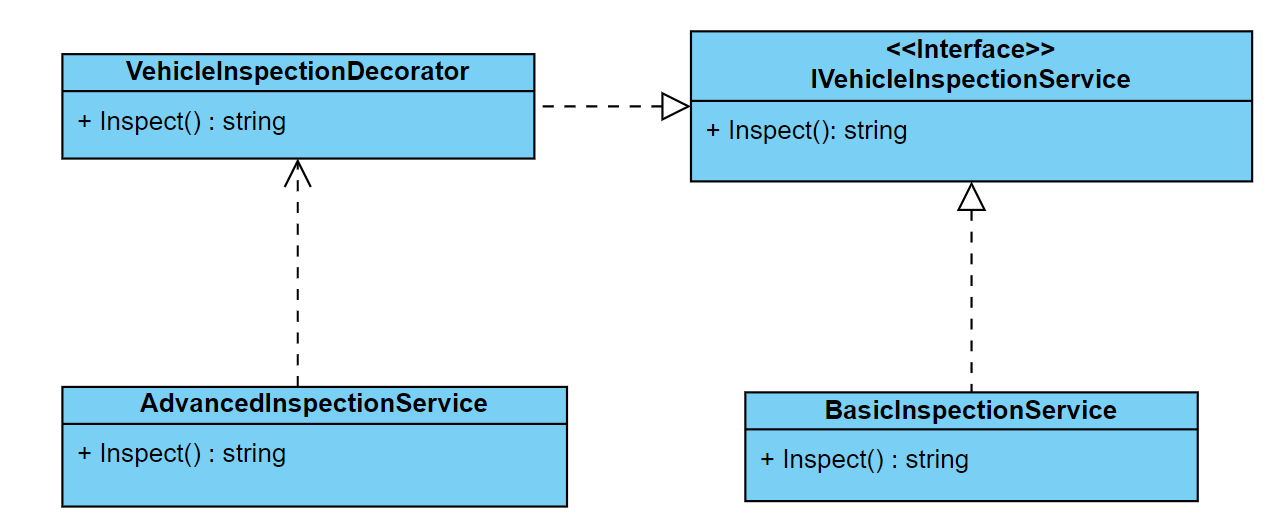


Figura 2.1.4 – Șablonul de proiectare Decorator

Patternul Decorator este folosit pentru a adăuga funcționalitate suplimentară unui obiect într-un mod flexibil și transparent, fără a modifica structura sa de bază. Acesta este realizat prin intermediul unor clase decorator care suprascriu obiectul original, adăugându-i comportamente noi. Interfața IvehicleInspectionService, definește metoda Inspect, care returnează un string. Aceasta este comună pentru toate serviciile de inspecție a vehiculelor. Clasele BasicInspectionService și AdvancedInspectionService sunt implementări concrete ale interfeței IvehicleInspectionService. BasicInspectionService oferă o inspecție de bază a vehiculului. AdvancedInspectionService oferă o inspecție avansată a vehiculului. Clasa VehicleInspectionDecorator, este un decorator abstract care implementează interfața IvehicleInspectionService. Deține o referință către un obiect de tip IVehicleInspectionService, permițând astfel decorarea acestuia. Metoda Inspect() este delegată către obiectul decorat, dar poate fi extinsă de decoratorii concreți pentru a adăuga funcționalitate suplimentară.

Următorul tip de șabloane de proiectare folosite, sunt șabloane comportamentale. Dintre ele, se folosește Mediator și Strategy. În figura 2.1.5, este reprezentată schema UML al șablonului Mediator.



Figura 2.1.5 – Șablonul de proiectare Mediator

Diagrama prezentată ilustrează în mod clar utilizarea patternului Mediator pentru gestionarea eficientă a interacțiunilor dintre diverse cereri și handlerii acestora în sistemul de gestionare a închirierii vehiculelor. Patternul Mediator se bazează pe ideea de a centraliza comunicarea dintre obiecte prin intermediul unui mediator, reducând astfel complexitatea și îmbunătățind modularitatea sistemului. Avem mai multe clase de cereri, cum ar fi:

- GetRentalRequest;

- RentCarRequest;

- GetByIdRequest;

- UpdateVehicleStatusRequest.

Acestea reprezintă diferite tipuri de cereri care pot fi trimise în sistem. Fiecare dintre aceste clase conține datele necesare pentru a efectua cererea respectivă și metodele corespunzătoare. De exemplu, GetRentalRequest gestionează obținerea unei liste de închirieri, în timp ce RentCarRequest se ocupă de procesul de închiriere a unui vehicul. Handler-ul, reprezentat de clasa Handler, este responsabil pentru gestionarea diferitelor cereri. Acesta conține metode pentru a manipula fiecare tip de cerere. Aceste metode sunt apelate de mediator pentru a procesa cererile corespunzătoare, asigurând astfel o separare clară între logica de rutare a cererilor și logica specifică de procesare a acestora.

Următorul șablon utilizat, este Strategy. Schema lui este afișată în figura 2.1.6.

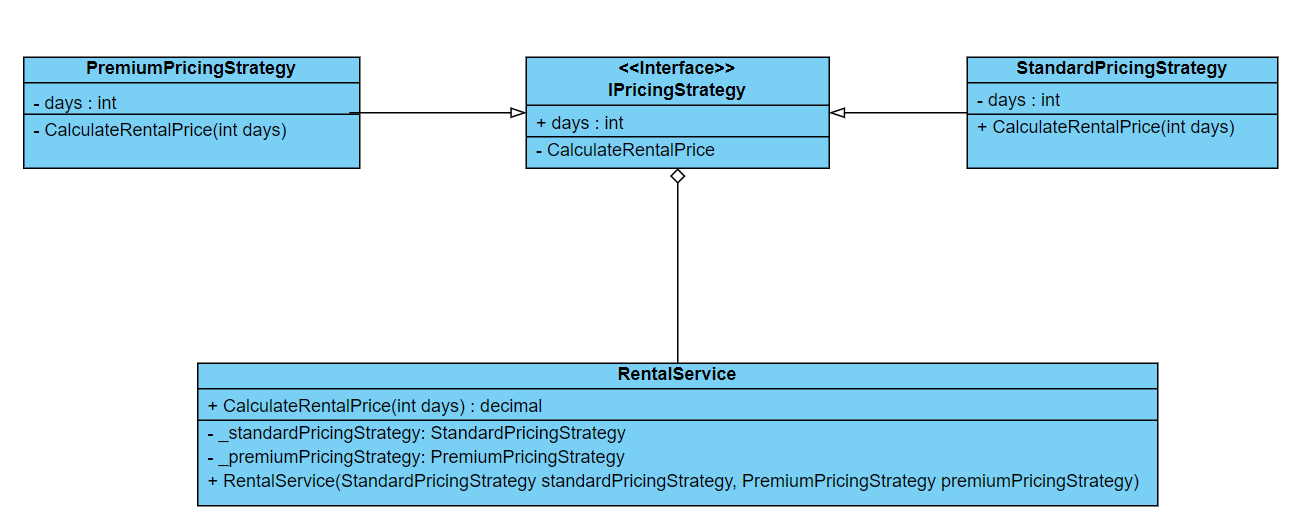


Figura 2.1.6 – Șablonul de proiectare Strategy

Șablonul Strategy este aplicat pentru a calcula prețurile de închiriere a vehiculelor utilizând diferite strategii de preț. Interfața IPricingStrategy definește metoda CalculateRentalPrice(int days), care este utilizată pentru a calcula prețul de închiriere în funcție de numărul de zile. Această interfață este implementată de două clase concrete: StandardPricingStrategy și PremiumPricingStrategy. Clasa StandardPricingStrategy implementează metoda CalculateRentalPrice(int days) pentru a calcula prețul standard de închiriere, în timp ce clasa PremiumPricingStrategy calculează prețul premium de închiriere. Prin separarea acestor algoritmi în clase distincte, sistemul devine mai flexibil și mai ușor de extins. Dacă, de exemplu, ar fi necesară adăugarea unei noi strategii de preț (de exemplu, o strategie de preț de tip discount), acest lucru poate fi realizat prin crearea unei noi clase care implementează interfața IPricingStrategy, fără a afecta codul existent. Clasa RentalService este responsabilă pentru utilizarea acestor strategii de preț. Ea deține referințe către instanțele strategiilor de preț și utilizează aceste strategii pentru a calcula prețul de închiriere prin intermediul metodei CalculateRentalPrice(int days).

# 2.2 Descrierea tehnologiilor pentru sistem

În cadrul dezvoltării aplicației de închiriere a vehiculelor, am folosit o gamă variată de tehnologii pentru a asigura o experiență fluidă, sigură și ușor de utilizat pentru utilizatori. Iată o prezentare succintă a acestor tehnologii. Aceste tehnologii sunt reprezentate în figura 2.2.1



Figura 2.2.1 – Afișare tehnologii utilizate în proiect

**Angular** a fost ales pentru a dezvolta interfața utilizatorului (UI) a aplicației noastre. Acest cadru de lucru JavaScript oferă o structură robustă pentru construirea interfețelor interactive și dinamice, facilitând astfel navigarea și interacțiunea utilizatorilor cu aplicația noastră. C#, un limbaj de programare puternic și versatil, a fost utilizat pentru dezvoltarea logică a aplicației noastre. Cu ajutorul C#, am implementat funcționalități complexe, gestionarea datelor și logica de afaceri din spatele operațiunilor de închiriere a vehiculelor. Pentru gestionarea datelor în aplicația noastră, am integrat Microsoft.EntityFrameworkCore. Acesta este un framework ORM (Object-Relational Mapping) care facilitează lucrul cu baza de date, permițându-ne să manipulăm datele într-un mod eficient și sigur. Acest pachet adițional pentru Entity Framework a jucat un rol crucial în procesul de proiectare a bazei de date și a schemelor de date asociate. Microsoft.EntityFrameworkCoreDesign a furnizat instrumente esențiale pentru modelarea și configurarea entităților noastre în cadrul aplicației. Pentru persistența datelor, am optat pentru Microsoft.EntityFrameworkCore.SqlServer, o extensie care ne-a permis să lucrăm cu baza noastră de date SQL Server. Acest lucru a asigurat o integrare perfectă între aplicația noastră și sistemul de gestionare a bazelor de date.

Pentru a oferi un backend robust și scalabil pentru aplicația noastră, am implementat un API (.NET Core API). Acesta a servit drept intermediar între interfața utilizatorului și baza de date, gestionând cererile HTTP, procesând datele și returnând răspunsurile corespunzătoare. Framework-ul .NET 8.0 a fost fundația pe care am construit întreaga aplicație. Cu caracteristicile sale avansate și performanța îmbunătățită, acest framework a contribuit la dezvoltarea unei aplicații rapide, scalabile și sigure pentru închirierea vehiculelor.

# 2.3 Descrierea la nivel de cod pe module

Pentru a prezeta o înțelegere mai clară, asupra șabloanelor de proiectare utilizate, mai jos se va prezenta o detalizare cu exemple de cod, care au fost utilizate în proiect. Pentru aceasta, se va lua mai întâi șablonul Mediator. Mai jos este reprezentat un exemplu de request, care implementeaza interfața IRequest și este primul pas în implementarea mediatorului.

public class RentCarRequest : IRequest<RentalProc>

{

public int CustomerId { get; set; }

public string CarNumber { get; set; }

public int VehicleId { get; set; }

public string VehicleType { get; set; }

public DateTime StarTime { get; set; }

public DateTime EndTime { get; set; }

public int TotalPrice { get; set; }

public RentCarRequest() { }

public RentCarRequest(int customerId, string carNumber, int vehicleId, string vehicleType, DateTime starTime, DateTime endTime, int totalPrice)

{

CustomerId = customerId;

CarNumber = carNumber;

VehicleId = vehicleId;

VehicleType = vehicleType;

StarTime = starTime;

EndTime = endTime;

TotalPrice = totalPrice;

}

}

Se observă că, pentru acest request sunt nevoie de careva proprietăți care ulterior vor fi folosite pentru apel. Acest request este folosit de către un Handler, care deja gestionează în continuare. Handler-ul respectiv este afișat mai jos.

public class CreateRentalHandler : IRequestHandler<RentCarRequest, RentalProc>{

private readonly IVehicleRepository \_carRepository;

public CreateRentalHandler(IVehicleRepository vehicleRepository)

{

\_carRepository = vehicleRepository;

}

public async Task<RentalProc> Handle(RentCarRequest request, CancellationToken cancellationToken)

{

var data = new RentalProc();

data.CarNumber = request.CarNumber;

data.VehicleId = request.VehicleId;

data.VehicleType = request.VehicleType;

data.CustomerId = request.CustomerId;

data.StarTime = request.StarTime;

data.EndTime = request.EndTime;

data.TotalPrice = request.TotalPrice;

\_carRepository.UpdateClientRentals(request.CustomerId);

\_carRepository.AddRentalRegistration(data);

return data;}}

Acest cod definește un handler pentru cereri de închiriere de mașini. Prin metoda Handle, primește o cerere de închiriere și o procesează, creând o înregistrare de închiriere. Acesta actualizează și înregistrările clientului și adaugă înregistrarea de închiriere în repository-ul de vehicule (IVehicleRepository). Apoi, returnează înregistrarea de închiriere creată.

private readonly IMediator \_mediator;

public MediatrController(IMediator mediator)

{

\_mediator = mediator;

}

public async Task<IActionResult> CreateRental([FromBody] RentModel dataRental)

{

RentCarRequest dataRent = new RentCarRequest();

var res = \_mediator.Send(dataRent); }

Acest fragment de cod definesc un controller numit MediatrController care utilizează biblioteca MediatR pentru a gestiona cererile. Constructorul acestui controller primește o dependență de tipul IMediator, care este utilizată pentru a trimite cererile către mediator. Metoda CreateRental este un endpoint care primește un obiect JSON prin intermediul corpului cererii HTTP și îl utilizează pentru a crea un obiect de tip RentCarRequest. Apoi, acest obiect este trimis la mediator folosind \_mediator.Send(dataRent) pentru a fi procesat.

Următorul șablon este Strategy, care implementează procesul de acordare a prețului total pentru chiria vehicolului.

asda

# 3. Documentarea produsul realizat

Nu poate fi mai mic de o pagina A4, la fel si un subcapitol... Daca nu aveti cu ce completa(desi pueti pune imagini/figuri/ tabele/scheme si deja e un capitol )

# Concluzii

Nu poate fi mai mic de o pagina A4!!!

O pagina cu concluzii ce ati facut voi in lucrare si ce obstacole ati intimpinat... Pentru ce sete necsar dezvoltarea si utilizarea sabloanelor

# Bibliografie

1. C# Introduction ; [ Resursă electronică. ] – Regim de acces: <https://www.w3schools.com/cs/cs_intro.php>
2. ASP .NET Core ; [ Resursă electornică.] – Regim de acces: <https://dotnet.microsoft.com/en-us/apps/aspnet>
3. Design Patterns ; [ Resursă electronică. ] – Regim de acces: <https://refactoring.guru/design-patterns/csharp>
4. Angular tutorials ; [ Resursă electronică. ] – Regim de acces: <https://angular.io/tutorial>
5. The SOLID Principles of Object-Oriented Programming Explained in Plain English ; [ Resursă electronică. ] – Regim de acces: <https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/>
6. SQL Server Tutorial ; [ Resursă electronică. ] – Regim de acces: <https://www.sqlservertutorial.net/>
7. Turo ; [ Resursă electronică. ] – Regim de acces: <https://turo.com/>

Cum se scriu referintele?

Skdbjdvfsjbkmbsfv dshbefdj efwjhknbefkd efdjwnhk,mf sdfnjkews fedhnkfsw wsfhnj,g edsnj,sdf fsdebjkfds fsehbjwe weihkneflw efswlkjnief ewfnhkfsdv dsike3n fikfdsn edfwljksdbnv efwbjfds dsjk,mdsvm sdfjnmdv[numarul care ii corespunde din bibliografie]

# Anexa A