Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**RAPORT**

**Proiect de an la disciplina**

**Tehnici și mecanisme de proiectare software**

**Tema: Analiza și modelarea unei aplicații care simulează gestiunea de mașini și conducerea lor**

A efectuat: Chircu Eugen, TI-202

A verificat: lect.univ. Poștaru Andrei

asis. univ. Mihai Gaidău

Chișinău – 2023

**Cuprins**

1. **Introducere.....................................................................................................................3**
2. **Patterns…..............................................…………………….…………….…………...4**
   1. **Creational Design Patterns…..................................................................................5**
   2. **Structural Design Patterns......................................................................................8**
   3. **Behavioural Design Patterns.................................................................................10**
3. **Realizarea proiectului…...…………………………………………………………...13**
   1. **Builder și Prototype……...……………………………………………………….13**
   2. **Iterator și State.......................................................................................................16**
   3. **Adapter și Proxy.....................................................................................................20**
4. **Concluzie………………………………………………………...……………………25**
5. **Introducere**

Industria auto este una dintre cele mai dinamice și inovatoare domenii ale economiei, având un impact semnificativ asupra vieții noastre și societății în ansamblu. Cu fiecare an ce trece, se dezvoltă tehnologii noi și se aduc îmbunătățiri continue în designul și performanțele mașinilor. În cadrul acestui context în continuă schimbare, proiectul dat se concentrează pe implementarea unui constructor de mașini, de 2 tipuri, Ford și Henessey, împreună cu o funcționalitate de calcul a valorii totale a mașinilor și gestionarea acestora.

Obiectivul principal al proiectului dat este de a dezvolta o aplicație software care să ofere utilizatorilor posibilitatea de a personaliza și construi mașini în funcție de variantele disponibile. Pe lângă funcționalitatea de construire a mașinilor, aplicația noastră va include și un sistem de calcul al valorii totale a mașinilor create de către utilizatori. Acest aspect este esențial în procesul de gestionare și evaluare a mașinilor, deoarece utilizatorii vor putea să estimeze costul final al mașinilor create.

Deasemena va fi o simulare simplă a conducerii unei mașini. Conducerea mașinii va fi pe baza la două stări în care se poate afla mașina. Starea Fixed semnifică că mașina este în stare funcțională și poate să conducă. Starea Broken semnifică că mașina este în stare stricată și conducerea nu este posibilă până nu este reparată.

1. **Patterns.**

În lumea programării, dezvoltarea software-ului eficient și modular reprezintă o provocare importantă. Atunci când ne confruntăm cu probleme complexe și nevoi de proiectare, Design Patterns reprezintă abordări reutilizabile și testate pentru a rezolva aceste diverse provocări. Utilizarea Design Patterns este o modalitate eficientă de a crea structuri flexibile și ușor de întreținut în dezvoltarea software-ului.

Design Patterns sunt soluții generice la problemele de proiectare pe care le întâlnim în mod frecvent. Ele reprezintă o colecție de idei, principii și reguli care au fost dezvoltate și rafinate de-a lungul timpului de către experți în domeniul programării. Pattern-urile sunt descrieri abstracte ale unor soluții probate pentru diferite tipuri de probleme de proiectare.

Prin înțelegerea și aplicarea corectă a Pattern-urilor, dezvoltatorii software pot crea cod de calitate, ușor de întreținut și extensibil. Pattern-urile reprezintă o modalitate eficientă de a face față provocărilor comune de proiectare și de a construi software durabil și robust. Prin explorarea și aplicarea Design Patterns, ne îmbunătățim abilitățile de proiectare și contribuim la dezvoltarea abilităților noastre ca programator.

Există mai multe categorii de Design Patterns, cum ar fi Creational Design Patterns de creare, Structural Design Patterns și Behavioural Design Patterns. Fiecare categorie abordează anumite tipuri de probleme și oferă soluții adaptabile la acestea.

**2.1 Creational Design Patterns.**

În ingineria software, modelele de design creațional sunt modele de design care se ocupă de mecanismele de creare a obiectelor, încercând să creeze obiecte într-o manieră potrivită situației. Forma de bază de creare a obiectelor poate duce la probleme de proiectare sau la un plus de complexitate designului. Modelele de design creațional rezolvă această problemă controlând cumva crearea acestui obiect.

Modelele de design creațional sunt compuse din două idei dominante. Una este încapsularea cunoștințelor despre clasele concrete pe care sistemul le folosește. Un altul ascunde modul în care sunt create și combinate instanțe ale acestor clase concrete.

Întrucât ingineria software modernă depinde mai mult de compoziția obiectului decât de moștenirea clasei, accentul se mută de la comportamentele de codare hard către definirea unui set mai mic de comportamente de bază care pot fi compuse în altele mai complexe. Comportamentele de codificare hard sunt inflexibile, deoarece necesită anularea sau reimplementarea întregului lucru pentru a schimba părți ale designului. În plus, codificarea hard nu promovează reutilizarea și face dificilă urmărirea erorilor. Din aceste motive, modelele de creație sunt mai utile decât comportamentele de codare hard. Modelele de creație fac designul să devină mai flexibil. Ele oferă modalități diferite de a elimina referințele explicite din clasele concrete din codul care trebuie să le instanțieze. Cu alte cuvinte, ele creează independență pentru obiecte și clase.

În cadrul acestui proiect vor fi revizuite următoarele pattern-uri:

* Builder - este un model de design creațional care separă construcția unui obiect de reprezentarea acestuia, permițând aceluiași proces de construcție să creeze reprezentări diferite. Este util atunci când doriți să creați obiecte complexe pas cu pas, oferind o abordare flexibilă și organizată a creării obiectelor. Ideea principală din spatele modelului Builder este să încapsuleze logica construcției într-o clasă de constructor separată. Această clasă oferă metode pentru setarea atributelor dorite ale obiectului construit. Fiecare metodă returnează de obicei generatorul însuși, permițând înlănțuirea metodelor și permițând o interfață fluentă. Folosind pattern-ul Builder, putem crea obiecte într-un mod mai lizibil și mai ușor de întreținut, mai ales atunci când avem de-a face cu obiecte complexe care au multe atribute sau configurații opționale. Permite să controlați procesul de construcție, să reutilizăm pașii de construcție și să îmbunătățim claritatea codului. În general, pattern-ul Builder promovează o separare clară între procesul de construcție și reprezentarea finală a obiectului, oferind o soluție flexibilă și extensibilă pentru crearea de obiecte complexe.

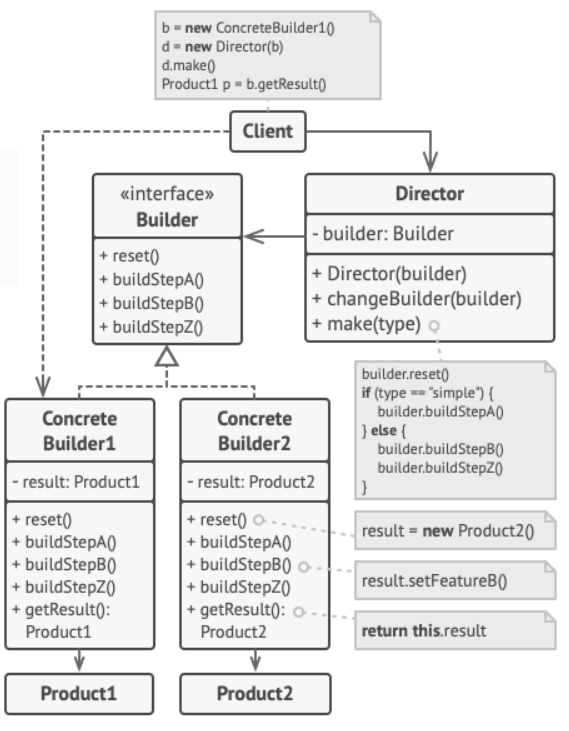


Figura 2.1.1 – Diagrama de clase. Builder

În figura 2.1.1 este prezentat pattern-ul Builder. Observăm interfața Builder cu niște funcții definite. Două clase moștenesc de la acestă interfață, ConcreteBuilder1 și ConcreteBuilder2, aceste două clase prezintă constructorii care ”construiesc” un obiect cu niște parametri predefiniți. ConcreteBuilder1 va construi un obiect Product1 iar ConcreteBuilder2 va construi Product2. Clasa director realizează interfața Builder. Este nevoie de director ca el să ”ordone” builder-ilor când să construiască obiectul și ce fel de obiect, în dependență la care builder se adresează directorul. În clasa Client, deja observăm cum se instanțiază un builder, iar în constructorul unui director se transmite acel concret builder care va crea un obiect de tipul Product1, care ulterior va fi asignat lui *p.*

* Prototype - este un model de design creațional care permite crearea de noi obiecte prin clonarea celor existente. Permite crearea a noi obiecte pe baza prototipului unui obiect existent, evitând astfel necesitatea subclasării sau a specificarii explicite a clasei obiectului creat. Pattern-ul Prototype implică două componente cheie: prototipul și clona. Prototipul este o interfață sau o clasă abstractă care declară metoda de clonare în sine. Acesta servește ca interfață comună pentru toate prototipurile din beton. Această interfață include de obicei o metodă precum clone(). Prototipurile concrete sunt obiectele reale care implementează interfața Prototype sau extind clasa abstractă Prototype. Ei își definesc propria implementare specifică a metodei clone(), care creează o copie sau o clonă a obiectului curent. Pattern-ul Prototype permite creare a noi obiecte prin copierea celor existente, eliminând nevoia de a crea obiecte de la zero sau de a specifica în mod explicit clasele acestora. Este deosebit de util atunci când crearea obiectelor este costisitoare sau complexă și când doriți să izolați logica de creare a obiectelor de codul clientului.

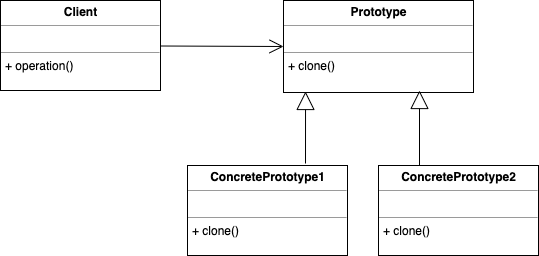


Figura 2.1.2 – Diagrama de clase. Prototype

În figura 2.1.2 este prezentat pattern-ul Prototype. Este prezentă o clasă abstractă cu o singură funcție, clone(), care returnează un obiect de tipul Prototype. ConcretePrototype1 și ConcretePrototype2 reprezintă clase pentru care, ulterior va fi nevoie implementarea funcției clone(). În interiorul clone(), de obicei se introduce returnarea unui obiect de tipul clasei care implementează metoda. De exemplu, în interiourul metodei clone(), pentru clasa ConcretePrototype1, va fi scris: return new ConcretePrototype1();. Clasa Client realizează prototipul prin instațierea obiectelor de tipurile ConcreteProtype1 și/sau ConcretePrototype2, apelând funcția clone(), cu condiția că este făcut casting în tipul obiectului care dorim să-l clonăm. De exemplu:

ConcretePrototype1 cp1 = new ConcretePrototype1();

ConcretePrototype1 cpClone = cp1.Clone() as ConcretePrototype1;

* 1. **Structural Design Patterns.**

Modelele de proiectare structurală sunt o categorie de modele de proiectare care se concentrează pe organizarea și compunerea claselor și obiectelor pentru a forma structuri mai mari și pentru a oferi soluții pentru construirea de sisteme software flexibile, reutilizabile și care pot fi întreținute. Aceste modele se ocupă de compoziția claselor și a obiectelor și definesc modul în care acestea pot fi combinate pentru a forma structuri mai mari, păstrând în același timp arhitectura generală a sistemului intactă.

Scopul principal al modelelor de proiectare structurală este de a simplifica designul prin identificarea modalităților simple de a realiza relațiile dintre obiecte. Aceste modele abordează probleme comune legate de compoziția obiectului, definirea interfeței și relațiile de clasă. Ele oferă linii directoare pentru proiectarea claselor și a obiectelor care lucrează împreună pentru a forma structuri complexe și pentru a rezolva probleme recurente de proiectare.

În cadrul acestui proiect vom revizui următoarele pattern-uri:

* Adapter - Modelul de proiectare Adapter este un model de proiectare structurală care permite obiectelor cu interfețe incompatibile să lucreze împreună. Acționează ca o punte între două interfețe incompatibile, permițându-le să colaboreze fără a-și modifica codul existent. Modelul Adapter implică trei componente principale: Target: target este interfața cu care codul client se așteaptă să interacționeze. Acesta definește metodele sau operațiile pe care le va invoca codul clientului. Adaptee: Adaptee este clasa sau interfața existentă care trebuie adaptată pentru a funcționa cu interfața țintă. Are o interfață incompatibilă care nu poate fi utilizată direct de codul clientului. Adapter: adapter este clasa care face puntea dintre țintă și destinatar. Implementează interfața țintă și încapsulează intern o instanță a adaptatului. Adaptorul traduce apelurile de la interfața țintă în apelurile corespunzătoare de pe destinatar, permițându-le să lucreze împreună fără probleme. Pattern-ul Adapter permite codului clientului să funcționeze fără probleme cu adaptee, fără a fi nevoie să modifice codul clientului sau codul adaptee. Promovează reutilizarea codului, deoarece clasele sau componentele existente pot fi adaptate și reutilizate în contexte diferite.

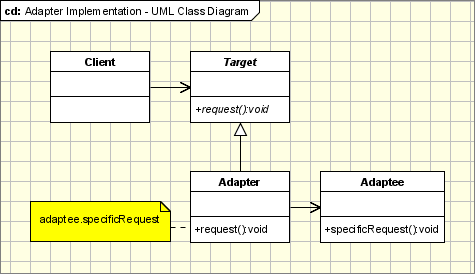


Figura 2.2.1 – Diagrama de clase. Adapter

În figura 2.1.3 este reprezentat pattern-ul Adapter. Target reprezintă interfața care dorim ca Adapterul să o implementeze. Adaptee este clasa care nu o putem realiza făra Adapter. Adapter este puntea care ne permite să realizăm Adaptee în interioul Adapter-ului.

* Proxy - este un model de design structural care oferă un substitut sau un substituent pentru alt obiect pentru a-i controla accesul. Acționează ca intermediar între codul clientului și obiectul real, permițând proxy-ului să controleze și să gestioneze interacțiunile cu obiectul. Modelul Proxy implică trei componente principale: Subjectt: Subject este interfața sau clasa abstractă pe care o implementează atât proxy-ul, cât și obiectul real. Acesta definește operațiunile comune pe care codul client le poate invoca pe obiectul real. Real subject: real subject este obiectul real pe care proxy-ul îl reprezintă și la care oferă acces. Implementează interfața subiect și reprezintă obiectul cu care codul client dorește să interacționeze. Proxy: proxy-ul este un obiect care se înfășoară în jurul subiectului real și oferă o interfață similară cu subiectul. Delegă solicitările de la codul client către subiectul real, adăugând funcționalități suplimentare sau controlând accesul după cum este necesar.

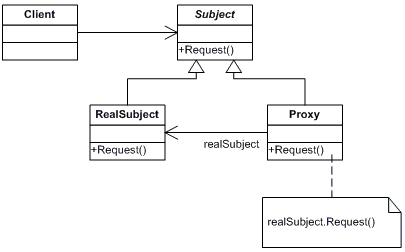


Figura 2.2.2 – Diagrama de clase. Proxy

Subject este o interfață de la care moștenesc ambii, RealSubject și Proxy. Proxy realizează RealSubject-ul, doar că până la realizarea merge o verificare de protecție în cadrul căreia se decide dacă se permite de realizat funcționalul lui RealSubject sau nu.

* 1. **Behavioural Design Patterns.**

Modelele de design comportamental sunt o categorie de modele de design care se concentrează pe interacțiunea și comunicarea dintre obiecte. Aceste tipare abordează responsabilitățile și comportamentul obiectelor, oferind soluții pentru gestionarea comportamentelor și interacțiunilor complexe în cadrul unui sistem software. Modelele de design comportamental captează modele comune de comunicare între obiecte și definesc modul în care acestea pot colabora și interacționa pentru a atinge un obiectiv specific. Acestea urmăresc să îmbunătățească flexibilitatea, extensibilitatea și mentenabilitatea sistemelor software prin promovarea cuplării libere și reutilizarea obiectelor.

În cadrul acestui proiect vom revizui următoarele pattern-uri:

* Modelul de proiectare Iterator este un model de design comportamental care oferă o modalitate de a accesa secvențial elementele unui obiect agregat fără a expune structura de bază. Acesta încapsulează logica pentru parcurgerea unei colecții de obiecte, permițând clienților să itereze peste colecție și să acceseze elementele acesteia într-un mod standardizat și controlat. Modelul Iterator conține trei componente principale: Iterator. Iterator este o interfață sau o clasă abstractă care definește metodele de parcurgere a unei colecții. De obicei, include metode precum hasNext() pentru a verifica dacă există mai multe elemente și next() pentru a prelua următorul element din iterație. Concrete Iterator. Concrete Iterator implementează interfața iteratorului și oferă implementarea specifică a logicii de traversare pentru o anumită colecție. Ține evidența poziției curente în colecție și implementează metodele definite în interfața iteratorului. Agregate: Agrege reprezintă colecția de obiecte pe care le traversează iteratorul. Acesta oferă o metodă de a crea un obiect iterator care corespunde colecției. Această metodă returnează o instanță a iteratorului concret care este adecvată pentru colecția specifică.

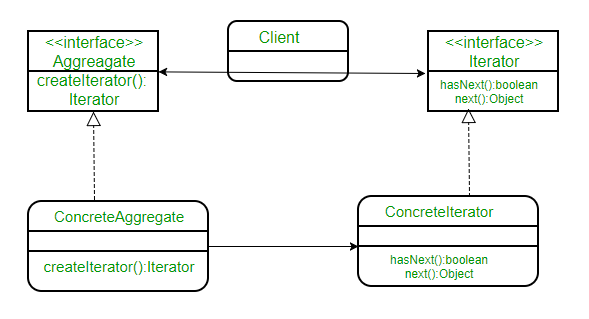


Figura 2.3.1 – Diagrama de clase. Iterator.

Interfața Iterator conține funcțiile necesare pentru a face iterare prin colecția de obiecte, ConcreteIterator implementează acele funcții după cum este necesar, în dependență de tipul de obiecte prin care se face iterarea. Interfața Aggregate conține metoda care răspunde de crearea iteratorului în colecția de obiecte, care presupune clasa ConcreteAggregate.

* State - Modelul State este un model de design comportamental care permite unui obiect să-și modifice comportamentul atunci când starea sa internă se schimbă. Acesta încapsulează fiecare stare ca o clasă separată, iar comportamentul obiectului variază în funcție de starea sa curentă. Modelul promovează conceptul de a avea obiecte care se comportă diferit în funcție de starea lor internă, mai degrabă decât să folosească mai multe instrucțiuni condiționale pentru a determina comportamentul. Pattern-ul State implică mai multe componente cheie: Context. Context este obiectul care menține o stare internă și definește interfața pentru a interacționa cu obiectele de stare. Delegă comportamentul obiectului de stare curentă. State: State este o interfață sau o clasă abstractă care definește metodele comune pe care trebuie să le implementeze clasele de stări concrete. Aceste metode reprezintă diferitele comportamente asociate fiecărei stări. ConcreteState: Clasele de stare concretă implementează interfața de stare. Fiecare clasă de stare concretă reprezintă o stare specifică a obiectului context și asigură implementarea metodelor definite în interfața de stare.

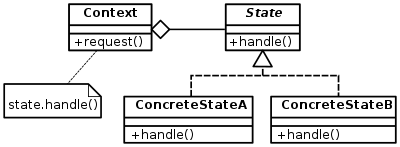


Figura 2.3.2 – Diagrama de clase. State

State reprezintă o interfața sau clasa abstractă care metoda handle() care are scopul de a schimba starea obiectului din starea A în starea B și vice-versa. În ConcreteStateA se implementează comportamentul obiectului când se află în starea A, analogic pentru ConcreteStateB. Context este clasa care comunică cu starea obiectului prin interfața de stare.

1. **Realizarea proiectului.**

**3.1 Builder și Prototype**

În cadrul proiectului dat, clasa de bază în jurul căreia se vor desfășura toate pattern-urile va fi clasa Car, care reprezintă o mașină simplă cu așa câmpuri ca nume, model, data creării, numărul de roți, preț, tipul de motorină.

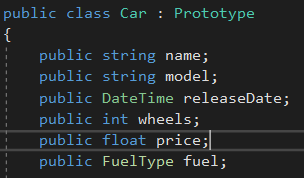


Figura 3.1.1 – Secvență de cod. Clasa Car.

După cum putem vedea deja, clasa Car moștenește clasa abstractă Prototype, care arată în felul următor:

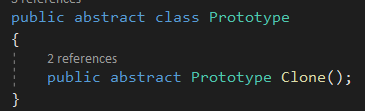


Figura 3.1.2 – Secvență de cod. Clasa Prototype.

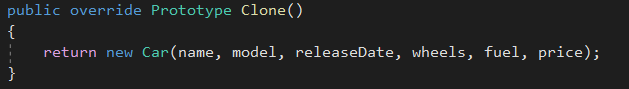


Figura 3.1.3 – Secvență de cod. Implementarea metodei Clone() în clasa Car

În figurile de mai sus este arătat structura clasei Prototype, care conține o singură metodă, Clone(), și implementarea acelei metode în clasa Car. După cum putem vedea metoda Clone() din clasa Car returnează o nouă instanță a clasei Car cu aceeași parametri.

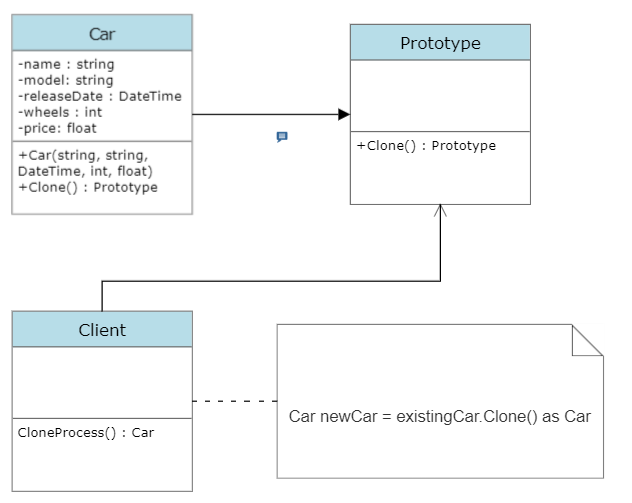


Figura 3.1.4 – Diagrama de clase în proiect. Prototype.

Cum a fost spus anterior. Clasa Prototype conține metoda Clone(), Car moștenește de la Prototype și implementează metoda Clone(), iar Client o realizează, făcând un obiect nou prin clonarea a unuia deja existent.

În cadrul acestui proiect avem două feluri de automobile, fiecare cu parametri diferiți, automobile Ford și automobile Henessey, fiecare diferențiindu-se prin nume, model, număr de roți s.a.m.d.

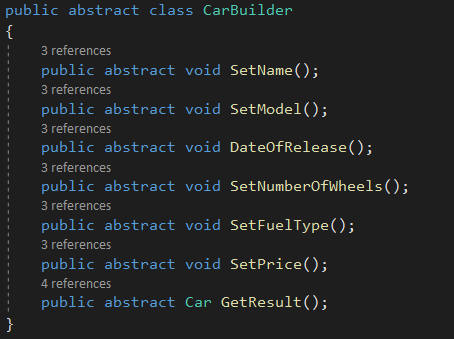


Figura 3.1.5 – Secvență de cod. Clasa CarBuilder

CarBuilder este o clasă abstractă, metodele căreia vor fi implementate de clasele FordBuilder și HenesseyBuilder. În cadrul implementării ambii builderi ”construiesc” mașinile cu parametrii setați din timp, evitând necistatea manual să creăm obiecte cu parametri care se repetă.

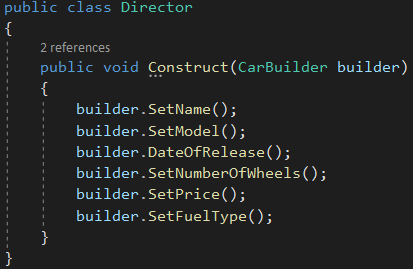


Figura 3.1.6 – Secvență de cod. Clasa Director

Clasa Director are rolul de a ”ordona” unui builder să construiască o mașină după felul în care el știe, în dependență de la ce builder se adresează directorul.

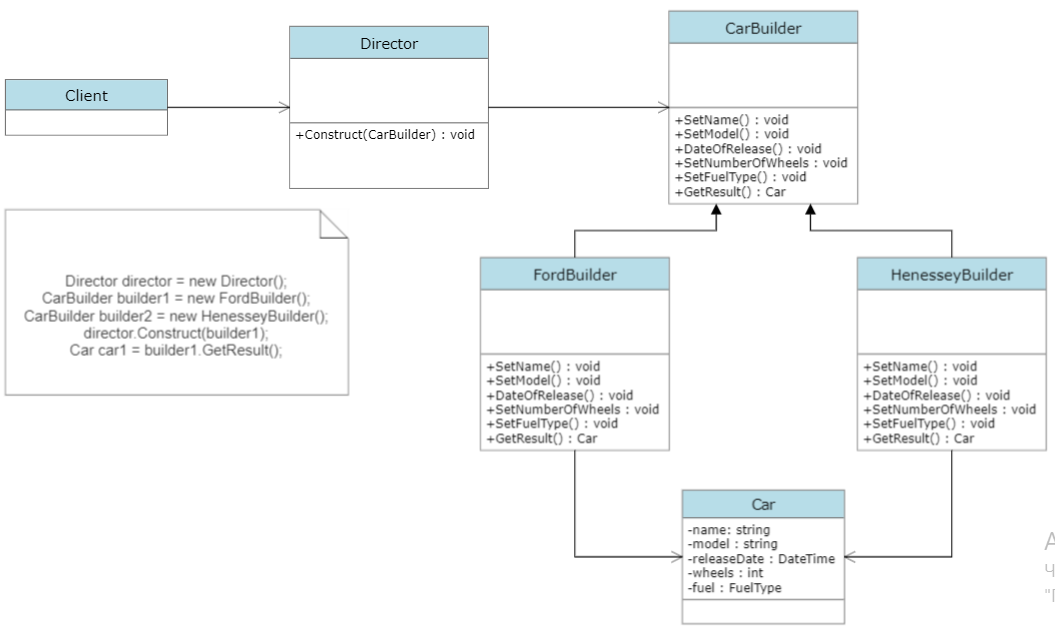


Figura 3.1.7 – Diagrama de clase în proiect. Builder.

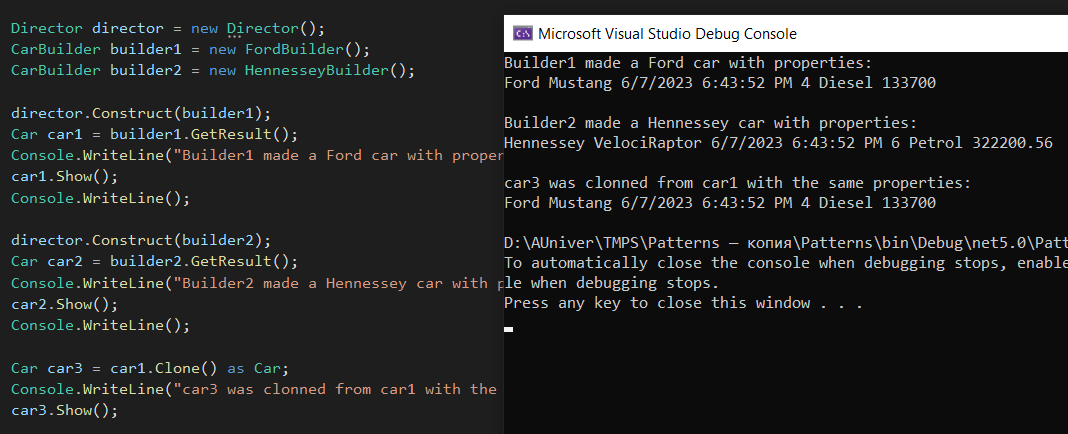


Figura 3.1.8 Utilizarea pattern-urilor Builder și Prototype

* 1. **Iterator și State**

Pentru implementarea pattern-ului Iterator vom avea de o interfață pentru Iterator care va conține metodele necesare, și anume găsirea elementului pe prima poziție, găsirea elementului următor de la care sunt acum și un bool pentru verificarea dacă am trecut prin toate elementele. CarCollection reprezintă colecția de mașini. CarCollection implementează interfața Icollection care conține metoda CreateIterator() care returnează Iteratorul.

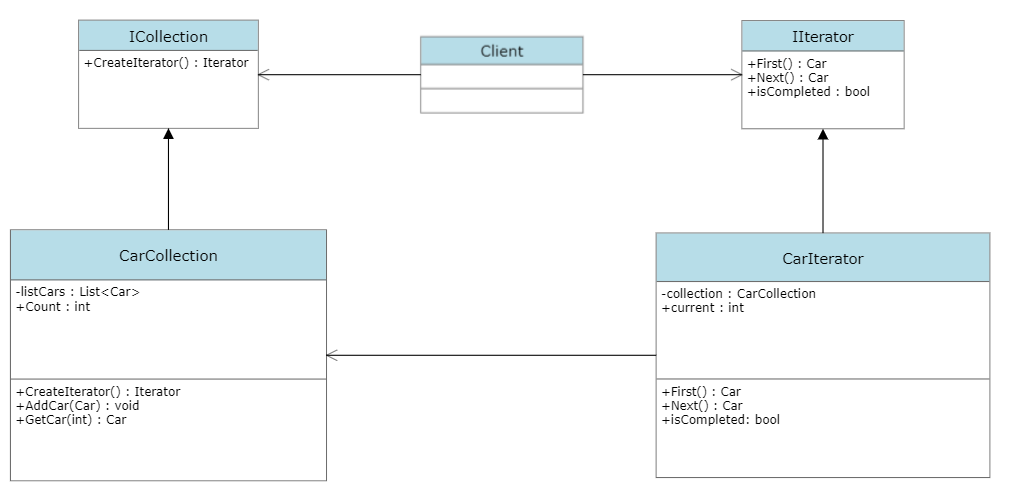
****

Figura 3.2.1 – Diagrama de clase în proiect. Iterator

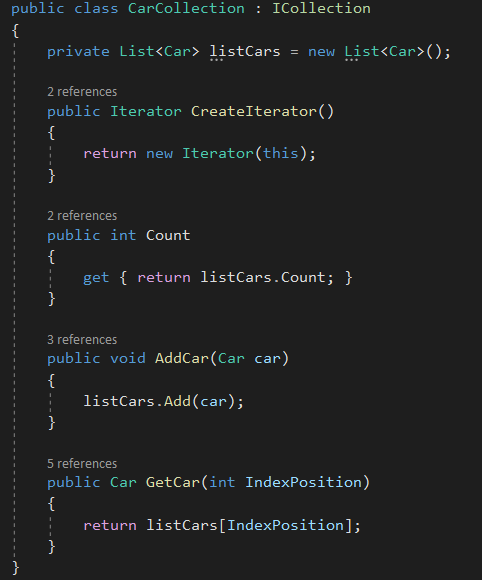


Figura 3.2.2 – Secvență de cod. Clasa CarCollection

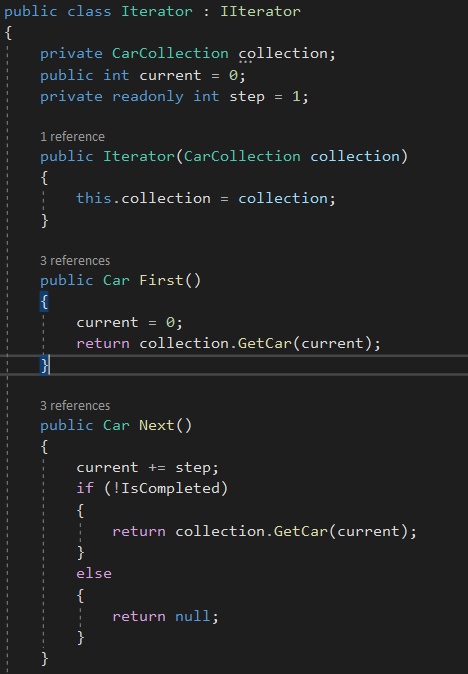


Figura 3.2.3 – Secvență de cod. Clasa Iterator.

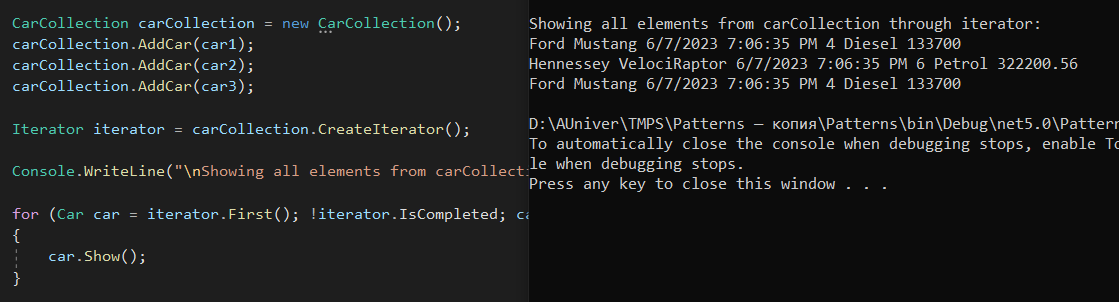


Figura 3.2.4 – Utilizarea pattern-ului Iterator

Pattern-ul State în cadrul proiectului dat răspunde de conducerea automobilului. Dacă automobilul este în starea Fixed, automobilul poate fi condus, dacă se află în stare Broken – automobilul nu poate fi condus până nu va fi reparat. Rolul de context pentru acest pattern îl are loc însăși clasa Car. Mașina se strică dacă este condusă prea multe ori și trece în starea Broken, iar cu ajutorul funcției FixCar() care se află în clasa Car, mașina trece în starea Fixed, cu condiția ca se află în starea Broken.

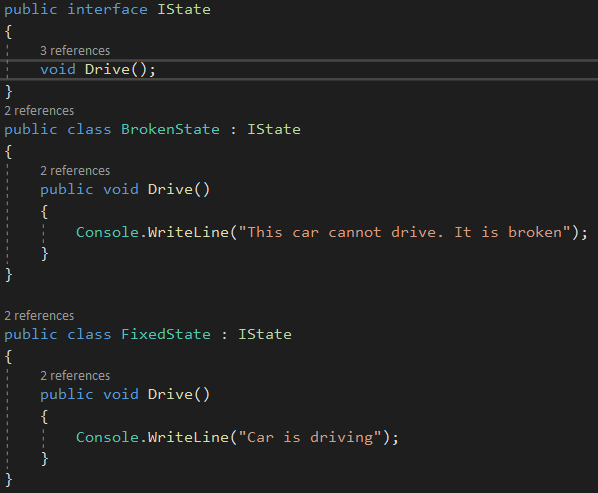


Figura 3.2.5 – Secvență de cod. State-urile unui Car

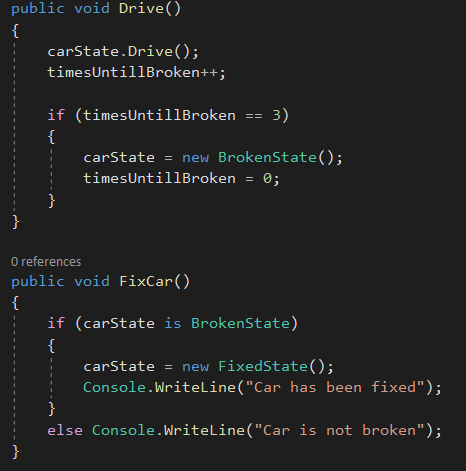


Figura 3.2.6 – Secvență de cod. Contextul stărilor

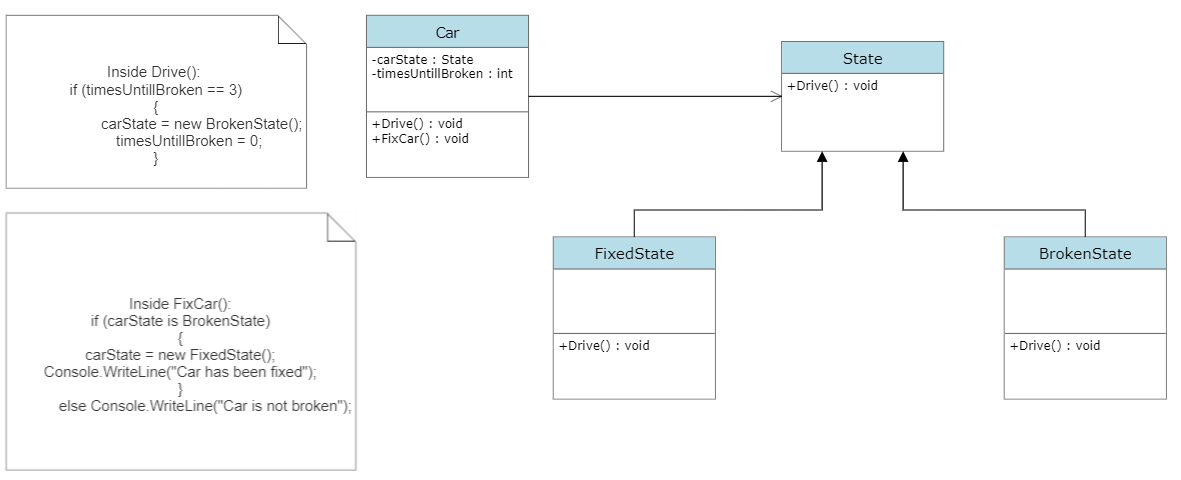


Figura 3.2.7 – Diagrama de clase în proiect. State

* 1. **Adapter și Proxy**

În cadrul acestui proiect, pattern-ul Adapter este folositor când avem o clasă, funcțiile căror nu pot fi apelate din cauza argumentelor pe funcția le primește.

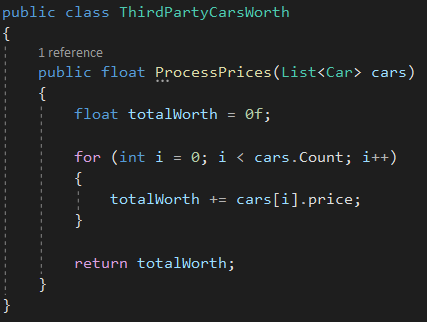


Figura 3.3.1 – Secvență de cod. Clasa ThirdPartyCarsWorth

Clasa ThirdPartyCarsWorth are o funcție ProcessPrices() care primește ca argument o listă de obiecte Car. În cadrul acestui proiect nu se folosește o listă pentru păstrarea obiectelor, ci se folosește obiect de tipul CarCollection, folosit anterior pentru pattern-ul Iterator. Totuși, avem nevoie să apelăm această funcție, de asta avem nevoie de adapter.

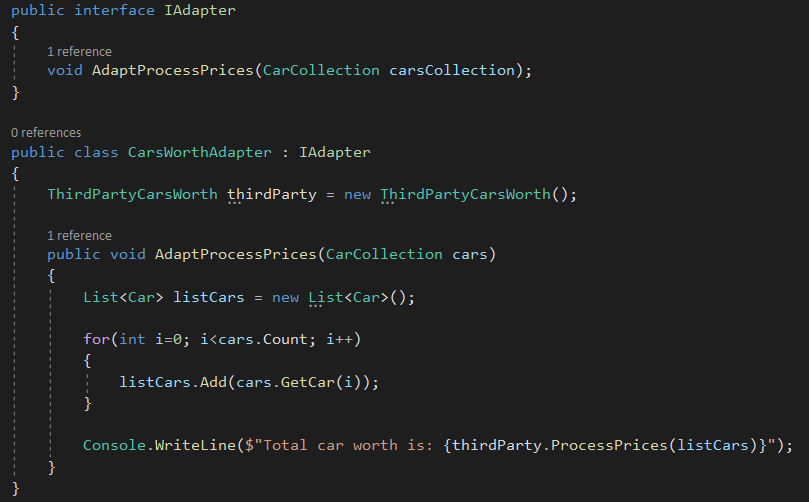


Figura 3.3.2 – Secvență de cod. Interfața Iadapter și Clasa CarsWorthAdapter

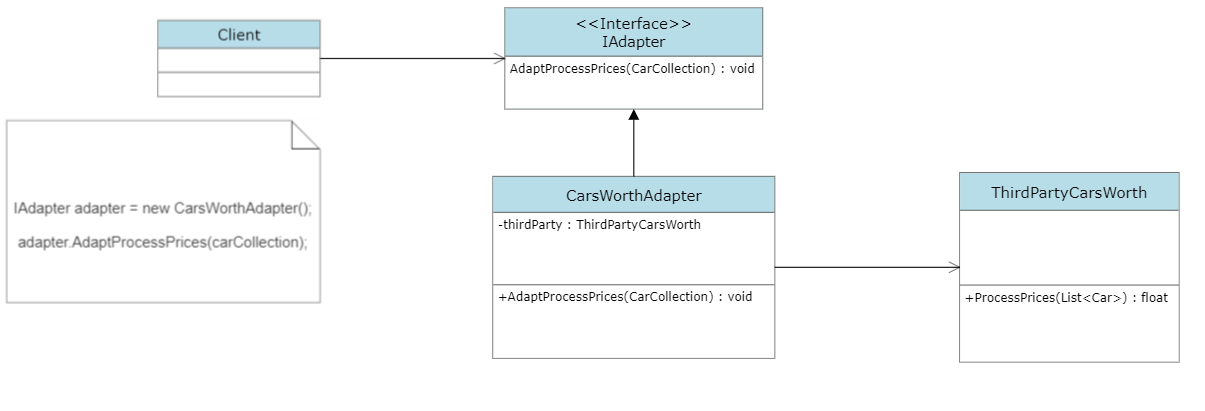


Figura 3.3.3 – Diagrama de clase în proiect. Adapter

După cum vedem din diagramă, CarsWorthAdapter moștenește IAdapter și realizează funcționalul clasei ThirdPartyCarsWorth, Client-ul neavând nevoia să apeleze direct la ThirdPartyCarsWorth.

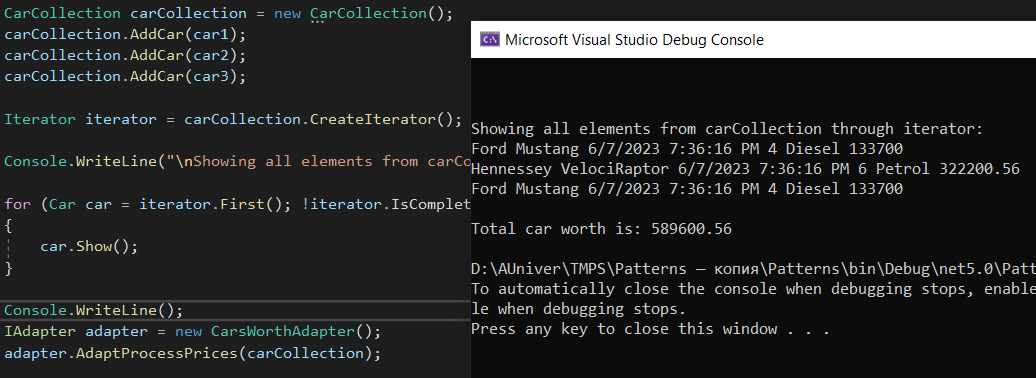


Figura 3.3.4 – Utilizarea pattern-ului Adapter

Pattern-ul Proxy în cadrul acestui proiect este folosit pentru a performa oarecare schimbări asupra mașinilor create, dar acele schimbări le poate performa doar cel al cărui rol permite performarea schimbărilor, aceștia fiind Manager și Director.

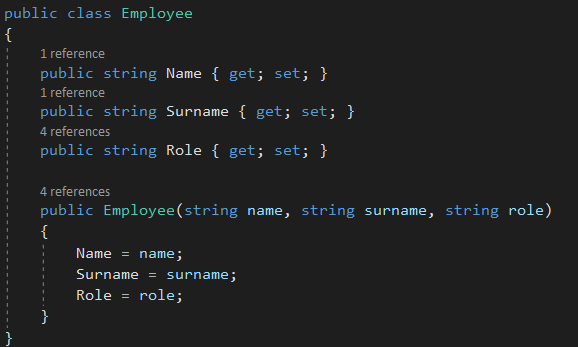


Figura 3.3.5 – Secvență de cod. Clasa Employee

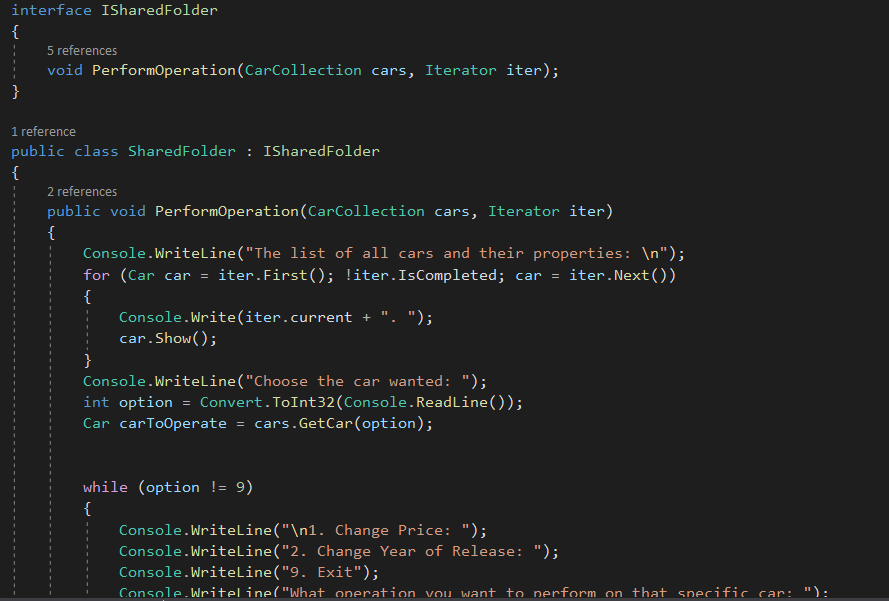


Figura 3.3.6 – Secvență de cod. Interfața IsharedFolder și Clasa SharedFolder

După cum putem vedea, în interiorul clasei SharedFolder nu este nici o referință la vreun Employee și rolul lui, la fel și funcția PerformOperation este implementată fără nici o verficarea a rolului.

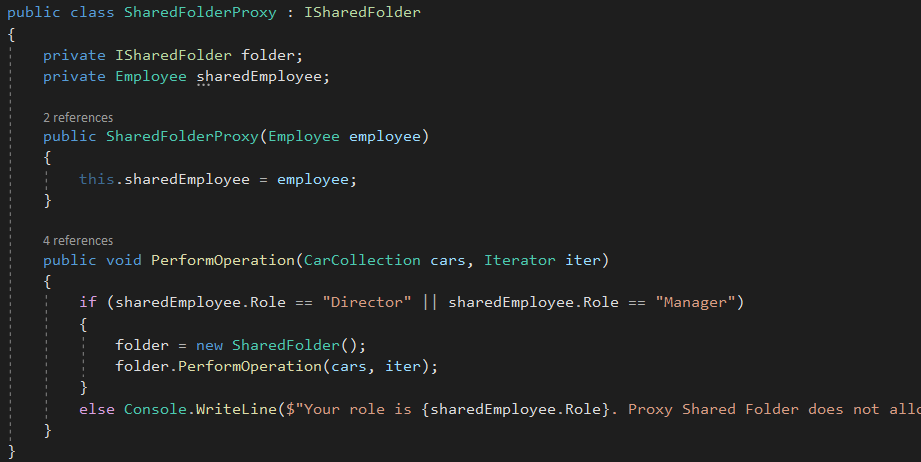


Figura 3.3.7 – Secvență de cod. Clasa SharedFolderProxy

Clasa SharedFolder nu conținea nici o referință la Employee deoarece SharedFolderProxy se ocupă cu verficarea rolului care îl are acel Employee și deja decide dacă funcționalul lui SharedFolder va fi implementat.

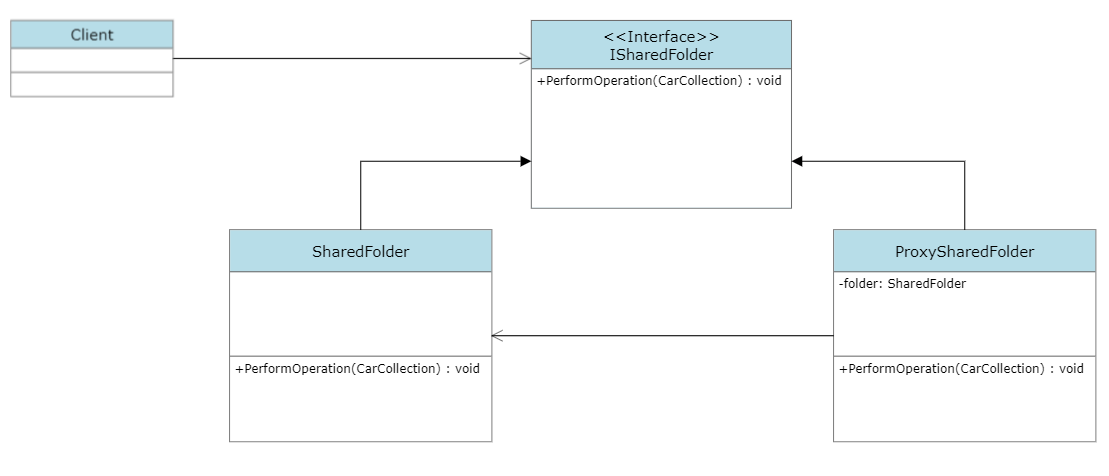


Figura 3.3.8 – Diagrama de clase în proiect. Proxy

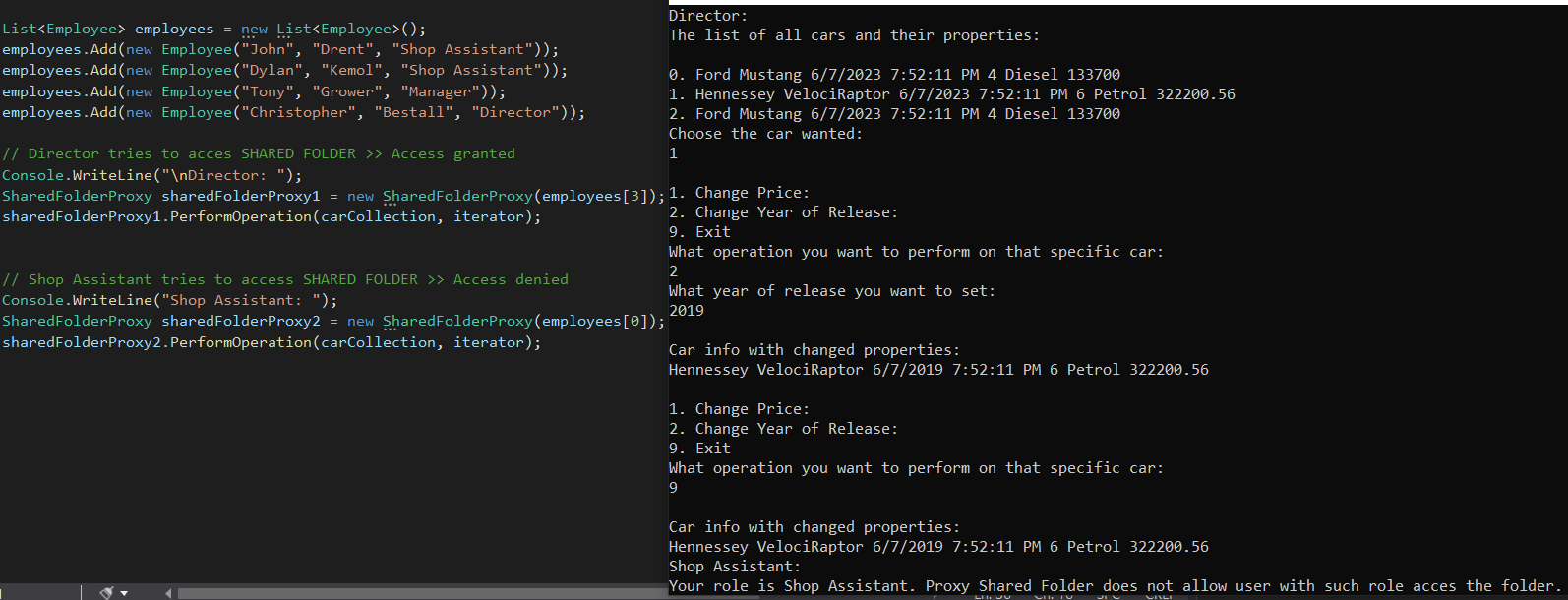


Figura 3.3.9 – Utilizarea pattern-ului Proxy.

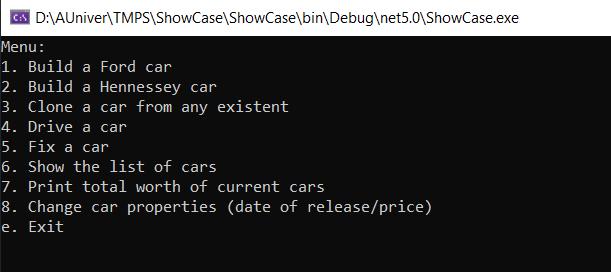


Figura 3.3.10 – Rularea proiectului

**Concluzie.**

În concluzie pot să spun că realizarea proiectului mi-a permis să-mi lărgesc cunoștințele referitoarea la Design Patterns și mi-a permis să înțeleg cu adevărat utilitatea lor în cadrul unui proiect real. Pattern-ul Builder ne-a ajutat să creăm obiecte complexe într-un mod incremental, facilitând astfel construirea și configurarea entităților noastre. Prin intermediul pattern-ului Prototype, am putut crea copii ale obiectelor existente, evitând astfel crearea de obiecte noi costisitoare și accelerând procesul de creare. Adapter-ul a fost extrem de util atunci când am trebuit să facem două interfețe incompatibile să lucreze împreună. Prin intermediul acestui pattern, am putut transforma o interfață existentă într-o altă interfață dorită, fără a modifica codul sursă existent. Pattern-ul Proxy a furnizat o abstracție suplimentară între componente, permițându-ne să controlăm și să gestionăm accesul la acestea. Iterator-ul ne-a ajutat să parcurgem și să accesăm elementele unei colecții într-un mod standardizat și eficient. Acest pattern a facilitat iterația și manipularea elementelor într-un mod simplu și elegant. Pattern-ul State a fost extrem de util atunci când am avut nevoie să schimbăm comportamentul unui obiect în funcție de starea sa internă. Am putut defini și gestiona diferite stări și tranzacții între acestea într-un mod modular și ușor de înțeles.