Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare Informatică și Microelectronică

Departamentul Inginerie Software și Automatică

**Proiect de curs**

Disciplina: Tehnici și Mecanisme de Proiectare Software.

Tema: Proiectarea și implementarea unei agenții de turism.

Efectuat: st. gr. TI-204, Sclearenco Irina

Verificat: asist. universitar, Mihai Gaidau

Chișinău, 2023

**Cuprins**

**Introducere**...................................................................................................................................................3

1. **Analiza domeniului de studiu…………….**.......................................................................................4
   1. Istoria creării design pattern-urilor................................................................................................5
   2. Clasificarea design pattern-urilor...................................................................................................6
   3. Creational design patterns..............................................................................................................7
   4. Structural design patterns...............................................................................................................8
   5. Behavioral design patterns.............................................................................................................9
2. **Proiectarea și implementarea. Noțiuni generale**............................................................................10

2.1. Abstract Factory...................................................................................................................11-13

2.2. Factory Method....................................................................................................................13-15

2.3. Facade..................................................................................................................................15-16

2.4. Bridge...................................................................................................................................16-17

2.5. Chain Of Responsibility.......................................................................................................17-19

2.6. Strategy................................................................................................................................19-21

**3. Concluzie**............................................................................................................................................22

**4. Bibliografie**.........................................................................................................................................23

**5. Anexe**..................................................................................................................................................24

**Introducere**

**Design Pattern: Ce reprezintă și pentru ce e utilizat?**

Design pattern-urile reprezintă soluții generale și reutilizabile ale unei probleme comune în design-ul software. Un design pattern este o descriere a soluției sau un template ce poate fi aplicat pentru rezolvarea problemei, nu o bucata de cod ce poate fi aplicata direct. În general pattern-urile orientate pe obiect arată relațiile și interacțiunile dintre clase sau obiecte, fără a specifica însă forma finală a claselor sau a obiectelor implicate. Acesta este un element esențial în programarea orientată pe obiecte. Este o infrastructură software formată dintr-un număr mic de clase care este utilizată pentru a rezolva o problemă tehnică. Oferă o abordare testată și dovedită pentru rezolvarea eficientă a unui tip specific de problemă. Design Patterns capturează cunoștințele și experiența colectivă a dezvoltatorilor de software în timp, oferind soluții bine-stabilite pentru problemele de design recurente.

Iată câteva motive pentru care se utilizează pattern-urile de design:

Reutilizabilitate: Design Pattern-urile promovează reutilizabilitatea prin furnizarea de soluții testate și dovedite pentru probleme comune. În loc să reinventezi roata de fiecare dată când te confrunți cu o problemă similară, poți apela la un pattern de design existent care a fost deja implementat și verificat.

Eficiență: Design Pattern-urile facilitează dezvoltarea de software eficient prin furnizarea unor structuri și abordări bine definite pentru rezolvarea problemelor. Ele reduc timpul și efortul necesare pentru proiectarea și implementarea unei soluții, deoarece oferă un ghid clar și detaliat pentru a aborda problema respectivă.

Comunicare și înțelegere: Utilizarea design pattern-urilor permite dezvoltatorilor să comunice și să înțeleagă mai ușor între ei. Prin utilizarea unor denumiri standardizate și cunoscute pentru soluții și interacțiuni de software, se facilitează colaborarea în cadrul echipei și se îmbunătățește înțelegerea comună a problemelor și soluțiilor.

Evitarea problemelor cunoscute: Design Pattern-urile sunt soluții recunoscute și testate pentru probleme frecvent întâlnite. Prin utilizarea lor, se evită erorile și problemele comune care pot apărea în proiectare și implementare. Dezvoltatorii pot beneficia de experiența și înțelepciunea acumulată de alți specialiști în domeniu.

În concluzie, utilizarea design attern-urilor aduce beneficii practice și eficiente în proiectarea și dezvoltarea de software. Ele oferă soluții testate și dovedite pentru probleme comune, promovează reutilizabilitatea și facilitează comunicarea și colaborarea între dezvoltatori.

În lucrarea dată, pentru dezvoltarea aplicației agenției de turism, s-a utilizat limbajul de programare C#, au fost aplicate diferite tipuri de șabloane de proiectare, precum cele creaționale, structurale și comportamentale, pentru a crea un software bine structurat, flexibil și ușor de întreținut. Agenția de turism oferă o gamă largă de tururi în toată lumea, spre exemplu Turcia, Grecia, Egipt, respectiv cu prețuri corespunzătoare, de la mic la mare, pe placul și buzunarul fiecăruia din clienții agenției.

1. **Analiza doemniului de studiu**

Utilitatea utilizării unui design pattern este evidentă. Design Pattern-ul poate accelera procesul de dezvoltare. Oferă paradigme de dezvoltare testate, ceea ce ajută la economisirea timpului fără a fi nevoie să reinventezi pattern-uri de fiecare dată când apare o problemă. Deoarece el este creat pentru a rezolva probleme cunoscute, acestea pot fi prevăzute înainte de a deveni vizibile în timpul procesului de implementare.

Pe scurt, design pattern-ul este util în trecerea de la un model de analiză la un model de dezvoltare. Poate fi utilizat în mod concret în mai multe cazuri, în special pentru gestionarea sistemului de salarizare la schimbarea unui salariu și menținerea informațiilor despre modificările implicate.

În lucrarea dată, pentru proiectarea și implementarea temei alese (agenția de turism), s-a utilizat limbajul de programare C#, unde se demonstrează utilizarea mai multor design pattern-uri în aplicație de turism. Agenția de turism oferă clienților posibilitatea de a căuta și rezerva călătorii în mod convenabil și eficient:

Căutare și rezervare ușoară: Prin intermediul aplicației online, clienții pot căuta destinații, tururi și pachete turistice în funcție de preferințe și datele dorite. Aplicația poate oferi un sistem de filtrare avansat, astfel încât clienții să poată selecta criterii specifice, cum ar fi destinația, data călătoriei, tipul de tur și bugetul disponibil. Rezervarea poate fi efectuată rapid și fără a fi necesară interacțiunea directă cu un agent de turism.

Informații detaliate și imagini: Aplicația poate oferi informații detaliate despre fiecare destinație și tur disponibil. Aceste informații pot include descrieri ale atracțiilor turistice, itinerarii, servicii incluse, tipuri de cazare și opțiuni de transport. Imaginile și videoclipurile pot fi utilizate pentru a oferi o perspectivă vizuală asupra destinațiilor și experiențelor oferite.

Recenzii și evaluări ale clienților: Aplicația poate permite clienților să lase recenzii și evaluări pentru călătoriile și serviciile oferite de agenție. Aceste recenzii pot ajuta alți potențiali clienți să-și formeze o opinie și să ia decizii informate atunci când își planifică călătoriile. Recomandările și feedback-ul pozitiv din partea clienților mulțumiți pot consolida încrederea în agenție și în serviciile sale.

Securitate și confidențialitate: Aplicația online poate asigura un mediu sigur și protejat pentru clienți. Datele personale și informațiile de plată pot fi protejate prin utilizarea tehnologiilor de securitate și criptare. Politica de confidențialitate a agenției poate fi clară și transparentă, asigurându-le clienților că datele lor sunt tratate cu grijă și în conformitate cu legile privind protecția datelor.

Suport și asistență online: Prin intermediul aplicației, clienții pot avea acces la un sistem de suport și asistență online. Pot fi integrate funcții de chat live sau mesagerie prin care clienții pot comunica direct cu agenții de turism pentru a primi răspunsuri la întrebări sau asistență în timp real. Acest lucru facilitează comunicarea și rezolvarea rapidă a problemelor sau solicitărilor clienților.

* 1. **Istoria creării design pattern-urilor.**

Design pattern-ul provine din domeniul arhitecturii. Conceptul este în mare parte îndatorat lui Christopher Alexander. Acest concept de design pattern permite crearea aplicațiilor în procesul de proiectare a formelor. Ideea nu a fost adoptată doar în arhitectură, ci și în antropologie, istoria artei și, desigur, în informatică.

Design pattern-urile au apărut ca rezultat al nevoii de a rezolva probleme de design recurente în dezvoltarea software-ului. Acestea au început să fie definite și documentate în anii '90, odată cu publicarea cărții "Design Patterns: Elements of Reusable Object-Oriented Software" scrisă de Erich Gamma, Richard Helm, Ralph Johnson și John Vlissides, cunoscută și sub numele de "Cartea Gang of Four".

Echipa cunoscută drept "Gang of Four" (GoF) a sintetizat experiența lor colectivă în domeniul dezvoltării software și a identificat un set de modele de design care rezolvau probleme comune întâlnite în proiectarea și implementarea sistemelor software orientate obiect. Aceste modele de design au devenit cunoscute sub numele de design pattern-uri și au fost împărțite în trei categorii principale: creational (de creare), structural (structurale) și behavioral (comportamentale).

Cartea GoF a prezentat 23 de design pattern-uri, fiecare cu o descriere detaliată a problemei rezolvate, a structurii pattern-ului și a modului în care acesta poate fi implementat într-un proiect software. Această lucrare a avut un impact major în comunitatea dezvoltatorilor și a pus bazele studiului și utilizării design pattern-urilor în industria software.

De atunci, au fost identificate și documentate alte design pattern-uri, iar comunitatea dezvoltatorilor a continuat să exploreze și să adapteze aceste pattern-uri în funcție de necesitățile specifice ale proiectelor lor. S-au dezvoltat și biblioteci și framework-uri care facilitează utilizarea și implementarea design pattern-urilor în diverse limbaje de programare.

Astăzi, design pattern-urile reprezintă o resursă valoroasă pentru dezvoltatorii software, oferind soluții testate și validate pentru probleme comune de design. Acestea contribuie la creșterea calității software-ului, la reducerea complexității și la îmbunătățirea reutilizabilității și a flexibilității codului. Design pattern-urile au devenit parte integrantă a limbajului comun al comunității dezvoltatorilor și sunt utilizate pe scară largă în industrie pentru a construi sisteme software robuste și scalabile.

De-a lungul anilor, cartea "Design Patterns" a devenit o sursă de inspirație pentru dezvoltatori și arhitecți software din întreaga lume. Ea a influențat modul în care se proiectează și se dezvoltă software-ul, promovând principii precum reutilizabilitatea, modularitatea și extensibilitatea.

Datorită impactului său semnificativ în comunitatea dezvoltatorilor, cartea "Design Patterns" a devenit una dintre cele mai citite și studiate lucrări din domeniul informaticii. Ea continuă să servească drept referință valoroasă și sursă de învățare pentru cei interesați de proiectarea software-ului de calitate superioară.

* 1. **Clasificarea design pattern-urilor.**

În general, clasificarea design pattern-urilor se bazează pe modul în care acestea rezolvă anumite probleme de proiectare și pe modul în care sunt organizate. Aceasta oferă o structură și un context înțelegerii și utilizării design pattern-urilor în dezvoltarea software. Una dintre cele mai utilizate clasificări este cea propusă de către Gang of Four (GoF), care împarte design pattern-urile în trei categorii principale: creational (de creare), structural (de structură) și behavioral (de comportament).

Design pattern-urile de creare se concentrează pe procesul de creare a obiectelor și asigurarea unei instanțe adecvate. Acestea includ pattern-uri precum Singleton, Factory Method și Abstract Factory.

Design pattern-urile de structură abordează modul în care obiectele sunt organizate și interconectate pentru a forma structuri mai complexe. Acestea includ pattern-uri precum Adapter, Decorator și Proxy.

Design pattern-urile de comportament se concentrează pe modul în care obiectele interacționează și comunica între ele. Ele rezolvă probleme legate de comportamentul și responsabilitățile obiectelor. Exemple de design pattern-uri de comportament includ Observer, Strategy și Template Method.

Este important de menționat că aceste categorii sunt doar o modalitate de a organiza design pattern-urile și nu sunt exhaustive. Există și alte clasificări propuse de alți autori și practicieni în domeniul dezvoltării software. Clasificarea design pattern-urilor este utilă deoarece oferă o structură și o direcție în selecția și aplicarea design pattern-urilor în proiectarea software. Ea permite dezvoltatorilor să identifice soluții comune și să beneficieze de experiența și bunele practici acumulate în domeniu.

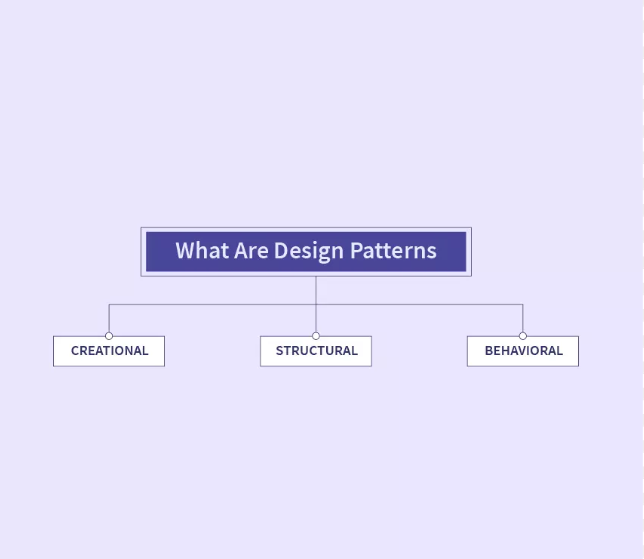


Figura 1 – Clasificarea Design Patterns: Creational, Structural, Behavioral.

* 1. **Creational design patterns.**

Creational design patterns se concentrează pe procesul de creare și inițializare a obiectelor, asigurându-se că acestea sunt create în mod corespunzător și că o singură instanță este utilizată în cazul în care este necesar. Aceste pattern-uri oferă un nivel de abstractizare pentru crearea obiectelor.

1. Singleton: Acest pattern se asigură că o clasă are o singură instanță în întregul sistem și oferă un punct global de acces la acea instanță. Singleton-ul este util atunci când avem nevoie de un obiect care trebuie să fie unic în întregul sistem, cum ar fi un obiect de configurare sau un obiect de gestionare a resurselor.
2. Factory Method: Acest pattern oferă o metodă de fabrică pentru crearea de obiecte, permitând subclass-urilor să decidă tipul de obiect pe care îl creează. Astfel, se realizează o separare a logicii de creare a obiectelor de logica de utilizare a acestora. Factory Method-ul este folosit când avem o clasă de bază abstractă care definește o interfață pentru crearea obiectelor, dar permite subclass-urilor să decidă exact ce tip de obiecte să creeze.
3. Abstract Factory: Acest pattern furnizează o interfață pentru crearea de familii de obiecte legate sau dependente, fără a specifica clasele concrete. Abstract Factory-ul permite crearea de obiecte care sunt concepute pentru a lucra împreună și sunt înlocuite în mod corespunzător de o altă familie de obiecte.
4. Builder: Acest pattern este folosit pentru a construi obiecte complexe pas cu pas. Builder-ul separă procesul de construcție a unui obiect de reprezentarea finală a obiectului, permițând construirea diferitelor reprezentări ale aceluiași obiect. Acesta oferă flexibilitate și ușurință în construirea obiectelor complexe, precum și posibilitatea de a utiliza același proces de construcție pentru obiecte diferite.
5. Prototype: Acest pattern se concentrează pe crearea de copii ale unui obiect existent în loc de crearea unui obiect nou de fiecare dată. Acesta permite clonarea obiectelor, economisind timp și resurse. Prototype-ul este util atunci când crearea de noi obiecte prin intermediul construcției este costisitoare sau complicată, iar copierea unui obiect existent este mai eficientă.

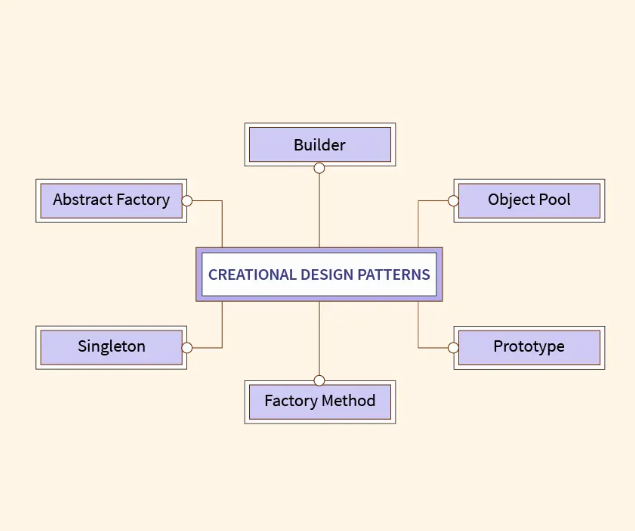


Figura 2 – Creational design patterns.

* 1. **Structural design patterns.**

Structural design patterns se concentrează pe organizarea obiectelor și relațiilor între ele pentru a forma structuri mai mari și mai complexe. Aceste pattern-uri oferă soluții pentru a obține o mai mare flexibilitate, modularitate și extensibilitate a sistemului software, reducând dependențele strânse între componente.

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Facade
6. Flyweight
7. Proxy

Mai detaliat despre undele din structural design patterns:

1. Adapter: Acest pattern permite comunicarea între două clase care au interfețe incompatibile prin intermediul unui adaptator. Adapterul acționează ca un intermediar între cele două clase, traducând cererile de la un sistem în cereri pe care celălalt sistem le poate înțelege. Acesta permite integrarea componentelor existente și reutilizarea acestora în alte părți ale sistemului.
2. Decorator: Acest pattern permite adăugarea de comportamente suplimentare la obiecte existente, fără a le modifica structura de bază. Decoratorul utilizează compunerea în locul moștenirii și oferă flexibilitate în adăugarea și înlăturarea dinamică a funcționalității la un obiect. Acesta oferă o alternativă la creșterea numărului de subclase pentru a extinde funcționalitatea unei clase.
3. Proxy: Acest pattern furnizează un substitut sau un intermediar pentru un obiect pentru a controla accesul la acesta. Proxy-ul acționează ca o îmbrățișare în jurul obiectului real și poate gestiona sarcini precum controlul accesului, gestionarea cererilor și furnizarea de comportamente suplimentare. Acesta poate fi utilizat pentru a controla accesul la obiecte sensibile sau costisitoare în ceea ce privește resursele.

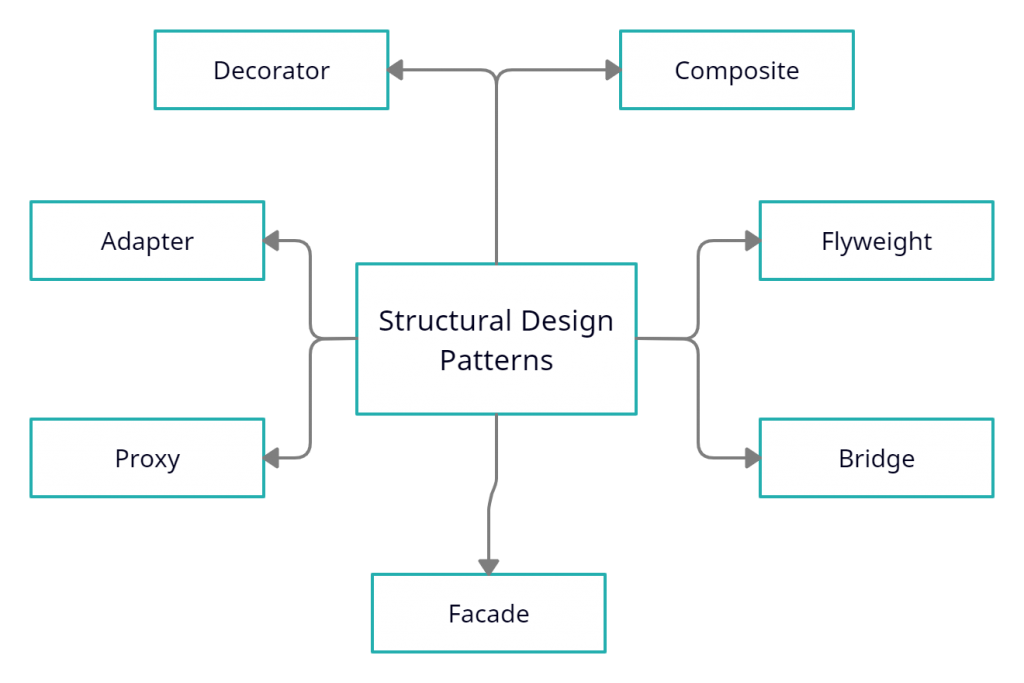


Figura 3 – Structural design patterns.

* 1. **Behavioral design patterns.**

Design pattern-urile de tip behavioral se concentrează pe gestionarea și organizarea comportamentului și interacțiunii între obiecte în cadrul sistemelor software. Ele oferă soluții pentru situații în care dorim să separăm și să modularizăm comportamentul, să gestionăm schimbările de stare și să faciliteze comunicarea și coordonarea între componente.

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template Method
11. Visitor

Mai detaliat despre undele din behavioral design patterns:

1. Observer (Observator): Acest pattern permite comunicarea între obiecte într-un mod decuplat, astfel încât atunci când starea unui obiect se schimbă, toate obiectele dependente de acesta sunt notificate automat și actualizate în consecință. Observer-ul facilitează crearea unui sistem de tip publicare-abonare, în care obiectele pot observa și reacționa la evenimente sau schimbări într-un mod flexibil.
2. Command (Comandă): Acest pattern encapsulează o cerere sub forma unui obiect, permițând parametrizarea clienților cu diferite cereri și gestionarea acestora într-o coadă sau într-o listă. El permite separarea emitentului unei cereri de receptorul acesteia, permițând astfel obiectelor să fie utilizate într-o gamă variată de scenarii de invocare și gestionare a comenzilor.



Figura 4 - Behavioral design patterns.

1. **Proiectarea și implementarea. Noțiuni generale.**

În această aplicație, C# a fost ales ca limbaj de programare. C# este un limbaj de programare potrivit pentru implementarea și utilizarea design patterns. Design patterns sunt soluții recunoscute și testate pentru probleme comune de design software, iar C# oferă suport puternic pentru implementarea acestora. C# oferă concepte cheie, cum ar fi clase, obiecte, moștenire, interfețe și polimorfism, care sunt fundamentale pentru implementarea design patterns. De asemenea, C# oferă funcționalități avansate, cum ar fi evenimente, delegați, expresii lambda și LINQ, care facilitează implementarea pattern-urilor de tip behavioral, cum ar fi Observer și Strategy. C# are și suport pentru pattern-uri de creare, cum ar fi Singleton și Factory Method, prin intermediul constructorilor, metodelor statice și a conceptelor de instanțiere de obiecte.



Figura 5 – Logo C#.

Editorul Visual Studio Code a fost folosit ca mediu de dezvoltare. VS Code este un editor de cod sursă extrem de popular și puternic, dezvoltat de Microsoft. Acesta oferă o experiență de dezvoltare ușoară și eficientă, fiind disponibil pe multiple platforme, inclusiv Windows, macOS și Linux. VS Code vine cu o gamă largă de funcționalități avansate, inclusiv completarea automată a codului, evidențierea sintaxei, depanarea integrată și integrarea cu controlul versiunii. De asemenea, permite utilizatorilor să personalizeze aspectul și comportamentul editorului prin intermediul temelor și extensiilor disponibile în modulul de extensibilitate. Un aspect remarcabil al VS Code este integrarea puternică cu diverse limbaje de programare, cum ar fi C#, JavaScript, Python, Java și multe altele. Aceasta înseamnă că dezvoltatorii pot utiliza VS Code pentru a dezvolta și depana aplicații în multiple limbaje, beneficiind de instrumente specifice pentru fiecare limbaj.

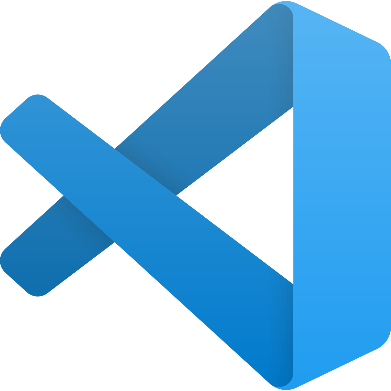


Figura 6 – Logo Visual Studio Code.

* 1. **Abstract Factory.**

Abstract Factory este un design pattern creational care permite crearea de familii de obiecte asociate și interconectate într-un mod abstract. Acesta oferă o interfață pentru crearea obiectelor, dar permite implementări multiple ale acestei interfețe, ceea ce permite crearea de obiecte diferite, dar coezive, fără a dezvălui detalii specifice de implementare. Abstract Factory este util în situațiile în care un sistem trebuie să fie independent de modul în care obiectele concrete sunt create, configurate și asamblate. Prin intermediul acestui pattern, clienții pot crea și utiliza obiecte fără a cunoaște detaliile de implementare specifice, ceea ce facilitează flexibilitatea și extensibilitatea sistemului.

S-a utilizat Abstract Factory pentru a crea și gestiona obiecte legate de tur (Interfața ITour), adică tuturile Prmeium, Regular și Budget, la fel și tipurile zborului – First class, Business class și Econom class, prețurile lor și tipul hotelului (Luxary, Standart și Budget hotel, de 5, 4 și 3 stele), în dependență de turul ales de client.

Interfața ITour este definită pentru a servi ca o fabrică abstractă. Aceasta conține trei metode: CreateHotel(), CreateTravel() și IPayment(). Metodele CreateHotel() și CreateTravel() sunt responsabile pentru crearea și returnarea obiectelor de tip TourA și TourB, în timp ce metoda IPayment() returnează informații despre plată.

*public interface ITour*

*{*

*TourA CreateHotel();*

*TourB CreateTravel();*

*int IPayment();*

*}*

Clasele PremiumTour, RegularTour și BudgetTour implementează interfața ITour și oferă implementări specifice pentru metodele de creare și plată. De exemplu, în clasa PremiumTour, metoda CreateHotel() returnează un obiect de tip LuxaryHotel, în timp ce metoda CreateTravel() returnează un obiect de tip FirstClassTravel. De asemenea, metoda IPayment() returnează o valoare specifică pentru turul premium – prețul turului de 1800.

*class PremiumTour : ITour*

*{*

*public TourA CreateHotel()*

*{*

*return new LuxaryHotel();*

*}*

*public TourB CreateTravel()*

*{*

*return new FirstClassTravel();*

*}*

*public int IPayment()*

*{*

*return 1500;*

*}*

*}*

Interfețele TourA și TourB definesc metodele UsefulFunctionA() și UsefulFunctionB(), respectiv. Acestea reprezintă produsele abstracte pe care fabrica abstractă le creează.

Clasele LuxaryHotel, StandartHotel și BudgetHotel implementează interfața TourA și oferă implementări specifice pentru metoda UsefulFunctionA(). De exemplu, clasa LuxaryHotel returnează un mesaj specific pentru turul premium.

Clasele FirstClassTravel, BusinessClassTravel și BudgetClassTravel implementează interfața TourB și oferă implementări specifice pentru metoda UsefulFunctionB(). De exemplu, clasa FirstClassTravel returnează un mesaj specific pentru turul premium.

*class FirstClassTravel : TourB*

*{*

*public string UsefulFunctionB()*

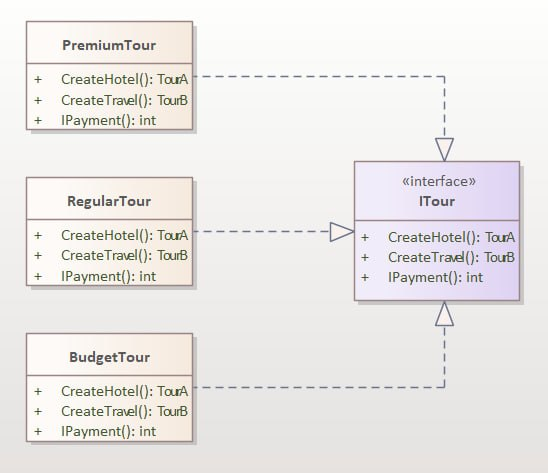
*{*

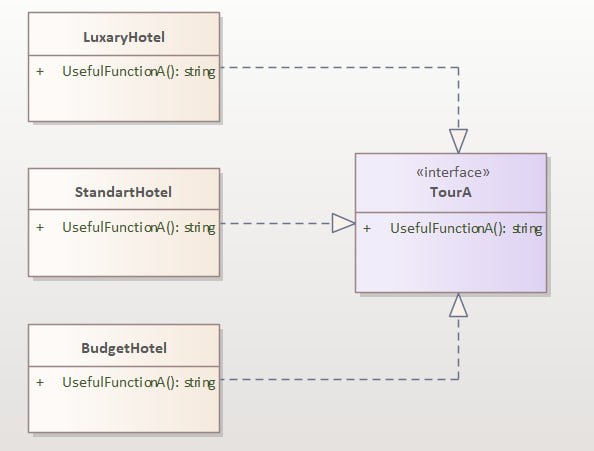
*return "Felicitari! Zborul dvs va fi de clasa intii!";*

*}*

*}*

Prin utilizarea design pattern-ului Abstract Factory, codul asigură că obiectele create sunt coerente și corespund unui anumit tip de tur ales. De exemplu, dacă se creează un tur de buget, fabrica abstractă va crea un hotel de buget și o călătorie de clasă economică.





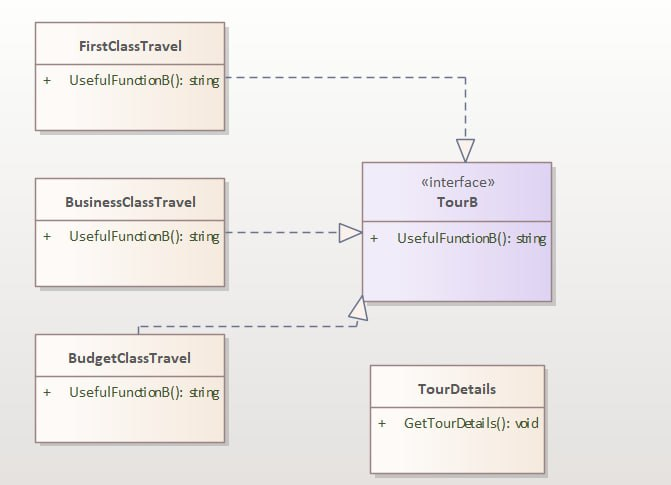


Figura 7 – Diagrama de clase pentru Abstract Factory.

* 1. **Factory Method.**

Factory Method este un design pattern creational care oferă o abordare pentru crearea de obiecte, permițând clasei să se ocupe de procesul de instanțiere fără a fi nevoie să specifice clar tipul obiectelor pe care le creează. În acest pattern, o clasă abstractă sau o interfață definește o metodă abstractă numită Factory Method, iar clasele derivate își implementează propriul Factory Method pentru a crea obiecte specifice. Factory Method oferă o flexibilitate ridicată în crearea de obiecte, deoarece permite Creatorului să fie decuplat de clasele concrete specifice pe care le creează. Astfel, se poate adăuga ușor noi Concrete Creatori pentru a crea noi tipuri de obiecte, fără a afecta codul existent. Acest pattern este util în situațiile în care există o logică complexă de creare a obiectelor sau când se dorește ca procesul de instanțiere să fie flexibil și extensibil. Prin intermediul Factory Method-ului, responsabilitatea creării obiectelor este transferată subclaselor, permițând crearea de obiecte specializate într-un mod modular și ușor de gestionat.

S-a utilizat Factory Method pentru a reprezenta călătoriile în diferite țări și informații despre client (nume, prenume, ID pașaport), respectiv în dependență de țara aleasă (Turcia, Grecia, Egipt) se afișează informația despre client și o mică prezentare a țării alese.

Clasa Tour reprezintă o călătorie și conține informații despre client (nume, prenume, ID pașaport) și un Factory Method abstract CreateCountryTour(). Această clasă servește ca bază pentru clasele concrete TurkeyTour, GreeceTour și EgyptTour, care reprezintă călătorii specifice în Turcia, Grecia și Egipt.

*public Tour(string clientName, string clientSurname, string passportId)*

*{*

*ClientName = clientName;*

*ClientSurname = clientSurname;*

*PassportId = passportId;*

*}*

*//facotry method*

*abstract public CountryTour CreateCountryTour();*

Fiecare clasă concretă a călătoriei extinde clasa Tour și implementează Factory Method-ul CreateCountryTour(). Acest Factory Method creează și returnează o instanță a unei clase specifice țării respective, cum ar fi TourInTurkey, TourInGreece și TourInEgypt. Aceste clase specifice țărilor reprezintă informații despre călătoriile în respectivele țări.

De exemplu, atunci când se creează o instanță TurkeyTour, se afișează un mesaj de selectare a călătoriei în Turcia și se returnează un obiect de tip TourInTurkey. Apoi, se poate apela metoda someInformation() pe obiectul returnat pentru a obține informații specifice despre călătoria în Turcia.

*public override CountryTour CreateCountryTour()*

*{*

*return new TourInEgypt();*

*}*

*}*

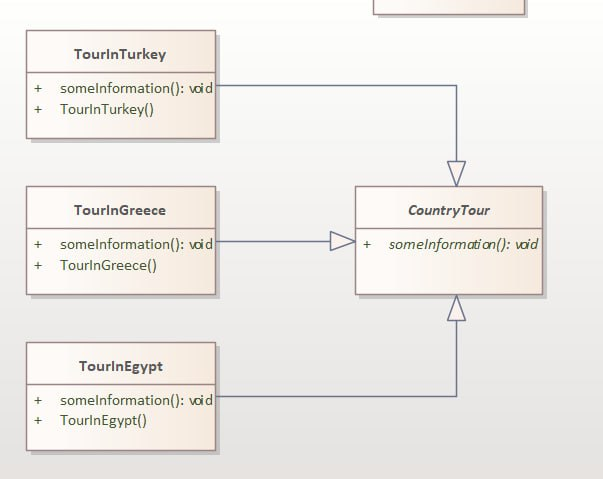
*abstract class CountryTour*

*{*

*abstract public void someInformation();*

*}*

Se utilizează design pattern-ul Factory Method pentru a abstractiza procesul de creare a călătoriilor și pentru a permite extinderea facilă a sistemului prin adăugarea de noi călătorii în alte țări. Acest design pattern ajută la decuplarea Creatorului (Tour) de produsele create (CountryTour), oferind flexibilitate în gestionarea diferitor tipuri de călătorii.



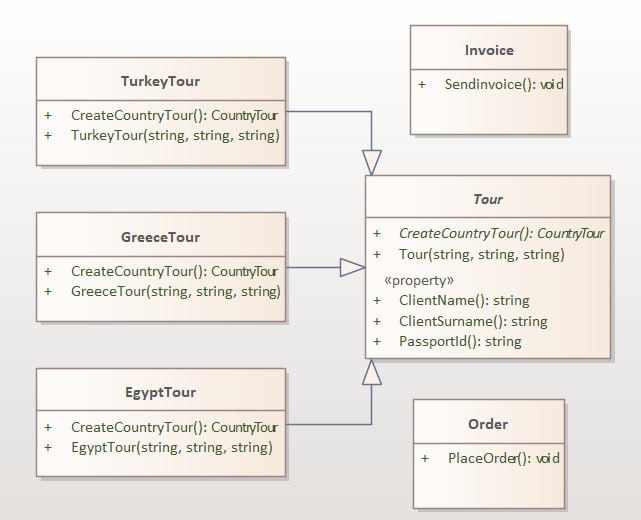


Figura 8 – Diagrama de clase pentru Factory Method.

* 1. **Facade.**

Design pattern-ul Facade este utilizat pentru a oferi o interfață simplificată pentru a accesa un subsistem complex de clase, structuri sau module. El ascunde complexitatea din spate și oferă o interfață unificată și mai ușor de utilizat. În design pattern-ul Facade, clasa Facade acționează ca un intermediar între client și subsistemul complex. Ea abstractizează interacțiunea cu subsistemul și oferă metode simple și coezive pentru a accesa funcționalitățile subsistemului. Astfel, clientul nu trebuie să interacționeze direct cu multiplele clase și componente ale subsistemului, ci doar cu clasa Facade.

S-a implementat design pattern Facade pentru trei clase care reprezintă subsistemele: TourDetails, Payment și Invoice. Fiecare dintre aceste clase are o metodă care reprezintă o acțiune specifică în cadrul subsistemului. De exemplu, clasa Payment:

*public class Payment*

*{*

*public void MakePayment()*

*{*

*Console.Write("\nPlata a trecut cu succes!");*

*}*

*}*

Clasa Order acționează ca o clasă Facade. Ea oferă o metodă numită PlaceOrder, care reprezintă o interfață simplificată pentru a accesa funcționalitățile subsistemelor. În cadrul metodei PlaceOrder, se creează obiecte ale claselor TourDetails, Payment și Invoice și se apelează metodele lor corespunzătoare pentru a colecta detalii despre tur, a efectua plata și a trimite factura. La final, se afișează un mesaj că comanda a fost plasată cu succes.

*public class Order*

*{*

*public void PlaceOrder()*

*{*

*TourDetails tour= new TourDetails();*

*tour.GetTourDetails();*

*Payment payment = new Payment();*

*payment.MakePayment();*

*Invoice invoice = new Invoice();*

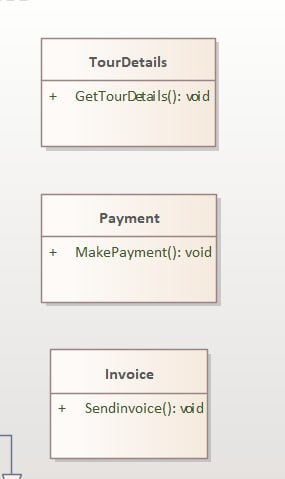
*invoice.Sendinvoice();*

*Console.WriteLine("\nComanda plasata cu succes");*

*}*

*}*

Prin utilizarea clasei Order ca o interfață unificată, clientul poate plasa o comandă fără a cunoaște detaliile și interacțiunile complexe din spate. Subsistemele sunt ascunse și complexitatea este gestionată în interiorul clasei Order. Design pattern-ul Facade simplifică interacțiunea cu subsistemele complexe, oferind o interfață simplă și unificată. El permite clientului să utilizeze funcționalitățile subsistemelor fără a cunoaște detaliile acestora.



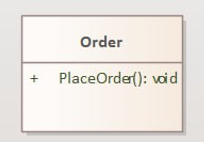


Figura 9 – Diagrama de clase pentru Facade.

* 1. **Bridge.**

Design pattern-ul Bridge permite separarea unei abstracții de implementarea sa, astfel încât ambele pot varia independent una de cealaltă. Prin implementarea Bridge, obiectele dintr-o ierarhie abstractă sunt legate cu obiecte dintr-o ierarhie de implementare separată. Acesta promovează encapsularea, decuplarea și extensibilitatea într-un mod flexibil. Conceptul de bază al pattern-ului Bridge este că există două ierarhii separate: ierarhia de abstracție și ierarhia de implementare. Ierarhia de abstracție definește interfețele și clasele abstracte pentru obiectele principale, iar ierarhia de implementare furnizează implementări specifice ale acestor interfețe și clase abstracte. Prin utilizarea unei legături (bridge) între cele două ierarhii, putem obține o conexiune flexibilă și dinamică între ele. Astfel, orice schimbare într-o parte a pattern-ului nu va afecta cealaltă parte.

Avantajele utilizării pattern-ului Bridge includ:

1. Separarea abstracției de implementare, permițându-le să evolueze independent.
2. Extensibilitatea și flexibilitatea îmbunătățite, deoarece putem adăuga noi clase de abstracție și implementare fără a afecta cele existente.
3. Ascunderea detaliilor de implementare de către clienți și reducerea cuplării dintre abstracție și implementare.
4. Suport pentru dezvoltarea de cod modular și ușor de întreținut.

S-a implementat design pattern Bridge, pentru două ierarhii separate: ierarhia de implementare (IAccount) și ierarhia de abstracție (Person). Ierarhia de implementare definește interfața IAccount, care are două metode: CreateAccount() și CreateDiscount().

*public interface IAccount*

*{*

*IAccount CreateAccount();*

*double CreateDiscount();*

*}*

În ierarhia de implementare, sunt două clase concrete: DiscountAccount și RegularAccount. Aceste clase implementează interfața IAccount și furnizează implementări specifice pentru metodele sale. DiscountAccount oferă un cont cu o reducere de 25%, în timp ce RegularAccount oferă un cont cu o reducere de 10%.

*public class DiscountAccount : IAccount*

*{*

*public double discount = 0.25;*

*public IAccount CreateAccount()*

*{*

*Console.WriteLine("Inregistrare cu succes!\n");*

*return new DiscountAccount();*

*}*

*public double CreateDiscount()*

*{*

*Console.WriteLine("Calculam reducerea\n\nFelicitari, aveti reducere pentru primul tur - 25%!");*

*return discount;*

*}*

În ierarhia de abstracție, sunt două clase concrete: NaturalPerson și LegalPerson. Aceste clase extind clasa abstractă Person și implementează metoda CreateAccount(). Clasele NaturalPerson și LegalPerson primesc un obiect IAccount prin intermediul constructorului și utilizează acest obiect pentru a crea un cont.

*public class NaturalPerson : Person*

*public class LegalPerson : Person*

Prin utilizarea pattern-ului Bridge, putem separa conceptul de cont bancar (IAccount) de tipul de persoană (Person). Astfel, putem crea diverse tipuri de conturi și de persoane și le putem combina în mod flexibil. De exemplu, putem avea o persoană fizică cu un cont cu reducere și o persoană juridică cu un cont regulat.

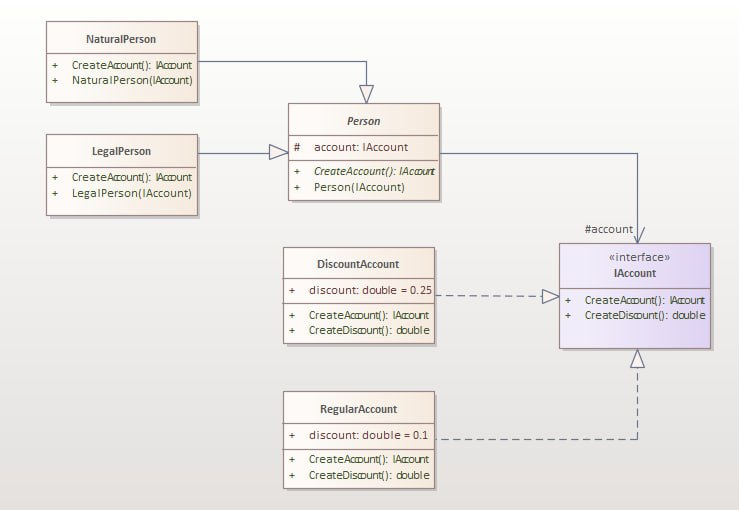


Figura 10 – Diagrama de clase pentru Bridge.

* 1. **Chain Of Reponsibility.**

Chain of Responsibility este un design pattern comportamental care permite construirea unui lanț de obiecte care prelucrează și gestionează o solicitare în mod secvențial. Fiecare obiect din lanț poate decide să prelucreze solicitarea sau să o transmită mai departe la următorul obiect din lanț. Principiul de bază al Chain of Responsibility este că solicitarea este trimisă de-a lungul lanțului de obiecte până când unul dintre obiecte o prelucrează sau până când ajunge la capătul lanțului, fără a fi prelucrată de niciunul dintre obiecte. Acest pattern oferă flexibilitate și extensibilitate, deoarece permite adăugarea și eliminarea ușoară a obiectelor din lanț fără a afecta codul existent.

S-a implementat design pattern Chain of Responsibility, împreună cu o interfață și clase adiționale legate de validarea plății.

Interfața IPayment definește o metodă Payment care primește o sumă de plată și o reducere și returnează suma finală de plată după aplicarea reducerii.

*interface IPayment*

*{*

*double Payment(double payment, double discount);*

*}*

Clasa abstractă ValidatorBase este clasa de bază pentru toți validatorii din lanț. Aceasta conține o referință la următorul validator succesor și furnizează o metodă SetSuccessor pentru a seta validatorul succesor. De asemenea, declară o metodă abstractă ProcessRequest care gestionează logica de procesare a cererii.

*public abstract class ValidatorBase*

*{*

*protected ValidatorBase successor;*

*public void SetSuccessor(ValidatorBase successor)*

*{*

*this.successor = successor;*

*}*

*public abstract void ProcessRequest(Request request);*

*}*

Clasa DateAvailableValidator este o implementare concretă a validatorului. Verifică dacă cererea se referă la disponibilitatea unei date pentru un tur. Dacă cererea se potrivește, afișează mesajul care indică disponibilitatea datei. Dacă cererea nu se potrivește, trece cererea la validatorul succesor.

*if (request.request == "data")*

*{*

*Console.WriteLine("Data libera pentru tur: {0}",*

*request.RandomDay());*

*}*

Clasa ApartmentAvailableValidator este o altă implementare concretă a validatorului. Verifică dacă cererea se referă la disponibilitatea unui apartament într-un hotel. Dacă cererea se potrivește, afișează mesajul care indică disponibilitatea apartamentului. Dacă cererea nu se potrivește, trece cererea la validatorul succesor.

*if (request.request == "numarul")*

*{*

*Console.WriteLine(*

*"Camera in hotel este disponibila"*

*);*

*}*

Clasa PaymentValidator este o implementare concretă a validatorului și implementează, de asemenea, interfața IPayment.

*public class PaymentValidator : ValidatorBase,IPayment*

Verifică dacă cererea se referă la plata unui tur. Dacă cererea se potrivește, afișează mesajul care indică calculul plății. De asemenea, implementează metoda Payment din interfața IPayment pentru a calcula suma finală de plată după aplicarea reducerii. Clasa Request reprezintă un obiect de cerere și moștenește de la DateAvailableValidator. Conține un câmp request care stochează tipul de cerere și o metodă RandomDay care generează o dată aleatoare între 2023 și 2027. Este folosită pentru a simula disponibilitatea unei date pentru un tur.

*public DateTime RandomDay()*

*{*

*DateTime start = new DateTime(2023, 1, 1,13,30,0);*

*DateTime end = new DateTime(2027, 1, 1,1,1,1);*

*int range = (end - start).Days;*

*return start.AddDays(gen.Next(range));*

*}*

Implementarea Chain of Responsibility, în care diferiți validatori sunt înlanțuiți împreună, iar fiecare validator are capacitatea de a gestiona un anumit tip de cerere sau de a o trece la validatorul succesor s-a efectuat cu succes.

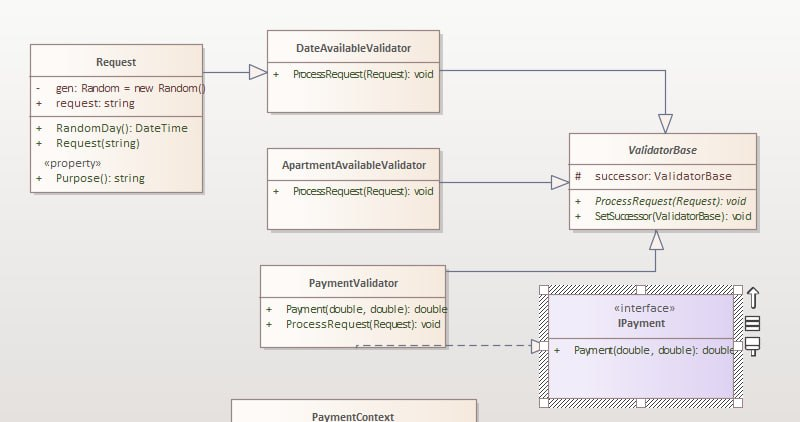


Figura 11 – Diagrama de clase pentru Chain of Responsibility.

* 1. **Strategy.**

Strategy este unul dintre cele mai cunoscute șabloane comportamentale din programarea orientată pe obiecte. Acesta permite definirea unei familii de algoritmi, încapsulează fiecare algoritm într-o clasă separată și le face interschimbabile între ele în timpul rulării. Strategy este un șablon de proiectare care permite definirea unei familii de algoritmi, încapsulează fiecare algoritm într-o clasă separată și le face interschimbabile. Acest șablon permite algoritmul să varieze independent de clienți care îl utilizează.

Avantaje:

1. Flexibilitate și înlocuire ușoară a algoritmilor: Pattern-ul Strategy permite definirea diferitelor strategii sau algoritmi și schimbarea acestora în timpul execuției fără a afecta obiectele care utilizează strategiile. Astfel, puteți înlocui și schimba comportamentul unui obiect prin schimbarea strategiei asociate, fără a modifica codul obiectului în sine.
2. Reutilizare și extensibilitate: Fiecare strategie este o entitate independentă și reutilizabilă, care poate fi utilizată în mai multe contexte sau de către diferite obiecte. Aceasta permite o structură modulară și extensibilă a codului, deoarece puteți adăuga cu ușurință strategii noi fără a modifica codul existent.
3. Simplificarea codului și evitarea duplicării: Pattern-ul Strategy permite separarea algoritmilor complexi de logica de bază a obiectului, astfel încât codul să fie mai ușor de citit, de înțeles și de întreținut. În plus, permite evitarea duplicării codului atunci când există algoritmi similari care trebuie utilizați în diferite contexte.
4. Dependențe reduse și încapsulare: Obiectele care utilizează strategii nu sunt dependente de implementările concrete ale acestora, ci se bazează pe interfața comună a strategiei. Aceasta duce la o încapsulare mai bună și reduce cuplarea între obiecte, ceea ce facilitează testarea, întreținerea și refolosirea codului.
5. Îmbunătățirea extensibilității și adaptabilității: Pattern-ul Strategy permite adăugarea de strategii noi fără a modifica codul existent. Astfel, puteți extinde funcționalitatea obiectelor și le puteți adapta pentru a satisface cerințele și schimbările ulterioare fără a afecta codul existent.

S-a implementat design pattern Strategy pentru sistemul de plată a agenției de turism.

Clasa abstractă PersonalData conține proprietăți pentru informațiile personale ale unui client, cum ar fi numele, prenumele, ID-ul pașaportului, tipul de tur, destinația, data și modalitatea de plată. Aceasta este utilizată ca clasă de bază pentru clasele de plată.

*public abstract class PersonalData*

*{*

*public string ClientName { get; set; }*

*public string ClientSurname { get; set; }*

*public string PassportId { get; set; }*

*public string QualityTour { get; set; }*

*public string Destination { get; set; }*

*public DateTime Date { get; set; }*

*public string Pay { get; set; }*

Clasa CashPayment implementează interfața IPaymentStrategy și reprezintă o strategie de plată în numerar. Metoda Payment() afișează un mesaj pentru plata în numerar, indicând că trebuie să fie făcută la biroul de pe bd. Moscova 17.

*public class CashPayment : IPaymentStrategy*

*{*

*public void Payment()*

*{*

*Console.WriteLine("Pentru plata cash va invitam la biroul de pe bd. Moscova 17");*

*}*

*}*

Clasa CardPayment extinde clasa PersonalData și implementează interfața IPaymentStrategy. Aceasta reprezintă o strategie de plată cu cardul. Constructorul primește informațiile personale ale clientului și le afișează. Metoda Payment() solicită utilizatorului să introducă numărul cardului și afișează un mesaj că acesta a fost găsit.

*public class CardPayment : PersonalData, IPaymentStrategy*

*{*

*public CardPayment (string clientName, string clientSurname, string passportId, string qualitytour, string destination, DateTime date, string pay) : base(clientName, clientSurname,passportId, qualitytour,destination, date, pay)*

*{*

Clasa EWalletPayment extinde clasa PersonalData și implementează interfața IPaymentStrategy. Aceasta reprezintă o strategie de plată cu e-Wallet. Constructorul primește informațiile personale ale clientului și le afișează. Metoda Payment() solicită utilizatorului să introducă numărul e-Wallet și afișează un mesaj că acesta a fost găsit.

*public void Payment()*

*{*

*Console.Write("\nIntroduceti numarul e-Wallet: ");*

*string card = Console.ReadLine();*

*System.Threading.Thread.Sleep(1000);*

*Console.Write("Clientul cu nr e-Wallet {0} este gasit\n", card);*

*}*

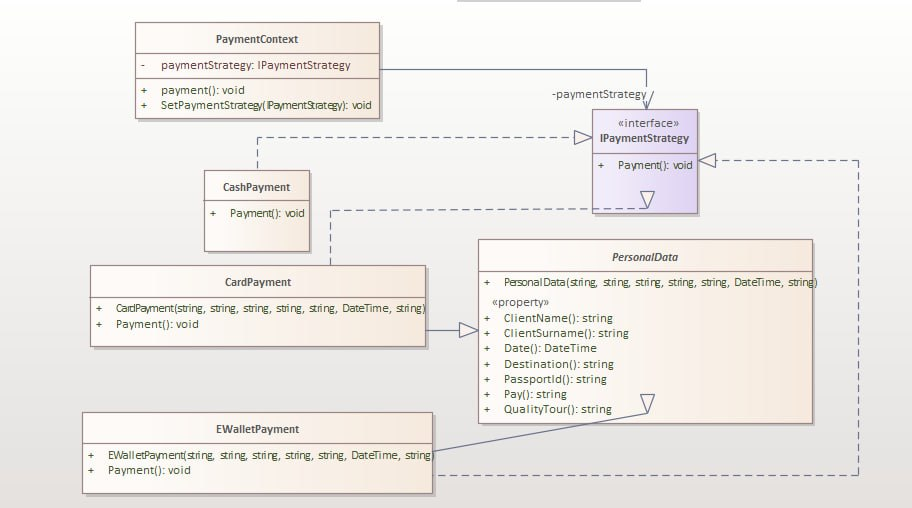


Figura 12 – Diagrama de clase pentru Strategy.

1. **Concluzie**

Aplicarea design patterns în dezvoltarea aplicației de agenție de turism aduce multiple beneficii și îmbunătățește calitatea și eficiența proiectului. Iată o concluzie despre acest subiect:

Utilizarea design patterns în crearea unei aplicații de agenție de turism oferă o structură clară și modulară, care facilitează dezvoltarea, întreținerea și extinderea ulterioară a aplicației. Prin aplicarea pattern-urilor precum Abstract Factory, Chain of Responsibility și Strategy, etc., se obține o arhitectură flexibilă, scalabilă și ușor de înțeles.

Abstract Factory permite crearea de familii de obiecte legate între ele, precum rezervări de hoteluri, bilete de avion sau excursii, într-un mod uniform și consistent. Astfel, se simplifică procesul de adăugare sau modificare a noilor servicii de călătorie în aplicație.

Chain of Responsibility permite gestionarea solicitărilor și fluxului de informații într-un mod eficient și modular. Acesta permite procesarea și distribuirea solicitarilor într-o manieră ierarhică și flexibilă, eliminând dependențele stricte între obiecte și permitând adăugarea ușoară a noi etape de validare sau procesare a solicitărilor.

Strategy permite implementarea diferitelor strategii de plată în cadrul aplicației de agenție de turism. Acest pattern facilitează schimbul flexibil între diferitele metode de plată, cum ar fi plata în numerar, plata cu cardul sau plata prin portofel electronic. Astfel, se oferă utilizatorilor opțiuni diverse de plată, adaptate preferințelor și nevoilor lor.

Prin aplicarea acestor design patterns, aplicația de agenție de turism devine mai modulară, ușor de extins și de întreținut. Aceasta permite o gestionare mai eficientă a funcționalităților, o separare clară a responsabilităților și o adaptare rapidă la schimbările cerințelor și nevoilor utilizatorilor.

În concluzie, utilizarea design patterns în dezvoltarea unei aplicații de agenție de turism aduce beneficii semnificative, precum modularitate, scalabilitate și ușurință în gestionarea și extinderea aplicației. Prin aplicarea corectă a acestor patterns, se creează o aplicație robustă, flexibilă și ușor de utilizat, care poate satisface nevoile și cerințele utilizatorilor în domeniul turismului.

**Link la repozitoriu:** https://github.com/irinasclearenco/TMPS-project.git

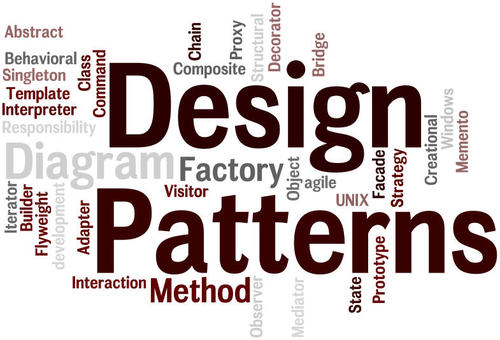


Figura 13 – Design Patterns.

1. **Bibliografie**
2. <https://refactoring.guru/design-patterns>
3. <https://sourcemaking.com/design_patterns>
4. <https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm>
5. <http://elf.cs.pub.ro/poo/laboratoare/design-patterns>
6. <https://ro.myservername.com/design-patterns-java>
7. <https://codecool.com/ro/blog/ghid-limbaj-c-incepatori/>
8. <https://ru.wikipedia.org/wiki/Visual_Studio_Code>
9. <https://www.javatpoint.com/creational-design-patterns>
10. <https://sourcemaking.com/design_patterns/structural_patterns>
11. https://www.scaler.com/topics/design-patterns/behavioral-pattern/
12. <https://www.dofactory.com/net/strategy-design-pattern>
13. https://www.geeksforgeeks.org/facade-design-pattern-introduction/
14. **Anexe**

