Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**Proiect de an**

Disciplina: **Tehnici și Mecanisme de Proiectare Software**

**Tema: Implementarea aplicației ”EXPENSE TRACKER”**

A efectuat: student, gr: TI-202

Pîntea Adina

A verificat: asistent universitar

Gaidau Mihai

Chișinău – 2023

**Cuprins:**

Introducere ................................................................................................................................. 3

Analiza domeniului de studiu ..................................................................................................... 4

1. Design Patterns(Modele de proiectare) ............................................................................... 5
   1. Importanța utilizării Modelelor de proiectare .............................................................. 5
   2. Tipurile Modelelor de Proiectare ................................................................................. 6
2. Implementarea aplicației ................................................................................................... 17
   1. Implementarea șablonului Adapter ............................................................................ 20
   2. Implementarea șablonului Builder ............................................................................. 22
   3. Implementarea șablonului Decorator ......................................................................... 23
   4. Implementarea șablonului Strategy ........................................................................... 25
   5. Implementarea șabloanelor Observer și Factory ........................................................ 27

Concluzie ................................................................................................................................. 30

Bibliografie .............................................................................................................................. 31

**Introducere**

Disciplina Tehnici și Mecanisme de Proiectare Software are ca scop familiarizarea studenților cu principiile și metodele de proiectare eficientă a software-ului, inclusiv utilizarea de șabloane de proiectare și tehnici avansate de dezvoltare.

Rolul său important constă în dezvoltarea abilităților necesare pentru a concepe și implementa soluții software robuste, extensibile și ușor de întreținut, care să îndeplinească cerințele și obiectivele proiectului, precum și să ofere o experiență de utilizare optimă pentru utilizatori. Realizarea unui proiect în cadrul acestei discipline permite aplicarea practică a cunoștințelor dobândite și consolidarea înțelegerii conceptelor teoretice prin intermediul unui produs software concret.

În cadrul proiectului meu de an la disciplina Tehnici și Mecanisme de Proiectare Software, am dezvoltat aplicația "EXPENSE TRACKER". Această aplicație, de tip consolă, a fost concepută pentru a gestiona cheltuielile personale și a interacționa cu o bază de date existentă.

În timpul implementării proiectului, am avut ocazia de a aplica și de a explora diverse sabloane de proiectare, în special cele din cele trei categorii principale: sabloanele creationale, structurale și comportamentale. Aceste sabloane reprezintă soluții recunoscute și testate în dezvoltarea software, care ajută la crearea unor aplicații robuste, extensibile și ușor de întreținut.

Raportul proiectului conține o analiză detaliată a cerințelor și obiectivelor, precum și o descriere a arhitecturii aplicației. De asemenea, am inclus o documentație tehnică cuprinzătoare, care explică în detaliu implementarea și funcționalitățile cheie ale aplicației.

În plus, am prezentat și rezultatele efectuate pentru a verifica performanța aplicației. Am abordat și aspecte legate de securitatea datelor și de gestionarea erorilor, asigurându-ne că aplicația noastră oferă o experiență fiabilă și sigură utilizatorilor.

**Analiza domeniului de studiu**

Un domeniu de studiu se referă la o ramură a cunoașterii. Într-o societate bine dezvoltattă, există mai multe domenii de studiu, unde fiecare corespunde unei specializări sau unui sector specific de cunoștințe.

Domeniul din care face parte proiectul creat, este: ”Tehnologii Informaționale”.

Tehnologia informației este un domeniu element de legătură între electrotehnica clasică și mult mai noua informatică.Ea a devenit metoda principală și necesară pentru prelucrarea(procurarea, procesarea, stocarea, convertirea și transmiterea) informației, în particular, prin folosirea computerelor(Calculatoarelor electronice).

Cât despre Universitatea Tehnică a Moldovei, domeniul Tehnologiei Informaționale, este unul dintre cele mai importante, fiind foarte dezvoltat și ales de viitorii ingineri doritori de ași dezvolta cunoștințele în Mass-Media.

Axându-ne pe acest domeniu, am decis să implementez o aplicație de tip Consolă care presupune ”Gestionarea cheltuielilor financiare”.

Problema finanțelor, este una globală, deoarece orice om, din orice colț al lumii, zilnic risipește o sumă de bani și nu-și poate aminti ”Oare ce am făcut cu acea sumă de bani?”.

Vin cu propunerea, și ideea, de a ușura problema cheltuielilor, sau mai bine zis de a monitoriza cheltuielile zilnice printr-o aplicație.

Această aplicație va oferi oricărui utilizator șansa de a adăuga categorii de cheltuieli, subcategorii, suma de bani cheltuită și careva detalii sau comentarii în legătură cu domeniul dat.

Sper că aplicația dezvoltată, va fi un salt spre economisirea unor sume de bani, deoarece, analizând categoriile în care cheltui finanțe foarte multe și fără rost, va fi o motivare în a înceta, sau în a investi această sumă în domeniul cunoașterii, ceea ce ne va dezvolta și ca societate.

1. **Design Patterns(Modele de proiectare).**

Modelele de proiectare, cunoscute și sub denumirea de Design Patterns, reprezintă abstracții conceptuale și soluții recurente la probleme complexe de proiectare și implementare software. Acestea oferă o modalitate standardizată și testată de a rezolva provocări comune, furnizând structuri, relații și interacțiuni între componente software pentru a atinge obiective precum flexibilitatea, extensibilitatea, ușurința înțelegerii și întreținerii codului.

Modelele de proiectare sunt bazate pe experiența acumulată de-a lungul anilor în dezvoltarea software și sunt documentate și comunicate sub forma unor soluții abstracte, independente de limbajul de programare sau de tehnologia specifică. Acestea descriu modul în care componente software pot interacționa și coopera pentru a rezolva probleme specifice, oferind structuri și comportamente flexibile și reutilizabile.

Prin aplicarea modelelor de proiectare, dezvoltatorii beneficiază de un vocabular comun și de un set de soluții testate, reducând complexitatea proiectării și codificării și asigurând creșterea calității și eficienței în dezvoltarea software. Modelele de proiectare acoperă diferite aspecte ale proiectării, precum organizarea și structurarea codului, gestionarea interacțiunilor între componente, gestionarea stărilor și a fluxului de date, și pot fi utilizate în diverse domenii și aplicații software.

* 1. **Importanța utilizării Modelelor de proiectare.**

Utilizarea modelelor de proiectare aduce numeroase beneficii în dezvoltarea software, îmbunătățind calitatea, flexibilitatea și eficiența aplicațiilor. Acestea reprezintă un instrument esențial pentru dezvoltatorii software, facilitând crearea de soluții robuste, reutilizabile și ușor de întreținut.

Iată câteva dintre cele mai importante beneficii de utilizare:

* Reutilizarea și extensibilitatea codului: Modelele de proiectare oferă soluții standardizate și flexibile la probleme comune, permițând reutilizarea componentelor de software și extinderea lor în mod eficient. Acest lucru duce la crearea unui cod modular și ușor de întreținut, cu potențial de reutilizare în proiecte viitoare.
* Îmbunătățirea calității și robusteței software-ului: Prin aplicarea modelelor de proiectare, dezvoltatorii pot identifica și implementa soluții testate și verificate în abordarea problemelor de proiectare. Acest lucru duce la crearea unui software mai robust, cu mai puține erori și cu performanță îmbunătățită.
* Ușurința înțelegerii și comunicării: Modelele de proiectare oferă un limbaj comun și o structură conceptuală clară, ceea ce facilitează înțelegerea și comunicarea între membrii echipei de dezvoltare. Aceasta duce la o colaborare mai eficientă și la scăderea timpului și efortului necesar pentru a înțelege și modifica codul.
* Flexibilitate și adaptabilitate: Modelele de proiectare permit dezvoltatorilor să construiască software care poate fi ușor adaptat la schimbări și cerințe noi. Aceasta înseamnă că aplicațiile pot fi mai flexibile și pot răspunde mai bine la nevoile utilizatorilor sau la evoluțiile pieței.
* Optimizarea performanței și a eficienței: Modelele de proiectare facilitează implementarea unor soluții eficiente și optimizate, permițând dezvoltatorilor să obțină performanțe superioare și să minimizeze consumul de resurse. Acest lucru poate duce la aplicații mai rapide, mai puțin consumatoare de energie și cu răspuns mai bun pentru utilizatori.
* Documentare și învățare continuă: Modelele de proiectare sunt bine documentate și reprezintă o resursă valoroasă pentru învățare și dezvoltare continuă în domeniul dezvoltării software. Utilizarea acestora încurajează dezvoltatorii să înțeleagă și să aplice concepte și principii de proiectare, contribuind la dezvoltarea unor competențe solide în acest domeniu.
  1. **Tipurile Modelelor de Proiectare.**

Există mai multe tipuri de modele de proiectare, fiecare având un set de patternuri specifice. Iată câteva dintre cele mai cunoscute tipuri de modele de proiectare:

1. **Modele de proiectare creaționale:** oferă diverse mecanisme de creare a obiectelor, care cresc flexibilitatea și reutilizarea codului existent.

* Singleton: Singleton este un model de design creațional care vă permite să vă asigurați că o clasă are o singură instanță, oferind în același timp un punct de acces global la această instanță.

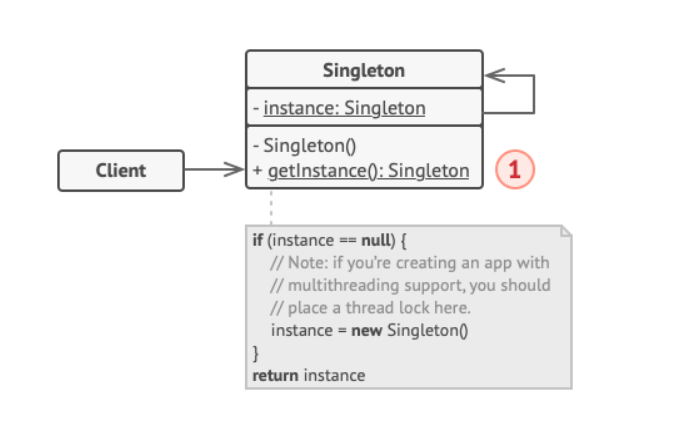


Figura 1.1 – Structura șablonului Singleton

* Builder: Builder este un model de design creațional care vă permite să construiți obiecte complexe pas cu pas. Modelul vă permite să produceți diferite tipuri și reprezentări ale unui obiect folosind același cod de construcție.

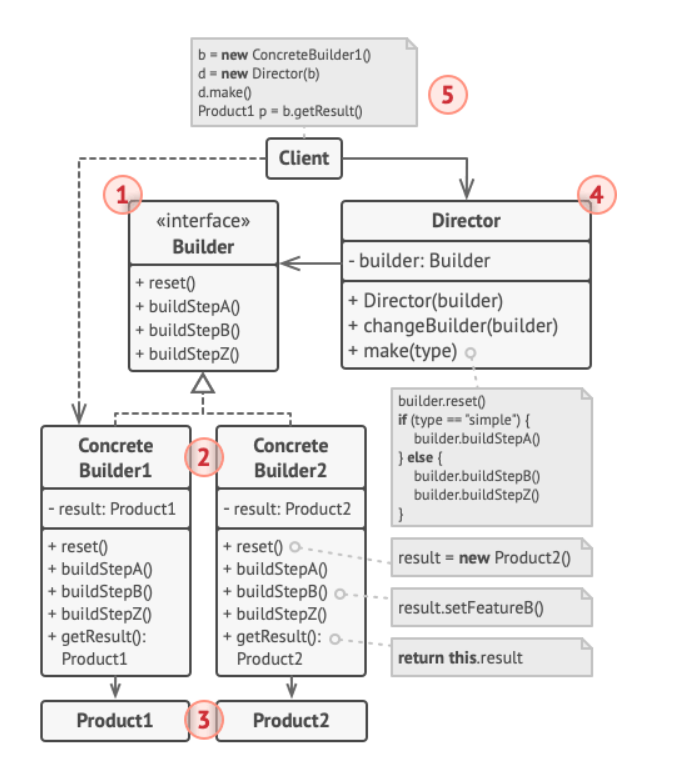


Figura 1.2 – Structura șablonului Builder

* Prototype: Prototipul este un model de design creațional care vă permite să copiați obiectele existente fără a face codul să depindă de clasele lor.

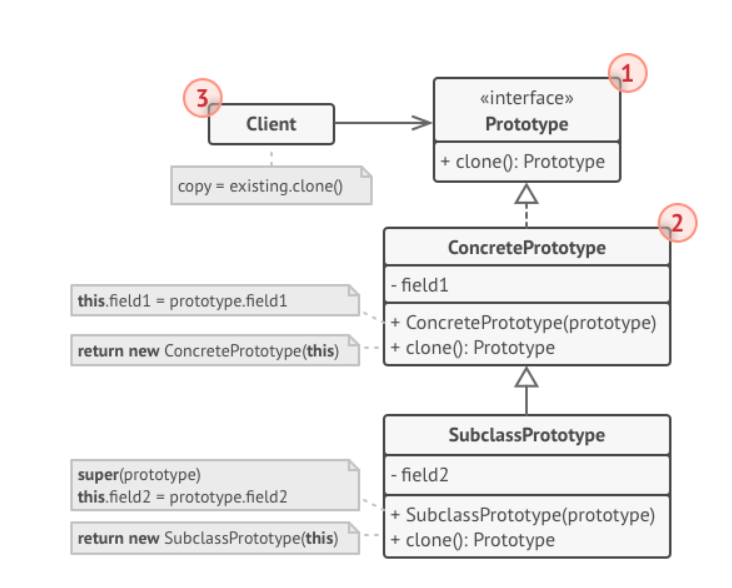


Figura 1.3 – Structura șablonului Prototype. Implementarea de bază

* Factory Method: Factory Method este un model de design creațional care oferă o interfață pentru crearea de obiecte într-o superclasă, dar permite subclaselor să modifice tipul de obiecte care vor fi create.

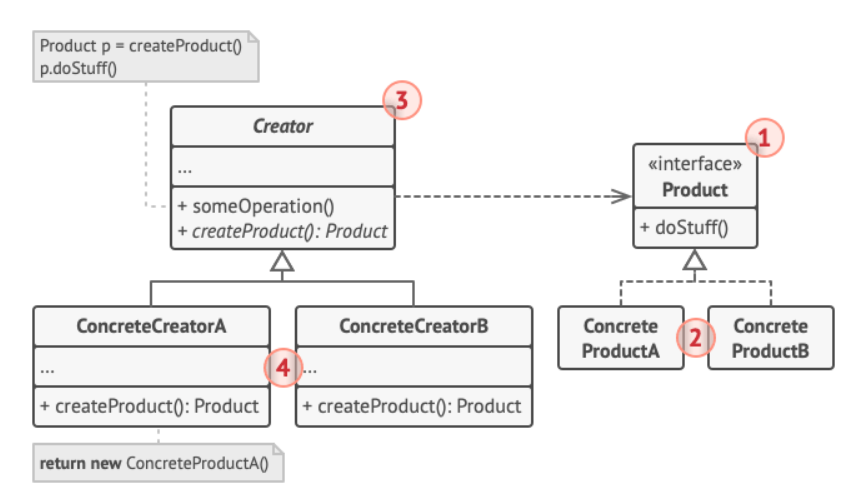


Figura 1.4 - Structura șablonului Factory Method

* Abstract Factory: Abstract Factory este un model de design creațional care vă permite să produceți familii de obiecte înrudite fără a specifica clasele lor concrete.

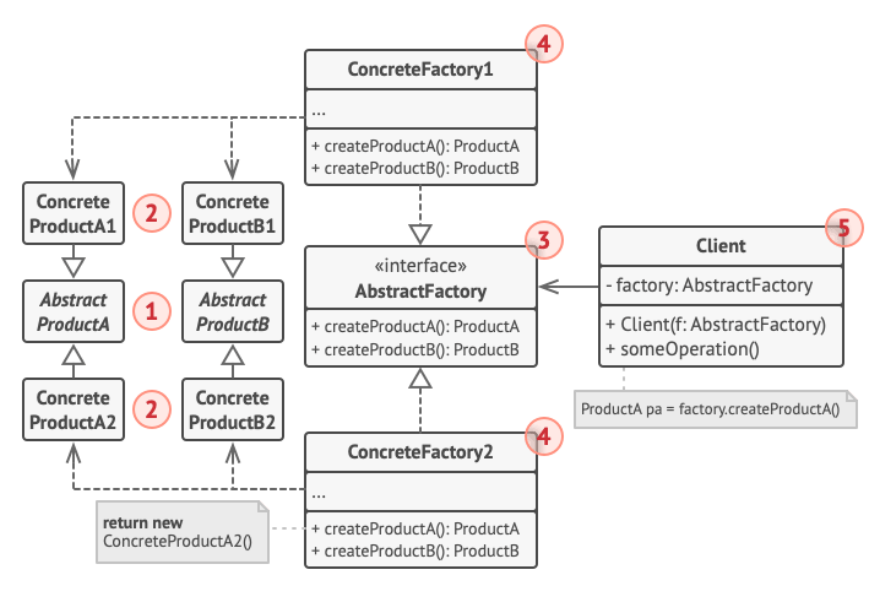


Figura 1.5 - Structura șablonului Factory Method

1. **Modele de proiectare structurale:** explică modul de asamblare a obiectelor și claselor în structuri mai mari, păstrând în același timp aceste structuri flexibile și eficiente.

* Adapter: Adaptorul este un model de design structural care permite obiectelor cu interfețe incompatibile să colaboreze.

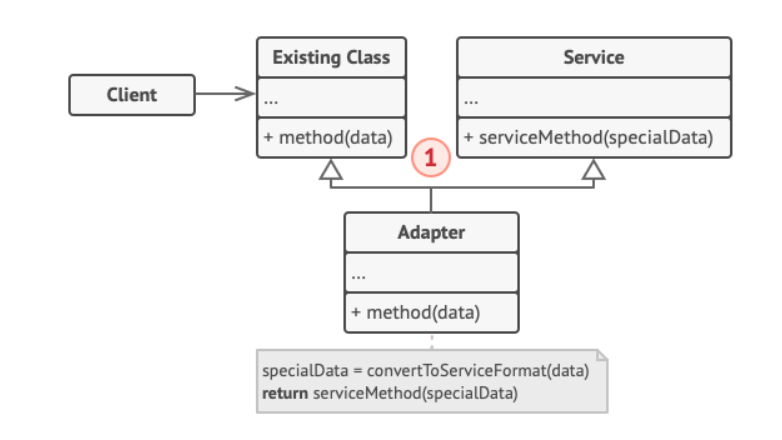


Figura 1.6 – Structura șablonului Adapter. Implementarea unei clase

* Bridge: Bridge este un model de proiectare structurală care vă permite să împărțiți o clasă mare sau un set de clase strâns legate în două ierarhii separate - abstracție și implementare - care pot fi dezvoltate independent una de cealaltă.

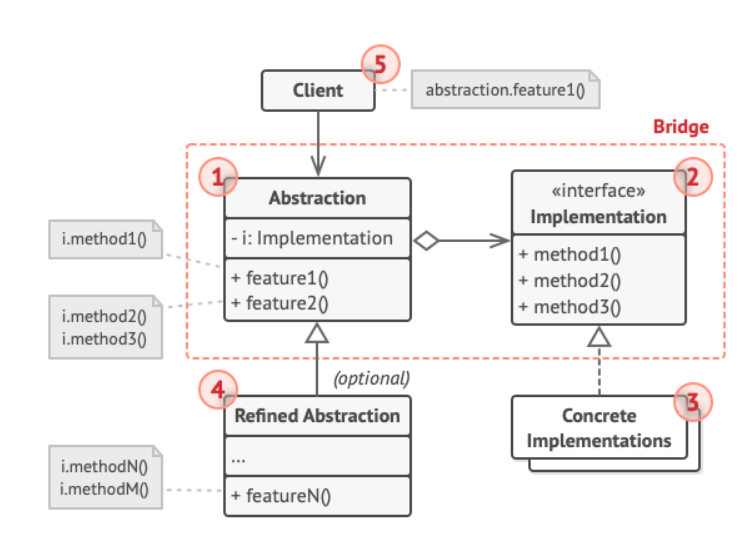


Figura 1.7 – Structura șablonului Bridge

* Composite: Compozitul este un model de proiectare structurală care vă permite să compuneți obiecte în structuri arborescente și apoi să lucrați cu aceste structuri ca și cum ar fi obiecte individuale.

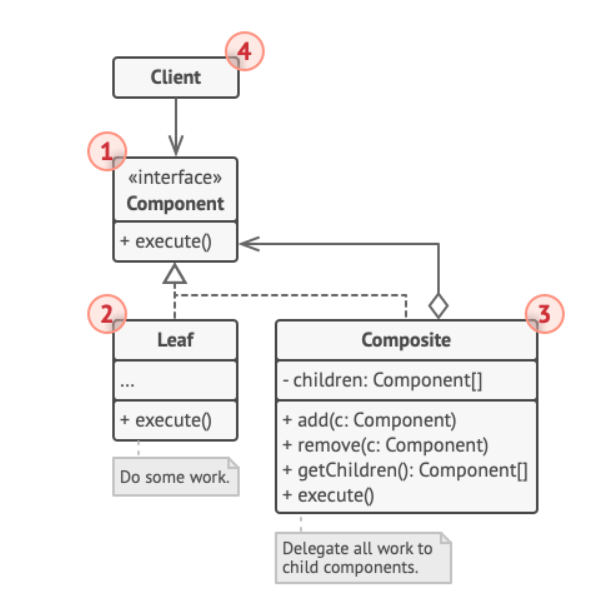


Figura 1.8 – Structura șablonului Composite

* Decorator: Decorator este un model de design structural care vă permite să atașați noi comportamente la obiecte prin plasarea acestor obiecte în interiorul obiectelor speciale care conțin comportamentele.

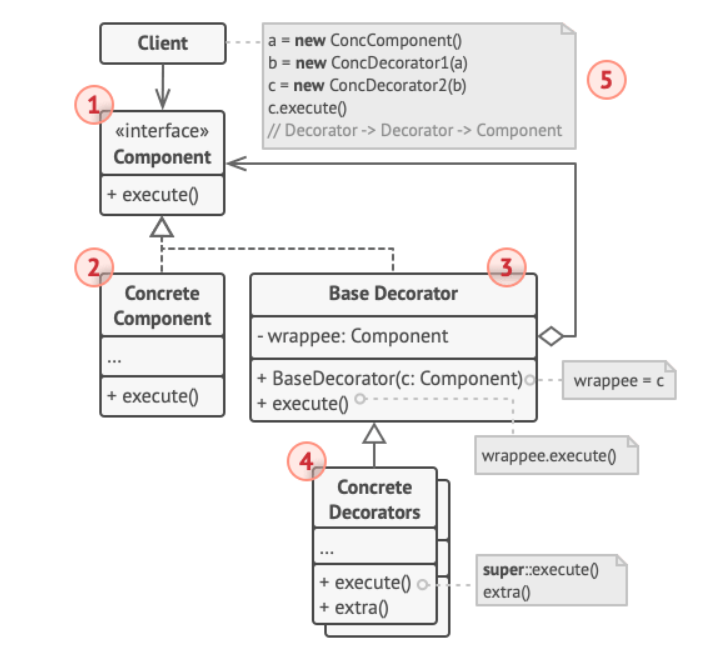


Figura 1.9 – Structura șablonului Decorator

* Facade: Fațada este un model de proiectare structurală care oferă o interfață simplificată la o bibliotecă, un cadru sau orice alt set complex de clase.

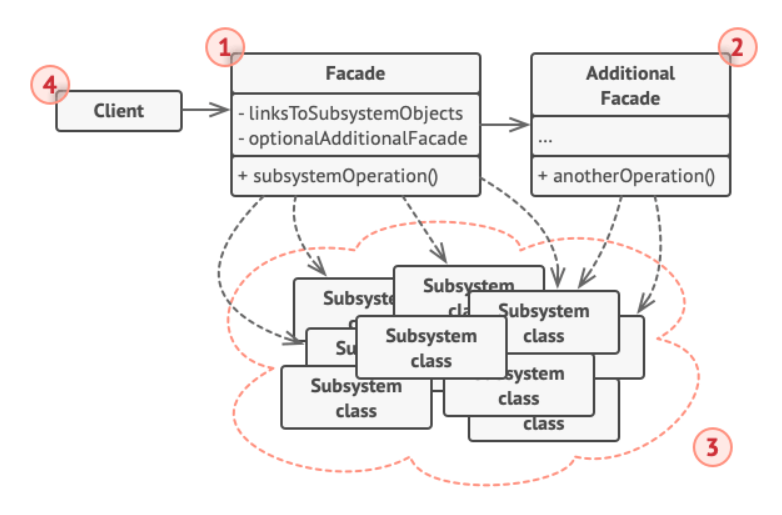


Figura 1.10 – Structura șablonului Facade

* Flyweight: Flyweight este un model de design structural care vă permite să încadrați mai multe obiecte în cantitatea disponibilă de RAM, partajând părți comune de stare între mai multe obiecte, în loc să păstrați toate datele în fiecare obiect.

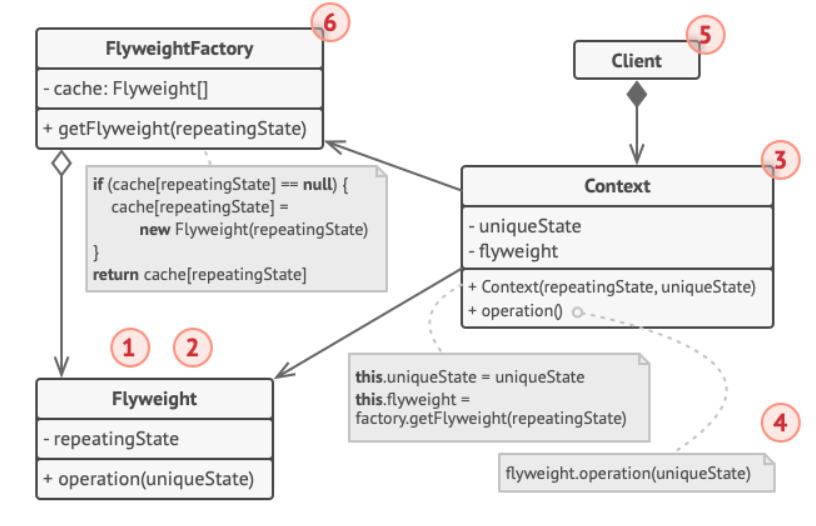


Figura 1.11 – Structura șablonului Flyweight

* Proxy: Proxy este un model de proiectare structurală care vă permite să oferiți un substitut sau un substituent pentru alt obiect. Un proxy controlează accesul la obiectul original, permițându-vă să efectuați ceva înainte sau după ce solicitarea ajunge la obiectul original.

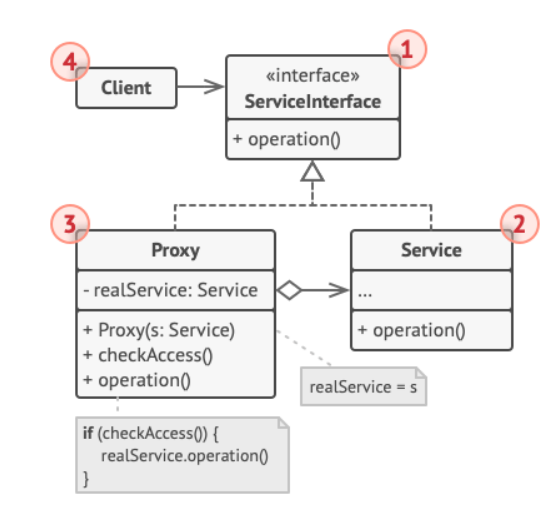


Figura 1.12 – Structura șablonului Proxy

1. **Modele de proiectare comportamentale:** Tiparele de design comportamental sunt preocupate de algoritmi și de atribuirea responsabilităților între obiecte.

* Chain of Responsibility: Chain of Responsibility este un model de design comportamental care vă permite să transmiteți cereri de-a lungul unui lanț de gestionari. La primirea unei cereri, fiecare handler decide fie să proceseze cererea, fie să o transmită următorului handler din lanț.

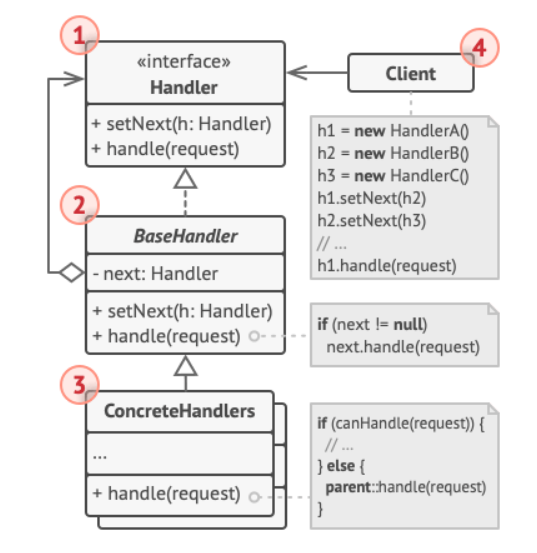


Figura 1.13 – Structura șablonului Chain of Responsibility

* Command: Comanda este un model de design comportamental care transformă o solicitare într-un obiect autonom care conține toate informațiile despre cerere. Această transformare vă permite să treceți cererile ca argumente de metodă, să întârziați sau să puneți în coadă execuția unei cereri și să suportați operațiuni care nu pot fi anulate.

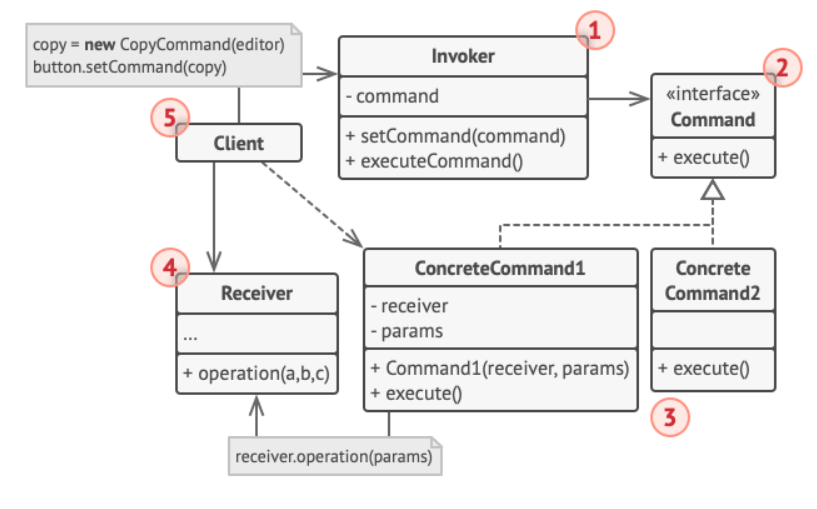


Figura 1.14 – Structura șablonului Command

* Iterator: Iteratorul este un model de design comportamental care vă permite să traversați elemente ale unei colecții fără a expune reprezentarea de bază (listă, stivă, arbore etc.).

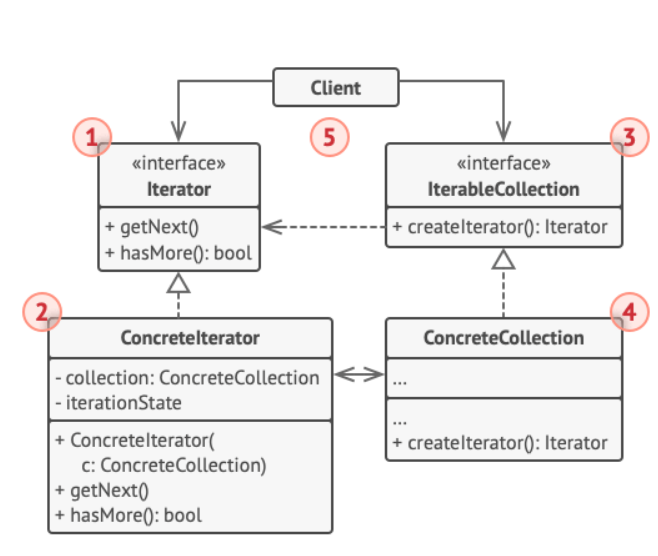


Figura 1.15 – Structura șablonului Iterator

* Mediator: Mediator este un model de design comportamental care vă permite să reduceți dependențele haotice dintre obiecte. Modelul restricționează comunicațiile directe între obiecte și le obligă să colaboreze numai prin intermediul unui obiect mediator.

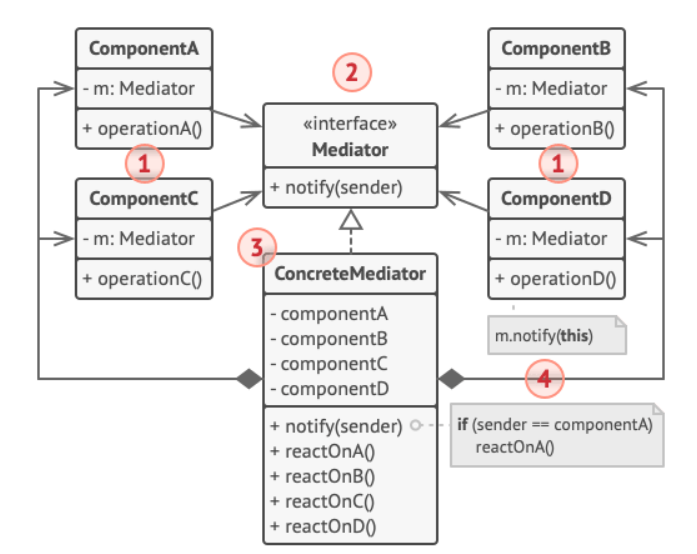


Figura 1.16 – Structura șablonului Mediator

* Observer: Observer este un model de design comportamental care vă permite să definiți un mecanism de abonament pentru a notifica mai multe obiecte despre orice eveniment care se întâmplă obiectului pe care îl observă.

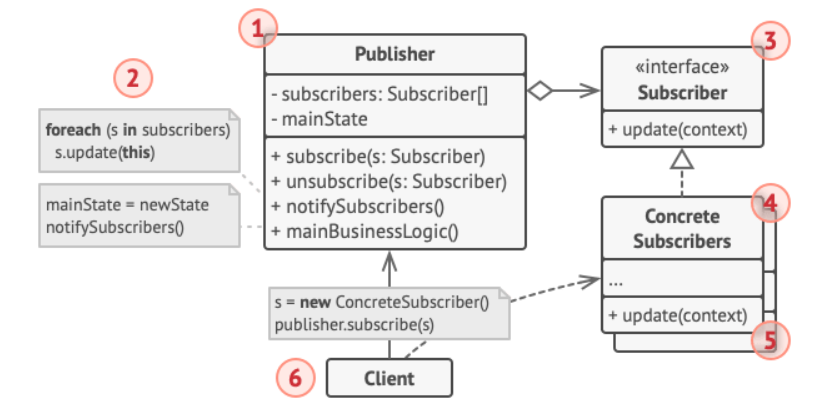


Figura 1.17 – Structura șablonului Observer

* Strategy: Strategia este un model de design comportamental care vă permite să definiți o familie de algoritmi, să puneți fiecare dintre ei într-o clasă separată și să faceți obiectele lor interschimbabile.fiecărui algoritm într-un obiect și utilizarea lor în mod înlocuibil.

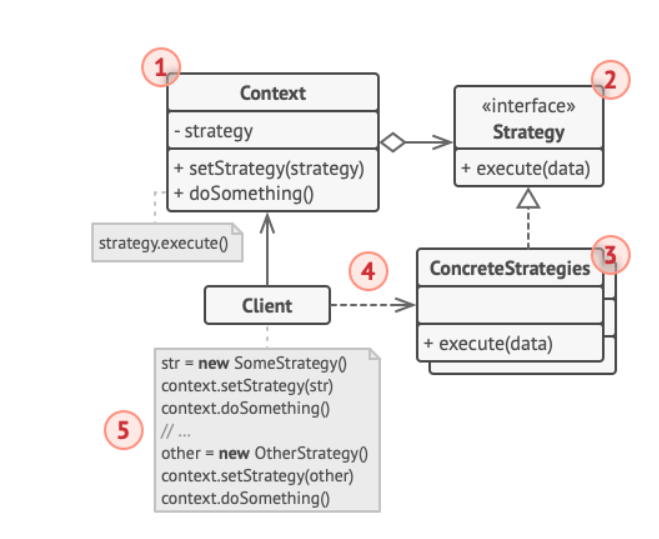


Figura 1.18 – Structura șablonului Strategy

Acestea sunt doar câteva exemple de modele de proiectare, iar lista completă conține mult mai multe patternuri și variante. Modelele de proiectare sunt soluții flexibile și recurente care ajută la rezolvarea problemelor comune de proiectare și implementare software, oferind un cadru conceptual și o abordare standardizată pentru construirea de aplicații robuste și extensibile.

1. **Implementarea aplicației.**

Pentru implementarea aplicației am ales Limbajul C#. Acest limbaj este un Orientat pe Obiect, cu ajutorul poți ușor implementa modelele de proiectare.

Conform obiectivului de bază a acestui proiect, am creat o aplicație de tip consolă, care interacționează cu o bază de date existentă din SQL Server. Aceasta bază de date ”HomeExpense”, conține tabelele de gestionare a cheltuielilor financiare. Ea a fost creată și adaptată la o aplicație web, iar cu inermediul aplicației de tip consolă ” EXPENSE TRACKER”, pot ușor gestiona datele din baza de date fără a accesa interfața web.

Expense Tracker, dispune de un funcțional obișnuit:

* afișează datele;
* editează datele;
* calculează suma totală a cheltuielilor.

Scopul principal în implementarea aplicației, a fost de a proiecta 6 modele:

* 2 modele creaționale;
* 2 modele structurale;
* 2 modele comportamentale.

Analizând toate tipurile modelelor de proiectare și ținând cont de principala condiție la crearea aplicației, am ales să implementez în aplicația Expense Tracker următoarele șabloane:

* Modele Creaționale: Factory și Builder;
* Modele Structurale: Adapter și Decorator;
* Modele Comportamentale: Observer și Strategy.

Pentru început, am creat o aplicație de tip consolă în Visual Studio”Expense Tracker”. După cum am menționat anterior, aplicația are responsabilitatea de a gestiona informația existentă din baza de date ”HomeExpense”. Respectiv, pentru a putea realiza conecțiunea dintre proiectul Expense Tracker și SQL Server, am instalat refrința necesare *System.Data.SqlClient*.

După ce am creat proiectul, am instalat packetul de date necesar, am început a adăuga fișiere .cs, deoarece nu este deloc flexibil și bine venit integrarea întregii aplicații într-un singur fișier.

Pentru a putea gestiona conexiunea și interogările către baza de date, am definit o clasă DatabaseManager. Această clasă, inițial avea doar funcționalul esențial de a se conecta la baza de date prin intermediul a două câmpuri private:

* ***connectionString:*** Un șir de caractere care conține informațiile necesare pentru a stabili conexiunea cu baza de date. În acest exemplu, se folosește un server numit "DESKTOP-7AGA2OD", o bază de date numită "HomeExpense" și se utilizează autentificare integrată (Integrated Security=True).
* ***connection:*** Un obiect de tip SqlConnection care va reprezenta conexiunea cu baza de date.

Clasa DatabaseManager oferă și două metode publice:

* ***OpenConnection():*** Această metodă deschide conexiunea către baza de date folosind metoda Open() a obiectului connection.
* ***CloseConnection():*** Această metodă închide conexiunea cu baza de date folosind metoda Close() a obiectului connection.

**public** **class** DatabaseManager

{

**private** **string** connectionString = "Data Source=DESKTOP-7AGA2OD;Initial Catalog=HomeExpense;Integrated Security=True";

**private** SqlConnection connection;

**public** SqlConnection Connection { **get** { **return** connection; } }

**public** DatabaseManager()

{

connection = new SqlConnection(connectionString);

}

**public** **void** OpenConnection()

{

connection.Open();

}

**public** **void** CloseConnection()

{

connection.Close();

}

}

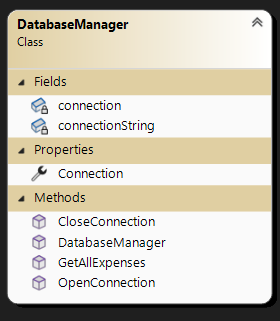


Figura 2.1 – Clasa DatabaseManager

Ulterior, după implementarea clasei pentru conexiunea cu SQL Server, am trecut la implementarea modelelor de proiectare. Am început cu crearea fișierelor. Fiecare fișier începe cu Expense\_ și respectiv numele modelul implementat în acel fișier.

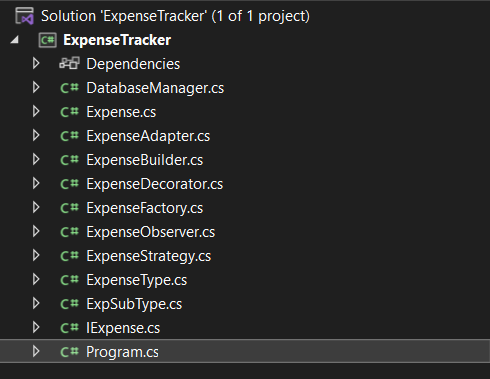


Figura 2.2 – Structura proiectului ExpenseTracker

În momentul implementării modelelor de proiectare, am decis să structurez apelarea metodelor în Main, prin adăugarea unui meniu, care va conține optiuni corespunzătoare fiecărui șablon implementat. Meniul Expense Tracker, conține 10 opțiuni, care sunt prezentate mai jos.

Console.WriteLine("Expense Tracker");

Console.WriteLine("----------------");

Console.WriteLine("1. View Expenses");

Console.WriteLine("2. Add Expense Category");

Console.WriteLine("3. Add Expense Subcategory");

Console.WriteLine("4. Modify Expense Category");

Console.WriteLine("5. Modify Expense Subcategory");

Console.WriteLine("6. Calculate Total Expenses");

Console.WriteLine("7. Add Expense");

Console.WriteLine("8. View Categories");

Console.WriteLine("9. View Subcategories");

Console.WriteLine("0. Exit");

Console.WriteLine("----------------");

Console.Write("Enter your choice: ");

Conform funcționalităților de bază, am început implementarea modelelor de proiectare.

* 1. **Implementarea șablonului Adapter**

Opțiunile 1,8 și 9, corespund cu implementarea șablonului **Adapter**, respectiv se apelează metodele declarate în acel context, care ne ajută la afișarea datelor existente din Baza de Date.

**case** "1":

*// View Expenses (Adapter Pattern)*

Console.WriteLine("Viewing Expenses:");

expenseAdapter.DisplayExpenses();

**break**;

**case** "8":

*// View Categories*

Console.WriteLine("Viewing Expense Categories:");

expenseAdapter.DisplayExpenseCategories();

**break**;

**case** "9":

*// View Subcategories*

Console.WriteLine("Viewing Expense Subcategories:");

expenseAdapter.DisplayExpenseSubcategories();

**break**;

Clasa ExpenseAdapter oferă o interfață simplificată pentru a accesa și afișa date dintr-o bază de date specificată prin intermediul unui obiect DatabaseManager. Acesta reprezintă un exemplu de utilizare a design pattern-ului Adapter pentru adaptarea interfeței bazei de date la nevoile specifice ale sistemului de urmărire a cheltuielilor.

În această clasă sunt implementate 3 metode de bază: DisplayExpenses, DisplayExpenseCategories, DisplayExpenseSubCategories.

Metoda DisplayExpenses() – afișează datele din tabelul AllExpense, adică toate cheltuielile împreună cu categorie, subcategorie, descriere, suma de bani, data și numele utilizatorului care a adaugat.

Metodele DisplayExpenseCategories() și DisplayExpenseSubCategories() – la fel afișează datele, doar că din alte tabele existente: ExpenseType și ExpSubType. Aceste metode generează categoriile și subcategoriile existente din baza de date.

**public** **class** ExpenseAdapter

{

//

**public** **void** DisplayExpenses()

{

//

}

**public** **void** DisplayExpenseCategories()

{

//

}

**public** **void** DisplayExpenseSubcategories()

{

//

}

Am atașat doar structura implementării modelului de proiectare Adapter.

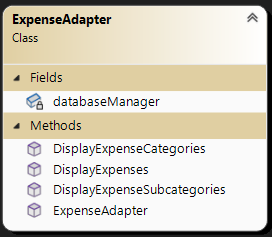


Figura 2.3 – Clasa ExpenseAdapter

* 1. **Implementarea șablonului** **Builder**

Implementarea șablonului Builder în cadrul clasei ExpenseBuilder este importantă pentru funcționalitățile de adăugare a categoriilor și subcategoriilor în sistemul de urmărire a cheltuielilor, conform cazurile 2 și 3 din meniul proiectului:

**case** "2":

*// Add Expense Category (Builder Pattern)*

Console.WriteLine("Adding Expense Category:");

expenseBuilder.AddExpenseCategory();

**break**;

**case** "3":

*// Add Expense Subcategory (Builder Pattern)*

Console.WriteLine("Adding Expense Subcategory:");

expenseBuilder.AddExpenseSubcategory();

**break**;

Prin implementarea șablonului Builder, se separă procesul de construcție a obiectelor de reprezentarea lor finală. În cazul nostru, avem nevoie să construim interogări SQL pentru adăugarea de categorii și subcategorii de cheltuieli în baza de date. Prin utilizarea metodelor AddExpenseCategory și AddExpenseSubcategory din clasa ExpenseBuilder, construcția interogărilor este separată de execuția acestora.

Prin intermediul metodelor GetNextCategoryId, GetNextSubcategoryId și ExecuteQuery, clasa ExpenseBuilder ascunde detaliile de implementare legate de obținerea ID-urilor și executarea interogărilor în baza de date. Aceasta oferă un nivel de abstractizare și securitate, evitând expunerea directă a acestor detalii către utilizatori.

Implementarea șablonului Builder în cadrul clasei ExpenseBuilder aduce multiple beneficii, cum ar fi separarea construcției de reprezentarea obiectelor, simplificarea procesului de construcție, flexibilitatea și extensibilitatea, precum și ascunderea detaliilor de implementare. Acestea contribuie la o arhitectură mai modulară, ușor de întreținut și extinsă în viitor.

**public** **class** ExpenseBuilder

{

**public** **void** AddExpenseCategory()

{

}

**private** **int** GetNextCategoryId()

{

}

**public** **void** AddExpenseSubcategory()

{

}

**private** **int** GetNextSubcategoryId()

{

}

**private** **void** ExecuteQuery(**string** query)

{

}

}

}

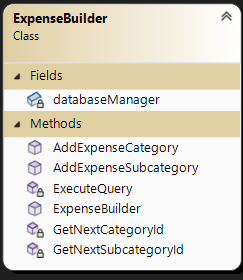


Figura 2.4 – Clasa ExpenseBuiler

* 1. **Implementarea șablonului Decorator**

În continuare, opțiunile 4 și 5, reprezintă modificarea Categoriilor și Subcategoriilor.

**case** "4":

*// Modify Expense Category (Decorator Pattern)*

Console.WriteLine("Modifying Expense Category:");

expenseDecorator.ModifyExpenseCategory();

**break**;

**case** "5":

*// Modify Expense Subcategory (Decorator Pattern)*

Console.WriteLine("Modifying Expense Subcategory:");

expenseDecorator.ModifyExpenseSubcategory();

**break**;

Conform acestor funcționalități de modificare a Categoriilor și a SubCategoriilor, am decis să implementez metodele necesară după modelul de proiectare **Decorator**, deoarece deținem deja o clasă de bază, ExpenseManager, care se ocupă de funcționalitatea principală de modificare a categoriilor și subcategoriilor. Prin implementarea șablonului Decorator, putem adăuga comportamente suplimentare înainte și după apelurile metodelor de modificare, fără a modifica direct clasa de bază.

Clasa ExpenseDecorator servește ca decorator pentru clasa ExpenseManager. Aceasta implementează interfața IExpenseManager, care definește metodele ModifyExpenseCategory() și ModifyExpenseSubcategory().

Decorarea funcționalității de modificare a categoriilor și subcategoriilor se realizează în interiorul metodelor ModifyExpenseCategory() și ModifyExpenseSubcategory() ale clasei ExpenseDecorator.

După apelul metodei din ExpenseManager, decoratorul poate efectua și alte acțiuni suplimentare, cum ar fi afișarea unui mesaj de finalizare a decorării ("Decoration Complete").

**public** **interface** IExpenseManager

{

}

**public** **class** ExpenseManager : IExpenseManager

{

**public** ExpenseManager(**string** connectionString)

{

}

**public** **void** ModifyExpenseCategory()

{

}

**public** **void** ModifyExpenseSubcategory()

{

}

**private** **void** ExecuteQuery(**string** query)

{

}

}

**public** **class** ExpenseDecorator : IExpenseManager

{

*expenseManager.ModifyExpenseCategory();*

}

**public** **void** ModifyExpenseSubcategory()

{

*expenseManager.ModifyExpenseSubcategory();*

}

}

}

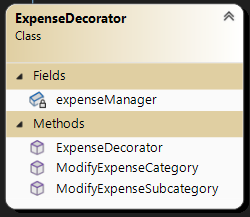


Figura 2.5 – Clasa ExpenseDecorator

* 1. **Implementarea șablonului Strategy**

Următorul model de proiectare implementat este **Strategy**, metoda căruia este apelată în opțiune nr 6:

**case** "6":

*// Calculate Total Expenses (Strategy Pattern)*

Console.WriteLine("Calculating Total Expenses:");

**decimal** totalExpenses = expenseStrategy.CalculateTotalExpenses();

Console.WriteLine($"Total Expenses: {totalExpenses}");

**break**

Implementarea șablonului Strategy în codul dat se referă la clasa ExpenseStrategy, care este responsabilă de calcularea cheltuielilor totale. Acest șablon permite schimbarea algoritmului de calcul al cheltuielilor totale fără a modifica clasa ExpenseStrategy în sine.

Metoda CalculateTotalExpenses() aplică un algoritm specific pentru a calcula suma totală a cheltuielilor. În acest caz, se folosește metoda Sum() a clasei Enumerable pentru a calcula suma cheltuielilor din lista allExpenses. Prin utilizarea expresiei lambda expense => expense.amount, se specifică că se dorește suma valorilor câmpului amount pentru fiecare obiect expense din lista.

Importanța implementării șablonului Strategy în acest caz constă în faptul că se oferă flexibilitatea de a schimba algoritmul de calcul al cheltuielilor totale fără a modifica direct clasa ExpenseStrategy. În viitor, dacă se dorește să se schimbe metoda de calcul, se poate crea o nouă clasă care implementează interfața IExpenseCalculationStrategy și să se înlocuiască strategia curentă în ExpenseStrategy cu noua implementare. Acest lucru permite extinderea și personalizarea algoritmului de calcul al cheltuielilor în funcție de cerințele și logica specifică.

**public** **class** ExpenseStrategy

{

**private** DatabaseManager databaseManager;

**public** ExpenseStrategy(DatabaseManager databaseManager)

{

**this**.databaseManager = databaseManager;

}

**public** **decimal** CalculateTotalExpenses()

{

**var** allExpenses = databaseManager.GetAllExpenses();

**decimal** totalExpenses = allExpenses.Sum(expense => expense.amount);

**return** totalExpenses;

}

}

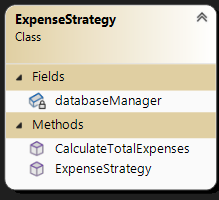


Figura 2.6 – Clasa ExpenseStrategy

* 1. **Implementarea șabloanelor Observer și Factory**

Anterior, am demonstrat implementarea a 4 șabloane, respectiv au mai ramas 2: **Observer** și **Factory.** Aceste două șabloane corespund undei singure opțiune: Adăugarea Cheltuielilor.

**case** "7":

*// Add Expense (Factory and Observer Patterns)*

Console.WriteLine("Adding Expense:");

expenseFactory.AddExpense();

**break**;

Aici este vizibilă apelarea metodei AddExpense(), din ExpenseFactory, dar metoda implementată în Observer este apelată în interiorul șablonului Factory.

Clasa ExpenseFactory acționează ca un Factory pentru crearea obiectelor de cheltuieli (Expense). Metoda AddExpense() este responsabilă pentru crearea și adăugarea unei noi înregistrări de cheltuială în baza de date. Utilizatorul introduce diferite informații despre cheltuială (tipul de cheltuială, subtipul, descrierea, suma etc.), iar apoi obiectul Expense este creat și adăugat în baza de date.

Implementarea șablonului Factory în acest caz oferă o separare clară între logica de creare a obiectelor de cheltuieli și logica de adăugare în baza de date. Acest lucru face codul mai modular și ușor de întreținut. De asemenea, prin utilizarea unei fabrici, este ușor să adăugăm și să gestionăm noi tipuri de cheltuieli în viitor, fără a afecta codul existent.

**public** **class** ExpenseFactory

{

**public** **void** AddExpense()

{

Console.Write("Enter Amount: ");

**decimal** amount = **decimal**.Parse(Console.ReadLine());

**if** (!expenseObserver.CheckExpenseAmount(amount))

{

Console.WriteLine("Expense cannot be added. Amount exceeds the maximum limit.");

**return**;

}

}

}

În secvența de cod de mai sus, am păstrat doar metoda AddExpenses, și apelarea implementării șablonului Observer, care este responsabil de stabilirea unei limite de cheltuieli.

Clasa ExpenseObserver acționează ca un Observer pentru monitorizarea cheltuielilor și impune o restricție asupra valorii maxim admise pentru o cheltuială. Metoda CheckExpenseAmount() verifică dacă suma cheltuielii depășește limita maximă și returnează rezultatul verificării.

Implementarea șablonului Observer în acest caz permite adăugarea unui mecanism de observare și notificare atunci când o cheltuială depășește limita maximă admisă. Aceasta permite separarea logicii de verificare a restricției de codul responsabil de adăugarea cheltuielilor. Prin utilizarea șablonului Observer, putem adăuga și alte observatoare sau restricții în viitor fără a afecta clasa ExpenseFactory.

**public** **class** ExpenseObserver

{

**public** **bool** CheckExpenseAmount(**decimal** amount)

{

**if** (amount > 10000)

{

Console.WriteLine("Restriction: Amount exceeds the maximum limit.");

**return** **false**;

}

**else**

{

Console.WriteLine("Expense amount is within the limit.");

**return** **true**;

}

Implementarea șablonului Factory facilitează crearea și adăugarea de obiecte de cheltuieli în baza de date, în timp ce implementarea șablonului Observer permite observarea și aplicarea de restricții asupra cheltuielilor înainte de adăugarea lor. Ambele șabloane contribuie la modularitate, flexibilitate și decuplare în proiectul de urmărire a cheltuielilor.

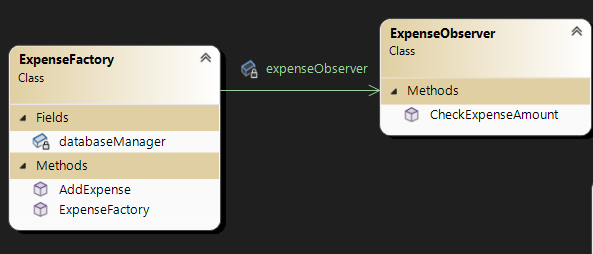


Figura 2.7 – Asocierea dintre clasele ExpenseFactory și ExpenseObserver

Conform structurei proiectului, a claselor existente, am creat prin intermediul instrumentelor din Visual Studio, Diagrama de clase a întregii aplicații. Diagramele de clase oferă o reprezentare vizuală a claselor, relațiilor și interacțiunilor acestora în cadrul aplicației. Prin generarea diagramei de clase, putem obține o imagine de ansamblu a structurii aplicației și a modului în care clasele interacționează între ele.

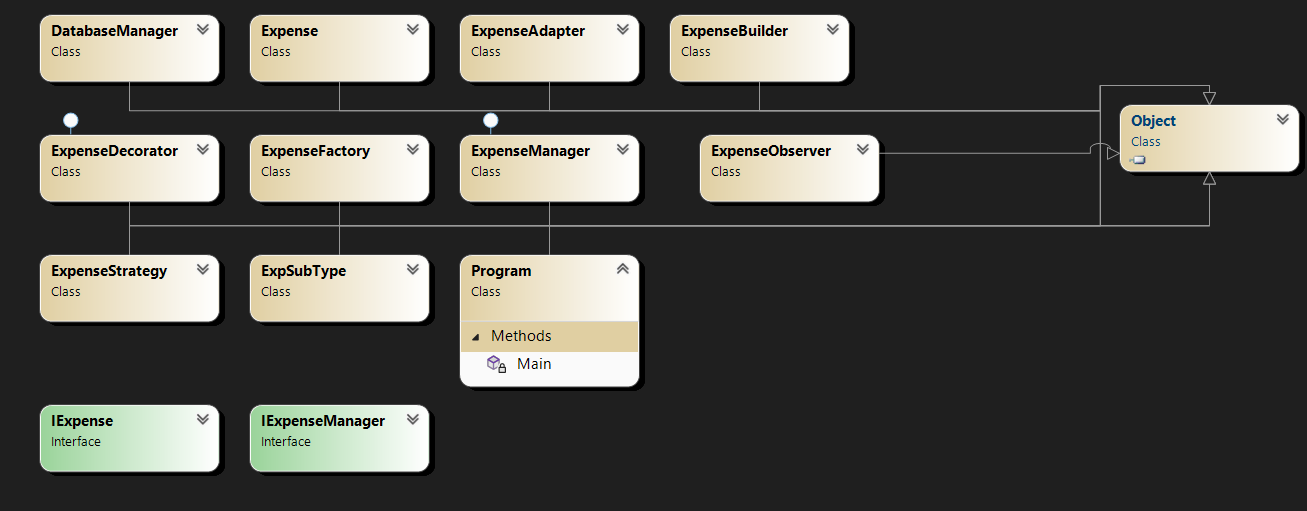


Figura 2.8 – Diagrama de clasa a aplicației Expense Tracker

**Concluzie**

Realizarea acestui proiect de an la disciplina "Tehnici și Mecanisme de Proiectare Software" a fost o experiență valoroasă și provocatoare. Prin implementarea aplicației Expense Tracker, care interacționează cu baza de date HomeExpense din SQL Server, am avut oportunitatea de a aplica diferite șabloane de proiectare și de a dobândi cunoștințe importante în domeniul proiectării software.

Unul dintre cele mai semnificative șabloane de proiectare implementate în această aplicație a fost **Adapter Pattern**. Am folosit acest șablon pentru a adapta interfața bazei de date existente la interfața necesară în aplicație, în vederea afișării cheltuielilor. Prin intermediul acestui șablon, am reușit să abstractizez detaliile de implementare ale bazei de date și să ofer o interfață comună și ușor de utilizat pentru utilizatorii aplicației.

Un alt șablon important implementat în aplicație a fost **Decorator Pattern**. Am utilizat acest șablon pentru a extinde funcționalitățile existente ale clasei ExpenseManager, adăugând capacitatea de a modifica categoriile și subcategoriile de cheltuieli. Prin intermediul Decorator Pattern, am obținut o soluție flexibilă și extensibilă, care ne-a permis să adăugăm noi comportamente la obiectele existente fără a modifica structura lor.

Importanța acestor șabloane de proiectare constă în faptul că m-au ajutat să obțin un cod mai modular, mai flexibil și mai ușor de întreținut. Prin separarea responsabilităților și utilizarea abstracțiunilor adecvate, am obținut un design scalabil și adaptabil la schimbări viitoare.

Pe parcursul acestui proiect, am acumulat cunoștințe solide despre diferitele șabloane de proiectare și despre modul în care pot fi aplicate într-un context real. Am înțeles importanța planificării și proiectării corecte a aplicațiilor software, precum și beneficiile pe care le aduc utilizarea șabloanelor de proiectare în dezvoltarea software.

În ansamblu, acest proiect m-a ajutat să înțeleg mai bine importanța unei proiectări software bune și m-a familiarizat cu aplicarea practică a șabloanelor de proiectare. Aceste cunoștințe și experiențe acumulate în timpul acestui proiect mă vor ajuta în cariera mea de dezvoltator software și mă vor face mai eficient în abordarea și rezolvarea problemelor complexe de proiectare.

**Bibliografie:**

1. Design Patterns, [Resursă electronică], [Citat 30.05.2023] - Link-ul de acces: <https://refactoring.guru/design-patterns>
2. Limbajul de Programare C#, [Resursă electronică], [Citat 01.06.2023] - Link-ul de acces: <https://ro.wikipedia.org/wiki/C_sharp>
3. ASP.NET, [Resursă electronică], [Citat 01.06.2023] - Link-ul de acces: [https://www.tutorialspoint.com/asp.net/asp.net\_introduction.htm#](https://www.tutorialspoint.com/asp.net/asp.net_introduction.htm)
4. Generator de diagrame, [Resursă electronică], [Citat 03.06.2023] - Link-ul de acces: <https://betterprogramming.pub/generating-class-diagrams-for-net-core-c4913db9398b>