Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Ingineria Software și Automatică

RAPORT

Lucrare de laborator nr.4

Disciplina: Tehnici și Mecanisme de Proiectare Software

Tema: Șabloane comportamentale

A efectuat: st. gr. TI-202, Aliona Tatar

A verificat: asist. univ., Mihai Gaidău

Chişinău 2023

**Modele comportamentale de proiectare**

Modelele comportamentale de proiectare sunt soluții de proiectare care abordează interacțiunea și comunicarea dintre obiecte sau clase într-un sistem software. Aceste modele se concentrează pe definirea responsabilităților și interacțiunilor dintre obiecte pentru a îmbunătăți flexibilitatea, reutilizarea și întreținerea codului. Iată câteva modele comportamentale de proiectare utilizate frecvent:

**Modelul Observer**: Modelul Observer stabilește o dependență de tip unu-la-mulți între obiecte, astfel încât atunci când un obiect (subiectul) își schimbă starea, toate obiectele dependente (observatorii) sunt notificate și actualizate automat. Acest model este util pentru implementarea sistemelor de gestionare a evenimentelor, arhitecturilor de tip publică-subscribe și pentru decuplarea subiectului de observatori.

**Modelul Strategy**: Modelul Strategy definește o familie de algoritmi interschimbabili și încapsulează fiecare algoritm într-o clasă separată. Acesta permite clienților să selecteze și să utilizeze dinamic algoritmi diferiți în timpul execuției, în funcție de contextul sau cerințele specifice. Acest model promovează reutilizarea codului, flexibilitatea și separarea preocupărilor.

**Modelul Command:** Modelul Command încapsulează o cerere ca obiect, decuplând astfel expeditorul cererii de destinatar. Acesta permite parametrizarea clienților cu diferite cereri, coada sau jurnalizarea cererilor și suportul pentru operațiuni de anulare. Acest model este util în implementarea sistemelor de meniuri, procesării tranzacțiilor și arhitecturilor bazate pe evenimente.

**Modelul Iterator:** Modelul Iterator oferă o modalitate de a accesa elementele unui obiect agregat secvențial, fără a expune structura sa internă. Acesta definește o interfață comună pentru parcurgerea diferitelor tipuri de colecții, permițând clienților să traverseze și să acceseze elementele într-un mod consistent. Acest model simplifică parcurgerea colecțiilor și oferă o modalitate standard de accesare a elementelor.

**Modelul Chain of Responsibility:** Modelul Chain of Responsibility creează o lanț de obiecte, fiecare având șansa de a trata o cerere. Cererea este trecută de-a lungul lanțului până când este tratată sau ajunge la sfârșitul lanțului. Acest model decuplează expeditorul și destinatarul unei cereri, permite tratarea dinamică a cererilor și oferă flexibilitate în adăugarea sau eliminarea tratatorilor.

**Elaborarea sarcinii**

**Modelul Observer**:

Interfata IObserver: Aceasta interfata defineste contractul pentru observatori. Clasa User implementeaza aceasta interfata, iar metoda Update este apelata de catre subiect atunci cand are loc o modificare.

public interface IObserver

{

void Update(string bookTitle);

}

Clasa Library: Aceasta clasa reprezinta subiectul (publisher). Aceasta mentine o lista de observatori si ofera metode pentru abonare (Subscribe), dezabonare (Unsubscribe) si notificare (NotifyObservers). Atunci cand o schimbare are loc in biblioteca (adugare sau eliminare de carti), subiectul notifica observatorii prin apelarea metodei Update pentru fiecare observator inregistrat.

public class Library

{

private List<IObserver> observers = new List<IObserver>();

public void AddBookToLibrary(string bookTitle)

{

Console.WriteLine($"Book added to the library: {bookTitle}");

NotifyObservers(bookTitle);

}

public void RemoveBookFromLibrary(string bookTitle)

{

Console.WriteLine($"Book removed from the library: {bookTitle}");

NotifyObservers(bookTitle);

}

public void NotifyObservers(string bookTitle)

{

foreach (var observer in observers)

{

observer.Update(bookTitle);

}

}

public void Subscribe(IObserver observer)

{

observers.Add(observer);

}

public void Unsubscribe(IObserver observer)

{

observers.Remove(observer);

}

}

Clasa User: Aceasta clasa implementeaza interfata IObserver si reprezinta observatorul (subscriber). Cand este notificat de subiect, prin apelul metodei Update, observatorul User poate executa anumite actiuni, cum ar fi afisarea unui mesaj in consola.

public class User : IObserver

{

private string name;

public User(string name)

{

this.name = name;

}

public void Update(string bookTitle)

{

Console.WriteLine($"User {name}: Book {bookTitle} has been updated.");

}

}

In esenta, implementarea patternului Observer permite ca unul sau mai multi observatori (utilizatori) sa fie inregistrati la subiectul (biblioteca) si sa fie notificati automat atunci cand se produce o schimbare in biblioteca. Astfel, observatorii pot reactiona la aceasta schimbare in functie de logica lor specifica, fara ca subiectul sa cunoasca detaliile despre observatori.

**Modelul Strategy**

Interfața IBookValidationStrategy: Această interfață definește contractul pentru diferitele strategii de validare a cărților. Aceasta ar putea avea o metodă ValidateBook care primește un titlu de carte și returnează rezultatul validării.

public interface IBookValidationHandler

{

bool ValidateBook(string bookTitle);

void SetNextHandler(IBookValidationHandler handler);

}

Clasa Library: În clasa Library, în loc să utilizezi un handler pentru validarea cărților, vei folosi o strategie de validare a cărților. Adaugă un membru de tip IBookValidationStrategy în clasa Library și un setter pentru a seta strategia de validare.

public class Library

{

private List<IObserver> observers = new List<IObserver>();

private IBookValidationStrategy validationStrategy;

public void SetValidationStrategy(IBookValidationStrategy strategy)

{

validationStrategy = strategy;

}

public void AddBookToLibrary(string bookTitle)

{

if (validationStrategy.ValidateBook(bookTitle))

{

Console.WriteLine($"Book added to the library: {bookTitle}");

NotifyObservers(bookTitle);

}

else

{

Console.WriteLine($"Book {bookTitle} is not valid and cannot be added to the library.");

}

}

}

Implementarea diferitelor strategii: Poți crea clase concrete care implementează interfața IBookValidationStrategy și definesc logica de validare specifică. De exemplu, poți avea o clasă TitleValidationStrategy și o clasă AuthorValidationStrategy:

public class TitleValidationStrategy : IBookValidationStrategy

{

public bool ValidateBook(string bookTitle)

{

if (bookTitle.Length >= 5)

{

return true;

}

else

{

return false;

}

}

}

public class AuthorValidationStrategy : IBookValidationStrategy

{

public bool ValidateBook(string bookTitle)

{

if (bookTitle.Contains("by"))

{

return true;

}

else

{

return false;

}

}

}

Utilizarea Modelului Strategy: În metoda Main a programului tău, poți crea o instanță a subiectului Library și alegi o strategie de validare a cărților prin intermediul metodei SetValidationStrategy. Apoi, adaugi cărți în bibliotecă și vezi cum se utilizează strategia de validare specificată:

static void Main(string[] args)

{

// Create a library

var library = new Library();

// Set up the strategy

var titleValidationStrategy = new TitleValidationStrategy();

library.SetValidationStrategy(titleValidationStrategy);

// Add books to the library

library.AddBookToLibrary("Book");

library.AddBookToLibrary("Book");

Console.ReadLine();

}

Prin utilizarea Modelului Strategy în acest mod, poate schimba comportamentul de validare a cărților dinamic, înlocuind strategia de validare in functie de nevoi.

**Modelul Chain of Responsibility**

Am adăugat interfața `IBookValidationHandler` care definește contractul pentru handler-urile de validare a cărților. Această interfață are metodele `ValidateBook` pentru a valida un titlu de carte și `SetNextHandler` pentru a seta următorul handler în lanț.

public interface IBookValidationHandler

{

bool ValidateBook(string bookTitle);

void SetNextHandler(IBookValidationHandler handler);

}

Am creat două clase concrete care implementează interfața `IBookValidationHandler`: `TitleValidationHandler` și `AuthorValidationHandler`. Fiecare handler are o implementare specifică pentru validarea unui titlu de carte. Dacă un handler nu poate valida titlul de carte, acesta trece cererea către următorul handler din lanț.

// Concrete Handler 1

public class TitleValidationHandler : IBookValidationHandler

{

private IBookValidationHandler nextHandler;

public bool ValidateBook(string bookTitle)

{

if (bookTitle.Length >= 5)

{

return true;

}

else if (nextHandler != null)

{

return nextHandler.ValidateBook(bookTitle);

}

else

{

return false;

}

}

public void SetNextHandler(IBookValidationHandler handler)

{

nextHandler = handler;

}

}

// Concrete Handler 2

public class AuthorValidationHandler : IBookValidationHandler

{

private IBookValidationHandler nextHandler;

public bool ValidateBook(string bookTitle)

{

if (bookTitle.Contains("by"))

{

return true;

}

else if (nextHandler != null)

{

return nextHandler.ValidateBook(bookTitle);

}

else

{

return false;

}

}

public void SetNextHandler(IBookValidationHandler handler)

{

nextHandler = handler;

}

}

În clasa `Library`, am adăugat metoda `SetValidationHandler` pentru a seta handler-ul de validare. În metoda `AddBookToLibrary`, înainte de a adăuga o carte în bibliotecă, apelăm metoda `ValidateBook` a handler-ului curent pentru a verifica dacă cartea este validă în funcție de criteriile specifice ale handler-ului. Dacă cartea trece de toate validările, aceasta este adăugată în bibliotecă și notificăm observatorii. În caz contrar, cartea nu este adăugată în bibliotecă.

public class Library

{

private List<IObserver> observers = new List<IObserver>();

private IBookValidationHandler validationHandler;

public void SetValidationHandler(IBookValidationHandler handler)

{

validationHandler = handler;

}

public void AddBookToLibrary(string bookTitle)

{

if (validationHandler.ValidateBook(bookTitle))

{

Console.WriteLine($"Book added to the library: {bookTitle}");

NotifyObservers(bookTitle);

}

else

{

Console.WriteLine($"Book {bookTitle} is not valid and cannot be added to the library.");

}

}

public void RemoveBookFromLibrary(string bookTitle)

{

Console.WriteLine($"Book removed from the library: {bookTitle}");

NotifyObservers(bookTitle);

}

public void NotifyObservers(string bookTitle)

{

foreach (var observer in observers)

{

observer.Update(bookTitle);

}

}

public void Subscribe(IObserver observer)

{

observers.Add(observer);

}

public void Unsubscribe(IObserver observer)

{

observers.Remove(observer);

}

}

În metoda `Main`, am creat o instanță a subiectului (`Library`) și două instanțe de observatori (`User`). Observatorii sunt abonați la bibliotecă și înregistrăm handler-urile de validare (`TitleValidationHandler` și `AuthorValidationHandler`) în lanțul de responsabilitate al bibliotecii prin intermediul metodei `SetValidationHandler`.

static void Main(string[] args)

{

// Create a library

var library = new Library();

// Create users

var user1 = new User("user1");

var user2 = new User("user2");

// Subscribe users to the library

library.Subscribe(user1);

library.Subscribe(user2);

// Set up the chain of responsibility

var titleValidationHandler = new TitleValidationHandler();

var authorValidationHandler = new AuthorValidationHandler();

titleValidationHandler.SetNextHandler(authorValidationHandler);

library.SetValidationHandler(titleValidationHandler);

// Add a valid book to the library

library.AddBookToLibrary("Book");

// Add an invalid book to the library

library.AddBookToLibrary("Book");

// Remove a book from the library

library.RemoveBookFromLibrary("Book");

Console.ReadLine();

}

Astfel, în acest cod, am implementat cu succes principiul comportamental Chain of Responsibility pentru a valida cărțile înainte de a le adăuga în bibliotecă.

**Concluzie**

In laboratorul 4 am imlementat diferite pattern-uri comportamentale în proiectul meu despre bibliotecă oferind o structură și o flexibilitate semnificativă în gestionarea comportamentului și interacțiunii între obiecte.

Observer: Pattern-ul Observer facilitează comunicarea între subiect și observatori într-un mod decuplat. Acesta permite subiectului (Library) să notifice automat observatorii (User) despre schimbările de stare, permițând observatorilor să reacționeze în funcție de nevoile lor.

Strategy: Modelul Strategy permite schimbarea dinamică a comportamentului în funcție de strategia selectată. Prin intermediul strategiilor de validare a cărților, poți modifica logica de validare într-un mod flexibil și modular, fără a modifica codul substanțial. Acesta favorizează decuplarea și extensibilitatea în gestionarea validării cărților.

Chain of Responsibility: Pattern-ul Chain of Responsibility permite procesarea succesivă și flexibilă a unor cereri sau acțiuni prin intermediul unui lanț de handler-i. Prin utilizarea acestui model, poți configura un lanț de handler-i care să gestioneze validarea cărților în ordinea specificată. Acesta oferă o modalitate elegantă de a separa și de a organiza logica de validare în componente individuale, fără a încărca sender-ul (Library) cu detaliile specifice ale validării.

Prin utilizarea acestor pattern-uri comportamentale, obții o arhitectură modulară, decuplată și extensibilă pentru gestionarea comportamentului în proiectul tău despre bibliotecă. Aceste pattern-uri promovează principii precum decuplarea, modularitatea, extensibilitatea și flexibilitatea, oferindu-ți un design ușor de întreținut.