Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Ingineria Software și Automatică

**Raport**

**Tema:** Cercetarea și studierea Design Pattern-urilor.

Crearea unei Biblioteci.

**A efectuat:**  Cozac Anastasia, TI-202

**Conducător proiect de curs:** asistent universitar, Gaidău Mihai

**Chișinău 2023**

**Сuprins**

**Introducere**………………………..…………………………………………...............2

**1.** Analiza domeniului problemei ...................................................................................4

1.1 Ce sunt *Design Patterns*?......………….....................................................5

1.2 Care sunt scopurile *Design Patterns* ? . ……………................................6

**2**. Catalog : *Design Patterns*...........................................................................................9

2.1 Creational ………………………............................................................9

2.1.1 *Beneficiile pattern-urilor creaționale*..........................................11

2.2 Structural ................…………..……………….......................................13

2.2.1 *Beneficiile pattern-urilor structurale*..........................................15

2.3 Behavioral.................……………………………...................................17

2.3.1 *Beneficiile pattern-urilor creaționale*..........................................20

**3**. Descrierea principiului de lucru al tematicii alese.......................................................22

3.1 Principiile SOLID ...............……………………...................………….....22

3.2 Implementarea *Design Patterns*………………………………..………..25

**Concluzie**……………………………………………………………......………..........30

**Bibliografie**………………………………………………………………......…...........32

**Introducere**

Design Patternurile sunt instrumente esențiale în dezvoltarea software, oferind o metodologie și o abordare structurată pentru rezolvarea problemelor comune întâlnite în procesul de dezvoltare. Ele reprezintă soluții testate și validate, dezvoltate de experți în domeniu, care pot fi aplicate într-o varietate de contexte și limbaje de programare.

Unul dintre cele mai importante seturi de principii care stau la baza Design Patternurilor este cunoscut sub numele de principiile SOLID. Aceste principii sunt ghiduri fundamentale pentru crearea de software modular, extensibil și ușor de întreținut. Ele facilitează dezvoltarea unei arhitecturi solide și flexibile, care poate rezista schimbărilor și adăugărilor ulterioare.

Primul principiu, Single Responsibility Principle (SRP), susține că o clasă ar trebui să aibă o singură responsabilitate și să fie responsabilă doar de un singur aspect al sistemului. Acest principiu promovează coeziunea și evită încălcarea separării logicii într-un sistem.

Al doilea principiu, Open-Closed Principle (OCP), afirmă că clasele trebuie să fie deschise pentru extindere, dar închise pentru modificare. Acest lucru înseamnă că trebuie să putem adăuga noi funcționalități prin extinderea claselor existente, fără a modifica codul existent.

Liskov Substitution Principle (LSP) subliniază importanța unei ierarhii de clase coerente, în care subtipurile pot fi utilizate în locul tipurilor de bază, fără a afecta corectitudinea programului. Acest principiu se bazează pe relațiile "este un" și ajută la construirea unui cod robust și modular.

Interface Segregation Principle (ISP) promovează ideea că interfețele nu ar trebui să fie prea mari sau prea specifice. Ele ar trebui să fie concepute astfel încât să satisfacă nevoile exacte ale modulelor care le utilizează. Acest principiu ajută la evitarea dependențelor inutile și la reducerea cuplării dintre module.

Dependency Inversion Principle (DIP) sugerează că modulele ar trebui să depindă de abstracții și nu de implementări concrete. Acest principiu facilitează flexibilitatea și permite schimbarea facilă a implementărilor fără a afecta modulele care le folosesc.

De exemplu, unul dintre cele mai cunoscute Design Patternuri este Singleton, care asigură că o clasă are o singură instanță și oferă un punct global de acces la acea instanță. Aceasta este utilă în situațiile în care avem nevoie de o singură instanță a unei clase în întregul sistem.

Un alt exemplu este Patternul Observer, care permite obiectelor să se aboneze și să primească actualizări de la un subiect. Acesta facilitează comunicarea între obiecte și permite o abordare modulară și extensibilă.

Design Patternurile și principiile SOLID reprezintă instrumente cheie pentru dezvoltatorii de software, oferindu-le un set de soluții testate și validate pentru probleme comune. Prin aplicarea acestora, se poate crea un software robust, extensibil și ușor de întreținut, care să îndeplinească cerințele complexe ale utilizatorilor și să faciliteze dezvoltarea continuă.

**1. Analiza domeniului problemei.**

**1.1 Ce sunt *Design Patterns* ?**

Design patterns, în traducere "modele de proiectare", sunt soluții recurente și testate în timp pentru problemele comune întâlnite în dezvoltarea software. Acestea sunt abstracții ale experienței acumulate de dezvoltatori și designeri în domeniul programării și reprezintă soluții generice pentru problemele specifice.

Un design pattern descrie o abordare specifică pentru structurarea și organizarea codului într-un mod flexibil, modular și ușor de înțeles. Ele oferă o metodă clară de rezolvare a problemelor comune întâlnite în dezvoltarea software și promovează principii precum reutilizarea, extensibilitatea și dezlipirea componentelor.

Există mai multe categorii de design patterns, printre care:

1. **Pattern-uri de creare (Creational Patterns):** Acestea se concentrează pe modul în care obiectele sunt create, oferind modalități flexibile de instanțiere a obiectelor. Exemple de astfel de pattern-uri includ Singleton (asigură că există o singură instanță a unei clase în întregul sistem), Factory (oferă o interfață comună pentru crearea obiectelor) și Builder (construiește obiecte complexe pas cu pas).
2. **Pattern-uri de structură** **(Structural Patterns):** Acestea se concentrează pe relațiile și structura obiectelor, oferind modalități de organizare a codului într-un mod flexibil și eficient. Exemple de astfel de pattern-uri includ Adapter (permite comunicarea între două clase cu interfețe incompatibile), Decorator (adăugarea de comportamente suplimentare la un obiect existent) și Proxy (ofere un obiect de înlocuire pentru a controla accesul la un alt obiect).3.
3. **Pattern-uri de comportament** **(Behavioral Patterns):** Acestea se concentrează pe comportamentul și interacțiunea dintre obiecte, oferind modalități de gestionare a fluxului de execuție și comunicare între obiecte. Exemple de astfel de pattern-uri includ Observer (oferă o modalitate de notificare a obiectelor despre modificările de stare), Strategy (permite selecția dinamică a unui algoritm) și Iterator (oferă un mod de acces secvențial la elementele unei colecții).
4. **Pattern-uri arhitecturale:**

* Model-View-Controller (MVC): Separă logica de afaceri (modelul), prezentarea (vizualizarea) și interacțiunea cu utilizatorul (controlerul).
* Model-View-ViewModel (MVVM): Similar cu MVC, dar se concentrează pe separarea clară între model, vizualizare și logica de vizualizare.
* Dependency Injection (DI): Permite injectarea dependențelor externe într-o clasă pentru a facilita testarea și flexibilitatea.

1. **Pattern-uri de optimizare:**

* Lazy Initialization: Amână crearea sau inițializarea unui obiect până când acesta este efectiv necesar, pentru a evita încărcarea inutilă a resurselor.
* Memoization: Salvează rezultatele costisitoare ale unei funcții și le returnează direct atunci când se primesc aceleași argumente din nou, pentru a evita recalcularea lor.

1. **Pattern-uri de acces la date:**

* Repository: Oferă o interfață comună pentru a accesa și manipula datele într-o sursă de date, ascunzând detalii specifice de implementare.
* Unit of Work: Încapsulează mai multe operații de modificare a datelor într-o singură tranzacție coezivă.

1. **Pattern-uri de securitate:**

* Strategy: Permite selecția dinamică a unei strategii de securitate în funcție de context, cum ar fi algoritmul de criptare folosit.
* Chain of Responsibility: Permite construirea unei lanțuri de obiecte care poate trata cereri succesive, fiecare obiect decidând dacă o prelucrează sau o trece mai departe în lanț.

1. **Pattern-uri de sincronizare și concurență:**

* Monitor: Oferă mecanisme pentru a asigura accesul exclusiv la resurse partajate între firele de execuție.
* Semaphore: Gestionează un număr limitat de permisiuni pentru a controla accesul concurent la resurse.

1. **Pattern-uri de interfață utilizator (UI):**

* Model-View-Presenter (MVP): Separă logica de prezentare a interfeței utilizator (prezenterul) de componenta vizuală a interfeței (vizualizarea) și modelul de date.
* Model-View-ViewModel (MVVM): Similar cu MVP, dar se concentrează pe legarea bidirecțională între model, vizualizare și logica de vizualizare.

1. **Pattern-uri de extensibilitate:**

* Plugin: Permite integrarea facilă a funcționalității suplimentare prin încărcarea și utilizarea de module externe (plugin-uri).
* Decorator: Permite adăugarea de funcționalități suplimentare la un obiect existent, fără a-l modifica direct.

1. **Pattern-uri de teste:**

* Mock: Simulează comportamentul obiectelor reale pentru a facilita testarea unitară a componentelor.
* Page Object: Abstractizează interacțiunea cu interfețele utilizator (UI) în teste automate, pentru a face teste mai ușor de menținut și de înțeles.

1. **Pattern-uri de gestionare a evenimentelor:**

* Observer: Oferă o modalitate de notificare automată a obiectelor interesate de modificările de stare ale unui alt obiect.
* Mediator: Definește o componentă intermediară care gestionează comunicarea între diverse obiecte, reducând dependențele directe dintre ele.

Design patterns nu sunt soluții definitive și nu trebuie utilizate mecanic în toate situațiile. Ele sunt recomandate atunci când se întâlnește o problemă specifică pentru care a fost identificat un pattern potrivit și când se urmărește crearea unui cod mai modular, ușor de înțeles și ușor de întreținut. Utilizarea design patterns poate contribui la dezvoltarea unei arhitecturi software mai solidă și la evitarea problemelor comune întâlnite în dezvoltare.

**1.2 Care sunt scopurile și beneficiile *Design Patterns* ?**

Scopurile Design Patterns sunt variate și esențiale în dezvoltarea software. Acestea includ furnizarea de soluții eficiente și testate pentru probleme comune în proiectarea software, promovarea reutilizabilității și modularității codului, facilitarea comunicării între membrii echipei de dezvoltare, crearea unui vocabular comun și a unei abordări standardizate în industria IT.

Design Patterns ajută la reducerea dependențelor strânse între componente și module, oferind astfel flexibilitate și extensibilitate în dezvoltarea de software. Ele contribuie la crearea unor sisteme robuste, ușor de întreținut și adaptabile la schimbările cerințelor. De asemenea, Design Patterns încurajează dezvoltarea de cod clar, coerent și ușor de înțeles, facilitând colaborarea în cadrul echipei și creșterea eficienței în procesul de dezvoltare.

De ce ar trebui să învățăm Design Patterns? Adevărul este că ai putea lucra ca programator mulți ani fără să cunoști niciun pattern. Mulți oameni fac exact asta. Totuși, chiar și în această situație, s-ar putea să implementezi unele patternuri fără să știi măcar despre ele. Atunci de ce ar trebui să îți petreci timpul învățându-le?

Design Patterns reprezintă un set de instrumente cu soluții testate și validate pentru probleme comune în proiectarea software. Chiar dacă nu te confrunți niciodată cu aceste probleme, cunoașterea patternurilor este utilă deoarece te învață cum să rezolvi tot felul de probleme folosind principii de design orientat pe obiecte.

Design Patterns definesc un limbaj comun pe care tu și colegii tăi îl puteți utiliza pentru a comunica mai eficient. Poți spune: "Oh, folosește doar un Singleton pentru asta", și toată lumea va înțelege ideea din spatele sugestiei tale. Nu este nevoie să explici ce este un singleton dacă cunoști patternul și numele său.

Cunoașterea Design Patterns îți permite să folosești soluții deja testate și validate, fără a reinventa roata. De exemplu, dacă întâlnești o situație în care ai nevoie de o singură instanță a unei clase în întregul sistem, poți aplica patternul Singleton, care îți oferă o soluție eficientă și ușor de înțeles.

Un alt beneficiu al învățării Design Patterns este că te ajută să dezvolți abilități de gândire abstractă și modulară. Înțelegerea și aplicarea patternurilor îți dezvoltă capacitatea de a vedea problemele într-un mod mai general și de a găsi soluții mai elegante și mai flexibile.

De asemenea, cunoașterea Design Patterns te face un programator mai eficient și mai productiv. Patternurile oferă soluții testate și validate, care au fost folosite și rafinate de către experți în domeniu. Aceste soluții te scutesc de a mai pierde timpul încercând să inventezi propriile soluții și te ajută să te concentrezi mai mult pe problemele specifice ale aplicației pe care o dezvolți.

Pe lângă aceste beneficii, învățarea Design Patterns te ajută să devii parte a unei comunități de programatori care împărtășesc aceeași limbă și aceleași abordări de proiectare. Poți învăța de la alții și poți contribui cu propriile idei și experiențe. Astfel, te poți conecta la o rețea de profesioniști și poți beneficia de schimbul de cunoștințe și perspective.

Învățarea Design Patterns aduce multiple beneficii. Acestea includ comunicarea eficientă între membrii echipei, utilizarea soluțiilor testate și validate, dezvoltarea abilităților de gândire abstractă și modulară, creșterea eficienței și productivității, precum și integrarea într-o comunitate de programatori. Deși poți lucra ca programator fără cunoștințe despre Design Patterns, acestea îți oferă un set de instrumente valoroase pentru rezolvarea problemelor de proiectare software.

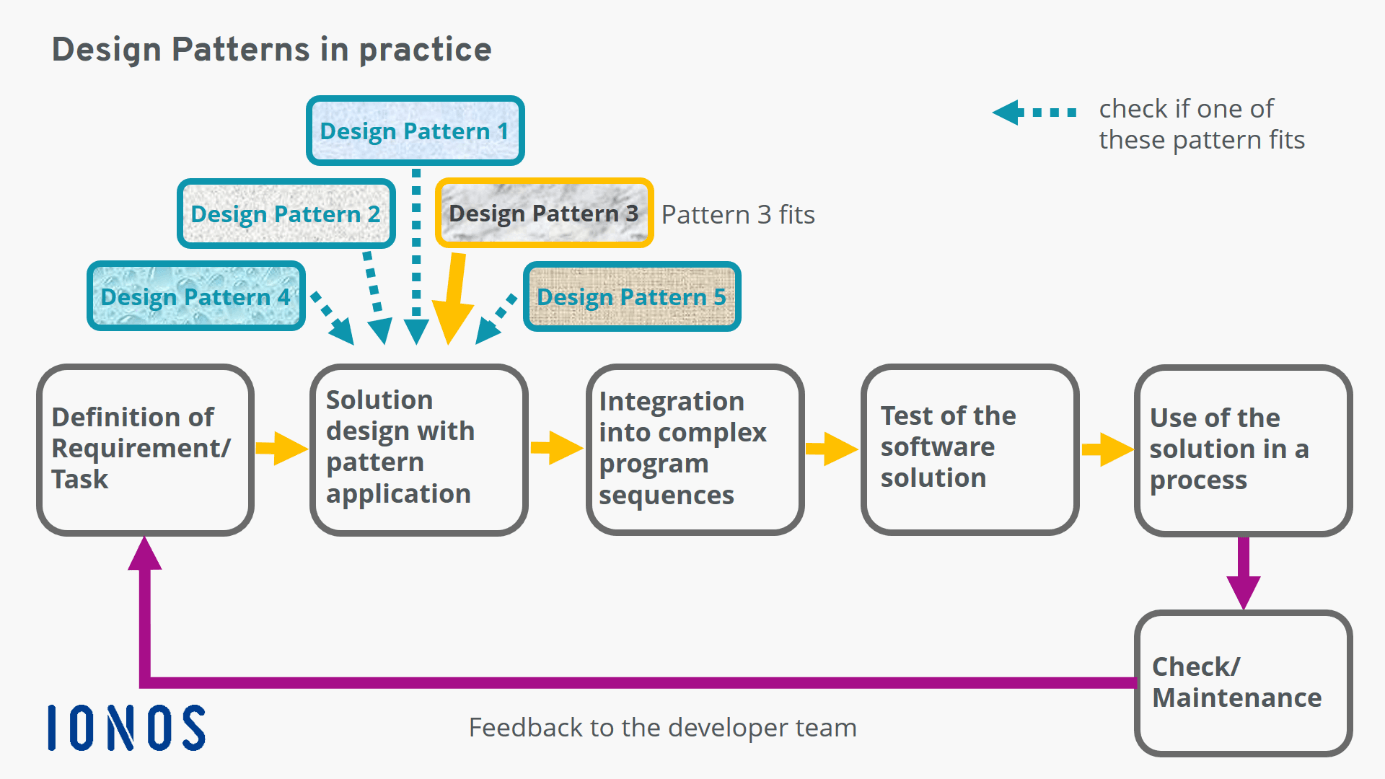


Fig.1.1 - utilizarea Design Patterns în practică

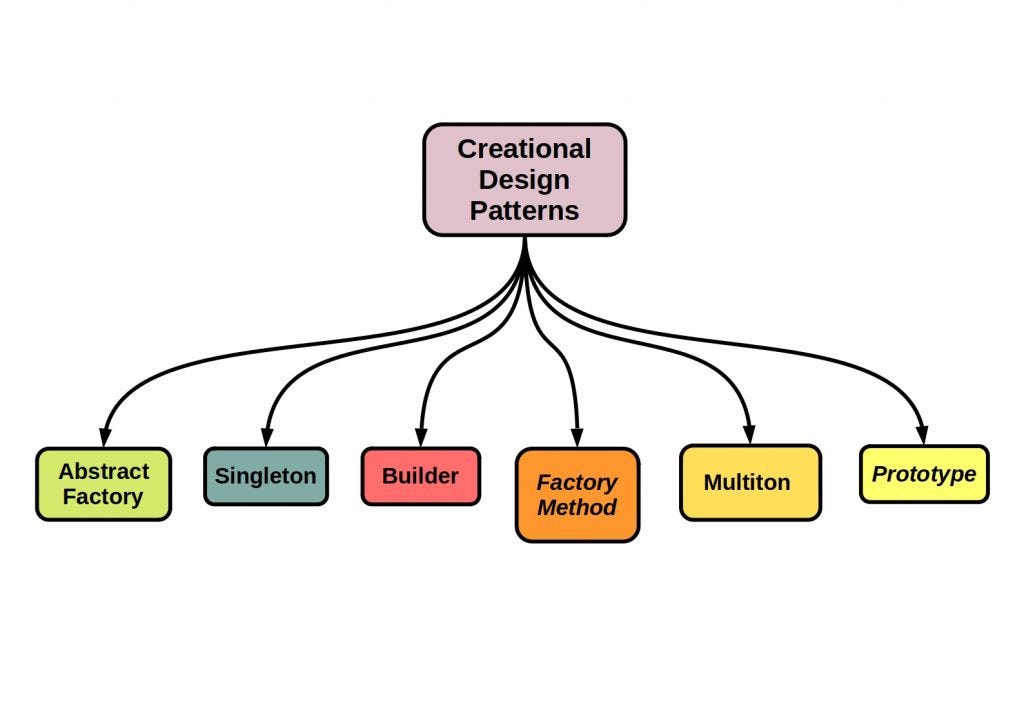
**2. Catalog: *Design Patterns.***

**2.1 Creational**

Design Patterns Creational reprezintă un subset de patternuri care se concentrează pe crearea obiectelor într-un mod flexibil și eficient. Aceste patternuri oferă soluții pentru diferite situații de creare a obiectelor și asigură că procesul este decuplat de clasele concrete și de detalii de implementare.

1. Factory Method este un pattern creational care permite crearea de obiecte într-o clasă de bază, dar permite claselor derivate să decidă tipul exact de obiect creat. Aceasta oferă o metodă de fabrică abstractă care este implementată în clasele derivate pentru a crea obiecte specifice. Astfel, se realizează o decuplare între codul care solicită obiecte și clasele concrete care le creează.
2. Singleton este un pattern creational care asigură că o clasă are o singură instanță și oferă un punct global de acces la acea instanță. Acest pattern este util în situațiile în care avem nevoie de o singură instanță a unei clase în întregul sistem. Singleton oferă un mecanism prin care se poate accesa instanța unică, indiferent de locul din cod.
3. Builder este un pattern creational care permite construirea unui obiect complex pas cu pas. Acesta separă procesul de construcție de reprezentarea finală a obiectului și permite utilizatorilor să construiască obiecte complexe într-un mod clar și flexibil. Builder poate fi util atunci când avem nevoie de crearea unor obiecte cu mulți parametri sau configurări variate.
4. Abstract Factory este un pattern creational care furnizează o interfață pentru crearea unei familii de obiecte conexe, fără a specifica clasele concrete ale obiectelor. Acest pattern permite crearea de familii de obiecte care funcționează împreună și oferă o modalitate de înlocuire a acestora în mod transparent. Abstract Factory încapsulează logica creării obiectelor și permite o creștere ușoară a funcționalității.
5. Prototype este un pattern creational care permite crearea de obiecte noi prin clonarea unui obiect existent (prototip). Acesta oferă o modalitate eficientă de creare a obiectelor când procesul de inițializare este costisitor sau complex. Prototipul servește ca bază pentru crearea de obiecte noi, iar fiecare obiect nou poate fi personalizat în funcție de necesități.

Setul de patternuri creationale oferă un cumul de soluții eficiente și flexibile pentru crearea obiectelor într-un mod decuplat și modular. Ele permit o creare mai clară și mai extensibilă a obiectelor, facilitând dezvoltarea de software modular, ușor de întreținut și adaptabil la schimbări de cerințe. Prin aplicarea adecvată a acestor patternuri, dezvoltatorii pot obține beneficii în ceea ce privește reutilizabilitatea, eficiența și flexibilitatea codului.

 Fig.2.1 - componența Design Pattern Creational

**2.1.1 Beneficiile pattern-urilor creaționale**

Pattern-urile creaționale oferă numeroase beneficii în dezvoltarea software-ului, contribuind la crearea unor sisteme flexibile, modulare și ușor de întreținut. Fiecare pattern crețional aduce avantaje specifice, spre exemplu :

1. *Factory Method* contribuie la decuplarea între codul care solicită obiecte și clasele concrete care le creează, oferind astfel o flexibilitate crescută. Clasa de bază definește o interfață comună pentru crearea obiectelor, în timp ce clasele derivate pot implementa metoda de fabrică în funcție de necesități. De asemenea contribuie la extensibilitatea sistemului, deoarece noi clase concrete pot fi adăugate fără a afecta codul existent. Astfel, este ușor să se aducă modificări sau să se introducă noi tipuri de obiecte într-o aplicație.
2. *Singleton* asigură existența unei singure instanțe a unei clase în întregul sistem. Aceasta este utilă în situațiile în care avem nevoie de o instanță globală și accesibilă în diferite părți ale codului. Simplifică gestionarea resurselor comune sau a obiectelor costisitoare, deoarece instanța unică poate fi partajată și utilizată eficient de către toate componentele sistemului. Asigură coerența și consistența datelor, deoarece accesul la o singură instanță garantează că obiectele manipulate sunt aceleași în toate contextele.
3. *Builder* permite construirea obiectelor complexe pas cu pas, oferind un control mai mare asupra procesului de construcție. Astfel, este ușor să se creeze obiecte cu mulți parametri sau configurări variate. Îmbunătățește claritatea și lizibilitatea codului, deoarece separă logica de construcție a obiectului de codul de utilizare. Astfel, codul client poate fi simplificat și concentrat pe utilizarea obiectului final. Facilitează crearea de obiecte imutabile sau parțial imutabile, deoarece procesul de construcție este separat de reprezentarea finală a obiectului.
4. *Abstract Factory* oferă o interfață comună pentru crearea unei familii de obiecte conexe, fără a specifica clasele concrete. Astfel, permite schimbul transparent al implementărilor și încurajează dezvoltarea orientată pe interfețe. Sporește flexibilitatea și extensibilitatea sistemului, deoarece permite adăugarea ușoară a noi familii de obiecte. Prin simpla adăugare a unei noi fabrici abstracte și implementarea corespunzătoare, se poate extinde funcționalitatea aplicației. Încurajează principiul deschis-închis, deoarece adăugarea de noi tipuri de obiecte nu afectează codul client existent. Acesta rămâne neschimbat, interacționând doar cu interfețele abstracte.
5. *Prototype* facilitează simplificarea creării de obiecte noi prin clonarea unui prototip existent. Acest lucru poate fi util în situațiile în care crearea unui obiect nou implică o inițializare costisitoare sau complexă. Are un impact semnificativ la îmbunătățirea performanței, deoarece clonarea unui obiect este adesea mai eficientă decât crearea unuia nou. Astfel, se reduce timpul și resursele necesare pentru inițializare. Flexibilitatea în generarea de obiecte dinamic, deoarece prototipurile pot fi personalizate în funcție de necesități. Astfel, se obține un control mai mare asupra creării obiectelor.

**2.2 Structural**

Design Patterns Structural reprezintă un subset al Design Patterns, care se concentrează pe modul în care obiectele și clasele sunt organizate într-un sistem software. Aceste patternuri abordează aspecte precum compoziția, relațiile și structura obiectelor pentru a crea soluții flexibile și extensibile. În continuare, vom explora câteva dintre patternurile structurale de bază, oferind o descriere a fiecăruia.

1. Adapter Pattern convertește interfața unei clase într-o altă interfață, astfel încât să poată colabora cu alte clase care au interfețe diferite. Acesta oferă o modalitate de a conecta componente incompatibile și facilitează reutilizarea codului existent.
2. Bridge Pattern separă abstractizarea de implementare, permițându-le să evolueze independent. Prin intermediul acestui pattern, o ierarhie de clase poate fi combinată cu o altă ierarhie, extinzând astfel flexibilitatea și modularitatea sistemului.
3. Composite Pattern permite tratarea unui grup de obiecte într-un mod similar cu un obiect individual. Acesta definește o ierarhie de clase compuse din obiecte simple și compuse, facilitând manipularea lor în mod uniform și eficient.
4. Decorator Pattern oferă o modalitate flexibilă de a adăuga comportamente suplimentare la un obiect existent, fără a modifica structura de bază a acestuia. Acesta utilizează agregarea în locul moștenirii și permite extinderea funcționalității obiectelor într-un mod dinamic.
5. Facade Pattern oferă o interfață simplificată pentru a accesa un subsistem complex, ascunzând detaliile de implementare și oferind o metodă coerentă de interacțiune. Acest pattern simplifică utilizarea și gestionarea subsistemului în ansamblu.
6. Flyweight Pattern optimizează utilizarea memoriei prin partajarea cât mai multor date posibile între obiecte. Acesta este util în situațiile în care crearea unui număr mare de obiecte are un impact semnificativ asupra performanței și resurselor sistemului.
7. Proxy Pattern furnizează o înlocuire sau o reprezentare a unui obiect pentru a controla accesul la acesta. Acesta poate fi utilizat pentru a oferi securitate, gestionare a resurselor sau pentru a amâna crearea unui obiect costisitor până când este solicitat în mod explicit.

Fiecare pattern aduce beneficii specifice în proiectarea și dezvoltarea software-ului, oferind soluții eficiente și flexibile pentru organizarea și gestionarea obiectelor și claselor. Prin utilizarea acestor patternuri, dezvoltatorii pot crea sisteme mai ușor de întreținut, extensibile și scalabile, contribuind astfel la dezvoltarea unui cod de calitate și a unor soluții software robuste.

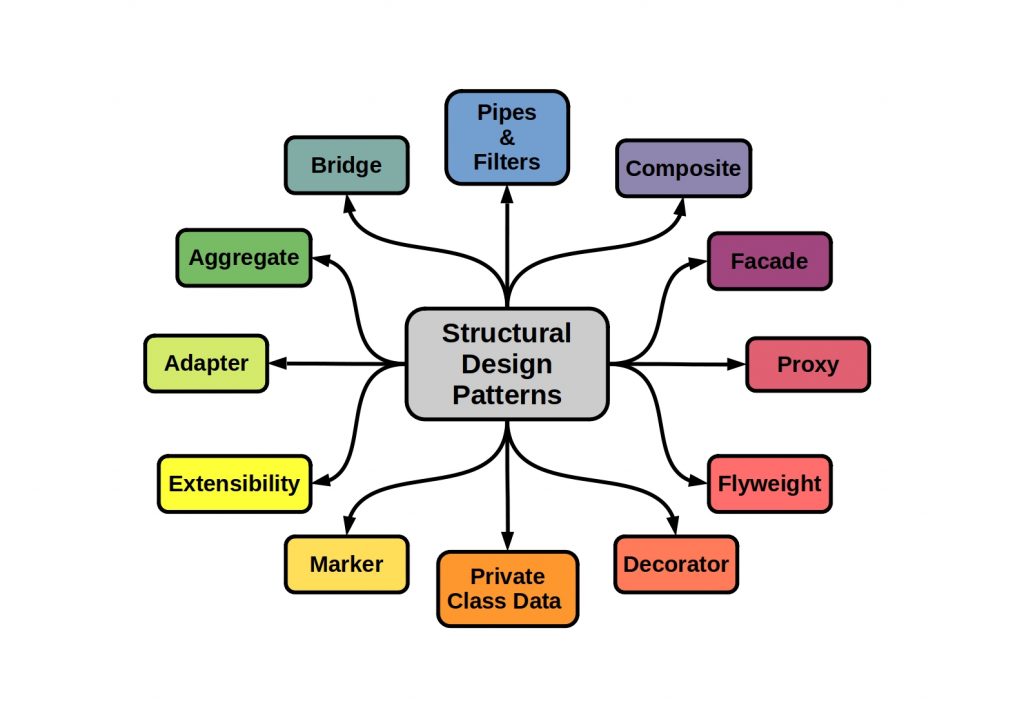


Fig.2.2 - componența Design Pattern Structural

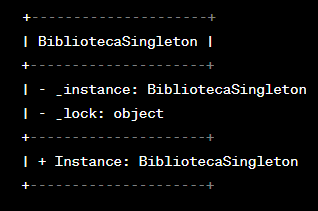


Fig.3.1 – Diagrama UML pentru Singleton

* Clasa BibliotecaSingleton este singura clasă din sablonul Singleton.
* Atributul privat \_instance reprezintă instanța unică a clasei BibliotecaSingleton.
* Atributul privat \_lock este utilizat pentru sincronizarea accesului la instanță.
* Metoda publică Instance permite obținerea instanței unice a clasei BibliotecaSingleton.



Fig.3.2 – Diagrama UML pentru Builder

* Clasa Carte reprezintă obiectul pe care dorim să îl construim folosind Builder.
* Clasa CarteBuilder este responsabilă pentru construirea obiectului Carte.
* Atributul privat \_carte reține starea curentă a obiectului în construcție.
* Metodele publice SetTitlu, SetAutor, SetAnPublicare permit setarea diferitelor proprietăți ale obiectului.
* Metoda publică Build returnează obiectul Carte finalizat.

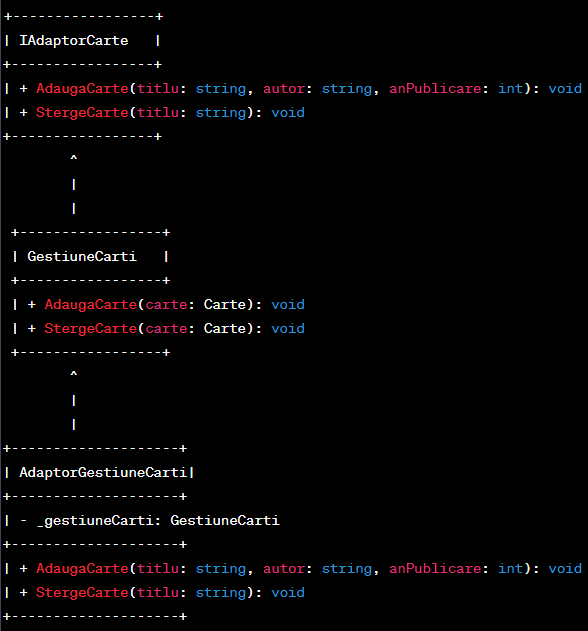


Fig.3.3 – Diagrama UML pentru Adapter

* Interfața IAdaptorCarte definește metodele comune pentru adăugarea și ștergerea de cărți.
* Clasa GestiuneCarti reprezintă sistemul existent pentru gestionarea cărților.
* Clasa AdaptorGestiuneCarti servește ca adaptor între IAdaptorCarte și GestiuneCarti, transformând apelurile către IAdaptorCarte în apeluri către GestiuneCarti.

**2.2.1 Beneficiile pattern-urilor structurale**

Design Patterns Structural aduc numeroase beneficii în proiectarea și dezvoltarea sistemelor software. Fiecare pattern structural oferă avantaje distincte și soluții pentru diferite aspecte ale arhitecturii software. Să explorăm acum beneficiile aduse de fiecare pattern în parte:

* *Adapter Pattern* :

- Permite interoperabilitatea între componente cu interfețe incompatibile.

- Ușurează integrarea și reutilizarea codului existent.

- Reduce dependențele între clase și module.

* *Bridge Pattern* :

- Separă abstractizarea de implementare, oferind flexibilitate și extensibilitate.

- Permite evoluția independentă a abstractizării și implementării.

- Facilitează modularitatea și adaptabilitatea sistemului.

* *Composite Pattern* :

- Permite tratarea uniformă a obiectelor individuale și a grupurilor acestora.

- Simplifică manipularea și gestionarea obiectelor într-o ierarhie.

- Oferă structură clară și flexibilitate în manipularea obiectelor compuse.

* *Decorator Pattern* :

- Permite adăugarea de comportamente suplimentare la obiecte existente.

- Evită modificarea structurii de bază a obiectului.

- Oferă flexibilitate și extensibilitate în adăugarea de funcționalități.

* *Facade Pattern* :

- Ascunde complexitatea unui subsistem și oferă o interfață simplificată.

- Reduce dependențele și oferă un punct de acces unic.

- Simplifică utilizarea și gestionarea subsistemului.

* *Flyweight Pattern* :

- Optimizează utilizarea memoriei prin partajarea datelor între obiecte.

- Reduce cantitatea de memorie necesară pentru obiectele similare.

- Îmbunătățește performanța și reduce costurile de resurse.

* *Proxy Pattern* :

- Oferă o reprezentare înlocuitoare a unui obiect și controlează accesul la acesta.

- Permite adăugarea de funcționalități suplimentare înainte sau după apelurile metodelor.

- Asigură controlul flexibil al accesului la obiect și gestionarea eficientă a resurselor.

Avantajele Design Patterns Structural se evidențiază mult și este propice în dezvoltarea sistemelor software, incluzând interoperabilitatea, reutilizarea, modularitatea, extensibilitatea, optimizarea performanței și gestionarea eficientă a resurselor. Prin aplicarea adecvată a acestor patternuri, dezvoltatorii pot crea soluții software robuste, ușor de întreținut și scalabile.

**2.3 Behavioral**

Design Patterns Behavioral reprezintă un set de modele de proiectare care se concentrează pe comportamentul obiectelor și interacțiunile dintre ele în cadrul unei aplicații software. Aceste modele oferă soluții pentru gestionarea fluxului de execuție, comunicarea între obiecte și definirea comportamentului dinamic al sistemului.

1. Observer

Observer permite obiectelor să se aboneze și să primească actualizări de la un subiect. Acesta facilitează comunicarea între obiecte și permite o abordare modulară și extensibilă. Prin implementarea acestui pattern, modificările într-un obiect sunt propagate automat către toți observatorii săi.

2. Strategy:

Strategy definește o familie de algoritmi și îi încapsulează în obiecte separate, permițându-le să fie interschimbabile. Astfel, un client poate alege în timpul execuției algoritmul potrivit dintr-o familie de strategii disponibile. Acest pattern facilitează flexibilitatea și extensibilitatea în cadrul unei aplicații.

3. Command:

Command encapsulează o solicitare ca un obiect, permițând astfel parametrizarea clienților cu diverse solicitări, coada sau înregistrarea solicitărilor și suportul pentru anularea operațiilor. Acest pattern separă solicitarea de obiectul care o primește, facilitând astfel cuplarea slabă între sender și receiver și oferind un nivel înalt de flexibilitate în gestionarea comenzilor.

4. Iterator:

Iterator oferă o modalitate de a accesa elementele unei colecții într-un mod secvențial fără a expune structura internă a colecției. Acesta definește o interfață comună pentru iterație și permite iterarea prin obiectele unei colecții fără a cunoaște detalii despre implementarea acesteia.

5. Chain of Responsibility:

Patternul Chain of Responsibility construiește o serie de obiecte care pot procesa o solicitare în lanț. Fiecare obiect în lanț decide dacă va procesa solicitarea sau o va pasa mai departe la următorul obiect din lanț. Acest pattern permite decuplarea sender-ului de receiver-i și oferă flexibilitate în modul în care se gestionează și se procesează solicitările.

6. State:

State permite unui obiect să-și schimbe comportamentul în funcție de starea internă. Acesta încapsulează stările în obiecte separate și permite obiectului să treacă de la o stare la alta în funcție de evenimente sau solicitări externe. Prin utilizarea acestui pattern, se obține o abordare modulară și flexibilă a gestionării stărilor în cadrul aplicației.

7. Visitor:

Visitor definește o operație nouă care poate fi aplicată unei structuri de obiecte fără a modifica clasele acestora. Acest pattern separă operațiile de obiectele pe care le aplică, oferind astfel o extensibilitate în gestionarea operațiilor și obiectelor.

Design Patterns Behavioral care pot fi folosite pentru a rezolva diverse probleme de proiectare și a îmbunătăți modularitatea, extensibilitatea și flexibilitatea în dezvoltarea software. Prin înțelegerea și aplicarea corectă a acestor pattern-uri, dezvoltatorii pot crea aplicații mai eficiente, ușor de întreținut și adaptabile la schimbările cerințelor.

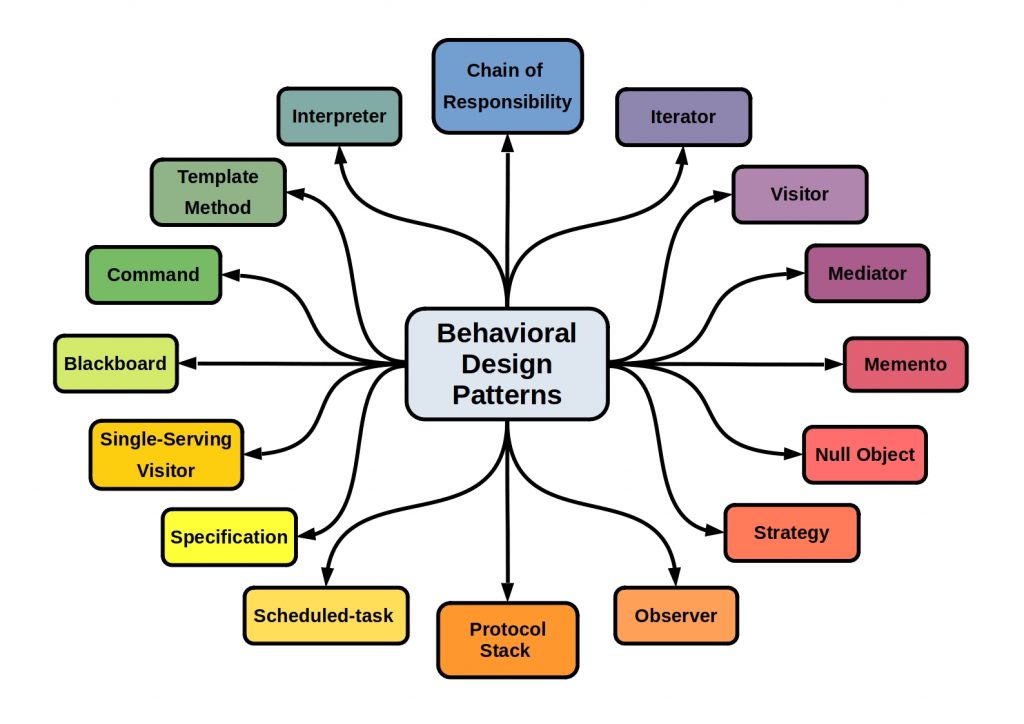
****

Fig. 2.3 – componența Design Pattern Behavioral

**2.3.1 Beneficiile pattern-urilor comportamentale**

Design Patterns Behavioral aduc o serie de beneficii semnificative în dezvoltarea software. Fiecare pattern oferă soluții specifice pentru problemele de comportament și interacțiune dintre obiecte.

* *Observer Pattern* :

- Simplifică comunicarea între obiecte prin intermediul unui sistem de notificări.

- Permite actualizări în timp real și sincronizare eficientă între obiecte.

- Sporește flexibilitatea și extensibilitatea prin abonarea și dezabonarea dinamică a observatorilor.

* *Strategy Pattern* :

- Oferă o modalitate elegantă de a schimba comportamentul unei aplicații în timpul execuției.

- Permite selectarea și utilizarea dinamică a unei strategii potrivite în funcție de context.

- Încurajează reutilizarea codului și separarea logică a algoritmilor.

* *Command Pattern* :

- Izolează solicitările în obiecte separate, facilitând extensibilitatea și configurabilitatea aplicației.

- Susține operații de anulare (undo) și reexecutare (redo) pentru solicitările efectuate.

- Simplifică gestionarea comenzilor complexe și crește nivelul de control asupra execuției acestora.

* *Iterator Pattern* :

- Ascunde detalii de implementare ale colecțiilor, oferind o interfață simplă și consistentă de iterare.

- Permite parcurgerea în mod secvențial a elementelor unei colecții fără a dezvălui structura internă.

- Sporește modularitatea și flexibilitatea în manipularea și procesarea datelor.

* *Chain of Responsibility Pattern* :

- Elimină cuplajul strâns între sender și receiver prin intermediul unui lanț de obiecte.

- Oferă flexibilitate în gestionarea solicitărilor, fiecare obiect decidând să le proceseze sau să le paseze mai departe.

- Simplifică adăugarea și eliminarea de noi obiecte din lanț, fără a afecta restul sistemului.

* *State Pattern* :

- Permite obiectelor să-și schimbe comportamentul în funcție de starea internă.

- Simplifică implementarea și gestionarea automată a tranzacțiilor între stări.

- Încurajează modularitatea și ușurința în extinderea aplicațiilor cu noi stări și comportamente.

* *Visitor Pattern* :

- Permite adăugarea de operații noi pe structuri de obiecte fără a modifica clasele acestora.

- Simplifică procesarea și manipularea complexă a obiectelor într-un mod structurat și modular.

- Sporește extensibilitatea prin adăugarea de vizitatori noi fără a afecta obiectele existente.

**3.**  **Descrierea principiului de lucru al tematicii alese.**

1. Singleton: Se utilizează sablonul Singleton pentru a asigura că există o singură instanță a clasei BibliotecaSingleton. Acesta asigură că obiectul este creat doar o singură dată și oferă un punct de acces global prin intermediul proprietății statice Instance. Astfel, orice parte a codului care are nevoie de instanța clasei BibliotecaSingleton poate accesa această instanță unică utilizând BibliotecaSingleton.Instance.
2. Builder: Se utilizează sablonul Builder pentru a construi obiecte de tip Carte. Clasa CarteBuilder oferă metode pentru a seta proprietățile cărții, cum ar fi titlul, autorul și anul de publicare. Aceasta permite construirea flexibilă a obiectelor Carte fără a fi nevoie să se utilizeze constructori supraincarcați sau setări separate ale proprietăților.
3. Adapter: Se utilizează sablonul Adapter pentru a adapta interfața IAdaptorCarte la clasa GestiuneCarti. Adaptorul AdaptorGestiuneCarti implementează interfața IAdaptorCarte și utilizează clasa GestiuneCarti pentru a adăuga și șterge cărți în gestiune. Astfel, interfețele diferite sunt unite prin intermediul adaptorului.
4. Decorator: Se utilizează sablonul Decorator pentru a extinde comportamentul clasei GestiuneCarti. Clasa GestiuneCartiDecorator implementează interfața IGestiuneCarti și primește o instanță de tip IGestiuneCarti în constructor. Aceasta adaugă informații suplimentare înainte de a apela metoda AdaugaCarte a instanței primite. Astfel, comportamentul inițial al clasei GestiuneCarti este extins fără a modifica clasa de bază.
5. Facade: Se utilizează sablonul Facade pentru a oferi o interfață simplificată pentru a accesa funcționalitățile complexe ale clasei GestiuneCarti. Clasa GestiuneCartiFacade ascunde detaliile implementării și oferă metode simple pentru a adăuga și șterge cărți în gestiune.
6. Proxy: Se utilizează sablonul Proxy pentru a controla accesul la clasa GestiuneCarti prin intermediul clasei GestiuneCartiProxy. Aceasta verifică drepturile utilizatorului înainte de a apela metoda AdaugaCarte a clasei GestiuneCarti. Astfel, accesul la funcționalitatea clasei este restricționat în funcție de drepturile utilizatorului.

Acestea sunt cele 6 sabloane de proiectare pe care le-ai enumerat și care sunt utilizate în codul tău. Fiecare sablon aduce beneficii specifice în termeni de structură, flexibilitate și extensibilitate a codului.

**Concluzie:**

În acest proiect, am identificat și analizat utilizarea a șase sabloane de proiectare: Singleton, Builder, Adapter, Decorator, Facade și Proxy. Aceste sabloane contribuie la îmbunătățirea structurii, flexibilității și extensibilității codului, oferind o abordare modulară și clară în dezvoltarea software.

Este important de remarcat că utilizarea sabloanelor de proiectare trebuie să fie în concordanță cu necesitățile și cerințele proiectului. În general, acestea pot oferi beneficii semnificative în termeni de eficiență, ușurință în dezvoltare și mentenanță, dar trebuie evaluate cu atenție în contextul specific al fiecărui proiect.

În concluzie, utilizarea sabloanelor de proiectare în codul furnizat aduce multiple beneficii în ceea ce privește organizarea, modularitatea și ușurința de întreținere a codului. Fiecare sablon abordează o problemă specifică și oferă o soluție elegantă și reutilizabilă.

Sabloanele Singleton și Builder permit crearea și gestionarea eficientă a instanțelor obiectelor, asigurându-se că există o singură instanță a unei clase și oferind o modalitate flexibilă de construire a obiectelor complexe.

Adapterul și Decoratorul facilitează integrarea și extinderea funcționalității existente. Adapterul asigură compatibilitatea între clase cu interfețe diferite, iar Decoratorul adaugă funcționalitate suplimentară la o clasă existentă.

Sabloanele Facade și Proxy oferă interfețe simple și controlate pentru a accesa funcționalități complexe. Facadeul ascunde detaliile complexe ale unei clase și oferă o interfață mai ușor de utilizat, în timp ce Proxy-ul controlează accesul și aplică verificări suplimentare înainte de a accesa funcționalitatea de bază.

Este important să menționăm că utilizarea sabloanelor de proiectare trebuie făcută cu discernământ și în funcție de cerințele specifice ale proiectului. Fiecare sablon are avantajele și limitările sale, iar aplicarea lor corectă necesită o înțelegere clară a contextului și a nevoilor proiectului.

În final, utilizarea sabloanelor de proiectare în codul furnizat denotă o abordare bine structurată și modulară a dezvoltării software. Aceste sabloane oferă beneficii în termeni de reutilizabilitate, flexibilitate și ușurință în dezvoltare, contribuind la crearea unui cod mai clar, mai extensibil și mai ușor de întreținut.