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 unui Quiz.

**A efectuat:**  Scutaru Dana , 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.

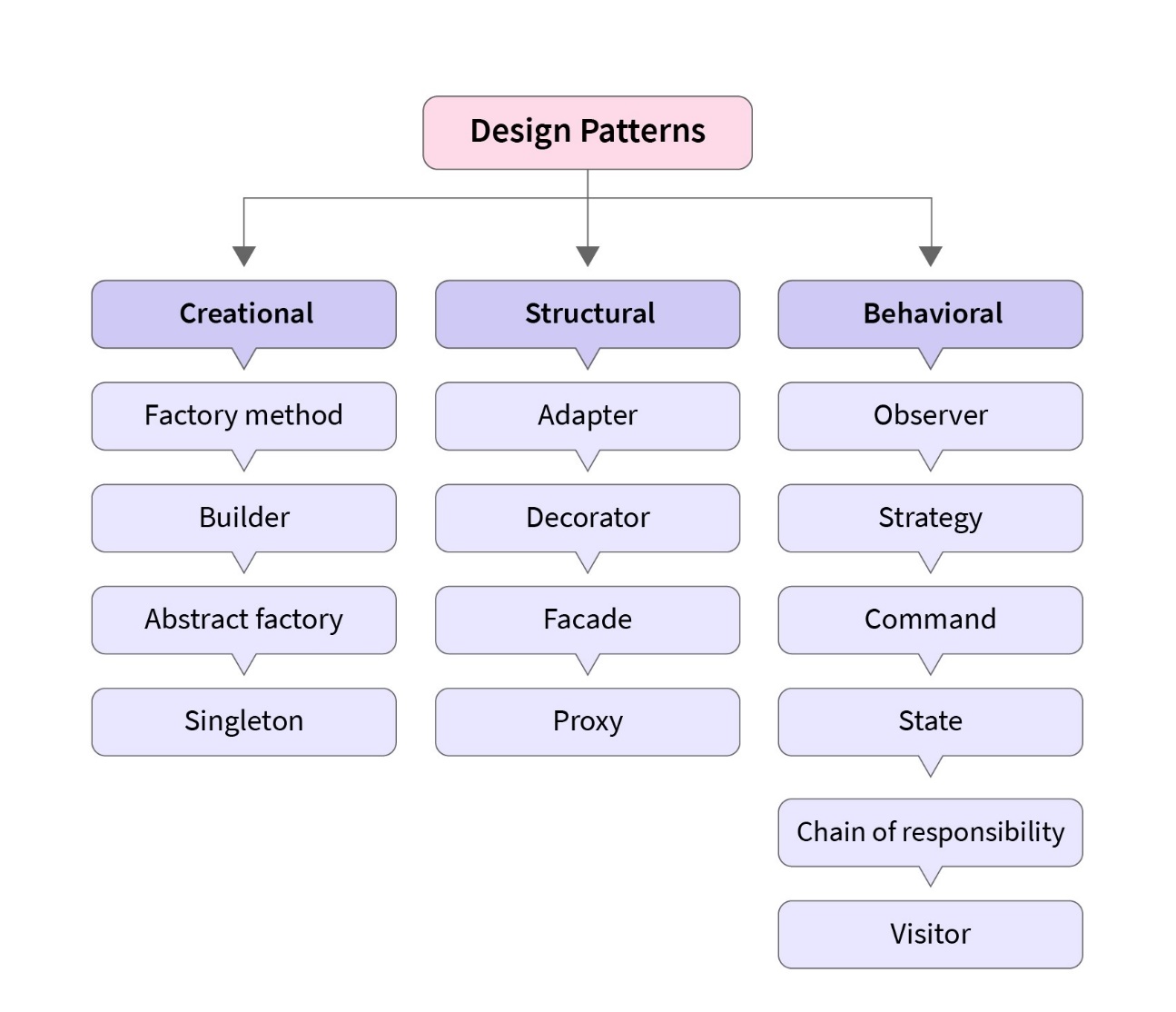
Interface Segregation Principle (ISP) promovează ideea că interfețele nu ar trebui să fie prea mari sau prea specifice. Ele ar trebui să fie concepute astfel încât să satisfacă nevoile exacte ale modulelor care le utilizează. Acest principiu ajută la evitarea dependențelor inutile și la reducerea cuplării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

  
  
Fig. 1 – Design Patterns

**1. Analiza domeniului problemei.**

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

Design Patterns, sau în limba română "modele de proiectare", sunt soluții generice pentru problemele frecvent întâlnite în dezvoltarea software. Acestea sunt reprezentate de abordări și structuri de proiectare care au fost dezvoltate și testate în timp de către experți în domeniu. Design Patterns oferă o metodologie și un vocabular comun pentru a rezolva anumite tipuri de probleme într-un mod eficient și reutilizabil.

Design Patterns sunt utilizate pentru a rezolva problemele comune de proiectare și implementare ale software-ului, oferind o abordare sistematică și organizată. Acestea furnizează soluții adaptabile și flexibile, care pot fi aplicate în diferite contexte și limbaje de programare.

Există o varietate de Design Patterns disponibile, iar acestea pot fi împărțite în categorii generale, cum ar fi creționale, structurale și comportamentale. Design Patterns creționale se concentrează pe procesul de creare a obiectelor. Acestea oferă mecanisme pentru a crea obiecte într-un mod flexibil și eficient, cum ar fi Singleton, care asigură că o clasă are o singură instanță, sau Factory Method, care furnizează o interfață pentru crearea obiectelor, dar lasă subclasele să decidă ce tip de obiect să creeze.

Design Patterns structurale se concentrează pe organizarea obiectelor și a relațiilor dintre ele. Acestea facilitează construirea unor structuri complexe prin compunerea obiectelor în moduri diferite. Exemple de Design Patterns structurale includ Adapter, care permite unui obiect să funcționeze cu alte obiecte incompatibile prin intermediul unui adaptor, sau   
Decorator, care permite adăugarea de comportamente suplimentare la un obiect existent fără a modifica structura sa.

Design Patterns comportamentale se concentrează pe comunicarea și interacțiunea dintre obiecte. Acestea descriu modul în care obiectele cooperează și își schimbă comportamentul în funcție de anumite evenimente sau situații. Exemple de Design Patterns comportamentale includ Observer, care permite unor obiecte să monitorizeze și să reacționeze la evenimente produse de alte obiecte, sau Strategy, care permite selecția și utilizarea dinamică a unui algoritm specific în timpul rulării.

Design Patterns au avantajul de a oferi soluții testate și validate pentru problemele comune de proiectare. Ele promovează reutilizabilitatea și modularitatea, permițând dezvoltatorilor să implementeze soluții eficiente și flexibile într-un mod consistent. Utilizarea Design Patterns facilitează, de asemenea, înțelegerea și colaborarea între dezvoltatori, deoarece aceștia împărtășesc un vocabular comun și o înțelegere comună a conceptelor și abordărilor de proiectare.

Design Patterns reprezintă soluții generice și testate pentru problemele frecvent întâlnite în dezvoltarea software. Acestea oferă un cadru și o metodologie pentru a rezolva problemele de proiectare într-un mod eficient și reutilizabil. Prin aplicarea Design Patterns, dezvoltatorii pot crea software modular, flexibil și ușor de întreținut, care să îndeplinească cerințele complexe ale utilizatorilor.

**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.

Bottom of Form

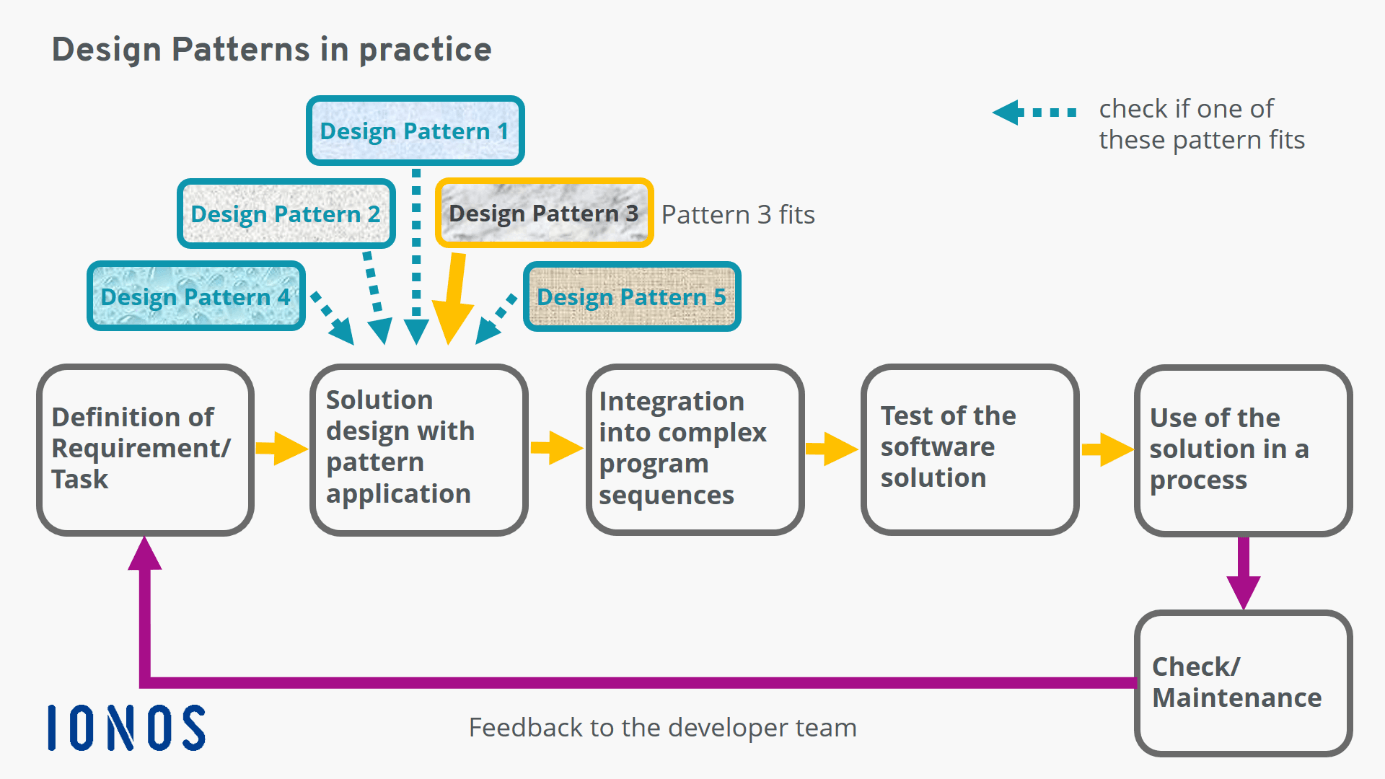


Fig.1.2 - utilizarea Design Patterns în practică

**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.

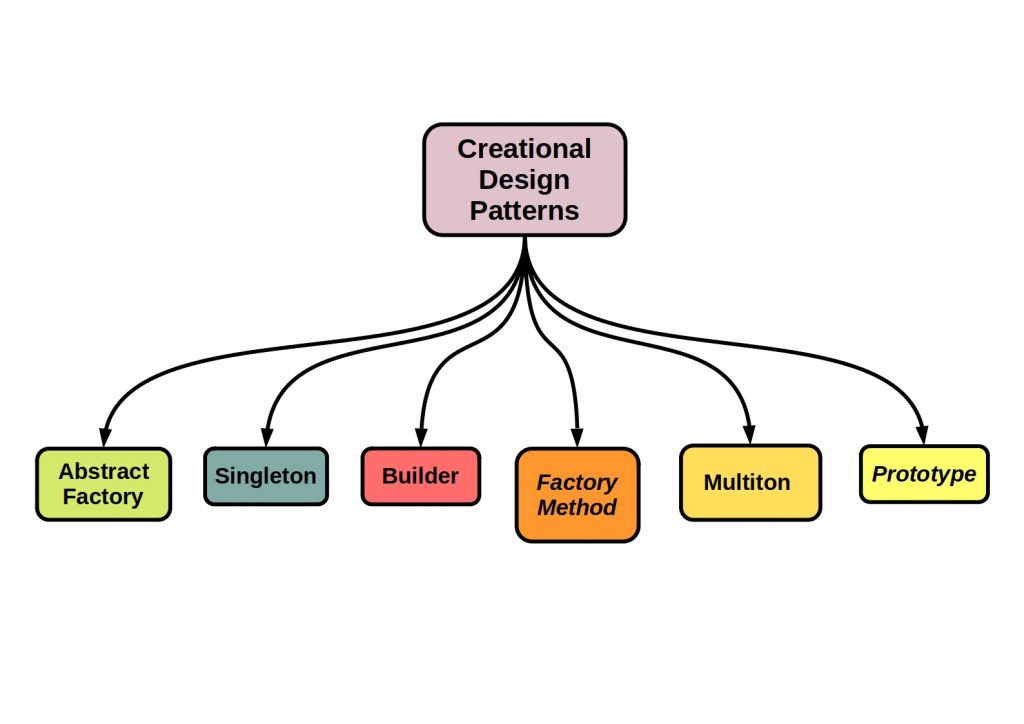


Fig.2.1 - componența Design Pattern Creational

**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.

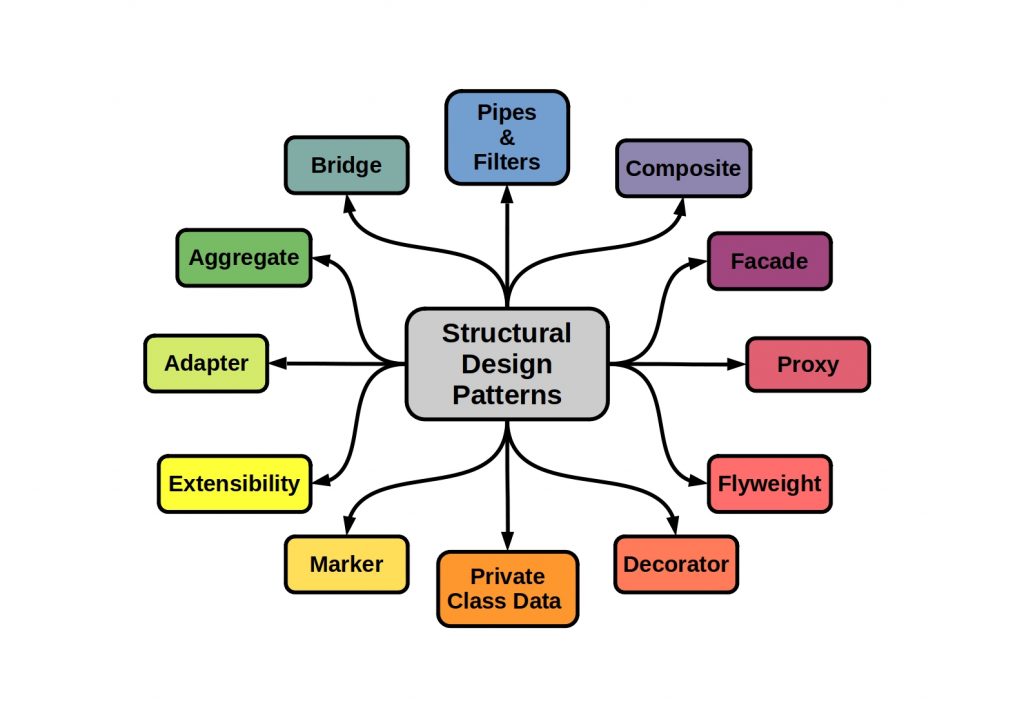


Fig.2.2 - componența Design Pattern Structural

**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.

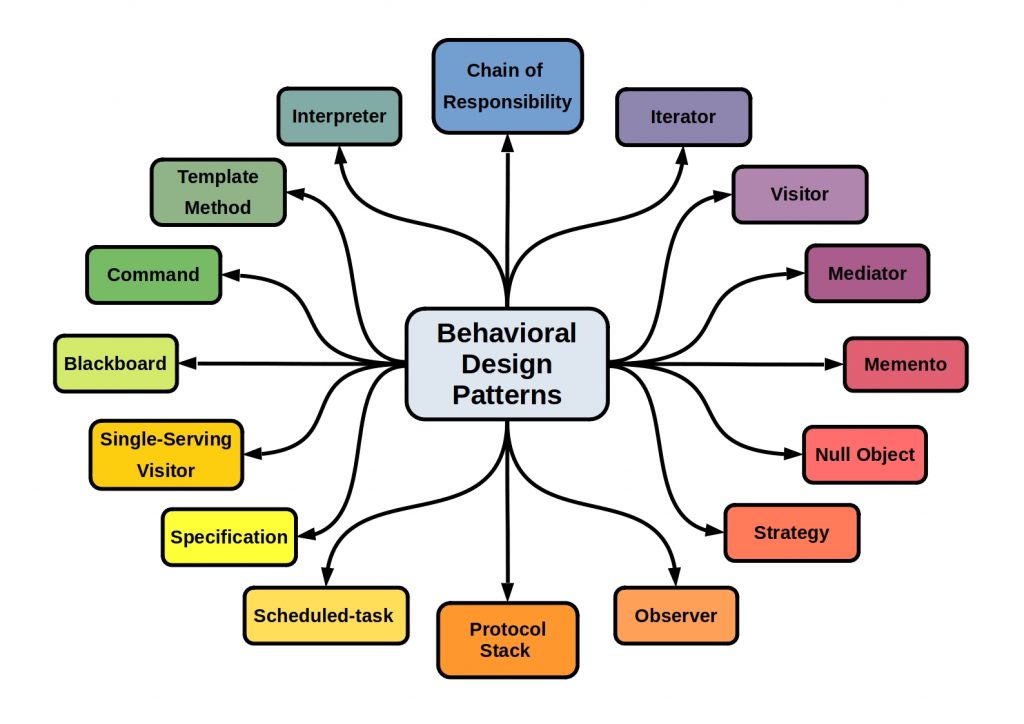
****

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.**

**3.1 Implementarea principiilor SOLID .**

Implementarea principiului Single Responsibility poate fi observată foarte ușor în proiect. Ea se evidențiază prin separarea responsabilităților în clase distincte.

Clasa **Question** se ocupă de gestionarea întrebărilor și verificarea răspunsurilor. Aceasta are o singură responsabilitate, și anume să reprezinte o întrebare și să ofere metode pentru afișarea întrebării și verificarea răspunsului utilizatorului.

Clasa **ScoreManager** se ocupă de gestionarea scorului. Aceasta are responsabilitatea de a actualiza și afișa scorul curent.

Prin aplicarea acestui principiu, fiecare clasă are o singură responsabilitate bine definită și nu se încalcă principiul în care o clasă ar trebui să aibă o singură motivare pentru a se schimba. Astfel, dacă se dorește modificarea modului în care sunt gestionate scorurile, se va interveni doar în clasa **ScoreManager**, fără a afecta clasa **Question**. De asemenea, dacă se dorește adăugarea de noi funcționalități pentru întrebări, se va interveni doar în clasa **Question**, fără a afecta modul în care scorul este gestionat.

Prin respectarea acestui principiu, obținem un cod mai modular, mai ușor de întreținut și ușor de extins în viitor.

Principiul Open-Closed se referă la faptul că entitățile software (clase, module, funcții etc.) ar trebui să fie deschise pentru extindere, adică să permită adăugarea de noi funcționalități, dar în același timp închise pentru modificare, evitându-se modificarea codului existent atunci când se adaugă noi funcționalități.

În Quiz, principiul Open-Closed este aplicat prin intermediul utilizării claselor abstracte și a metodelor abstracte din modulul **abc**. Clasa abstractă **GameQuestion** definește interfața comună pentru întrebările jocului și specifică metodele **display\_question** și **check\_answer** care trebuie implementate de către clasele derivate.

Clasele concrete **TrueFalseQuestion** și **MultipleChoiceQuestion1** extind clasa **GameQuestion** și implementează metodele abstracte pentru a afișa întrebările și a verifica răspunsurile. Aceste clase sunt deschise pentru extindere, deoarece pot fi derivate alte clase care adaugă noi tipuri de întrebări, fără a fi necesară modificarea codului existent.

Principiul Liskov este respectat prin utilizarea clasei abstracte **GameQuestion** ca clasă de bază și implementarea metodelor abstracte **display\_question** și **check\_answer**. Astfel, toate clasele derivate, precum **TrueFalseQuestion** și **MultipleChoiceQuestion**, respectă contractul impus de clasa de bază.

Clasa **TrueFalseQuestion** extinde clasa **GameQuestion** și suprascrie metodele **display\_question** și **check\_answer** pentru a afișa întrebarea și a verifica răspunsul în format specific întrebărilor cu adevărat/fals.

Clasa **MultipleChoiceQuestion** extinde, de asemenea, clasa **GameQuestion** și adaugă o nouă proprietate **options** pentru a stoca opțiunile de răspuns. Metodele suprascrise **display\_question** și **check\_answer** permit afișarea întrebării și verificarea răspunsului pentru întrebările cu opțiuni multiple.

Funcția **play\_game** primește o listă de întrebări și le parcurge într-un ciclu. Apelurile metodelor **display\_question** și **check\_answer** sunt realizate prin intermediul polimorfismului, respectând principiul Liskov. Astfel, putem trata obiectele de tip **TrueFalseQuestion** și **MultipleChoiceQuestion** ca obiecte de tipul clasei de bază **GameQuestion**, ceea ce ne oferă flexibilitate în gestionarea și procesarea întrebărilor într-un mod consistent.

Prin aplicarea principiului Liskov, avem un design coerent și extensibil, în care clasele derivate pot fi utilizate în locul clasei de bază, fără a afecta comportamentul programului și fără a introduce erori sau inconsistente.

Importanța principiului Interface Segregation constă în evitarea dependențelor inutile și a încălcării principiului "nu forța un client să depindă de lucruri pe care nu le folosește". Prin segregarea interfețelor, obiectele pot interacționa doar cu metodele care le sunt necesare, reducând dependențele și cuplarea între componente.

Implementarea acestui principiu implică identificarea grupurilor de metode relevante pentru fiecare client și definirea interfețelor separate pentru fiecare grup. Aceasta permite fiecărui client să utilizeze doar interfețele relevante, fără a fi nevoie să implementeze metodele care nu sunt necesare.

În esență, principiul Dependency Inversion afirmă că modulele de nivel superior nu ar trebui să depindă direct de modulele de nivel inferior, ci ambele ar trebui să depindă de abstracțiuni.

Implementarea acestui principiu în codul prezentat se realizează prin utilizarea de clase abstracte și prin inversarea direcției dependențelor. Interfața abstractă **GameQuestion** este folosită ca o abstracțiune pentru întrebările jocului, iar clasele **TrueFalseQuestion** și **MultipleChoiceQuestion** o implementează.

Clasa **QuizGame** primește dependința **questions** (întrebările jocului) și **score\_manager** (gestionarul scorului) prin intermediul constructorului. Această abordare permite utilizarea diferitelor implementări ale interfeței **GameQuestion** și a gestiunii scorului în cadrul jocului, fără ca **QuizGame** să depindă direct de implementările specifice.

Top of Form

**3.2 Design Pattern-uri implementate în proiect. Funcțiile lor .**

Funcția patternului Factory Method în Quiz este implementată prin intermediul clasei **CustomQuestionFactory**. Această clasă servește ca o fabrică de întrebări personalizate, care sunt create în funcție de tipul specificat de utilizator.

Atunci când se apelează metoda **create\_question()** a clasei **CustomQuestionFactory**, utilizatorul introduce textul întrebării și specifică tipul întrebării (cu opțiuni multiple sau cu un singur răspuns). Pe baza acestor informații, fabrica creează și returnează instanța corespunzătoare a clasei întrebare (**MultipleChoiceQuestion** sau **FillInTheBlankQuestion**).

Prin utilizarea patternului Factory Method, obținem o abstracție și encapsulare eficientă a procesului de creare a întrebărilor personalizate. Fabrica decuplează codul client de logica specifică creării întrebărilor, permițând extensibilitate și flexibilitate în adăugarea și gestionarea diferitelor tipuri de întrebări.

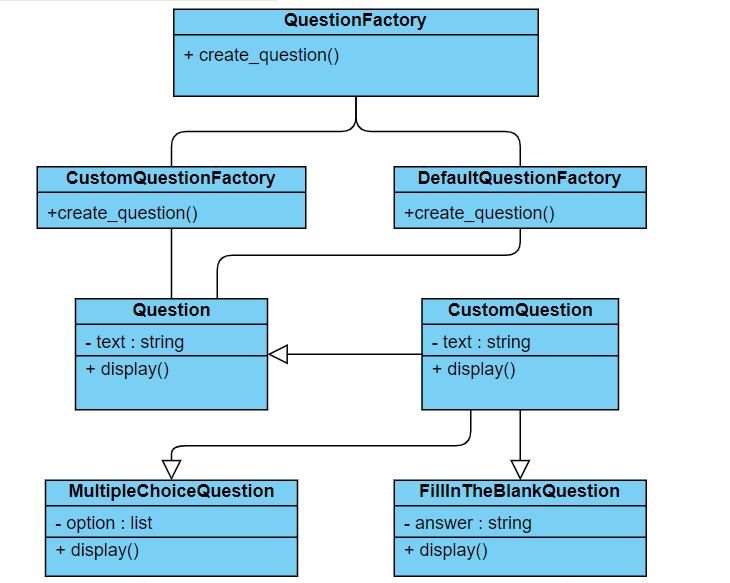


Fig. 3.1 – diagrama UML, pentru Factory Method

Singleton este un pattern de proiectare care asigură existența unei singure instanțe a unei clase în întregul sistem și oferă un punct central de acces la această instanță. În proiectul meu, clasa **GameManager** implementează Singleton.

Pentru a asigura că există o singură instanță a clasei **GameManager**, se folosește o variabilă statică **\_instance** care păstrează referința către instanța existentă. În metoda **\_\_new\_\_**, se verifică dacă instanța există deja și, în caz afirmativ, se returnează. În caz contrar, se folosește metoda **super().\_\_new\_\_** pentru a crea o nouă instanță și se atribuie variabilei **\_instance** pentru o reutilizare în apelurile ulterioare.

Prin implementarea Singleton, trebuie să fim siguri că toate apelurile către clasa **GameManager** se referă la aceeași instanță și că aceasta este partajată între diferitele componente ale sistemului. Astfel, orice modificare făcută în cadrul instanței va fi vizibilă în întregul sistem. Acest pattern este util în situațiile în care trebuie să existe o singură instanță a unei clase și trebuie să se evite crearea duplicatelor.

Singleton oferă acces centralizat la instanță și permite obținerea valorilor și modificarea lor prin intermediul metodelor publice definite în clasă. Astfel, putem accesa și actualiza scorul, nivelul și timpul rămas prin intermediul metodelor **update\_score()**, **increase\_level()**, **update\_time()**, **get\_score()**, **get\_level()** și **get\_time\_remaining()**.

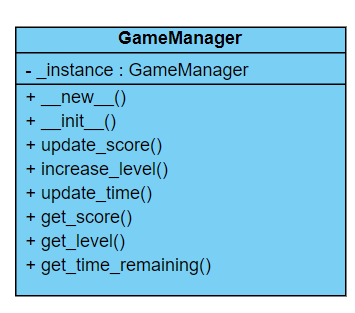


Fig. 3.2 – diagrama UML, pentru Singleton

Bridge este utilizat pentru a separa ierarhia de întrebări (**Question**, **TrueFalseQuestion**, **MultipleChoiceQuestion**) de ierarhia de moduri de afișare (**DisplayMode**, **TextDisplayMode**, **GraphicDisplayMode**).

Clasa **Question** este clasa de bază pentru toate tipurile de întrebări și acceptă un obiect **display\_mode** în constructorul său. Această clasă definește metodele comune pentru afișarea întrebărilor și verificarea răspunsurilor.

Clasele **TrueFalseQuestion** și **MultipleChoiceQuestion** extind clasa **Question** și suprascriu metoda **display\_question**. Ele utilizează obiectul **display\_mode** pentru a delega afișarea întrebării către modul corespunzător.

Clasa **DisplayMode** este clasa de bază pentru toate modurile de afișare și definește metoda **display\_question** care trebuie implementată în subclase. Astfel, ierarhia de moduri de afișare poate evolua independent de ierarhia de întrebări.

Subclasele **TextDisplayMode** și **GraphicDisplayMode** extind clasa **DisplayMode** și implementează metoda **display\_question** în funcție de modul de afișare specific. **TextDisplayMode** afișează întrebările în mod text simplu, iar **GraphicDisplayMode** poate fi implementată pentru a afișa întrebările într-un mod grafic specific.

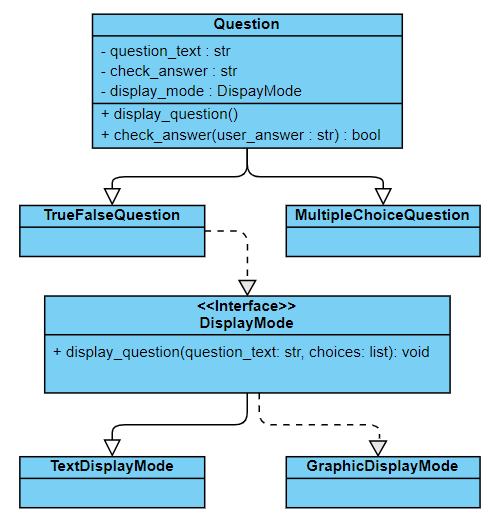


Fig. 3.3 – diagrama UML, pentru Bridge

Decoratorul este utilizat pentru a extinde comportamentul întrebărilor în mod flexibil.

Clasa **Question** este clasa de bază pentru toate întrebările și definește metodele **display\_question** și **check\_answer**, care trebuie implementate în subclase.

Clasele **TrueFalseQuestion** și **MultipleChoiceQuestion** extind clasa **Question** și implementează metodele specifice pentru afișarea întrebării și verificarea răspunsului.

Clasa **QuestionDecorator** este clasa de bază pentru decoratori și extinde clasa **Question**. Aceasta acceptă un obiect **decorated\_question** în constructor și redirecționează apelurile către metodele acestuia.

Clasele **CountingQuestionDecorator**, **SymbolAdding** și **NumberedQuestionDecorator** sunt exemple de clase decoratoare care extind clasa **QuestionDecorator**. Acestea adaugă funcționalități suplimentare la întrebări prin suprascrierea metodelor de afișare și adăugând comportament specific, cum ar fi numărul întrebării sau adăugarea unui simbol la întrebare și opțiuni.

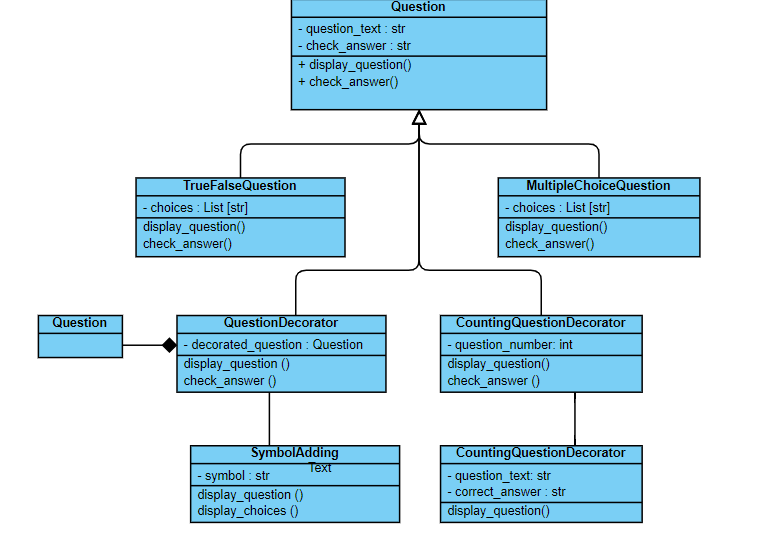


Fig. 3.4 – diagrama UML, pentru Bridge

Observer este un pattern de proiectare care permite notificarea automată a unor obiecte dependente (observatori) despre schimbările de stare ale unui obiect subiect.

Clasa **Subject** reprezintă subiectul care poate avea observatori atașați și notifică acești observatori despre schimbările de stare. Metodele **attach** și **detach** sunt utilizate pentru atașarea și detasarea observatorilor, iar metoda **notify** parcurge lista de observatori și apelează metoda **update** pentru fiecare observator în parte, oferindu-le astfel notificarea.

Clasa **Observer** este clasa de bază pentru observatori și definește metoda **update**, care trebuie implementată în subclase. În codul tău, clasele **QuizGameObserver** și **CustomQuestionFactoryObserver** extind clasa **Observer** și implementează metoda **update** pentru a reacționa la notificări specifice.

Clasa **ScoreManagerSubject** este un subiect concret și extinde clasa **Subject**. Aceasta conține o variabilă **score** și are funcționalitatea de a actualiza scorul și notifica observatorii despre schimbările acestuia prin intermediul metodei **update\_score** și a metodei suprascrise **notify**.

În plus,există și alte clase, cum ar fi **QuestionDecorator**, **CountingQuestionDecorator** și **SymbolAdding**, care nu sunt direct legate de patternul Observer, dar sunt utilizate pentru a extinde comportamentul întrebărilor prin decorare.

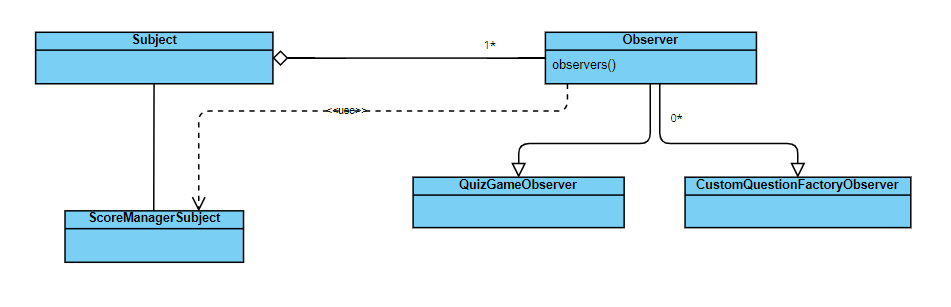


Fig. 3.5 – diagrama UML, pentru Observer

**Concluzii :**

În acest proiect, am explorat mai multe concepte și pattern-uri de proiectare, fiecare având un scop specific în dezvoltarea software-ului. Am acoperit următoarele pattern-uri: Factory Method, Singleton, Bridge, Decorator și Observer.

Factory Method este un pattern de creare a obiectelor, care permite obținerea unei instanțe a unei clase prin intermediul unei metode de fabrică. Acest pattern ajută la izolarea logicii de creare a obiectelor și oferă flexibilitate în alegerea tipului de obiect creat.

Singleton este un pattern care asigură că o clasă are o singură instanță și oferă un punct global de acces la aceasta. Acesta este util în situațiile în care trebuie să existe o singură instanță a unei clase și se dorește accesul la aceasta din diverse părți ale codului.

Bridge este un pattern care separă abstractizarea de implementare, permițându-le să evolueze independent. Acest pattern ajută la gestionarea complexității și permite flexibilitate în modificarea și extinderea sistemului.

Decorator este un pattern care permite adăugarea de comportamente suplimentare la un obiect existent, fără a modifica structura de bază. Acesta oferă o alternativă la moștenirea multiplă și permite adăugarea dinamică de funcționalități.

Observer este un pattern care facilitează comunicarea între obiecte, unde un obiect subiect notifică automat și în mod transparent obiectele observatoare despre schimbările de stare. Acest pattern promovează slab cuplarea între obiecte și permite extensibilitatea și modularitatea sistemului.

În proiectul meu, am folosit aceste pattern-uri în contextul unui joc de întrebări și răspunsuri. Am definit clase pentru întrebări, manageri de scor, moduri de afișare și fabrici de întrebări personalizate. Utilizarea acestor pattern-uri a adus beneficii semnificative:

* Factory Method a permis crearea de obiecte de întrebare diverse în funcție de cerințe specifice, oferind o metodă simplă și flexibilă de creare a acestora.
* Singleton a asigurat că există o singură instanță a managerului de joc și a oferit acces global la acesta, evitând crearea duplicatelor și facilitând interacțiunea cu acesta.
* Bridge a permis separarea modurilor de afișare a întrebărilor de clasele de întrebări în sine, oferind posibilitatea de a adăuga noi moduri de afișare fără a afecta structura întrebărilor.
* Decorator a extins comportamentul întrebărilor prin decorare, permițând adăugarea de funcționalități suplimentare, cum ar fi numerotarea și adăugarea de simboluri.
* Observer a facilitat comunicarea între obiectele de joc, notificând automat observatorii despre schimbările de scor și de răspunsuri, oferind un sistem flexibil și modular.

Spre final am înțeles că, utilizarea acestor pattern-uri în proiectul meu a permis o structură modulară, flexibilă și ușor de extins. Am demonstrat înțelegerea și aplicarea acestor concepte, ceea ce va ajuta la crearea unui cod mai clar, mai ușor de întreținut și mai adaptabil la schimbări viitoare.

**Bibliografie :**

* Pentru informații despre Design Patterns:

<https://refactoring.guru/design-patterns/what-is-pattern>

* Pentru informații despre Creational și documentație:

<https://refactoring.guru/design-patterns/creational-patterns>

* Pentru informații despre Structural și documentație:

<https://refactoring.guru/design-patterns/structural-patterns>

* Pentru informații despre Behavioral și documentație:

<https://refactoring.guru/design-patterns/behavioral-patterns>

* Imagini:

<https://www.ionos.com/digitalguide/websites/web-development/what-are-design-patterns/>

<https://auth0.com/blog/strategy-design-pattern-in-python/>

<https://starship-knowledge.com/software-design-patterns>

<https://www.scaler.com/topics/design-patterns/types-of-design-pattern/>