Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare Informatică şi Microelectronică

Departamentul Ingineria Software și Automatică

**Proiect de an**

**la Tehnici și Mecanisme de Proiectare Software**

**Tema: Proiectarea și dezvoltarea aplicației**

**“Gestionarea unei universități”**

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

A verificat: asist. univ., Gaidău Mihai

Chișinău 2023

**Cuprins**

[Introducere 3](#_Toc136878906)

[1 Analiza domeniului problemei 4](#_Toc136878907)

[1.1 Importanța modelelor de proiectare 4](#_Toc136878908)

[1.2 Analiza modelelor de proiectare 6](#_Toc136878909)

[1.3.1 Modele de design creaționale 6](#_Toc136878910)

[1.3.2 Modele de design structurale 9](#_Toc136878911)

[1.3.3 Modele de design comportamental 12](#_Toc136878912)

[1.3 Scopul implementarii aplicației 14](#_Toc136878913)

[2 Proiectarea și dezvoltarea aplicației 15](#_Toc136878914)

[Concluzii 26](#_Toc136878915)

[Bibliografie 27](#_Toc136878916)

# Introducere

În ingineria software, un model de proiectare este o soluție generală repetabilă la o problemă frecventă în proiectarea software. Acestea sunt abordări testate și validate, care ajută la crearea unui cod modular, flexibil și ușor de întreținut. Un Design Pattern oferă o modalitate standardizată de a rezolva anumite probleme de proiectare, aducând beneficii în ceea ce privește reutilizarea codului, scalabilitatea, extensibilitatea și înțelegerea clară a structurii aplicației. Un model de design nu este un design finit care poate fi transformat direct în cod. Este o descriere sau un șablon pentru rezolvarea unei probleme care poate fi folosită în multe situații diferite.

Modelele de proiectare pot accelera procesul de dezvoltare prin furnizarea de paradigme de dezvoltare testate și dovedite. Proiectarea eficientă a software-ului necesită luarea în considerare a problemelor care pot să nu devină vizibile decât mai târziu în implementare. Reutilizarea modelelor de proiectare ajută la prevenirea problemelor subtile care pot cauza probleme majore și îmbunătățește lizibilitatea codului pentru programatori și arhitecții familiarizați cu modelele.

Adesea, oamenii înțeleg doar cum să aplice anumite tehnici de proiectare software la anumite probleme. Aceste tehnici sunt dificil de aplicat la o gamă mai largă de probleme. Modelele de design oferă soluții generale, documentate într-un format care nu necesită specificații legate de o anumită problemă.

În plus, modelele permit dezvoltatorilor să comunice folosind nume binecunoscute și bine înțelese pentru interacțiunile software. Modelele comune de design pot fi îmbunătățite în timp, făcându-le mai robuste decât modelele ad-hoc.

Prin explorarea și implementarea acestor Design Patterns, se câștigă o înțelegere mai profundă a beneficiilor pe care le aduc în procesul de proiectare software și cum pot fi aplicate într-un context real.

# Analiza domeniului problemei

# Importanța modelelor de proiectare

Modelul de design este un element esențial în programarea orientată pe obiecte. Este o infrastructură software alcătuită dintr-un număr mic de clase care este utilizată pentru a rezolva o problemă tehnică. Pentru a înțelege ce este, iată care este modelul de design și de ce ar trebui să fie folosit.

Modelul de design își are originea în faptul că dezvoltatorii au observat probleme de design recurente și similare. Prin urmare, a devenit necesară conceptualizarea problemelor de proiectare în așa fel încât aceleași răspunsuri să fie refolosite de fiecare dată când a apărut problema.

Pentru a conceptualiza, este adesea esențial să folosiți codul direct . Dar în cazul modelului de design, dezvoltatorii traduc conceptualizarea într-o idee abstractă despre cum să rezolve problema. Fiecare dezvoltator implementează apoi rezolvarea problemei folosind limbajul cu care este familiarizat. Există exemple de aplicații de modele de design pentru diferite limbi .

Prin urmare, modelul de design este conceput ca un proces general de proiectare care se bazează pe experiența deja dobândită în proiectarea software. Modelul de design are un rol important de jucat în sistemul informatic și arhitectura acestuia.

Design patterns reprezintă soluții generice și recunoscute pentru problemele comune de proiectare a software-ului. Acestea au fost dezvoltate și rafinate de-a lungul timpului și sunt utilizate pe scară largă în industrie. Importanța design patterns constă în capacitatea lor de a furniza structuri și abordări testate și validate, care îmbunătățesc calitatea și eficiența dezvoltării software.

Utilitatea utilizării unui model de design este evidentă. Modelul de proiectare poate accelera procesul de dezvoltare . Oferă paradigme de dezvoltare dovedite, care ajută la economisirea de timp fără a fi nevoie să reinventeze modele de fiecare dată când apare o problemă.

Aplicarea design patterns în dezvoltarea software aduce numeroase beneficii. Acestea includ:

* Reutilizabilitate: Design patterns permit reutilizarea soluțiilor existente, evitând astfel duplicarea codului și facilitând dezvoltarea eficientă și ușor de întreținut,
* Flexibilitate: Design patterns oferă flexibilitate în adaptarea și extinderea aplicațiilor la schimbările cerințelor și contextului,
* Înțelegere și comunicare: Utilizarea design patterns facilitează înțelegerea clară a structurii și comportamentului aplicației, ceea ce duce la o comunicare mai bună între membrii echipei de dezvoltare,
* Scalabilitate: Design patterns permit construirea de aplicații scalabile, care pot gestiona creșterea volumului de date și a cerințelor utilizatorilor fără a afecta performanța;

Deoarece modelul de proiectare este creat pentru a rezolva problemele cunoscute, acestea pot fi prezise înainte de a deveni vizibile în timpul procesului de implementare. Din nou, modelul de design accelerează procesul de dezvoltare . Standardizarea legată de modelul de proiectare este, de asemenea, foarte utilă pentru a facilita lizibilitatea codului .

Pe scurt, modelul de proiectare este util atunci când treceți de la un model de analiză la un model de dezvoltare . Poate fi folosit în mod concret în mai multe cazuri, în special pentru gestionarea sistemului de salarizare la modificarea unui salariu și pentru a menține sistemul informat cu privire la modificările pe care aceasta le presupune.

Datorită modelului de proiectare, este posibilă documentarea soluțiilor care trebuie adoptate pe baza practicilor anterioare și a lecțiilor învățate. Mai multe componente software sunt utilizate în implementarea modelului de proiectare. Modelul accelerează așadar un proces care implică mai multe componente. Dezvoltatorii folosesc limbajul cu care sunt familiarizați în aplicarea fiecărei soluții.

Atât arhitectul computerului, cât și programatorul sunt chemați să folosească același limbaj în raport cu modelul de proiectare. Arhitectul computerului trebuie doar să dea numele tipului de model de proiectare, astfel încât programatorul să înțeleagă despre ce este vorba. Nu va fi nevoie de explicații lungi. Prin urmare, este important ca profesioniștii să știe în ce constau principalele modele de design .

Modelul de proiectare poate fi utilizat în toate etapele muncii de programare. Poate fi folosit înainte, deoarece poate fi folosit ca ghid în scrierea codului sursă. Este, de asemenea, folosit frecvent după activitatea de programare, deoarece eficiența sa îi permite să fie folosit ca șablon pentru a lega modulele de cod sursă existente.

Desigur, modelul de proiectare este foarte util în timpul programării, deoarece poate stabili puncte comune cu modelele existente, ceea ce va ajuta la evitarea și anticiparea erorilor. De cele mai multe ori, modelul de design este utilizat atunci când este clar că dezvoltatorul are nevoie de flexibilitatea sa .

Fiecare model de design are propria sa abordare. Strategiile prezente în fiecare model de design sunt prin urmare originale și nu sunt niciodată aceleași de la un model de design la altul. În acest fel, este posibilă rezolvarea fiecărei probleme într-un mod organizat și rapid . Doar aplicați modelul de design potrivit, care sa dovedit deja. De asemenea, este posibil să combinați modele în dezvoltarea unei soluții.

# Analiza modelelor de proiectare

Modelele de design sunt soluții la probleme comune în dezvoltarea de software. În acest caz, se presupune că există un set de probleme generale formalizate care apar destul de des, iar tiparele oferă un set de principii pentru rezolvarea acestor probleme.

# Modele de design creaționale

Modelele de creație sunt tipare care abstrag procesul de instanțiere sau, cu alte cuvinte, procesul de generare a claselor și a obiectelor. Printre acestea se numără următoarele:

* Fabrica de abstracte
* Constructor
* Fabrica
* Prototip
* Singleton

**Fabrica de abstracte**

**Scop** : O fabrică abstractă oferă o interfață pentru crearea unei familii de obiecte dependente sau înrudite fără a specifica clasele lor concrete.

Cu alte cuvinte: o fabrică abstractă este o strategie de creare a unei familii de obiecte înrudite sau înrudite.

**Când de utilizat:**

* Când sistemul nu ar trebui să depindă de modul în care sunt create obiectele noi
* Când obiectele create urmează a fi utilizate împreună și sunt legate

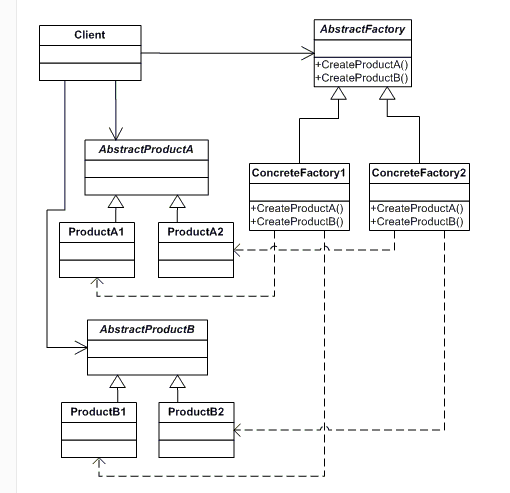


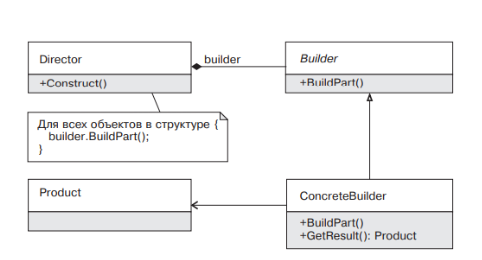
Diagrama modelului UML Fabrica abstractă

**Builder**

**Scop** : Constructorul decuplează construcția unui obiect complex de reprezentarea acestuia, astfel încât același proces de construcție să poată produce reprezentări diferite.

**Când de utilizat:**

* Când procesul de creare a unui nou obiect nu ar trebui să depindă de ce părți constă acest obiect și de modul în care aceste părți sunt interconectate
* Când este necesar să se asigure că în timpul creării sale se obțin diverse variații ale unui obiect

****

**Membrii:**

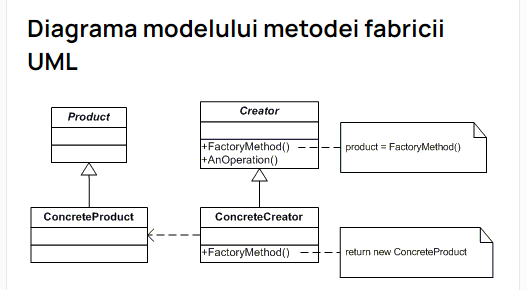
* Builder definește interfața pentru construirea unui produs bucată cu bucată;
* Director gestionează procesul de creare fără a ști ce produs va fi creat ca rezultat;
* ConcreteBuilder este un constructor de beton care creează doar un obiect din clasa Product pe care îl cunoaște .

**Metoda fabricii**

**Scop** : Definește o interfață pentru crearea unui obiect, dar lasă subclasele să decidă ce clasă să instanțieze. O metodă din fabrică permite unei clase să delege instanțierea subclaselor.

**Când de utilizat:**

* Când nu se știe dinainte ce tipuri de obiecte trebuie create;
* Când sistemul trebuie să fie independent de procesul de creare de noi obiecte și extensibil: este ușor să introduci noi clase în el, ale căror obiecte trebuie să le creeze sistemul;
* Când crearea de noi obiecte trebuie să fie delegată de la clasa de bază către clasele descendente;

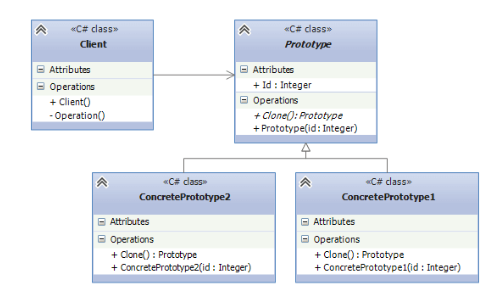
****

**Prototip**

**Scop** : Prototype vă permite să creați noi obiecte prin clonarea celor existente. De fapt, acest model oferă o tehnică pentru clonarea obiectelor.

**Când de utilizat:**

* Clasa obiectului generat este determinată în momentul execuției.
* Când este de dorit să se evite moștenirea creatorului unui obiect. În acest caz, Prototipul este un concurent al Fabricii Abstracte.
* Când clonați un obiect este de preferat să îl creați și să îl inițializați cu un constructor.
* Atunci când creați o copie a unui obiect este mai ușor și mai rapid decât generarea acesteia în mod standard, folosind noul operator și incluzând inițializarea câmpului.

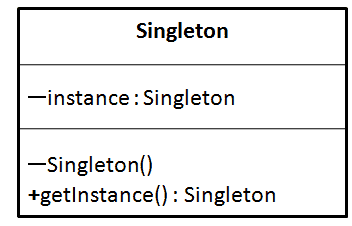


**Singleton**

**Scop** : asigură că clasa are o singură instanță și oferă un punct de acces global la aceasta

**Când de utilizat:**

În aproape orice aplicație, este nevoie de variabile globale sau obiecte cu un număr limitat de instanțe. Cel mai simplu mod de a rezolva această problemă este de a crea un obiect global care va fi accesibil de oriunde în aplicație. Prin definiție, un singleton asigură că o clasă are o singură instanță. În unele cazuri, analiza domeniului necesită strict ca o clasă să existe într-o singură instanță. Cu toate acestea, în practică, modelul Singleton este de obicei folosit pentru a oferi acces la o resursă care este cerută de diferite părți ale aplicației.



# Modele de design structurale

Modele structurale - ia în considerare modul în care clasele și obiectele formează structuri mai mari - clase și obiecte care sunt de natură mai complexă. Aceste șabloane includ:

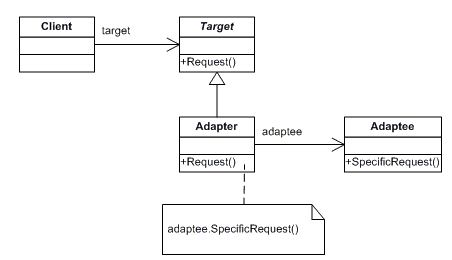
* Adaptor
* Pod
* Compozit
* Decorator
* Faţadă
* Greutatea muștei
* Adjunct (împuternicit)

**Adaptor**

**Scop** : Convertește o interfață de la o clasă la alta la care se așteaptă clienții. Un adaptor face posibil ca clasele cu interfețe incompatibile să lucreze împreună.

**Când de utilizat:**

* Când este necesar să se folosească o clasă existentă, dar interfața acesteia nu satisface nevoile logicii de afaceri.
* Când trebuie să utilizați o clasă deja existentă împreună cu alte clase ale căror interfețe nu sunt compatibile.



**Pod**

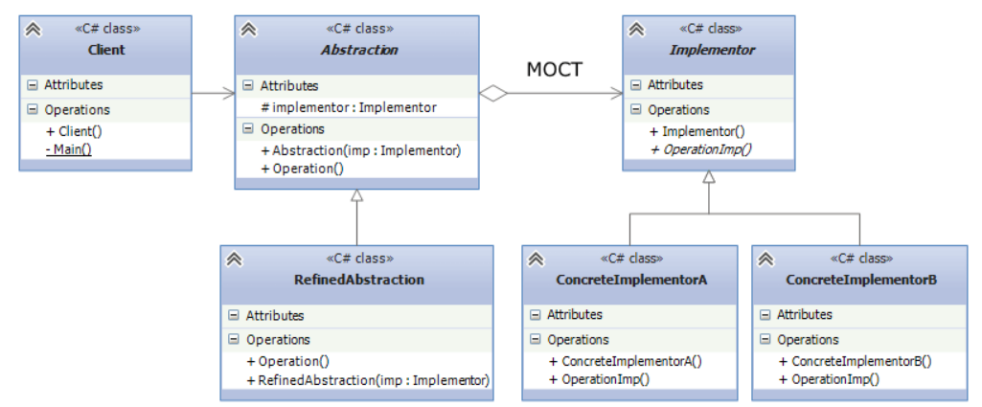
**Scop** :

Bridge (Bridge) - un model de proiectare structurală care vă permite să separați abstracția de implementare, astfel încât atât abstracția, cât și implementarea să poată fi modificate independent una de cealaltă.

Modelul Bridge presupune că codul de bază necesar pentru funcționarea unui obiect este transferat în implementare. Orice altceva, inclusiv interacțiunea cu clientul, este conținut în abstractizare. Metodele sale, dacă este necesar, pot fi modificate sau completate. De asemenea, conține o instanță de implementare și o folosește pentru a gestiona solicitările de la clienți. Procesarea înseamnă atât redirecționarea directă a cererii, cât și apelarea unui grup de metode de implementare pentru a obține rezultatul.

**Când de utilizat:**

* Când să evitați legarea constantă a abstracției de implementare
* Când, odată cu implementarea, este necesară schimbarea abstracției independent una de cealaltă. Adică, o schimbare a abstractizării nu ar trebui să ducă la o schimbare a implementării.

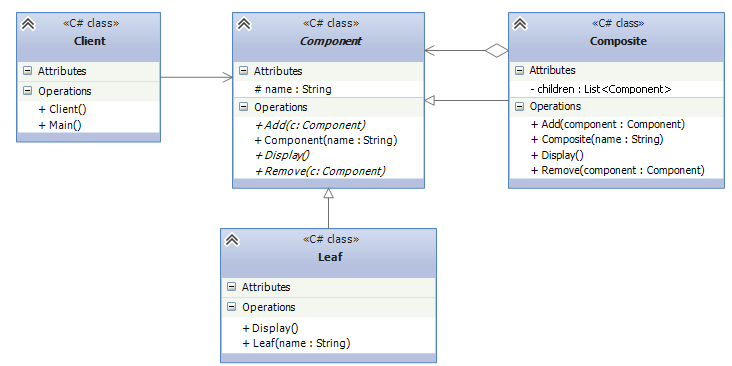
****

**Compozit**

**Scop** : Aranjează obiectele în structuri arborescente pentru a reprezenta ierarhii parțial-întreg. Permite clienților să trateze obiectele individuale și compuse într-un mod consistent.

**Când de utilizat:**

* Când obiectele ar trebui implementate ca o structură arborescentă ierarhică
* Când clienții trebuie să gestioneze atât obiectele întregi, cât și părțile lor componente într-un mod uniform. Adică, întregul și părțile sale trebuie să implementeze aceeași interfață

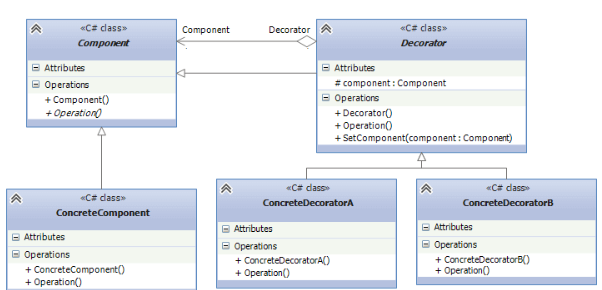


**Decorator**

**Scop** : Adaugă dinamic noi responsabilități unui obiect. Este o alternativă flexibilă la subclasare pentru a extinde funcționalitatea.

**Când de utilizat:**

* Când trebuie să adăugați în mod dinamic o nouă funcționalitate unui obiect. În acest caz, aceste caracteristici pot fi eliminate din obiect
* Când folosirea moștenirii este inacceptabilă. De exemplu, dacă trebuie să definim multe funcționalități diferite și să moștenim o clasă separată pentru fiecare funcționalitate, atunci structura clasei poate crește foarte mult. Poate crește și mai mult dacă trebuie să creăm clase care să implementeze toate combinațiile posibile de funcționalitate adăugată.



**Faţadă**

**Scop** : oferă o interfață unificată în locul unui set de interfețe al unui subsistem. O fațadă definește o interfață de nivel superior care face subsistemul mai ușor de utilizat.

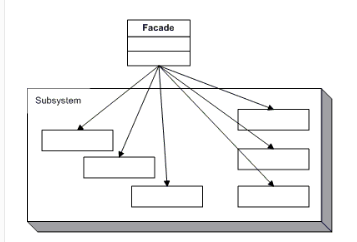
Modelul Fațadă grupează un grup de obiecte într-o singură interfață specializată și transmite apelurile sale de metodă către acele obiecte.

**Când de utilizat:**

* Când există un sistem complex și este necesar să se simplifice lucrul cu acesta. Fațada vă va permite să definiți un punct de interacțiune între client și sistem.
* Când este necesar să se reducă numărul de dependențe dintre client și un sistem complex. Obiectele de fațadă vă permit să separați, să izolați componentele sistemului de client și să le dezvoltați și să lucrați cu ele independent.
* Când trebuie să definiți subsisteme de componente într-un sistem complex. Crearea de fațade pentru componentele fiecărui subsistem individual va simplifica interacțiunea dintre ele și va crește independența lor unul față de celălalt.

**Diagrama unui exemplu de utilizare a modelului „Fațadă” din viața reală:**

O fațadă definește o interfață unificată de nivel superior la un subsistem care îl face mai ușor de utilizat. Consumatorii se confruntă cu o fațadă atunci când comandă dintr-un catalog. Un consumator sună la un număr și vorbește cu un reprezentant al serviciului pentru clienți. Reprezentantul serviciului pentru clienți acționează ca un front, oferind o interfață pentru departamentele de onorare a comenzilor, facturare și expediere.

****

# Modele de design comportamental

Tipare comportamentale - definesc algoritmi și interacțiunea dintre clase și obiecte, adică comportamentul lor. Aceste șabloane includ următoarele:

* Lanț de responsabilitate
* Comanda
* Interpret
* Iterator
* Mediator
* Gardian (Memento)
* Observator
* Stat
* Strategie
* Metoda șablonului
* Vizitator

**Iterator**

**Scop** : Reprezintă accesul la toate elementele unui obiect compozit fără a-și expune reprezentarea internă

**Când de utilizat:**

* Când este necesar să ocoliți un obiect fără a-i dezvălui structura internă;
* Când există un set de obiecte compuse, și este necesar să se furnizeze o singură interfață pentru iterare peste ele;
* Când este necesar să se furnizeze mai multe iterații alternative ale aceluiași obiect;

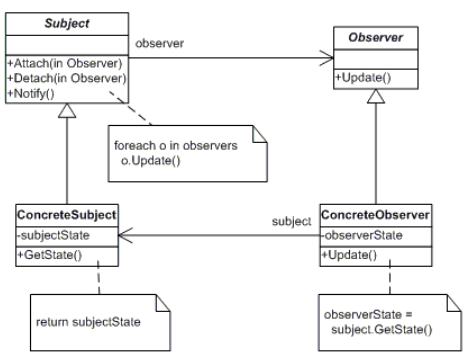
**Observator**

**Scop** : Definește o dependență unu-la-mulți între obiecte, astfel încât, atunci când starea unui obiect se schimbă, toate persoanele dependente de acesta sunt notificate și actualizate automat.

Cu alte cuvinte: observatorul anunță toate părțile interesate despre un eveniment care a avut loc sau despre o schimbare a stării acestuia.

**Când de utilizat:**

* Când sistemul este format din mai multe clase ale căror obiecte trebuie să fie în stări consistente
* Când schema generală de interacțiune între obiecte implică două părți: una trimite mesaje și este cea principală, cealaltă primește mesaje și reacționează la acestea. Separarea logicii ambelor părți le permite să fie considerate independent și utilizate separat una de cealaltă.
* Când există un singur obiect care trimite mesaje și mulți abonați care primesc mesaje. În același timp, numărul exact de abonați nu este cunoscut dinainte și programul se poate schimba pe parcursul programului.



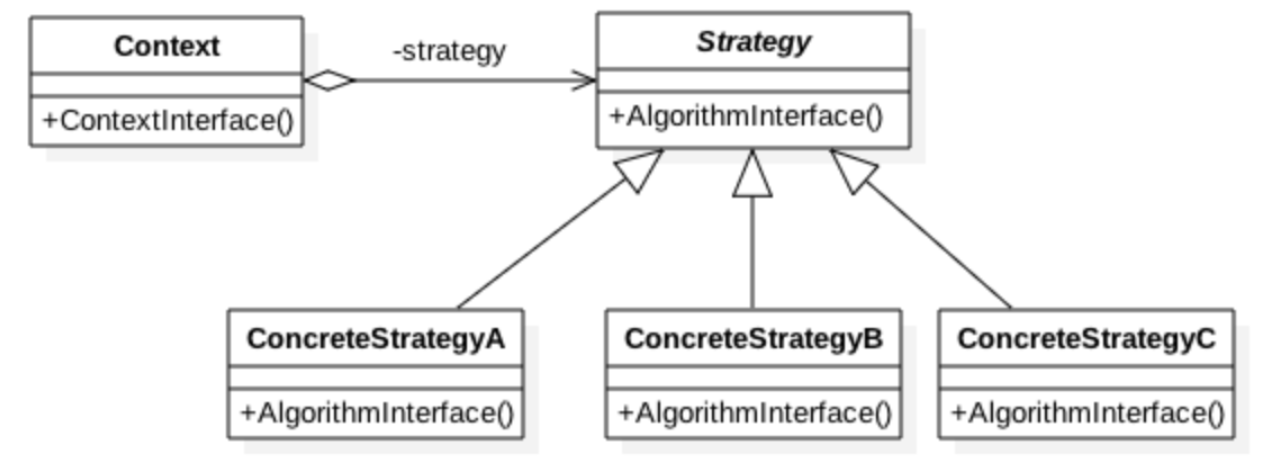
**Strategie**

**Scop** : Definește o familie de algoritmi, îi încapsulează pe fiecare și îi face interschimbabili. Strategia vă permite să schimbați algoritmii indiferent de clienții care îi folosesc.

Cu alte cuvinte : o strategie încapsulează un anumit comportament cu posibilitatea înlocuirii acestuia.

**Când de utilizat:**

* Când există mai multe clase similare care diferă ca comportament. Puteți seta o clasă principală și puteți pune diferite comportamente în clase separate și le puteți aplica dacă este necesar;
* Cand este necesar sa se ofere o alegere a mai multor solutii care pot fi schimbate cu usurinta in functie de conditii;
* Când este necesară modificarea comportamentului claselor și obiectelor în etapa de execuție a programului;
* Când o clasă care utilizează o anumită funcționalitate nu trebuie să știe nimic despre implementarea ei



# Scopul implementarii aplicației

Scopul temei de gestionare a unei universități este de a dezvolta un sistem care să faciliteze administrarea și gestionarea diferitelor aspecte ale unei instituții de învățământ superior. Acest sistem poate fi utilizat pentru a efectua următoarele funcționalități:

* Înrolarea studenților: Sistemul permite înrolarea studenților în cadrul universității. Aceasta poate include adăugarea informațiilor personale ale studenților și alocarea unui număr de identificare unic pentru fiecare student.
* Angajarea profesorilor: Sistemul permite angajarea profesorilor în cadrul universității. Aceasta implică înregistrarea informațiilor personale ale profesorilor, precum și asocierea acestora cu disciplinele pe care le predau.
* Gestionarea notelor: Sistemul permite introducerea și gestionarea notelor studenților. Profesorii pot introduce notele obținute de studenți în diferite discipline, iar acestea pot fi stocate și afișate ulterior în vederea evaluării performanțelor academice.
* Vizualizarea informațiilor despre studenți și profesori: Sistemul permite vizualizarea detaliilor despre studenți și profesori, cum ar fi numele, notele obținute, disciplinele predate și alte informații relevante.
* Decorarea studenților: Sistemul permite aplicarea de decorațiuni speciale asupra studenților, utilizând decoratorul GoodStudentDecorator. Acesta poate fi utilizat pentru a evidenția studenții cu rezultate academice remarcabile sau alte criterii specifice.
* Observarea modificărilor în numărul de studenți: Sistemul include un mecanism de observare prin intermediul căruia se poate monitoriza și notifica observatorii despre orice modificare în numărul de studenți înregistrați în universitate.

Prin implementarea acestor funcționalități, sistemul de gestionare a universității facilitează procesul de administrare și urmărire a informațiilor relevante despre studenți, profesori și performanțele academice, permițând astfel o gestionare mai eficientă și o interacțiune mai facilă cu toate părțile implicate în cadrul universității.

# Proiectarea și dezvoltarea aplicației

**Singleton** (Singleton Design Pattern):

Patternul Singleton este utilizat în codul de mai sus pentru a asigura că există doar o singură instanță a clasei Universityși pentru a oferi un punct de acces global la această instanță.

class University:

\_instance = None

@staticmethod

def get\_instance():

if not University.\_instance:

University()

return University.\_instance

def \_\_init\_\_(self):

if University.\_instance:

raise Exception("University class is a singleton.")

else:

University.\_instance = self

self.students = []

self.teachers = []

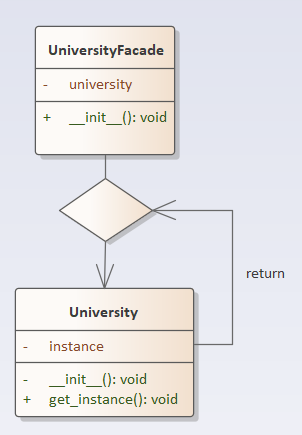
În codul de mai sus, clasa University implementează Singleton prin următoarele modificări:

Se declară o variabilă de clasă \_instance care va reține instanța clasei School. Inițial, această variabilă este setată la None.

Se definește o metodă statică get\_instance() care returnează instanța clasei School. Dacă variabila \_instance este None, înseamnă că nu a fost creată încă o instanță a clasei School, așa că se creează una nouă și se atribuie variabilei \_instance. În caz contrar, înseamnă că există deja o instanță și se returnează acea instanță.

Constructorul clasei Universityeste modificat pentru a nu permite crearea mai multor instanțe. Dacă variabila \_instance are deja o valoare, înseamnă că există deja o instanță și se aruncă o excepție cu mesajul "Universityclass is a singleton. Use get\_instance() method to get the instance." În caz contrar, se atribuie instanța curentă variabilei \_instance.

Prin implementarea Singleton, se asigură că există o singură instanță a clasei University în cadrul aplicației. Acest lucru poate fi util în situațiile în care trebuie să avem o singură școală în întreaga aplicație și să ne asigurăm că toate operațiile sunt realizate pe aceeași instanță.



În codul din clasa UniversityFacade, putem vedea cum este utilizată metoda get\_instance() pentru a obține instanța clasei Universityîn cadrul obiectului facade de tip SchoolFacade. Astfel, UniversityFacade utilizează aceeași instanță a clasei University pentru a efectua operațiile de înregistrare a studenților și angajare a profesorilor.

**Fabrică** (Factory Design Pattern):

Patternul Factory este utilizat în codul de mai sus pentru a crea obiecte de tipul Student sau Teacher în funcție de un anumit tip (person\_type). Aceasta oferă o modalitate de a separa logica de creare a obiectelor de restul codului.

class Person:

def \_\_init\_\_(self, name):

self.name = name

class Student(Person):

def \_\_init\_\_(self, name):

super().\_\_init\_\_(name)

self.grades = []

def add\_grade(self, grade):

self.grades.append(grade)

def get\_average\_grade(self):

if len(self.grades) == 0:

return 0

return sum(self.grades) / len(self.grades)

def \_\_str\_\_(self):

return f"Student: {self.name}"

class Teacher(Person):

def \_\_init\_\_(self, name):

super().\_\_init\_\_(name)

self.subjects = []

def add\_subject(self, subject):

self.subjects.append(subject)

def get\_subjects(self):

return self.subjects

def \_\_str\_\_(self):

return f"Teacher: {self.name}"

class PersonFactory:

def create\_person(self, name, person\_type):

if person\_type == "student":

return Student(name)

elif person\_type == "teacher":

return Teacher(name)

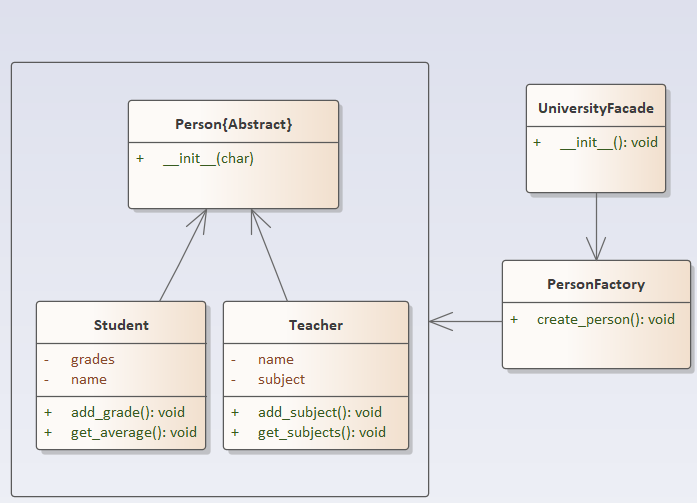
else:

raise ValueError("Invalid person type.")

În codul de mai sus, clasa PersonFactory este responsabilă pentru crearea obiectelor de tip Person (în cazul nostru, Student și Teacher). Metoda create\_person primește un nume și un tip de persoană și returnează un obiect de tipul respectiv. Dacă tipul persoanei este "student", atunci este creat un obiect de tip Student, iar dacă este "teacher", este creat un obiect de tip Teacher. Dacă tipul persoanei nu este valid, este aruncată o excepție cu mesajul "Invalid person type."

Prin utilizarea unui Factory, putem ascunde detalii specifice de creare a obiectelor într-o singură locație, facilitând extensibilitatea și menținerea codului. De exemplu, dacă dorim să adăugăm un nou tip de persoană în viitor, putem extinde clasa PersonFactory și să adăugăm o nouă verificare de tip pentru acesta.

În codul din clasa UniversityFacade, putem vedea cum este utilizată clasa PersonFactory pentru a crea obiecte de tip Student sau Teacher în funcție de alegerea utilizatorului. Astfel, putem separa logica de creare a obiectelor de logica principală a aplicației.



**Decorator** (Decorator Design Pattern):

Patternul Decorator este utilizat în codul de mai sus pentru a extinde funcționalitatea unui obiect existent fără a modifica clasa de bază. Acesta permite atașarea de comportamente suplimentare la obiecte într-un mod flexibil și dinamic.

class PersonDecorator(Person):

def \_\_init\_\_(self, person):

self.person = person

def \_\_str\_\_(self):

return self.person.\_\_str\_\_()

class GoodStudentDecorator(PersonDecorator):

def \_\_str\_\_(self):

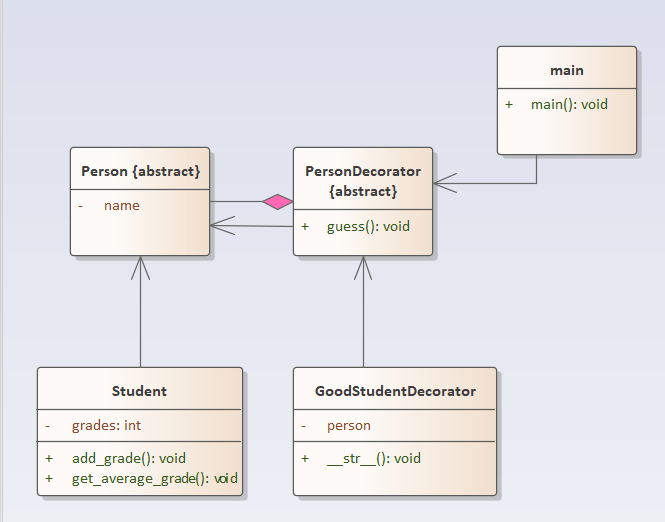
return f"Good {self.person.\_\_str\_\_()}"

În codul de mai sus, clasa PersonDecorator este clasa de bază pentru decoratori. Aceasta primește un obiect de tip Person în constructor și îl stochează într-o variabilă de instanță person. Aceasta oferă o metodă suprascrisă \_\_str\_\_() pentru a returna reprezentarea subiectului decorat.

De exemplu, clasa GoodStudentDecorator extinde comportamentul unui student prin adăugarea cuvântului "Good" înainte de reprezentarea sa. Aceasta derivă din clasa PersonDecorator și suprascrie metoda \_\_str\_\_() pentru a returna reprezentarea decorată a subiectului.

În metoda main(), se utilizează decoratorul pentru a decora un obiect de tip Student. După crearea obiectului student, acesta este învelit într-un decorator de tip GoodStudentDecorator, care adaugă cuvântul "Good" în reprezentarea sa. Astfel, putem extinde comportamentul unui obiect existent prin atașarea de decoratori fără a modifica clasa de bază.

Această abordare permite adăugarea de funcționalități suplimentare la obiecte într-un mod modular și flexibil. Putem adăuga sau elimina decoratori la nevoie, obținând combinații diferite de comportamente extinse pentru obiectele de bază, fără a afecta funcționalitatea lor de bază.



**Fațadă** (Facade Design Pattern):

Patternul Facade (Fațadă) este utilizat în codul de mai sus pentru a oferi o interfață simplificată către un subsistem complex, cum ar fi gestionarea unei școli. Fațada ascunde detaliile complexe ale subsistemului și oferă metode simple și coerente pentru utilizatorul final.

class UniversityFacade:

def \_\_init\_\_(self):

self.university = University.get\_instance()

self.person\_factory = PersonFactory()

def enroll\_student(self, name):

student = self.person\_factory.create\_person(name, "student")

self.university.students.append(student)

def hire\_teacher(self, name):

teacher = self.person\_factory.create\_person(name, "teacher")

self.university.teachers.append(teacher)

def display\_students(self):

for student in self.university.students:

print(student)

print("Grades:", student.grades)

print("Average Grade:", student.get\_average\_grade())

print()

def enter\_grades(self):

student\_name = input("Enter student name: ")

found\_student = None

for student in self.university.students:

if student.name == student\_name:

found\_student = student

break

if found\_student:

grades\_str = input("Enter grades (comma-separated): ")

grades = [float(grade.strip()) for grade in grades\_str.split(",")]

found\_student.grades.extend(grades)

else:

print("Student not found.")

if grades != 0:

print("Grades added successfully.")

def display\_grades(self):

student\_name = input("Enter student name: ")

for student in self.university.students:

if student.name == student\_name:

print(f"Grades for {student.name}:")

for grades in student.grades:

print(grades)

return

print("Student not found.")

def display\_teachers(self):

for teacher in self.university.teachers:

print(teacher)

def display\_students\_with\_only\_tens(self):

print("Students with only grade 10:")

student\_list = StudentList(self.university.students)

student\_iterator = iter(student\_list)

try:

while True:

student = next(student\_iterator)

print(student)

print("Grades:", student.grades)

print("Average Grade:", student.get\_average\_grade())

print()

except StopIteration:

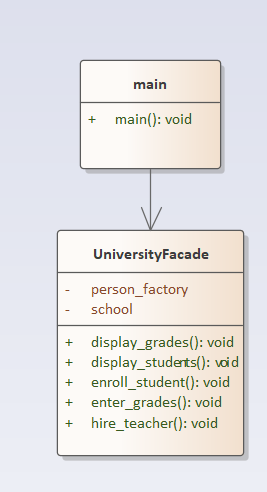
Pass

În codul de mai sus, clasa SchoolFacade acționează ca o fațadă pentru gestionarea școlii. Aceasta are o referință la obiectul singleton Universityși la obiectul PersonFactory pentru a efectua operațiuni precum înregistrarea studenților și angajarea profesorilor.

Clasa SchoolFacade oferă metode simple și corespunzătoare, cum ar fi enroll\_student(), hire\_teacher(), display\_students(), display\_teachers(), care sunt folosite pentru a interacționa cu subsistemul școlii. Aceste metode ascund detaliile de implementare a subsistemului și oferă utilizatorului final o interfață simplă pentru a interacționa cu școala.

De exemplu, metoda enroll\_student(name) din SchoolFacade utilizează PersonFactory pentru a crea un obiect Student cu numele specificat și îl înregistrează în lista de studenți a școlii. Utilizatorul final nu trebuie să interacționeze direct cu obiectele Universityși PersonFactory, ci doar cu fațada SchoolFacade, care gestionează în mod transparent operațiunile necesare.

Astfel, prin utilizarea patternului Facade, se obține o separare clară între interfața utilizatorului și detaliiile de implementare ale subsistemului. Acesta simplifică utilizarea subsistemului complex și reduce cuplarea între componente. De asemenea, permite fațada să ofere o interfață simplă și coerentă, adaptată nevoilor utilizatorului final.



**Observator** (Observer Design Pattern):

Patternul Observer (Observator) este utilizat în codul de mai sus pentru a implementa un mecanism de notificare automată atunci când se produce o anumită modificare într-un obiect sau subsistem. Aceasta permite altor obiecte sau componente să fie informate automat despre modificări și să reacționeze în consecință.

class Observer:

def update(self):

pass

class StudentCountObserver(Observer):

def \_\_init\_\_(self, facade):

self.facade = facade

def update(self):

student\_count = len(self.facade.university.students)

print(f"Student count changed. Total students: {student\_count}")

class UniversityObservable:

def \_\_init\_\_(self):

self.observers = []

def add\_observer(self, observer):

self.observers.append(observer)

def remove\_observer(self, observer):

self.observers.remove(observer)

def notify\_observers(self):

for observer in self.observers:

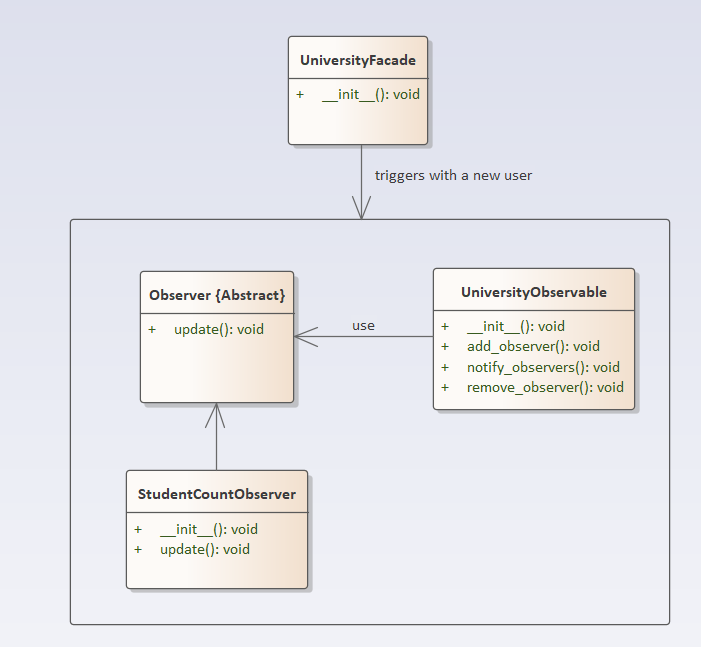
observer.update()

În codul de mai sus, clasa Observer reprezintă o interfață pentru observatori. Aceasta definește o metodă update() care poate fi implementată de către clasele concrete pentru a reacționa la evenimente sau modificări.

Clasa StudentCountObserver este o implementare concretă a Observer și reprezintă un observator specific pentru numărul de studenți. Aceasta primește o referință la obiectul SchoolFacade în constructor și implementează metoda update() pentru a afișa numărul actualizat de studenți atunci când are loc o modificare.

Clasa SchoolObservable reprezintă obiectul observabil sau sursa de evenimente. Aceasta menține o listă de observatori și oferă metode pentru adăugarea, eliminarea și notificarea observatorilor. Metoda add\_observer(observer) permite adăugarea unui observator la lista de observatori, remove\_observer(observer) permite eliminarea unui observator, iar notify\_observers() notifică toți observatorii înregistrati apelând metoda update() pe fiecare.

În metoda main(), se creează un obiect SchoolObservable și se înregistrează un observator de tip StudentCountObserver în aceasta, utilizând obiectul facade de tip SchoolFacade ca sursă de evenimente. Astfel, atunci când se înregistrează un nou student prin apelarea metodei enroll\_student(name) în SchoolFacade, sursa de evenimente SchoolObservable notifică observatorul StudentCountObserver, care afișează numărul actualizat de studenți.



Clasa SchoolFacade declanșează notificarea către observatori atunci când apar schimbări, cum ar fi înscrierea unui nou elev.

Patternul Observer permite decuplarea observatorilor de subiectul observabil și permite implementarea unei logici flexibile de notificare. Observatorii pot fi adăugați sau eliminați dinamic, iar subiectul observabil poate notifica observatorii în funcție de evenimente specifice sau modificări în subsistem. Acest pattern este util în situații în care există dependențe între obiecte și este necesară comunicarea și sincronizarea între ele.

**Iterator** (Iterator Design Pattern):

Patternul Iterator (Iteratorul) este utilizat în codul de mai sus pentru a oferi o modalitate de a itera în mod eficient peste elementele unei colecții fără a expune detaliile de implementare ale colecției. Acesta separă iterația de structura subiacentă a colecției și oferă o interfață unificată pentru a accesa și itera prin elementele colecției.

class Iterator:

def \_\_next\_\_(self):

pass

class StudentIterator(Iterator):

def \_\_init\_\_(self, students):

self.students = students

self.index = 0

def \_\_next\_\_(self):

while self.index < len(self.students):

student = self.students[self.index]

grades = student.grades

if all(grade == 10 for grade in grades):

self.index += 1

return student

self.index += 1

raise StopIteration

class StudentList:

def \_\_init\_\_(self, students):

self.students = students

def \_\_iter\_\_(self):

return StudentIterator(self.students)

În codul de mai sus, clasa StudentIterator reprezintă iteratorul care implementează iterația peste colecția de studenți. Acesta menține o referință la colecția de studenți și un indice care urmărește poziția curentă în iterație. Metoda \_\_next\_\_() este implementată pentru a returna următorul student în iterație și pentru a actualiza indicele.

Clasa StudentList reprezintă colecția de studenți. Aceasta primește o listă de studenți în constructor și implementează metoda \_\_iter\_\_() care returnează un obiect de tip StudentIterator. Acesta oferă acces la iterația prin studenții colecției.

În metoda clasa UniversityFacade,, se utilizează iteratorul pentru a itera prin lista de studenți. Prin apelarea metodei \_\_iter\_\_() pe obiectul StudentList, se obține un obiect StudentIterator, care poate fi utilizat într-o buclă for pentru a itera prin studenți. La fiecare iterație, se obține următorul student utilizând metoda \_\_next\_\_() a iteratorului.

Patternul Iterator permite separarea logicii de iterație de structura colecției și oferă o modalitate simplă și coerentă de a itera prin elementele colecției fără a expune detaliile de implementare. Acesta oferă un mod eficient și sigur de a accesa elementele colecției și facilitează implementarea algoritmilor care necesită iterație prin elemente.

# Concluzii

În cadrul proiectului de an la disciplina “Tehnici și Mecanisme de Proiectare Software”, am creat o aplicație de gestionare a unei universități în care am aplicat deiverse tipuri de modele de proiectare de tip creaționale, structurale și comportamentale. Implementarea acestor pattern-urilor de design în aplicația de gestionare a unei universități a fost de a îmbunătăți arhitectura, flexibilitatea și eficiența sistemului, precum și de a facilita administrarea și monitorizarea activităților universitare. Prin utilizarea acestor pattern-uri, am urmărit:

* Crearea unei structuri coerente și bine organizate a aplicației, separând responsabilitățile și facilitând reutilizarea componentelor.
* Asigurarea unui cod mai modular, mai ușor de înțeles și de întreținut, datorită separării funcționalităților în clase și metode corespunzătoare.
* Reducerea cuplajului între componentele sistemului, permițând schimbări și extinderi ulterioare cu un impact minim asupra restului aplicației.
* Oferirea de mecanisme standardizate și flexibile pentru crearea, gestionarea și interacțiunea cu obiecte și entități, precum crearea de studenți și profesori, adăugarea de note și afișarea rezultatelor academice.
* Monitorizarea și notificarea automată a observatorilor despre modificările în numărul de studenți înregistrați, furnizând informații actualizate și relevante despre starea universității.
* Simplificarea interacțiunii utilizatorilor cu sistemul prin intermediul unei interfețe coerente și ușor de utilizat.

Ca rezultat al aplicării acestor pattern-uri de design, am obținut o aplicație de gestionare a universității care prezintă o multitudine de beneficii. Codul modular, bine structurat și ușor de extins, permite adăugarea de noi funcționalități și adaptarea la schimbările ulterioare. Interfața simplă și coerentă pentru utilizatori, facilitează navigarea și interacțiunea cu sistemul. Administrare eficientă a datelor despre studenți, profesori și rezultatele academice, cu mecanisme clare și standardizate pentru introducerea, stocarea și afișarea acestora. Este un sistem care permite monitorizarea și raportarea automată a modificărilor în numărul de studenți, oferind o imagine actualizată și precisă a situației universității.

Aplicarea și implementarea pattern-urilor de design în aplicația de gestionare a unei universități a condus la obținerea unui sistem robust, modular și eficient, care facilitează administrarea, monitorizarea și interacțiunea cu informațiile academice și contribuie la îmbunătățirea experienței utilizatorilor.

# Bibliografie

1. Model de design: ce este și de ce să-l folosești? [citat 02.06.2023] Disponibil: https://ryax.tech/design-pattern-what-is-it-and-why-use-it/#:~:text=The%20usefulness%20of%20using%20a,every%20time%20a%20problem%20arises.
2. Шаблоны проектирования "банды четырёх (GoF)" [citat 04.06.2023] Disponibil:

https://bool.dev/blog/detail/gof-design-patterns