**Ministerul Educaţiei, Culturii și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Departamentul Informatică și Ingineria Sistemelor**

**RAPORT**

**Proiect de an la TMPS**

A efectuat:

st. gr. TI-203 Gaida Maxim

A verificat:

asist. univ. Mihai Gaidau

Chişinău – 2023

Cuprins :

[INTRODUCERA 3](#_Toc136588721)

[DESCRIEREA PRICIPIULUI DE LUCRU AL TEMATICII ALESE 4](#_Toc136588723)

[DEMONSTRAREA PE UN EXEMPLU CONCRET PRINCIPIUL DE LUCRU 5](#_Toc136588724)

[MODELAREA SISTEMULUI UML 8](#_Toc136588725)

[CONCLUZIE 12](#_Toc136588726)

[BIBLIOGRAFIE 14](#_Toc136588727)

# INTRODUCERA

Design Patterns sunt soluții recurente la probleme comune de proiectare de software. Acestea oferă un set de reguli și abordări pentru dezvoltarea și organizarea codului, facilitând astfel reutilizarea, întreținerea și extinderea acestuia. În cele ce urmează, vom explora în detaliu fiecare design pattern menționat: Singleton, Builder, Factory, Adapter, Composite, Decorator, Chain of Responsibility și Strategy.

**Singleton** este un pattern de creare care permite obținerea unei singure instanțe a unei clase într-un program. Acesta restricționează crearea de noi obiecte și asigură că doar o singură instanță a clasei este disponibilă în întregul sistem. Singleton-ul este util în situații în care trebuie să existe o singură instanță a unei clase care să ofere un punct centralizat de acces la resurse sau servicii.

**Builder** este un pattern de creare care separă procesul de construcție a unui obiect complex de reprezentarea sa. Prin intermediul acestui pattern, putem construi obiecte complexe pas cu pas, oferindu-ne flexibilitate și control asupra diferitelor combinații de parametri și opțiuni. Builder-ul este util atunci când există multiple variante de construcție a unui obiect și când dorim să separăm logica de construcție de implementarea obiectului în sine.

**Factory** este un pattern de creare care oferă o interfață comună pentru crearea unor obiecte dintr-o familie sau ierarhie. Această interfață permite crearea obiectelor fără a specifica clasa exactă a obiectului care va fi creat. Astfel, Factory ascunde detaliile de creare a obiectelor și oferă o metodă flexibilă și extensibilă pentru instanțierea acestora.

**Adapter** este un pattern structural care permite obiectelor cu interfețe incompatibile să colaboreze. Acesta convertește interfața unei clase într-o altă interfață așteptată de client. Adaptorul funcționează ca un intermediar între două obiecte care altfel nu ar putea să comunice direct. Prin intermediul acestui pattern, putem adapta și reutiliza clase existente în cadrul unei arhitecturi fără a modifica codul sursă original.

**Composite** este un pattern structural care permite gruparea obiectelor într-o ierarhie în formă de arbore, astfel încât acestea să fie tratate ca un singur obiect. Astfel, putem trata atât obiectele individuale, cât și colecțiile acestora într-un mod uniform. Composite este util atunci când dorim să reprezentăm o structură de obiecte într-o formă ierarhică și să operăm asupra acestora într-un mod recursiv.

**Decorator** este un pattern structural care permite adăugarea de comportamente suplimentare la un obiect existent într-un mod dinamic. Acesta oferă o alternativă la moștenirea clasică, permițând extinderea funcționalității unui obiect prin încapsularea acestuia în obiecte decorator. Decoratorul se aplică într-un mod transparent obiectului de bază, permițând adăugarea și eliminarea de comportamente la runtime.

**Chain of Responsibility** este un pattern comportamental care permite transmiterea unei solicitări într-o serie de obiecte potențial responsabile, până când unul dintre ele o preia și o procesează. Fiecare obiect în lanț are posibilitatea să prelucreze solicitarea sau să o transmită mai departe în lanț. Acest pattern ajută la desprinderea senderului de destinatarul real al solicitării și oferă flexibilitate în gestionarea și configurarea responsabilităților.

**Strategy** este un pattern comportamental care permite definirea mai multor algoritmi sau strategii și selectarea uneia dintre ele la runtime. Acesta separă algoritmii de utilizatorul care îi utilizează, permițând înlocuirea sau schimbarea strategiei fără a afecta codul clientului. Strategy este util atunci când dorim să implementăm diferite variante ale aceluiași algoritm și să le selectăm în funcție de anumite condiții sau preferințe.

Astfel, folosirea tuturor acestor patternuri împreună într-un sistem software poate aduce numeroase beneficii și poate contribui la crearea unei arhitecturi robuste, flexibile și ușor de întreținut. Iată câteva motive importante pentru utilizarea lor în combinație:

1.Separarea responsabilităților: Utilizarea acestor patternuri permite separarea clară a responsabilităților și funcționalităților în diferite clase și componente. Astfel, avem o structură mai modulară și mai ușor de înțeles, iar fiecare componentă este specializată într-un anumit aspect al sistemului.

2.Flexibilitate și extensibilitate: Patternurile oferă o flexibilitate sporită în dezvoltarea și extinderea sistemului. De exemplu, utilizarea patternului Factory ne permite să adăugăm noi tipuri de obiecte într-un mod ușor și să extindem logica de creare fără a modifica codul existent. Combinarea acestui pattern cu Builder ne oferă posibilitatea de a crea obiecte complexe cu diferite combinații de parametri și opțiuni.

3.Reutilizare și modularitate: Patternurile facilitează reutilizarea componentelor și a logicilor de implementare. De exemplu, prin utilizarea patternului Adapter, putem adapta și reutiliza clase existente într-o nouă arhitectură, fără a modifica codul sursă original. Prin utilizarea patternului Composite, putem trata atât obiectele individuale, cât și colecțiile acestora într-un mod uniform, facilitând astfel reutilizarea și modularitatea în codul nostru.

# DESCRIEREA PRICIPIULUI DE LUCRU AL TEMATICII ALESE

Această aplicație este un manager de sarcini (task manager) simplu, care permite utilizatorilor să adauge, să vizualizeze și să șteargă sarcini. Scopul său principal este să ofere o interfață utilizator intuitivă și funcționalități de bază pentru gestionarea sarcinilor personale sau profesionale.

Aplicația folosește diverse modele de design pentru a implementa funcționalitățile sale și pentru a asigura o structură modulară și flexibilitate în dezvoltarea și gestionarea codului. Modelele de design utilizate în acest proiect includ Singleton pentru gestionarea centralizată a sarcinilor, Builder pentru crearea flexibilă a obiectelor de sarcină, Factory pentru crearea obiectelor de sarcină prin intermediul unei fabrici, Adapter pentru adaptarea interfeței obiectelor de sarcină, Composite pentru gestionarea grupurilor de sarcini, Decorator pentru extinderea funcționalității sarcinilor, Chain of Responsibility pentru tratarea în lanț a sarcinilor și Strategy pentru sortarea dinamică a sarcinilor.

1. Singleton Pattern:
   * Obiectiv: Asigurarea că există o singură instanță a clasei **TaskManager** în întreaga aplicație.
   * Beneficii: Centralizarea gestionării sarcinilor și asigurarea unui acces global la instanța **TaskManager**.
2. Builder Pattern:
   * Obiectiv: Separarea procesului de construcție a obiectelor complexe de reprezentarea lor.
   * Beneficii: Permite crearea flexibilă a obiectelor de sarcină, prin pas cu pas, cu opțiuni variabile și posibilitatea de a crea diferite reprezentări ale sarcinilor.
3. Factory Pattern:
   * Obiectiv: Furnizarea unei interfețe pentru crearea obiectelor fără a specifica clasele lor concrete.
   * Beneficii: Permite abstractizarea procesului de creare a sarcinilor, facilitând posibilitatea de a schimba implementarea și de a crea diverse tipuri de sarcini prin intermediul fabricii.
4. Adapter Pattern:
   * Obiectiv: Permiterea colaborării între obiecte cu interfețe incompatibile prin împachetarea unui obiect cu altul.
   * Beneficii: Adaptarea interfeței obiectelor de sarcină la o altă formă sau interfață, facilitând integrarea și utilizarea acestora în diverse contexte.
5. Composite Pattern:
   * Obiectiv: Construirea unei ierarhii de obiecte într-o structură de arbore pentru a reprezenta ierarhii de parte-total.
   * Beneficii: Permite gestionarea uniformă a sarcinilor individuale și a grupurilor de sarcini, tratându-le ca entități omogene și oferind operații comune pentru toate.
6. Decorator Pattern:
   * Obiectiv: Extinderea funcționalității unui obiect existent fără a modifica clasa sa de bază.
   * Beneficii: Permite adăugarea de comportamente suplimentare sau funcționalități la obiectele de sarcină, fără a modifica structura sau comportamentul lor inițial.
7. Chain of Responsibility Pattern:
   * Obiectiv: Permiterea unui obiect să transmită o cerere printr-o lanț de potențiali handleri până când aceasta este tratată sau ajunge la capătul lanțului.
   * Beneficii: Simplificarea procesului de gestionare și tratare a diferitelor aspecte ale sarcinilor prin intermediul unei lanțuri de handleri specializați.
8. Strategy Pattern:
   * Obiectiv: Definirea unei familii de algoritmi interșanjabili și încapsularea fiecăruia într-o clasă separată.
   * Beneficii: Permite selectarea și schimbul dinamic al algoritmilor de sortare a sarcinilor, oferind flexibilitate și extensibilitate în procesul de sortare.

Implementarea acestor design pattern-uri în proiect are ca obiectiv general crearea unei structuri modulare, flexibile și ușor de întreținut, care facilitează gestionarea sarcinilor și extinderea funcționalităților ulterioare ale aplicației de gestionare a sarcinilor.

# DEMONSTRAREA PE UN EXEMPLU CONCRET PRINCIPIUL DE LUCRU

Imaginați-vă că sunteți un utilizator care dorește să folosească aplicația pentru a gestiona sarcinile zilnice. În primul rând, deschideți aplicația și vedeți interfața principală a managerului de sarcini.

În interfața principală, aveți câteva câmpuri de intrare pentru a adăuga o nouă sarcină. Introduceți titlul sarcinii, descrierea și selectați data limită pentru finalizare din datele afișate de selectorul de dată.

După ce ați completat toate informațiile, apăsați butonul "Add Task". Aici intervine principiul de lucru al design pattern-ului Builder. Clasa TaskBuilder preia datele introduse de utilizator și le utilizează pentru a construi un obiect de tip Task. Cu ajutorul metodelor din clasa TaskBuilder, precum setTitle, setDescription și setDueDate, specificați atribuțiile dorite pentru obiectul Task. La final, apelați metoda build pentru a obține obiectul Task finalizat.

Acum, obiectul Task este creat și este adăugat în managerul de sarcini, care implementează principiul de lucru al design pattern-ului Singleton. Aplicația utilizează instanța unică a clasei TaskManager, obținută prin intermediul metodei statice shared, pentru a adăuga sarcina nou-creată în lista de sarcini existente.

După adăugarea sarcinii, câmpurile de intrare pentru titlu, descriere și dată limită se golesc automat, pregătindu-vă pentru adăugarea unei noi sarcini. În același timp, lista de sarcini din interfață se actualizează pentru a afișa toate sarcinile existente, inclusiv cea nou adăugată.

Pentru a șterge o sarcină, puteți utiliza funcționalitatea de eliminare oferită de aplicație. După ce apăsați butonul "Remove" al unei sarcini, principiul de lucru al design pattern-ului Adapter intră în joc. Clasa TaskAdapter asigură adaptarea interfeței obiectului Task la o formă mai convenabilă pentru gestionarea în cadrul aplicației. De exemplu, metoda getTitle a clasei TaskAdapter vă permite să obțineți titlul sarcinii pentru afișare sau alte scopuri.

Aplicația noastră de gestionare a sarcinilor utilizează și alte design pattern-uri, cum ar fi Composite pentru gestionarea grupurilor de sarcini, Decorator pentru extinderea funcționalității sarcinilor și Strategy pentru sortarea dinamică a sarcinilor. Aceste pattern-uri oferă flexibilitate, extensibilitate și modularitate în implementarea și gestionarea sarcinilor.

Pentru a exemplifica principiul de lucru al acestei aplicații, vom folosi un scenariu simplu:

1. Utilizatorul deschide aplicația Task Manager.
2. Pe ecranul principal, utilizatorul poate vedea un titlu mare "Task Manager" și un formular pentru adăugarea de noi sarcini.
3. Utilizatorul completează câmpurile "Title", "Description" și selectează o "Due Date" pentru sarcină.
4. Utilizatorul apasă butonul "Add Task" pentru a adăuga sarcina.
5. Sarcina nouă este creată utilizând Builder Pattern și este adăugată în TaskManager folosind Singleton Pattern.
6. Formularul se curăță, iar lista de sarcini este actualizată automat.
7. Utilizatorul poate vedea lista de sarcini în partea de jos a ecranului, cu informații precum titlul, descrierea și data limită a fiecărei sarcini.
8. Utilizatorul poate derula în sus și în jos pentru a vizualiza toate sarcinile.
9. Pentru a șterge o sarcină, utilizatorul apasă pe butonul "Delete" lângă sarcina respectivă.
10. Sarcina este eliminată din lista de sarcini și actualizată automat în UI.
11. Utilizatorul poate adăuga, vizualiza și șterge sarcini în mod repetat, iar lista de sarcini este actualizată în timp real.

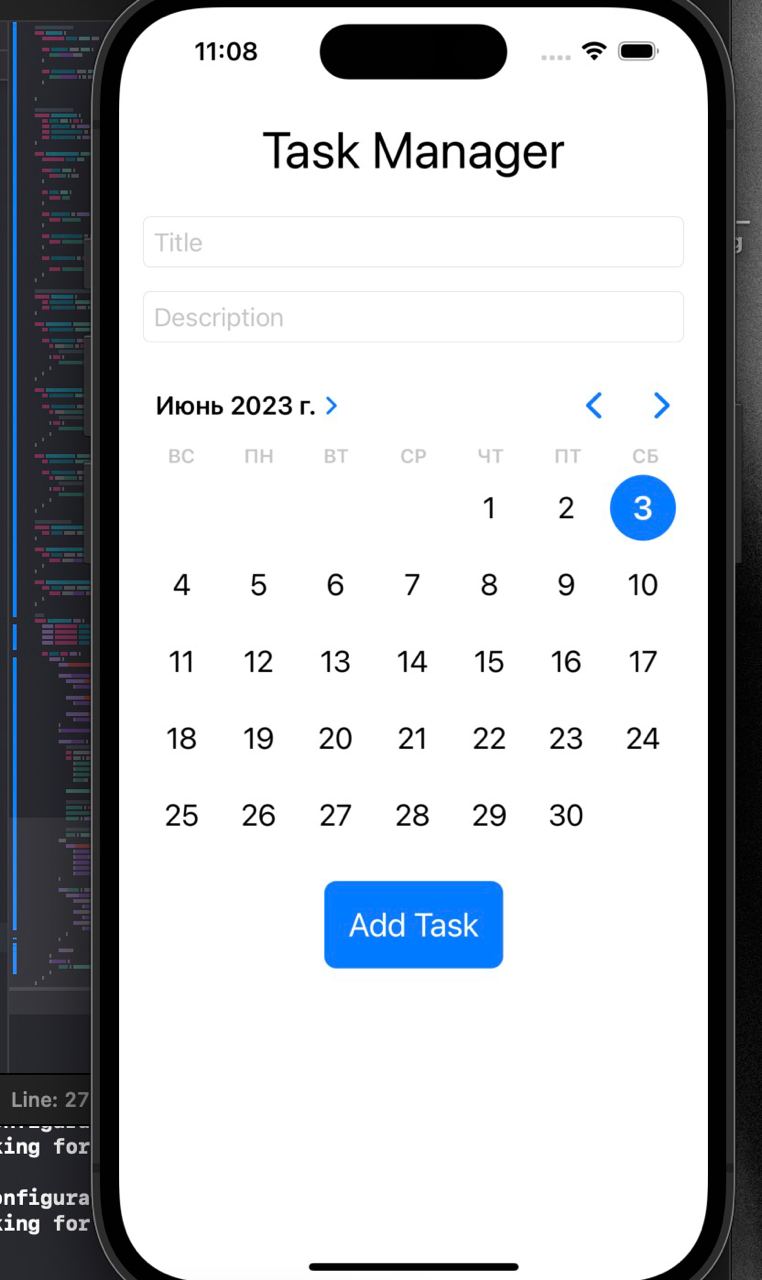


Figura 1.1 Interfata grafica a aplicatiei

În această aplicație, avem o interfață cu un câmp de introducere numit "Title" unde putem introduce denumirea unei sarcini. De asemenea, avem și un selector de dată calendaristică, care ne permite să selectăm o dată dorită pentru sarcină. După ce am introdus denumirea și am selectat data, putem apăsa pe butonul "Add Task".

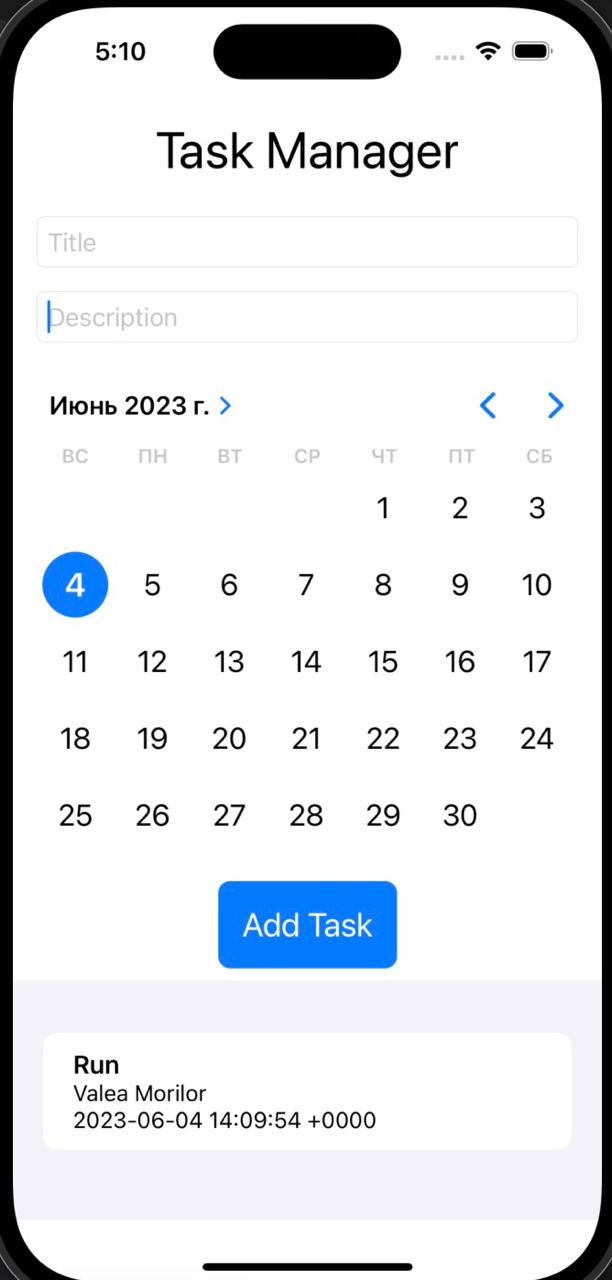


Figura 1.2 Adaugam task

După ce apăsăm butonul "Add Task", în partea de jos a interfeței, sarcina pe care am programat-o apare într-o listă sau într-un spațiu desemnat pentru a afișa sarcinile programate. Acest lucru ne permite să vedem în mod clar și organizat toate sarcinile pe care le-am adăugat.

# MODELAREA SISTEMULUI UML

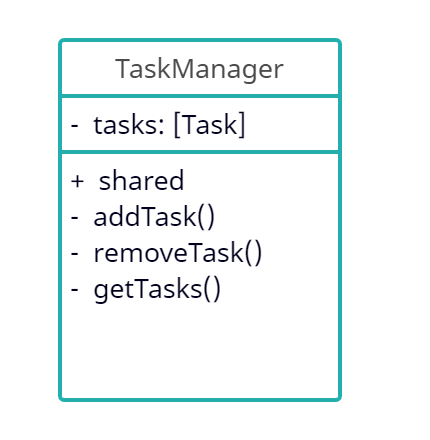


Figura 2.1 - Singleton Pattern

Singleton pattern asigură existența unei singure instanțe a clasei TaskManager pe parcursul întregii aplicații. Aceasta are un constructor privat și o instanță statică partajată (shared) care oferă acces la unica instanță a clasei. Clasa TaskManager menține o listă de sarcini (tasks) și furnizează metode pentru adăugarea, eliminarea și obținerea sarcinilor.

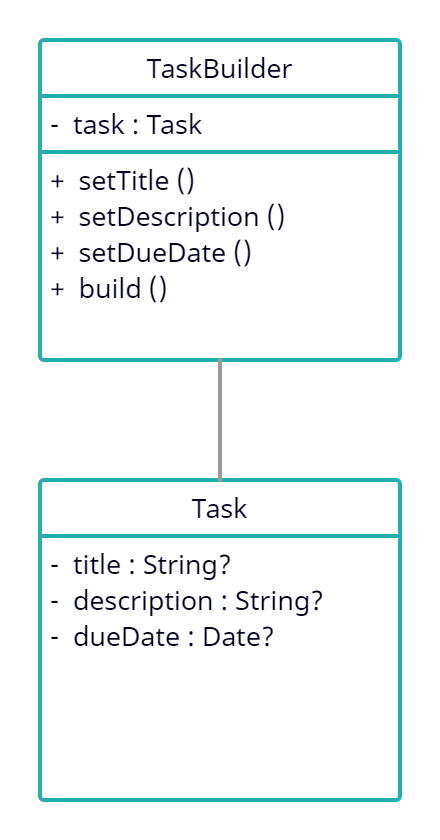


Figura 2.2 Builder Pattern

Builder pattern este utilizat pentru a construi un obiect de tip Task cu pași de inițializare flexibili. Clasa TaskBuilder are o interfață fluentă cu metode precum setTitle, setDescription și setDueDate pentru a seta proprietățile obiectului Task. Metoda build returnează obiectul Task complet construit.

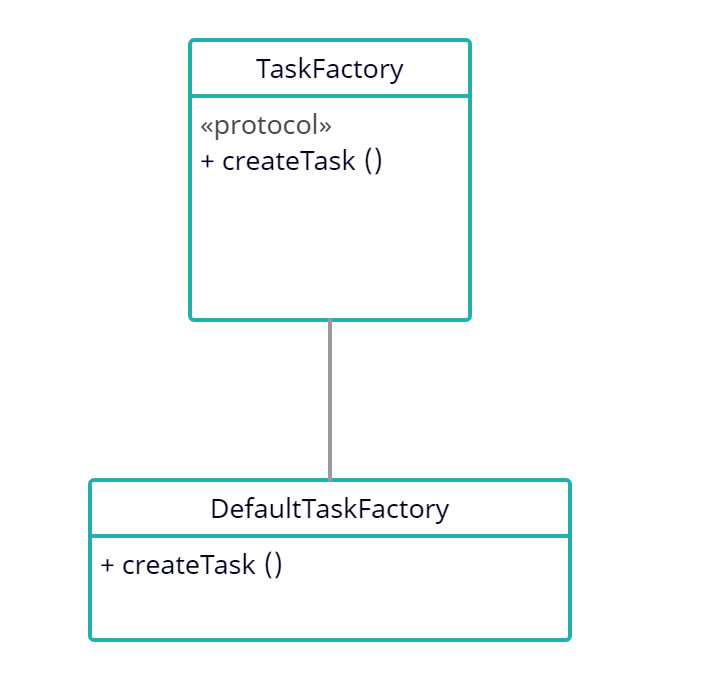


Figura 2.3 Factory Method

Factory pattern definește o interfață (TaskFactory) pentru crearea sarcinilor. Clasa DefaultTaskFactory implementează această interfață și oferă o implementare implicită pentru crearea unui obiect Task. Pattern-ul factory permite crearea sarcinilor fără a expune logica de creare.

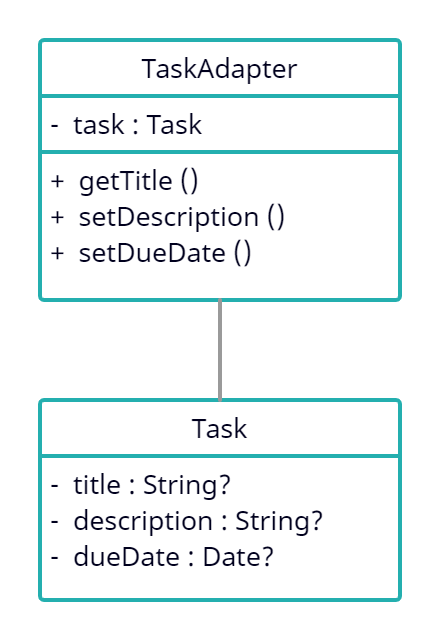


Figura 2.4 Adapter Pattern

Adapter pattern este utilizat pentru a adapta interfața unei clase existente pentru a fi compatibilă cu o altă interfață. În acest caz, clasa TaskAdapter înfășoară un obiect Task și furnizează metode (getTitle, getDescription, getDueDate) care adaptează interfața Task la o altă interfață.

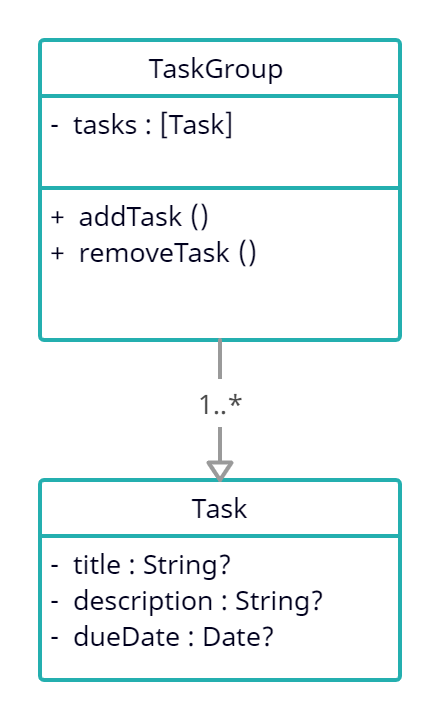


Figura 2.5 Composite Pattern

Composite pattern permite tratarea unui grup de obiecte ca un singur obiect. Clasa TaskGroup reprezintă un grup de sarcini și are metode pentru adăugarea și eliminarea sarcinilor. Menține o listă internă de obiecte Task și se comportă ca un singur obiect Task.

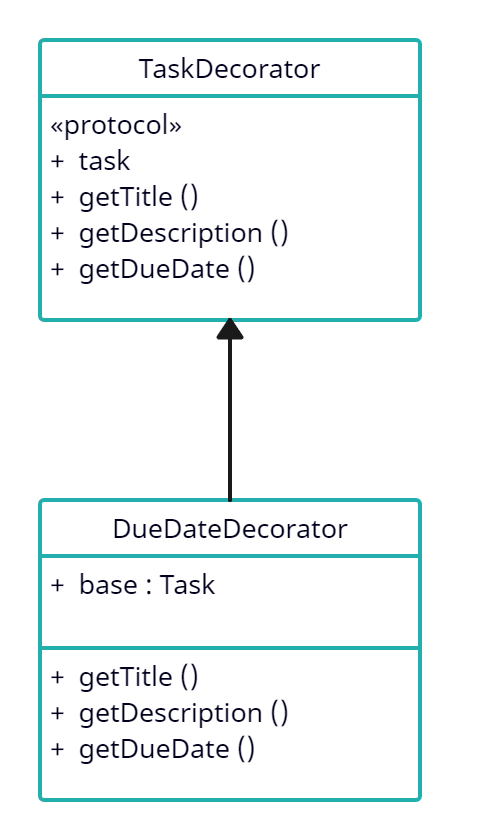


Figura 2.6 Decorator Pattern

Decorator pattern permite adăugarea de comportamente noi la un obiect în mod dinamic. Protocolul TaskDecorator definește interfața comună pentru decoratori, iar clasa DueDateDecorator implementează acest protocol. Decorează un obiect Task și adaugă comportament personalizat la metodele getTitle, getDescription și getDueDate.

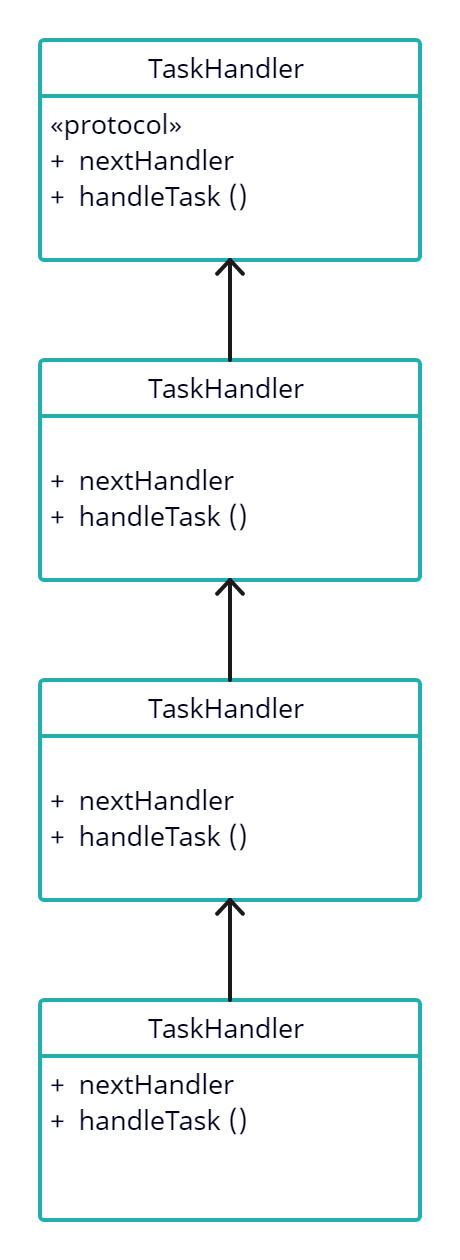


Figura 7 - Chain of Responsability:

Chain of Responsibility pattern permite unui lanț de obiecte handler să gestioneze o cerere într-un mod secvențial. Fiecare handler are o referință către următorul handler din lanț. În acest caz, clasele TitleHandler, DescriptionHandler și DueDateHandler gestionează aspecte specifice ale unui obiect Task. Verifică dacă o proprietate specifică lipsește și, în caz afirmativ, efectuează acțiunea necesară sau delegă cererea către următorul handler din lanț.

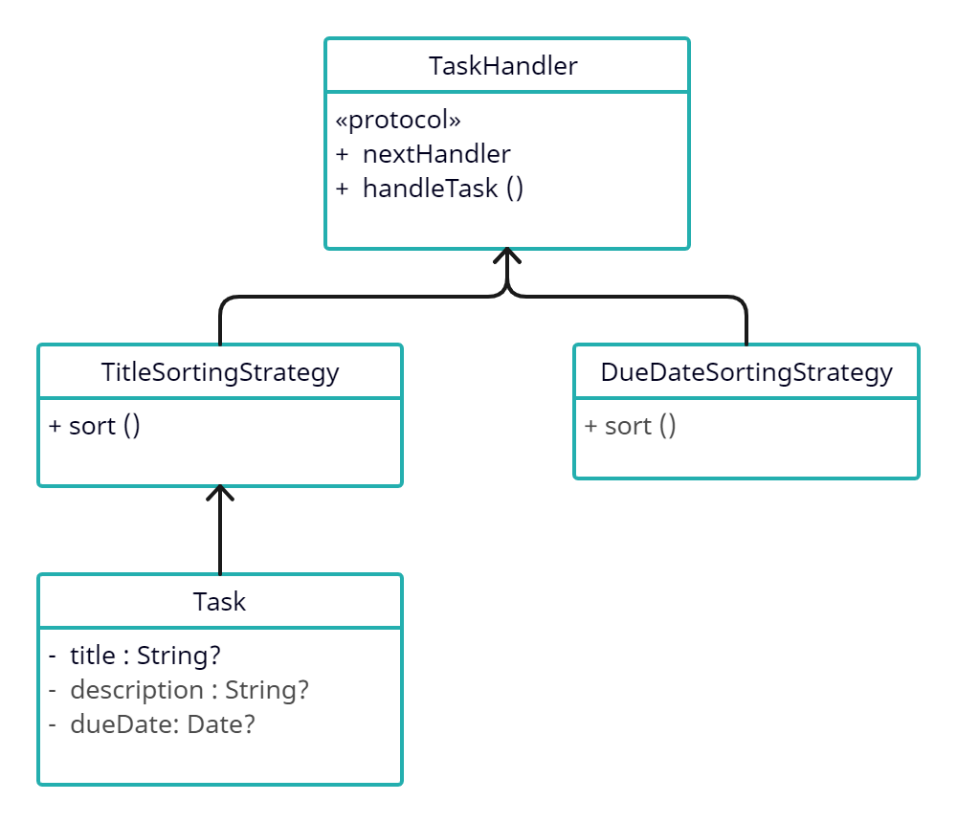


Figura 3.8 Strategy Pattern

Strategy pattern definește o familie de algoritmi interschimbabili și încapsulează fiecare algoritm în mod separat. Protocolul TaskSortingStrategy reprezintă o strategie de sortare pentru sarcini. Clasele TitleSortingStrategy și DueDateSortingStrategy implementează acest protocol și oferă algoritmi de sortare diferiți pentru sarcini. Acest lucru permite sortarea sarcinilor în funcție de criterii diferite prin selectarea dinamică a strategiei de sortare dorite.

# CONCLUZIE

Design pattern-urile sunt instrumente puternice în dezvoltarea software, care oferă soluții eficiente și reutilizabile pentru probleme comune de proiectare. În cazul acestei aplicații de gestionare a sarcinilor, utilizarea design pattern-urilor a condus la o arhitectură solidă, modulară și ușor extensibilă.

Singleton pattern a asigurat că există o singură instanță a clasei TaskManager în întreaga aplicație, permițând accesul simplu și consistent la sarcini. Acesta a asigurat un nivel ridicat de coerență în gestionarea sarcinilor și a prevenit crearea mai multor instanțe care ar fi putut duce la inconsistențe sau conflicte.

Builder pattern a oferit un mod flexibil și ușor de utilizat pentru construirea obiectelor de tip Task, permițând inițializarea pas cu pas a proprietăților. Acesta a facilitat crearea sarcinilor personalizate cu proprietăți specifice și a oferit o metodă clară și expresivă de construcție a obiectelor.

Factory pattern a furnizat un mecanism de creare a sarcinilor, izolând logica de creare într-o clasă separată și abstractizând procesul de creare. Aceasta a facilitat extinderea și înlocuirea facilă a metodei de creare a sarcinilor fără a afecta restul aplicației.

Adapter pattern a permis adaptarea interfeței unei clase existente (Task) la o altă interfață prin intermediul clasei TaskAdapter. Aceasta a facilitat interacțiunea cu alte componente ale aplicației care aveau nevoie de o interfață diferită și a permis utilizarea clasei Task într-un mod flexibil și compatibil.

Composite pattern a permis tratarea grupului de sarcini ca un singur obiect coeziv și a furnizat metode consistente pentru gestionarea sarcinilor individuale și a grupului. Acesta a permis operații uniforme asupra sarcinilor individuale și a grupurilor de sarcini, oferind un model intuitiv și ușor de utilizat.

Decorator pattern a adăugat funcționalități suplimentare la obiectele de tip Task prin intermediul decoratoarelor. Acesta a oferit o modalitate flexibilă de extindere a comportamentului obiectelor Task fără a modifica direct clasa de bază, permițând adăugarea de funcționalități opționale sau personalizate.

Chain of Responsibility pattern a oferit o abordare modulară pentru gestionarea cererilor de la sarcini prin intermediul lanțului de handleri. Acesta a permis aplicarea logică individuală a diferitelor aspecte ale sarcinilor și a oferit flexibilitate în configurarea lanțului și adăugarea sau eliminarea de handleri.

Strategy pattern a permis selectarea dinamică a strategiilor de sortare pentru sarcini, permițând utilizatorului să aleagă criteriul preferat de sortare. Acesta a oferit o modalitate simplă și eficientă de extindere a metodelor de sortare fără a modifica clasa de bază, asigurând flexibilitate în gestionarea sarcinilor.

În concluzie, utilizarea design pattern-urilor în această aplicație a adus numeroase beneficii, cum ar fi reutilizabilitatea codului, modularitatea, extensibilitatea și ușurința în gestionarea diferitelor aspecte ale sarcinilor. Aceste design pattern-uri au demonstrat eficiența și utilitatea lor în îmbunătățirea proiectării și dezvoltării aplicațiilor software.

# BIBLIOGRAFIE

1. "Design Patterns: Elements of Reusable Object-Oriented Software" de Erich Gamma, Richard Helm, Ralph Johnson și John Vlissides.
2. "Head First Design Patterns" de Eric Freeman și Elisabeth Robson.
3. "Design Patterns in Ruby" de Russ Olsen.
4. "Design Patterns: Elements of Reusable Object-Oriented Software" (cartea clasică a design pattern-urilor).
5. "Refactoring: Improving the Design of Existing Code" de Martin Fowler.
6. Site-ul oficial al Gang of Four (GOF): <http://www.oodesign.com/>
7. Site-ul oficial al Design Patterns: <https://www.designpatterns.com/>
8. Tutorialspoint: <https://www.tutorialspoint.com/design_pattern/index.htm>