**MINISTERUL EDUCAŢIEI ȘI CERCETĂRII AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare Informatică şi Microelectronică Departamentul Ingineria Software și Automatică Programul de studii Tehnologia Informației**

**Tema: Realizarea aplicației ToDoList.**

**A realizat: Gîrdei Dumitru,TI-203**

**A verificat: Gaidău Mihai**

**Chişinău, 2023**

**Cuprins**

[Introducere 2](#_bookmark0)

1. [Analiza domeniului de studiu 3](#_bookmark1)-4
2. Modelarea sistemului informational. 5-6
3. Documentația produsului realizat..........................................................................................................7-14
4. Concluzie.................................................................................................................................................15

# Introducere

În dezvoltarea de aplicații software, designul joacă un rol fundamental în asigurarea succesului proiectelor. Un design bine conceput poate face diferența între un sistem software eficient, ușor de întreținut și de extins, și unul care este greu de gestionat și de modificat. Pentru a ajuta la atingerea unui design de calitate, dezvoltatorii folosesc design pattern-uri și principii SOLID. Design pattern-urile reprezintă soluții generice și testate în timp pentru probleme comune întâlnite în dezvoltarea software. Acestea sunt modele de proiectare care oferă o abordare standardizată pentru a rezolva anumite tipuri de probleme de design. De exemplu, pattern-ul "Singleton" oferă o modalitate de a asigura că o clasă are o singură instanță în întregul sistem, iar pattern-ul "Factory" furnizează o metodă simplă de creare a unor obiecte de același tip. Utilizarea design pattern-urilor aduce numeroase beneficii în dezvoltarea de aplicații software. Acestea promovează reutilizabilitatea codului și modularitatea, deoarece soluțiile standardizate pot fi aplicate în mai multe situații. Ele facilitează dezvoltarea rapidă și eficientă, deoarece dezvoltatorii nu trebuie să reinventeze roata de fiecare dată când se confruntă cu o problemă similară. De asemenea, design pattern-urile asigură o structură coerentă și ușor de înțeles, ceea ce facilitează colaborarea între membrii echipei și întreținerea proiectului pe termen lungă durată.

# Analiza domeniului de studiu

În era digitală actuală, tehnologia informației și a comunicațiilor (TIC) reprezintă un domeniu esențial pentru prelucrarea informației, utilizând calculatoarele electronice și alte componente hardware și software. Această tehnologie găsește aplicare într-o varietate de domenii, cum ar fi procesoarele, calculatoarele, limbajele de programare, structurile de date și altele, toate având în comun prelucrarea și manipularea datelor, informațiilor și cunoștințelor.

Ocupațiile din domeniul TIC sunt extrem de diverse și acoperă un spectru larg de activități, de la instalarea software-ului și până la proiectarea unor rețele de calculatoare complexe sau bazelor de date. Acestea includ managementul datelor, construcția hardware-ului pentru calculatoare, proiectarea software-ului și administrarea sistemelor informaționale extinse. Cu evoluția tehnologică din ultimii ani, domeniul TIC s-a extins și cuprinde nu doar calculatoare și rețele, ci și dispozitive mobile, televizoare inteligente, automobile automate, aplicații militare și multe altele. Ca rezultat, cererea de specialiști calificați în acest domeniu este în continuă creștere. În continuare voi vorbi despre aplicatii web si rolul șabloanelor de proiectare pentru a creia aplicații web cât și alte tipuri de aplicații software.

În dezvoltarea de aplicații web, designul este un aspect fundamental care influențează succesul și eficiența proiectelor. Pentru a asigura un cod de calitate, ușor de înțeles, de întreținut și de extins, dezvoltatorii se bazează pe două concepte-cheie: design pattern-uri și principii SOLID. Acestea aduc numeroase beneficii și facilitează procesul de proiectare și implementare a aplicațiilor web, asigurând astfel dezvoltarea unui software robust și de încredere. Un design pattern reprezintă o soluție testată în timp pentru o problemă comună în dezvoltarea software. În contextul aplicațiilor web, există o varietate de design pattern-uri care oferă soluții standardizate și eficiente pentru provocări specifice. De exemplu, pattern-ul "Model-View-Controller" (MVC) oferă o abordare pentru separarea clară a logicii de afaceri, prezentării și gestionării datelor într-o aplicație web. Acesta asigură o structură coerentă și modularitate, facilitând dezvoltarea, întreținerea și extinderea aplicației. Prin utilizarea design pattern-urilor în dezvoltarea aplicațiilor web, se obțin avantaje semnificative. Reutilizabilitatea și modularitatea sunt două aspecte-cheie pe care le aduc design pattern-urile. Soluțiile standardizate pot fi utilizate în mai multe contexte, economisind timp și efort prin evitarea rescrierii repetate a codului. În plus, modularitatea permite dezvoltatorilor să construiască aplicații web ca o colecție de componente independente, care pot fi gestionate și testate separat. Astfel, aplicațiile devin mai flexibile și extensibile, facilitând adăugarea de funcționalități noi sau modificarea celor existente fără a afecta întregul sistem. Pe lângă design pattern-uri, principiile SOLID reprezintă un set de ghiduri fundamentale pentru proiectarea software propuse de Robert C. Martin. Aceste principii se aplică în mod specific și eficient în dezvoltarea de aplicații web.

Principiul Responsabilității Unice afirmă că o clasă ar trebui să aibă o singură responsabilitate și să o îndeplinească în mod corespunzător. În contextul aplicațiilor web, acest principiu se traduce în structurarea modulară a codului, astfel încât fiecare componentă să aibă o responsabilitate clară și bine definită. Aceasta facilitează înțelegerea, întreținerea și extinderea aplicației web, precum și testarea individuală a fiecărei componente. ie deschise pentru extensie, dar închise pentru modificare. Acest principiu promovează flexibilitatea și evită modificările directe ale codului existent atunci când se adaugă sau se modifică funcționalități într-o aplicație web. Prin utilizarea acestui principiu, dezvoltatorii pot adăuga noi caracteristici sau funcționalități prin extinderea codului existent, fără a afecta întregul sistem. Astfel, se minimizează riscul de introducere a erorilor și se facilitează întreținerea și evoluția aplicației web pe termen lung.Principiul Liskov de Înlocuire (Liskov Substitution Principle) subliniază importanța faptului că obiectele unei clase derivate trebuie să poată fi folosite în locul obiectelor clasei de bază, fără a afecta corectitudinea programului. În dezvoltarea aplicațiilor web, acest principiu este esențial pentru asigurarea interoperabilității și compatibilității între diferite componente și module. Respectarea acestui principiu permite dezvoltatorilor să creeze interfețe comune și să schimbe implementările specifice fără a afecta funcționalitatea generală a aplicației web. Principiul Segregării Interfețelor subliniază importanța separării interfețelor în unități logice și corespunzătoare. În contextul aplicațiilor web, acest principiu sugerează că interfețele ar trebui să fie mici, specifice și adaptate la nevoile fiecărei componente. Aceasta evită supraîncărcarea cu funcționalități nedorite și permite dezvoltatorilor să se concentreze pe interfețele relevante pentru fiecare componentă în parte. Prin respectarea acestui principiu, aplicațiile web devin mai flexibile, mai ușor de întreținut și de extins. Principiul Inversiunii Dependințelor pune accentul pe dependențele slabe și inversarea controlului în dezvoltarea aplicațiilor web. În loc să depindem direct de implementări concrete, acest principiu promovează dependențe de abstracții și interfețe. Aceasta facilitează testarea, modularitatea și extensibilitatea aplicației web, permițând schimbarea implementărilor specifice fără a afecta dependențele și funcționalitatea globală. Design pattern-urile și principiile SOLID joacă un rol crucial în dezvoltarea de aplicații web. Design pattern-urile oferă soluții standardizate și eficiente pentru problemele comune, contribuind la crearea de aplicații web reutilizabile, scalabile și ușor de întreținut. Pe de altă parte, principiile SOLID oferă ghiduri fundamentale pentru proiectarea modulară și flexibilă a aplicațiilor web, asigurând un cod, prin intermediule elementelor de interfață, butoane, formulare, meniuri, facilitând astfel gestionarea și manipularea datel

# Modelarea sistemului informațional (Designul)

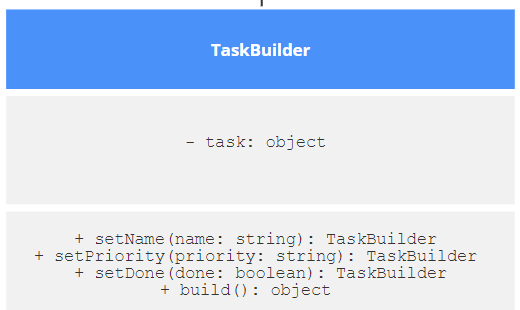


Figura 1. Diagrama UML pentru TaskBuilder

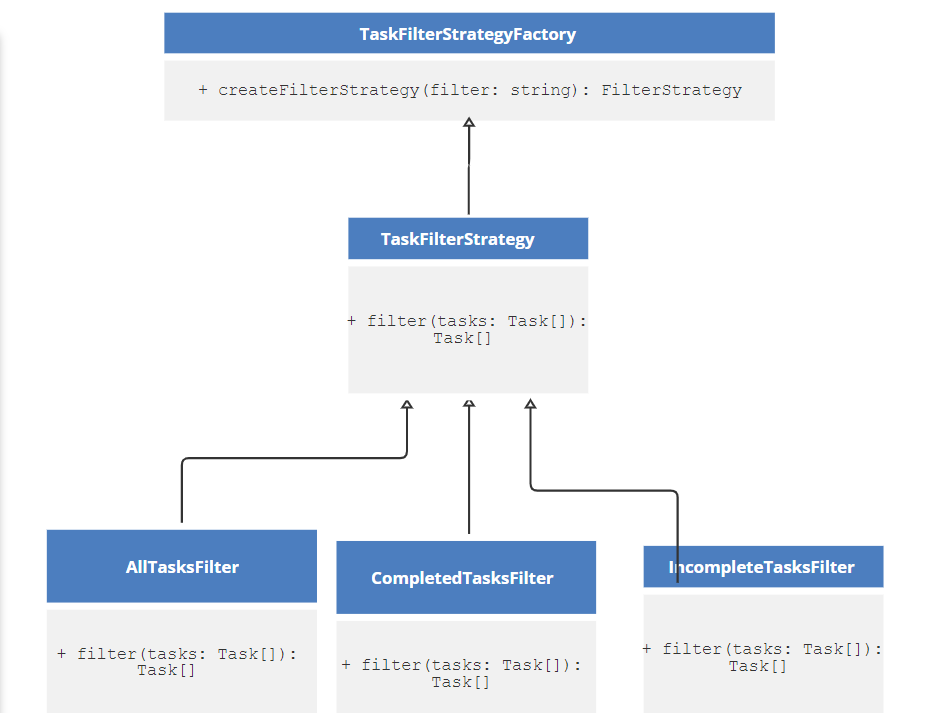


Figura 2. Diagrama UML pentru FactoryMethod si Strategy

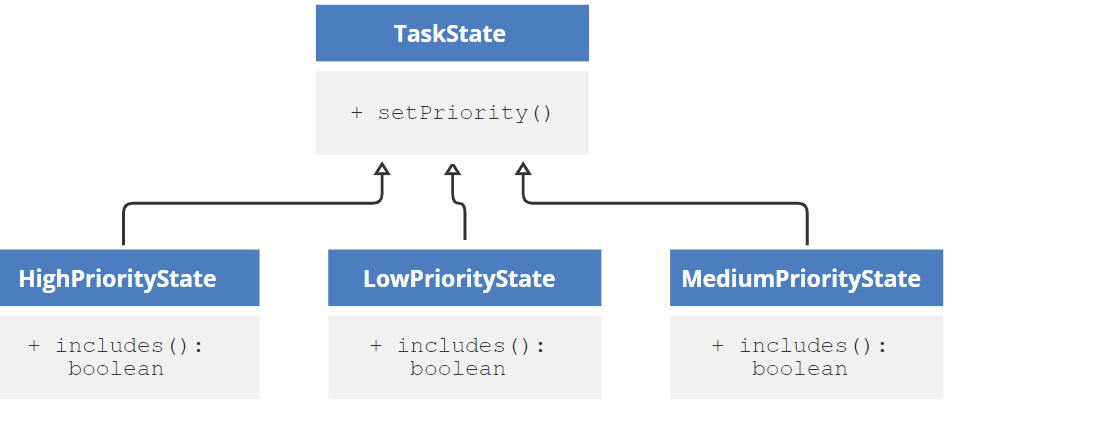


Figura 3. Diagrama UML pentru State

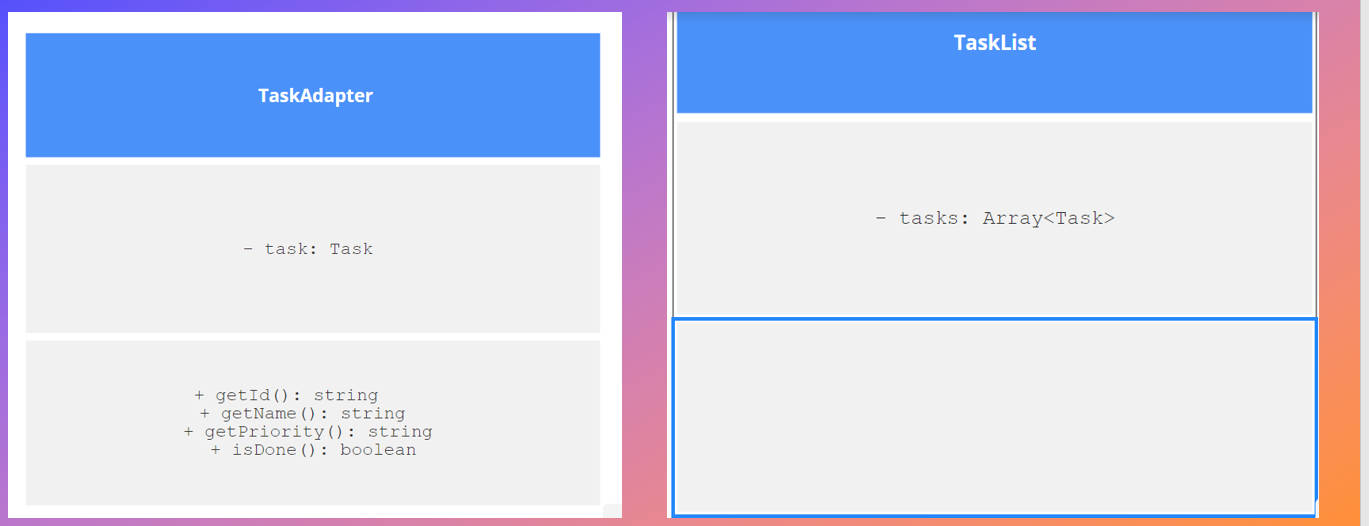


Figura 4. Diagrama claselor pentru adapter si tasklist

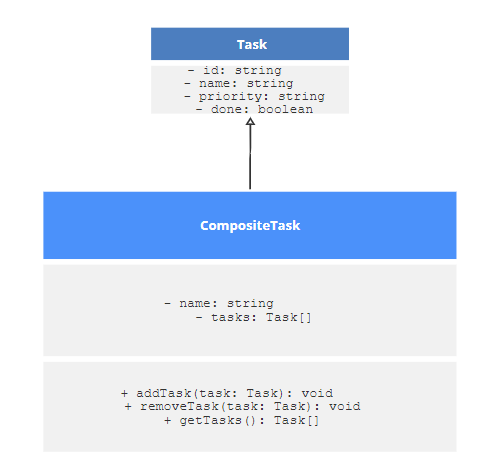


Figura 5. Diagramele claselor pentru CompositeTask si Task

# Documentația produsului realizat

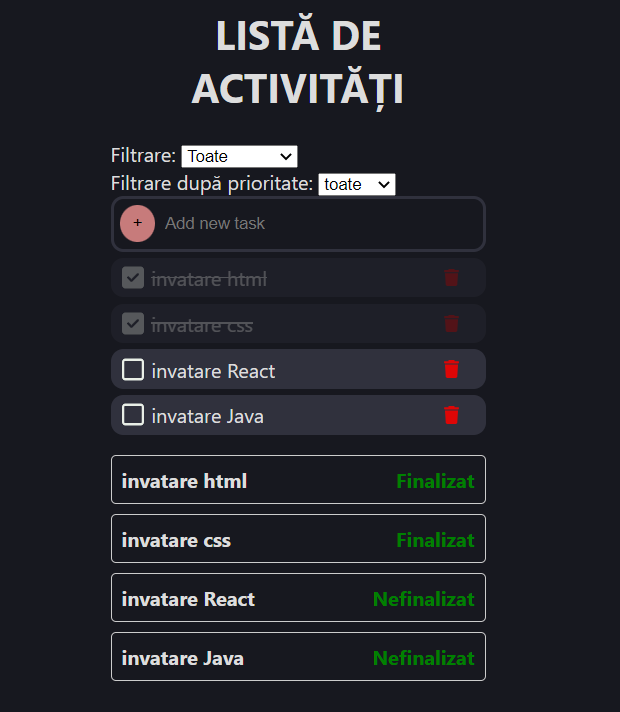


Figura 6. Lista de activități



Figura 7. Lista de activități filtrată după gradul de finalizare

În acest caz, modelul de proiectare a strategiei este folosit pentru a implementa funcționalitatea de filtrare în lista de activități. Trei strategii de filtrare diferite sunt definite ca clase separate: AllTasksFilter, CompletedTasksFilter, și IncompleteTasksFilter. Fiecare strategie implementează o filtermetodă care acceptă o listă de sarcini și aplică logica de filtrare corespunzătoare. Funcția filterTasks din Appcomponentă determină strategia de filtrare adecvată pe baza stării curente a filter variabilei. Acesta creează o instanță a clasei de strategie corespunzătoare și apelează filtermetoda acesteia pentru a obține lista filtrată de sarcini.

**Builder:**



Figura 8. Builder

În acest exemplu, clasa TaskBuilder este responsabilă de construirea obiectelor Task cu ajutorul metodelor setName, setPriority și setDone. Metoda build returnează obiectul Task final construit. În funcția addTask, putem utiliza TaskBuilder pentru a crea un obiect Task și apoi îl adăugăm în starea componentei App.

**Factory Method:**

Design pattern-ul Factory Method se utilizează atunci când doriți să creați obiecte de același tip, dar cu implementări diferite, pe baza unui anumit criteriu sau opțiuni.



Figura 9. Factory Method

În proiectul meu, TaskFilterStrategyFactory exemplifică Factory Method. Această este responsabilă pentru crearea instanțelor strategiilor de filtrare bazate pe criteriul specificat (filter). Prin intermediul acestei metode, putem crea și returna o instanță adecvată a clasei de filtrare corespunzătoare (AllTasksFilter, CompletedTasksFilter sau IncompleteTasksFilter).

Clasa TaskFilterStrategyFactory este implementată ca o Factory. Aceasta primește un criteriu (filter) și returnează o instanță a unei clase care implementează strategia de filtrare specificată. Factory vă permite să creați diferite strategii de filtrare (clase precum AllTasksFilter, CompletedTasksFilter, IncompleteTasksFilter) și să alegeți strategia potrivită în funcție de valoarea criteriului (filter).

**Strategy Design Patern:**

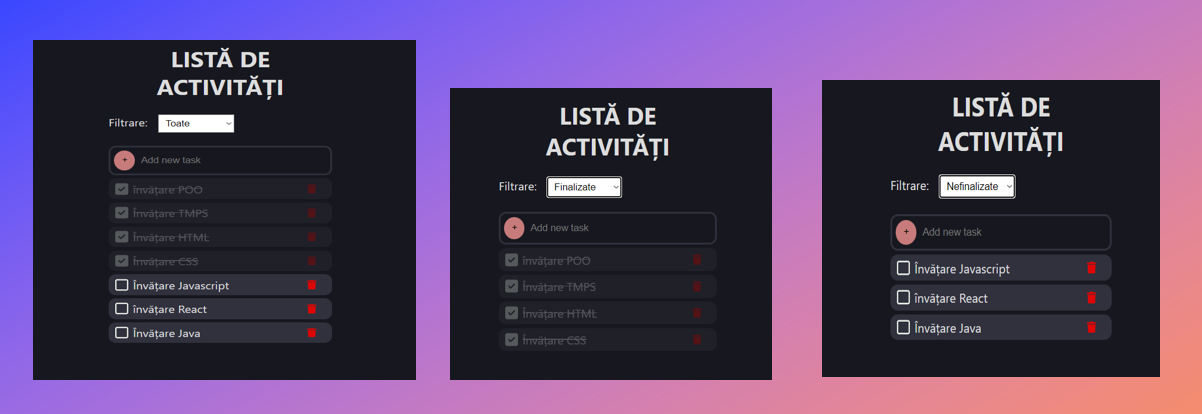


Figura 10. Partea produsului realizată cu ajutorul paternului Strategy

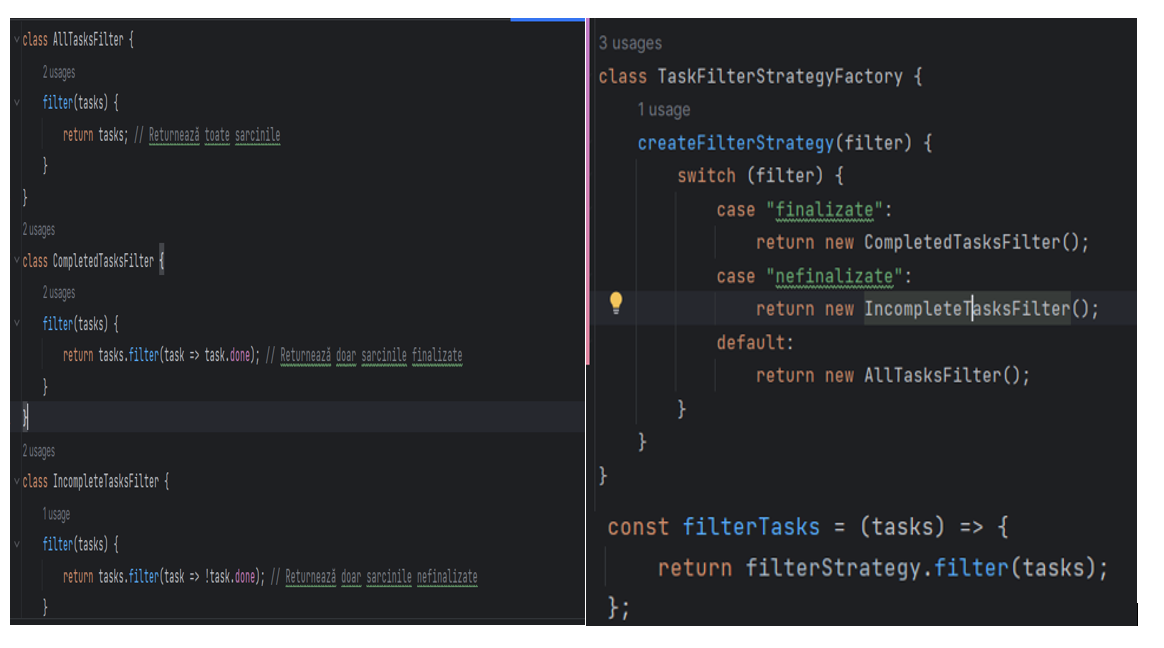


Figura 11. Strategy Design Patern

În acest caz, modelul de proiectare a strategiei este folosit pentru a implementa funcționalitatea de filtrare în lista de activități. Trei strategii de filtrare diferite sunt definite ca clase separate: AllTasksFilter, CompletedTasksFilter, și IncompleteTasksFilter. Fiecare strategie implementează o filtermetodă care acceptă o listă de sarcini și aplică logica de filtrare corespunzătoare. Funcția filterTasks din Appcomponentă determină strategia de filtrare adecvată pe baza stării curente a filter variabilei. Acesta creează o instanță a clasei de strategie corespunzătoare și apelează filtermetoda acesteia pentru a obține lista filtrată de sarcini.

**Composite Design Patern:**

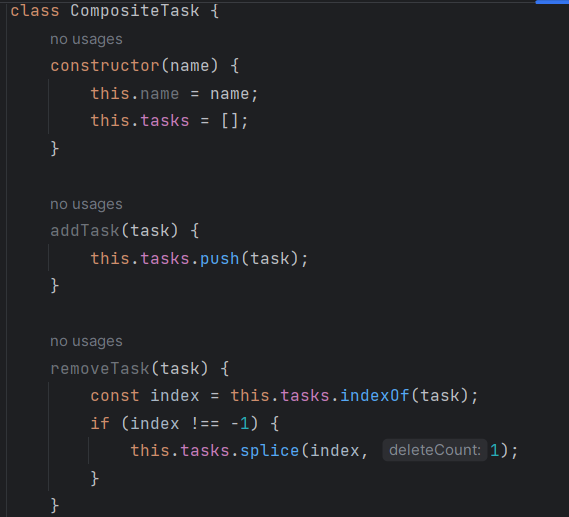
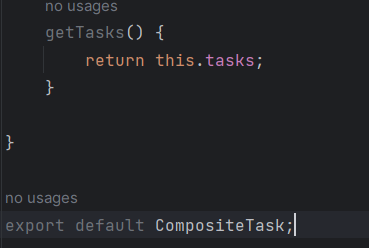
 

Figura 12. Composite Design Patern

Clasa CompositeTask reprezintă un nod în arborele de taskuri. Această clasă are o proprietate name care stochează numele taskului și o proprietate tasks care este un array pentru a stoca subcategoriile sau sarcinile individuale. În cadrul funcției addTask(name, priority, parentTaskId = null) din clasa App, am adăugat logica pentru a verifica dacă există un task părinte și pentru a adăuga noul task fie la taskul părinte, fie direct în lista de taskuri. Am adăugat metoda getTasks() în clasa CompositeTask pentru a returna lista de subcategorii sau sarcini individuale dintr-un nod. Această metodă pur și simplu returnează array-ul tasks.

**Adapter:**

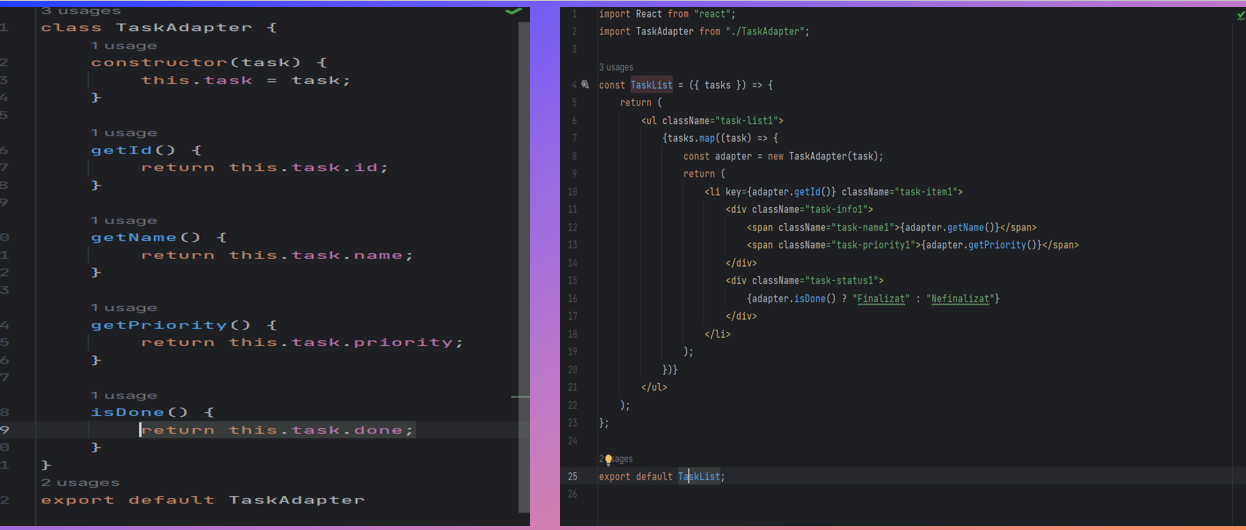


Figura 13. Adapter Design Patern

Adapterul folosit în codul furnizat are rolul de a adapta obiectele de tip task la interfața așteptată de componenta TaskList. Acesta facilitează integrarea obiectelor task în cadrul componentei și permite afișarea informațiilor relevante despre task-uri într-un format specific. Adapterul este utilizat în componenta TaskList pentru a adapta fiecare obiect task într-un format compatibil cu afișarea în listă. Adapterul permite extragerea informațiilor relevante despre task și afișarea acestora în componente React, cum ar fi <span> pentru nume, prioritate și starea finalizată.

Adapterul primește un obiect task ca parametru în constructor și îl stochează într-o proprietate internă (this.task). Aceasta permite accesul la informațiile specifice ale task-ului.Adapterul oferă metode de acces pentru a obține informații specifice despre task-ul adaptat, cum ar fi getId(), getName(), getPriority(), și isDone(). Aceste metode returnează valorile corespunzătoare ale task-ului stocat în proprietatea this.task.

Prin utilizarea acestui adapter, se realizează o separare a responsabilităților între obiectele task și componentele React, permițând adaptarea obiectelor la interfața dorită fără a modifica direct structura sau comportamentul obiectelor de bază.

Astfel, adapterul oferă un nivel de abstractizare și flexibilitate, permițând adaptarea și utilizarea diferitelor tipuri de obiecte în cadrul unei componente fără a afecta codul existent sau a crea dependențe strânse între componente și obiectele adaptate.

**State:**

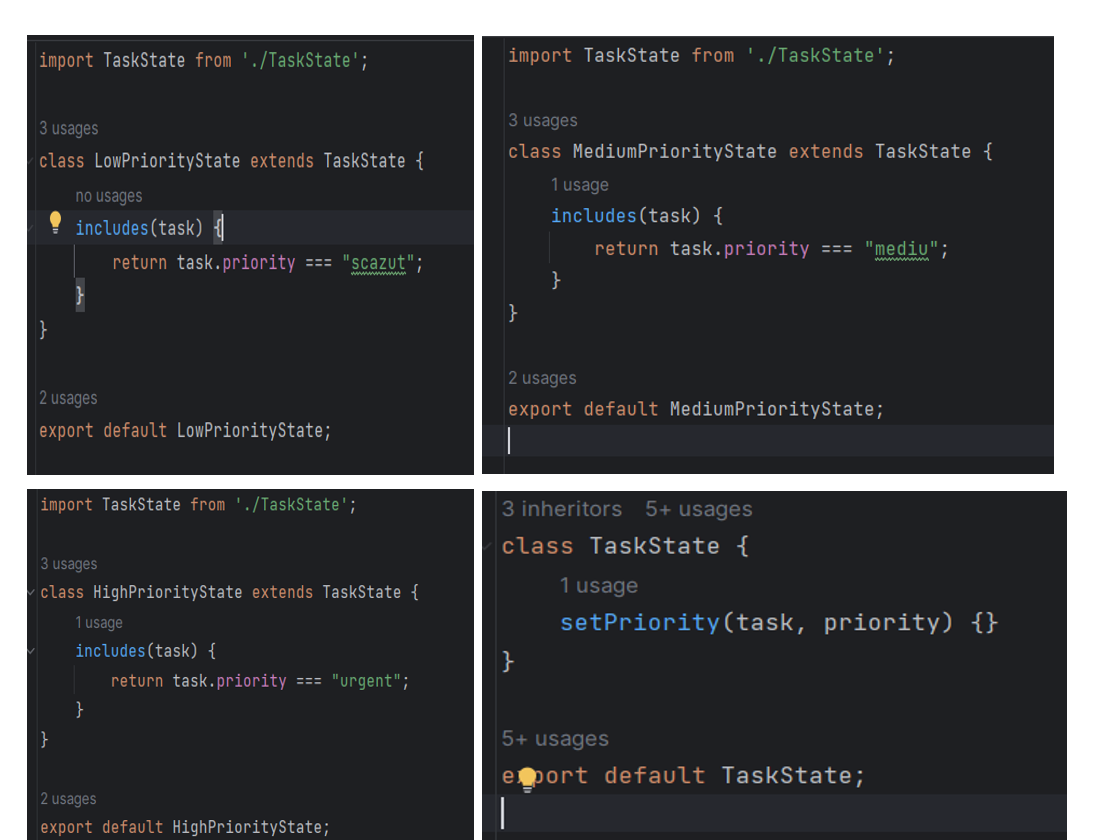


Figura 14. State Design Patern (Partea 1)

Design pattern-ul State este utilizat în proiectul meu pentru a gestiona diferitele stări și comportamente ale task-urilor în funcție de prioritate. Implementarea design pattern-ului State implică definirea unei clase de stare abstracte și a claselor concrete de stare care implementează comportamentul specific pentru fiecare stare. În proiectul meu sunt trei clase care reprezintă stări diferite:

* highPriorityState,
* mediumPriorityState
* lowPriorityState.

Ele reprezintă stările specifice priorității unui task. Aceste clase sunt Definite ca stări concrete și implementează comportamentul specific pentru fiecare stare, precum și metodele necesare pentru tranziția între stări.



Figura 15. State Design Patern (Continuare)

Ideea de bază a design pattern-ului "State" în aplicația mea este de a gestiona și reprezenta diferitele stări ale priorității pentru sarcinile tale. În funcție de filtrul de priorități selectat, aplicația va schimba comportamentul și prezentarea sarcinilor în funcție de starea priorității lor. În funcție de prioritatea unui task, starea corespunzătoare este atribuită și comportamentul specific al acelei stări este aplicat asupra task-ului.În funcția filterPriority, starea task-ului este verificată și, în funcție de valoarea priorityFilter, se aplică comportamentul specific al stării respective pentru a determina clasa de prioritate.

**Concluzii:**

Design Patterns, principiile SOLID și alte principii de proiectare au un rol esențial în dezvoltarea software de calitate. Ele oferă abordări și soluții testate și validate în timp, care facilitează crearea unui cod modular, flexibil și ușor de întreținut. Design Patterns reprezintă soluții la probleme comune și recurente în dezvoltarea software. Ele oferă o modalitate de a aborda anumite situații și de a organiza și structura codul într-un mod eficient. Fiecare pattern are un scop specific și un set de principii și reguli asociate. Utilizarea adecvată a acestor pattern-uri poate duce la o arhitectură mai solidă, o flexibilitate mai mare și o extensibilitate mai bună a aplicației.

Principiile SOLID sunt un set de principii de proiectare orientate pe obiecte, care promovează modularitatea, flexibilitatea și ușurința înțelegerii și modificării codului. Aceste principii includ Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP) și Dependency Inversion Principle (DIP). Respectarea acestor principii conduce la un cod mai modular, ușor de testat și de întreținut, și facilitează încapsularea și reutilizarea componentelor.

În proiectul prezentat, am folosit diverse design patterns și principii SOLID pentru a crea o arhitectură robustă și flexibilă. Am utilizat Composite Pattern pentru a organiza și structura task-urile și categoriile într-o ierarhie, Decorator Pattern pentru a adăuga funcționalități suplimentare la obiectele Task, Adapter Pattern pentru a adapta obiectele Task la un format convenabil pentru afișare, și Factory Method Pattern pentru a crea obiecte de tip Task în funcție de un anumit tip. Am urmat și principiile SOLID, cum ar fi SRP și DIP, pentru a asigura coeziunea și decuplarea componentelor, și pentru a promova reutilizarea și extensibilitatea.

La final de concluzie, design patterns și principiile SOLID sunt instrumente valoroase în dezvoltarea software, oferind direcție și structură pentru a construi aplicații de calitate. Prin aplicarea acestor pattern-uri și principii, putem crea cod modular, flexibil și ușor de întreținut, care să îndeplinească cerințele actuale și să faciliteze viitoare modificări și extensii.