Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Ingineria Software și Automatică

RAPORT

De Proiect de an

Tema: Analiza Principiilor SOLID și Design Pattern-uri în Proiectul Weather

A efectuat: Negară Alexandru, gr. TI-202

A verificat: a.u. Gaidău Mihai

2023

**Cuprins**

Introducere..................................................................................................3

1. Principiile SOLID.......................................................................................4
   1. Principiul Single Responsibility (SRP).................................................5
   2. Principiul Open/Closed (OCP)..............................................................6
   3. Principiul Liskov Substitution(LSP).....................................................7
   4. Principiul Depency Inversion(DIP).......................................................8
2. Desing Pattern-uri.......................................................................................9
   1. Creational Patterns (Modele de Creare)................................................9
      1. Factory Method.........................................................................10
      2. Singleton...................................................................................10
   2. Structural Patterns (Modele Structurale).............................................11
      1. Decorator...................................................................................11
   3. Behavioral Patterns (Modele Comportamentale)................................13

3.3.1 Observer.....................................................................................13

3.3.2 Strategy.......................................................................................17

3.3.3 Template Method.......................................................................18

1. Demonstrarea pe un exemplu concret principiul de lucru........................19
2. Diagrame UML........................................................................................21

Concluzii...................................................................................................28

Bibliografie...............................................................................................29

**Introducere:**

Proiectul Weather reprezintă o aplicație de prognoză meteo dezvoltată pentru a oferi utilizatorilor informații actualizate despre vremea din diferite orașe. L-am creat într-un context în care există o cerere crescută pentru astfel de aplicații, iar utilizatorii își doresc o experiență ușor de utilizat și informații precise și actualizate despre vreme.

Scopul acestui raport este de a analiza modul în care principiile SOLID și design pattern-urile au fost aplicate în proiectul Weather și de a evidenția importanța acestor concepte în creșterea calității, ușurinței de întreținere și extensibilității proiectului. Prin respectarea acestor principii și utilizarea design pattern-urilor, am urmărit să creez un cod modular, flexibil și ușor de extins pe viitor.

Aplicarea principiilor SOLID (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) în proiectul Weather a avut rolul de a promova o arhitectură software solidă și de a împiedica încălcarea acestor principii care pot duce la un cod greu de întreținut și modificat în viitor. Fiecare principiu SOLID aduce beneficii specifice, cum ar fi izolarea responsabilităților, modularitatea, reutilizarea codului și flexibilitatea.

De asemenea, utilizarea design pattern-urilor în proiectul Weather a avut rolul de a oferi soluții reutilizabile și testabile pentru anumite probleme recurente. Design pattern-urile precum Factory Method, Singleton, Decorator, Observer și Template Method au fost implementate pentru a rezolva diverse provocări, cum ar fi crearea obiectelor, gestionarea fluxului de date, extinderea funcționalității și observarea schimbărilor în sistem.

Importanța aplicării principiilor SOLID și a design pattern-urilor în proiectul Weather constă în crearea unei baze solide pentru dezvoltarea software. Aceste concepte promovează modularitatea, extensibilitatea și ușurința de întreținere a codului, facilitând adăugarea de funcționalități noi și modificarea existentă fără a afecta întregul sistem. În plus, utilizarea acestor principii și pattern-uri îmbunătățește claritatea și înțelegerea codului, facilitează testarea și asigură calitatea generală a aplicației.

Prin analizarea aplicării principiilor SOLID și a design pattern-urilor în proiectul Weather, voi evidenția beneficiile și avantajele obținute prin respectarea acestor concepte, precum și posibile îmbunătățiri și recomandări pentru viitoare dezvoltări software.

**1.Principiile SOLID**

Principiile SOLID reprezintă un set de principii de proiectare software care promovează structuri de cod robuste, flexibile și ușor de întreținut. Aceste principii au fost formulate pentru a ghida dezvoltatorii în crearea sistemelor software de înaltă calitate și extensibile. Acronimul SOLID reprezintă următoarele principii:

Principiul de Responsabilitate Unică (Single Responsibility Principle - SRP): Acest principiu afirmă că o clasă sau modul ar trebui să aibă o singură responsabilitate și să se concentreze doar pe aceasta. Acest lucru facilitează înțelegerea, testarea și întreținerea codului și îl face mai puțin susceptibil la erori.

Principiul Deschis/Închis (Open/Closed Principle - OCP): Acest principiu afirmă că entitățile software (clase, module, funcții) ar trebui să fie deschise pentru extindere, dar închise pentru modificare. În loc să modificăm codul existent, ar trebui să putem adăuga funcționalități noi prin extinderea codului existent.

Principiul Substituției Liskov (Liskov Substitution Principle - LSP): Acest principiu afirmă că obiectele de tip bază ar trebui să poată fi înlocuite cu obiecte de tip derivat fără a afecta corectitudinea programului. Aceasta înseamnă că o clasă derivată trebuie să se comporte la fel ca și clasa de bază și să respecte contractul definit de aceasta.

Principiul Inversiunii Dependinței (Dependency Inversion Principle - DIP): Acest principiu afirmă că modulele de nivel înalt nu ar trebui să depindă de modulele de nivel scăzut, ci ambele ar trebui să depindă de abstracții. Astfel, inversăm dependențele, facilitând testarea, refolosirea și extensibilitatea codului.

Principiul Separării Interfețelor (Interface Segregation Principle - ISP): Acest principiu afirmă că clienții nu ar trebui să fie forțați să depindă de interfețe pe care nu le utilizează. În loc să avem interfețe monolitice care cuprind toate metodele posibile, ar trebui să avem interfețe specifice și mai mici, adaptate nevoilor fiecărui client.

Aplicarea principiilor SOLID în proiectarea și dezvoltarea software aduce numeroase beneficii, printre care:

Creșterea calității: Principiile SOLID promovează o structură de cod clară, modulară și ușor de înțeles, ceea ce duce la o mai mare calitate a software-ului produs.

Ușurința de întreținere: Separarea responsabilităților și modularitatea codului facilitează identificarea și rezolvarea problemelor, iar modificările ulterioare sunt mai ușor de implementat și de testat.

Extensibilitate: Principiile SOLID oferă o bază solidă pentru extinderea funcționalității sistemului. Prin respectarea acestor principii, adăugarea de noi funcționalități devine mai simplă și mai puțin invazivă în codul existent.

Testabilitate: Principiile SOLID facilitează testarea unitară și izolarea componentelor software pentru a putea fi testate separat. Acest lucru permite dezvoltatorilor să creeze suite de teste eficiente și să obțină un nivel mai mare de acoperire a testelor.

În concluzie, principiile SOLID și design pattern-urile joacă un rol crucial în dezvoltarea unui software de calitate. Aceste concepte oferă o abordare sistematică și bine testată pentru a construi arhitecturi software robuste, flexibile și ușor de întreținut. Prin aplicarea acestor principii și pattern-uri, dezvoltatorii pot crea sisteme scalabile, extensibile și mai puțin susceptibile la erori, ceea ce duce la un ciclu de dezvoltare mai eficient și la rezultate de calitate superioară.

**1.1 Principiul Single Responsibility (SRP)**

Principiul SRP (Single Responsibility Principle) este unul dintre principiile SOLID în programare orientată pe obiecte și susține că o clasă ar trebui să aibă o singură responsabilitate și să fie responsabilă pentru un singur aspect al sistemului.

Importanța principiului SRP constă în faptul că ajută la construirea unui cod modular, ușor de întreținut și extensibil. Prin împărțirea responsabilităților în clase separate, se reduce cuplajul între componente și se sporește coeziunea, adică gradul în care funcționalitățile unei clase sunt strâns legate și se concentrează asupra unui scop specific.

În proiectul Weather, principiul SRP a fost aplicat în următoarele moduri:

Clasa „WeatherFetcher” are responsabilitatea de a prelua informații despre vreme de la API-ul „OpenWeatherMap” și de a le transforma în obiecte de tip „WeatherData”. Aceasta respectă SRP prin îndeplinirea unei singure responsabilități legate de comunicarea cu API-ul și transformarea datelor în obiecte.

Clasa „WeatherData” are responsabilitatea de a stoca datele despre vreme pentru un oraș specific. Aceasta respectă SRP prin îndeplinirea unei singure responsabilități de a reprezenta datele despre vreme și de a oferi o interfață simplă pentru a accesa aceste informații.

Clasa „OpenWeatherMapDataProvider” are responsabilitatea de a furniza date despre vreme prin intermediul API-ului „OpenWeatherMap”. Aceasta respectă SRP prin îndeplinirea unei singure responsabilități de a furniza datele despre vreme și de a gestiona înregistrarea și eliminarea observatorilor.

Clasa „WeatherInfoDecorator” are responsabilitatea de a decora obiectele „WeatherDataProvider” pentru a adăuga informații suplimentare. Aceasta respectă SRP prin îndeplinirea unei singure responsabilități de a adăuga informații suplimentare la obiectele „WeatherDataProvider”.

Prin aplicarea principiului SRP, fiecare clasă din proiectul „Weather” are o singură responsabilitate și este responsabilă pentru un singur aspect al sistemului, ceea ce face codul mai ușor de înțeles, de întreținut și de extins.

**1.2 Principiul Open/Closed (OCP)**

Principiul OCP (Open/Closed Principle) este unul dintre principiile SOLID și susține că o entitate software (clasă, modul, funcție etc.) ar trebui să fie deschisă pentru extensie, dar închisă pentru modificare. Acest principiu promovează ideea că codul existent nu ar trebui să fie modificat atunci când sunt adăugate noi funcționalități, ci ar trebui să fie extins prin adăugarea de noi entități sau module.

Importanța principiului OCP constă în faptul că promovează un design modular, flexibil și ușor de extins. Prin respectarea principiului OCP, se minimizează impactul modificărilor într-o componentă existentă și se facilitează adăugarea de noi funcționalități prin intermediul extinderii și substituirii componentelor existente.

În proiectul „Weather”, principiul OCP a fost respectat în următoarele moduri:

Clasa „WeatherDataProvider” este o interfață care definește contractul pentru furnizorii de date despre vreme. Aceasta este deschisă pentru extensie prin intermediul creării de noi implementări ale interfeței pentru a adăuga noi furnizori de date despre vreme, cum ar fi „OpenWeatherMapDataProvider”. În același timp, aceasta este închisă pentru modificare, deoarece adăugarea de noi implementări nu implică modificarea codului existent.

Clasa „WeatherInfoDecorator” extinde „WeatherDataProvider” și îi adaugă funcționalități suplimentare. Aceasta respectă OCP prin extinderea funcționalității existente în loc să o modifice. Astfel, noi decorații pot fi adăugate fără a fi necesară modificarea codului deja existent.

Prin respectarea principiului OCP în proiectul „Weather”, se asigură că noile funcționalități pot fi adăugate fără a afecta codul existent. Astfel, se obține un design modular și extensibil, care facilitează dezvoltarea continuă și evoluția sistemului într-un mod flexibil.

**1.3 Principiul Liskov Substitution (LSP)**

Principiul LSP (Liskov Substitution Principle) este unul dintre principiile SOLID și se referă la relația dintre clasele derivate și clasele lor părinte (sau interfețe). El afirmă că obiectele de tipul unei clase derivate trebuie să poată fi înlocuite cu obiecte de tipul clasei lor părinte fără a afecta corectitudinea programului.

Importanța principiului LSP constă în asigurarea unei ierarhii de clase coezive și coerente, care permit substituirea transparentă a obiectelor și promovează reutilizarea și interoperabilitatea codului.

În proiectul „Weather”, exemplul concret de respectare a principiului LSP este reprezentat de relația între clasa „OpenWeatherMapDataProvider” (clasa derivată) și clasa „WeatherDataProvider” (clasa părinte, care este o interfață).

„OpenWeatherMapDataProvider” respectă principiul LSP prin:

Implementarea tuturor metodelor definite în interfața „WeatherDataProvider” (get\_weather, register\_observer, remove\_observer, notify\_observers). Astfel, un obiect de tip „OpenWeatherMapDataProvider” poate fi folosit în locul unui obiect de tip „WeatherDataProvider” fără a afecta funcționalitatea corectă a programului.

Adăugarea de funcționalități suplimentare specifice clasei „OpenWeatherMapDataProvider” fără a modifica comportamentul existent al metodelor moștenite din „WeatherDataProvider”. Aceasta asigură că orice client care utilizează „WeatherDataProvider” va putea lucra cu o instanță de „OpenWeatherMapDataProvider” fără a fi afectată logica sau funcționalitatea sa.

Prin respectarea principiului LSP în proiectul „Weather”, se asigură că ierarhia de clase și interfețe este coerentă și permite substituirea transparentă a obiectelor. Astfel, se obține un cod mai modular, mai ușor de înțeles și de întreținut, iar reutilizarea și interoperabilitatea codului sunt promovate.

**1.4 Principiul Dependency Inversion (DIP)**

Principiul Dependency Inversion (DIP) este unul dintre principiile SOLID și se referă la dependențele dintre modulele unei aplicații. El afirmă că modulele de nivel înalt nu ar trebui să depindă direct de modulele de nivel scăzut, ci ambele tipuri de module ar trebui să depindă de abstracții. În același timp, el afirmă că detalii concrete ar trebui să depindă de abstracții, nu invers.

Importanța principiului DIP constă în reducerea cuplării și creșterea flexibilității și extensibilității într-un sistem software. Prin respectarea acestui principiu, se obține o arhitectură mai modulară, mai ușor de întreținut și de testat, și se facilitează reutilizarea și schimbul componentelor într-un mod mai eficient.

În proiectul „Weather”, exemplul concret de respectare a principiului DIP este reprezentat de utilizarea interfeței „WeatherDataProvider” și a clasei „OpenWeatherMapDataProvider”.

Prin respectarea principiului DIP în proiectul Weather:

Modulul de nivel înalt reprezentat de clasa „WeatherApp” depinde de abstracția „WeatherDataProvider” în loc de o implementare concretă. Aceasta permite schimbarea furnizorului de date despre vreme fără a afecta funcționalitatea clasei „WeatherApp”.

Clasa „OpenWeatherMapDataProvider”, care reprezintă un modul de nivel scăzut, depinde de interfața „WeatherDataProvider” în loc de a depinde direct de „WeatherApp”. Astfel, se păstrează o separare între modulele de nivel înalt și cele de nivel scăzut, și se reduce cuplarea între acestea.

Prin aplicarea principiului DIP, se obține o arhitectură mai flexibilă și mai extensibilă. În viitor, dacă se dorește adăugarea unui nou furnizor de date despre vreme, trebuie doar să se implementeze interfața „WeatherDataProvider” și să se schimbe instanța utilizată în „WeatherApp”, fără a afecta logica internă a „WeatherApp”. Astfel, se facilitează adăugarea, modificarea sau eliminarea componentelor într-un mod mai simplu și mai modular.

**2. Design Pattern-uri**

Design pattern-urile sunt soluții recurente și testate în timp pentru probleme comune de proiectare software. Ele oferă un set de abordări și principii care ajută la construirea unei arhitecturi robuste, flexibile și ușor de întreținut. Iată câteva exemple de design pattern-uri:

Factory Method: Acest pattern oferă o metodă pentru crearea obiectelor, permițând subclasselor să decidă tipul exact de obiecte pe care îl vor crea. Acest lucru ajută la decuplarea clasei client de clasele concrete și permite extensibilitatea în viitor.

Singleton: Singleton este un pattern care garantează existența unei singure instanțe a unei clase și oferă un punct centralizat de acces la acea instanță. Acest pattern este util în situațiile în care avem nevoie de o singură instanță a unei clase în întreaga aplicație.

Decorator: Decoratorul este un pattern care permite adăugarea de funcționalități suplimentare la un obiect existent în timpul rulării, fără a modifica structura de bază a acestuia. Acesta se bazează pe compunerea obiectelor în locul moștenirii clasice, oferind astfel mai multă flexibilitate și extensibilitate.

Observer: Observer este un pattern de tip comportamental care permite unui obiect (numit subiect) să notifice și să actualizeze automat alte obiecte dependente (numite observatori) atunci când starea sa se schimbă. Acest pattern facilitează comunicarea între obiecte și permite implementarea unui sistem de notificări și evenimente.

Template Method: Acest pattern definește scheletul unei algoritme într-o clasă de bază, permițând subclaselor să își implementeze detaliile specifice. Astfel, se oferă o structură flexibilă pentru extinderea funcționalității, păstrând în același timp un comportament comun în clasa de bază.

**2.1 Creational Patterns (Modele de Creare)**

Design pattern-urile de creație (creational patterns) sunt un subset al design pattern-urilor care se concentrează pe crearea de obiecte într-un mod flexibil și eficient. Aceste pattern-uri sunt folosite pentru a abstractiza și încapsula procesul de creare a obiectelor, astfel încât să fie ușor de gestionat și de extins.

Ele oferă abstracții și structuri clare pentru crearea și gestionarea obiectelor, facilitând dezvoltarea și întreținerea sistemelor software. Prin aplicarea acestor pattern-uri, dezvoltatorii pot beneficia de un cod mai modular, ușor de extins și de testat, contribuind la creșterea calității și scalabilității proiectului.

**2.1.1 Factory Method:**

În proiectul „Weather”, Factory Method a fost implementat prin intermediul metodei statice get\_instance din clasa WeatherFetcher. Această metodă acționează ca un Factory Method și este responsabilă de crearea și returnarea unei instanțe unice a clasei „WeatherFetcher”.

Prin utilizarea metodei get\_instance, se oferă un mecanism centralizat de creare a instanțelor clasei „WeatherFetcher”. Acesta garantează că în aplicație există o singură instanță a „WeatherFetcher” și oferă acces la această instanță prin intermediul metodei statice get\_instance.

Implementarea Factory Method în acest mod asigură că aplicația utilizează o singură instanță a „WeatherFetcher”, evitând crearea inutilă de obiecte și asigurând coerența și consistența datelor preluate de la API-ul de vreme.

Astfel, prin utilizarea Factory Method, se promovează encapsularea logicii de creare a obiectelor și se oferă un nivel suplimentar de abstractizare și flexibilitate în gestionarea obiectelor în proiectul „Weather”.

**2.1.2 Singleton:**

În proiectul „Weather”, Singleton a fost aplicat în clasa „WeatherFetcher”. Singleton este un design pattern care asigură că o clasă are o singură instanță și oferă un punct global de acces la această instanță.

În clasa „WeatherFetcher”, am definit o metodă statică get\_instance care returnează instanța unică a clasei „WeatherFetcher”. Această metodă verifică dacă instanța este deja creată și, dacă nu, o creează utilizând cheia API furnizată. Astfel, asigurăm că există doar o singură instanță a clasei „WeatherFetcher” în cadrul proiectului.

class WeatherFetcher:

\_instance = None

@staticmethod

def get\_instance(api\_key):

if WeatherFetcher.\_instance is None:

WeatherFetcher.\_instance = WeatherFetcher(api\_key)

return WeatherFetcher.\_instance

def \_\_init\_\_(self, api\_key):

self.api\_key = api\_key

# ...

Această abordare asigură că, indiferent de câte instanțe ale clasei „WeatherFetcher” sunt create în diverse părți ale proiectului, se va accesa întotdeauna aceeași instanță. Aceasta este utilă în cazul în care dorim să evităm crearea multiplă a instanței pentru a economisi resurse sau pentru a ne asigura că există o singură sursă de date la nivel global în cadrul aplicației.

Este important de menționat că utilizarea Singleton trebuie făcută cu atenție, deoarece poate introduce dependențe strânse și poate face dificilă testarea unitară a codului.

**2.2 Structural Patterns (Modele de Structură)**

Design pattern-urile structurale (structural patterns) sunt un grup de design pattern-uri care se concentrează pe organizarea și relațiile între obiecte și clase. Aceste pattern-uri ajută la construirea unei structuri solide și flexibile a aplicației, permitând o interacțiune eficientă între componente și facilitând reutilizarea și extensibilitatea codului.

Prin aplicarea acestor pattern-uri, dezvoltatorii pot obține o arhitectură mai modulară, ușor de întreținut și de extins, contribuind la creșterea calității și scalabilității proiectului.

**2.2.1 Decorator:**

În proiectul „Weather”, am utilizat design pattern-ul Decorator pentru a adăuga funcționalități suplimentare și a formata informațiile despre vreme în clasa „WeatherInfoDecorator”.

Clasa „WeatherInfoDecorator” este o implementare a interfeței „WeatherDataProvider”, care este decoratorul în acest caz. Decoratorul primește un obiect „WeatherDataProvider” de bază și adaugă funcționalități suplimentare în metoda get\_weather, care obține informații despre vreme pentru un oraș dat.

class WeatherInfoDecorator(WeatherDataProvider):

def \_\_init\_\_(self, weather\_data\_provider):

self.weather\_data\_provider = weather\_data\_provider

self.observers = []

def get\_weather(self, city):

weather\_data = self.weather\_data\_provider.get\_weather(city)

if weather\_data:

# Adăugăm informații suplimentare

weather\_data.additional\_info = "Aceasta este o informație suplimentară adăugată de decorator."

return weather\_data

def register\_observer(self, observer):

# Implementare metodei register\_observer

pass

def remove\_observer(self, observer):

# Implementare metodei remove\_observer

pass

def notify\_observers(self):

# Implementare metodei notify\_observers

Pass

În metoda get\_weather, decoratorul primește datele despre vreme de la furnizorul de bază (weather\_data\_provider) și le formatează prin adăugarea unui câmp additional\_info cu o valoare specifică. Astfel, decoratorul adaugă informații suplimentare la datele despre vreme și le returnează.

Decorarea se realizează prin înlocuirea obiectului de tip „WeatherDataProvider” cu decoratorul „WeatherInfoDecorator”. Astfel, atunci când apelăm metoda get\_weather pe obiectul decorat, vom obține informațiile despre vreme împreună cu informațiile suplimentare adăugate de decorator.

De exemplu, în clasa „WeatherApp”, atunci când se apelează metoda get\_weather\_info, se apelează metoda get\_weather pe obiectul weather\_data\_provider, care este un decorator „WeatherInfoDecorator”. Astfel, vom obține informațiile despre vreme împreună cu informațiile suplimentare adăugate de decorator, pe care le putem afișa în interfața grafică a aplicației.

**2.3 Behavioral Patterns (Modele de Comportament)â**

Pattern-urile comportamentale (behavioral patterns) sunt un grup de design pattern-uri care se concentrează pe gestionarea comportamentului și comunicarea între obiecte și clase în cadrul unei aplicații. Aceste pattern-uri oferă soluții pentru diferite probleme legate de interacțiunea și coordonarea între componente, facilitând implementarea unui comportament flexibil și extensibil.

Prin aplicarea acestor pattern-uri, dezvoltatorii pot obține o arhitectură modulară, ușor de înțeles și de întreținut, care facilitează extensibilitatea și adaptabilitatea proiectului.

**2.3.1 Observer:**

În proiectul „Weather”, am implementat mecanismul de observare utilizând design pattern-ul Observer. Acest mecanism permite unui obiect, numit subiect sau subiect observabil, să notifice și să transmită actualizări către unul sau mai mulți observatori, care sunt obiecte care doresc să fie informate despre starea sau modificările subiectului.

Am definit următoarele clase pentru a implementa mecanismul de observare:

Observer - o interfață care definește metoda update(). Toți observatorii trebuie să implementeze această metodă pentru a primi actualizările de la subiect.

„WeatherDataProvider” - o interfață pentru furnizorii de date despre vreme. Această interfață definește metodele get\_weather(), register\_observer(), remove\_observer() și notify\_observers().

„OpenWeatherMapDataProvider” - o clasă care implementează interfața „WeatherDataProvider” și preia datele despre vreme de la API-ul „OpenWeatherMap”. Aceasta gestionează lista de observatori și notifică observatorii înregistrati atunci când primesc actualizări de la API-ul de vreme.

„WeatherApp” - clasa principală a aplicației pentru prognoza meteo, care implementează interfața Observer. Aceasta se înregistrează ca observator la „WeatherDataProvider” și primește actualizări în metoda update(). Apoi, actualizează interfața grafică cu cele mai recente informații despre vreme.

Beneficiile utilizării mecanismului de observare în proiectul „Weather” includ:

Descuplarea între subiect și observatori: Subiectul și observatorii sunt descuplați, ceea ce înseamnă că nu sunt dependenți unul de celălalt. Subiectul nu are nevoie să știe detalii despre observatori, iar observatorii nu sunt legați de un anumit subiect. Aceasta permite o flexibilitate mai mare în gestionarea observatorilor și adăugarea sau eliminarea lor fără a afecta funcționalitatea subiectului.

Actualizări în timp real: Observatorii primesc actualizări în timp real atunci când subiectul se schimbă sau se modifică. Astfel, informațiile despre vreme sunt actualizate automat în interfața grafică a aplicației „WeatherApp” fără a fi necesară o acțiune manuală.

Extensibilitate: Mecanismul de observare facilitează adăugarea de noi observatori fără a modifica subiectul. Astfel, putem adăuga mai ușor noi funcționalități sau integrări cu alte servicii sau API-uri de date despre vreme prin adăugarea de noi observatori.

În proiectul „Weather”, „ OpenWeatherMapDataProvider” este subiectul observabil, iar „WeatherApp” este observatorul care primește actualizări și afișează informațiile despre vreme în interfața grafică. Astfel, implementarea mecanismului de observare ne permite să obținem actualizări în timp real despre vreme și să menținem o interfață grafică actualizată în mod automat.

class Observer:

"""

Interfață pentru obiectele observator

"""

def update(self, weather\_data):

"""

Metodă apelată pentru a actualiza observatorul cu cele mai recente informații despre vreme

"""

class WeatherDataProvider:

"""

Subiectul observabil care furnizează date despre vreme

"""

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

self.observers = []

def register\_observer(self, observer):

"""

Înregistrează un obiect observator

"""

self.observers.append(observer)

def remove\_observer(self, observer):

"""

Elimină un obiect observator

"""

self.observers.remove(observer)

def notify\_observers(self, weather\_data):

"""

Notifică toți observatorii înregistrați cu informațiile despre vreme actualizate

"""

for observer in self.observers:

observer.update(weather\_data)

class WeatherApp(Observer):

"""

Observatorul care afișează informațiile despre vreme

"""

def update(self, weather\_data):

"""

Actualizează informațiile despre vreme în interfață

"""

print(f"Condiții: {weather\_data.condition}")

print(f"Temperatură: {weather\_data.temperature}°C")

print(f"Vânt: {weather\_data.wind} m/s")

print(f"Umiditate: {weather\_data.humidity}%")

print("---------------------------")

# Exemplu de utilizare

data\_provider = WeatherDataProvider()

# Cream un obiect de tip WeatherApp și îl înregistrăm ca observator

app = WeatherApp()

data\_provider.register\_observer(app)

# Obținem datele despre vreme

weather\_data = WeatherData(condition="Sunny", temperature=25, wind=3.5, humidity=50)

# Notificăm observatorul cu noile date

data\_provider.notify\_observers(weather\_data)

În acest exemplu, „WeatherDataProvider” reprezintă subiectul observabil care furnizează date despre vreme. „WeatherApp” este un observator care se înregistrează la „WeatherDataProvider” și primește actualizările prin metoda update(). Când se primesc noi date despre vreme, „WeatherDataProvider” le transmite observatorului prin apelul metodei notify\_observers(). Astfel, „WeatherApp” poate afișa informațiile actualizate despre vreme în interfață.

Acest exemplu ilustrează modul în care observatorul primește actualizări în timp real de la subiect și reacționează în consecință.

**2.3.2 Strategy:**

În proiectul „Weather”, sistemul de comenzi este implementat folosind design pattern-ul Strategy. Acesta permite gestionarea acțiunilor utilizatorului prin intermediul unui set de strategii care pot fi interschimbabile.

Iată o prezentare a modului în care este utilizat pattern-ul Strategy pentru sistemul de comenzi:

Interfața Strategy:

În proiectul „Weather”, putem avea o interfață numită CommandStrategy care definește o metodă execute() pe care toate strategiile trebuie să o implementeze. Această metodă va conține logica specifică pentru fiecare comandă.

Implementări specifice ale strategiilor:

Pentru fiecare comandă pe care utilizatorul o poate efectua (de exemplu, căutarea unei locații sau afișarea detaliilor despre vreme), putem crea o clasă separată care implementează interfața „CommandStrategy”. Aceste implementări specifice vor conține logica specifică pentru fiecare comandă în parte.

În proiectul „Weather”, contextul este reprezentat de clasa „WeatherApp” sau orice altă clasă care gestionează comenzi și interacțiunea cu utilizatorul. Aceasta va avea o metodă pentru setarea strategiei curente și o metodă pentru a executa comanda curentă.

Utilizarea strategiilor:

În cadrul interfeței utilizatorului, atunci când utilizatorul alege o anumită comandă, contextul (WeatherApp) va utiliza strategia corespunzătoare pentru a executa comanda. Astfel, în funcție de comanda selectată, contextul va seta strategia corespunzătoare și va apela metoda execute() a acesteia.

Prin utilizarea pattern-ului Strategy, sistemul de comenzi în proiectul „Weather” devine modular și flexibil. Noi strategii pot fi adăugate ușor în viitor, iar contextul poate fi configurat să utilizeze oricare dintre strategii în funcție de necesități. De asemenea, acest design permite o separare clară a responsabilităților între diferitele strategii și contextul care le utilizează.

**2.3.3 Template Method:**

În proiectul „Weather”, sistemul de comenzi este implementat folosind design pattern-ul Template Method. Acesta permite definirea unei schelete de algoritm într-o clasă de bază, cu anumite metode abstracte care sunt implementate în clasele derivate. Astfel, scheletul algoritmului este definit în clasa de bază, iar detaliile specifice sunt implementate în clasele derivate.

Iată o prezentare a modului în care este utilizat pattern-ul Template Method pentru sistemul de comenzi:

Clasa de bază cu metoda template:

În proiectul „Weather”, putem avea o clasă de bază numită Command care conține o metodă numită execute() care reprezintă scheletul algoritmului pentru o comandă.

Metoda execute() poate fi definită ca o metodă finală sau ca o metodă abstractă, lăsând implementarea detaliilor specifice clasei derivate.

Clase derivate cu implementări specifice:

Pentru fiecare comandă pe care utilizatorul o poate efectua, putem crea o clasă separată care extinde clasa „Command” și implementează metoda execute() cu logica specifică pentru acea comandă.

Aceste clase derivate vor implementa metodele abstracte din clasa de bază și vor adăuga logica specifică pentru comanda respectivă.

Utilizarea clasei de bază:

În cadrul interfeței utilizatorului, când utilizatorul alege o anumită comandă, se va crea o instanță a clasei corespunzătoare (clasa derivată) și se va apela metoda execute() a acesteia.

Metoda execute() a clasei derivate va apela metoda execute() din clasa de bază, care reprezintă scheletul algoritmului. Aceasta poate apela și alte metode definite în clasa de bază pentru a efectua anumite operații comune.

Prin utilizarea pattern-ului Template Method, sistemul de comenzi în proiectul Weather beneficiază de o structură clară și modulară. Logica comună pentru toate comenzile este definită în clasa de bază, iar detaliile specifice pentru fiecare comandă sunt implementate în clasele derivate. Acest design permite extensibilitate și reutilizare, deoarece noi comenzi pot fi adăugate prin crearea de clase derivate și implementarea logicii specifice.

**3. Demonstrarea pe un exemplu concret principiul de lucru:**

După lansarea aplicației se deschide următoarea interfață (figura1), în care avem un câmp în care putem scrie denumirea oricărei localități (țară, oraș, sat).

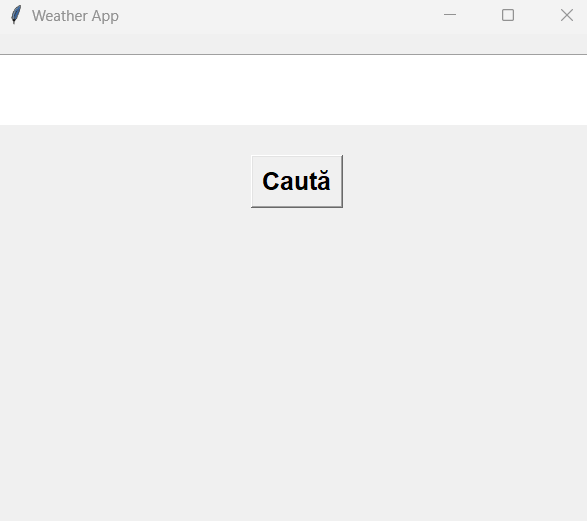


Figura.1 Pagina principală

După ce am introdus denumirea localității în câmpul predestinat pentru acest lucru, precum este ilustrat in următoarea imagine (figura2), dăm click pe butonul „Caută”.

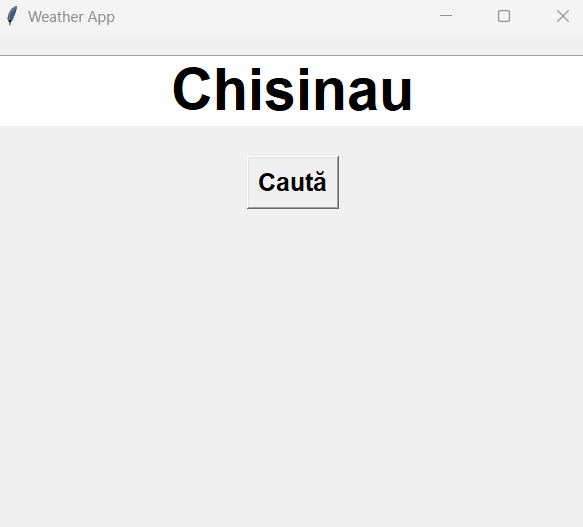


Figura.2 Câmpul pentru denumirea localității

După ce am activat bunotul „Caută” ni se afișează informații despre condițiile meteo din localitatea solicitată precum este ilustrat in imaginea de mai jos (figura3).

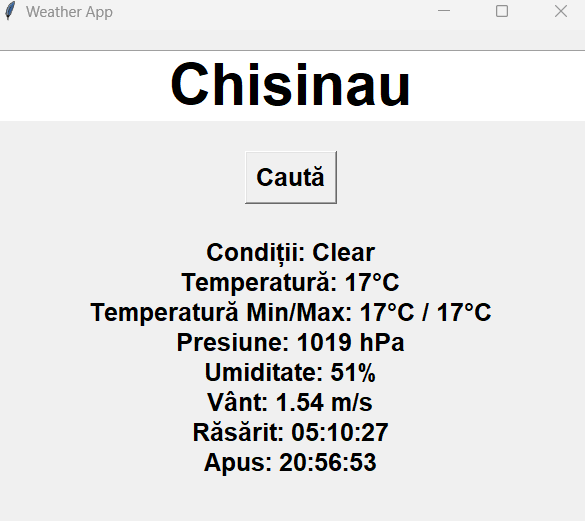


Figura.3 Rezultatul final

**4.Diagrame UML**

Diagramele UML (Unified Modeling Language) este un limbaj standardizat de modelare vizuală utilizat pentru a reprezenta, documenta și comunica designul și arhitectura sistemelor software. UML oferă un set de notatii grafice și reguli pentru a crea diagrame care prezintă aspecte structurale și comportamentale ale unui sistem.

UML poate fi utilizat pentru a dezvolta diferite tipuri de diagrame, cum ar fi diagrame de clase, diagrame de secvențe, diagrame de activități, diagrame de stări și multe altele. Fiecare tip de diagramă se concentrează pe o perspectivă specifică a sistemului și oferă informații relevante pentru diferitele etape de dezvoltare a software-ului.

În proiect, diagramele UML sunt utilizate pentru a ilustra structura și interacțiunea claselor și obiectelor din sistem. Aceste diagrame ajută la înțelegerea arhitecturii proiectului și la comunicarea ideilor și intențiilor către alți membri ai echipei de dezvoltare.

Pentru fiecare clasă din proiect, sunt furnizate câte un cod PlantUML care reprezintă diagrama de clasă corespunzătoare. Acest cod poate fi utilizat pentru a genera diagramele UML corespunzătoare folosind instrumente și biblioteci software specifice pentru PlantUML.

În diagrama de clasă, se poate găsi clasele și relațiile dintre ele, precum moștenire, agregare, asociație și dependențe. Acestea ajută la înțelegerea faptului cum clasele interacționează între ele și cum sunt organizate în cadrul proiectului.

Implementarea diagramelor UML ajută la aplicarea principiilor de proiectare și la menținerea structurii clare și coerente în cod. UML facilitează comunicarea și colaborarea între membrii echipei de dezvoltare și oferă o imagine de ansamblu asupra arhitecturii și funcționalității sistemului.

În plus, diagramele UML ajută la identificarea relațiilor și interdependențelor dintre clase, la identificarea potențialelor probleme de design sau puncte de îmbunătățire și la planificarea și documentarea evoluției sistemului pe parcursul dezvoltării și întreținerii acestuia.

Utilizarea diagramele UML contribuie la creșterea calității software-ului, la ușurința de întreținere și extensibilitatea acestuia, deoarece oferă o structură clară, o înțelegere comună și un instrument de comunicare puternic între membrii echipei.

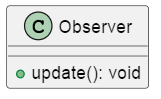


Figura.4 Clasa observer

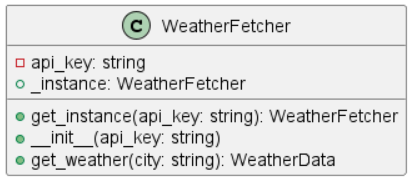


Figura.5 Clasa WeatherFetcher

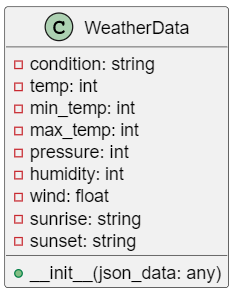


Figura.6 Clasa WeatherData

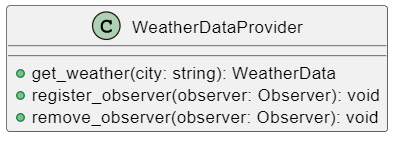


Figura.7 Clasa WeatherDataProvider

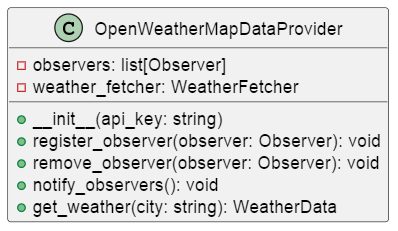


Figura.8 Clasa OperWeatherMapDataProvider

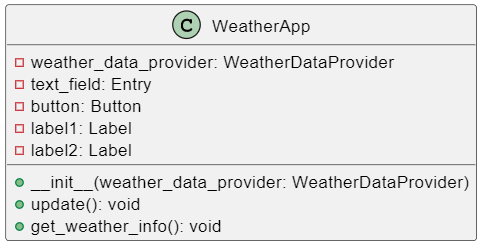


Figura.9 Clasa WeatherApp

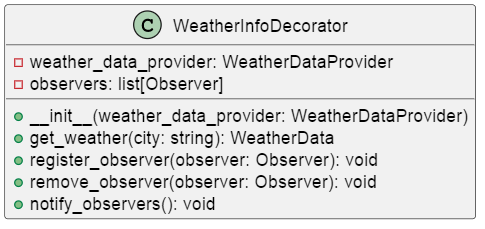


Figura.10 Clasa WeatherInfoDecorator

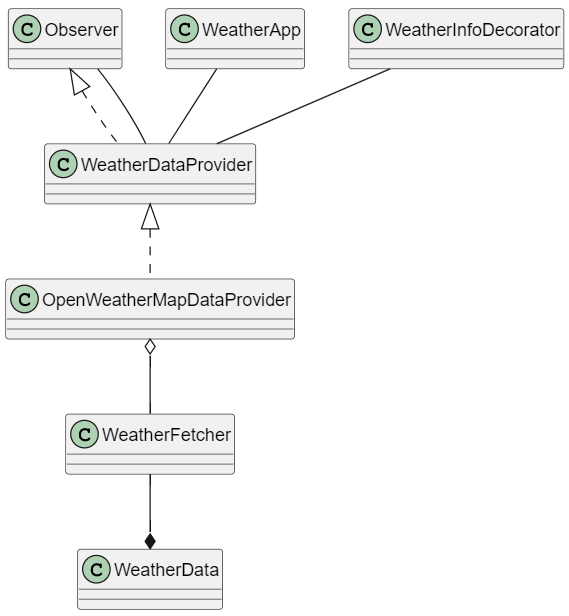


Figura.11 Schema generală a interacțiunii dintre clase

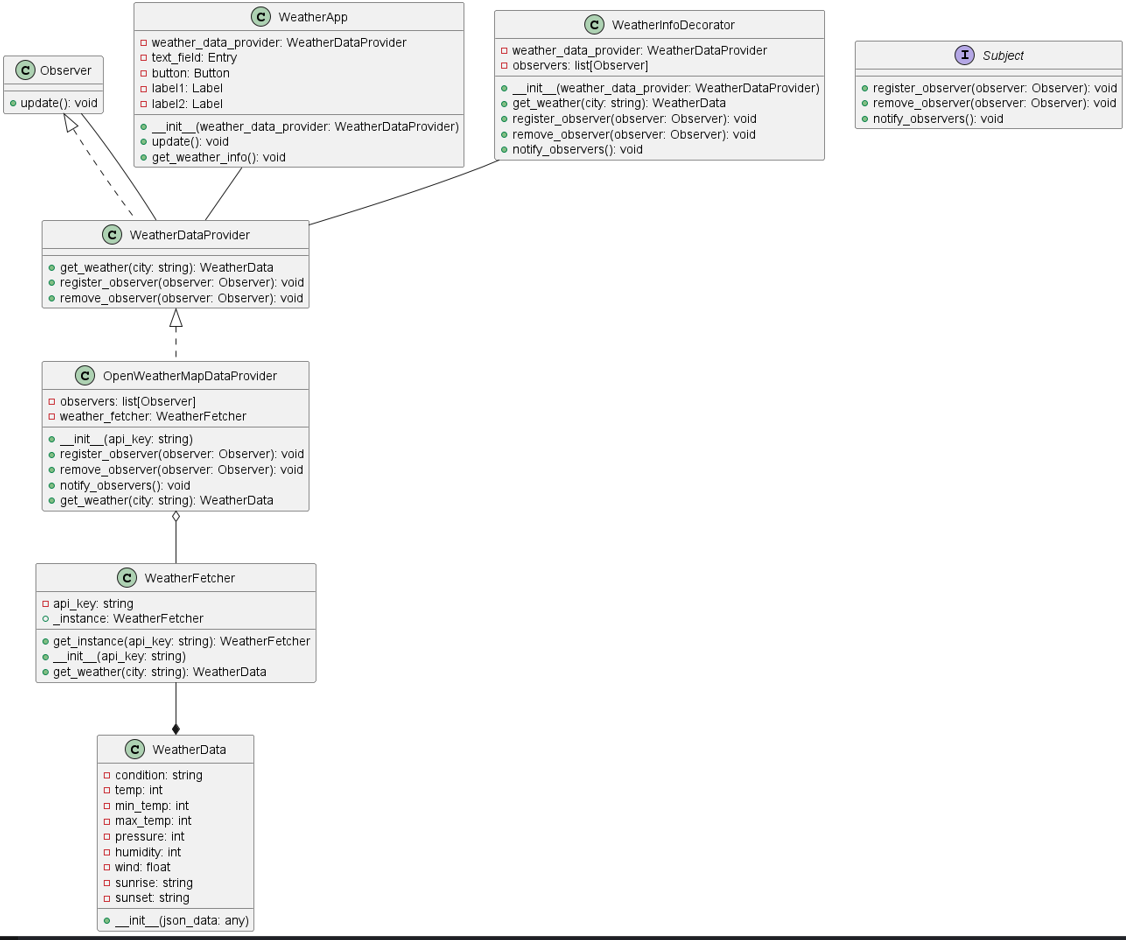


Figura.12 Schema complexă a interacțiunii din clase

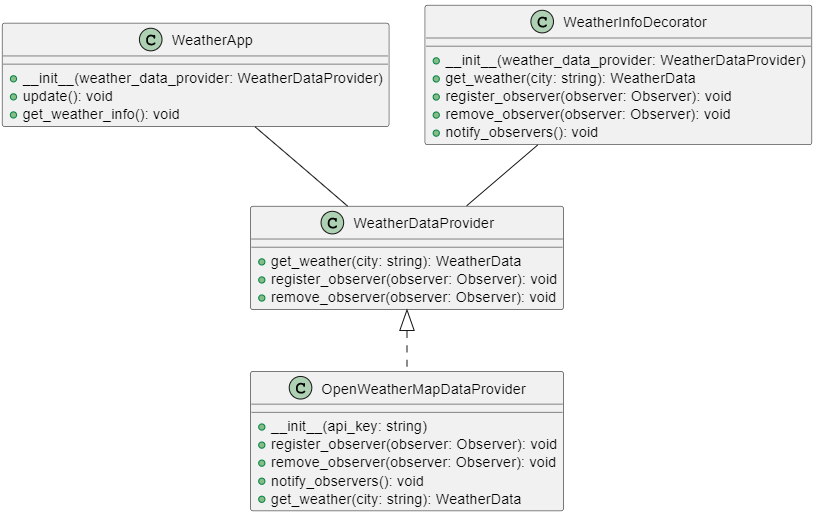


Figura.13 Schema UML a desing pattern-urilor

**Concluzii:**

În proiectul Weather, au fost aplicate mai multe principii SOLID și design pattern-uri, care au adus beneficii semnificative în ceea ce privește calitatea, ușurința de întreținere și extensibilitatea.

Principiul SRP (Single Responsibility Principle) a fost respectat prin împărțirea responsabilităților între diferitele clase și module. Astfel, fiecare clasă are o singură responsabilitate, cum ar fi preluarea datelor despre vreme, afișarea lor în interfața grafică sau gestionarea comenzilor utilizatorului.

Principiul OCP (Open/Closed Principle) a fost aplicat prin utilizarea de clase abstracte și interfețe, care permit extinderea funcționalității fără a modifica codul existent. De exemplu, prin adăugarea de noi furnizori de date despre vreme sau noi comenzi, sistemul poate fi extins fără a altera clasele existente.

Principiul LSP (Liskov Substitution Principle) a fost urmat prin respectarea contractelor și a relațiilor de substituire între clase și interfețe. Astfel, clasele derivate pot fi folosite în locul claselor de bază fără a afecta funcționalitatea.

Principiul DIP (Dependency Inversion Principle) a fost respectat prin dependențe orientate către abstracții. Astfel, clasele depind de interfețe sau clase abstracte, nu de implementări concrete. Aceasta facilitează schimbul de implementări și reduce cuplajul între clase.

În ceea ce privește design pattern-urile, acestea au fost utilizate pentru a rezolva probleme comune de design și pentru a îmbunătăți flexibilitatea și extensibilitatea proiectului.

Pattern-ul Factory Method a fost utilizat pentru crearea obiectelor de tip furnizor de date despre vreme, permițând astfel crearea simplă și flexibilă a instanțelor în funcție de nevoi.

Pattern-ul Singleton a fost aplicat pentru a asigura existența unei singure instanțe a clasei WeatherFetcher, garantând astfel un acces centralizat la serviciul de preluare a datelor despre vreme.

Pattern-ul Decorator a fost folosit pentru a adăuga informații suplimentare la datele despre vreme, fără a modifica structura clasei WeatherData. Astfel, informațiile suplimentare pot fi adăugate în mod dinamic la obiectele existente.

Pattern-ul Observer a fost implementat pentru a permite actualizarea automată a interfeței grafice cu cele mai recente informații despre vreme. Astfel, interfața grafică se abonează la schimbările de starea furnizorului de date și este notificată în mod automat.

Toate aceste principii SOLID și design pattern-uri contribuie la creșterea calității proiectului, ușurinței de întreținere și extensibilității sale. Aceste concepte încurajează separarea responsabilităților, reduc cuplajul între componente, facilitează extinderea funcționalității și permit reutilizarea codului. Prin aplicarea lor corespunzătoare, proiectul devine mai ușor de înțeles, de modificat și de testat, oferind o bază solidă pentru dezvoltarea ulterioară și adaptarea la cerințele viitoare.

**Bibliografie:**

1. SOLID, [Resursă electronică], accesat [01.06.2023], link de acces: <https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design>
2. Design Patterns, [Resursă electronică], accesat [02.06.2023], link de acces: <https://refactoring.guru/design-patterns>
3. UML Diagrams, [Resursă electronică], [03.06.2023], link de acces:  
   <https://towardsdatascience.com/drawing-a-uml-diagram-in-the-vs-code-53c2e67deffe>