Użyte wzorce w projekcie

- **Singleton** zapewnia istnienie tylko jednej instancji menedżerów (notatek, zadań, tagów), gwarantując spójność danych w całym programie.
- **Fabryka (Factory Method)** oddziela proces tworzenia obiektów zadań od logiki menu, pozwalając na centralne zarządzanie ich inicjalizacją.
- **Builder** umożliwia czytelne i krokowe tworzenie złożonych zadań z wieloma opcjonalnymi parametrami bez mnożenia konstruktorów.
- **Dekorator** pozwala na dynamiczne rozszerzanie informacji wyświetlanych o wpisach (np. o tagi lub stan) bez modyfikowania klas bazowych.
- **Stan (State)** automatycznie zmienia zachowanie zadania w zależności od jego bieżącego statusu (np. blokuje edycję po wykonaniu zadania).

1. Wzorzec Dekorator (*Znajduje się w pliku Dekoratory.cs*)

1.1 Przyporządkowanie klas do wszystkich ról wzorca

A) Komponent IWpis

Interfejs definiujący wspólny kontrakt dla obiektów dekorowanych. Deklaruje metodę **string WypiszInformacje();**

Konkretny komponent

Notatka i Zadanie. Klasy bazowe, które implementują IWpis i zawierają podstawowe dane wpisu (tytuł, treść, ID itd.). Mogą być dekorowane dodatkowymi funkcjonalnościami

DekoratorWpisow

- Implementuje IWpis
- Przechowuje referencję do dekorowanego obiektu: ***protected IWpis wpis;***
- Deleguje wywołanie metody **WypiszInformacje()** do obiektu dekorowanego.

Dekoratory konkretne

- **DekoratorTagowy**
  - Rozszerza wynik **WypiszInformacje()** o listę tagów przypisanych do Notatka / Zadanie
- **DekoratorStanowy**
  - Rozszerza wynik **WypiszInformacje()** o aktualny stan zadania (Aktywne, Wykonane, Zaległe)

Wektor zmian dla wzorca Dekorator

1.2 Lokalizacja użycia w kodzie

Wywołanie dekoratorów znajduje się w program.cs linie 360–440. Dekoratory są używane przy wypisywaniu informacji o notatkach i zadaniach w menu głównym.

1.3. Cel użycia Dekoratora

Pozwala dynamicznie rozszerzać funkcjonalność WypiszInformacje() dla obiektów Notatka i Zadanie bez modyfikowania ich klas.

Za pomocą dekoratorów można

- dodawać informacje o tagach (DekoratorTagowy)
- dodawać informacje o stanie zadania (DekoratorStanowy)

1.4. Co można zmieniać lub dodawać dzięki Dekoratorowi

Dekorator daje elastyczny sposób rozszerzania funkcjonalności bez naruszania klas bazowych (Notatka, Zadanie).

Przykłady rozszerzeń:

A) Dodanie nowego dekoratora, np.:

- DekoratorKolorowy - wypisuje informacje w kolorach w konsoli w zależności od priorytetu lub tagów.
- DekoratorSkrocony - wypisuje tylko tytuł i ID wpisu dla szybkiego podglądu.

B) Wprowadzenie dekoratorów warunkowych:

Na przykład dekorator, który wypisuje dodatkowe informacje tylko dla zadań zaległych lub wysokiego priorytetu.

C) Rozszerzenie funkcji wypisywania:

Dodanie nowych pól lub formatów (np. daty utworzenia, autora notatki, komentarze) bez zmiany klas bazowych.

2. Korzyści dla rozbudowy programu

- **Otwarte na rozszerzenia,** zamknięte na modyfikacje. Możliwość dodawania nowej funkcji wypisywania bez zmiany klas Notatka czy Zadanie.
- **Łatwa integracja z menu.** W program.cs wystarczy zmienić wywołanie dekoratora przy wypisywaniu, aby zmienić wygląd informacji dla użytkownika.
- **Dowolna kombinacja dekoratorów.** Możliwość łączenia funkcjonalności (tagi + stan + kolor + skrócony widok) bez tworzenia nowych klas bazowych dla każdej kombinacji.
- **Minimalizacja ryzyka błędów w istniejących klasach.** Dekoratory rozszerzają funkcjonalność bez ingerencji w logikę Notatek i Zadań.

2. Wzorzec Singleton

2.1. Lokalizacja:

- **MenedzerNotatek** → Notatka-Manager.cs linie ok. 20–60
- **MenedzerZadan** →Zadanie-Manager.cs linie ok. 150–190
- **MenedzerTagow** → Tag.cs linie ok. 50–90

2.2. Przyporządkowanie klas do wszystkich ról wzorca

A) Singleton

- MenedzerNotatek – zarządza wszystkimi notatkami programu
- MenedzerZadan – zarządza wszystkimi zadaniami programu
- MenedzerTagow – zarządza wszystkimi tagami programu

B) Prywatny konstruktor:

- Zapobiega tworzeniu instancji spoza klasy

C) Metoda dostępu do instancji:

- GetterInstancji() – zwraca jedyną instancję Singletona

2.3. Użycie Singletona w program.cs linie 15–18:

```
// Singletony
MenedzerNotatek menedzerNotatek = MenedzerNotatek.GetterInstancji();
MenedzerZadan menedzerZadan = MenedzerZadan.GetterInstancji();
MenedzerTagow menedzerTagow = MenedzerTagow.GetterInstancji();
```

Wszystkie późniejsze operacje w menu głównym korzystają z tych jedynych instancji (dodawanie, usuwanie, wypisywanie).

2.4. Cel użycia Singletona

- Zapewnienie jednej, spójnej instancji menedżera dla całego programu.
- Ułatwia centralne zarządzanie notatkami, zadaniami i tagami oraz kontrolę unikalnych ID.
- Pozwala wszystkim częściom programu operować na tej samej liście danych, unikając problemów z synchronizacją lub duplikacją danych.

Wektor zmian dla wzorca Singleton

Obecne użycie w program.cs linie 14 - 17 pobranie instancji i używanie jej w menu.

2.5. Co można zmieniać lub dodawać dzięki Singletonowi:

- Dodanie nowych metod zarządzających wpisami, zadaniami lub tagami, bez modyfikowania reszty programu.
- Możliwość wprowadzenia nowych menedżerów jako Singletony, Na przykład **MenedzerKomentarzy** - zarządzający komentarzami do notatek i zadań.
- Rozbudowa programu o nowe funkcje wyszukiwania, filtrowania, sortowania czy raportowania.

Korzyści dla rozbudowy programu:

- Centralne zarządzanie danymi, spójność stanu aplikacji.
- Łatwe wprowadzanie nowych funkcji do menedżerów.
- Minimalizacja ryzyka błędów wynikających z niejednolitego stanu danych.
- Program staje się łatwy do rozbudowy, np. dodanie nowych typów wpisów, nowych kryteriów wyszukiwania czy filtrów tagów.

3. Wzorzec Fabryka (Factory Method)

Lokalizacja: FabrykaZadan.cs

3.1 Przyporządkowanie klas do wszystkich ról wzorca

Twórca (Factory)

FabrykaWpisow to klasa abstrakcyjna lub bazowa dla fabryk wpisów (notatek i zadań). Deklaruje metodę tworzenia obiektów

```
// Metoda fabrykująca wpis (specyficzna dla Notatki)
// Tutaj metoda jest pusta, zawartość powinna być nadpisana przez pochodną fabrykę.
public virtual Wpis UtworzWpis(string tytul, string tresc, List<string> nazwyTagow)
{
    return null;
}

// Metoda fabrykująca wpis (specyficzna dla Zadań, bo te potrzebują więcej danych)
// Tutaj metoda jest pusta, zawartość powinna być nadpisana przez pochodną fabrykę.
public virtual Wpis UtworzWpis(string tytul, string tresc, Priorytet priorytet, DateTime termin, List<string> nazwyTagow)
{
    return null;
}
```

Twórca (Fabryka zadań)

FabrykaZadan to klasa konkretnie implementująca metodę fabrykującą zadania (Zadanie).

- **Tworzy obiekty** Zadanie z podanymi parametrami i przypisuje do nich tagi.
- **Deleguje obsługę tagów do MenedzerTagow** w przypadku istnienia lub tworzenia nowych tagów.

Produkt (Zadanie)

Wpis to klasa bazowa wszystkich wpisów (abstrakcja dla Notatka i Zadanie). Definiuje wspólny interfejs dla wszystkich produktów fabryki (**Edytuj, WypiszInformacje).**

Konkretny Produkt

- Zadanie to konkretny produkt tworzony przez FabrykaZadan.
- Implementuje dodatkowe właściwości: priorytet, termin, stan zadania oraz listę tagów.

3.2 Lokalizacja w kodzie i użycie

A) Definicja fabryki i produktu:

- **FabrykaWpisow** →plik FabrykaZadan.cs, linie ok. 10–40
- **FabrykaZadan** → plik FabrykaZadan.cs, linie ok. 50–130
- **Zadanie** → plik Zadanie-Manager.cs, linie ok. 50–150

B) **Użycie fabryki w programie:**

- Wywołanie przykładowe w program.cs przy tworzeniu nowego zadania w menu (linie 29 - 32)

```
// Przykładowe Zadania
menedzerZadan.UtworzZadaniePrzezFabryke("Zadanie z wysokim priorytetem bez Tagów", "",
    Priorytet.Wysoki, new DateTime(2026, 2, 12));
menedzerZadan.UtworzZadaniePrzezFabryke("Zadanie z niskim pr. z Tagami", "Ma podany jeden nieistniejący domyślnie Tag",
    Priorytet.Niski, new DateTime(2021, 12, 28), new List<string>{ "Betelgeuse", "Capella", "Sirius" });
```

3.3 Cel użycia wzorca Fabryka w programie

- Oddzielenie tworzenia obiektów od logiki biznesowej.
- Pozwala centralnie decydować, jakie konkretne obiekty (Zadanie) są tworzone oraz w jaki sposób są inicjalizowane (tagi, priorytet, termin).
- Ułatwia późniejszą zmianę sposobu tworzenia zadań bez modyfikowania menedżera lub logiki programu.

3.4 Co można zmieniać lub dodawać dzięki Fabryce

A) Nowe typy produktów

- Można dodać np. **FabrykaNotatekSpecjalnych** tworząc nowe klasy **NotatkaSpecjalna**, bez zmiany kodu menedżera zadań.

B) Centralizacja logiki inicjalizacji

- Zmiana sposobu przypisywania tagów, priorytetów lub domyślnego stanu zadań wymaga tylko modyfikacji fabryki, nie całego programu.

C) Rozszerzenie programu

- Możliwość dodania nowych pól dla zadań lub notatek w przyszłości (np. autor, kategorie) bez zmiany miejsc wywołania tworzenia obiektów w całym programie.

3.5. Korzyści:

- **Oddzielenie tworzenia od użycia:** program korzysta z gotowych obiektów bez znajomości szczegółów inicjalizacji.
- **Łatwe rozszerzanie:** dodanie nowych typów wpisów, tagów lub właściwości wymaga tylko modyfikacji fabryki.
- **Centralizacja logiki inicjalizacji:** minimalizacja powtarzania kodu w menedżerach lub menu programu.
- **Otwarte na zmiany, zamknięte na modyfikacje:** zmiany w sposobie tworzenia obiektów nie wymagają modyfikacji kodu menedżera, menu ani dekoratorów.

4. Wzorzec Stan (State)

Lokalizacja: Zadanie-Manager.cs

4.1 Przyporządkowanie klas do wszystkich ról wzorca

Zadanie to klasa posiadająca stan (IStanZadania stan).

- Przechowuje referencję do obiektu reprezentującego bieżący stan zadania.
- Deleguje operacje zmiany stanu do aktualnego obiektu stanu

```
// Prosi obecny stan o zmianę stanu zadania na Wykonane
public void OznaczJakoWykonane()
{
    stan.wykonane(this);
}

// Prosi obecny stan o zmianę stanu zadania na Aktywne
public void OznaczJakoAktywne()
{
    stan.aktywne(this);
}

// Prosi obecny stan o zmianę stanu zadania na Zaległe
public void OznaczJakoZalegle()
{
    stan.zalegle(this);
}
```

Interfejs stanu

- IStanZadania - deklaruje metody wymagane do zmiany stanu zadania

```
// Stany Zadań
//
// Interfejs stanów Zadania (Wzorzec State)
public interface IStanZadania
{
    // Metody wykorzysywane przez każdy Stan
    public void wykonane(Zadanie zadanie);
    public void aktywne(Zadanie zadanie);
    public void zalegle(Zadanie zadanie);
}
```

Konkretne stany

- StanAktywne - implementuje interfejs IStanZadania.
  - Zmiana stanu z Aktywnego na Wykonane lub Zaległe.
- StanWykonane - implementuje interfejs IStanZadania.
  - Stan końcowy, brak możliwości zmiany stanu.
- StanZalegle - implementuje interfejs IStanZadania.
  - Pozwala na zmianę stanu na Aktywne lub Wykonane w zależności od terminu zadania.

4.2 Lokalizacja w kodzie i użycie

Definicja wzorca:

- IStanZadania →Zadanie-Manager.cs linie ok. 132 - 138
- StanAktywne →Zadanie-Manager.cs linie ok. 165 - 186
- StanWykonane →Zadanie-Manager.cs linie ok. 143 - 161
- StanZalegle →Zadanie-Manager.cs linie ok. 189 - 213
- Zadanie →Zadanie-Manager.cs linie ok. 19 – 125

Użycie wzorca w programie:

- W menu głównym (program.cs linie 500–550) przy zmianie stanu zadania:

```
// Wybranie odpowiedniej metody
if (command == "1") zadanie.OznaczJakoWykonane();
else if (command == "2") zadanie.OznaczJakoAktywne();
else if (command == "3") zadanie.OznaczJakoZalegle();
// Podano niepoprawna wartość
```

Metody te delegują odpowiedzialność do konkretnego obiektu stanu (**StanAktywne, StanWykonane, StanZalegle**).

4.3 Cel użycia wzorca Stan w programie

- Pozwala dynamicznie zmieniać zachowanie zadania w zależności od jego stanu.
- Każdy stan zarządza własną logiką, np.:
  - StanWykonane blokuje możliwość ponownej aktywacji lub oznaczenia jako zaległe.
  - StanZalegle sprawdza termin zadania przed zmianą na Aktywne.
- Ułatwia utrzymanie kodu – logika stanu nie jest mieszana z logiką zadania

4.4 Co można zmieniać lub dodawać dzięki State

A) Nowe stany

- Można wprowadzić np. StanWstrzymane lub StanPriorytetowe bez modyfikowania klasy Zadanie.

B) Rozszerzenie logiki stanu

- Można dodać dodatkowe akcje przy przechodzeniu między stanami, np.: powiadomienia użytkownika, logi lub automatyczne zmiany w menedżerze zadań.

C) Zależność od warunków

- Stany mogą decydować o dostępnych operacjach w zależności od kontekstu (termin, priorytet, tagi).

4.5 Wektor zmian i korzyści dla rozbudowy programu

Obecne użycie:

- Zmiana stanu zadania w menu głównym (program.cs linie 413 - 416).
- Sprawdzenie stanu zadania w menedżerze (SprawdzCzyZalegle()).

Korzyści:

- **Łatwe dodawanie nowych stanów** nie trzeba modyfikować klasy Zadanie.
- **Centralizacja logiki stanu** każda klasa stanu odpowiada za swoje zachowanie.
- **Dynamiczne zmiany zachowania** zadania mogą reagować różnie w zależności od swojego stanu.
- **Minimalizacja błędów** brak mieszania logiki stanu z logiką danych zadania.
- **Otwarte na rozszerzenia** wprowadzenie nowych typów stanu nie wymaga zmian w istniejącym kodzie menu czy menedżera.

5. Wzorzec Builder (Budowniczy) ZadanieBuilder + ZadanieDirector

5.1. Lokalizacja:

Lokalizacja:

- **Interfejs IZadanieBuilder** →Builder.cs, linie ok. 10–20
- **Klasa ZadanieBuilder** →Builder.cs, linie ok. 25–120
- **Klasa ZadanieDirector** → Director.cs, linie ok. 10–50

5.1 Przyporządkowanie klas do wszystkich ról wzorca

Builder (interfejs)

- IZadanieBuilder deklaruje wszystkie metody konfiguracyjne do tworzenia obiektu Zadanie

```
// Interfejs Buildera
public interface IZadanieBuilder
{
    ZadanieBuilder UstawTytul(string tytul);
    ZadanieBuilder UstawTresc(string tresc);
    ZadanieBuilder UstawStan(IStanZadania stan);
    ZadanieBuilder UstawPriorytet(Priorytet priorytet);
    ZadanieBuilder UstawTermin(DateTime termin);
    ZadanieBuilder UstawTagi(List<string> nazwyTagow);
}
```

Builder

- ZadanieBuilder implementuje IZadanieBuilder i przechowuje prywatne pole _zadanie (wskaźnik na aktualnie konstruowane Zadanie).
- Każda metoda Ustaw Tag()/ Ustaw termin() ustawia odpowiednie pole i zwraca referencję do buildera (return this), umożliwiając łańcuchowe wywołania.
- Metoda Build() finalizuje konstrukcję, dodaje zadanie do MenedzerZadan i resetuje builder do tworzenia nowego zadania.

Director

- ZadanieDirector to zarządza sekwencją kroków tworzenia zadania i może tworzyć gotowe konfiguracje (np. zadanie pilne, krótkoterminowe, długoterminowe).
- Używa ZadanieBuilder do skonstruowania obiektów z określonymi parametrami.

  Produkt (Zadanie)
- Zadanie to finalny obiekt tworzony przez Builder, zawiera wszystkie właściwości: tytuł, treść, stan, priorytet, termin, tagi.

5.2 Cel użycia Buildera w programie

- Umożliwia tworzenie obiektów Zadanie w sposób elastyczny, krok po kroku, bez potrzeby stosowania długich konstruktorów z wieloma parametrami.
- Pozwala definiować gotowe konfiguracje zadań (pilne, długoterminowe, krótkoterminowe) w ZadanieDirector.
- Poprawia czytelność kodu dzięki łańcuchowym wywołaniom metod (fluent interface).
- Integruje się z istniejącym menedżerem zadań (MenedzerZadan) – Builder automatycznie dodaje utworzone zadanie do listy zadań

5.3 Lokalizacja w kodzie i użycie

Definicja wzorca:

- IZadanieBuilder → Builder.cs linie 6 - 14
- ZadanieBuilder → Builder.cs linie 17 - 95
- ZadanieDirector → Director.cs linie 4 - 52

Przykładowe użycie w programie

```
// Builder i Director Zadań
ZadanieBuilder zadanieBuilder = new ZadanieBuilder();
ZadanieDirector zadanieDirector = new ZadanieDirector(zadanieBuilder);
```

```
// Przykładowe Zadanie utworzone przez Buildera i poprzez Directora
zadanieBuilder.UstawTytul("Zadanie z Buildera")
    .UstawTresc("zawiera Tagi")
    .UstawPriorytet(Priorytet.Wysoki)
    .UstawTermin(DateTime.Now.AddDays(7))
    .UstawTagi(new List<string> { "Capella", "Sirius" })
    .Build();
zadanieDirector.KonstruujZadanieDlugoterminowe("Zadanie z Directora", "(długoterminowe)");
```

5.4 Co można zmieniać lub dodawać przez Builder

A) Nowe pola zadania

- Można dodać pola typu Autor, DataUtworzenia, Komentarze, Alarmy, bez modyfikowania konstruktora Zadanie.

B) Różne konfiguracje zadań

- Director może definiować różne szablony zadań: pilne, krótkoterminowe, wysokiego priorytetu, przypisane do użytkownika itd.

C) Integracja z innymi wzorcami

- Builder może współpracować z Fabryką lub Singletonem (MenedzerZadan), aby automatycznie rejestrować nowe zadania lub generować unikalne ID.

5.5 Wektor zmian i korzyści dla rozbudowy programu

Korzyści:

- **Elastyczne tworzenie obiektów** nie trzeba pamiętać o wszystkich parametrach konstruktora.
- **Łatwe dodawanie nowych pól.** Wystarczy dodać metodę Ustawulubione() w Builderze.
- **Łatwa konfiguracja zadań**. Director umożliwia tworzenie gotowych szablonów.
- **Bezpieczne tworzenie obiektów.** Minimalizacja błędów przy przypisywaniu właściwości.
- **Fluent interface**. Czytelne, łańcuchowe wywołania metod.
- **Integracja z menedżerem zadań**. Builder sam dodaje utworzone zadanie do MenedzerZadan.

Specyficzne rozwiązania dla języka C# i frameworka .NET

1. Partial Classes (partial class)

- **Opis:**
  W C# można dzielić definicję jednej klasy na wiele plików dzięki słowu kluczowemu partial.
- **Zastosowanie w programie:**
  - Klasa Program jest zdefiniowana jako public partial class Program w kilku plikach (Program.cs, Builder.cs, Director.cs, itp.).
  - Pozwala to podzielić logikę aplikacji (menu, builder, director) na osobne pliki, zachowując jedną spójną klasę Program.
- **Korzyści:**
  - Lepsza organizacja kodu w dużych projektach.
  - Ułatwia pracę zespołową (różni programiści mogą pracować nad różnymi plikami tej samej klasy).

2. Static Imports (using static)

- **Opis:**
  W C# using static pozwala używać statycznych metod lub pól klasy bez konieczności kwalifikowania ich pełną nazwą.
- **Zastosowanie:**
  - W Program.cs jest użyte using static Program;.
  - Dzięki temu można wywoływać statyczne metody klasy Program bez prefiksu Program..
- **Korzyści:**
  - Czytelniejszy kod w miejscach, gdzie metoda statyczna jest wywoływana wielokrotnie.

3. Singletony w C#

- **Opis:**
  - Standardowy sposób implementacji Singletona w C# prywatny konstruktor + statyczna metoda zwracająca instancję.
- **Zastosowanie w programie:**
  - MenedzerNotatek, MenedzerZadan, MenedzerTagow - wszystkie zarządzają centralnymi zasobami.
- **Korzyści:**
  - Zapewnienie jednej wspólnej instancji menedżera.
  - Wszystkie części programu operują na tym samym stanie, unikając duplikacji lub niespójności danych.

Wzorce projektowe

- **Dekorator:**
  - W C# implementowany przez interfejsy (IWpis) i klasy dekorujące (DekoratorWpisow, DekoratorTagowy, DekoratorStanowy).
  - Pola prywatne i delegacja metod (protected IWpis wpis;) wykorzystują językowe możliwości C# do enkapsulacji i dziedziczenia.
- **Builder:**
  - Płynny interface w C# metoda zwraca this, co pozwala łączyć wywołania w jednej linii

State:

- Interfejs IStanZadania i klasy StanAktywne, StanWykonane, StanZalegle.
- Dynamiczne zmienianie stanu zadań bez modyfikacji logiki klasy Zadanie.

5. Kolekcje i typy generyczne

- **Użycie List<T>:**
  - Notatki, zadania i tagi są przechowywane w List<Notatka>, List<Zadanie>, List<Tag>.
  - Pozwala na łatwe dodawanie, usuwanie i iterowanie po elementach.
- **Listy stringów do tagów:**
  - Płynna konwersja string Tag w metodzie Buildera lub Fabryki.

6. Interfejsy

- **Użycie w programie:**
  - IWpis – dla Dekoratora.
  - IZadanieBuilder – dla Buildera.
  - IStanZadania – dla State.
- **Korzyści:**
  - Polimorfizm i łatwe rozszerzanie funkcjonalności bez zmiany klas bazowych.

7. Konsolowa aplikacja .NET

- Cała aplikacja jest napisana jako **aplikacja konsolowa (.NET Console App)**.
- **Biblioteki:**
  - Podstawowe klasy .NET: System, System.Collections.Generic, System.Linq, System.DateTime, System.Console.
- **Korzyści:**
  - Prosty interfejs tekstowy.
  - Łatwe testowanie wzorców projektowych i mechanizmów menu.

Instrukcja użytkownika
System zarządzania notatkami, zadaniami i tagami (aplikacja konsolowa)

1. Informacje ogólne

Program jest aplikacją konsolową służącą do zarządzania:

- **notatkami**,
- **zadaniami** (z priorytetami, terminami i stanami),
- **tagami**, które można przypisywać zarówno do notatek, jak i zadań.

Po uruchomieniu programu użytkownik obsługuje go poprzez **menu tekstowe**, wybierając odpowiednie opcje za pomocą klawiatury.

2. Menu główne obsługa programu

Po uruchomieniu aplikacji wyświetlane jest menu główne z listą komend (numerowanych od 1 do 18). Aby wykonać operację, należy wpisać **numer komendy** i zatwierdzić klawiszem Enter.

3. Zarządzanie tagami

Tagi służą do kategoryzowania notatek i zadań.

Dostępne funkcje:

- Wyświetlanie wszystkich tagów
- Dodawanie nowego tagu (jeśli tag już istnieje nie zostanie zdublowany)
- Usuwanie tagu
- Dodawanie tagu do notatki
- Dodawanie tagu do zadania
- Usuwanie tagu z notatki
- Usuwanie tagu z zadania

Wpisanie nazwy nieistniejącego tagu podczas dodawania go do notatki lub zadania spowoduje **automatyczne utworzenie tagu**.

4. Zarządzanie notatkami

Notatki składają się z:

- tytułu,
- treści,
- listy tagów (opcjonalnie).

Dostępne funkcje:

- Wyświetlanie listy notatek
- Wyświetlanie notatek wraz z przypisanymi tagami
- Dodawanie nowej notatki
- Usuwanie notatki
- Przypisywanie i usuwanie tagów z notatek

Podczas dodawania nowej notatki użytkownik może:

- podać tytuł i treść,
- dodać dowolną liczbę tagów (wpisanie pustej linii kończy dodawanie tagów).

5. Zarządzanie zadaniami

Zadania są bardziej rozbudowane niż notatki i posiadają:

- tytuł,
- treść,
- priorytet (niski, średni, wysoki),
- termin wykonania,
- stan (aktywne, wykonane, zaległe),
- listę tagów.

Dostępne funkcje:

- Wyświetlanie listy zadań
- Wyświetlanie zadań wraz z tagami
- Wyświetlanie zadań wraz z ich aktualnym stanem
- Dodawanie nowego zadania
- Usuwanie zadania
- Dodawanie i usuwanie tagów z zadań
- Zmiana stanu zadania

Podczas dodawania zadania użytkownik:

1. podaje tytuł i treść,
2. wybiera priorytet,
3. wpisuje termin wykonania,
4. opcjonalnie dodaje tagi.

6. Stany zadań

Każde zadanie posiada jeden z trzech stanów:

- **Aktywne** – zadanie aktualnie realizowane,
- **Wykonane** – zadanie ukończone,
- **Zaległe** – zadanie niewykonane w terminie.

Użytkownik może w dowolnym momencie zmienić stan wybranego zadania, wybierając odpowiednią opcję w menu

7. Identyfikatory (ID)

- Każda notatka i każde zadanie posiada unikalny identyfikator (ID).
- Operacje takie jak dodawanie tagów, usuwanie czy zmiana stanu wymagają podania poprawnego ID.
- Jeśli podane ID nie istnieje, program poinformuje o błędzie.

Jak zainstalować i uruchomić program

1. **Zainstaluj .NET SDK**
   - Pobierz i zainstaluj **.NET SDK (np. .NET 6 lub nowszy)** ze strony Microsoft.
   - Jest to wymagane do uruchomienia programu konsolowego.
2. **Pobierz projekt**
   - Skopiuj pliki projektu do jednego folderu *(wszystkie pliki *.cs z public partial class Program muszą być w tym samym projekcie).*
3. **Uruchom w Visual Studio**
   - Otwórz **Visual Studio**.
   - Wybierz **Open →Project/Solution** i otwórz plik .sln
   - Kliknij **Start (F5)** albo **Ctrl + F5**.

albo (bez Visual Studio):

4. **Uruchom z konsoli**
   - Otwórz terminal w folderze projektu.
   - Wpisz dotnet build i dotnet run
   - Program uruchomi się w konsoli i wyświetli menu.

Podział pracy w zespole

1. Kacper Stefanowicz (KS)

- Implementacja klasy Wpis oraz interfejsu IWpis
- Tworzenie klasy Tag oraz jej Menedżera
- Zaimplementowanie wszystkich Dekoratorów (DekoratorWpisów, DekoratorTagowy, DekoratorZadań, DekoratorStanowy)
- Menu główne aplikacji
- Interfejs i klasy dla stanów zadań (State)

2. Milena Zarachowicz (MZ)

- Implementacja wzorca Builder do konstrukcji zadań
- Implementacja wzorca Director do kierowania procesem budowy obiektów

3. Łukasz Piotrowski (LP)

- Klasa Notatka dziedzicząca po Wpis wraz z MenedżeremNotatek
- Klasa Zadanie dziedzicząca po Wpis wraz z MenedżeremZadan
- FabrykaNotatek
- FabrykaZadań
- Wyszukiwanie wpisów w jednej klasie (po słowach kluczowych i tagach)
- Przygotowanie dokumentacji projektu