Projekt z Programowania Obiektowego

Aleksander Mierzwa, Marcin Kuczok, Kamil Miodoński

Aplikacja wspomagająca organizację zaliczeń studenckich z interfejsem wizualnym

Spis treści

Informacje wstępne

Tytuł projektu

Tytułem projektu jest: *Aplikacja wspomagająca organizację zaliczeń studenckich z interfejsem wizualnym*.

Cele projektu

- Stworzenie intuicyjnej aplikacji umożliwiającej śledzenie terminów zaliczeń, zadań i projektów na studiach z przejrzystą wizualizacją harmonogramu

- Rozwój umiejętności programistycznych poprzez praktyczne zastosowanie:
    - Projektowania interfejsu w Windows Forms.
    - Implementacji zapisu, odczytu, edycji i usuwania danych (CRUD)
    - Algorytmów do bezpiecznej obsługi wysyłanych i odczytywanych danych
    - Systemy kontroli zmian plików Git i serwisu GitHub

Krótki opis działania

Aplikacja pozwala na zarządzanie zadaniami studenckimi w podziale na semestry. Każdy semestr to osobny profil z listą przedmiotów.

Po uruchomieniu aplikacji, wyświetla nam się ekran główny w którym możemy:

- Wybrać (zalogować się) na istniejący profil
- Stworzyć nowy profil
- Usunąć istniejący profil

Po wybraniu istniejącego semestru wyświetla nam się okno po zalogowaniu gdzie możemy albo wrócić do menu głównego albo stworzyć zadanie. Wybieramy wówczas rodzaj zadania (projekt/ćwiczenie/egzamin) a następnie wpisujemy odpowiednie wartości w opisane okienka.

Stworzone zadania wyświetlane są jako kafelki (nazywane dalej również jako karty) – w podstawowym widoku pokazują przedmiot, typ zadania tytuł i krótki opis.

Po rozwinięciu zadania wyświetlają się dodatkowo pełny opis i charakterystyczne dla każdego rodzaju zadań dodatkowe wartości, tzn. dla projektu są to członkowie projektu, dla ćwiczenia – źródło ćwiczeń np. konkretna strona z podręcznika, a dla egzaminu – zakres materiału.

Dane każdego semestru zapisywane są w osobnym pliku, aby uniknąć zbyt dużej ilości danych przy wielu semestrach. Każdy profil to osobny plik .JSON.

Dodatkowymi funkcjonalnościami są m.in. edytowanie zadań i usuwanie zadań.

Realizacja projektu

Zastosowane elementy programowania obiektowego

W naszym projekcie wykorzystaliśmy, zgodnie z wymaganiami, następujące elementy programowania obiektowego:

a) Dwa interfejsy:
   - ITaskCRUD - odpowiedzialna za operacje na zadaniach konkretnego rodzaju, konkretnie tworzenia, wyświetlania i usuwania
   - ISerialize – odpowiedzialna za operacje zapisywania informacji do pliku .JSON w oparciu o nasz własny konwerter

b) Jedną klasę abstrakcyjną – Base_Task – która jest podstawą do klas związanych z rodzajami zadań

c) Jedną klasę finalną - _Task_Manager_ - służy do zarządzania zadaniami z różnych kategorii. Zawiera ona jedną metodę DisplayTasksOnLoggedInScreen która stanowi pewnego rodzaju kontroler, który sam rozpoznaje dany rodzaj taska. Ponieważ mamy wszystkie zadania zapisane w jednej liście w pliku .JSON, musielibyśmy tworzyć różne metody – dzięki temu kontrolerowi, unikamy bałagnu w kodzie i jest to bardziej pewne rozwiązanie.

d) Jedną klasę statyczną – Side_Format - zawierająca metody do formatowania wszystkich danych (aczkolwiek w projekcie formatuje głównie stringi).

Ustanowiona struktura projektu

Nasz projekt ma ustanowioną przez nas strukturę która jest według skali ważności i użyteczności. Dzięki temu wiemy jak można zachować nazewnictwo a co za tym idzie – pełen ład i przejrzystość w naszym kodzie.

Plik projektu dzielą się na dwie kategorie (a także foldery):

a) Logika kodu – CS – tj. zapisywanie, wczytywanie, formatowanie danych

- Base – klasy które są najważniejsze oraz klasy które mogą stanowić podstawę do dla klas abstrakcyjnych itp. Nazwy tych klas rozpoczynamy od napisania „Base_" np. Base_ NazwaKlasy

- Interfaces – interfejsy które następnie implementujemy w różnych klasach, niezależnie od stopnia ważności. Nazwy tych interfejsów rozpoczynamy od wielkiej litery I np. ISerialize

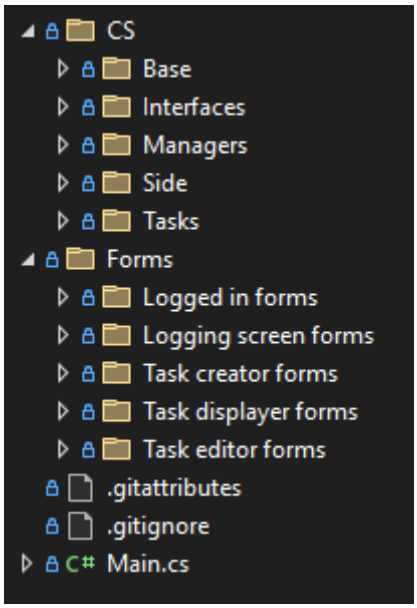
- Managers – klasy końcowe (finalne, opisane słowem kluczowym sealed) – klasy które w założeniu są zwolnione z dalszego dziedziczenia po nich. Są to klasy końcowe, które mają już zbiorczo operować na obiektach różnych klas. Nazwy klas rozpoczynamy od napisania znaku podkreślenia na początku i końcu, oraz dopisanie Manager – np. _Task_Manager_

- Side – klasy statyczne, które mają wspomagać w formatowaniu danych aby uniknąć kopiowania wielu linii kodu. Nazewnictwo rozpoczynamy od napisania „Side_" np. Side_Format

- Tasks – klasy dziedziczące po klasie Base_Task oraz implementujące interfejsy ITaskCRUD i ISerialize. Każda z tych klas powinna przejmować wspólne pola, właściwości i metody z klasy bazowej, z możliwością ich nadpisywania. Jednocześnie każda konkretna klasa typu Task powinna definiować własne, unikalne pola i właściwości, tak aby różniła się od pozostałych. Nazwę każdego nowego rodzaju zadania rozpoczynamy od wpisania „Task_" np. Task_Exam.

b) Interfejs graficzny windows forms – Forms – wewnątrz tych plików wywołujemy jedynie metody z plików z CS

Plik main.cs znajduje się poza tymi kategoriami i jest on w tym samym folderze co powyższe dwa katalogi.



Zastosowanie kluczowych mechanizmów programowania obiektowego

Hermetyzacja

Enkapsulacja jest fundamentalnym mechanizmem programowania obiektowego, dlatego w naszym projekcie mocno przykuliśmy do tego uwagę. Dla każdego pola przypisana jest także właściwość, która tym polem steruje. Oczywiście samo pole jest niedostępne poza klasą. Dostępna jest jedynie właściwość, która w swoim ciele formatuje w odpowiedni sposób wartość dla rodzaju i zastosowania pola, jednocześnie sprawdzając czy formatowanie się powiodło – jeśli nie, to wyrzuca odpowiedni wyjątek, przez co zmusza programistę do implementacji wprowadzania logiki w blokach try-catch.

Przykładowe zastosowanie hermetyzacji wewnątrz klasy abstrakcyjnej Base_Task dla krótkiego opisu:

```
/// <summary> Prosty, krótki opis zwiazany z zadaniem

protected string _shortDescription;
Odwołania: 18
public string ShortDescription
{
    get { return this._shortDescription; }
    set
    {
        if (!string.IsNullOrEmpty(value) && !string.IsNullOrWhiteSpace(value) && value.Length > 0 && value.Length <= 32)
            this._shortDescription = value;

        else if (value.Length > 32)
            throw new ArgumentException("Opis jest za długi! Maksymalnie 32 znaki!", "ShortDescriptionTooLong");

        else if (string.IsNullOrEmpty(value) || string.IsNullOrWhiteSpace(value))
            throw new ArgumentException("Opis nie może być pusty!", "EmptyShortDescription");
    }
}
```

Dziedziczenie

W naszym projekcie zastosowaliśmy logikę dziedziczenia w przypadku tworzenia różnych rodzajów zadań. Mamy jedną klasę abstrakcyjną, która stanowi pewien wzorzec dla pozostałych klas które po niej dziedziczą. Klasa Base_Task zawiera takie pola jak tytuł zadania, pełny opis zadania, krótki opis zadania, wybrany przedmiot itp. – w skrócie, są to pola, które powinny zostać zaimplementowane także we wszystkich rodzajach zadań, ponieważ niezależnie, czy to projekt, czy egzamin – każdy z nich dotyczy jakiegoś przedmiotu, każdy z nich ma swój termin końcowy itp.

Coś co wyróżnia egzamin od projektu, to to, czy projekt jest grupowy i jeśli tak to jacy członkowie biorą w nich udział. Egzamin jest indywidualny (przynajmniej w założeniu), zatem nie powinien mieć w sobie pola _members. Egzamin ma natomiast swoją własną indywidualną cechę – mianowicie zawsze dotyczy jakiegoś zakresu materiału, dzięki czemu zamiast podawać członków zespołu, podajemy zakres materiału. Taka operacja pozwala nam zachować czytelność i logiczną strukturę projektu bez niepotrzebnego powielania kodu.

Polimorfizm

Klasa abstrakcyjna Base_Task, jak wyżej napisaliśmy, stanowi wzorzec dla wszystkich klas związanych z zadaniami, które po niej dziedziczą. Klasa Base_Task posiada cztery metody oznaczone słowem kluczowym abstract z czego:

a) Dwie metody związane bezpośrednio z operacjami na zadaniach:

```
/// <summary> Uruchamianie kreatora zadań

Odwołania: 6
public abstract void TaskCreator();


/// <summary> Uruchamianie ekranu wyświetlenia zadania

Odwołania: 6
public abstract void TaskDisplayer();
```

b) Dwie metody związane z wyświetlaniem zadań w formie kart

```
/// <summary> Logika wyświetlania kart na ekranie po zalogowaniu

Odwołania: 4
public abstract void ShowTaskCard(Panel panelToShowOn);


/// <summary> Akcja która się wykona po naciśnięciu karty

Odwołania: 9
public abstract void CardPanel_Click(object sender, EventArgs e);
```

Powyższe metody, zgodnie z istotą dziedziczenia metod abstrakcyjnych, nadpisują (przeciążają) je wedle własnych potrzeb i zastosowań. Poniżej znajduje się przykład nadpisanych metod przez klasę Task_Exam

```
Odwołania: 2
public override void TaskCreator()
{
    Form_CreateExam screenExamCreator = new Form_CreateExam();
    screenExamCreator.Show();
}

Odwołania: 2
public override void TaskDisplayer()
{
    Form_DisplayExam screenTaskDisplayer = new Form_DisplayExam(this);
    screenTaskDisplayer.Show();
}
```

Wyświetlanie egzaminów w formie kart na ekranie po wybraniu semestru:

```
Odwołania: 2
public override void ShowTaskCard(Panel panelToShowOn)
{

    Panel cardPanel = new Panel...;


    Label titleLabel = new Label...;


    Label taskType = new Label...;

    Label shortDescriptionLabel = new Label...;

    Label deadlineLabel = new Label...;


    cardPanel.Controls.Add(shortDescriptionLabel);
    cardPanel.Controls.Add(taskType);
    cardPanel.Controls.Add(titleLabel);
    cardPanel.Controls.Add(deadlineLabel);

    cardPanel.Click += CardPanel_Click;
    foreach (Control control in cardPanel.Controls)
        control.Click += CardPanel_Click;

    panelToShowOn.Controls.Add(cardPanel);
    Base_AppState.CardCount++;
}
```

Oraz nadpisanie akcji, która wykona się po naciśnięciu karty:

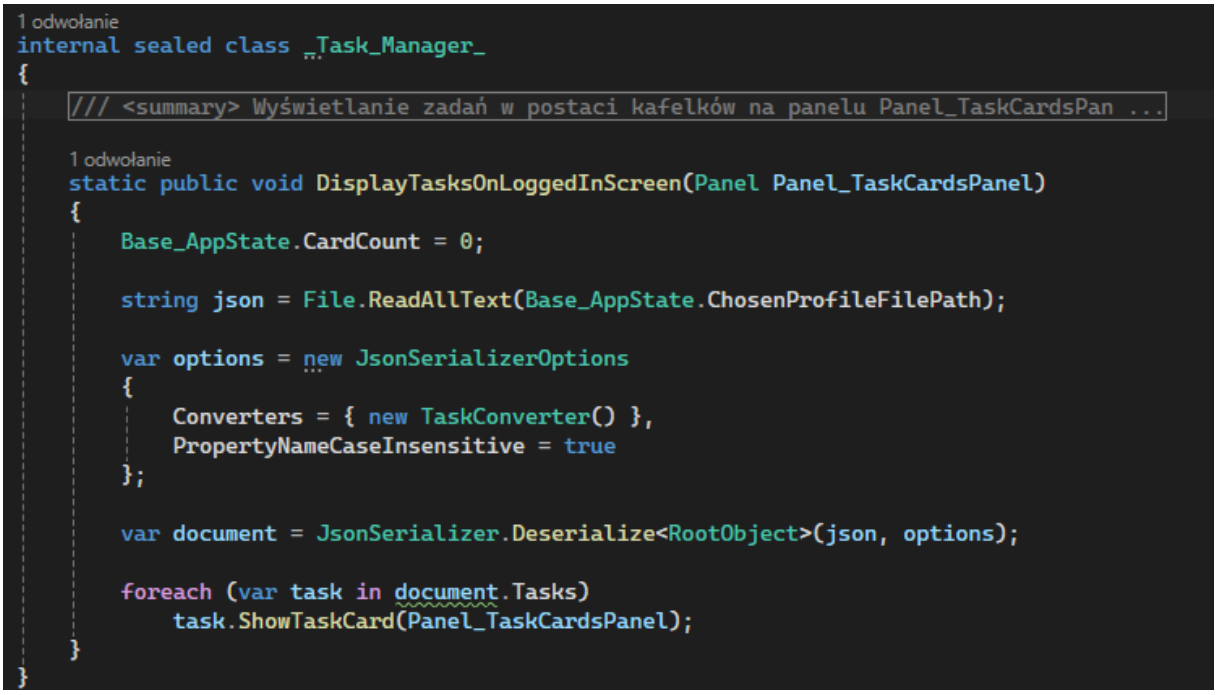
```
Odwołania: 3
public override void CardPanel_Click(object sender, EventArgs e) => DisplayTask();
```

```
Odwołania: 2
public void DisplayTask()
{
    if (this is Task_Exam examTask)
    {
        for (int i = Application.OpenForms.Count - 1; i >= 0; i--)
            Application.OpenForms[i].Close();

        this.TaskDisplayer();
    }

    else
        throw new ArgumentException("FATAL: konflikt typów zadań", "FATALTaskTypeConflict");
}
```

Przykładowe wywołanie metody w klasie _Task_Manager_



```
1 odwołanie
internal sealed class _Task_Manager_
{
    /// <summary> Wyświetlanie zadań w postaci kafelków na panelu Panel_TaskCardsPan ...

    1 odwołanie
    static public void DisplayTasksOnLoggedInScreen(Panel Panel_TaskCardsPanel)
    {
        Base_AppState.CardCount = 0;

        string json = File.ReadAllText(Base_AppState.ChosenProfileFilePath);

        var options = new JsonSerializerOptions
        {
            Converters = { new TaskConverter() },
            PropertyNameCaseInsensitive = true
        };

        var document = JsonSerializer.Deserialize<RootObject>(json, options);

        foreach (var task in document.Tasks)
            task.ShowTaskCard(Panel_TaskCardsPanel);
    }
}
```

Wnioski

Realizacja powyższego projektu pozwoliła nam zrozumieć zarówno techniczne, jak i organizacyjne aspekty pracy zespołowej w środowisku programistycznym. Choć mogłoby się wydawać, że praca w grupie jest łatwiejsza — ponieważ zadania można rozdzielić między większą liczbę osób — w praktyce okazała się bardziej wymagająca. Szczególnie istotnym wyzwaniem było zarządzanie konfliktami — nie tylko interpersonalnymi, ale również tymi wynikającymi z pracy z systemem kontroli wersji Git. Częste konflikty podczas scalania zmian uświadomiły nam, jak ważna jest dobra komunikacja i synchronizacja działań w zespole.

Projekt pokazał także, jak kluczowe znaczenie ma posiadanie spójnej wizji aplikacji oraz jej szczegółowe rozplanowanie jeszcze przed rozpoczęciem kodowania. Odpowiednio przygotowana dokumentacja i plan działania znacząco ułatwiają późniejsze etapy implementacji. Zrozumieliśmy, że choć samo programowanie — przy znajomości podstawowych mechanizmów takich jak enkapsulacja, dziedziczenie czy polimorfizm — nie musi być bardzo trudne, to prawdziwym wyzwaniem jest właściwa organizacja pracy i utrzymanie porządku w całym procesie tworzenia oprogramowania.