

Materiały do przedmiotu:

Podstawy programowania

Laboratorium nr 8

Dziedziczenie

Polimorfizm

Autor:

dr Michał Dolecki



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć	3
Cel laboratorium	6
Instrukcja laboratoryjna	7
Ćwiczenia samodzielne	16
Pytania pomocnicze	21
Podsumowanie	21
Literatura	21



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Jednym z ważniejszych zagadnień programowania obiektowego, które ułatwia codzienną pracę programistom, jest code reusability – a więc wielokrotne wykorzystanie kodu w tworzonych projektach. Pozwala to na szybszą i efektywniejszą pracę nad kodem, a posiadając zgromadzoną dużą bazę przygotowanych bibliotek, możemy łatwo dopasowywać je do nowo tworzonych rozwiązań.

Agregacja i kompozycja

Wielokrotne wykorzystanie implementacji może być realizowane w tak zwany sposób poziomy – w sytuacji, w której wykorzystujemy obiekty jednej klasy w drugiej klasie jako pola, a także wykorzystujemy ich zachowania zakodowane w metodach. Takie składanie nowych klas z wykorzystaniem już istniejących obiektów nazywamy agregacją bądź kompozycją.

Agregacja i kompozycja to dwa podstawowe mechanizmy wykorzystywania istniejących klas do budowania bardziej złożonych struktur:

Agregacja – reprezentuje relację "posiada" (has-a), gdzie obiekty mogą istnieć niezależnie:

- Obiekt części może należeć do wielu obiektów całości
- Cykl życia obiektów jest niezależny

Przykład: Student i Kurs – student może uczestniczyć w wielu kursach, kurs może mieć wielu studentów

Kompozycja – reprezentuje silniejszą relację "zawiera" (contains-a), gdzie obiekty części są ściśle zależne od całości:

- Obiekt części należy tylko do jednej całości
- Cykl życia jest zsynchronizowany – usunięcie obiektu całości powoduje usunięcie części

Przykład: Samochód i Silnik – silnik jest integralną częścią samochodu

Różnica między tymi dwoma zagadnieniami będzie szczegółowo omawiana na inżynierii oprogramowania – na tym etapie będziemy stosować ogólnie pojęcie agregacja albo będziemy stosować te pojęcia wymiennie. Tego typu pracę z kodem macie już Państwo przećwiczone w poprzednich laboratoriach.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Dziedziczenie

Drugim typem wykorzystania implementacji jest tak zwana relacja w pionie – wykorzystujemy tutaj jeden z filarów programowania obiektowego, mianowicie dziedziczenie. W relacji pionowej tworzymy hierarchię klas, nazywając elementy klasami nadrzędnymi i podrzędnymi, bądź klasami bazowymi i klasami dziedziczącymi.

Generalnie, klasa znajdująca się wyżej w hierarchii modeluje bardziej ogólny byt z dziedziny problemowej. Schodząc w dół, umieszczamy klasy bardziej szczegółowe. Klasy te dostarczają dodatkowych funkcjonalności, rozszerzając możliwości klasy bazowej. Klasa dziedzicząca może wykonywać wszystkie nieprywatne metody klasy bazowej, a także ma dostęp do nieprywatnych pól tej klasy.

Dobrym przykładem do zrozumienia tego związku może być relacja między klasą Student a StudentErasmus. W klasie bazowej Student znajdują się typowe pola, takie jak na przykład imię, nazwisko, numer indeksu, kierunek studiów, data urodzenia oraz funkcje związane ze studiowaniem, takie jak na przykład WyliczŚrednią(), SprawdźCzyOtrzymaStypendium(). Natomiast StudentErasmus dziedziczy wszystkie operacje z klasy Student, ale dodatkowo ma przypisany obiekt opiekuna oraz metody związane z kontaktem z opiekunem w sytuacjach wymagających jego interwencji.

Od samego początku należy zwrócić tutaj uwagę na właściwe rozumienie relacji dziedziczenia, ponieważ niewłaściwe jej użycie oraz generalnie nadużywanie w tworzonych projektach prowadzi do licznych błędów.

Najważniejszą regułą służącą do sprawdzenia, czy należy użyć relacji dziedziczenia, jest sprawdzenie czy w dziedzinie problemowej między dwiema klasami występuje związek, który opisujemy terminem "is-a" – a więc klasa potomna jest szczególnym przypadkiem klasy bazowej. Jeżeli zamiast tego sformułowania użyliśmy "has-a", wówczas właściwe będzie wykorzystanie agregacji.

Właściwe użycie dziedziczenia ("is-a"):

- Pojazd – Samochód (Samochód jest Pojazdem),
- Samochód – SamochódOsobowy (Samochód osobowy jest Samochodem),
- Zwierzę – Pies (Pies jest Zwierzęciem),
- Kształt – Prostokąt (Prostokąt jest Kształtem),



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- KontoUżytkownika – KontoAdministratora (Konto administratora jest Kontem użytkownika).

Niewłaściwe dla dziedziczenia, lepsza agregacja ("has-a"):

- Koło – Punkt (Koło ma Punkt będący jego środkiem, ale Punkt nie jest Kołem),
- Samochód – Silnik (Samochód ma Silnik, ale Silnik nie jest Samochodem),
- Student – Adres (Student ma Adres, ale Adres nie jest Studentem),
- Szkoła – SalaLekcyjna (Szkoła ma Sale lekcyjne, ale Sala nie jest Szkołą).

Na pierwszy rzut oka może się to wydawać oczywiste, ale skupmy się na pierwszym przykładzie koła i punktu. Załóżmy, że ktoś zbudował klasę Punkt, która zawiera dwie składowe x i y . Następnie chciałby zbudować klasę Koło. Wówczas może pomyśleć, że w zasadzie do zdefiniowania koła potrzebuje punktu i jeszcze promienia. W związku z czym spróbowałby stworzyć relację dziedziczenia tak, aby Koło dziedziczyło po Punkcie. Wówczas koło uzyska dwie składowe x i y z klasy bazowej wraz ze wszystkimi metodami, a w klasie potomnej wystarczy dołożyć tylko jedno pole na promień i związane z nim metody.

Na pierwszy rzut oka implementacja jest prosta i wykorzystuje mechanizmy programowania obiektowego. Niestety pojawiają się pewne problemy. Jeżeli zbudujemy w tym samym projekcie odcinek definiowany za pomocą dwóch jego końców będących punktami, możemy przekazać jako końce odcinka obiekty typu Koło, ponieważ stworzyliśmy koło jako szczególny przypadek punktu – a to niestety nie jest sensowne.

Poza tym jeżeli dla punktu mamy metodę obliczającą odległość od początku układu współrzędnych, to ta metoda będzie odziedziczona w kole i bez zmiany jej implementacji będzie zwracać odległość środka, a nie całego koła.

Problemy z porównywaniem obiektów – dla punktu porównujemy współrzędne, a co z dwoma kołami o tym samym środku i różnych promieniach?

Również metody spoza klasy, które oczekują w parametrze obiektu typu punkt będą problemem, bo jak miałyby się zachować metoda sprawdzająca czy przekazany w parametrze punkt leży w pierwszej ćwiartce, jeżeli zostanie wywołana z argumentem typu Koło?

Zatem zawsze zadawaj sobie pytanie "Czy X jest szczególnym rodzajem Y ?" Jeśli odpowiedź brzmi NIE, prawdopodobnie potrzebujesz agregacji, a nie dziedziczenia.

Polimorfizm



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Kolejnym ważnym aspektem włączenia klas w hierarchię dziedziczenia jest możliwość modyfikowania zachowania przez klasę dziedziczącą. Obiekt stworzony na podstawie klasy potomnej dziedziczy wszystkie zachowania, czyli wszystkie metody z klasy bazowej. Natomiast może się okazać, że któraś z tych metod powinna być wykonana inaczej. Wówczas w klasie potomnej wprowadza się nową implementację metody o nagłówku identycznym jak metoda z klasy bazowej, ale z innym ciałem.

W implementacji tej nowej metody można wykorzystać metodę z klasy bazowej i dodać nowe fragmenty kodu, bądź też stworzyć zupełnie nową implementację. W takiej sytuacji w obu klasach występują dwie metody o tych samych nazwach, ale o innym ciele – stąd nazwa polimorfizm, a więc wielopostaciowość.

Polimorfizm pozwala pisać kod, który działa na ogólnych typach, ale wykonuje specyficzne operacje w zależności od rzeczywistego typu obiektu w czasie wykonania programu.

Cel laboratorium

Celem zajęć laboratoryjnych jest praktyczne opanowanie mechanizmów dziedziczenia i polimorfizmu w języku C#. Uczestnicy nauczą się poprawnie projektować hierarchie klas, stosując relację "is-a" oraz wykorzystywać mechanizmy przesłaniania metod. W trakcie ćwiczeń studenci przećwiczą tworzenie klas bazowych i pochodnych, operować na ich składowych i świadomie dobierać modyfikatory dostępu. Dodatkowo zajęcia mają na celu wyrobienie umiejętności rozróżniania przypadków, w których należy zastosować dziedziczenie od tych wymagających agregacji, co pozwoli uniknąć typowych błędów projektowych.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Instrukcja laboratoryjna

Prześledzimy poszczególne zagadnienia związane z dziedziczeniem i polimorfizmem na przykładzie zasobów multimedialnych. Zaczniemy od ogólnej klasy `MultimediaResource`, z której dziedziczyć będą klasy `Audio`, `Image` i `Movie`.

Przykład 8.1. Dziedziczenie

Klasa bazowa posiada pola wspólne dla wszystkich zasobów multimedialnych:

- autor,
- tytuł,
- rozmiar pliku,
- data utworzenia,
- tagi (lista napisów).

Kod klasy `MultimediaResource`:

```
public class MultimediaResource
{
    public string Author { get; set; }
    public string Title { get; set; }
    public int FileSizeMB { get; set; }
    public DateTime CreationDate { get; set; }
    public List<string> Tags { get; set; }

    protected MultimediaResource(string title, string author, int
    fileSizeMB)
    {
        Title = title;
        Author = author;
        CreationDate = DateTime.Now;
        Tags = new List<string>();
        FileSizeMB = fileSizeMB;
    }

    public virtual void AddTag(string tag)
    {
        Tags.Add(tag);
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
public virtual void RemoveTag(string tag)
{
    Tags.Remove(tag);
}

public void DisplayTags()
{
    Console.WriteLine($"Tags: {string.Join(", ", Tags)}");
}
}
```

Na podstawie tej klasy zbudujemy teraz klasę dziedziczącą Audio.

```
public class Audio : MultimediaResource
{
    public string Album { get; set; }
    public TimeSpan Duration { get; set; }
    public string FileType { get; set; }
    public int Bitrate { get; set; }

    public Audio(string title, string author, string album, TimeSpan
duration, int fileSizeMB, string fileType = "MP3")
        : base(title, author, fileSizeMB)
    {
        Album = album;
        Duration = duration;
        Bitrate = 320; // domyślna wartość
        FileType = fileType;
    }

    public void Play()
    {
        Console.WriteLine($"Playing audio: {Title} - {Author}");
    }

    public void Stop()
    {
        Console.WriteLine($"Stopped audio: {Title}");
    }
}
```

Zwróćmy uwagę na wywołanie konstruktora klasy bazowej w konstruktorze klasy dziedziczącej Audio. Możemy teraz przetestować działanie klasy Audio w metodzie Main. W szczególności należy zwrócić uwagę na możliwość wykorzystania przez obiekt klasy Audio pól klasy bazowej oraz metod związanych z tagami, które są



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



odziedziczone z klasy bazowej. Widzimy tutaj wielokrotne wykorzystanie implementacji.

Przykład 8.2. Polimorfizm (method override)

Dodajmy do klasy Audio metodę następującej postaci:

```
public override void AddTag(string tag)
{
    base.AddTag(tag);
    Console.WriteLine($"Added tag '{tag}' to audio file {Title}");
}
```

Jest to metoda, która zmienia działanie metody AddTag z klasy bazowej realizując polimorfizm. Metody takie mogą posiadać absolutnie nową implementację albo wykorzystywać oryginalną postać metody z klasy bazowej i ją rozszerzać, tak jak ta powyżej. Metody polimorficzne oznaczamy słowem kluczowym override, a metody z klasy bazowej możemy wywoływać stosując słowo base.

Przykład 8.3. Klasa abstrakcyjna

Tworząc hierarchie klas, na jej szczycie może znaleźć się tak ogólna klasa, że ciężko sobie wyobrazić istnienie konkretnego obiektu tej klasy. Istniejące w niej metody mogą mieć sens w każdej klasie dziedziczącej, ale w ogólnym przypadku nie można podać sensownej implementacji takiej metody. W analizowanym przypadku można zaprezentować każdy typów multimedialnych jak na przykład odtworzyć dźwięk czy wyświetlić zdjęcie, ale jak zaimplementować taką metodę dla klasy bazowej MultimediaResource? Jeżeli w klasie występuje taka sytuacja, to tworzymy metodę abstrakcyjną, co wymaga, aby cała klasa stała się klasą abstrakcyjną. Uzupełniona wersja klasy wygląda następująco:

```
public abstract class MultimediaResource
{
    public string Author { get; set; }
    public string Title { get; set; }
    public int FileSizeMB { get; set; }
    public DateTime CreationDate { get; set; }
    public List<string> Tags { get; set; }
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
protected MultimediaResource(string title, string author, int
fileSizeMB)
{
    Title = title;
    Author = author;
    CreationDate = DateTime.Now;
    Tags = new List<string>();
    FileSizeMB = fileSizeMB;
}

public abstract void DisplayInfo();

public virtual void AddTag(string tag)
{
    Tags.Add(tag);
}

public virtual void RemoveTag(string tag)
{
    Tags.Remove(tag);
}

public void DisplayTags()
{
    Console.WriteLine($"Tags: {string.Join(", ", Tags)}");
}
}
```

Metoda Display jest abstrakcyjna, nie posiada implementacji i została oznaczona słowem kluczowym `abstract`, podobnie jak cała klasa. Nie jest możliwe już teraz tworzenie obiektów tej klasy, ponieważ nie możliwe byłoby wykonanie na ich rzecz metody bez implementacji. Natomiast metoda ta jest dziedziczona przez klasy potomne, które albo dostarczą odpowiedniej dla ich typu implementacji, albo same muszą stać się abstrakcyjne. Poniżej klasa Audio dostarczająca implementacji metody Display:

```
public class Audio : MultimediaResource
{
    public string Album { get; set; }
    public TimeSpan Duration { get; set; }
    public string FileType { get; set; }
    public int Bitrate { get; set; }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
public Audio(string title, string author, string album, TimeSpan
duration, int fileSizeMB, string fileType = "MP3")
    : base(title, author, fileSizeMB)
{
    Album = album;
    Duration = duration;
    Bitrate = 320; // domyślna wartość
    FileType = fileType;
}

public override void DisplayInfo()
{
    Console.WriteLine($"Audio: {Title}");
    Console.WriteLine($"Artist: {Author}");
    Console.WriteLine($"Album: {Album}");
    Console.WriteLine($"Duration: {Duration:mm\\:ss}");
    Console.WriteLine($"File Type: {FileType}, Bitrate:
{Bitrate}kbps");
    Console.WriteLine($"File Size: {FileSizeMB}MB");
    DisplayTags();
}

public override void AddTag(string tag)
{
    base.AddTag(tag);
    Console.WriteLine($"Added tag '{tag}' to audio file {Title}");
}

public void Play()
{
    Console.WriteLine($"Playing audio: {Title} - {Author}");
}

public void Stop()
{
    Console.WriteLine($"Stopped audio: {Title}");
}
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 8.4. Kolejne klasy dziedziczące

Przygotowana w ten sposób hierarchia klas jest łatwo rozszerzalna o nowe typy multimediów. Poniżej znajduje się implementacja kolejnej klasy tym razem dla zdjęć:

```
public class Image : MultimediaResource
{
    public int Width { get; set; }
    public int Height { get; set; }
    public string Format { get; set; }
    public string Resolution { get; set; }

    public Image(string title, string author, int width, int height, int
fileSizeMB, string format = "JPEG")
        : base(title, author, fileSizeMB)
    {
        Width = width;
        Height = height;
        Format = format;
        Resolution = $"{width}x{height}";
    }

    public override void DisplayInfo()
    {
        Console.WriteLine($"Image: {Title}");
        Console.WriteLine($"Photographer: {Author}");
        Console.WriteLine($"Resolution: {Resolution}");
        Console.WriteLine($"Format: {Format}");
        Console.WriteLine($"File Size: {FileSizeMB}MB");
        DisplayTags();
    }

    public override void AddTag(string tag)
    {
        base.AddTag(tag);
        Console.WriteLine($"Added tag '{tag}' to image {Title}");
    }

    public void Resize(int newWidth, int newHeight)
    {
        Width = newWidth;
        Height = newHeight;
        Resolution = $"{newWidth}x{newHeight}";
        Console.WriteLine($"Image resized to: {Resolution}");
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
public double CalculateAspectRatio()
{
    return (double)Width / Height;
}

public string GetOrientation()
{
    if (Width > Height) return "Landscape";
    if (Height > Width) return "Portrait";
    return "Square";
}
}
```

Przykład 8.5. Operowanie na obiektach za pomocą klasy bazowej

Utworzenie hierarchii dziedziczenia bazuje na fakcie, że nagranie i zdjęcie są szczególnym typem multimediów. Możemy więc stosować zmienne typu **MultimediaResource** przypisując do nich obiekty typów dziedziczących. Sama klasa bazowa jest abstrakcyjna, więc nie można stworzyć obiektu jej typu, ale można przechowywać obiekty typów pochodnych.

```
class Program
{
    static void Main()
    {
        // Tworzenie obiektów jako referencje klasy bazowej
        MultimediaResource
        MultimediaResource photo = new Image("Mountain Landscape", "Nature
        Photographer",
                                           1920, 1080, 8, "JPEG");

        MultimediaResource song = new Audio("Imagine", "John Lennon",
        "Imagine",
                                           TimeSpan.FromMinutes(3.02), 7,
        "MP3");

        // Dodawanie tagów - polimorfizm w działaniu
        photo.AddTag("nature");
        photo.AddTag("landscape");
        photo.AddTag("mountains");
        song.AddTag("classic");
        song.AddTag("peace");
        song.AddTag("1971");
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
// Wyświetlanie informacji - każdy obiekt używa swojej własnej
implementacji DisplayInfo()
Console.WriteLine("=== PHOTO INFORMATION ===");
photo.DisplayInfo();

Console.WriteLine("\n=== SONG INFORMATION ===");
song.DisplayInfo();

// Demonstracja, że to naprawdę są różne typy obiektów
Console.WriteLine("\n=== TYPE INFORMATION ===");
Console.WriteLine($"Photo actual type: {photo.GetType()}");
Console.WriteLine($"Song actual type: {song.GetType()}");

// Próba dostępu do metod specyficznych dla klas pochodnych
Console.WriteLine("\n=== SPECIFIC METHODS ACCESS ===");

// Aby użyć metod specyficznych, potrzebujemy rzutowania
if (photo is Image image)
{
    Console.WriteLine($"Photo orientation:
{image.GetOrientation()}");
    Console.WriteLine($"Aspect ratio:
{image.CalculateAspectRatio():F2}");
}

if (song is Audio audio)
{
    audio.Play();
    audio.Stop();
}
}
```

W kodzie pokazano również jak rzutować obiekty do odpowiednich typów jeżeli będzie to potrzebne.

Możemy również budować metody operujące na typach bazowych (również abstrakcyjnych) tworząc bardzo elastyczne implementacje.

```
class Program
{
    public static void AddTags(MultimediaResource resource, List<string>
tags)
    {
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
foreach (string tag in tags)
{
    resource.AddTag(tag); // Korzystamy z istniejącej metody AddTag
}
Console.WriteLine($"Added {tags.Count} tags to {resource.Title}");
}

static void Main()
{
    // Tworzenie obiektów jako referencje klasy bazowej
    MultimediaResource photo = new Image("Mountain Landscape", "Nature
Photographer",
                                     1920, 1080, 8, "JPEG");

    MultimediaResource song = new Audio("Imagine", "John Lennon",
"Imagine",
                                     TimeSpan.FromMinutes(3.02), 7,
"MP3");

    // Tworzenie list tagów
    List<string> photoTags = new List<string>
    {
        "nature",
        "landscape",
        "mountains",
        "outdoors",
        "scenery"
    };

    List<string> songTags = new List<string>
    {
        "classic",
        "peace",
        "1971",
        "rock",
        "inspirational"
    };

    // Użycie zewnętrznej metody AddTags - dodawanie wielu tagów na raz
    Console.WriteLine("=== ADDING TAGS TO PHOTO ===");
    AddTags(photo, photoTags);

    Console.WriteLine("\n=== ADDING TAGS TO SONG ===");
    AddTags(song, songTags);
    // Wyświetlanie informacji z nowymi tagami
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
Console.WriteLine("\n=== UPDATED PHOTO INFORMATION ===");
photo.DisplayInfo();

Console.WriteLine("\n=== UPDATED SONG INFORMATION ===");
song.DisplayInfo();

// Dodatkowa demonstracja - użycie z pojedynczym tagiem
Console.WriteLine("\n=== ADDING SINGLE ADDITIONAL TAGS ===");
AddTags(photo, new List<string> { "sunset" });
AddTags(song, new List<string> { "piano" });

Console.WriteLine("\n=== FINAL TAGS ===");
photo.DisplayTags();
song.DisplayTags();
}
}
```

Ćwiczenia samodzielne

Zadanie 8.1.

Stwórz klasę `Film` dziedziczącą po `MultimediaResource`, która reprezentować będzie filmy w systemie zarządzania multimediami.

Klasa powinna zawierać następujące dodatkowe pola:

- `Director (string)` – reżyser filmu,
- `Duration (TimeSpan)` – czas trwania filmu,
- `ReleaseYear (int)` – rok produkcji,
- `Genre (string)` – gatunek filmu,
- `Cast (List<string>)` – lista aktorów występujących w filmie.

Zaimplementuj konstruktor przyjmujący parametry:

- `title (string)` – tytuł filmu,
- `director (string)` – reżyser,
- `duration (TimeSpan)` – czas trwania,
- `fileSizeMB (int)` – rozmiar pliku,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- releaseYear (int) – rok produkcji,
- genre (string) – gatunek (domyślnie "Unknown").

Konstruktor powinien:

- Wywołać konstruktor klasy bazowej z odpowiednimi parametrami,
- Ustawić autora na "Film" (stała wartość) albo zgodnie z polem reżyser,
- Inicjalizować listę Cast jako pustą listę.

Wymagane przesłonięcia (override):

- DisplayInfo() – przesłonięta metoda z klasy bazowej, która wyświetla:
 - Tytuł, reżysera, rok produkcji, gatunek,
 - Czas trwania w formacie godzin:minuty,
 - Rozmiar pliku,
 - Listę tagów.
- AddTag() – przesłonięta metoda z dodatkowym komunikatem specyficznym dla filmów.

Nowe metody specyficzne dla filmu:

- AddActor(string actor) – dodaje aktora do listy Cast,
- DisplayCast() – wyświetla listę aktorów,
- GetAge() – oblicza wiek filmu (bieżący rok minus rok produkcji).

Sprawdź działanie klasy w głównym programie. Zastosuj metodę AddTags z przykładu do obiektu nowej klasy.

Zadanie 8.2.

Napisz klasę Car posiadającą następujące pola chronione:

- brand,
- fuelCapacity (pojemność baku),
- maxSpeed (prędkość maksymalna),
- fuelUsage (zużycie paliwa na 100 km).



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Posiada ona następujące funkcje publiczne:

- konstruktor ustawiający wszystkie pola na podstawie swoich parametrów,
- `Drive(float speed, float distance)` – wyświetla komunikat mówiący, jak szybko samochód pojedzie (nie szybciej niż jego prędkość maksymalna), ile litrów paliwa zostanie zużytych na trasie, ile razy po drodze będzie musiał tankować i ile czasu zajmie podróż (każde tankowanie zajmuje 10 minut). Na początku podróży bak samochodu jest pełny. Jeśli samochód jedzie za wolno (poniżej 30% prędkości maksymalnej) lub za szybko (powyżej 80% prędkości maksymalnej), to zużycie paliwa wzrasta o 20%.

W rozwiązaniu można wykorzystać dodatkowe metody (np. dla wyznaczenia chwilowego zużycia paliwa).

Następnie zdefiniuj klasę `Cabriolet` dziedziczącą po klasie `Car`. Dodatkowo posiada ona pole logiczne `roofOpen` ustawiane w konstruktorze na `false` oraz funkcje:

- `OpenRoof()`
- `CloseRoof()`

W zmodyfikowanej funkcji `Drive` wyświetlamy informację o tym, czy dach kabrioletu jest otwarty. Dodatkowo należy wziąć pod uwagę, że z otwartym dachem cabriolet pali o 15% więcej.

Zadanie 8.3.

Zdefiniuj klasy opisujące pracownika zatrudnionego tylko na etat oraz pracownika mającego oprócz etatu godziny nadliczbowe. Dokładniej, klasa `Employee` pozwala na zapamiętanie następujących danych o pracowniku:

- `pesel` – ciąg cyfr, domyślnie pusty łańcuch tekstowy albo `long`,
- `firstName`, `lastName` – łańcuchy tekstowe,
- `salary` – miesięczna pensja pracownika w groszach, liczba całkowita, domyślnie 0 zł 0 gr.

oraz metody:

- `SetPesel`, `SetFirstName`, `SetLastName`, `SetSalary` – ustawiają wartości pól,
- `GetGross` – zwraca wartość pola `salary`,
- `GetDeductions` – wyznacza kwotę potrąceń w groszach, 30% z `Gross`,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- ShowPaymentInfo – wyświetla na ekranie imię, nazwisko, kwotę potrąceń, kwotę do wypłaty; kwoty należy wyświetlić w złotych i groszach zawsze z dwiema cyframi groszy,
- Raise – pozwalająca na zwiększenie pensji o zadaną kwotę w groszach.

Klasa EmployeeOvertime jest klasą potomną klasy Employee posiadającą dodatkowo pola:

- overtimeHours – ilość godzin nadliczbowych (liczba rzeczywista),
- overtimeRate – stawka za jedną godzinę nadliczbową w groszach,

oraz metody:

- SetOvertimeHours – ustawia wartość pola overtimeHours,
- SetOvertimeRate – ustawia wartość pola overtimeRate,
- GetGross – zwraca wartość odziedziczonej metody GetGross zwiększoną o overtimeHours * overtimeRate (w groszach),
- GetDeductions – oblicza potrącenia jako sumę potrąceń z typu bazowego plus 20% z kwoty overtimeHours * overtimeRate,
- Odziedziczona metoda ShowPaymentInfo powinna wyświetlić kwotę potrąceń i kwotę do wypłaty pracownika mającego godziny nadliczbowe.

Korzystając z referencji na typ bazowy zadeklaruj pracowników:

- Jan Kowalski, pesel: 80121349135, z pensją 4200 zł,
- Marian Nowicki, pesel: 72110334254, z pensją 4095 zł, 45.5 godz. nadliczbowymi płatnymi po 42 zł.

Następnie Kowalskiemu podnieś pensję o 133 zł 23 gr. Wyświetl dla obu pracowników kwotę potrąceń i kwotę do wypłaty.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 8.4.

Stwórz klasę Department z polami:

- string name,
- List<Employee> Employees.

i konstruktorem parametrowym pobierającym nazwę departamentu i inicjującym pustą listę pracowników.

Metody do zaimplementowania:

- AddEmployee(Employee e) – dodaje pracownika,
- RemoveEmployee(int idx) – usuwa pracownika o podanym indeksie,
- double GetTotalSalaries() – zwraca sumę wypłat dla działu,
- double GetAverageSalary() – zwraca średnią wypłat,
- Employee GetMinSalaryEmployee() – zwraca pracownika z najmniejszą wypłatą,
- Employee GetMaxSalaryEmployee() – zwraca pracownika z największą wypłatą,
- List<Employee> GetTop4BySalary() – zwraca top 4 wg pensji malejąco,
- List<Employee> FindByLastNamePrefix(string prefix) – zwraca pracowników wyszukanych po początkowym fragmencie nazwiska,
- double GetOvertimeCost() – suma kosztów nadgodzin w dziale,
- List<Employee> GetEmployeesAbove(double x) – lista pracowników, którzy zarabiają powyżej x.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Pytania pomocnicze

1. Wyjaśnij, czym różni się relacja "is-a" od relacji "has-a" i podaj po jednym praktycznym przykładzie zastosowania każdej z nich.
2. Dlaczego konstruktory nie są dziedziczone w C# i w jaki sposób klasa pochodna powinna inicjalizować składowe klasy bazowej.
3. co się stanie gdy stworzysz referencję typu klasy bazowej do obiektu klasy pochodnej i wywołasz metodę przesłoniętą – która wersja metody zostanie wykonana i dlaczego?
4. Jakie problemy mogą wynikać z nieprawidłowego użycia dziedziczenia (na dowolnym przykładzie) i jak można je rozwiązać stosując właściwy mechanizm?

Podsumowanie

Umiejętne wykorzystanie mechanizmów dziedziczenia i polimorfizmu pozwala na szybkie i łatwe tworzenie projektów dzięki wielokrotnemu wykorzystaniu raz stworzonej implementacji oraz zachowanie wysokiej elastyczności dzięki wprowadzaniu modyfikacji tam, gdzie są one potrzebne. Dzięki dziedziczeniu możemy budować logiczne hierarchie klas, gdzie klasy potomne rozszerzają funkcjonalność klas bazowych, unikając powielania kodu. Polimorfizm natomiast umożliwia obiektom różnych klas reagowanie w specyficzny sposób na te same wywołania metod, co znacznie zwiększa elastyczność naszych rozwiązań. Kluczowe jest przy tym poprawne rozróżnienie relacji "is-a" właściwych dla dziedziczenia od relacji "has-a", które realizujemy poprzez agregację. Stosowanie słów kluczowych `virtual`, `override` i `base` pozwala na kontrolowane modyfikowanie zachowań dziedziczonych z klas nadrzędnych. Właściwe zastosowanie tych mechanizmów prowadzi do tworzenia kodu, który jest nie tylko bardziej zwięzły i czytelny, ale również łatwiejszy w utrzymaniu i rozszerzaniu. Opanowanie tych technik stanowi fundament dla budowania zaawansowanych aplikacji obiektowych, przygotowując jednocześnie do studiowania bardziej złożonych wzorców projektowych.

Literatura

- [1] Michaelis M., *C# 8.0: kompletny przewodnik dla praktyków*, Helion, Gliwice 2021. s. 31–97.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

