Programowanie obiektowe

Cel zajęć. Celem zajęć jest zapoznanie z praktycznymi aspektami projektowania oraz implementacji klas i obiektów.

Wprowadzenie teoretyczne. Rozważana w ramach niniejszych zajęć tematyka jest ważna, gdyż tworzenie klas i obiektów jest elementarną częścią programowania obiektowego, a klasy i obiekty są powszechnie używane w aplikacjach desktopowych oraz internetowych. Aby ze zrozumieniem zrealizować zadania, przewidziane do wykonania w ramach zajęć laboratoryjnych, należy znać znaczenie pojęć takich jak: klasy i obiekty, stałe, pola, właściwości, metody. Należy również znać podstawy języka modelowania systemów informatycznych UML.

1. Klasy i obiekty

Programowanie zorientowane obiektowo jest obecnie najbardziej rozpowszechnionym paradygmatem programowania i zastąpiło techniki proceduralne opracowane w latach '70. Program zorientowany obiektowo składa się z obiektów, przyjmujących odpowiedni stan i udostępniających określony zestaw funkcji.

Klasa jest szablonem, z którego tworzy się obiekty. Definiuje ona nowy typ danych. Klasa zawiera zbiór operacji oraz zbiór danych reprezentujących różne, abstrakcyjne wartości, które obiekty tej klasy mogą przyjmować.

Obiekt to egzemplarz klasy zgodny z jej opisem, który jest tworzony w pamięci komputera.

Przykład 1 – Deklaracja klasy i ciała klasy w języku C#

```
1     public class Dom
2     {
3
4         const int ioscOkien = 10; //stała
5         private double metraz = 200; //pole
6         private string adres; //pole
7
8         public Dom() //konstruktor domyślny
9         {
10            adres = "nieznany";
11        }
12
13        public Dom(string adres_) //konstruktor
14        {
15            adres = adres_;
16        }
17
18        public string Adres //właściwość
19        {
20            get { return adres; }
21            set { adres = value; }
22        }
23
24        public double ObliczPodatek(double podatekZaMetr) //metoda
25        {
26            return podatekZaMetr * metraz;
27        }
28    }
```

2. Składowe klasy w języku C#

Stałe, pola i właściwości to składowe klasy, które reprezentują jej faktyczną zawartość lub stan.

Stała to symbol reprezentujący niezmienną wartość. Kompilator wiąże tę wartość z daną klasą, a nie z tym czy innym obiektem klasy. Stałych należy używać dla wartości, które nigdy nie będą zmieniane.
Język C# wykorzystuje słowo kluczowe „const" do deklarowania zmiennych reprezentujących stałą wartość. Stała musi być typu prostego, np. string, double, itd.

Pole to zmienna zawierająca wartość danych. Pól należy używać dla danych w ramach klasy.

Pole różni się od stałej w następujących aspektach: jego wartość jest określona dopiero w momencie wykonywania programu, nie kompilacji, a jego typ nie musi być typem prostym.

Poza modyfikatorami dostępu, pole obsługuje dwa dodatkowe modyfikatory:

- static – pole tak oznaczone jest częścią stanu klasy, nie jej obiektów. Można się do niego odwołać stosując konstrukcję „nazwaKlasy.nazwaPola" bez konieczności tworzenia obiektu danej klasy.

- readonly – wartość pola można przypisać jedynie w wyrażeniu deklaracji lub w konstruktorze klasy. Przekształca on pole w swoistą stałą.

Właściwość obsługuje dostęp do pewnej wartości składowej w ramach danej klasy. Wykorzystuje metodę akcesora, która definiuje kod wykonywany w celu odczytania lub zapisania tej wartości. Kod odczytujący lub zapisujący wartość właściwości jest niejawnie implementowany przez środowisko .NET w formie dwóch osobnych metod. Właściwości powinny być wykorzystywane do kontroli dostępu do danych składowanych wewnątrz klasy.

Przykład 2 – Składnia właściwości

```
1.  <modfikator dostepu> <typ danych> <nazwa właściwości>
2.  {
3.      get
4.      {
5.          //...
6.          return [nazwa pola]; //zwrócenie wartości pola
7.      }
8.      set
9.      {
10.         //...
11.         [nazwa pola] = value; //przypisanie wartości value do pola
12.     }
13. }
```

Przykład 3 – Wykorzystanie właściwości dla obiektu klasy „Dom" z przykładu 1

```
1.      Dom d = new Dom();
2.      d.Adres = "ul. Testowa 1";
3.      Console.WriteLine(d.Adres);
4.      Console.ReadKey();
```

Metody odpowiadają za wykonywanie działań, które definiuje zachowanie klas. Metoda musi składać się z: modyfikatora dostępu, nazwy zwracanego typu, nazwy metody oraz listy przyjmowanych parametrów (uwzględniając ich typ i nazwę, lista może być też pusta). Metoda musi także zwracać wartość zgodną z określonym wcześniej zwracanym typem. Odbywa się to za pomocą wyrażenia „return wartość".

Przykład 4 – Wywołanie metody dla obiektu klasy „Dom" z przykładu 1

```
1.    Dom d = new Dom();
2.    double podatek = d.ObliczPodatek(20.5);
```

Poza modyfikatorami dostępu metody mogą zawierać także inne modyfikatory, które w większości zostaną omówione na kolejnych zajęciach.

Jednym z tych modyfikatorów jest modyfikator:

- static – metoda tak oznaczona jest częścią stanu klasy, a nie któregokolwiek z jej obiektów. Można się do niej odwołać stosując konstrukcję „nazwaKlasy.nazwaMetody" bez konieczności tworzenia obiektu danej klasy.

Konstruktor jest szczególną metodą klasy wywoływaną podczas tworzenia obiektu tej klasy. Każdy konstruktor musi mieć taką samą nazwę jak klasa, w której się znajduje. Konstruktor nie może zwracać żadnej wartości. Klasa może zawierać wiele konstruktorów, muszą one jednak różnić się listą przyjmowanych parametrów. Konstruktor bezparametrowy nazywany jest konstruktorem domyślnym.

Przykład 5 – Tworzenie obiektów klasy „Dom" z przykładu 1

```
1.    Dom d = new Dom();
2.    Dom d2 = new Dom("ul. Testowa 1");
```

UWAGA! W języku C# nie istnieje pojęcie **destruktora**. Usuwaniem z pamięci komputera nieużywanych obiektów zajmuje się mechanizm zwany Garbage Collector, będący składową środowiska .NET. Usuwanie nieużywanych obiektów z pamięci komputera przeprowadzane jest samoczynnie, bez ingerencji programisty. Nie można też określić, w którym momencie obiekt zostanie zwolniony z pamięci.

3. Modyfikatory dostępu

Służą do określania dostępności (zasięgu, widoczności) klas i ich składowych.

W języku C# istnieją następujące modyfikatory dostępu, które mają określone działanie dla składowych klas oraz samych klas:

- public – Składowa klasy jest dostępna dla dowolnej klasy. Klasa jest dostępna z poziomu każdego zestawu .NET.

- protected - Składowa klasy jest dostępna jedynie w obrębie klasy, w której się znajduje oraz w obrębie klas dziedziczących. W przypadku klas ma zastosowanie jednie dla klas zagnieżdżonych. Klasa jest dostępna jedynie dla klasy zawierającej lub pochodnych klasy zawierającej.

- internal – Składowa klasy jest dostępna jedynie w obrębie klasy, w której się znajduje oraz klasy w obrębie zestawu .NET. Klasa jest dostępna jedynie dla klas należących do tego samego zestawu.

- private – Składowa klasy jest dostępna jedynie w obrębie klasy, w której się znajduje. W przypadku klas ma zastosowanie jednie dla klas zagnieżdżonych. Klasa jest dostępna jedynie dla klasy zawierającej.

- protected internal - Składowa klasy jest dostępna jedynie w obrębie klasy, w której się znajduje, klasy w obrębie zestawu .NET oraz klas dziedziczących. Dostęp do klasy ogranicza się do bieżącego zestawu oraz typów potomnych względem klasy zawierającej.

Domyślnym modyfikatorem dostępu dla klas w języku C# jest „internal". Dla składowych klas domyślnym modyfikatorem jest „private".

„Zestaw .NET" należy rozumieć jako tę samą jednostkę kompilacji, np. jedną bibliotekę DLL.

4. Podstawowa struktura programu C#

Język C# należy do języków czysto obiektowych. Wymaga to, aby wszystkie komendy wykonywane przez program zostały zawarte w definicji pewnej klasy. W przeciwieństwie do np. języka C++ nie jest możliwe tworzenie globalnych zmiennych oraz funkcji.

Najprostsza postać jaką może przyjmować program w języku C#, tworzona jest przez Visual Studio jako szablon aplikacji konsolowej.

Przykład 6 – Szablon aplikacji konsolowej

```
1.    using System;
2.    using System.Collections.Generic;
3.    using System.Linq;
4.    using System.Text;
5.
6.    namespace ConsoleApplication1
7.    {
8.        class Program
9.        {
10.           static void Main(string[] args)
11.           {
12.           }
13.       }
14.   }
```

W liniach 1-4 powyższego kodu znajduje się instrukcja „using", wykorzystywana do dołączenia przestrzeni nazw, z których będziemy korzystać w dalszej części kodu.

W linii 6 zdefiniowana została przestrzeń nazw, do której należy klasa „Program" będąca główną klasą aplikacji.

Każda aplikacja powinna zawierać statyczną metodę „Main", która stanowi swoisty punkt wejściowy aplikacji. Po uruchomieniu aplikacji zostaną wykonane operacje ujęte wewnątrz metody „Main". Poniższy przykład prezentuje aplikację tworzącą obiekt typu „Dom" i wypisującą na ekranie konsoli adres domu.

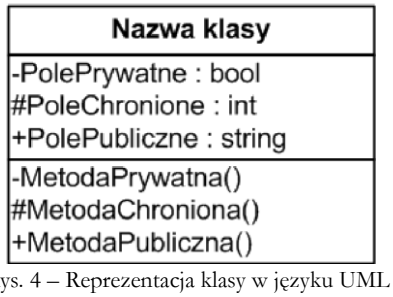
Przykład 7 – Prosta aplikacja konsolowa

```
1.    using System;
2.    using System.Collections.Generic;
3.    using System.Linq;
4.    using System.Text;
5.
6.    namespace ConsoleApplication1
7.    {
8.        public class Dom
9.        {
10.           private string adres;
11.
12.           public Dom(string adres_)
13.           {
14.               adres = adres_;
15.           }
16.
17.           public string Adres
18.           {
19.               get { return adres; }
20.               set { adres = value; }
21.           }
22.       }
23.
24.       class Program
25.       {
26.           static void Main(string[] args)
27.           {
28.               Dom d = new Dom("ul. Przykładowa 2");
29.               string adresD = d.Adres;
30.               Console.WriteLine(adresD);
31.               Console.ReadKey();
32.           }
33.       }
34.   }
```

5. Język modelowania UML – diagram klas

UML (Unified Modeling Language) jest językiem modelowania systemów informatycznych. Pozwala on na graficzne przedstawienie – w postaci diagramów – projektowanego systemu, w celu sprawdzenia czy nasze wyobrażenie o nim odpowiada wyobrażeniu innych osób. Każdy z nich prezentuje relacje zachodzące pomiędzy różnymi elementami tworzonego oprogramowania. Choć szczegółowe przedstawienie składni tego języka wykracza poza zakres tego przedmiotu, pewne jego elementy konieczne do zajęć zostaną wprowadzone. Na zajęciach skupimy się głównie na diagramie klas.

Diagram klas jest podstawowym diagramem, który umożliwia przedstawienie klas i opisanie relacji zachodzących pomiędzy nimi. Klasa przedstawiana jest w postaci prostokąta podzielonego na trzy części (rys. 4). Pierwsza z nich zawiera nazwę klasy, druga pola klasy, a trzecia metody.



Rys. 4 – Reprezentacja klasy w języku UML

Modyfikatory dostępu są zaprezentowane na rysunku za pomocą znaków:

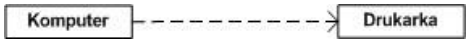
- „-” – private
- „#” – protected
- „+” – public

Czasem dla zwiększenia czytelności diagramów klasy przedstawiane są w uproszczony sposób, np.:

Rys. 5 – Uproszczone reprezentacje klas w języku UML

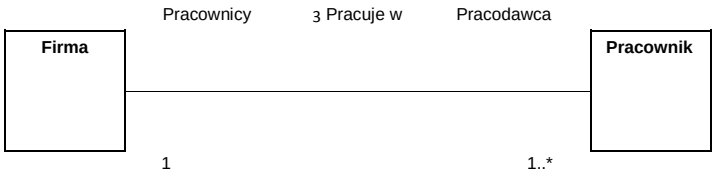
Relacje zachodzące pomiędzy klasami:

- **Zależność** – jest najsłabszym rodzajem powiązania między klasami. Zależność pomiędzy klasami oznacza, że jedna klasa używa drugiej klasy lub posiada dotyczące niej informacje. Związek ten jest zazwyczaj krótkotrwały. Zależność występuje, kiedy zmiana specyfikacji jednej z klas może powodować konieczność wprowadzania zmian w innej klasie. Zależności używa się najczęściej do pokazania, że jedna klasa używa innej jako parametru jakiejś operacji.



Rys. 6 – Graficzna reprezentacja relacji zależności

- **Asocjacja** – jest silniejszym typem powiązań niż zależność. Oznacza, że jeden obiekt jest związany z innym przez pewien okres czasu, jednak czas życia obiektów nie jest od siebie zależny (usunięcie jednego nie powoduje usunięcia drugiego).



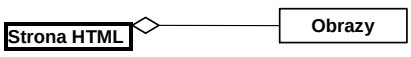
Rys. 7 – Graficzna reprezentacja relacji asocjacji

Asocjacja może zawierać nazwę, wraz z kierunkiem asocjacji, role (opcjonalne nazwy zadań klas w ramach zależności) oraz specyfikację liczebności (liczba obiektów z poszczególnych klas, mogących kojarzyć się z obiektami innej klasy).
Liczebność może być reprezentowana przez liczby całkowite lub przez przedziały:

- 0..1 – zero lub jeden obiekt
- 0..* - od zera do nieograniczonej liczby obiektów (często reprezentowane również w postaci „*”)
- n ..* - od n do nieograniczonej liczby obiektów
- m..n – liczba obiektów z przedziału od m do n

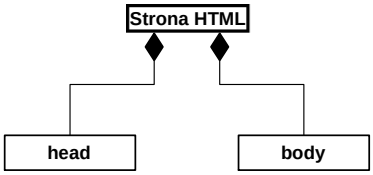
Należy pamiętać, że za n i m można również podstawić liczby całkowite.

- **Agregacja** – agregacja jest silniejszą formą asocjacji. Stanowi relację typu całość-część, w której część może należeć do kilku całości, a całość nie zarządza czasem istnienia części (usunięcie całości nie powoduje usunięcia części).



Rys. 8 – Graficzna reprezentacja relacji asocjacji

- **Kompozycja (agregacja całkowita)** – kompozycja jest najsilniejszym rodzajem powiązania między klasami. Stanowi relację całość-część, w której części są tworzone i zarządzane przez obiekt reprezentujący całość. „Część" może być zaangażowana tylko w jeden związek tego typu w danym czasie. Czasy życia instancji biorących udział w takim związku są ze sobą powiązane (usunięcie całości powoduje usunięcie części).



Rys. 9 – Graficzna reprezentacja relacji kompozycji

Zadanie 1. Proszę zrealizować aplikację obiektową, która powinna odznaczać się następującymi cechami:

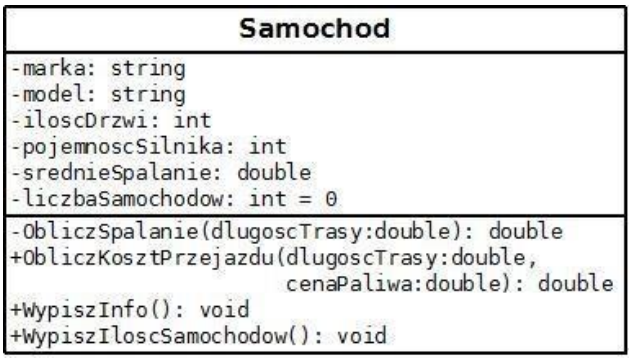
- Aplikacja powinna zawierać klasę „Samochod", umożliwiającą przechowywanie określonych informacji oraz wykonywanie określonych działań.
- Klasa „Samochod" ma umożliwiać przechowywanie informacji na temat samochodu: marki, modelu, ilości drzwi, pojemności silnika, średniego spalania na 100 km.
- Klasa „Samochod" ma posiadać właściwości dostępowe do wszystkich pól, które nie są statyczne.
- Klasa „Samochod" ma umożliwiać przechowywanie informacji na temat liczby utworzonych obiektów tej klasy (poprzez pole statyczne).
- Klasa „Samochod" ma zawierać konstruktor domyślny oraz konstruktor przyjmujący parametry.
- Klasa „Samochod" ma umożliwiać obliczanie średniego spalania na danej trasie na podstawie podanej długości tej trasy.
- Klasa „Samochod" ma umożliwiać obliczanie ceny paliwa potrzebnego do przejechania danej trasy na podstawie podanej długości trasy i ceny paliwa za litr.

Klasa „Samochod" ma umożliwiać wypisanie na ekran konsoli wszystkich informacji o danym samochodzie (wartości pól, które nie są statyczne).

- Klasa „Samochod" ma umożliwiać wypisanie na ekran konsoli informacji o liczbie utworzonych obiektów (poprzez metodę statyczną).

- Uwagi:
  - Do wypisywania treści na ekranie konsoli służy polecenie: Console.WriteLine([treść])
  - Po wykonaniu zadania należy je przetestować za pomocą kodu testowego umieszczonego na końcu opisu realizacji zadania.

- Reprezentacja klasy na diagramie UML:



Aby zrealizować zadanie należy wykonać następujące kroki:

- Proszę o utworzenie nowego projektu konsolowego w środowisku Visual Studio 2017.

- Proszę o utworzenie klasy „Samochod".

- Proszę o utworzenie prywatnych pól klasy „Samochod" o nazwie: „marka", „model", „iloscDrzwi", „pojemnoscSilnika", „srednieSpalanie". Proszę zwrócić uwagę na odpowiedni dobór typów pól. Np.:

```
private double srednieSpalanie;
```

- Proszę o utworzenie właściwości dostępowych do wszystkich pól (oprócz pola statycznego). Np.:

```
public double SrednieSpalanie
{
        get { return srednieSpalanie; }
        set { srednieSpalanie = value; }
}
```

- Proszę o utworzenie w klasie „Samochod" prywatnego statycznego pola typu *int* o nazwie „iloscSamochodow" i przypisanie mu wartości 0:

```
private static int iloscSamochodow = 0;
```

- Proszę o utworzenie konstruktora domyślnego, który wszystkim polom przyporządkowuje wartości: „nieznana" lub „nieznany" dla pól typu *string*, „0" dla pól typu *int*, „0.0" dla pól typu *double*. Wywołanie konstruktora powinno zwiększać o 1 wartość pola statycznego „iloscSamochodow". Proszę pamiętać o tym, że konstruktor domyślny nie przyjmuje żadnych parametrów.

- Proszę o utworzenie konstruktora, przyjmującego następujące parametry: „marka_", „model_", „iloscDrzwi_", „pojemnoscSilnika_", „srednieSpalanie_". Typy parametrów powinny odpowiadać typom pól klasy „Samochod". Konstruktor ma przekazywać wartości parametrów polom. Wywołanie konstruktora powinno zwiększać o 1 wartość pola statycznego „iloscSamochodow".

- Proszę o utworzenie prywatnej metody „ObliczSpalanie" zwracającej wartość typu *double* i przyjmującej parametr „dlugoscTrasy" typu *double*. Metoda ta ma obliczać spalanie samochodu na podstawie podanej wartości długości trasy i wartości pola „srednieSpalanie". Spalanie obliczamy według wzoru:

  spalanie = (srednieSpalanie * dlugoscTrasy) / 100.0;

- Proszę o utworzenie publicznej metody „ObliczKosztPrzejazdu" zwracającej wartość typu *double* i przyjmującej parametry „dlugoscTrasy" typu *double* i „cenaPaliwa" typu *double*. Metoda ta ma obliczać koszt przejazdu na trasie o podanej długości, zakładając podaną cenę paliwa za litr. Metoda ta ma wykorzystywać prywatną metodę „ObliczSpalanie". Koszt przejazdu obliczamy według wzoru:

  kosztPrzejazdu = spalanie * cenaPaliwa;

- Proszę o utworzenie publicznej metody „WypiszInfo" zwracającej wartość typu *void* i nie przyjmującej żadnych parametrów. Metoda ta ma wypisywać na ekranie konsoli wartości wszystkich pól klasy „Samochod". Np.:

  ```
  Console.WriteLine("Marka: " + marka);
  ```

- Proszę o utworzenie publicznej statycznej metody „WypiszIloscSamochodow" zwracającej wartość typu *void* i nie przyjmującej żadnych parametrów. Metoda ta ma wypisywać na ekranie konsoli wartość pola statycznego „iloscSamochodow".

- Proszę o przetestowanie poprawności wykonania zadania za pomocą kodu testowego:

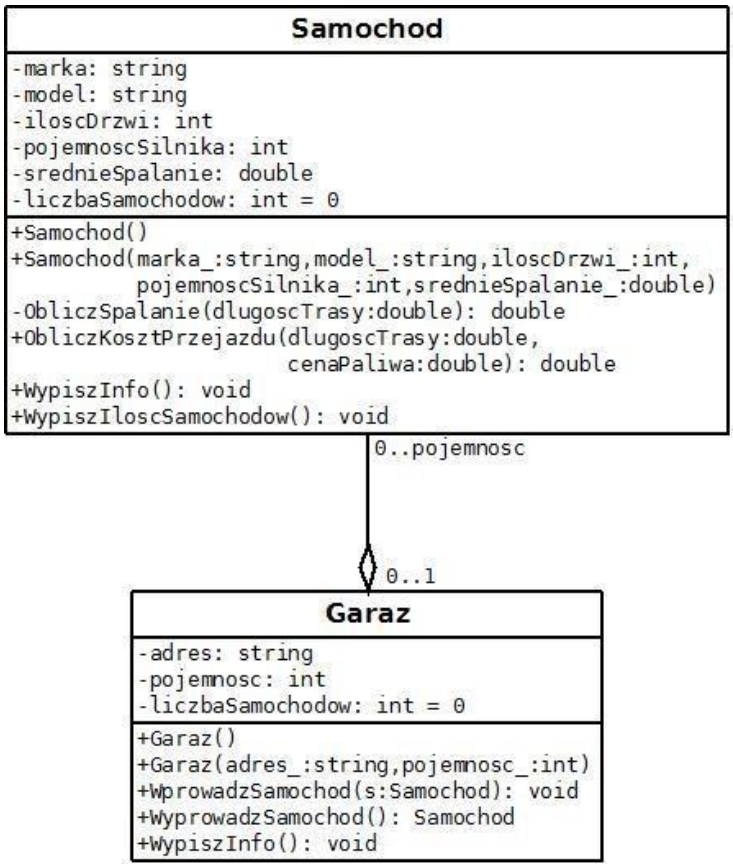
  ```
  Samochod s1 = new Samochod();

  s1.WypiszInfo();

  s1.Marka = "Fiat";
  s1.Model = "126p";
  s1.IloscDrzwi = 2;
  s1.PojemnoscSilnika = 650;
  s1.SrednieSpalanie = 6.0;

  s1.WypiszInfo();

  Samochod s2 = new Samochod("Syrena", "105", 2, 800, 7.6);
  ```

```
s2.WypiszInfo();

double kosztPrzejazdu = s2.ObliczKosztPrzejazdu(30.5, 4.85);
Console.WriteLine("Koszt przejazdu: " + kosztPrzejazdu);
Samochod.WypiszIloscSamochodow();

Console.ReadKey();
```

Zadanie 2. Proszę zrealizować aplikację obiektową, która powinna odznaczać się następującymi cechami:

- Do wykonania zadania 2 konieczne jest uprzednie wykonanie zadania 1.
- Aplikacja z zadania 1 ma zostać uzupełniona o klasę „Garaz".
- Klasa „Garaz" ma przechowywać następujące informacje dotyczące garażu: adres, pojemność, liczba garażowanych samochodów, informacje dotyczące garażowanych samochodów.
- Klasa „Garaz" ma umożliwiać dodawanie do kolekcji przechowywanych samochodów nowego samochodu oraz wyprowadzanie z tej kolekcji ostatnio wprowadzonego samochodu.
- Kolekcja samochodów ma zostać zrealizowana za pomocą dynamicznej tablicy.
- Uwagi:
  - Dynamiczną tablicę tworzymy według schematu:

    <typ> [ ] nazwaTablicy = new <typ> [rozmiar];
- Reprezentacja klas na diagramie UML:


Samochod
-marka: string
-model: string
-iloscDrzwi: int
-pojemnoscSilnika: int
-srednieSpalanie: double
-liczbaSamochodow: int = 0
+Samochod()
+Samochod(marka_:string,model_:string,iloscDrzwi_:int,
pojemnoscSilnika_:int,srednieSpalanie_:double)
-ObliczSpalanie(dlugoscTrasy:double): double
+ObliczKosztPrzejazdu(dlugoscTrasy:double,
cenaPaliwa:double): double
+WypiszInfo(): void
+WypiszIloscSamochodow(): void
0..pojemnosc
0..1
Garaz
-adres: string
-pojemnosc: int
-liczbaSamochodow: int = 0
+Garaz()
+Garaz(adres_:string,pojemnosc_:int)
+WprowadzSamochod(s:Samochod): void
+WyprowadzSamochod(): Samochod
+WypiszInfo(): void

Aby zrealizować zadanie należy wykonać następujące kroki:

- Proszę o utworzenie klasy „Garaz".

- Proszę o utworzenie prywatnych pól klasy „Garaz" o nazwie: „adres", „pojemnosc", „liczbaSamochodow", „samochody". Pole „samochody" powinno posiadać typ tablicowy, przechowujący obiekty klasy „Samochod":

```
private Samochod[] samochody;
```

  Pole „liczbaSamochodow" powinno być zainicjowane wartością „0":

```
private int liczbaSamochodow = 0;
```

- Proszę o utworzenie właściwości dostępowych do pól „adres" i „pojemnosc". Właściwość „Set" pola „pojemność" powinna przydzielać także pamięć polu „samochody":

```
set
{
    pojemnosc = value;
    samochody = new Samochod[pojemnosc];
}
```

- Proszę o utworzenie konstruktora domyślnego, który wszystkim polom przyporządkowuje wartości: „nieznana" lub „nieznany" dla pól typu *string*, „0" dla pól typu *int*, „null" dla pól typu tablicowego. Proszę pamiętać o tym, że konstruktor domyślny nie przyjmuje żadnych parametrów. Proszę zauważyć, że pole „liczbaSamochodow" zostało już zainicjowane i nie jest konieczne przypisanie mu wartości w ciele konstruktora.

- Proszę o utworzenie konstruktora, przyjmującego następujące parametry: „adres_", „pojemnosc_". Typy parametrów powinny odpowiadać typom pól klasy „Garaz". Konstruktor ma przekazywać wartości parametrów polom. Wywołanie konstruktora powinno przydzielić pamięć polu „samochody", tworząc tablicę o rozmiarze odpowiadającym wartości pola „pojemnosc".

- Proszę o utworzenie publicznej metody „WprowadzSamochod", zwracającej wartość typu *void* i przyjmującej parametr typu *Samochod*. Metoda ta ma sprawdzać, czy garaż jest zapełniony. Jeśli tak, ma wypisywać na ekranie konsoli odpowiedni komunikat. Jeśli nie jest zapełniony, ma wprowadzić do niego dany samochód. Nowy samochód powinien być dodany do tablicy za znajdującymi się w niej obiektami typu *Samochod*. Do określenia tej pozycji ma służyć wartość pola „liczbaSamochodow", która po wprowadzeniu nowego samochodu powinna być odpowiednio zmodyfikowana.

- Proszę o utworzenie publicznej metody „WyprowadzSamochod", zwracającej wartość typu *Samochod* i nie przyjmującej żadnych parametrów. Metoda ta ma sprawdzać, czy garaż jest pusty. Jeśli tak, ma wypisywać na ekranie konsoli odpowiedni komunikat. Jeśli nie jest pusty, ma wyprowadzić z niego ostatnio wprowadzony samochód. Do określenia pozycji samochodu w tablicy ma służyć wartość pola „liczbaSamochodow", która po wyprowadzeniu nowego samochodu powinna być odpowiednio zmodyfikowana. Po wyłuskaniu obiektu typu *Samochod* z tablicy, do pozycji na której się znajdował należy podstawić wartość „null".

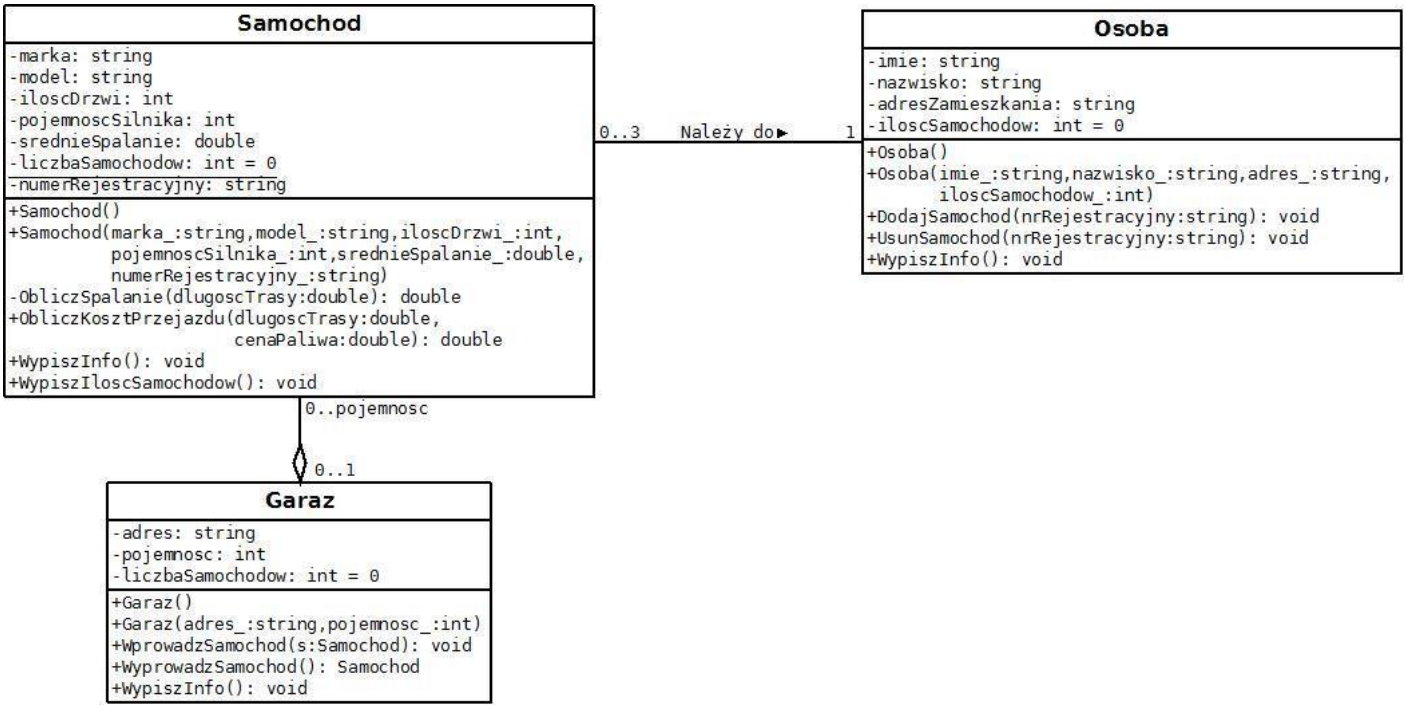
- Proszę o utworzenie publicznej metody „WypiszInfo" zwracającej wartość typu *void* i nie przyjmującej żadnych parametrów. Metoda ta ma wypisywać na ekranie konsoli wartości wszystkich pól typu prostego klasy „Samochod". Dodatkowo ma wyświetlać wszystkie informacje dotyczące garażowanych samochodów. Realizacja wyświetlania informacji o samochodach ma opierać się na zastosowaniu pętli „for" oraz metodzie „WypiszInfo" zaimplementowanej w klasie „Samochod".

- Proszę o przetestowanie poprawności wykonania zadania za pomocą kodu testowego:

```
Samochod s1 = new Samochod("Fiat", "126p", 2, 650, 6.0);
Samochod s2 = new Samochod("Syrena", "105", 2, 800, 7.6);

Garaz g1 = new Garaz();
g1.Adres = "ul. Garażowa 1";
g1.Pojemnosc = 1;

Garaz g2 = new Garaz("ul. Garażowa 2", 2);
g1.WprowadzSamochod(s1);
g1.WypiszInfo();
g1.WprowadzSamochod(s2);

g2.WprowadzSamochod(s2);
g2.WprowadzSamochod(s1); g2.WypiszInfo();

g2.WyprowadzSamochod();
g2.WypiszInfo();
g2.WyprowadzSamochod();
g2.WyprowadzSamochod();

Console.ReadKey();
```

Zadanie do domu. Proszę zrealizować aplikację obiektową, która powinna odznaczać się następującymi cechami:

- Do wykonania zadania 2 konieczne jest uprzednie wykonanie poprzednich zadań.
- Aplikacja powinna być zgodna z przedstawionym poniżej diagramem klas.
- Klasa „Samochod" została rozszerzona o pole „numerRejestracyjny".
- Obiekty klasy „Osoba" mają przechowywać informacje o posiadanych samochodach w formie ich numerów rejestracyjnych (numery te można przechowywać w tablicy). Proszę zauważyć, że każda osoba może posiadać maksymalnie 3 samochody.
- Metoda „DodajSamochod" ma dodawać numer rejestracyjny do kolekcji numerów rejestracyjnych (tabeli) w obiekcie typu „Osoba". Przy wywołaniu tej metody należy pamiętać o sprawdzeniu czy osoba może posiadać kolejny samochód oraz o zwiększeniu licznika posiadanych samochodów po dodaniu numeru rejestracyjnego.
- Metoda „UsunSamochod" ma usuwać numer rejestracyjny podany jako parametr z kolekcji numerów rejestracyjnych (tabeli) w obiekcie typu „Osoba". Operację tę można zaimplementować z wykorzystaniem pętli „for". Usuwanie ma polegać na przyporządkowaniu danej pozycji w kolekcji wartości „null". Przy wywołaniu tej metody należy pamiętać o zmniejszeniu licznika posiadanych samochodów po usunięciu numeru rejestracyjnego
- Metoda „WypiszInfo" ma wypisywać informacje o osobie oraz numery rejestracyjne posiadanych przez nią samochodów.



Zagadnienia, które należy uznać za przyswojone w trakcie zajęć. Po zajęciach będzie obowiązywać praktyczna znajomość:

- Pojęcia klasy i obiektu.
- Pojęcia pola, metody, właściwości, konstruktora, konstruktora domyślnego.
- Rodzaje modyfikatorów dostępu.
- Utworzenie aplikacji konsolowej w środowisku Visual Studio.
- Znajomość struktury aplikacji w języku C#.
- Pojęcia UML, diagram klas.
- Znajomość związków wykorzystywanych na diagramach klas - zależności, asocjacji, agregacji, kompozycji.
- Znajomość liczebności wykorzystywanych na diagramach klas.