Typy generyczne i zasady dobrego programowania.

C#

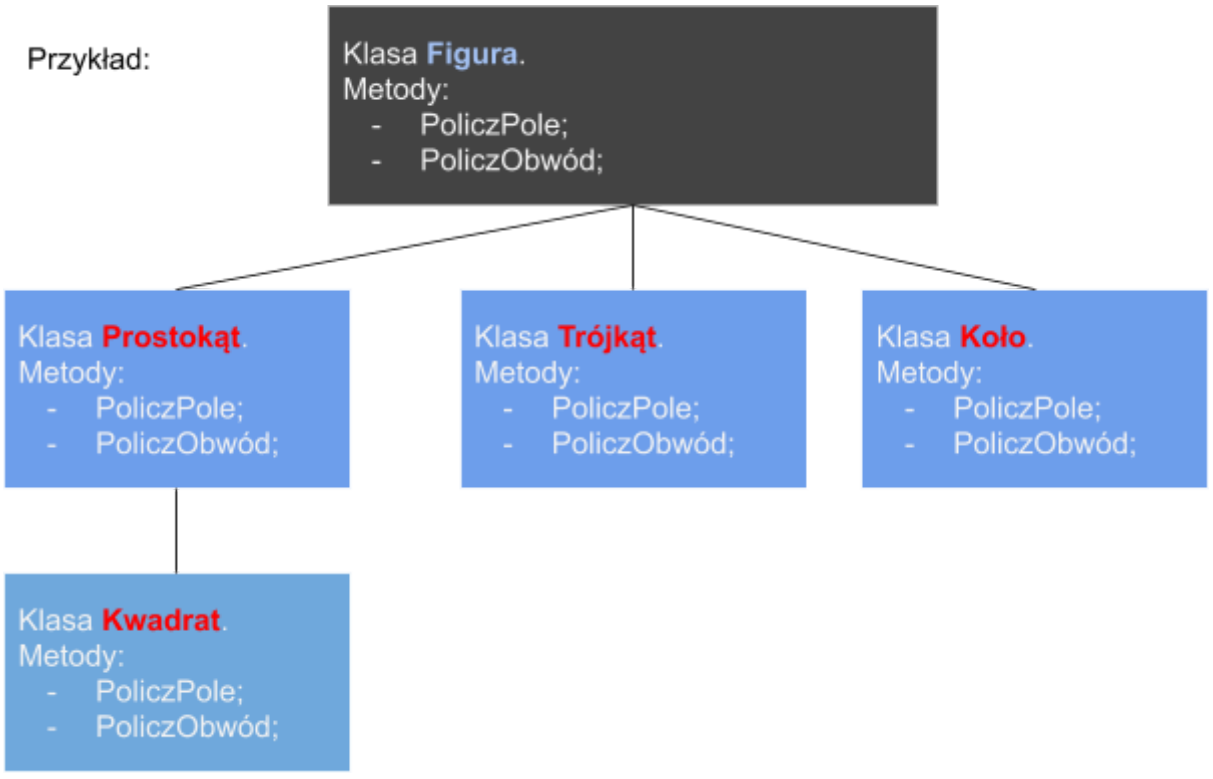
Wydział Matematyki i Informatyki Stosowanej
Ćwiczenia: Michał Mnich

Projektowanie z użyciem dziedziczenia

ZASADA PODSTAWIENIA LISKOV

Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dok ładnej znajomości tych obiektów

Inaczej mówiąc, klasa dziedzicząca powinna tylko rozszerzać możliwości klasy bazowej i w pewnym sensie nie zmieniać tego, co ona robiła już wcześniej. Mówiąc jeszcze inaczej — jeśli będziemy tworzyć egzemplarz klasy potomnej, to niezależnie od tego, co znajdzie się we wskaźniku na zmienną, wywoływanie metody, którą pierwotnie zdefiniowano w klasie bazowej, powinno dać te same rezultaty.



Jak w przykładzie powyżej Mamy zdefiniowaną klasę bazową(abstrakcyjne) Figura, która definiuje dwie wirtualne metody obliczeniowe. Każda klasa definiując metody musi się powinna trzymać się zdefiniowanych przez klasę figura założeń co do zwracanych i przyjmowanych wartości w metodach “nadpisując” ich implementację.

Przyjrzyjmy się klasycznemu przykładowi łamania zasady Liskov (źródło: http://www.oodesign.com). Załóżmy, że użytkownik napisał metodę:

```
public void Method(Rectangle r)
{
    r.Width = 5;
    r.Height = 10;
    int area = r.GetArea();
}
```

Naturalnym wydaje się założenie, że area będzie równe 50. W przypadku jednak, gdy programista napisał kod łamiący zasadę Liskov jak na przykład poniżej:

```
class Rectangle
{
    public virtual int Height {  get; set; }
    public virtual int Width  { get; set; }

    public int GetArea()
    {
        return Width * Height;
    }
}

class Square : Rectangle
{
    public override int Height
    {
        get => base.Height;
        set => base.Height = base.Width = value;
    }

    public override int Width
    {
        get => base.Width;
        set => base.Width = base.Height = value;
    }
}
```

Wynik może być inny. W tym konkretnym przypadku area będzie równe 100. Dodając więc nowe klasy do hierarchii należy pamiętać o tym, że rozszerzać funkcjonalność a nie modyfikować już istniejącą.

ZADANIE 3.1

Jak można przeprojektować powyższe klasy tak, aby nie naruszały LSP?

Kompozycja kontra dziedziczenie

Po poznaniu wszystkich technik związanych z dziedziczeniem i polimorfizmem można zacząć je stosować w miejscach do tego nie nadających się.

Klasycznym przykładem obrazującym problem jest poniższy kod:

```
class Queue : ArrayList
{
    public void Enqueue(Object value) { }
    public Object Dequeue() { }
}
```

Programista może uzupełnić metody i sprawić, że klasa będzie mogła działać jako kolejka, ale interfejs takiej klasy będzie zaśmiecony niepotrzebnymi metodami.

Ponadto warto zauważyć również następujące problemy:

- Na pierwszych ćwiczeniach mówiliśmy, że dziedziczenie wyraża relację bycia czymś a kolejka nie jest ArrayListą.
- Łamiemy enkapsulację – ludzie z zewnątrz mają łatwy dostęp do ArrayListy.
- Zamykamy się na rozszerzenie – dużo trudniej będzie podmienić implementację ArrayListy na jakąś inną kolekcję.

Więcej na ten temat można poczytać na przykład tutaj:

https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

ZADANIE 3.2

Proszę dokończyć implementację kolejki powyżej. Potem proszę stworzyć drugą implementację tak, aby kolejka działała na ArrayList przez kompozycję a nie dziedziczenie.

ZADANIE 3.3

Co należy zrobić w obydwu przypadkach, że kolejka działała nie na ArrayList tylko na tablicy?

Typy generyczne

Używanie ArrayList, to z wielu powodów nie jest to najlepszy wybór, jeżeli chodzi o kolekcję.

The ArrayList class is designed to hold heterogeneous collections of objects. However, it does not always offer the best performance. Instead, we recommend the following:

- *For a heterogeneous collection of objects, use the **List<Object>** (in C#) or **List(Of Object)** (in Visual Basic) type.*
- *For a homogeneous collection of objects, use the List<T> class.*

ZADANIE 3.4

Dlaczego użycie klasy ArrayList może powodować problemy wydajnościowe w języku C#?

Składnia uogólnienia klas:

```
public class Collection<T>
```

Składnia uogólnienia metod:

```
public void Metoda<T>()
```

Oczywiście może być więcej niż jeden typ generyczny. Typy generyczne mogą być używane do określania typów argumentów funkcji oraz typu zwracanego. Mogą również posłużyć jako dalsze parametry generyczne.

Typy ogólne współpracują również z interfejsami:

```
public interface Collection<T>
```

ZADANIE 3.5

Proszę stworzyć klasę liczby zespolonej `Complex` przechowującą składowe dowolnego typu. Proszę dodać metody zwracające część rzeczywistą i część urojoną.

Ramy typów ogólnych

Na typy konkretyzujące uogólnienie można nałożyć obostrzenia. Możemy zażądać, żeby typ konkretyzujący implementował interfejs albo dziedziczył po jakiejś klasie. W ten sposób uzyskujemy możliwość wywoływania metod dla typów mieszczących się w tych ramach.

```
public interface Collection<T> where T : List<int>, IComparable
```

ZADANIE 3.6

Proszę utworzyć klasę macierzy(`Matrix`) pozwalającą na elementy dowolnego sensownego typu(np. `double`, `float`, `int`). Niech po klasie macierzy dziedziczy klasa macierzy kwadratowej. Niech zwykła macierz ma możliwość dodawania i mnożenia oraz dostępu do wybranego elementu. Niech macierz kwadratowa ma dodatkowo możliwość sprawdzenia czy jest macierzą diagonalną. Proszę zadbać o design klas tak, aby nie naruszał zasad z poprzednich ćwiczeń(np. zasady podstawieniowej Liskov).

ZADANIE 3.7

Proszę zmodyfikować powyższe rozwiązanie tak, aby można było przekazać do macierzy obiekt klasy `Complex`.