Typy generyczne i zasady dobrego programowania.

C#

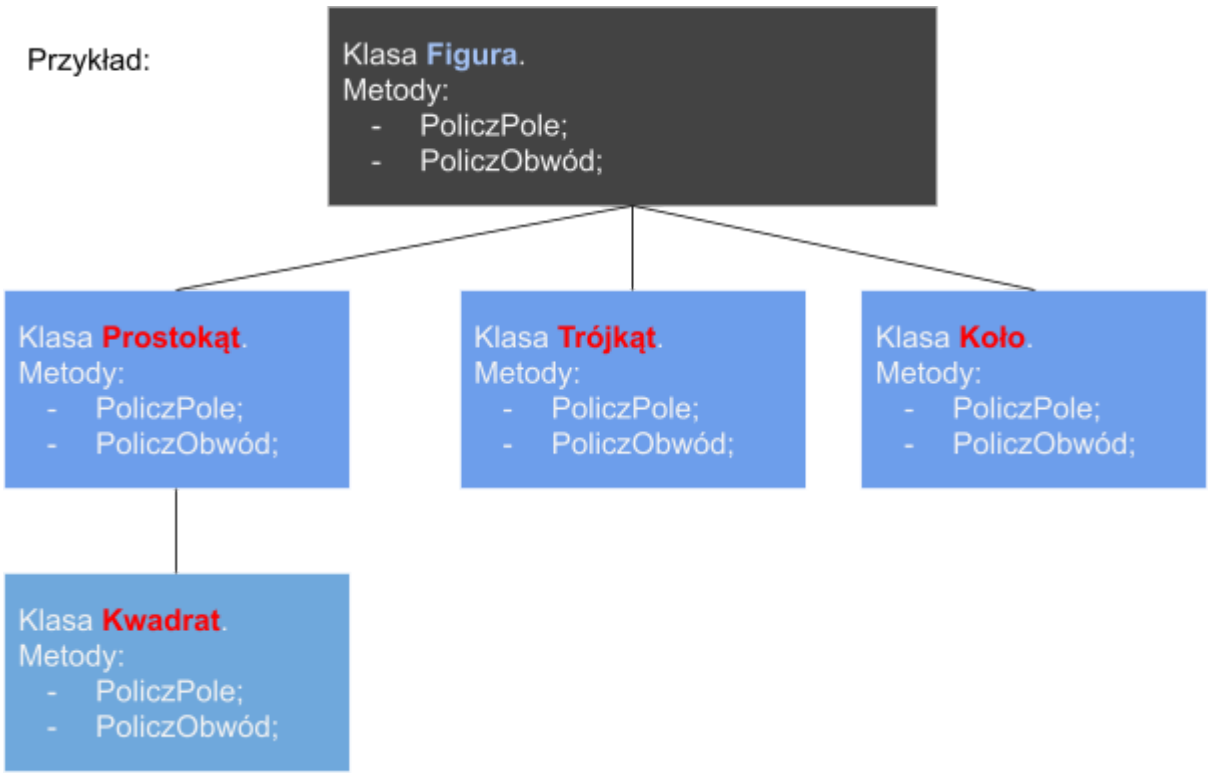
Wydział Matematyki i Informatyki Stosowanej
Ćwiczenia: Michał Mnich

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 dokładnej 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 łamiąc 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

Typy generyczne pozwalają na opóźnienie w dostarczeniu specyfikacji typu danych w elementach takich jak klasy czy metody do momentu użycia ich w trakcie wykonywania programu. Innymi słowy, typy generyczne pozwalają na napisanie klasy lub metody, która może działać z każdym typem danych.
W przypadku typów generycznych w czasie kompilacji każde wykorzystanie typu generycznego tworzy odpowiednią implementację z każdym konkretnym typem, na którym taki typ operuje.

Deklarowanie Typów Generycznych

Przy deklaracji klasy wszystkie jej elementy muszą mieć sprecyzowany typ, aby móc go nie określać przy deklaracji. Konieczne jest nadanie mu jakiegoś aliasu, który to w deklaracji klasy będzie wykorzystywany zamiast typu. Alias ten jest zapisywany w nawiasach trójkątnych <>.

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>
```

Ograniczenia

Różnorodność typów obsługiwanych przez nasz typ generyczny może być ograniczona. W tym celu po deklaracji naszej klasy wykorzystujemy słowo kluczowe where aliasTypu :

class – typ musi być typem referencyjnym
struct – typ musi być typem wartości
new() – typ musi posiadać publiczny konstruktor bez parametrów. To ograniczenie musi być stosowane na samym końcu listy ograniczeń
nazwaTypuBazowego lub NazwaInterfejsu – parametr musi dziedziczyć po klasie bazowej lub implementować interfejs
Przykładowa implementacja może wyglądać tak

Przykłady Kodu.

Klasa wykorzystująca typy generyczne.

```
using System;
namespace Generics
{
   // Definicja klasy generycznej, w tym przypadku własnej tablicy
   class MyGenericArray<T>
   {
      // definicja typowanej tablicy
      private int[] array;
      // definicja generycznej tablicy
      private T[] genericArray;
      // konstruktor klasy przyjmujący jako parametr rozmiar tablicy
      public MyGenericArray(int size)
      {
         // ustalenie rozmiaru zwykłej tablicy
         array = new int[size + 1];
         // ustalenie rozmiaru tablicy generycznej
         genericArray = new T[size + 1];
      }
      public int getItem(int index)
      {
         return array[index];
      }
      // Powyżej metoda zwracająca typ danych jako int
      // Poniżej metoda pozwalająca na zwrócenie dowolnego typu danych
      public T getGenericItem(int index)
      {
         return genericArray[index];
      }
      public void setValue(int index, int value)
      {
         array[index] = value;
      }
      // Powyżej metoda ustawiająca dane typu całkowitego (int)
      // Poniżej metoda pozwalająca na ustawienie dowolnego typu danych
      public void setGenericValue(int index, T value)
      {
         genericArray[index] = value;
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         // Utworzenie tablicy liczb całkowitych oraz jej wypełnienie
         MyGenericArray<int> intArray = new MyGenericArray<int>(5);
         for (int i = 0; i < 5; i++)
         {
            intArray.setGenericValue(i, i * 3);
         }
         // Wypisanie wszystkich danych
         for (int i = 0; i < 5; i++)
         {
            Console.WriteLine("Liczba: {0}", intArray.getGenericItem(i));
         }
         // Używając tej samej generycznej klasy jesteśmy w stanie zadeklarować innym typ danych
         MyGenericArray<char> charArray = new MyGenericArray<char>(5);
```

```
        for (int i = 0; i < 5; i++)
        {
            charArray.setGenericValue(i, (char)(i + 97));
        }
        // Wypisanie wszystkich danych
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine(charArray.getGenericItem(i));
        }
        Console.ReadKey();
        // Wynik działania programu
        // Liczba: 0
        // Liczba: 3
        // Liczba: 6
        // Liczba: 9
        // Liczba: 12
        // a
        // b
        // c
        // d
        // e
    }
  }
}
```

Metoda wykorzystująca typy generyczne.

```
using System;
namespace Generics
{
  // Definicja klasy generycznej, w tym przypadku własnej tablicy
  class MyGenericArray<T>
  {
    // definicja typowanej tablicy
    private int[] array;
    // definicja generycznej tablicy
    private T[] genericArray;
    // konstruktor klasy przyjmujący jako parametr rozmiar tablicy
    public MyGenericArray(int size)
    {
      // ustalenie rozmiaru zwykłej tablicy
      array = new int[size + 1];
      // ustalenie rozmiaru tablicy generycznej
      genericArray = new T[size + 1];
    }
    public int getItem(int index)
    {
      return array[index];
    }
    // Powyżej metoda zwracająca typ danych jako int
    // Poniżej metoda pozwalająca na zwrócenie dowolnego typu danych
    public T getGenericItem(int index)
    {
      return genericArray[index];
    }
    public void setValue(int index, int value)
    {
      array[index] = value;
```

```
    }
    // Powyżej metoda ustawiająca dane typu całkowitego (int)
    // Poniżej metoda pozwalająca na ustawienie dowolnego typu danych
    public void setGenericValue(int index, T value)
    {
      genericArray[index] = value;
    }
  }
  class Program
  {
    static void Main(string[] args)
    {
      // Utworzenie tablicy liczb całkowitych oraz jej wypełnienie
      MyGenericArray<int> intArray = new MyGenericArray<int>(5);
      for (int i = 0; i < 5; i++)
      {
        intArray.setGenericValue(i, i * 3);
      }
      // Wypisanie wszystkich danych
      for (int i = 0; i < 5; i++)
      {
        Console.WriteLine("Liczba: {0}", intArray.getGenericItem(i));
      }
      // Używając tej samej generycznej klasy jesteśmy w stanie zadeklarować innym typ danych
      MyGenericArray<char> charArray = new MyGenericArray<char>(5);
      for (int i = 0; i < 5; i++)
      {
        charArray.setGenericValue(i, (char)(i + 97));
      }
      // Wypisanie wszystkich danych
      for (int i = 0; i < 5; i++)
      {
        Console.WriteLine(charArray.getGenericItem(i));
      }
      Console.ReadKey();
      // Wynik działania programu
      // Liczba: 0
      // Liczba: 3
      // Liczba: 6
      // Liczba: 9
      // Liczba: 12
      // a
      // b
      // c
      // d
      // e
    }
  }
}
```

Delegata wykorzystująca typy generyczne.

```
using System;
namespace GenericDelegates
{
  // Definicja delegata
  delegate int NumberChangeNormalDef(int i);
  // Definicja generycznego delegate
```

```
delegate T NumberChange<T>(T i);
// Statyczna klasa testowa z metodami do dodawania, mnożenia oraz wracania liczby
static class Test
{
   static int num = 10;
   public static int AddNumber(int a)
   {
      num += a;
      return num;
   }
   public static int MultiplyNumber(int m)
   {
      num *= m;
      return num;
   }
   public static int GetNumber()
   {
      return num;
   }
}
class Program
{
   static void Main(string[] args)
   {
      // Definicja delegata - dla porównania
      NumberChangeNormalDef normal = new NumberChangeNormalDef(Test.AddNumber);
      // Deklaracja instancji generycznych delegatów
      NumberChange<int> d1 = new NumberChange<int>(Test.AddNumber);
      NumberChange<int> d2 = new NumberChange<int>(Test.MultiplyNumber);
      // Wywołanie metod używając obiektu delegata
      d1(5);
      Console.WriteLine("Liczba: {0}", Test.GetNumber());
      d2(10);
      Console.WriteLine("Liczba: {0}", Test.GetNumber());
      Console.ReadKey();
      // Wynik działania programu
      // Liczba: 15
      // Liczba: 150
   }
}
}
```

ZADANIE 3.4

Proszę utworzyć 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 3.5

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 3.6

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