Programowanie w .NET – OOP

Tomasz Kujawa
Poznań 7-03-2016
Wersja 1.1

GFT

AGENDA

1. Programowanie obiektowe
2. Interfejs i jego implementacja
3. Hierarchia klas i klasy abstrakcyjne

Programowanie obiektowe - powtórzenie

Za Wikipedią:
„Programowanie obiektowe (ang. object-oriented programming) — paradygmat programowania, w którym programy definiuje się za pomocą obiektów — elementów łączących stan (czyli dane, nazywane najczęściej polami) i zachowanie (czyli procedury, tu: metody). Obiektowy program komputerowy wyrażony jest jako zbiór takich obiektów, komunikujących się pomiędzy sobą w celu wykonywania zadań.

[…]

Podstawowe założenia paradygmatu obiektowego:

- Abstrakcja
- Hermetyzacja
- Polimorfizm
- Dziedziczenie

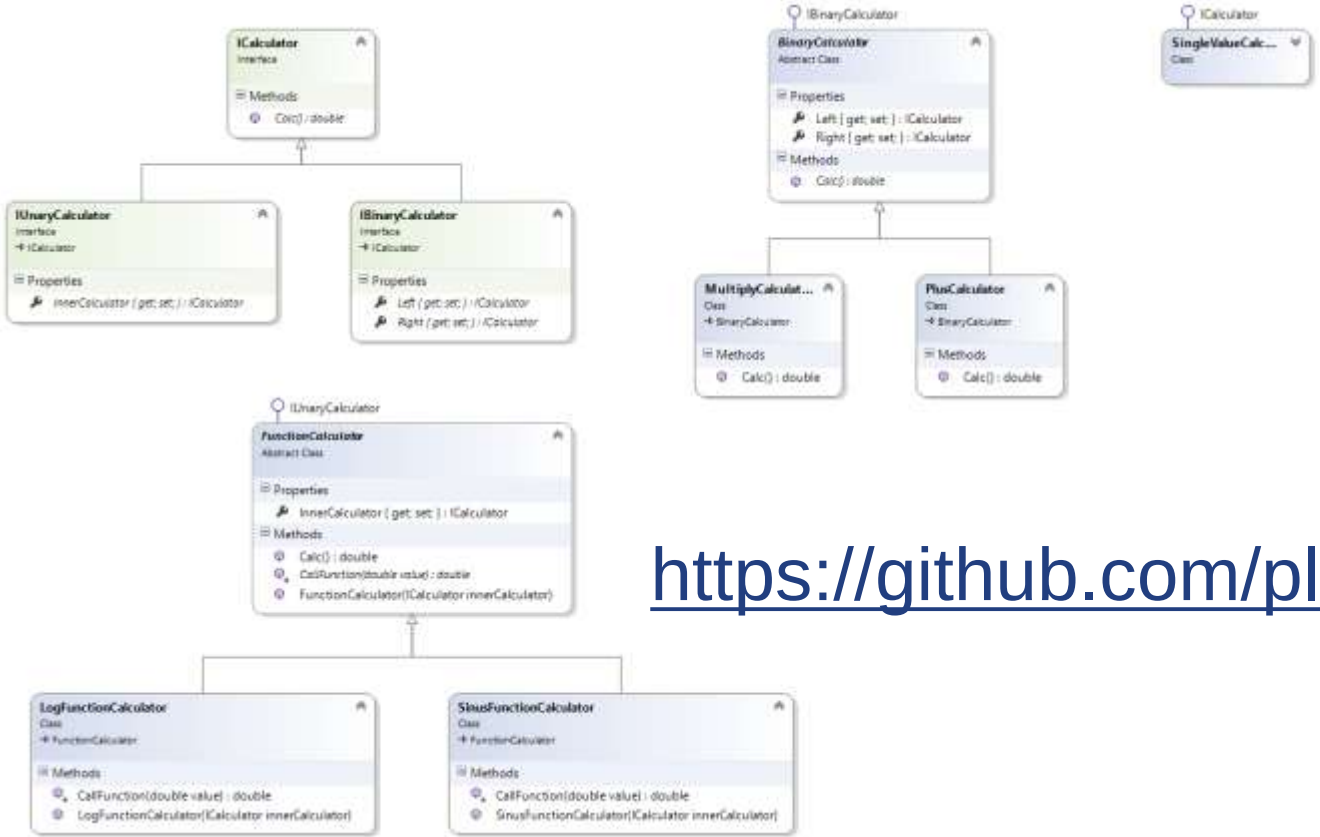
” http://pl.wikipedia.org/wiki/Programowanie_obiektowe [dostęp marzec 2015]

Przykład: kalkulator – opis

- Konstrukcje obliczeniowe będą traktowane jako osobne byty:
  - Symbole terminalne (liczby) - SingleValueCalculator
  - Dodawania – PlusCalculator
  - Mnożenia – MultiplyCalculator
  - Funkcja sinus – SinusFunctionCalculator
  - Funkcja logarytm – LogFunctionCalculator
- Wszystkie byty implementują ten sam interfejs – ICalculator
- Byty są pogrupowane w hierarchie za pomocą bytów pomocniczych

https://github.com/plbo/NETLab03

Przykład: kalkulator – docelowy stan (diagram)



https://github.com/plbo/NETLab03

Interfejs - wprowadzenie

- Interfejs =~ kontrakt. Zawiera zbiór funkcji jakie ma realizować implementująca go klasa lub struktura
- Interfejs wprowadza kategoryzację typów danych
  - Typ można zapytać o to czy implementuje interfejs (np. operator „is”, celem doboru dedykowanego rozwiązania)
  - Typ można rzutować na interfejs (np. operator „as”, celem ukrycia)
- Interfejs może zawierać sygnatury operacji, bez ich implementacji („metody bez ciała”)
- Wszystkie operacje zdefiniowane w interfejsie są publicznymi operacjami w ramach klasy lub struktury (brak modyfikatora dostępu w ramach interfejsu)
- Interfejs może zawierać opis tylko:
  - Metod
  - Właściwości
  - Zdarzeń
  - Opis indeksowania (ang. Indexer)
- Opisy składowych mogą być dostarczone w dowolnej kombinacji, liczbie i kolejności

Interfejs – implementacja

- Jeden interfejs może być implementowany przez wiele klas
- Jedna klasa może implementować wiele interfejsów
- W przypadku konieczności z korzystania z nieistniejące na platformie .NET wielo-dziedziczenia (dla klas), stosuje się interfejsy
- Implementacja składowych interfejsu musi być:
  - Publiczna
  - Niestatyczna
  - O takiej samej nazwie i sygnaturze – w celu identyfikacji 1:1
- Implementowany interfejs musi być zaimplementowany w całości (*blad kompilacji*)
- Implementacja może dostarczać elementy niezdefiniowane w interfejsie (np. dodatkowe metody składowe)
- Implementacja może być polimorficzna
- Interfejsu nie można powołać do życia – tylko obiekt implementujacy, choc przypisac mozna!
- Interfejs można zaimplementować jawnie (bez modyfikatora publicznego, ale z przedrostkiem typu interfejsu przy nazwie składowej, np. ICalculator.Calc() { })

Przykład: Prosty Interfejs ICalculator

- Interfejs ICalculator zawiera jedną operację:
  - Nazwa: Calc
  - Typ zwracany: double
  - Parametry: brak
- Interfejs ICalculator jest bezpośrednio implementowany przez klasę SingleValueCalculator, której pozostałe składowe to:
  - Publiczna właściwość (get/set) o nazwie Value i typie double
  - Publiczny konstruktor z parametrem typu double, który jest przypisywany do właściwości
  - Implemetnacja metody Calc polega na zwróceniu wartości właściwości Value

Interfejsy a dziedziczenie

- Interfejs może dziedziczyć po jednym lub więcej innych interfejsach
- Interfejs nie może dziedziczyć po innych bytach niż interfejsy (np. po klasach)
- Klasa implementująca interfejs, który dziedziczy po innych interfejsach musi dostarczyć implementację dla wszystkich interfejsów

Przykład: Dziedziczenie interfejsu ICalculator

- Interfejs ICalculator jest dziedziczony przez dwa interfejsy:
  - IUnaryCalculator – dla operacji, które bazują na pojedynczym wyniku innej operacji, np. wywołanie funkcji, której argument jest wyliczany z sumy, sin(x + y)
  - IBinaryCalculator – dla operacji, które mają dwie składowe, np. operacje dodawania x + y
- IUnaryCalculator
  - Zawiera dodatkową właściwość (get/set) o nazwie InnerCalculator i typu ICalculator
- IBinaryCalculator
  - Zawiera dodatkowe dwie właściwości (get/set) typu ICalculator o nazwach Left i Right

Klasy abstrakcyjne

- Od tradycyjnych klas różnią się modyfikatorem „abstract”
- Nie można ich bezpośrednio powołać do życia
- Posiadają niepełną implementację
- Brakująca implementacja musi być dostarczona przez pierwszą nieabstrakcyjną klasę po niej dziedziczącą w hierarchii dziedziczenia
- Mogą dziedziczyć po innych klasach i implementować interfejsy
- Składowe z brakującą implementacją mają modyfikator „abstract”
  - Metody nie mogą zawierać ciała ({ }), w zamian tego jest średnik

- Przeciwieństwem dla klas abstrakcyjnych są klasy „zapieczętowane” (ang. sealed), które
  - Zawierają pełną implementację
  - Nie można po nich dziedziczyć => Nie można modyfikować składowych

Klasy abstrakcyjne – przykład

- W projekcie można zdefiniować dwie klasy abstrakcyjne:
  - BinaryCalculator – klasa odpowiedzialna za dokonywanie „nieznanych” operacji na dwóch innych obiektach implementujących ICalculator
  - FunctionCalculator – klasa odpowiedzialna za wywołanie „nieznanych” funkcji
- BinaryCalculator:
  - Implementuje interfejs IBinaryCalculator
  - Posiada implementacje dla właściwości Left i Right (metody get/set)
  - Metoda Calc (przychodząca z interfejsu) jest publiczna i abstrakcyjna
- FunctionCalculator:
  - Implementuje interfejs IUnaryCalculator
  - Posiada implementację dla właściwości InnerCalculator (metody get/set)
  - Posiada konstruktor, który przyjmuje jako parametr typ ICalculator, który jest przypisany do w/w właściwości
  - Posiada dodatkową metodę chronioną i abstrakcyjną o nazwie CallFunction, której parametrem i typem zwracanym jest double
  - Posiada implementację publicznej metody Calc (przychodząca z interfejsu), w której ciele jest wywołana metoda abstrakcyjna CallFunction. Argumentem dla CallFunction jest wartość wywołania metody Calc na obiekcie InnerCalculator. Wartość zwrócona przez CallFunction jest wynikiem działania metody Calc.

Dziedziczenie – wprowadzenie

- Mechanizm pozwalający na tworzenie nowych klas przez:
  - Ponowne użycie istniejących klas
  - Rozszerzenie funkcjonalności istniejących klas
  - Modyfikacja funkcjonalności istniejących klas (new/virtual/override)
- Składowe klasy bazowej (słowo kluczowe base) stają się składowymi klasy dziedziczącej, pod warunkami:
  - Publiczne stają publiczne
  - Chronione są chronione
  - Prywatne są niedostępne
- Klasa może mieć tylko jedną klasę basową (z której dziedziczy)
- Zachodzi relacja przechodniości:
  - Jeżeli klasa A dziedziczy po B, a klasa B po C, to klasa A dziedziczy po C
  - Jeżeli klasa A dziedziczy po B, a klasa B implementuje interfejs ID, to klasa A implementuje interfejs ID
- Struktury nie wspierają dziedziczenia
- Jeżeli składowe klasy bazowej są abstrakcyjne to klasa musi dostarczyć ich implementację (chyba, ze klasa jest abstrakcyjna!)

Dziedziczenie – modyfikacja funkcjonalności istniejących klas

- W klasa dziedziczącej istnieje możliwość zmodyfikowania metody z klasy bazowej, za pomocą dwóch mechanizmów:
  - Przesłonięcia (słowo kluczowe override)
  - Utworzenia nowej metody o takiej sygnaturze (słowo kluczowe new)
- Przesłonięcie:
  - Działa niezależnie od kontekstu wywołania obiektu
  - Konieczne przy metodach abstrakcyjnych
  - Możliwe przy metodach w klasie bazowej zdefiniowanych jako wirtualne (słowo kluczowe virtual)
- Utworzenie nowej składowej o takiej samej sygnaturze
  - Zależy od kontekstu:
    - Jeżeli jest użyte rzutowanie na typ bazowy, to przy wywołaniu zostanie użyta implementacja składowej z typu bazowego

Dziedziczenie – przykład (1)

- Po klasie BinaryCalculator dziedziczą dwie nieabstrakcyjne klasy:
  - PlusCalculator – odpowiedzialna za realizację operacji dodawania
  - MultiplyCalculator – odpowiedzialna za realizację operacji mnożenia
- Obie klasy:
  - Przesłaniają i dostarczają implementację dla metody Calc
  - Odpowiednio dodają/mnożą wyniki wywołania metody Calc dla właściwości Left i Right

Dziedziczenie – przykład (2)

- Po klasie FunctionCalculator dziedziczą dwie nieabstrakcyjne klasy:
  - SinusFunctionCalculator - odpowiedzialna za realizację wywołania funkcji sinus (Math.Sin)
  - LogFunctionCalculator – odpowiedzialna za realizację wywołania funkcji logarytm naturalny (Math.Log)
- Obie klasy:
  - Przesłaniają i dostarczają implementację dla metody CallFunction
  - W ramach implementacji metody CallFunction:
    - Parametr jest przekazywany do odpowiedniej funkcji matematycznej (sin/log)
    - Wartość funkcji matematycznej jest wartością zwracaną przez metodę CallFunction
  - Dostarczają konstruktor z parametrem typu ICalculator, który jest przekazywany jest do konstruktora klasy bazowej

Hermetyzacja/Kapsułkowanie

- Mechanizm ukrywania implementacji
- Zapobiega niepowoływanemu dostaniu się do poszczególnych kroków implementacji
- Zazwyczaj realizowane za pomocą jawnego interfejsu i publicznych składowych

- Przykład:
  - Wszystkie klasy są ukryte za interfejsem ICalculator
  - Szczegóły implementacji operacji matematycznych są niezależne od innych kroków dla przewidzianych operacji kalkulacji

Polimorfizm

- Dostosowanie zachowania do rodzaju obiektu, na którym wykonywana jest operacja
  - Na podstawie informacji o typie, środowisko uruchomieniowe wybiera właściwą implementacji operacji
- Domyślne zachowanie dla interfejsów
- Działa w przypadku metod przesłoniętych (override)
- NIE DZIAŁA dla nowych metod o tej samej sygnaturze (słowo kluczowe new)

Polimorfizm – przykład

- Należy utworzyć nową klasę w projekcie testowym (MyCalculator.Tests)
- Klasę należy opatrzyć atrybutem TestClass z przestrzeni nazw Microsoft.VisualStudio.TestTools.UnitTesting
- Każdą z metod (właściwych testów) należy opatrzeć atrybutem TestMethod z przestrzeni nazw: Microsoft.VisualStudio.TestTools.UnitTesting
- Należy przetestować zachowanie odpowiedniej hierarchii obiektów w testach:
  1. (1 + 2) * (4 + 6) //Oczekiwany wynik 30
  2. (sin(Math.PI) + sin(Math.PI / 2)) //Oczekiwany wynik 1.0
  3. ((log(Math.E) * 6) + 4) * 10 //Oczekiwany wynik 100

- Porównania z wartością oczekiwaną należy dokonywać za pomocą metody Assert.AreEqual
- Należy pamiętać o reprezentacji liczb zmiennoprzecinkowych w pamięci