

**Zawartość:**

- plusy dziedziczenia
- co to jest dekoracja
- co to jest kompozycja
- dlaczego kompozycja jest spoko

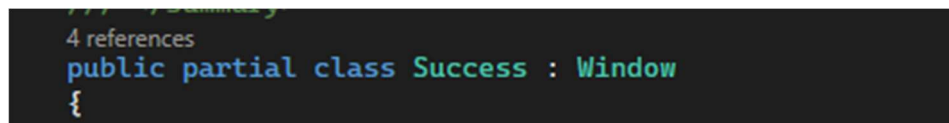
## Dziedziczenie

Jak działa dziedziczenie w programowaniu obiektowym wie mniej więcej każdy. Po co się je stosuje?

- aby nie powtarzać wielokrotnie tego samego kodu
- aby skorzystać z polimorfizmu, na którym twarde stoją frameworki
- aby dekorować klasy
- aby klasa B(potomek) mogła używać funkcjonalności klasy A(rodzic)

Pierwsze dwa podpunkty są w porządku. Zwłaszcza ten drugi. Aby uzyskać efekty z punktu trzeciego i czwartego istnieje lepsza droga.

Przykład z WPFa:



```
4 references  
public partial class Success : Window  
{
```

Klasa **Success** dziedziczy z klasy **Window**:

- pierwszy punkt jest odhaczony bo nie trzeba powtarzać kodu dla każdego okna
- drugi punkt jest odhaczony bo dzięki polimorfizmowi framework łączy naszą klasę jakby była klasą Window i jednocześnie wie, że ma wykonywać metody Success, a nie Window (o ile je nadpisujemy)

Na takim mechanizmie stoją w większości frameworki. Dzięki dziedziczeniu programista dostaje kod, którego nie chce mu się pisać/nie powinien pisać za każdym razem oraz obiekt klasy **potomnej** da się wepchnąć do referencji typu **klasy bazowej**.

## Dekoracja za pomocą dziedziczenia

Chodzi o to, że mamy klasę A, której nie możemy modyfikować z jakiegoś powodu w celu dopisania własnego kodu (bo jest to klasa z biblioteki lub zmiana taka spowoduje konieczność modyfikacji miejsc gdzie dana klasa jest używana, a takowych może być sporo). Dekoracja polega na utworzeniu nowej klasy B, która dziedziczy z klasy A oraz dopisanie kodu do odziedziczonej zawartości. Przykład poniżej:

```

1 reference
class A
{
    1 reference
    public void func1()
    {
        Console.WriteLine("ASD");
        //do something
    }

    0 references
    public void func2()
    {
        //do something
    }
}

0 references
class B : A
{
    0 references
    public void func1()
    {
        //do something
        base.func1();
    }
}

```

Metoda **func1** klasy B dekoruje metodę **func1** klasy A. W tym przypadku następuje dopisanie jakiejś logiki i wywołanie metody bazowej (dekoracja). W C++ sytuacja jest bardzo podobna.

## Kompozycja

Kompozycja to umieszczanie obiektu w obiekcie tak by ten mógł korzystać z funkcjonalności obiektu zagnieżdżonego.

```

2 references
class A
{
    1 reference
    public void func1()
    {
        Console.WriteLine("ASD");
        //do something
    }

    0 references
    public void func2()
    {
        //do something
    }
}

0 references
class B
{
    private A a = new A();

    0 references
    public void func1()
    {
        a.func1();
        //do something
    }
}

```

Każdy kto przeanalizuje dwa powyższe obrazki dojdzie do wniosku, że efekt jest taki sam. Co jest zatem lepsze?

Aby odpowiedzieć na to pytanie trzeba sobie przypomnieć co to jest **dependency inversion** oraz **dependency injection**.

Zacznijmy od tego drugiego pojęcia. Wstrzykiwanie zależności polega na wstrzykiwaniu zagnieżdżonego obiektu do obiektu z zewnątrz np. w momencie tworzenia go poprzez konstruktor.

```
1 reference
class B
{
    private A _a;

    0 references
    public B(A a) => _a = a;

    0 references
    public void func1()
    {
        _a.func1();
        //do something
    }
}
```

Proste prawda? Aby odpowiedzieć jakie są plusy DI trzeba wiedzieć co to jest polimorfizm oraz **dependency inversion**. Pojęcie to mówi by stosować zawsze typ bazowy jako typ referencji.

```
4 references
class A
{
    3 references
    public virtual void func1()
    {
        Console.WriteLine("ASD");
        //do something
    }
}

0 references
class A1 : A
{
    2 references
    public override void func1()
    {
        Console.WriteLine("QWE");
        //do something diffrent way
    }
}

0 references
class A2: A
{
    2 references
    public override void func1()
    {
        Console.WriteLine("ZQW");
        //do something diffrent way
    }
}
```

Mamy tutaj trzy implementacje tej samej logiki.

```

7 references
class B
{
    private A _a;

    3 references
    public B(A a) => _a = a;

    0 references
    public void func1()
    {
        _a.func1();
        //do something
    }
}

0 references
internal class Program
{
    0 references
    static void Main(string[] args)
    {
        B b1 = new B(new A());
        B b2 = new B(new A1());
        B b3 = new B(new A2());
    }
}

```

Dzięki inwersji możliwe jest podanie trzech różnych implementacji do prywatnego pola typu **A**. Mało tego. Dzięki słowu `virtual` kompilator będzie wiedział, której **func1** użyć.

Można teraz powiedzieć, że kompozycja jest lepsza od dziedziczenia, ponieważ jest bardziej mobilna gdy wykorzystujemy inwersje referencji oraz wstrzykiwanie zależności. W programie można dynamicznie decydować, którą implementację podsunąć do konstruktora (dzięki czemu dekoracja będzie inna za każdym razem).

Dziedziczenie to coś mocno statycznego (nie mylić ze `static`). Nie da się zmodyfikować dziedziczenia w trakcie działania programu. Przez to jest ono bardziej mobilne.

Wydaje mi się, że dziedziczenie jest przydatne, gdy chcemy korzystać z polimorfizmu. Poza tym kompozycja wygrywa.

**COMPOSITION OVER INHERITANCE**

**PREFER YOU MUST**

[memegenerator.net](http://memegenerator.net)