

# Projektowanie obiektowe oprogramowania

## Zestaw B

Inversion of Control (3)

2021-05-25

Liczba punktów do zdobycia: **6/74**

Zestaw ważny do: 2021-06-08

*Uwaga! Kontynuacja/zakończenie pracy nad silnikiem Inversion of Control na identycznych zasadach. W szczególności obowiązkową częścią każdego zadania są testy jednostkowe, nawet jeśli nie wspomina się o tym w treści zadań.*

### 1. (2p) (Wstrzykiwanie właściwości)

Silnik IoC rozbudować o możliwość wstrzykiwania właściwości/metod (do wyboru).

Ściślej - kontener w trakcie rozwikływania zależności obiektu dodatkowo analizuje właściwości i metody i te, które są opatrzone atrybutami (anotacjami) [DependencyProperty]/[DependencyMethod] i spełniają warunki wstrzykiwania (na przykład: metoda ma określoną sygnaturę, typu `setXYZ( X x, Y y, ... )`) próbuje rozwikływać i wstrzykiwać tak samo, jak robi to z parametrami konstruktora.

Warunki wstrzykiwania są następujące.

Dla właściwości (C# i inne języki z właściwościami, ale nie Java), "wstrzykiwalne" są właściwości które mają publiczny akcesor `set`.

Dla metod (wszystkie języki), "wstrzykiwalne" są metody które nie zwracają żadnego wyniku `void` i mają niepustą listę argumentów.

Kolejność wstrzykiwania właściwości/metod nie ma znaczenia.

```
public class A {  
    ...  
  
    // wstrzykiwanie przez konstruktor (to już mamy)  
    public A( B b ) { }  
  
    // wstrzykiwanie przez właściwość  
    [DependencyProperty]  
    public C TheC { get; set; }  
  
    // wstrzykiwanie przez metodę  
    [DependencyMethod]  
    public void setD( D d )  
    {  
        this.d = d;  
    }  
}  
  
public class B {}  
public class C {}
```

```

public class D {}

SimpleContainer c = new SimpleContainer();
A a = c.Resolve<A>();

// tworzy nową instancję A z B wstrzykniętym przez konstruktor,
// C wstrzykniętym przez właściwość i D wstrzykniętym przez metodę.

```

## 2. (2p) (Uzupełnianie zależności istniejącego obiektu)

Silnik IoC rozbudować o możliwość uzupełniania zależności istniejących obiektów, zbudowanych poza kontenerem.

```

public class SimpleContainer
{
    ...
    public void BuildUp<T>( T Instance );
}

SimpleContainer c = new SimpleContainer();

A theA = ....; // obiekt theA SKĄDŚ pochodzi, ale nie z kontenera
c.BuildUp( theA );

// wstrzykuje do theA zależności przez właściwości, tu: TheC

```

## 3. (1p) (Lokalizator usług (Service locator))

Dodajemy do silnika funkcjonalność lokalizatora usług (*Service Locator*). Typowo lokalizator jest singletonem, z zadaną logiką rozwikływania kontenera (w szczególności kontener też może być singletonem), a klient używa lokatora do rozwikłania zależności do usług (implementacji).

Lokalizator pozwala zmniejszyć zależności między komponentami - zamiast wymagać zależności do usług (nawet wstrzykiwane!), obiekt może wyłącznie znać odwołanie do lokalizatora, a w przypadku gdy lokalizator jest singletonem obiekt w ogóle nie musi wymagać żadnych zależności.

```

public delegate SimpleContainer ContainerProviderDelegate();

public class ServiceLocator {

    public static void SetContainerProvider( ContainerProviderDelegate ContainerProvider ) {
        ...
    }

    public static ServiceLocator Current
    {
        get {
            ... singleton ...
        }
    }

    public T GetInstance<T>();
}

// Przykład 1, rejestrowanie lokatora

SimpleContainer c = new SimpleContainer();
ContainerProviderDelegate containerProvider = () => c;

ServiceLocator.SetContainerProvider( containerProvider );

/* zapamiętano funkcję rozwikłującą referencję do kontenera - to zawsze będzie TEN sam kontener */

```

```
// Przykład 2, rejestrowanie lokatora

ContainerProviderDelegate containerProvider = () => new SimpleContainer();
ServiceLocator.SetContainerProvider( containerProvider );

/* zapamiętano funkcję rozwikłującą referencję do kontenera - to zawsze będzie NOWY kontener */

// Przykład 3, kod kliencki

class Foo

    void Bar() {

        IService service = ServiceLocator.Current.GetInstance<IService>();

        // GetInstance używa zarejestrowanej metody do rozwikływania kontenera
        // do pobrania kontenera i rozwikłania zależności
        // Tym razem jednak IService nie musiało być wstrzykiwane
    }
}
```

Zwrócić uwagę na pewien szczegół implementacyjny - klient lokalizatora może zechcieć otrzymać referencję do kontenera, którego wewnętrznie używa lokalizator.

```
SimpleContainer container = ServiceLocator.Current.GetInstance<SimpleContainer>();
```

Taka potrzeba ma swoje uzasadnienie - zwyczajowo interfejs kontenera jest szerszy niż interfejs lokalizatora (np. kontener potrafi uzupełniać zależności w istniejących obiektach, patrz poprzednie zadanie).

Poprawnie obsłużyć ten przypadek.

#### 4. (1p) (Dostawca zależności (Local Factory/Depdency Resolver))

Z wykładu wiemy, że **Service Locator** to nie jest najlepszy pomysł na zapewnienie dostępności usług w dowolnym miejscu aplikacji (przy założeniu że nie mogą być dostarczone przez wstrzykiwanie zależności). Wiemy, że lepszym sposobem jest lokalny dostawca zależności, (**Local Factory**).

Taki lokalny dostawca tym różni się od globalnego lokatora, że jest zdefiniowany lokalnie i nie korzysta z żadnej konkretnej ramy wstrzykiwania zależności. Zamiast tego - określa abstrakcję (specyfikację, interfejs) usługi i pozwala ją dowolnie zaimplementować w korzeniu kompozycji (**Composition Root**).

Pokazać przykład takiego lokalnego dostawcy zależności dla jakiegoś podsystemu (wysyłanie powiadomień, drukowanie, pobieranie danych) wraz z implementacją dla swojej własnej ramy IoC.

*Wskazówka! W notatkach do wykładu znajduje się wskazanie artykułu omawiającego tę kwestię, należy tylko ten artykuł przeczytać, zrozumieć i umieć stosować w praktyce. Na wykładzie również pokazywaliśmy kilka razy przykład implementacji takiego lokalnego dostawcy.*

Wiktor Zychla