

Projektowanie obiektowe oprogramowania

Wzorce architektury aplikacji (2)

Wykład 10

Inversion of Control/Dependency Injection

Wiktor Zychla 2024

1 Inversion of Control vs Dependency Injection

Inversion of Control (Dependency Inversion) = zestaw technik pozwalających tworzyć struktury klas o luźniejszym powiązaniu.

Trzy kluczowe skojarzenia:

1. **Późne wiązanie** – możliwość modyfikacji kodu bez rekompilacji, wyłącznie przez rekonfigurację, „*programming against interfaces*” (DIP)
2. **Ułatwienie tworzenia testów jednostkowych** – zastąpienie podsystemów przez ich stuby/fake’i
3. **Uniwersalna fabryka** – tworzenie instancji dowolnych typów według zadanych wcześniej reguł

Dependency Injection = konkretny sposób realizacji IoC w językach obiektowych

... *Inversion* = ogólna zasada

... *Injection* = jej implementacja

2 ... więc przypomnijmy Dependency Inversion Principle

Zalety:

1. **Rozszerzalność** (OCP) – teoretycznie możliwe rozszerzenia o konteksty nie znane w czasie planowania
2. **Równoległa implementacja** – dobrze zdefiniowany kontrakt zależności pozwala rozwijać oba podsystemy niezależnie
3. **Konserwowalność (maintainability)** – dobrze zdefiniowana odpowiedzialność to zawsze lepsza architektura i zwykle łatwiejsza konserwacja
4. **Łatwość testowania** - obie klasy mogą być testowane niezależnie; ta z wstrzykiwaną zależnością może być testowana przez wstrzyknięcie stuba/fake’a
5. **Późne wiązanie** – możliwość określenia konkretnej implementacji nawet bez rekompilacji

3 Twarde zależności vs miękkie zależności

Jeszcze inne spojrzenie na modularność:

1. **Sztywna zależność** (stable dependency) – klasyczna modularność; zależne moduły już istnieją, są stabilne, znane i przewidywalne (np. biblioteka standardowa)
2. **Miękka zależność** (volatile dependency) – modularność dla której zachodzi któryś z powodów wprowadzenia spoiny:
 - a. Konkretnie środowisko może być konfigurowane dopiero w miejscu wdrożenia (późne wiązanie)
 - b. Moduły powinny być rozwijane równolegle
3. **Spoina** (seam) – miejsce, w którym decydujemy się na zależność od interfejsu zamiast od konkretnej klasy

Uwaga. O ile zastosowanie technik DI pozwala na wprowadzenie miękkich zależności w miejscach spoin, o tyle zwykłe zależności do samych ram (*frameworków*) DI mają charakter sztywny.

Innymi słowy, nie projektuje się aplikacji w taki sposób, żeby móc miękko przekonfigurowywać je na różne implementacje ram DI. Zobaczymy jednak wzorzec **Local Factory**, który w praktyce na tyle izoluje użycie ramy DI w aplikacji, że jej wymiana na inną nie stanowi większego problemu.

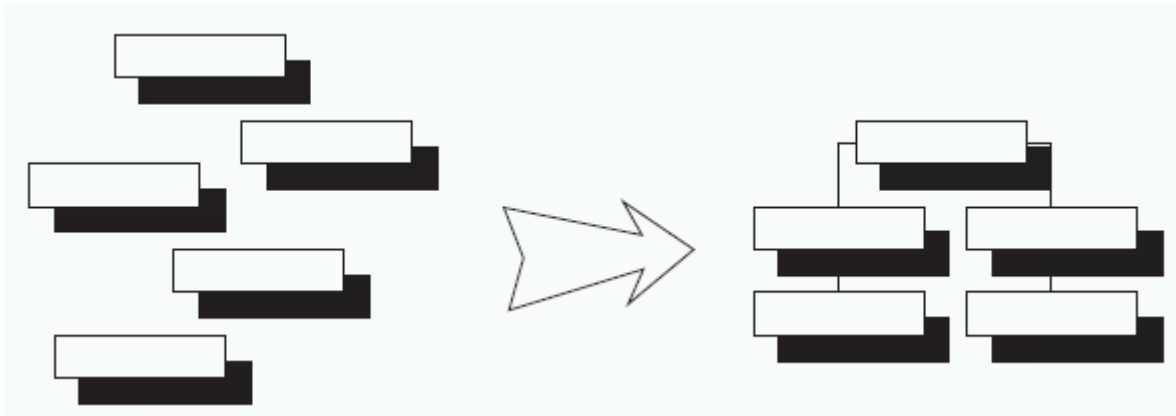
4 Kluczowe podwzorce Dependency Injection (na przykładzie kontenera Unity)

Uwaga. O ramie (*frameworku*) DI mówimy żargonowo **kontener DI** (rzadziej **Kernel DI**).

[Kontener Unity](#) to jedna z wielu implementacji ramy Dependency Injection dla .NET. Inne implementacje:

- dla .NET – NInject, Castle Windsor
- dla Java – CDI, Spring DI

4.1 Składanie obiektów (Composition)



1. **Kontener/kernel** – obiekt usługowy którego zadaniem jest tworzenie instancji i rozwiązywanie zależności (ang. *dependency resolving*). To jest ta **uniwersalna fabryka**.
2. **Rozwiązywanie zależności sztywnych** (opisanych konkretnym typem)

```
class Program
{
    static void Main(string[] args)
    {
        var container = new UnityContainer();

        var foo = container.Resolve<Foo>();

        Console.WriteLine( foo.GetType() );

        Console.ReadLine();
    }
}

public class Foo
{
}
```

3. Rozwiązywanie zależności miękkich (opisanych mapowaniem abstrakcji na implementację)

```
class Program
{
    static void Main(string[] args)
    {
        var container = new UnityContainer();
        container.RegisterType<IFoo, Foo>();

        var foo = container.Resolve<IFoo>();

        Console.WriteLine( foo.GetType() );

        Console.ReadLine();
    }
}

public interface IFoo
{
}

public class Foo : IFoo
{
}
```

4. Rozwiązywanie instancji

```
class Program
{
    static void Main(string[] args)
    {
        var container = new UnityContainer();

        container.RegisterInstance<IFoo>( new Foo() );

        var foo = container.Resolve<IFoo>();

        Console.WriteLine( foo.GetType() );

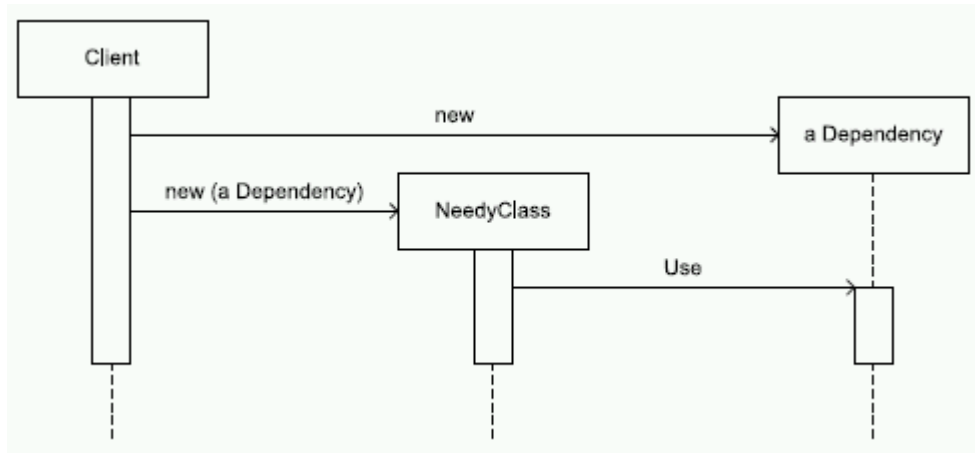
        Console.ReadLine();
    }
}

public interface IFoo
{
}

public class Foo : IFoo
{
}
```

5. Rozwiązywanie grafu zależności - a co jeśli w grafie występują cykle?

6. **Wstrzykiwanie przez konstruktor** – najdłuższy lub wskazany ([InjectionConstructor]). Zależność jest zawsze dostępna, bo nie da się wykonstruować obiektu nie rozwiązując jego zależności (ang. *satisfy its dependencies*).. Wstrzykiwanie przez konstruktor jest z tego powodu rekomendowane.



```
class Program
{
    static void Main(string[] args)
    {
        var container = new UnityContainer();

        container.RegisterType<IFoo, Foo>();
        container.RegisterType<IBar, Bar>();

        var foo = container.Resolve<IFoo>();

        Console.WriteLine(foo.GetType());

        Console.ReadLine();
    }
}

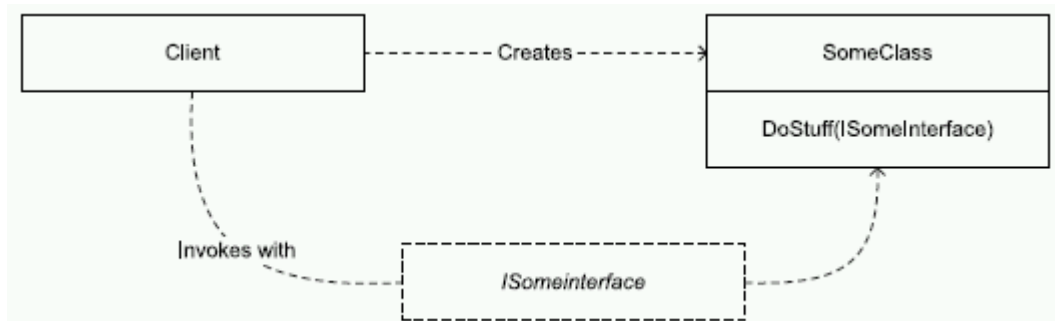
public interface IFoo
{
}

public class Foo : IFoo
{
    public Foo( IBar bar )
    {
    }
}

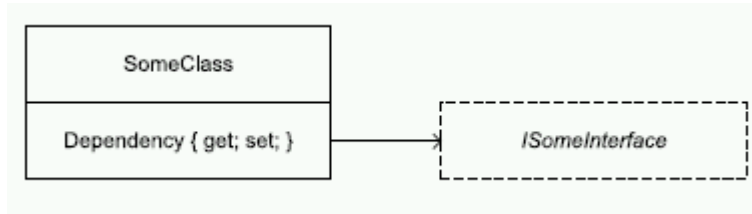
public interface IBar
{
}

public class Bar : IBar
{
}
```

7. **Wstrzykiwanie przez metodę** – atrybut [InjectionMethod] – zapewnienie że w różnych kontekstach (metodach) wstrzykiwane mogą być inne zależności (różne metody mogą mieć różne zależności)



8. **Wstrzykiwanie przez właściwości** ([Dependency]) – zapewnienie że domyślna zależność jest dostępna, ale może być zmodyfikowana (bo klient wartość właściwości może zawsze zmienić)



9. „Budowanie” obiektu wyprodukowanego na zewnątrz (ang *build-up*)

```

class Program
{
    static void Main(string[] args)
    {
        var container = new UnityContainer();

        container.RegisterType<IBar, Bar>();

        var foo = new Foo();
        container.BuildUp(foo);

        Console.WriteLine(foo.Bar.GetType());

        Console.ReadLine();
    }
}

public interface IFoo
{
}

public class Foo : IFoo
{
    [Dependency]
    public IBar Bar { get; set; }
}

public interface IBar
{
}

```

```
public class Bar : IBar
{

}
```

10. **Rejestracja metody fabrykującej** (Injection Factory)– zapewnienie możliwości tworzenia zależności przez dowolną metodę fabrykującą. To **najogólniejszy**, najbardziej uniwersalny sposób określania zależności i nadaje się do najbardziej złożonych scenariuszy. Przykład: należy wykonstruować obiekt z rozwiązanymi zależnościami, a następnie zwrócić proxy do niego.

```
class Program
{
    static void Main(string[] args)
    {
        var container = new UnityContainer();

        container.RegisterType<IBar, Bar>();
        container.RegisterFactory<IFoo>(
            c =>
            {
                var b = c.Resolve<IBar>();
                var f = new Foo();
                f.Bar = b;

                return f;
            });

        var foo = container.Resolve<IFoo>();

        Console.WriteLine(foo.GetType());
        Console.WriteLine(foo.Bar.GetType());

        Console.ReadLine();
    }
}

public interface IFoo
{
    IBar Bar { get; set; }
}

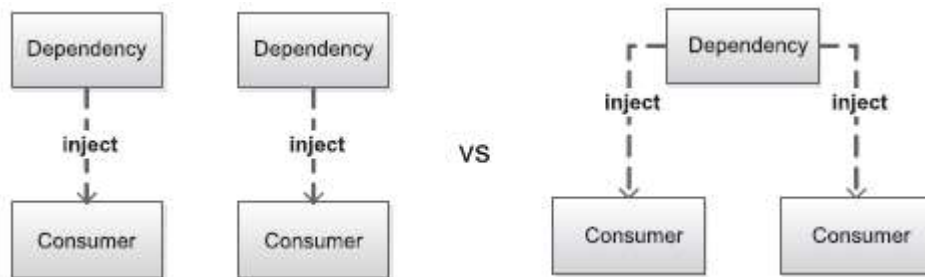
public class Foo : IFoo
{
    [Dependency]
    public IBar Bar { get; set; }
}

public interface IBar
{
}

public class Bar : IBar
{
}
```

}

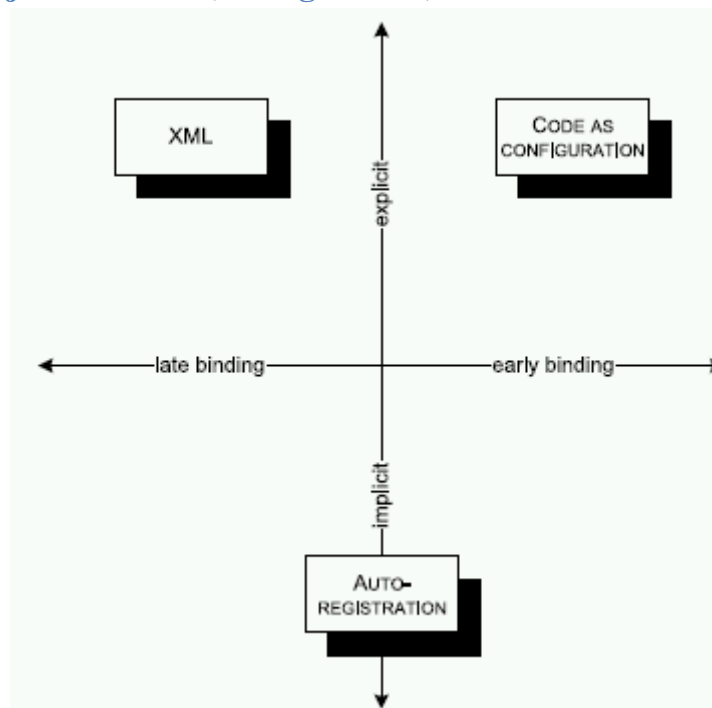
4.2 Zarządzanie czasem życia obiektów (Lifecycle Management)



[http://msdn.microsoft.com/en-us/library/ff660872\(PandP.20\).aspx](http://msdn.microsoft.com/en-us/library/ff660872(PandP.20).aspx)

1. **Transient** – ulotne
2. **ContainerControlled** – singletony
3. **Hierarchical** – singletony, ale inne w dziedziczonych kontenerach
4. **PerThread** – inny obiekt per wątek
5. **PerHttpContext** – inny obiekt per żądanie HTTP do serwera aplikacyjnego
6. **Custom**

4.3 Konfiguracja kontenera (Configuration)



(tu: wymaga pakietu **Unity.Configuration**)

Zwyczajowo kontenery dostarczają trzech sposobów konfiguracji:

- **Konfiguracja deklaratywna** – mapowania typów opisane są w pliku konfiguracyjnym, zwykle w formacie XML. Rekonfiguracja polega na modyfikacji pliku XML w miejscu osadzenia aplikacji. To bardzo praktyczna możliwość, ponieważ tę samą aplikację można różnie skonfigurować w różnych miejscach wdrożenia, bez potrzeby rekompilacji.

Przykładowy **app.config**

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="unity"
      type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
        Unity.Configuration"/>
  </configSections>
  <unity configSource="Configuration\unity.config"/>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
  </startup>
</configuration>
```

Przykładowy **unity.config**

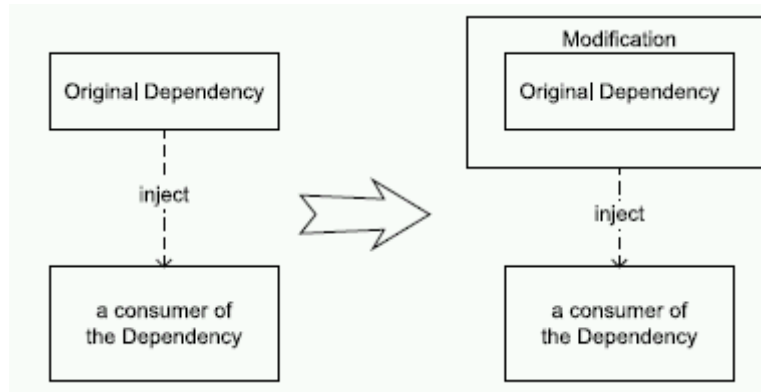
```
<?xml version="1.0" encoding="utf-8" ?>
<unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
  <alias alias="IFoo" type="ConsoleApp.IFoo, ConsoleApp"/>
  <alias alias="Foo" type="ConsoleApp.Foo, ConsoleApp"/>
  <container>
    <register type="IFoo" mapTo="Foo" />
  </container>
</unity>
```

Ładowanie konfiguracji deklaratywnej

```
var container = new UnityContainer();
container.LoadConfiguration();
```

- **Konfiguracja imperatywna** – mapowanie odbywa się w kodzie, w miejscu zwanym **Composition Root** (o tym dalej)
- **Autokonfiguracja** – wariant konfiguracji imperatywnej, który polega na wskazaniu zestawu (assembly) / pakietu (package), a kontener automatycznie rejestruje napotkane interfejsy na ich napotkane implementacje.

4.4 Przechwytywanie żądań (Proxy)

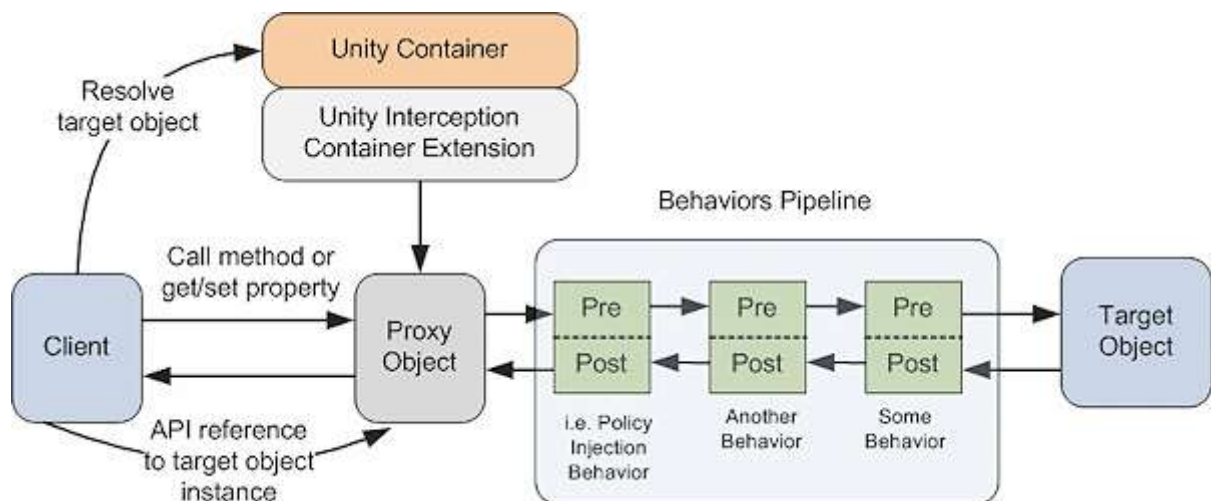


Typowe zagadnienia przekrojowe (cross-cutting concerns):

1. Audytowanie
2. Logowanie
3. Monitorowanie wydajności
4. Bezpieczeństwo
5. Cache'owanie
6. Obsługa błędów

Frameworki DI często pozwalają obsłużyć tak zdefiniowane AOP dzięki temu że zamiast obiektu mogą zwracać proxy do niego. Przykład w Unity (wymaga pakietu **Unity.Interception**):

1. **InterfaceInterceptor** – tworzy proxy przez delegowanie, pozwala przechwycić tylko metody interfejsu
2. **VirtualMethodInterceptor** – tworzy proxy przez dziedziczenie, pozwala przechwycić tylko metody wirtualne



```
using System;
using System.Collections.Generic;
using Unity;
using Unity.Interception;
using Unity.Interception.ContainerIntegration;
using Unity.Interception.InterceptionBehaviors;
using Unity.Interception.Interceptors.InstanceInterceptors.InterfaceInterception;
```

```

using Unity.Interception.PolicyInjection.Pipeline;

class Program
{
    static void Main(string[] args)
    {
        var container = new UnityContainer();
        container.AddNewExtension<Interception>();

        // zarejestrowanie typu z interceptorem
        container.RegisterType<ICalc, CalcImpl>(
            new Interceptor<InterfaceInterceptor>(),
            new InterceptionBehavior<LogInterceptionBehavior>());

        // klient konstruuje instancję zwyczajnie
        // klient nie wie że dostaje proxy
        var calc = container.Resolve<ICalc>();

        Console.WriteLine(calc.GetType());

        calc.Sum(5, 6);

        Console.ReadLine();
    }
}

public interface ICalc
{
    int Sum(int p, int q);
}

public class CalcImpl : ICalc
{
    public int Sum(int p, int q)
    {
        return p + q;
    }
}

/// <summary>
/// Interceptor, element programowania aspektowego
/// Przechwytyuje wywołania metod z docelowego obiektu
/// </summary>
public class LogInterceptionBehavior : IInterceptionBehavior
{
    #region IInterceptionBehavior Members

    public IEnumerable<Type> GetRequiredInterfaces()
    {
        return Type.EmptyTypes;
    }

    public IMethodReturn Invoke(IMethodInvocation input, GetNextInterceptionBehaviorDelegate getNext)
    {
        Console.WriteLine("wywołanie metody {0}", input.MethodBase.Name);
        foreach (object param in input.Arguments)
            Console.WriteLine("parametr {0}", param);
    }
}

```

```
        var ret = getNext().Invoke(input, getNext);

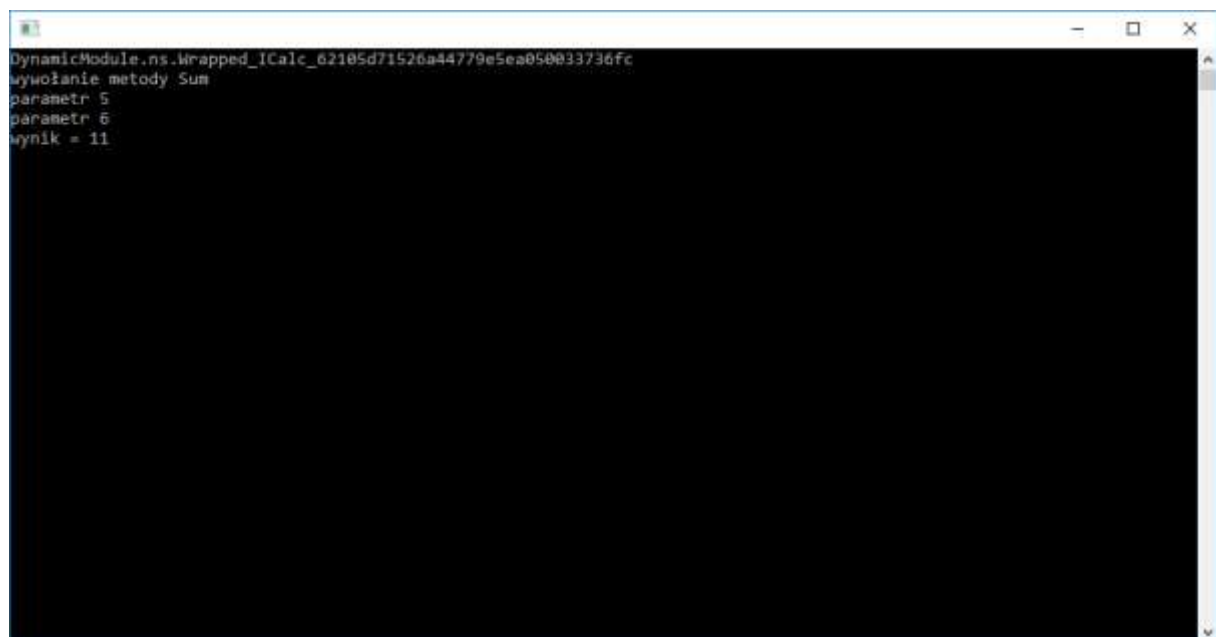
        Console.WriteLine("wynik = {0}", ret.ReturnValue);

        return ret;
    }

    public bool WillExecute
    {
        get { return true; }
    }

    #endregion
}
```

Wynik działania programu:



```
DynamicModule.ns.Wrapped_ICalc_62105d71526a44779e5ea050033736fc
wywołanie metody Sum
parametr 5
parametr 6
wynik = 11
```

5 ServiceLocator vs Composition Root+Factory/Resolver

Jak w rozbudowanej, wielomodułowej aplikacji radzić sobie z rozwiązywaniem zależności do usług?

Żeby rozwiązać zależność potrzebny jest kontener. Innymi słowy, w kodzie, w miejscu w którym potrzebujemy instancji usługi, potrzebny jest kontener.

Najgorsze rozwiązanie – przekazywać kontener jako parametr do klas/metod.

Trochę lepsze rozwiązanie – **Service Locator**.

Service Locator = schowanie singletona kontenera DI za fasadą, pozwalającą z dowolnego miejsca aplikacji na rozwiązanie zależności do usługi. Pozwala znacznie zredukować jawne zależności między klasami. Service Locator nie musi być przekazywany jako zależność, bo jako singleton, może być osiągalny z dowolnego miejsca.

Użycie:

```
// konfiguracja fasady na kontener
var locator = new UnityServiceLocator(container);
ServiceLocator.SetLocatorProvider(() => locator);

// ... w dowolnym miejscu kodu:
var foo = ServiceLocator.Current.GetInstance<IFoo>();
```

Uwaga! Service Locator uważa się za antywzorzec z uwagi na dwa niepożądane zjawiska:

1. Service Locator powoduje owszem zredukowanie zależności między klasami, ale kosztem wprowadzenia zależności do podsystemu DI. To bardzo nieeleganckie. Zastosowanie DI powinno być **przezroczyste** dla kodu – struktura klas powinna być taka sama bez względu na to czy wspomagamy się ramą DI czy nie.
2. Zależności rozwiązywane przez SL są niejawne – rozwiązywanie pojawia się w implementacji. Na poziomie struktury (metadanych) nie ma jawnej informacji że klasa A zależy od B – A sobie samo wykonstruuje B za pomocą SL kiedy jest mu to potrzebne. Problem w tym, że ponieważ tej zależności nie widać na poziomie struktury, może być trudna do wychwycenia i przez to powodować **błędy w czasie wykonania programu** (wtedy, gdy zapomni się zarejestrować implementację B w kontenerze).

Alternatywą dla SL jest **Composition Root + Local Factory (Dependency Resolver)**

Composition Root = fragment kodu wykonywany zwykle na starcie aplikacji, odpowiedzialny za zdefiniowanie wszystkich zależności. W idealnej rzeczywistości, tylko w Composition Root pojawia się zależność do DI, a cała reszta aplikacji jest jej pozbawiona.

W praktyce – w aplikacji typu desktop, CR jest funkcja wywoływana z **Main** lub jej okolic, w aplikacji typu web to funkcja wywoływana w potoku z handlera zdarzenia **Application_Start** lub

jego okolic. W stosie aplikacyjnym jest to warstwa najwyższa.

Każde inne miejsce na konfigurację grafu zależności to już potknięcie projektowe.

Sam CR jest zbyt słaby żeby rozwiązać problem rozwiązywania zależności. Naiwne zastosowanie spowodowałoby konieczność wytworzenia **wszystkich instancji** obiektów ze wstrzykiwanymi zależnościami już na starcie aplikacji. To oczywiście niemożliwe.

W praktyce CR należy wesprzeć lokalną fabryką – fabryką z miękką zależnością do implementacji dostawcy obiektów. Taką fabrykę nazywa się **Local Factory** (lub **Dependency Resolver**). Różnica między LF a SL jest taka, że LF jest częścią klas domeny, w której występują zależności, natomiast SL jest częścią obcego świata, zewnętrzną zależnością.

Local Factory (Dependency Resolver) = fabryka odpowiedzialna za tworzenie instancji jednej lub wielu klas której funkcja fabrykująca nie ma gotowej implementacji. Zamiast tego, konkretna implementacja jest wstrzykiwana do fabryki z poziomu **Compositon Root**.

Z kolei fabryka jest jedynym legalnym z punktu widzenia API danego podsystemu sposobem wytwarzania instancji obiektów tej jednej lub kilku klas.

Obrazowo można powiedzieć, że w stosie aplikacyjnym cała logika od miejsca określenia fabryki w górę korzysta z tej fabryki do tworzenia instancji, ale konkretna implementacja jest „wstrzyknięta do fabryki” z samego wierzchu stosu aplikacyjnego, z poziomu CR.

Na Local Factory można patrzeć jak na lokalnego Service Locatora, który tym się różni od swojego „dużego” brata że rozwiązuje problem lokalnie, na potrzeby jednej/kilku klas z jakiegoś konkretnego podsystemu i jest częścią dziedziny (API) tego podsystemu, a nie częścią obcego API, wymuszającego zewnętrzne zależności (jak SL). W prawdziwej aplikacji takich Local Factories jest więc wiele – występują one wszędzie tam, gdzie pojawiają się miękkie zależności w ramach podsystemów.

Dzięki LF możliwe jest zbudowanie zbioru klas zadanej domeny, który to zbiór jako zestaw (assembly) / pakiet (package) nie ma żadnych zewnętrznych zależności (w szczególności – zależności do ramy DI). Z kolei zależności do ramy DI pojawiają się wyłącznie w CR. Możliwe jest jednak skonfigurowanie fabryki na dowolnego dostawcę, w tym takiego który nie używa DI tylko konstruuje instancje konkretnych typów (na przykład do testów).

Przykład:

<http://www.wiktorzychla.com/2016/01/di-factories-and-composition-root.html>

6 Literatura uzupełniająca

1. Dhanji R. Prasanna – *Dependency Injection* (2009, Java)
2. Mark Seemann – *Dependency Injection in .NET* (2012, C#) (źródło ilustracji i planu prezentacji)
3. Wiktor Zychla – DI, Factories and Composition Root Revisited, online: <http://www.wiktorzychla.com/2016/01/di-factories-and-composition-root.html>