

Uniwersytet Wrocławski  
Wydział Matematyki i Informatyki  
Kierunek: Informatyka

Adrian Mularczyk

**Stworzenie wydajnego wzorca  
wstrzykiwania zależności dla złożonych  
grafów zależności**

Praca wykonana pod kierunkiem dr. Wiktora Zychli

Wrocław, 2016

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>2</b>
1.1	Cel pracy . . . . .	2
<b>2</b>	<b>Wstrzykiwanie zależności</b>	<b>3</b>
<b>3</b>	<b>Implementacja</b>	<b>3</b>
3.1	Reflection.Emit . . . . .	5
3.2	Rozwiązanie 1 - PartialEmitFunction . . . . .	5
3.3	Rozwiązanie 2 - FullEmitFunction . . . . .	5
<b>4</b>	<b>Testy wydajnościowe</b>	<b>5</b>
<b>5</b>	<b>Podsumowanie</b>	<b>5</b>

# 1 Wstęp

## 1.1 Cel pracy

Wstrzykiwanie zależności jest wzorcem projektowym, który pozwala na tworzenie kodu o luźniejszych powiązaniach, łatwiejszego w testowaniu i modyfikacji. Najbardziej popularnymi implementacjami tego wzorca w języku C# są Autofac, StructureMap, Unity i Windsor, a najbardziej wydajnymi DryIoc, LightInject i SimpleInjector. Celem niniejszej pracy magisterskiej jest stworzenie wydajnej implementacji tego wzorca dla złożonych grafów zależności. Do tego celu zostanie wykorzystana funkcjonalności z przestrzeni nazw Reflection.Emit. W tej pracy zostaną przedstawione dwa rozwiązania.

## 2 Wstrzykiwanie zależności

Jest to zbiór zasad projektowania oprogramowania i wzorców, które pozwalają nam rozwijać luźno powiązany kod [1] (str. 4).

Jakiemu celowi ma służyć wstrzykiwanie zależności? Wstrzykiwanie zależności nie jest celem samym w sobie, raczej jest to środek do celu. Ostatecznie celem większości technik programowania jest dostarczenie jak najwydajniej działającego oprogramowania. Jednym z aspektów tego jest napisanie utrzymywalnego kodu.

O ile nie pisze się prototypu lub aplikacji, które nigdy nie mają kolejnych wersji (kończą się na wersji 1), to wkrótce będzie trzeba zająć się utrzymaniem i rozwijaniem istniejącego kodu. Aby być w stanie pracować wydajnie z takim kodem bazowym, musi on być jak najlepiej utrzymywalny.

Wstrzykiwanie zależności jest niczym więcej niż techniką, która umożliwia luźne powiązania, a luźne powiązania sprawiają, że kod jest rozszerzalny i łatwy w utrzymaniu. [1] (str. 5)

Wstrzykiwanie zależności może odbywać się na 3 sposoby:

- wstrzykiwanie przez konstruktor (główna i najbardziej popularna);
- wstrzykiwanie przez metodę;
- wstrzykiwanie przez właściwość.

## 3 Implementacja

Stworzyłem interfejs IContainer, który zawiera w sobie metody niezbędne w każdym kontenerze:

- IContainerMember RegisterType<T>() where T : class;
- IContainerMember RegisterType<TFrom, TTo>() where TTo : TFrom;
- IContainerMember RegisterType<T>(Func<object> objectFactory) where T : class;
- IContainerMember RegisterInstance<T>(T instance);
- T Resolve<T>(ResolveKind resolveKind);
- void BuildUp<T>(T instance, ResolveKind resolveKind).

Pierwsze 4 metody służą do rejestracji typów do kontenera. W pierwszej metodzie możemy zarejestrować zwykłe klasy. W drugiej interfejsy i klasy, które implementują dany interfejs. W trzeciej metodzie rejestrujemy klasę jako fabrykę obiektów - funkcję, która ma nam zwrócić pożądany obiekt. W czwartej możemy zarejestrować konkretną instancję danej klasy. W mojej implementacji kontenera każdy typ może być zarejestrowany tylko raz - ponowna rejestracja tego samego typu nadpisuje istniejącą rejestrację. Każda z tych czterech pierwszych metod zwraca interfejs IContainerMember, który umożliwia nam zarejestrowanie danej klasy lub interfejsu z określonym menadżerem czasu życia. Jest to po to, ponieważ dla różnych przypadków biznesowych możemy potrzebować, aby obiekt danego typu miał konkretny czas życia. Ten interfejs daje nam możliwość ustawienia czasu życia na:

- Singleton;
- Transient;
- PerThread;
- PerHttpContext;
- Custom; (własna implementacja interfejsu `IObjectLifetimeManager`).

W mojej implementacji kontenera każdy typ domyślnie ma czas życia `Transient`.

Interfejs `IObjectLifetimeManager` zawiera w sobie następujące metody:

- `Func<object> ObjectFactory get; set;`
- `object GetInstance()`.

Pierwsza z nich służy to ustawienia fabryki, która zwraca obiekt. Druga służy do zwracania obiektu przy użyciu fabryki. W zależności od konkretnego czasu życia, to obiekt zwracany z metody `GetInstance` może być zwsze ten sam, zwsze różny albo ten sam tylko dla określonych sytuacji (np. ten sam dla tego samego wątku albo ten sam dla tego samego żądania).

Metoda piąta - `Resolve`, to główna metoda. `Register` można nazwać sercem kontenera, a `Resolve` mózgiem. Odpowiada ona za stworzenie i zwrócenie obiektu odpowiedniego typu. W mojej pracy zaproponowałem dwa rozwiązania - `PartialEmitFunction` i `FullEmitFunction`, dlatego ta metoda jako parametr przyjmuje wartość enuma `ResolveKind`. Szósta metoda to taki dodatek - gdy mamy stworzony obiekt, ale nie jest w pełni uzupełniony, wtedy możemy go zbudować (używając odpowiednio `PartialEmitFunction` lub `FullEmitFunction`). Z tą metodą są powiązane bezpośrednio dwa pojęcia - wstrzykiwanie przez metodę i wstrzykiwanie przez właściwość. Do tego celu zostały stworzone dwa atrybuty:

- `DependencyMethod` (dla metod);
- `DependencyProperty` (dla właściwości).

Podczas operacji `BuildUp` uzupełniane są wywoływane wszystkie metody i właściwości, które mają te atrybuty. `BuildUp` jest również wykonywany podczas operacji `Resolve`. Warto tutaj odnotować, że ze względu na szczegóły implementacyjne tylko jedno z moich rozwiązań wspiera operację `BuildUp` - `PartialEmitFunction`. W `FullEmitFunction` ta funkcjonalność nie została zaimplementowana. Jest to spowodowane skomplikowaniem `FullEmitFunction` i małą potrzebą biznesową używania operacji `BuildUp`.

W aplikacji istnieje również atrybut `DependencyConstructor`. Można go użyć przy definicji konstruktora danej klasy. Obiekty danej klasy tworzy się przy użyciu konstruktora. Dana klasa może mieć kilka konstruktorów. W mojej implementacji stworzyłem logikę wyboru konstruktora przy pomocy którego ma zostać stworzony obiekt. Jeśli jest kilka konstruktorów, odpowiedni jest wybierany w następującej kolejności:

- Konstruktor z atrybutem `DependencyConstructor`;
- Konstruktor z największą liczbą parametrów.

Jeśli jest kilka konstruktorów z atrybutem `DependencyConstructor` albo kilka z największą liczbą parametrów, to rzucony jest wyjątek.

### **3.1 Reflection.Emit**

<Wyjaśnić jak działa Reflection.Emit i do czego zostanie użyte>

### **3.2 Rozwiązanie 1 - PartialEmitFunction**

<opis PartialEmitFunction>

### **3.3 Rozwiązanie 2 - FullEmitFunction**

<opis FullEmitFunction>

## **4 Testy wydajnościowe**

<wyniki testów i ich opis>

## **5 Podsumowanie**

<parę słów na koniec>

## Literatura

- [1] Dependency Injection in .NET, Mark Seemann