

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>3</b>
1.1	Cel pracy . . . . .	3
1.2	Układ pracy . . . . .	3
<b>2</b>	<b>Wstrzykiwanie zależności</b>	<b>4</b>
2.1	Wstęp . . . . .	4
2.1.1	Wstrzykiwanie przez konstruktor . . . . .	4
2.1.2	Wstrzykiwanie przez metodę . . . . .	5
2.1.3	Wstrzykiwanie przez właściwość . . . . .	5
2.2	Implementacje przemysłowe . . . . .	6
<b>3</b>	<b>Implementacja</b>	<b>7</b>
3.1	Środowisko pracy . . . . .	7
3.2	Wstęp . . . . .	7
3.2.1	Microsoft Intermediate Language . . . . .	7
3.2.2	Reflection.Emit . . . . .	8
3.3	Opis . . . . .	8
3.3.1	Register . . . . .	9
3.3.2	Resolve . . . . .	10
3.3.3	BuildUp . . . . .	10
3.4	Rozwiązanie . . . . .	11
3.4.1	Krok pierwszy . . . . .	12
3.4.2	Rozwiązanie 1 - PartialEmitFunction . . . . .	14
3.4.3	Rozwiązanie 2 - FullEmitFunction . . . . .	17
<b>4</b>	<b>Testy wydajnościowe</b>	<b>23</b>
4.0.1	Register as Singleton . . . . .	23
4.0.2	Register as Transient . . . . .	23

4.0.3	Register as TransientSingleton . . . . .	23
4.0.4	Register as PerThread . . . . .	24
4.0.5	Register as FactoryMethod . . . . .	24
4.1	Przypadek testowy A . . . . .	24
4.1.1	Opis . . . . .	24
4.1.2	Wyniki dla Singleton . . . . .	26
4.1.3	Wyniki dla Transient . . . . .	26
4.1.4	Wyniki dla TransientSingleton . . . . .	27
4.1.5	Wyniki dla PerThread . . . . .	28
4.2	Przypadek testowy B . . . . .	28
4.2.1	Opis . . . . .	28
4.2.2	Wyniki dla Singleton . . . . .	30
4.2.3	Wyniki dla Transient . . . . .	31
4.2.4	Wyniki dla TransientSingleton . . . . .	31
4.2.5	Wyniki dla PerThread . . . . .	32
4.3	Przypadek testowy C . . . . .	33
4.3.1	Opis . . . . .	33
4.3.2	Wyniki dla Singleton . . . . .	34
4.3.3	Wyniki dla Transient . . . . .	34
4.3.4	Wyniki dla TransientSingleton . . . . .	35
4.3.5	Wyniki dla PerThread . . . . .	35
4.4	Przypadek testowy D . . . . .	36
4.4.1	Opis . . . . .	36
4.4.2	Wyniki dla Singleton . . . . .	38
4.4.3	Wyniki dla Transient . . . . .	38
4.4.4	Wyniki dla TransientSingleton . . . . .	39
4.4.5	Wyniki dla PerThread . . . . .	39

# 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 Unity i Ninject. Celem niniejszej pracy magisterskiej jest stworzenie wydajnej implementacji tego wzorca dla złożonych grafów zależności. Do tego celu zostanie wykorzystana funkcjonalność z przestrzeni nazw `Reflection.Emit`. W tej pracy zostaną przedstawione dwa rozwiązania.

## 1.2 Układ pracy

Poza wstępem i podsumowaniem praca składa się jeszcze z trzech rozdziałów. W pierwszym znajduje się opis teoretyczny czym jest wstrzykiwanie zależności. Drugi rozdział opisuje moją implementację tego wzorca. Trzeci rozdział skupia się na testach wydajnościowych, w którym porównuję moją implementację z kilkoma najbardziej popularnym i kilkoma najszybszymi implementacjami.

## 2 Wstrzykiwanie zależności

### 2.1 Wstęp

Jest to zbiór zasad projektowania oprogramowania i wzorców, które pozwalają nam rozwijać luźno powiązany kod.

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] Wstrzykiwanie zależności może odbywać się na 3 sposoby:

- wstrzykiwanie przez konstruktor
- wstrzykiwanie przez metodę
- wstrzykiwanie przez właściwość

#### 2.1.1 Wstrzykiwanie przez konstruktor

```
public class SampleClass
{
    [DependencyConstructor]
    public SampleClass(EmptyClass emptyClass)
    {
        EmptyClass = emptyClass;
    }

    public EmptyClass EmptyClass { get; }
}
```

Jest to główny i najbardziej popularny sposób wstrzykiwania zależności. Niektóre klasy mają kilka konstruktorów i atrybut "DependencyConstructor" przydaje się wtedy do oznaczenia, który z nich ma zostać wybrany przy tworzeniu nowego obiektu. Jednakże nie zawsze jest on potrzebny. W większości przypadków klasy mają tylko jeden konstruktor, a także rozwiązania przemyślane mają logikę, która wybierze odpowiedni konstruktor (np. ten oznaczony atrybutem, albo ten co ma najwięcej parametrów, albo ten co ma najmniej parametrów).

### 2.1.2 Wstrzykiwanie przez metodę

```
public class SampleClass
{
    [DependencyMethod]
    public void SetEmptyClass(EmptyClass emptyClass)
    {
        EmptyClass = emptyClass;
    }

    public EmptyClass EmptyClass { get; private set; }
}
```

W przemysłowych rozwiązaniach to wstrzykiwanie z reguły odbywa się albo poprzez oznaczenie metody przez którą chcemy wstrzyknąć zależności odpowiednim atrybutem, albo przy rejestracji danej klasy definiujemy przez jakie metody chcemy wstrzyknąć zależności.

### 2.1.3 Wstrzykiwanie przez właściwość

```
public class SampleClass
{
    [DependencyProperty]
    public EmptyClass EmptyClass { get; set; }
}
```

Tutaj podobnie jak dla wstrzykiwania przez metodę w przemysłowych rozwiązaniach to wstrzykiwanie z reguły odbywa się albo poprzez oznaczenie właściwości przez którą chcemy wstrzyknąć zależności odpowiednim atrybutem, albo przy rejestracji danej klasy definiujemy przez jakie właściwości chcemy wstrzyknąć zależności.

## 2.2 Implementacje przemysłowe

Na rynku jest wiele implementacji wstrzykiwania zależności. Przedstawię tutaj kilka najbardziej popularnych (według ilości pobrań z NuGet) oraz kilka najszybszych (według rankingu na stronie: <http://www.palmmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison>). Dane zostały wzięte z dnia 21-02-2017. W nawiasie znajduje się wersja implementacji, która została użyta w testach (najnowsza na ten dzień).

Najbardziej popularne:

- Unity (4.0.1) - ponad 5.2 mln pobrań
- NInject (3.2.2) - ponad 4.0 mln pobrań
- Autofac (4.3.0) - ponad 3.7 mln pobrań
- StructureMap (4.4.3) - ponad 1.6 mln pobrań
- Windsor (3.4.0) - ponad 1.4 mln pobrań

Najszybsze:

- Grace (5.1.0)
- DryIoc (2.10.1)
- LightInject (5.0.1)
- SimpleInjector (3.3.2)

## 3 Implementacja

Kod źródłowy programu jest dostępnym w repozytorium pod adresem:

<https://github.com/amularczyk/NiquIoC>

Znajduje się tam również kod programu, który posłużył do wykonania testów wydajnościowych, a także ta praca napisana w języku LaTeX i wszystkie obrazki.

### 3.1 Środowisko pracy

Prac oraz wszystkie testy powstały na komputerze z parametrami:

- Intel Core i7-4720HQ (2.60GHz)
- 12 GB pamięci RAM
- Dysk SSD

Narzędzia użyte do stworzenia pracy i testów:

- System operacyjny Windows 10 Pro
- .Net Framework w wersji 4.6.1
- Visual Studio 2017 Community
- MSTest
- ReSharper
- dotCover
- Dia

### 3.2 Wstęp

Na początku chciałbym pokrótce opisać dwie rzeczy, które są istotne dla mojego rozwiązania. Pierwszą z nich jest Microsoft Intermediate Language, a drugą przestrzeń nazw Reflection.Emit.

#### 3.2.1 Microsoft Intermediate Language

Microsoft Intermediate Language - MSIL (w skrócie IL) to język pośredni do którego kod C# jest kompilowany. Język ten pozwala na komunikację między aplikacjami napisanymi na platformie .Net, a systemem operacyjnym. Jest on jądrem tej platformy.



### 3.2.2 Reflection.Emit

Przestrzeń nazw Reflection.Emit pozwala w języku C# na stworzenie ciągu operacji w języku IL, a następnie zapamiętaniu ciągu tych operacji jako delegat. Za każdym razem, gdy ten delegat zostanie wywołany, to wykona się ciąg wcześniejszych zdefiniowanych operacji IL.

## 3.3 Opis

Aplikacja składa się z 1 projektu i 8 projektów na potrzeby testów. Rozwiązanie jest skomplikowane i aby mieć pewność, że działa w pełni dobrze zostało stworzone ponad 1200 testów jednostkowych, a pokrycie kodu testami wynosi ponad 97%.

W wykonanej implementacji został stworzony interfejs IContainer, który definiuje operacje, jakie powinny się znaleźć w każdym kontenerze:

```
public interface IContainer
{
    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 cztery metody służą do rejestracji typów w kontenerze. Metoda piąta (Resolve) służy do tworzenia i zwracania obiektów wcześniejszych zarejestrowanych typów. Ostatnia metoda (BuildUp) do uzupełnienia istniejącej instancji obiektu z wykorzystaniem wstrzykiwania zależności przez metodę i właściwość - jest to metoda opcjonalna i nie każde przemysłowe rozwiązanie ją zawiera.

Poniżej znajduje się dokładniejszy opis każdej z metod.

### 3.3.1 Register

W pierwszej metodzie możemy zarejestrować zwykłe klasy. W drugiej interfejsy i klasy, które implementują dany interfejs lub klasy i klasy po nich dziedziczące. W trzeciej metodzie rejestrujemy klasę jako fabrykę obiektów - funkcję, która ma nam zwrócić pożądaną instancję danego typu. W czwartej natomiast możemy zarejestrować konkretną instancję danego typu.

W moim rozwiązaniu każdy typ może być zarejestrowany tylko raz - ponowna rejestracja tego samego typu nadpisuje istniejącą rejestrację.

Każda z tych czterech metod rejestracji zwraca interfejs `IContainerMember`, który umożliwia nam zarejestrowanie danego typu z określonym menadżerem czasu życia (czyli implementacją interfejsu `IObjectLifetimeManager`). Jest to po to, ponieważ dla różnych przypadków biznesowych możemy potrzebować, aby obiekt danego typu miał konkretny czas życia. Interfejs `IContainerMember` wygląda następująco:

```
public interface IContainerMember
{
    void AsSingleton();

    void AsTransient();

    void AsPerThread();

    void AsPerHttpContext();

    void AsCustomObjectLifetimeManager(IObjectLifetimeManager objectLifetimeManager);
}
```

Pierwsze cztery metody tego interfejsu to wbudowane implementacje interfejsu `IObjectLifetimeManager`. Piąta metoda dostarcza możliwość podania przez użytkownika jego własnej implementacji tego interfejsu. W moim rozwiązaniu każdy typ domyślnie ma czas życia `Transient`.

Wyjaśnienie rozdajów czasu życia:

- Singleton - za każdym razem zwracany jest ten sam obiekt
- Transient - za każdym razem zwracany jest nowy obiekt
- PerThread - wewnątrz danego wątku zwracany jest ten sam obiekt, ale dla innego wątku zwracany jest nowy (inny) obiekt
- PerHttpContext - wewnątrz danego żądania Http zwracany jest ten sam obiekt, ale dla innego żądania zwracany jest nowy (inny) obiekt

Interfejs `IObjectLifetimeManager` prezentuje się tak:

```
public interface IObjectLifetimeManager
{
    Func<object> ObjectFactory { get; set; }
    object GetInstance();
}
```

Właściwość `ObjectFactory` służy do ustawienia funkcji, która zwraca obiekt. Metoda `GetInstance` służy do pobrania obiektu.

W zależności od implementacji tego interfejsu, to obiekt zwracany z metody `GetInstance` może być zawsze taki sam, zawsze różny albo taki sam tylko dla określonych sytuacji (np. taki sam dla tego samego wątku albo tego samego żądania http).

### 3.3.2 Resolve

Metoda piąta - `Resolve`, jest to główna operacja. 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` (dzięki temu w przyszłości może być ona w łatwy sposób rozszerzona o kolejne rozwiązania).

### 3.3.3 BuildUp

Szósta metoda to taki dodatek - gdy mamy stworzony obiekt, ale nie jest w pełni uzupełniony, wtedy możemy go zbudować (używając odpowiedniej wartości z enuma `ResolveKind`). Z tą metodą są powiązane bezpośrednio dwa pojęcia - wstrzykiwanie przez metodę i wstrzykiwanie przez właściwość. Do tego celu w moim rozwiązaniu stworzyłem dwa atrybuty:

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

Podczas operacji `BuildUp` wywoływane są wszystkie metody i uzupełniane są wszystkie właściwości, które mają te atrybuty. Ta operacja jest również wykonywana podczas operacji `Resolve`.

Warto tutaj odnotować, że ze względu na szczegóły implementacyjne tylko jedno z moich rozwiązań wspiera operację `BuildUp` - jest to rozwiązanie `PartialEmitFunction`. W rozwiązaniu `FullEmitFunction` ta funkcjonalność nie została zaimplementowana. Jest

to spowodowane skomplikowaniem tego rozwiązania i małą potrzebą biznesową używania tej operacji. Jednakże w przyszłości istnieje możliwość dodania implementacji tej funkcjonalności.

W aplikacji istnieje również atrybut `DependencyConstructor`. Można go użyć przy definicji konstruktora danej klasy. Obiekt każdej klasy jest tworzony przy użyciu konstruktora. Klasa może mieć kilka konstruktorów. W moim rozwiązaniu stworzyłem logikę wyboru odpowiedniego konstruktora, przy pomocy którego ma zostać stworzony obiekt. Wygląda ona następująco:

- Jeśli jest jeden konstruktor, to go wybierz.
- Jeśli jest kilka konstruktorów, to odpowiedni wybierz w poniższej kolejności:
  1. Konstruktor z atrybutem `DependencyConstructor`
  2. Konstruktor z największą liczbą parametrów
- Jeśli jest kilka konstruktorów z atrybutem `DependencyConstructor` albo nie ma żadnego konstruktora z tym atrybutem i jest kilka z największą liczbą parametrów, to rzuć wyjątek.

### 3.4 Rozwiązanie

Stworzenie nowego obiektu zajmuje czas. Gdy graf zależności dla jakiegoś typu jest bardzo rozbudowany, to stworzenie obiektu takiego typu zajmuje dużo czasu. Proces ten można podzielić na trzy etapy:

- Uzyskanie informacji jakich typów obiekty są potrzebne do stworzenia danego obiektu.
- Stworzenie tych pomocniczych obiektów.
- Stworzenie docelowego obiektu.

Gdy mamy rozbudowane grafy zależności, to często zdarza się, że niektóre typy się powtarzają. Zatem pewne informacje możemy uzyskać raz, a następnie je zapamiętać. Aby implementacja wzorca wstrzykiwania zależności działała wydajnie dla złożonych grafów zależności, należy jak najwięcej informacji przechowywać w pamięci podręcznej i należy to robić mądrze.

W mojej implementacji stworzyłem dwie strategie, które realizują te założenia. Pierwszy krok jest taki sam dla obu rozwiązań (uzyskanie informacji jakich typów obiekty są potrzebne do stworzenia danego obiektu), natomiast kolejne kroki już się różnią. W pierwszym rozwiązaniu, które nazwałem `PartialEmitFunction` całym proces tworzenia nowego obiektu został rozbity na mniejsze części (docelowy obiekt jest tworzony

po kawałku). Każda taka osobna część jest zapisywana w pamięci podręcznej. W drugim rozwiązaniu w pamięci podręcznej jest zapisana tylko jedna operacja. Zawiera ona listę wszystkich kroków, które są niezbędne do stworzenia docelowego obiektu. Więc finalnie docelowy obiekt jest tworzony przy pomocy jednej operacji (kroki drugi i trzeci są połączone). To rozwiązanie nazwałem FullEmitFunction.

W obu rozwiązaniach do zapamiętania kroków potrzebnych do stworzenia obiektu danego typu wykorzystałem operacje z przestrzeni nazw Reflection.Emit.

### 3.4.1 Krok pierwszy

Na początku algorytmu znajdujemy odpowiedni konstruktor, przy pomocy którego ma zostać stworzony nowy obiekt. Jeśli robimy to poraz pierwszy dla danego typu, to informację o tym konstruktorze zapisujemy w pamięci podręcznej. Do tego celu została użyta struktura danych Dictionary, gdzie kluczem jest typ obiektu, a wartością obiekt klasy ContainerMember (w którym przechowujemy wszystkie zapamiętane informacje dla danego typu). Następnym krokiem, jest rekurencyjne wywołanie tej operacji dla wszystkich typów, których obiekty są niezbędne do stworzenia obiektu docelowego typu.

W tej operacji jest kilka wyjątków - są nimi typy zarejestrowane jako Instance albo FactoryObject. Dla pierwszego przypadku nie musimy uzyskiwać informacji o tym jak stworzyć obiekt takiego typu, ponieważ mamy już taki obiekt stworzony i go po prostu wykorzystamy. Dla drugiego przypadku obiekt tworzony jest przy użyciu wcześniej zdefiniowanej przez użytkownika funkcji, której wywołanie spowoduje stworzenie obiektu oczekiwanego typu.

Klasa ContainerMember prezentuje się następująco:

```
internal class ContainerMember : IContainerMemberPrivate
{
    public ContainerMember(IObjectLifetimeManager objectLifetimeManager) ...
    internal ConstructorInfo Constructor { get; set; }
    internal List<ParameterInfo> Parameters { get; set; }
    internal List<PropertyInfo> PropertiesInfo { get; set; }
    internal List<MethodInfo> MethodsInfo { get; set; }
    internal bool? IsCycleInConstructor { get; set; }
    internal bool ShouldCreateCache { get; set; }

    #region IContainerMemberPrivate
    public Type RegisteredType { get; set; }
    public Type ReturnType { get; set; }
    public IObjectLifetimeManager ObjectLifetimeManager { get; private set; }
    #endregion IContainerMemberPrivate

    IContainerMember
}
```

Ta klasa przechowuje:

- Informacje o konstruktor, przy pomocy którego należy utworzyć obiekt danego typu.
- Informacje o parametrach tego konstruktora.
- Informacje o właściwościach danego typu (na potrzeby operacji BuildUp).
- Informacje o metodach danego typu (na potrzeby operacji BuildUp).
- Czy występuje cykl w konstruktorze.
- Przy należy zapamiętać informacje dla danego typu (domyślnie tak, dla typów zarejestrowanych jako Instance albo ObjectFactory - nie).
- Zarejestrowany typ (ta sama wartość, co klucz ze słownika).
- Zwracany typ.
- Informacje o menadżerze cyklu życia.

### 3.4.2 Rozwiązanie 1 - PartialEmitFunction

Cały algorytm jest zawarty w metodzie Resolve. Sama ta metoda jest dość krótka, ale wywołuje ona kolejne metody (które już są dłuższe).

Docelowy obiekt jest tworzony przy pomocy funkcji. Na początku sprawdzamy, czy już wcześniej utworzyliśmy taką funkcję (jeśli tak, będzie on zapisany w klasie ContainerMember we właściwości ObjectLifetimeManager). Wywołanie funkcji kończy działanie algorytmu. Jeśli funkcja nie została jeszcze wcześniej utworzona, to ją tworzymy. Funkcja ta nie przyjmuje żadnych parametrów, a jej typem wynikowym jest obiekt. Ciało tej funkcji jest bardzo proste - po prostu ma ona wykonać metodę i zwrócić jej rezultat, którym jest nasz docelowy obiekt.

```
public object Resolve(ContainerMember containerMember,
    Action<object, ContainerMember> afterObjectCreate)
{
    if (containerMember.ObjectLifetimeManager.ObjectFactory == null)
    {
        containerMember.ObjectLifetimeManager.ObjectFactory =
            () => GetObject(containerMember, afterObjectCreate);
    }

    return containerMember.ObjectLifetimeManager.GetInstance();
}
```

Metoda GetObject jest trochę bardziej rozbudowana. Na początku pobieramy informacje o parametrach konstruktora danego typu. Następnie dla każdego z tych parametrów tworzymy obiekt (o docelowym typie) przy pomocy funkcji Resolve (opisanej wyżej). Korzystamy tutaj z rekurencji. Gdy już mamy stworzone obiekty dla każdego z parametrów konstruktora, to listę tych parametrów przekazujemy do metody CreateInstanceFunction, która zwróci nam instancję obiektu oczekiwanego typu. Na koniec wywołujemy callback afterObjectCreate i zwracamy nasz obiekt.

```
private object GetObject(ContainerMember containerMember,
    Action<object, ContainerMember> afterObjectCreate)
{
    var ctorParameters = containerMember.Parameters;
    var ctorParametersCount = ctorParameters.Count;

    var parameters = new object[ctorParametersCount];
    for (var i = 0; i < ctorParametersCount; i++)
    {
        var parameterContainerMember =
            _registeredTypesCache.GetValue(ctorParameters[i].ParameterType);
        parameters[i] = Resolve(parameterContainerMember, afterObjectCreate);
    }

    var obj = CreateInstanceFunction(containerMember, parameters);
    afterObjectCreate(obj, containerMember);

    return obj;
}
```

Na początku metody `CreateInstanceFunction` sprawdzamy, czy mamy utworzoną funkcję, która umie zwrócić obiekt docelowego typu. Jeśli tak, to przy pomocy tej funkcji tworzymy obiekt i go zwracamy. Ta funkcja jako argument przyjmuje listę obiektów w kolejności zgodnej z listą parametrów konstruktora. Jeśli nie, to przy pomocy metody `CreateObjectFunction` tworzymy taką funkcję, a następnie zapisujemy ją w pamięci podręcznej (również w strukturze `Dictionary`, której kluczem jest `type`, a wartością jest funkcja).

```
private object CreateInstanceFunction(ContainerMember containerMember, object[] parameters)
{
    if (!_createPartialEmitFunctionForConstructorCache.ContainsKey(
        containerMember.ReturnType))
    {
        var factoryMethod = CreateObjectFunction(containerMember);
        _createPartialEmitFunctionForConstructorCache.Add(
            containerMember.ReturnType, factoryMethod);
    }

    var obj =
        _createPartialEmitFunctionForConstructorCache[containerMember.ReturnType](
            parameters);

    return obj;
}
```

`CreateObjectFunction` jest najbardziej zaawansowaną metodą w tym algorytmie. To w niej korzystamy z metod z przestrzeni nazw `Reflection.Emit`.

Na początku pobieramy informacje o konstrktorze. Następnie tworzymy `DynamicMethod` i z niej pobieramy `ILGenerator`. W nim będziemy przechowywać listę kroków niezbędnych do utworzenia docelowego obiektu.

Dla każdego z parametrów konstruktora wykonujemy następujące operacje:

1. Dodaj do listy kroków operację, która umieści na szczycie stosu pierwszy parametr (będzie nim lista obiektów zgodna z parametrami konstruktora).
2. Dodaj do listy kroków operację, która umieści na szczycie stosu indeks parametru (który to jest parametr z kolei).
3. Dodaj do listy kroków operację, która zdejmie ze szczytu stosu listę i indeks, a umieści na jego szczycie element znajdujący się pod danym indeksem na liście.
4. Pobieramy typ parametru.
5. Dodaj do listy kroków operację, która rzutuje obiekt ze szczytu stosu na odpowiedni typ.

Po wykonaniu kroków z listy na stosie będziemy mieli wszystkie obiekty, które są wymagane przez konstrktor do utworzenia danego typu. Teraz więc do listy kroków dodajemy operację, która stworzy obiekty przy pomocy danego konstruktora i umieści



go na szczycie stosu. Na koniec dodajemy krok, który zwróci nam obiekt ze szczytu stosu.

Wszystkie kroki niezbędne do stworzenia nowego elementu są zapisane w zmiennej typu `DynamicMethod`. Jako ostatnią operację w metodzie `CreateObjectFunction`, ze zmiennej typu `DynamicMethod`, tworzymy i zwracamy delegata, który jako parametr będzie przyjmował tablicę obiektów (nasza lista obiektów zgodna z parametrami konstruktora), a zwracał obiekt (nasz docelowy obiekt).

```
private static Func<object[], object> CreateObjectFunction(ContainerMember containerMember)
{
    var ctor = containerMember.Constructor;
    var dm = new DynamicMethod($"Create_{ctor.DeclaringType?.FullName.Replace('.', '_')}",
        typeof(object), new[] {typeof(object[])}, true);
    var ilgen = dm.GetILGenerator();

    var parameters = ctor.GetParameters();
    for (var i = 0; i < parameters.Length; i++)
    {
        ilgen.Emit(OpCodes.Ldarg_0);
        EmitHelper.EmitIntOntoStack(ilgen, i);
        ilgen.Emit(OpCodes.Ldelem_Ref);
        var paramType = parameters[i].ParameterType;
        ilgen.Emit(
            paramType.IsValueType ? OpCodes.Unbox_Any : OpCodes.Castclass,
            paramType);
    }
    ilgen.Emit(OpCodes.Newobj, ctor);
    ilgen.Emit(OpCodes.Ret);

    return (Func<object[], object>) dm.CreateDelegate(typeof(Func<object[], object>));
}
```

### 3.4.3 Rozwiązanie 2 - FullEmitFunction

Tutaj podobnie jak dla PartialEmitFunction cały algorytm zawarty jest w metodzie Resolve. Wygląda ona identycznie jak w rozwiązaniu 1 - docelowy obiekt jest tworzony przy pomocy funkcji, która woła w siebie metodę GetObject. Jeśli funkcji nie ma w zapisanej w pamięci, to ją tworzymy i zapisujemy. Na końcu ją wywołujemy.

```
public object Resolve(ContainerMember containerMember,
    Action<object, ContainerMember> afterObjectCreate)
{
    if (containerMember.ObjectLifetimeManager.ObjectFactory == null)
    {
        containerMember.ObjectLifetimeManager.ObjectFactory =
            () => GetObject(containerMember, afterObjectCreate);
    }

    return containerMember.ObjectLifetimeManager.GetInstance();
}
```

Metoda GetObject wygląda trochę inaczej. Nie pobieramy tutaj żadnych dodatkowych informacji, tylko od razu wywołujemy metodę CreateInstanceFunction, która zwróci nam instancję obiektu oczekiwanego typu. Na koniec również wywołujemy callback afterObjectCreate i zwracamy nasz obiekt.

```
private object GetObject(ContainerMember containerMember,
    Action<object, ContainerMember> afterObjectCreate)
{
    var obj = CreateInstanceFunction(containerMember);
    afterObjectCreate(obj, containerMember);

    return obj;
}
```

CreateInstanceFunction ma jeden dodatkowy krok względem rozwiązania 1. Na początku również sprawdzamy, czy mamy utworzoną funkcję, która umie zwrócić obiekt docelowego typu. Jeśli nie, to przy pomocy metody CreateObjectFunction tworzymy taką funkcję, a następnie zapisujemy ją w pamięci podręcznej (również w strukturze Dictionary, której kluczem jest type, a wartością jest typ pomocniczy FullEmitFunction-Result, który ma tylko jedną właściwość - Result typu funkcja). Następnym krokiem jest zwalidowanie zapisanych danych dla typów przy pomocy metody ValidateTypesCache. Na końcu funkcji tworzymy obiekt i go zwracamy. Tutaj funkcja jako argument przyjmuje dwa słowniki. Pierwszy zawiera informacje o typie (kluczem jest typ, a wartością obiekt ContainerMember), a drugi informacje o indeksie typu (kluczem jest liczba oznaczająca hasz danego typu, a wartością typ).

Najpierw opiszę metodę ValidateTypesCache, ponieważ nie wywołuje ona w sobie żadnych innych metod.

W pierwszym kroku metody ValidateTypesCache sprawdzamy, czy od ostatniego zapisywania danych o typach w pamięci podręcznej jakiś typ został zarejestrowany w kontenerze (lub czy jest to pierwsze zapisanie tych danych). Jeśli tak, to tworzymy z

```

private object CreateInstanceFunction(ContainerMember containerMember)
{
    if (!_createFullEmitFunctionForConstructorCache.ContainsKey(containerMember.ReturnType))
    {
        var factoryMethod = CreateObjectFunction(containerMember, _registeredTypesCache);
        _createFullEmitFunctionForConstructorCache.Add(
            containerMember.ReturnType, factoryMethod);
    }

    ValidateTypesCache();

    var obj =
        _createFullEmitFunctionForConstructorCache[containerMember.ReturnType]
            .Result(_registeredTypesCache, _typesIndexCache);

    return obj;
}

```

zarejestrowanych typów tworzymy słownik, w którym kluczem jest hasz typu, a wartością typ. Potem aktualizujemy liczbę aktualnie zarejestrowanych typów.

```

private void ValidateTypesCache()
{
    var registeredTypesCacheCount = _registeredTypesCache.Count;
    if (_registeredTypesCacheCount != registeredTypesCacheCount)
    {
        _typesIndexCache =
            _registeredTypesCache.ToDictionary(k => k.Value.GetHashCode(), v => v.Key);
        _registeredTypesCacheCount = registeredTypesCacheCount;
    }
}

```

Metoda CreateObjectFunction jest dużo bardziej rozbudowana niż w rozwiązaniu 1. W niej również korzystamy z metod z przestrzeni nazw Reflection.Emit.

Na początku tworzymy DynamicMethod i z niej pobieramy ILGenerator, w którym będziemy przechowywać listę kroków niezbędnych do utworzenia docelowego obiektu. Dla każdego z parametrów konstruktora wywołujemy metodę CreateObjectFunction-Private, która uzupełni listę kroków o tworzenie pośrednich obiektów. To w tym miejscu jest główna różnica między oboma rozwiązaniami - w pierwszym rozwiązaniu w liście kroków nie przejmowaliśmy się utworzeniem pośrednich obiektów, ponieważ przychodziły one jako parametr. W tym rozwiązaniu lista kroków będzie również zawierać kroki do utworzenia wszystkich obiektów pośrednich (obiektów wymaganych przez konstruktory).

Dalsze operacje są takie same jak w rozwiązaniu 1 - do listy kroków dodajemy operację, która stworzy obiekty przy pomocy danego konstruktora i umieści go na szczycie stosu, a następnie dodajemy operację, który zwróci nam obiekt ze szczyci stosu.

Na końcu ze zmiennej typu DynamicMethod tworzymy delegata i go zwracamy. Delegat będzie przyjmował dwa parametry - oba typu Dictionary. Jeden z informacjami o typie, a drugi z informacjami o indeksie typu.

```

private FullEmitFunctionResult CreateObjectFunction(ContainerMember containerMember,
    IReadOnlyDictionary<Type, ContainerMember> registeredTypesCache)
{
    var dm = new DynamicMethod(
        $"Create_{containerMember.Constructor.DeclaringType?.FullName.Replace('.', '_')}",
        typeof(object),
        new[] {typeof(Dictionary<Type, ContainerMember>), typeof(Dictionary<int, Type>)},
        typeof(Container).Module, true);
    var ilgen = dm.GetILGenerator();

    foreach (var parameter in containerMember.Parameters)
    {
        CreateObjectFunctionPrivate(parameter.ParameterType, registeredTypesCache, ilgen,
            new Dictionary<Type, LocalBuilder>());
    }
    ilgen.Emit(OpCodes.Newobj, containerMember.Constructor);
    ilgen.Emit(OpCodes.Ret);

    var func = dm.CreateDelegate(
        typeof(Func<Dictionary<Type, ContainerMember>, Dictionary<int, Type>, object>));
    return new FullEmitFunctionResult
    {
        Result =
            (Func<Dictionary<Type, ContainerMember>, Dictionary<int, Type>, object>)func
    };
}

```

W metodzie `CreateObjectFunctionPrivate` mamy trzy przypadki. W pierwszych dwóch przypadkach pomijamy obiekty, które zostały zarejestrowane jako `Instance` albo `FactoryObject` (określa to parametr `ShouldCreateCache` z obiektu typu `ContainerMember`). Pierwszy przypadek, to gdy typu jest `Transient` (obiekt został zarejestrowany jako `Transient`).

Wtedy dla każdego z parametrów konstruktora wywołujemy rekurencyjnie tę metodę (`CreateObjectFunctionPrivate`), a na koniec do listy kroków dodajemy operację, która stworzy obiekty przy pomocy danego konstruktora i umieści go na szczycie stosu.

W drugim przypadku typ musi być singletonem (został on zarejestrowany jako `Singleton`, `PerThread` lub `PerHttpContext`).

Na początku sprawdzamy czy już wcześniej natrafiliśmy na ten typ. Jeśli tak, to z pamięci podręcznej pobieramy zmienną lokalną dla danego typu, a następnie do listy kroków dodajemy operację, która doda na szczyt stosu obiekt z tej zmiennej. Jeśli nie, to najpierw tworzymy nową zmienną lokalną dla danego typu i zapisuje ją w pamięci podręcznej (wykorzystujemy do tego słownik, gdzie kluczem jest typ, a wartością obiekt typu `LocalBuilder`). Następnie wywołujemy metodę `AddObjectCreatedByObjectLifetimeManager`. Na końcu dodajemy dwie operacje. Pierwsza z nich zdejmie obiekt ze szczytu stosu i zapisze go w zmiennej lokalnej, a druga umieści na szczycie stosu obiekt z tej zmiennej lokalnej. Jest to po to, ponieważ na szczycie stosu chcemy mieć dany obiekt, ale również chcemy zapamiętać sobie dany obiekt, aby nie trzeba było go tworzyć, jeśli będzie potrzebny poraz drugi (obiekt jest singletonem, więc za każdym razem będziemy chcieli mieć ten sam obiekt - w kontekście tworzenia danego

```

private void CreateObjectFunctionPrivate(Type type,
    IReadOnlyDictionary<Type, ContainerMember> registeredTypesCache, ILGenerator ilgen,
    IDictionary<Type, LocalBuilder> localSingletons)
{
    var containerMember = registeredTypesCache.GetValue(type);

    if (IsTransient(containerMember) && containerMember.ShouldCreateCache)
    {
        foreach (var parameter in containerMember.Parameters)
        {
            CreateObjectFunctionPrivate(parameter.ParameterType, registeredTypesCache,
                ilgen, localSingletons);
        }

        ilgen.Emit(OpCodes.Newobj, containerMember.Constructor);
    }
    else if (IsSingleton(containerMember) && containerMember.ShouldCreateCache)
    {
        if (localSingletons.ContainsKey(type))
        {
            ilgen.Emit(OpCodes.Ldloc, localSingletons[type]);
        }
        else
        {
            var localSingleton = ilgen.DeclareLocal(type);
            localSingletons.Add(type, localSingleton);

            AddObjectCreatedByObjectLifetimeManager(type, ilgen, containerMember);

            var localVariable = localSingletons[type];
            ilgen.Emit(OpCodes.Stloc, localVariable);
            ilgen.Emit(OpCodes.Ldloc, localVariable);
        }
    }
    else
    {
        AddObjectCreatedByObjectLifetimeManager(type, ilgen, containerMember);
    }
}

```

typu, czyli jednej operacji Resolve).

Metoda AddObjectCreatedByObjectLifetimeManager, za wyjątkiem pierwszej operacji jaką jest stworzenie zmiennej lokalnej typu Type, zawiera jedynie operacje, które dodają kolejne kroki do listy kroków:

1. Dodaj do listy kroków operację, która umieści na szczycie stosu drugi parametr (będzie nim słownik z indeksami typów).
2. Dodaj do listy kroków operację, która umieści na szczycie stosu hasz danego typu.
3. Dodaj do listy kroków operację, która pobierze ze szczytu stosu słownik i indeks, a umieści na jego szczycie element znajdujący się pod danym kluczem w słowniku.

```

private static bool IsTransient(ContainerMember containerMember)
{
    return containerMember.ObjectLifetimeManager is TransientObjectLifetimeManager;
}

private static bool IsSingleton(ContainerMember containerMember)
{
    return containerMember.ObjectLifetimeManager is SingletonObjectLifetimeManager ||
           containerMember.ObjectLifetimeManager is ThreadObjectLifetimeManager ||
           containerMember.ObjectLifetimeManager is HttpContextObjectLifetimeManager;
}

```

4. Dodaj do listy kroków operację, która zdejmie obiekt ze szczytu stosu i zapisze go w zmiennej lokalnej.
5. Dodaj do listy kroków operację, która umieści na szczycie stosu pierwszy parametr (będzie nim słownik z informacjami o typach).
6. Dodaj do listy kroków operację, która umieści na szczycie stosu obiekt ze zmiennej lokalnej.
7. Dodaj do listy kroków operację, która zdejmie ze szczytu stosu słownik i typ, a umieści na jego szczycie element znajdujący się pod danym kluczem w słowniku.
8. Dodaj do listy kroków operację, która zdejmie obiekt ze szczytu stosu (będzie to obiekt typu ContainerMember), wywoła na nim metodę, która zwróci menadżer czasu życia obiektu i na szczycie stosu umieści rezultat tej metody.
9. Dodaj do listy kroków operację, która zdejmie obiekt ze szczytu stosu (będzie to obiekt typu IObjectLifeTimeManager), wywoła na nim metodę, która zwróci instancję obiektu i na szczycie stosu umieści rezultat tej metody.
10. Dodaj do listy kroków operację, która rzuci obiekt ze szczytu stosu na odpowiedni typ.

Po wykonaniu tych 10 operacji na szczycie stosu znajdzie się obiekt danego typu utworzony przy pomocy menadżera czasu życia.

Trzeci przypadek zachodzi gdy typ został zarejestrowany jako Instance, FactoryObject lub przy pomocy własnego menadżera czasu życia (implementacji interfejsu IObjectLifetimeManager).

W tej sytuacji wywołujemy po prostu metodę AddObjectCreatedByObjectLifetimeManager.

```

private void AddObjectCreatedByObjectLifetimeManager(Type type, ILGenerator ilgen,
    ContainerMember containerMember)
{
    if (containerMember.ObjectLifetimeManager.ObjectFactory == null)
    {
        containerMember.ObjectLifetimeManager.ObjectFactory =
            () => CreateInstanceFunction(containerMember);
    }

    var localTypeVariable = ilgen.DeclareLocal(typeof(Type));
    ilgen.Emit(OpCodes.Ldarg_1);
    EmitHelper.EmitIntOntoStack(ilgen, containerMember.GetHashCode());
    ilgen.Emit(OpCodes.Call, typeof(Dictionary<int, Type>).GetMethod("get_Item"));
    ilgen.Emit(OpCodes.Stloc, localTypeVariable);
    ilgen.Emit(OpCodes.Ldarg_0);
    ilgen.Emit(OpCodes.Ldloc, localTypeVariable);
    ilgen.Emit(OpCodes.Call,
        typeof(Dictionary<Type, ContainerMember>).GetMethod("get_Item"));
    ilgen.Emit(OpCodes.Call,
        typeof(ContainerMember).GetMethod("get_ObjectLifetimeManager", Type.EmptyTypes));
    ilgen.Emit(OpCodes.Call,
        typeof(IObjectLifetimeManager).GetMethod("GetInstance", Type.EmptyTypes));
    ilgen.Emit(type.IsValueType ? OpCodes.Unbox_Any : OpCodes.Castclass, type);
}

```



## 4 Testy wydajnościowe

Do przeprowadzania testów wydajnościowych stworzyłem osobną aplikację w której utworzyłem 4 przypadki testowe:

- Przypadek testowy A,
- Przypadek testowy B,
- Przypadek testowy C,
- Przypadek testowy D.

Dla każdego z przypadków sprawdzany jest czas wykonania operacji "Resolve" dla różnych rodzajów rejestracji (operacji Register). Testy zostały wykonane dla następujących wariantów rejestracji:

- Register as Singleton,
- Register as Transient,
- Register as TransientSingleton,
- Register as PerThread (dla niektórych przemysłowych rozwiązań - PerScope),
- **Register as FactoryMethod.**

Każdy z testów dla każdego rozwiązania był uruchamiany w osobnym procesie. Każdy test był uruchamiany 10 razy, a w wynikach zostały przedstawione następujące czasy: minimalny, maksymalny i średni.

### 4.0.1 Register as Singleton

Każdy typ jest zarejestrowany jako "Singleton", czyli obiekt jest tworzony raz, a następnie cały czas zwracany.

### 4.0.2 Register as Transient

Każdy typ jest zarejestrowany jako "Transient", czyli za każdym razem jest tworzony nowy obiekt.

### 4.0.3 Register as TransientSingleton

Każdy typ jest zarejestrowany jako "Transient" za wyjątkiem typów, które mają konstruktor bezparametrowy - są one zarejestrowane jako "Singleton".



#### 4.0.4 Register as PerThread

Każdy typ jest zarejestrowany jako "PerThread", czyli obiekt jest tworzony raz dla każdego wątku, a następnie w obrębie tego wątku cały czas zwracany.

#### 4.0.5 Register as FactoryMethod

Typ główny jest zarejestrowany jako "FactoryMethod", czyli jako funkcja, która zwraca nam obiekt danego typu.

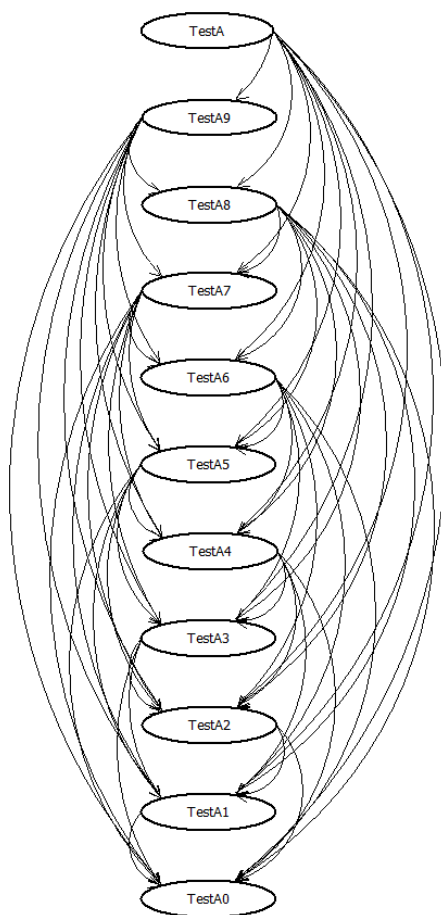
### 4.1 Przypadek testowy A

#### 4.1.1 Opis

W tym teście mamy zdefiniowanych 11 typów. Każdy z nich przyjmuje w konstruktorze o jeden parametr mniej niż typ poprzedni (czyli przyjmują one kolejno od 10 do 0 parametrów w konstruktorze). Typem głównym, a zarazem typem o największej liczbie parametrów, jest typ "TestA". Przyjmuje on w konstruktorze 10 parametrów następujących typów: "TestA0", "TestA1", "TestA2", "TestA3", "TestA4", "TestA5", "TestA6", "TestA7", "TestA8", "TestA9". Każdy z tych 10 typów w konstruktorze przyjmuje tyle parametrów, jaki ma numer w nazwie (czyli obiekt typu "TestA0" ma konstruktora bezparametrowy, obiekt typu "TestA1" ma konstruktor z jednym parametrem; i tak dalej aż do typu "TestA9", który ma konstruktor z dziewięcioma parametrami). Wszystkie typy jako parametry w konstruktorze przyjmuje obiekty typów z niższymi numerkami (czyli obiekt typu "TestA1" w konstruktorze przyjmuje obiekt typu "TestA0", obiekt typu "TestA2" przyjmuje w konstruktorze obiekty typów "TestA0" i "TestA1"; i tak dalej aż do typu "TestA9", który w konstruktorze przyjmuje obiekty z typami od "TestA0" do "TestA8"). Graf zależności poszczególnych typów został przedstawiony na Rys. 1.

Łatwo wywnioskować, że tworząc obiekty poszczególnych typów ilość tworzonych obiektów rośnie dwukrotnie:

- TestA0 - 1 obiekt,
- TestA1 - 2 obiekty (obiekt typu TestA1 i obiekt typu TestA0),
- TestA2 - 4 obiekty (obiekt typu TestA2, obiekt typu TestA1 - 2 obiekty, obiekt typu TestA0 - 1 obiekt),
- TestA3 - 8 obiektów (obiekt typu TestA3, obiekt typu TestA2 - 4 obiekty, obiekt typu TestA1 - 2 obiekty, obiekt typu TestA0 - 1 obiekt),
- TestA4 - 16 obiektów,



Rys. 1: Graf zależności dla testu A.

- TestA5 - 32 obiektów,
- TestA6 - 64 obiektów,
- TestA7 - 128 obiektów,
- TestA8 - 256 obiektów,
- TestA9 - 512 obiektów,
- TestA - 1 024 obiektów.

Zatem tworząc nasz główny obiekt typu TestA, tworzymy: 1 obiekt typu TestA, 1 obiekt typu TestA9, 2 obiekty typu TestA8, 4 obiekty typu TestA7, 8 obiektów typu TestA6, 16 obiektów typu TestA5, 32 obiektów typu TestA4, 64 obiektów typu TestA3, 128 obiektów typu TestA2, 256 obiektów typu TestA1 i 512 obiektów typu TestA0 - co w sumie daje 1024 obiekty.

#### 4.1.2 Wyniki dla Singleton

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	0	0	0	0	0	0	0	0	0	0	0	0
DryLoc	2	2	2	2	2	2	2	2	2	2	2	2
Grace	2	2	2	2	2	2	2	2	2	2	2	2
LightInject	2	2	2	2	2	2	2	2	2	2	2	2
Ninject	3	3	3	3	3	3	3	3	3	6	6	6
NiquIoCPartial	1	1	1	1	1	1	1	1	1	1	1	1
NiquIoCFull	2	2	2	2	2	2	2	2	2	2	3	2
SimpleInjector	2	2	2	2	2	2	2	2	2	2	2	2
StructureMap	9	10	9	9	9	9	9	9	9	10	10	10
Unity	7	7	7	7	7	7	7	7	7	8	8	8
Windsor	0	0	0	0	0	0	0	0	0	0	0	0

Dla tego testu najlepiej poradziły sobie dwa z najpopularniejszych rozwiązania - Windsor i Autofac. Rozwiązanie NiquIoCPartial uplasowało się na 3 miejscu. Najślabiej poradził sobie pozostałe najpopularniejsze rozwiązania: StructureMap, Unity oraz Ninject. Pozostałe rozwiązania miały zbliżone czasy.

#### 4.1.3 Wyniki dla Transient

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	0	0	0	6	7	6	59	63	60	593	635	602
DryLoc	14	14	14	14	15	14	16	16	16	29	29	29
Grace	15	15	15	15	16	15	18	19	18	36	37	36
LightInject	10	10	10	10	10	10	11	12	11	19	20	19
Ninject	10	11	10	86	91	87	843	1089	875	8585	9298	8759
NiquIoCPartial	1	1	1	3	3	3	19	20	19	170	173	171
NiquIoCFull	8	8	8	8	8	8	9	10	9	18	19	18
SimpleInjector	13	14	13	13	14	13	15	15	15	28	29	29
StructureMap	10	10	10	13	13	13	53	55	54	415	445	420
Unity	8	8	8	15	15	15	84	86	84	762	765	764
Windsor	1	1	1	16	18	16	151	156	152	1504	1652	1533

W tym teście jest już spora rozbieżność czasów. Gdy mamy tylko 1 operację najlepiej radzi sobie Autofac, a zaraz za nim NiquIoCPartial i Windsor. Najślabiej natomiast SimpleInjector, DryLoc i Grace. Pozostałe rozwiązania wypadły przeciętnie i bliżej im było do czasów najgorszych, niż najlepszych.

Dla 10 operacji sytuacja zaczyna się lekko zmieniać. Tym razem znacząco najlepiej radzi

sobie NiquIoCPartial (ponad dwa razy lepiej niż drugi Autofac). Kolejne miejsca należą do NiquIoCFull i LightInject. Pozostałe rozwiązania miały podobne, trochę słabsze czasy. Wyjątkiem jest jedynie Ninject, który poradził sobie najgorzej i jego czas jest ponad 5 razy większy niż dla rozwiązania z przedostatnim czasem (Windsor).

Przy 100 operacji na prowadzenie wysunęły się mniej popularne rozwiązania. Na pierwszym miejscu jest NiquIoCFull, a kolejne miejsca to LightInject i SimpleInjector. Do grona najsłabszych (Unity, Windsor, Ninject) w tym przypadku dołączył również Autofac, który z drugiego miejsca spadł na ósme. StructureMap miał czasy tylko trochę lepszy niż Autofac. DryLoc, Grace i NiquIoCPartial wypadły akceptowalnie - poradziły sobie trochę gorzej niż trzeci SimpleInjector.

Już dla 1000 operacji od czołówki oddalił się NiquIoCPartial. Na pierwszym miejscu wciąż pozostaje NiquIoCFull, a zaraz za nim LightInject i SimpleInjector. DryLoc i Grace wciąż trochę słabiej, ale wciąż zadowalająco. Pozostałe rozwiązania mają czasy od kilku do kilkunastu razy gorsze.

Warto tutaj zaznaczyć, że najmniejszy wzrost czasu miały kolejno rozwiązania: LightInject, NiquIoCFull, SimpleInjector, DryLoc oraz Grace.

#### 4.1.4 Wyniki dla TransientSingleton

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	0	0	0	6	6	6	54	56	54	540	566	544
DryLoc	18	18	18	19	19	19	20	20	20	31	32	31
Grace	9	9	9	9	10	9	10	11	10	18	18	18
LightInject	14	14	14	14	14	14	14	14	14	20	20	20
Ninject	9	10	10	67	73	69	637	678	652	6426	6810	6544
NiquIoCPartial	1	1	1	2	2	2	14	14	14	118	120	118
NiquIoCFull	5	5	5	5	6	5	6	6	6	10	11	10
SimpleInjector	9	9	9	9	10	9	10	10	10	17	17	17
StructureMap	9	9	9	12	13	12	35	38	36	252	287	257
Unity	8	8	8	14	14	14	72	72	72	643	650	646
Windsor	1	1	1	12	12	12	112	129	117	1108	1386	1151

Przy tym teście dla 1 operacji również najlepiej poradziły sobie Autofac, NiquIoCPartial i Windsor. Trochę gorzej NiquIoCFull. Czasy dla Unity, SimpleInjector, Grace, StructureMap i Ninject są zbliżone, jednak duże. Najgorzej wypadły natomiast LightInject i DryLoc.

Gdy mamy 10 operacji na drugie miejsce wskoczył NiquIoCFull. Na pierwsze przesunął się NiquIoCPartial, a Autofac spadł na miejsce trzecie. Kolejne miejsca zajmują SimpleInjector i Grace. Trochę gorzej poradził sobie Windsor, który z podium spadł na 6 miejsce. Zaraz za nim jest StructureMap, Unity i LightInject. DryLoc nadal słabo, a ostatnie miejsce zajął Ninject. Dla 100 operacji na pierwszym miejscu znalazł się NiquIoCFull. NiquIoCPartial spadł poza podium i ma czasy zbliżone do LightInject. Na

podium znaleźli się za to SimpleInjector i Grace. DryIoc ma jeszcze akceptowalny czas. Pozostałe rozwiązań bardzo słabo, a ranking zamykają Unity, Windsor i Ninject. Test dla 1000 operacji nie spowodował żadnych zmian z rankingu. Jedynie różnica czasów pomiędzy pierwszą 5, a pozostały miejscami znacząco się zwiększyła. Dla pierwszej piątki czasy zwrosły około dwukrotkie, a dla pozostały miejsc ponad siedmiokrotnie.

#### 4.1.5 Wyniki dla PerThread

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	0	0	0	0	0	0	0	0	0	0	0	0
DryIoc	73	81	76	72	78	75	74	79	76	74	83	77
Grace	4	4	4	4	4	4	4	4	4	4	5	4
LightInject	51	52	51	51	52	51	51	52	51	51	52	51
Ninject	3	3	3	3	3	3	3	3	3	6	6	6
NiquIoCPartial	1	1	1	1	1	1	1	1	1	1	1	1
NiquIoCFull	2	2	2	2	2	2	2	2	2	2	2	2
SimpleInjector	7	8	8	7	8	8	7	8	8	8	8	8
StructureMap	9	9	9	9	9	9	9	9	9	10	10	10
Unity	7	7	7	7	8	7	7	7	7	8	8	8
Windsor	0	0	0	0	0	0	0	0	0	0	0	0

Czasy dla tego przypadku powinny być zbliżone do czasów dla Singleton, ponieważ wszystko było uruchamiane w jednym wątku. Niestety część rozwiązań sobie z tym nie poradziła. Czasy zbliżone do czasów dla Singletona posiadają: Autofac, Grace, Ninject, NiquIoCPartial, NiquIoCFull, StructureMap, Unity i Windsor, a dla DryIoC, LightInject i SimpleInjector czasy są od kilka do nawet kilkadziesiąt razy większe.

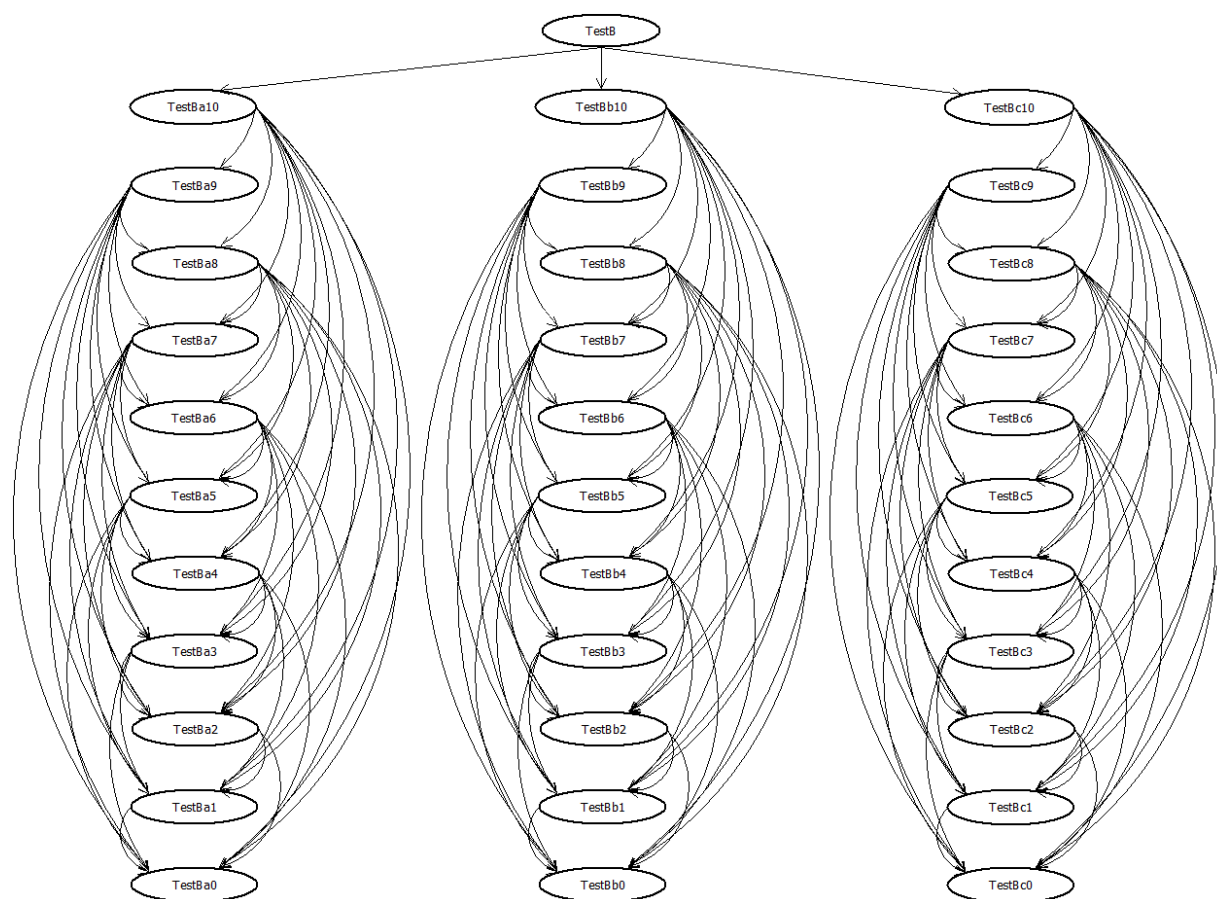
Jeśli chodzi o najlepsze rozwiązania, to wyglądają one tak samo jak dla Singleton. Pierwsze trzy miejsca to Windsor, Aurofac i NiquIoCPartial. Najślabiej poradziły sobie LightInject i DryIoc. W drugiej połowie rankingu znalazły się również Unity, SimpleInjector i StructureMap, a tuż za czołówką NiquIoCFull, Grace i Ninject.

## 4.2 Przypadek testowy B

### 4.2.1 Opis

Ten test jest bardzo podobny do przypadku testowego A, tylko dochodzi nam 1 dodatkowy poziom, który wygląda trochę inaczej. W głównym obiekcie "TestB" konstruktor przyjmuje 3 parametry następujących typów: "TestBa10", "TestBb10", "TestBc10". Każdy z tych 3 typów odpowiada typowi "TestA", więc przyjmuje on w konstruktorze 10 parametrów. Dla "TestBa10" są to parametry typów od "TestBa0" do "TestBa9", dla

"TestBb10" są to parametry typów od "TestBb0" do "TestBb9", a dla "TestBc10" są to parametry typów od "TestBc0" do "TestBc9". Zależności tych typów wyglądają tak samo, jak dla typów z przypadku testowego A - rys. 2 przedstawia te zależności.



Rys. 2: Graf zależności dla testu B.

Łatwo wywnioskować, że tworząc obiekty poszczególnych typów ilość tworzonych obiektów rośnie dwukrotnie (tak jak dla testu A):

- TestBa0 - 1 obiekt,
- TestBa1 - 2 obiekty (obiekt typu TestBa1 i obiekt typu TestBa0),
- TestBa2 - 4 obiekty (obiekt typu TestBa2, obiekt typu TestBa1 - 2 obiekty, obiekt typu TestBa0 - 1 obiekt),
- TestBa3 - 8 obiektów (obiekt typu TestBa3, obiekt typu TestBa2 - 4 obiekty, obiekt typu TestBa1 - 2 obiekty, obiekt typu TestBa0 - 1 obiekt),
- TestBa4 - 16 obiektów,
- TestBa5 - 32 obiektów,

- TestBa6 - 64 obiektów,
- TestBa7 - 128 obiektów,
- TestBa8 - 256 obiektów,
- TestBa9 - 512 obiektów,
- TestBa10 - 1 024 obiektów,
- ... (dla typów od TestBb0 do TestBb10 i od TestBc0 do TestBc10 sytuacja wygląda dokładnie tak samo jak dla typów od TestBa0 do TestBa10),
- TestB - 3 073 obiektów.

Zatem tworząc obiekt typu TestB, tworzymy: 1 obiekt typu TestB, 1 obiekt typu TestBa10, TestBb10 i TestBc10, 1 obiekt typu TestBa9, TestBb9 i TestBc9, 2 obiekty typu TestBa8, TestBb8 i TestBc8, 4 obiekty typu TestBa7, TestBb7 i TestBc7, 8 obiektów typu TestBa6, TestBb6 i TestBc6, 16 obiektów typu TestBa5, TestBb5 i TestBc5, 32 obiektów typu TestBa4, TestBb4 i TestBc4, 64 obiektów typu TestBa3, TestBb3 i TestBc3, 128 obiektów typu TestBa2, TestBb2 i TestBc2, 256 obiektów typu TestBa1, TestBb1 i TestBc1, 512 obiektów typu TestBa0, TestBb0 i TestBc0 - co daje w sumie 3 073 obiektów.

#### 4.2.2 Wyniki dla Singleton

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	0	0	0	0	0	0	0	0	0	0	0	0
DryLoc	8	9	8	8	9	9	8	9	9	8	9	8
Grace	8	8	8	8	8	8	8	8	8	8	8	8
LightInject	6	7	6	6	7	6	6	6	6	6	7	6
Ninject	9	9	9	9	9	9	9	10	9	12	13	12
NiquIoCPartial	4	4	4	4	4	4	4	4	4	4	4	4
NiquIoCFull	8	8	8	8	8	8	8	8	8	8	8	8
SimpleInjector	6	6	6	6	6	6	6	6	6	6	6	6
StructureMap	30	30	30	30	30	30	30	30	30	32	33	33
Unity	23	23	23	23	23	23	23	23	23	24	24	24
Windsor	0	0	0	0	0	0	0	0	0	0	0	0

Dla tego przypadku testowego również najlepiej poradziły sobie najpopularniejsze rozwiązania - Autofac i Windsor. NiquIoCPartial uplasował się tutaj również na 3 miejscu, jednak z czasami gorszymi niż dla testu A. Koniec rankingu jest taki sam, czyli StructureMap, Unity oraz Ninject. Tutaj też pozostałe rozwiązania miały zbliżone do siebie czasy, ale ponad 3 razy gorsze niż dla testu A.

### 4.2.3 Wyniki dla Transient

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	2	2	2	21	23	22	194	218	204	1972	2043	2008
DryLoc	44	46	45	45	46	45	52	55	53	100	104	102
Grace	51	52	51	52	53	52	58	59	58	123	126	124
LightInject	32	32	32	32	33	32	37	39	37	71	72	72
Ninject	35	42	36	271	297	277	2697	3014	2787	27790	28442	28038
NiquIoCPartial	5	5	5	10	10	10	57	58	58	522	529	524
NiquIoCFull	26	32	27	26	27	26	32	33	32	66	67	66
SimpleInjector	40	40	40	40	41	41	46	47	47	94	95	94
StructureMap	33	33	33	44	45	45	162	165	163	1296	1364	1311
Unity	27	28	28	48	50	49	257	265	258	2324	2334	2329
Windsor	6	6	6	48	64	50	466	498	474	4627	4893	4702

W tym teście sytuacja wygląda niemal identycznie jak w teście A, z tą różnicą, że czasy są około 3 razy gorsze.

Dla 1 operację najlepiej radzi sobie Autofac, a tuż za nim NiquIoCPartial i Windsor. Ostatnie trzy miejsca to: SimpleInjector, DryLoc i Grace. Pozostałe rozwiązania wypadły słabo i jeszcze bliżej im do czasów najgorszych.

Gdy mamy 10 operacji najlepiej radzi sobie NiquIoCPartial (również ponad dwa razy lepiej niż drugi Autofac). Kolejne miejsca należą do NiquIoCFull i LightInject. Tutaj również Ninject ma ponad 5 razy gorszy czas niż miejsce przedostatnie, jednakże zajmuje je Grace, a nie Windsor, który jest miejsce wyżej. Tylko trochę lepiej od Windsor poradziły sobie StructureMap, DryLoc i Unity.

Przy 100 operacji pierwsze trzy miejsca zajmują NiquIoCFull, LightInject i SimpleInjector. Tutaj nic się nie zmieniło w stosunku do testu A. Grace z miejsca przedostatniego dla 10 operacji, awansował na miejsce szóste i jego czasy w tym przypadku wyglądają zadowalająco. Trochę lepiej od niego poradziły sobie DryLoc i NiquIoCPartial. Grono najsłabszych to: StructureMap, Autofac, Unity, Windsor i Ninject.

Jak można się było spodziewać dla 1000 operacji sytuacja wygląda identycznie jak dla testu A - od czołówki oddalił się NiquIoCPartial, a na pierwszym miejscu wciąż pozostaje NiquIoCFull. LightInject i SimpleInjector są zaraz za nim. DryLoc i Grace słabiej, ale wciąż akceptowalnie. Pozostałe rozwiązania mają czasy o rząd lub dwa większe, niż NiquIoCPartial..

Tutaj również najmniejszy wzrost czasu miały rozwiązania: LightInject, NiquIoCFull, SimpleInjector, DryLoc oraz Grace.

### 4.2.4 Wyniki dla TransientSingleton



Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	1	1	1	18	19	18	175	179	176	1732	1806	1751
DryLoc	56	56	56	57	57	57	63	64	63	107	108	108
Grace	30	30	30	30	30	30	34	35	34	67	68	68
LightInject	47	48	48	48	49	48	50	52	51	80	80	80
Ninject	29	31	30	214	220	217	2016	2102	2045	20354	21113	20622
NiquIoCPartial	4	5	5	8	8	8	42	42	42	369	371	369
NiquIoCFull	16	16	16	16	16	16	18	20	18	37	38	37
SimpleInjector	27	28	28	28	28	28	32	33	32	60	62	61
StructureMap	32	33	32	39	40	39	111	118	113	795	805	800
Unity	27	28	28	46	47	46	225	228	227	2010	2025	2018
Windsor	3	4	3	35	36	35	338	395	348	3368	3589	3416

«opis»

#### 4.2.5 Wyniki dla PerThread

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	0	0	0	0	0	0	0	0	0	0	0	0
DryLoc	247	260	252	247	257	250	247	271	255	256	272	263
Grace	13	13	13	13	13	13	13	13	13	13	13	13
LightInject	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Ninject	9	9	9	9	9	9	9	10	9	12	13	12
NiquIoCPartial	4	4	4	4	4	4	4	4	4	4	4	4
NiquIoCFull	8	8	8	8	8	8	8	8	8	8	8	8
SimpleInjector	23	23	23	23	23	23	23	23	23	23	23	23
StructureMap	30	30	30	30	30	30	30	30	30	32	33	33
Unity	23	23	23	23	23	23	23	23	23	24	24	24
Windsor	0	0	0	0	0	0	0	0	0	0	0	0

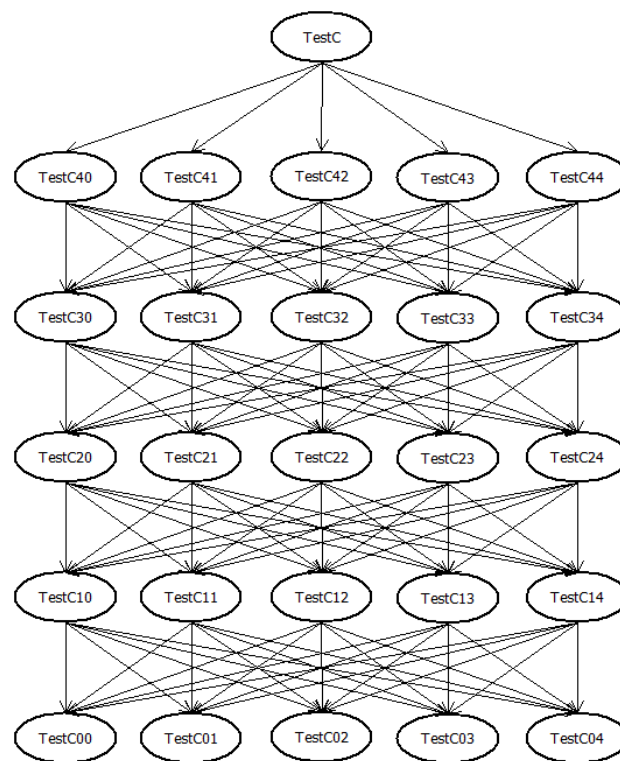
Tutaj również nie wszystkie rozwiązania mają czasy zbliżone do Singleton. Są to te same rozwiązania co dla przypadku testowego B, czyli: DryLoc, Grace, LightInject, SimpleInjector.

Tak samo jak w przypadku Singleton czołówka wygląda następująco: Autofac, Windsor i NiquIoCPartial. Kolejne miejsca to NiquIoCFull, Ninject i Grace. Najslabiej poradził sobie rozwiązania LightInject i DryLoc.

## 4.3 Przypadek testowy C

### 4.3.1 Opis

W tym teście mamy zdefiniowanych 26 typów. 5 z tych typów ma konstruktor bezparametrowy, a pozostałe 21 ma konstruktor z pięcioma parametrami. Typem głównym jest typ "TestC". Obiekt tego typu w konstruktorze przyjmuje 5 innych obiektów, kolejno następujących typów: "TestC40", "TestC41", "TestC42", "TestC43" i "TestC44". Każdy z tych pięciu typów ma taki sam konstruktor - przyjmuje w nim 5 obiektów o typach z pierwszym numerkiem o 1 mniejszym (czyli przyjmują w konstruktorze obiekty typów od "TestC30" do "TestC34"). Dla typów od "TestC30" do "TestC34" zasad z konstruktorami wygląda tak samo. Na końcu dochodzimy do typów od "TestC00" do "TestC04", które mają konstruktor bezparametrowy. Rys. 3 przedstawia graf zależności typów dla tego przypadku testowego.



Rys. 3: Graf zależności dla testu C.

Łatwo wywnioskować, że tworząc obiekty poszczególnych typów ilość tworzonych obiektów rośnie ponad dziesięciokrotnie:

- typy od TestC00 do TestC04 - 1 obiekt,
- typy od TestC10 do TestC14 - 6 obiektów (obiekt danego typu plus 5 obiektów typów od TestC00 do TestC04),

- typy od TestC20 do TestC24 - 31 obiektów (obiekt danego typu plus 5 obiektów typów od TestC10 do TestC14),
- typy od TestC30 do TestC34 - 156 obiektów,
- typy od TestC40 do TestC44 - 781 obiektów,
- TestC - 3 906 obiektów.

Zatem tworząc obiekt typu TestC, tworzymy: 1 obiekt typu TestC, 5 obiektów typów od TestC40 do TestC44, 25 obiektów typów od TestC30 do TestC34, 125 obiektów typów od TestC20 do TestC24, 625 obiektów typów od TestC10 do TestC14, 3 125 obiektów typów od TestC00 do TestC04 - co daje w sumie 3 906 obiektów.

#### 4.3.2 Wyniki dla Singleton

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	0	0	0	0	0	0	0	0	0	0	0	0
DryIoc	5	5	5	5	5	5	5	5	5	5	5	5
Grace	5	5	5	5	5	5	5	5	5	5	5	5
LightInject	4	4	4	4	4	4	4	4	4	4	4	4
Ninject	6	6	6	6	6	6	6	7	6	9	10	9
NiquIoCPartial	3	3	3	3	3	3	3	3	3	3	3	3
NiquIoCFull	5	6	5	5	5	5	5	5	5	5	6	5
SimpleInjector	4	4	4	4	4	4	4	4	4	4	4	4
StructureMap	22	22	22	22	22	22	22	22	22	23	23	23
Unity	15	16	15	15	16	15	15	16	16	16	16	16
Windsor	0	0	0	0	0	0	0	0	0	0	0	0

«opis»

#### 4.3.3 Wyniki dla Transient

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	2	2	2	24	25	24	234	240	235	2315	2330	2322
DryLoc	39	39	39	40	40	40	47	48	47	98	99	98
Grace	60	61	61	61	64	62	72	73	72	150	155	151
LightInject	36	37	36	36	37	37	43	44	44	86	86	86
Ninject	38	40	38	343	354	348	3386	3498	3432	36355	37506	36938
NiquIoCPartial	3	4	3	10	10	10	71	72	71	676	683	678
NiquIoCFull	31	33	31	31	33	32	38	38	38	81	86	84
SimpleInjector	40	41	40	41	42	41	49	49	49	113	114	114
StructureMap	26	26	26	40	40	40	179	182	180	1530	1553	1542
Unity	19	19	19	47	49	47	311	351	316	2954	3069	2974
Windsor	6	7	7	59	75	61	575	585	581	5755	6318	5846

«opis»

#### 4.3.4 Wyniki dla TransientSingleton

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	1	2	1	19	20	19	182	188	184	1790	1827	1806
DryLoc	58	58	58	59	107	64	64	65	64	104	106	105
Grace	22	22	22	22	23	22	25	25	25	43	44	44
LightInject	52	52	52	52	54	52	55	55	55	82	84	83
Ninject	26	28	27	199	219	204	1908	2017	1938	19238	20310	19572
NiquIoCPartial	3	3	3	6	7	6	38	38	38	337	346	341
NiquIoCFull	12	13	12	12	12	12	13	14	13	25	25	25
SimpleInjector	21	21	21	21	21	21	23	24	23	44	44	44
StructureMap	22	22	22	29	29	29	79	80	80	546	549	547
Unity	18	18	18	39	40	40	240	242	241	2246	2372	2268
Windsor	3	3	3	32	33	33	319	383	328	3162	3316	3195

«opis»

#### 4.3.5 Wyniki dla PerThread

Ilość	1			10			100			1000		
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	0	0	0	0	0	0	0	0	0	0	0	0
DryLoc	155	156	155	155	155	155	155	157	155	155	156	156
Grace	8	8	8	8	8	8	8	8	8	8	8	8
LightInject	523	541	526	523	529	526	523	527	525	523	528	527
Ninject	6	7	6	6	6	6	6	7	6	9	10	9
NiquIoCPartial	3	3	3	3	3	3	3	3	3	3	3	3
NiquIoCFull	5	5	5	5	5	5	5	5	5	5	5	5
SimpleInjector	15	16	16	15	16	16	15	16	16	16	16	16
StructureMap	22	22	22	22	22	22	22	22	22	23	23	23
Unity	15	16	15	15	16	16	15	16	16	16	16	16
Windsor	0	0	0	0	0	0	0	0	0	0	0	0

«opis»

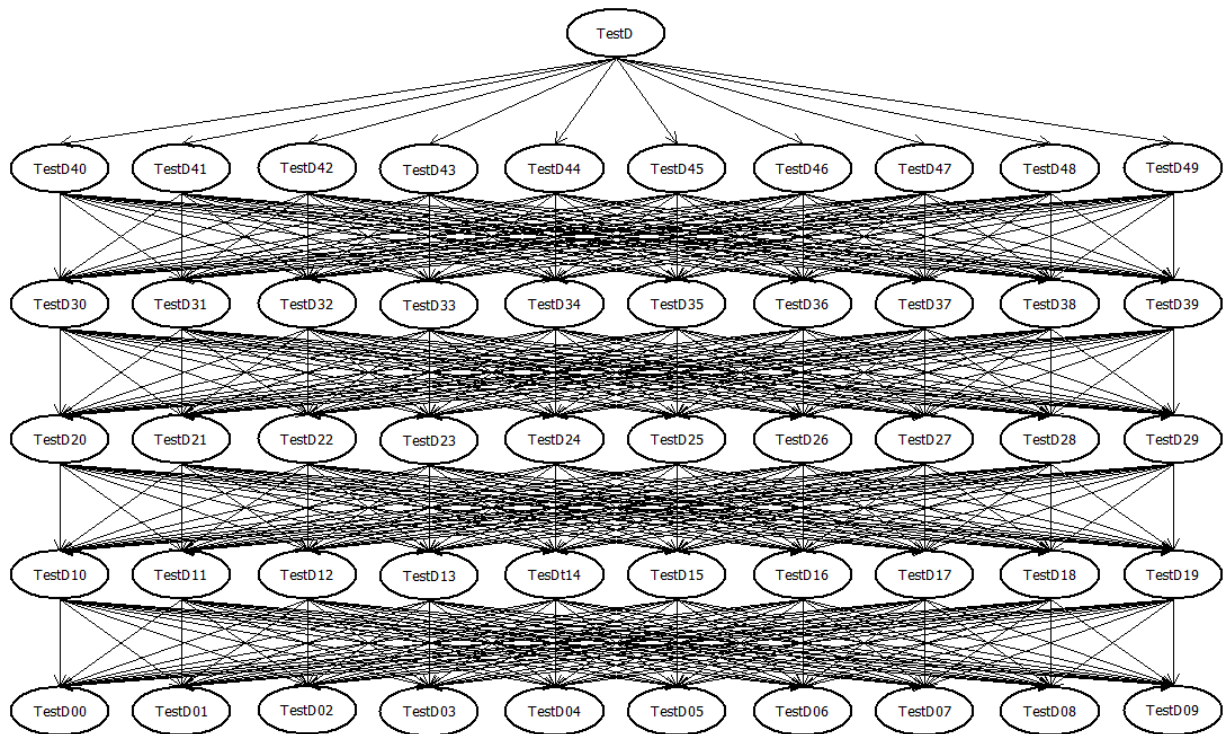
## 4.4 Przypadek testowy D

### 4.4.1 Opis

W tym teście mamy zdefiniowanych 51 typów. 10 z tych typów ma konstruktor bezparametrowy, a pozostałe 41 ma konstruktor z dziesięcioma parametrami. Typem głównym jest typ "TestD". Obiekt tego typu w konstruktorze przyjmuje 10 innych obiektów, kolejno następujących typów: "TestD40", "TestD41", "TestD42", "TestD43", "TestD44", "TestD45", "TestD46", "TestD47", "TestD48", "TestD49". Każdy z tych 10 typów w konstruktorze przyjmuje 10 obiektów o takich samych typach, ale z pierwszym numerkiem o 1 mniejszym (czyli obiekty typów od "TestD40" do "TestD49", przyjmują w konstruktorze obiekty typów od "TestD30" do "TestD39"). Dla typów od "TestD30" do "TestD39" zasad z konstruktorami wygląda tak samo. Na końcu dochodzimy do typów od "TestD00" do "TestD09", które mają konstruktor bezparametrowy. Rys. 4 przedstawia graf zależności typów dla tego przypadku testowego.

Łatwo wywnioskować, że tworząc obiekty poszczególnych typów ilość tworzonych obiektów rośnie ponad dziesięciokrotnie:

- typy od TestD00 do TestD09 - 1 obiekt,
- typy od TestD10 do TestD19 - 11 obiektów (obiekt danego typu plus 10 obiektów typów od TestD00 do TestD09),
- typy od TestD20 do TestD29 - 111 obiektów (obiekt danego typu plus 10 obiektów typów od TestD10 do TestD19),
- typy od TestD30 do TestD39 - 1 111 obiektów,



Rys. 4: Graf zależności dla testu D.

- typy od TestD40 do TestD49 - 11 111 obiektów,
- TestD - 111 111 obiektów.

Zatem tworząc obiekt typu TestD, tworzymy: 1 obiekt typu TestD, 10 obiektów typów od TestD40 do TestD49, 100 obiektów typów od TestD30 do TestD39, 1 000 obiektów typów od TestD20 do TestD29, 10 000 obiektów typów od TestD10 do TestD19, 100 000 obiektów typów od TestD00 do TestD09 - co daje w sumie 111 111 obiektów..

#### 4.4.2 Wyniki dla Singleton

Ilość	1			10		
	min	max	avg	min	max	avg
Autofac	0	0	0	0	0	0
DryIoc	10	11	10	10	13	10
Grace	17	20	18	17	18	18
LightInject	14	14	14	14	14	14
Ninject	21	21	21	21	22	21
NiquIoCPartial	8	9	9	8	9	9
NiquIoCFull	18	18	18	18	18	18
SimpleInjector	11	12	11	11	12	11
StructureMap	51	51	51	51	51	51
Unity	51	52	52	51	52	52
Windsor	0	0	0	0	0	0

Podobnie jak dla przypadku testowego A, tutaj również najlepiej poradziły sobie najpopularniejsze rozwiązania - Windsor i Autofac. Na trzecim miejscu znajduje się rozwiązanie NiquIoCPartial. Najslabiej poradził sobie rozwiązania StructureMap i Unity.

#### 4.4.3 Wyniki dla Transient

Ilość	1			10		
	min	max	avg	min	max	avg
Autofac	145	169	156	1302	1352	1318
DryIoc	1038	1086	1062	1068	1101	1089
Grace	1480	1553	1507	1511	1586	1551
LightInject	818	839	831	870	878	873
Ninject	1039	1189	1068	10104	10604	10299
NiquIoCPartial	36	37	36	284	304	289
NiquIoCFull	568	573	570	628	651	635
SimpleInjector	175	176	175	203	204	204
StructureMap	104	106	105	580	635	588
Unity	149	151	150	1069	1089	1074
Windsor	176	204	181	1777	1800	1785

Dla tego testu rozbieżność czasów jest dość duża.

Gdy mamy tylko 1 operację najlepiej radzi sobie NiquIoCPartial. Czasy pozostały są znacząco większe. Najslabiej poradziły sobie Grace, Ninject, DryIoc i LightInject.

Dla 10 operacji znacząco najlepiej radzi sobie SimpleInjector i NiquIoCPartial. Czołówkę zamykają StructureMap i NiquIoCFull. Pozostałe rozwiązania miały podobne, dużo słabsze czasy. Wyjątkiem jest ponownie Ninject, którego czasy są kilkukrotnie większe. W tym przypadku najmniejszy wzrost czasu miały kolejno rozwiązania: DryIoc, SimpleInjector, LightInject, Grace oraz NiquIoCFull.

#### 4.4.4 Wyniki dla TransientSingleton

Ilość	1			10		
	min	max	avg	min	max	avg
Autofac	77	78	77	898	917	904
DryLoc	891	896	893	921	959	928
Grace	362	369	364	369	378	371
LightInject	1278	1368	1303	1290	1332	1303
Ninject	511	561	524	4824	5067	4881
NiquIoCPartial	17	17	17	100	101	101
NiquIoCFull	143	144	143	150	158	154
SimpleInjector	85	86	85	90	91	90
StructureMap	60	66	61	166	169	167
Unity	116	117	116	693	699	695
Windsor	83	89	85	832	918	853

«opis»

#### 4.4.5 Wyniki dla PerThread

Ilość	1			10		
	min	max	avg	min	max	avg
Autofac	0	0	0	0	0	0
DryLoc	1062	1126	1093	1078	1117	1097
Grace	28	28	28	28	28	28
LightInject	-1	-1	-1	-1	-1	-1
Ninject	21	21	21	21	22	21
NiquIoCPartial	8	9	9	8	9	9
NiquIoCFull	18	18	18	18	18	18
SimpleInjector	46	46	46	46	47	46
StructureMap	51	51	51	51	51	51
Unity	51	53	52	51	52	52
Windsor	0	0	0	0	0	0

Tutaj tak samo jak dla przypadku testowego A czasy powinny być zbliżone do czasów dla Singleton. Niestety dla tego testu również nie wszystkie rozwiązania sobie z tym poradziły. Dużo słabsze czasy niż w przypadku Singleton miały rozwiązania: DryLoc, Grace, LightInject, SimpleInjector.

Zarówno dla 1 jak i 10 powtórzeń najlepiej radzi sobie Autofac i Windsor. Podium po-  
nownie zamyka NiquIoCPartial.

Najgorzej poradził sobie tutaj LightInject, który trwał ponad 20 minut (stąd w rozwią-  
żaniu czas "-1"). DryLoc również poradził sobie bardzo słabo - czasy ponad 20 raz gor-  
sze niż rozwiązanie na miejscu o 1 wyższym (Unity).



## 5 Podsumowanie

<parę słów na koniec>

## **Literatura**

- [1] Mark Seemann, Dependency Injection in .NET, 2012
- [2] Serge Lidin, Expert .NET 2.0 IL Assembler, 2006