

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>5</b>  |
| 1.1      | Cel pracy . . . . .                           | 5         |
| 1.2      | Układ pracy . . . . .                         | 5         |
| <b>2</b> | <b>Wstrzykiwanie zależności</b>               | <b>6</b>  |
| 2.1      | Wstęp . . . . .                               | 6         |
| 2.1.1    | Wstrzykiwanie przez konstruktor . . . . .     | 6         |
| 2.1.2    | Wstrzykiwanie przez metodę . . . . .          | 7         |
| 2.1.3    | Wstrzykiwanie przez właściwość . . . . .      | 7         |
| 2.2      | Implementacje przemysłowe . . . . .           | 8         |
| <b>3</b> | <b>Implementacja</b>                          | <b>9</b>  |
| 3.1      | Środowisko pracy . . . . .                    | 9         |
| 3.2      | Wstęp . . . . .                               | 9         |
| 3.2.1    | Microsoft Intermediate Language . . . . .     | 9         |
| 3.2.2    | Reflection.Emit . . . . .                     | 10        |
| 3.3      | Opis . . . . .                                | 10        |
| 3.3.1    | Register . . . . .                            | 11        |
| 3.3.2    | Resolve . . . . .                             | 12        |
| 3.3.3    | BuildUp . . . . .                             | 12        |
| 3.4      | Rozwiązanie . . . . .                         | 13        |
| 3.4.1    | Krok pierwszy . . . . .                       | 14        |
| 3.4.2    | Rozwiązanie 1 - PartialEmitFunction . . . . . | 15        |
| 3.4.3    | Rozwiązanie 2 - FullEmitFunction . . . . .    | 18        |
| <b>4</b> | <b>Testy wydajnościowe</b>                    | <b>25</b> |
| 4.0.1    | Register as Singleton . . . . .               | 25        |
| 4.0.2    | Register as Transient . . . . .               | 26        |

|       |  |    |
|-------|--|----|
| 4.0.3 | Register as TransientSingleton . . . . . | 26 |
| 4.0.4 | Register as PerThread . . . . .          | 26 |
| 4.0.5 | Register as FactoryMethod . . . . .      | 26 |
| 4.1   | Przypadek testowy A . . . . .            | 26 |
| 4.1.1 | Opis . . . . .                           | 26 |
| 4.1.2 | Wyniki dla Register . . . . .            | 28 |
| 4.1.3 | Wyniki dla Singleton . . . . .           | 29 |
| 4.1.4 | Wyniki dla Transient . . . . .           | 29 |
| 4.1.5 | Wyniki dla TransientSingleton . . . . .  | 30 |
| 4.1.6 | Wyniki dla PerThread . . . . .           | 31 |
| 4.1.7 | Wyniki dla FactoryMethod . . . . .       | 32 |
| 4.2   | Przypadek testowy B . . . . .            | 32 |
| 4.2.1 | Opis . . . . .                           | 32 |
| 4.2.2 | Wyniki dla Register . . . . .            | 34 |
| 4.2.3 | Wyniki dla Singleton . . . . .           | 35 |
| 4.2.4 | Wyniki dla Transient . . . . .           | 35 |
| 4.2.5 | Wyniki dla TransientSingleton . . . . .  | 36 |
| 4.2.6 | Wyniki dla PerThread . . . . .           | 37 |
| 4.2.7 | Wyniki dla FactoryMethod . . . . .       | 38 |
| 4.3   | Przypadek testowy C . . . . .            | 38 |
| 4.3.1 | Opis . . . . .                           | 38 |
| 4.3.2 | Wyniki dla Register . . . . .            | 40 |
| 4.3.3 | Wyniki dla Singleton . . . . .           | 40 |
| 4.3.4 | Wyniki dla Transient . . . . .           | 41 |
| 4.3.5 | Wyniki dla TransientSingleton . . . . .  | 42 |
| 4.3.6 | Wyniki dla PerThread . . . . .           | 43 |
| 4.3.7 | Wyniki dla FactoryMethod . . . . .       | 43 |
| 4.4   | Przypadek testowy D . . . . .            | 44 |

|          |   |           |
|----------|---|-----------|
| 4.4.1    | Opis . . . . .                          | 44        |
| 4.4.2    | Wyniki dla Register . . . . .           | 46        |
| 4.4.3    | Wyniki dla Singleton . . . . .          | 46        |
| 4.4.4    | Wyniki dla Transient . . . . .          | 47        |
| 4.4.5    | Wyniki dla TransientSingleton . . . . . | 48        |
| 4.4.6    | Wyniki dla PerThread . . . . .          | 49        |
| 4.4.7    | Wyniki dla FactoryMethod . . . . .      | 49        |
| <b>5</b> | <b>Podsumowanie</b>                     | <b>50</b> |
| 5.1      | Kontynuacja projektu . . . . .          | 50        |

## **Streszczenie**

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

Wstrzykiwanie zależności jest to zbiór zasad projektowania oprogramowania, które umożliwiają luźne powiązania, a dzięki temu kod jest rozszerzalny i łatwiejszy w utrzymaniu. Celem tej pracy było stworzenie takiej realizacji, która będzie sprawdzała się dla złożonych grafów zależności. Aplikacja została napisana w języku C# z wykorzystaniem przestrzeni nazw Reflecion.Emit oraz jest w pełni przetestowana. W aplikacji zaproponowano dwa rozwiązania, które działają niezależnie i można ich używać naprzemiennie. Testowy wydajnościowe zostały przeprowadzone przy użyciu aplikacji, która powstała na potrzeby tej pracy. Porównano w nich 5 najbardziej popularnych i 4 najszybsze rozwiązania. Końcowe wyniki wykazały, że z obecnie dostępnych rozwiązań na rynku, moje radzi sobie najlepiej.

# 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 funkcjonalności 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 moje rozwiązanie z kilkoma najbardziej popularnym i kilkoma najszybszymi rozwiązaniami.

## 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.[2] Wstrzykiwanie zależności może odbywać się na 3 sposoby:

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

#### 2.1.1 Wstrzykiwanie przez konstruktor

Jest to główny i najbardziej popularny sposób wstrzykiwania zależności. Niektóre klasy mają kilka konstruktor 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). Przykład klasy z atrybutem DependencyConstructor przedstawia Rys. 1.

```

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

    public EmptyClass EmptyClass { get; }
}

```

Rys. 1: Przykładowa klasa z atrybutem DependencyConstructor

### 2.1.2 Wstrzykiwanie przez metodę

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. Przykład klasy z oznaczeniem metody odpowiednim atrybutem przedstawia Rys. 2.

```

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

    public EmptyClass EmptyClass { get; private set; }
}

```

Rys. 2: Przykładowa klasa z atrybutem DependencyMethod

### 2.1.3 Wstrzykiwanie przez właściwość

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. Przykład klasy z oznaczeniem właściwości odpowiednim atrybutem przedstawia Rys. 3.



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

Rys. 3: Przykładowa klasa z atrybutem DependencyProperty

## 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 głównego 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 (Rys. 4).

```
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);
}
```

Rys. 4: Interfejs IContainer

Pierwsze cztery metody służą do rejestracji typów w kontenerze. Metoda piąta (Resolve) jest odpowiedzialna za tworzenie i zwracanie obiektów wcześniej zarejestrowanych typów. Ostatnia metoda (BuildUp) służy 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 interfejsu IContainer możemy zarejestrować zwykłe klasy. W drugiej interfejsy oraz 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 obiekt. 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 (Rys. 5), który umożliwia nam zarejestrowanie danego typu z określonym menadżerem czasu życia (czyli implementacją interfejsu IObjectLifetimeManager - Rys. 6). Zostało to tak zrobione, ponieważ dla różnych przypadków biznesowych możemy potrzebować, aby obiekt danego typu miał konkretny czas życia.

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

    void AsTransient();

    void AsPerThread();

    void AsPerHttpContext();

    void AsCustomObjectLifetimeManager(IObjectLifetimeManager objectLifetimeManager);
}
```

Rys. 5: Interfejs IContainerMember

Pierwsze cztery metody interfejsu IContainerMember, 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

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

    object GetInstance();
}
```

Rys. 6: Interfejs IObjectLifetimeManager

W interfejsie IObjectLifetimeManager 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 strategię, 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 process 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. Węć 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` (Rys. 7), 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.

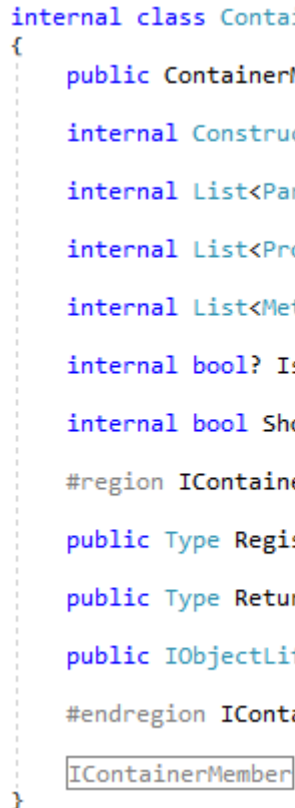
W tej operacji jest kilka wyjątków - są nimi typy zarejestrowane jako `Instance` albo `FactoryObject`. Dla pierwszego przypadku nie musimy uzyskiwać informacji o tym jak stworzyć obiekt takiego typu, ponieważ mamy już taki obiekt stworzony i go po prostu wykorzystamy. Dla drugiego przypadku obiekt tworzony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

Klasa `ContainerMember` 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.

```
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
}

```

Rys. 7: Klasa ContainerMember

### 3.4.2 Rozwiązanie 1 - PartialEmitFunction

Cały algorytm jest zawarty w metodzie Resolve (Rys. 8). 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();
}

```

Rys. 8: Metoda Resolve klasy PartialEmitFunction

Metoda GetObject (Rys. 9) 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;
}

```

Rys. 9: Metoda GetObject klasy PartialEmitFunction

Na początku metody CreateInstanceFunction (Rys. 10) 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;
}

```

Rys. 10: Metoda CreateInstanceFunction klasy PartialEmitFunction

CreateObjectFunction (Rys. 11) jest najbardziej zaawansowaną metodą w tym algorytmie. To w niej korzystamy z metod z przestrzeni nazw Reflection.Emit.

Na początku pobieramy informacje o konstruktorze. 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 konstruktor 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>));
}
```

Rys. 11: Metoda CreateObjectFunction klasy PartialEmitFunction

### 3.4.3 Rozwiązanie 2 - FullEmitFunction

Tutaj podobnie jak dla PartialEmitFunction cały algorytm zawarty jest w metodzie Resolve (Rys. 12). 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 zapisuje. 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();
}
```

Rys. 12: Metoda Resolve klasy FullEmitFunction

Metoda `GetObject` (Rys. 13) 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;
}
```

Rys. 13: Metoda `GetObject` klasy `FullEmitFunction`

`CreateInstanceFunction` (Rys. 14) 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 `FullEmitFunctionResult`, 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 oznaczając hasz danego typu, a wartością `typ`).

Najpierw opiszę metodę `ValidateTypesCache`, ponieważ nie wywołuje ona w sobie żadnej innej metody.

```

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;
}

```

Rys. 14: Metoda CreateInstanceFunction klasy FullEmitFunction

W pierwszym kroku metody ValidateTypesCache sprawdzamy (Rys. 15), 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 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;
    }
}

```

Rys. 15: Metoda ValidateTypesCache klasy FullEmitFunction

Metoda CreateObjectFunction (Rys. 16) 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 szczytu stosu.

Na końcu ze zmiennej typu `DynamicMethod` tworzymy delegata i go zwracamy. Delegat będzie przyjmował dwa parametry - oba typu `Dictionary`. Jeden z informacjami o `type`, 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
    };
}
```

Rys. 16: Metoda `CreateObjectFunction` klasy `FullEmitFunction`

W metodzie `CreateObjectFunctionPrivate` (Rys. 17) mamy trzy przypadki. W pierwszych dwóch przypadkach pomijamy obiekty, które zostały zarejestrowane jako `Instance` albo `FactoryObject` (określa to parametr `ShouldCreateCache` z obiektu typu `ContainerMember`).

```

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);
    }
}

```

Rys. 17: Metoda CreateObjectFunctionPrivate klasy FullEmitFunction

Pierwszy przypadek, to gdy typ został zarejestrowany jako Transient - Rys. 18. 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.

```

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

```

Rys. 18: Metoda IsTransient klasy FullEmitFunction



W drugim przypadku typ musi być singletonem (został on zarejestrowany jako Singleton, PerThread lub PerHttpContext) - Rys. 19. 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 typu, czyli jednej operacji Resolve).

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

Rys. 19: Metoda IsSingleton klasy FullEmitFunction

Metoda AddObjectCreatedByObjectLifetimeManager (Rys. 20), za wyjątkiem pierwszej operacji jaką jest stworzenie zmiennej lokalnej typu Type, zawiera jedynie operacje, które dodają kolejne pozycje 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.
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 zrzutuje 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.

```
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);
}
```

Rys. 20: Metoda `AddObjectCreatedByObjectLifetimeManager` klasy `FullEmitFunction`

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`.

## 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 "Register" i "Resolve" dla różnych rodzajów rejestracji. 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. Najpierw zostaną zaprezentowane czasy dla operacji "Register" dla wszystkich rodzajów rejestracji, a następnie dla każdego z rodzaju czasy dla operacji "Resolve". Wynika to z tego, że czasy dla rejestracji są prawie zawsze zbliżone do 0. Ze względu na nieduży narzut czasu, te testy zostały uruchamiane tylko 1 raz. Przy operacji "Resolve" każdy test był uruchamiany 100 razy, a w wynikach zostały przedstawione następujące czasy w milisekundach: minimalny, maksymalny i średni. Dla pierwszych trzech przypadków testy były wykonywane dla: 1, 10, 100 i 1000 powtórzeń, a dla ostatniego przypadku dla: 1 i 10 powtórzeń. Spowodowane jest to ilością obiektów, które są tworzone jednocześnie dla poszczególnych przypadków testowych.

### 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.

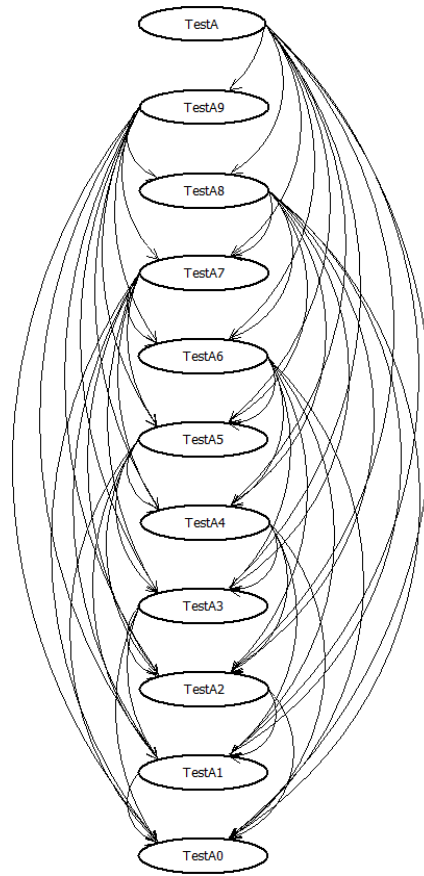
#### **4.0.5 Register as FactoryMethod**

Typ główny jest zarejestrowany jako "FactoryMethod", czyli jako funkcja, która zwraca nam obiekt danego typu. Obiekty pozostałych typów są tworzone wewnątrz tej funkcji za pomocą "new".

### **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 konstruktore 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. 21.



Rys. 21: Graf zależności dla testu A.

Łatwo z niego 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,
- 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 Register

| Test           | Singleton | Transient | TransientSingleton | PerThread |
|----------------|-----------|-----------|--------------------|-----------|
| Autofac        | 0         | 0         | 0                  | 0         |
| DryIoc         | 0         | 0         | 0                  | 0         |
| Grace          | 0         | 0         | 0                  | 0         |
| LightInject    | 0         | 0         | 0                  | 0         |
| Ninject        | 0         | 0         | 0                  | 0         |
| NiquIoCPartial | 0         | 0         | 0                  | 0         |
| NiquIoCFull    | 0         | 0         | 0                  | 0         |
| SimpleInjector | 0         | 0         | 0                  | 0         |
| StructureMap   | 0         | 0         | 0                  | 0         |
| Unity          | 0         | 0         | 0                  | 0         |
| Windsor        | 0         | 0         | 0                  | 0         |

Tab. 1: Wyniki testów dla operacji Register dla Przypadku testowego A

Dla każdego z rozwiązań, dla wszystkich testów w tym przypadku testowym, czasy rejestracji są równe 0, czyli poniżej 1 ms.

### 4.1.3 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    | 3   | 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    | 7   | 6   |
| NiquIoCPartial | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1    | 1   | 1   |
| NiquIoCFull    | 2   | 2   | 2   | 2   | 2   | 2   | 2   | 3   | 2   | 2    | 2   | 2   |
| SimpleInjector | 2   | 2   | 2   | 2   | 2   | 2   | 2   | 2   | 2   | 2    | 2   | 2   |
| StructureMap   | 9   | 10  | 9   | 9   | 10  | 9   | 9   | 10  | 9   | 10   | 11  | 10  |
| Unity          | 7   | 8   | 7   | 7   | 8   | 7   | 7   | 8   | 7   | 8    | 8   | 8   |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |

Tab. 2: Wyniki testów dla operacji Singleton dla Przypadku testowego A

Dla tego testu najlepiej poradziły sobie dwa z najpopularniejszych rozwiązania - Autofac i Windsor. Rozwiązanie NiquIoCPartial uplasowało się na 3 miejscu. Najślabiej poradził sobie pozostałe najpopularniejsze rozwiązania: Ninject, Unity oraz StructureMap. Reszta rozwiązań miała zbliżone czasy - dwukrotnie większe niż NiquIoCPartial i trzykrotnie mniejsze niż Ninject.

### 4.1.4 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   | 58  | 63   | 59  | 584  | 609  | 587  |
| DryLoc         | 14  | 14  | 14  | 14  | 15  | 15  | 16  | 16   | 16  | 29   | 30   | 29   |
| Grace          | 15  | 16  | 15  | 15  | 16  | 16  | 18  | 19   | 18  | 37   | 38   | 37   |
| LightInject    | 10  | 10  | 10  | 10  | 10  | 10  | 11  | 12   | 11  | 19   | 20   | 19   |
| Ninject        | 10  | 12  | 11  | 88  | 95  | 90  | 864 | 1035 | 882 | 8745 | 9610 | 8934 |
| NiquIoCPartial | 1   | 1   | 1   | 3   | 3   | 3   | 19  | 22   | 19  | 171  | 198  | 173  |
| NiquIoCFull    | 8   | 9   | 8   | 8   | 9   | 8   | 9   | 10   | 9   | 18   | 19   | 18   |
| SimpleInjector | 13  | 14  | 13  | 13  | 14  | 14  | 15  | 15   | 15  | 28   | 30   | 29   |
| StructureMap   | 10  | 10  | 10  | 13  | 14  | 14  | 53  | 55   | 54  | 414  | 457  | 417  |
| Unity          | 8   | 9   | 8   | 16  | 17  | 16  | 88  | 91   | 88  | 803  | 961  | 813  |
| Windsor        | 1   | 2   | 1   | 16  | 21  | 16  | 152 | 192  | 155 | 1511 | 1799 | 1529 |

Tab. 3: Wyniki testów dla operacji Transient dla Przypadku testowego A

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. DryLoc, Grace oraz NiquIoCPartial wypadły akceptowalnie - poradziły sobie trochę gorzej niż trzeci 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.

Natomiast 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 nadal zadowalająco. Pozostałe rozwiązania mają czasy od kilku do kilkunastu razy gorsze niż reszta. Bez cienia wątpliwości najgorzej wypadł Ninject.

Warto tutaj zaznaczyć, że wraz ze wzrostem operacji najmniejszy wzrost czasów miały rozwiązania: LightInject, NiquIoCFull, SimpleInjector, DryLoc oraz Grace.

#### 4.1.5 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   | 7   | 6   | 53  | 62  | 54  | 531  | 561  | 535  |
| DryLoc         | 18  | 19  | 18  | 19  | 19  | 19  | 20  | 21  | 20  | 31   | 32   | 32   |
| Grace          | 9   | 10  | 10  | 10  | 10  | 10  | 10  | 11  | 11  | 18   | 19   | 18   |
| LightInject    | 14  | 15  | 14  | 14  | 14  | 14  | 14  | 15  | 15  | 20   | 21   | 20   |
| Ninject        | 9   | 10  | 10  | 65  | 82  | 67  | 626 | 743 | 641 | 6297 | 7218 | 6463 |
| NiquIoCPartial | 1   | 1   | 1   | 2   | 3   | 2   | 14  | 15  | 14  | 120  | 123  | 121  |
| NiquIoCFull    | 5   | 6   | 5   | 5   | 6   | 5   | 6   | 6   | 6   | 10   | 12   | 11   |
| SimpleInjector | 9   | 9   | 9   | 9   | 10  | 9   | 10  | 11  | 10  | 17   | 19   | 17   |
| StructureMap   | 9   | 10  | 10  | 12  | 12  | 12  | 35  | 40  | 36  | 252  | 306  | 256  |
| Unity          | 8   | 9   | 8   | 14  | 15  | 14  | 73  | 82  | 74  | 659  | 753  | 663  |
| Windsor        | 1   | 2   | 1   | 12  | 14  | 12  | 117 | 140 | 119 | 1161 | 1302 | 1177 |

Tab. 4: Wyniki testów dla operacji TransientSingleton dla Przypadku testowego A

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 trzecie. Kolejne miejsca zajmują SimpleInjector i Grace. Trochę gorzej poradził sobie StructureMap. Zaraz za nim jest Windsor, który z podium spadł na 7 miejsce. Kolejne miejsca to LightInject i Unity. DryLoc nadal słabo,

jednakże wzrostu czasu był nieduży. Ostatnie miejsce zajął Ninject.

Dla 100 operacji na pierwszym miejscu znalazł się NiquIoCFull. Na podium znaleźli się jeszcze SimpleInjector i Grace. NiquIoCPartial spadł poza podium, a tuż za nim jest LightInject. DryIoC nadal z niedużym wzrostem czasu i tym razem jego wynik jest już akceptowalny. Pozostałe rozwiązań bardzo słabo. Kolejne miejsca to StructureMap oraz Autofac, a ranking zamykają Unity, Windsor i Ninject.

Test dla 1000 operacji nie spowodował żadnych zmian w 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.

#### 4.1.6 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         | 71  | 73  | 71  | 71  | 72  | 71  | 71  | 73  | 72  | 72   | 74  | 72  |
| Grace          | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4    | 4   | 4   |
| LightInject    | 50  | 53  | 50  | 50  | 51  | 50  | 50  | 54  | 51  | 50   | 52  | 51  |
| Ninject        | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 3   | 6    | 7   | 6   |
| NiquIoCPartial | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1   | 1    | 1   | 1   |
| NiquIoCFull    | 2   | 2   | 2   | 2   | 2   | 2   | 2   | 3   | 2   | 2    | 2   | 2   |
| SimpleInjector | 7   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8    | 8   | 8   |
| StructureMap   | 9   | 10  | 9   | 9   | 10  | 9   | 9   | 10  | 9   | 10   | 10  | 10  |
| Unity          | 7   | 8   | 7   | 7   | 8   | 7   | 7   | 8   | 7   | 8    | 9   | 8   |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |

Tab. 5: Wyniki testów dla operacji PerThread dla Przypadku testowego A

Czasy dla tego przypadku powinny być zbliżone do czasów dla przypadku Singleton, ponieważ wszystko było uruchamiane w jednym wątku. Niestety część rozwiązań sobie z tym nie poradziła.

Wyniki zbliżone posiadają: Autofac, Grace, Ninject, NiquIoCPartial, NiquIoCFull, StructureMap, Unity i Windsor. Natomiast dla SimpleInjector, LightInject i DryIoC czasy są od kilku do nawet kilkudziesięciu razy większe.

Jeśli chodzi o najlepsze rozwiązania, to wyglądają one tak samo jak dla Singleton. Pierwsze trzy miejsca to Aurofac, Windsor i NiquIoCPartial. Najsłabiej poradziły sobie LightInject i DryIoC. Tuż za czołówką znalazły się: NiquIoCFull, Grace i Ninject. Dla 1, 10 i 100 operacji wyżej uplasował się Ninject, a dla 1000 Grace. W drugiej połowie rankingu miejsca przypadły: SimpleInjector, Unity i StructureMap. Unity poradziło sobie gorzej niż SimpleInjector dla 1000 operacji, ale lepiej w pozostałych trzech przypadkach.



#### 4.1.7 Wyniki dla FactoryMethod

| 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         | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| Grace          | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| LightInject    | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| Ninject        | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 1   | 1   | 5    | 7   | 5   |
| NiquIoCPartial | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| NiquIoCFull    | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| SimpleInjector | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| StructureMap   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1    | 1   | 1   |
| Unity          | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1    | 1   | 1   |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1    | 1   | 1   |

Tab. 6: Wyniki testów dla operacji FactoryMethod dla Przypadku testowego A

Dla tego testu prawie wszystkie rozwiązania mają czasy zbliżone do 0 ms. Wyjątkiem są 4 rozwiązania. Trzy z nich, czyli: StrucutreMap, Unity oraz Windsor dla 1000 operacji uzyskały czas zbliżony do 1 ms. Najgorzej poradził sobie Ninject i już przy 100 operacjach uzyskał czasy na poziomie 1 ms. Natomiast w przypadku 1000 operacji poradził on sobie 5 razy gorzej.

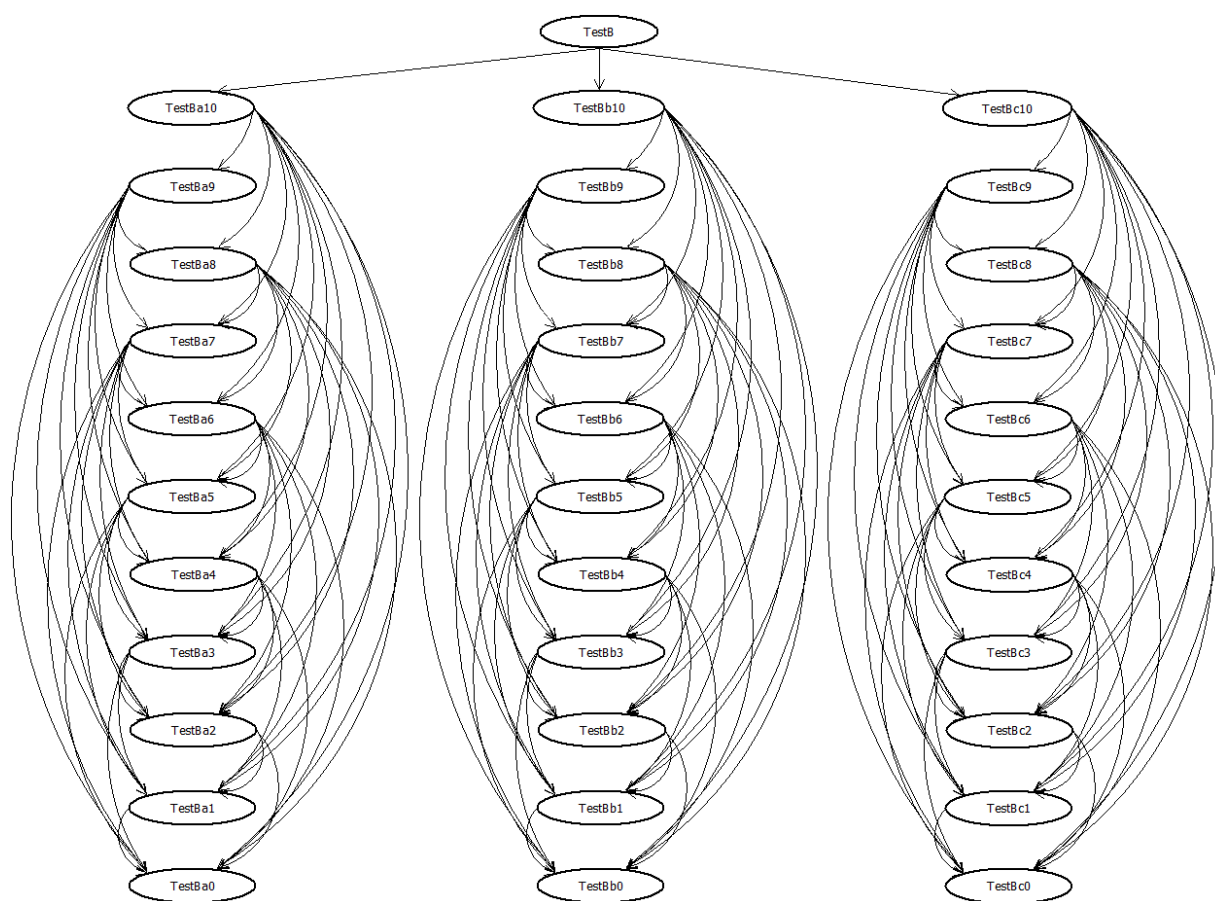
## 4.2 Przypadek testowy B

### 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. 22 przedstawia te zależności.

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),



Rys. 22: Graf zależności dla testu B.

- 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 Register

| Test           | Singleton | Transient | TransientSingleton | PerThread |
|----------------|-----------|-----------|--------------------|-----------|
| Autofac        | 0         | 0         | 0                  | 0         |
| DryIoc         | 0         | 0         | 0                  | 0         |
| Grace          | 0         | 0         | 0                  | 0         |
| LightInject    | 0         | 0         | 0                  | 0         |
| Ninject        | 0         | 0         | 0                  | 0         |
| NiquIoCPartial | 0         | 0         | 0                  | 0         |
| NiquIoCFull    | 0         | 0         | 0                  | 0         |
| SimpleInjector | 0         | 0         | 0                  | 0         |
| StructureMap   | 1         | 1         | 1                  | 1         |
| Unity          | 0         | 0         | 0                  | 0         |
| Windsor        | 1         | 1         | 1                  | 1         |

Tab. 7: Wyniki testów dla operacji Register dla Przypadku testowego B

Tutaj wszystkie rozwiązania za wyjątkiem StructureMap i Windsor mają zawsze czasy zbliżone do 0. Te dwa rozwiązania wszystkich testów mają czasy zbliżone do 1 ms.

#### 4.2.3 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   | 8   | 8   | 8   | 9   | 8   | 8   | 9   | 8   | 8    | 9   | 8   |
| Grace          | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8    | 8   | 8   |
| LightInject    | 6   | 7   | 6   | 6   | 7   | 6   | 6   | 7   | 6   | 6    | 7   | 6   |
| Ninject        | 9   | 10  | 9   | 9   | 10  | 9   | 9   | 11  | 10  | 12   | 13  | 12  |
| NiquIoCPartial | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 5   | 4   | 4    | 4   | 4   |
| NiquIoCFull    | 8   | 9   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8    | 8   | 8   |
| SimpleInjector | 6   | 6   | 6   | 6   | 6   | 6   | 6   | 6   | 6   | 6    | 6   | 6   |
| StructureMap   | 32  | 33  | 33  | 32  | 33  | 33  | 32  | 34  | 33  | 33   | 34  | 33  |
| Unity          | 23  | 24  | 23  | 23  | 24  | 23  | 23  | 24  | 24  | 24   | 25  | 24  |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |

Tab. 8: Wyniki testów dla operacji Singleton dla Przypadku testowego B

Podobnie jak dla przypadku testowego A, tutaj również najlepiej poradziły sobie najpopularniejsze rozwiązania - Autofac i Windsor. Warto zaznaczyć, że to jedyne rozwiązania, które nie zanotowały znaczącego wzrostu czasu. Dla pozostałych rozwiązań czasy wzrosły od 2 do 4 razy. NiquIoCPartial uplasował się tutaj również na 3 miejscu, a za nim SimpleInjector oraz LightInject. Koniec rankingu jest taki sam, czyli Ninject, Unity i StructureMap. NiquIoCFull, Grace, a także DryLoc miały zbliżone do siebie czasy - trochę słabsze niż LightInject i trochę lepsze niż Ninject.

#### 4.2.4 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  | 25  | 21  | 189  | 201  | 191  | 1890  | 2044  | 1903  |
| DryLoc         | 43  | 44  | 43  | 44  | 45  | 44  | 51   | 52   | 51   | 97    | 100   | 98    |
| Grace          | 50  | 59  | 51  | 51  | 55  | 52  | 57   | 59   | 58   | 122   | 128   | 124   |
| LightInject    | 31  | 33  | 32  | 32  | 33  | 32  | 37   | 38   | 37   | 70    | 73    | 71    |
| Ninject        | 36  | 38  | 36  | 276 | 337 | 284 | 2733 | 3595 | 2790 | 28220 | 29976 | 28720 |
| NiquIoCPartial | 5   | 5   | 5   | 10  | 10  | 10  | 59   | 79   | 61   | 543   | 633   | 550   |
| NiquIoCFull    | 26  | 27  | 26  | 26  | 27  | 26  | 32   | 33   | 32   | 66    | 68    | 66    |
| SimpleInjector | 40  | 41  | 40  | 40  | 42  | 41  | 47   | 48   | 47   | 95    | 97    | 96    |
| StructureMap   | 33  | 34  | 33  | 45  | 46  | 45  | 167  | 175  | 168  | 1333  | 1565  | 1346  |
| Unity          | 28  | 29  | 28  | 49  | 50  | 49  | 260  | 271  | 262  | 2355  | 2419  | 2366  |
| Windsor        | 6   | 7   | 6   | 49  | 69  | 50  | 476  | 610  | 484  | 4726  | 5665  | 4799  |

Tab. 9: Wyniki testów dla operacji Transient dla Przypadku testowego B

Dla tego testu sytuacja wygląda niemal identycznie jak w teście A, z tą różnicą, że czasy dla każdego rozwiązania są ponad 3 razy większe.

Przy 1 operację najlepiej radzi sobie Autofac, a tuż za nim NiquIoCPartial i Windsor. Pozostałe rozwiązania wypadły słabo i mają czasy od kilku do kilkunastu razy większe niż czołówka. Kolejne miejsca to: NiquIoCFull, Unity, LightInject, StructureMap oraz Ninject, a ranking zamykają SimplyInjector, DryLoc i Grace.

Gdy mamy 10 operacji najlepiej radzi sobie NiquIoCPartial, a zaraz za nim z czasem dwukrotnie gorszym jest Autofac. Następnie w rankingu znajduje się NiquIoCFull i LightInject. Tutaj również jak w przypadku testowym A, Ninject ma ponad 5 razy gorszy czas niż miejsce przedostatnie, z tą różnicą, że zajmuje je Grace, a nie Windsor, który uzyskał trochę lepszy czas i jest miejsce wyżej. Lepiej od Windsor poradziły sobie jeszcze: SimpleInjector, DryLoc, StructureMap i Unity, ale do czołówki brakowało im bardzo dużo.

Dla 100 operacji nic się nie zmieniło w stosunku do testu A. Pierwsze trzy miejsca zajmują: NiquIoCFull, LightInject i SimpleInjector. Grace z miejsca przedostatniego dla 10 operacji, awansował na miejsce piąte. Przed nim znalazł się DryLoc, a tuż za nim NiquIoCPartial. Czas całej trójki są zadowalające. Grono najslabszych to: StructureMap, Autofac, Unity, Windsor i Ninject.

Jak można się było spodziewać przy 1000 operacji sytuacja wygląda identycznie - pod względem czasu od czołówki oddalił się NiquIoCPartial, a na pierwszym miejscu wciąż pozostaje NiquIoCFull. LightInject oraz SimpleInjector są kawałek za nim. Dalej DryLoc i Grace wciąż akceptowalnie. Miejsce w środku rankingu zajmuje NiquIoCPartial, a pozostałe rozwiązania mają od niego o rząd lub dwa większe czasy.

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

#### 4.2.5 Wyniki dla TransientSingleton

| Ilość          | 1   |     |     | 10  |     |     | 100  |      |      | 1000  |       |       |
|----------------|-----|-----|-----|-----|-----|-----|------|------|------|-------|-------|-------|
|                | min | max | avg | min | max | avg | min  | max  | avg  | min   | max   | avg   |
| Autofac        | 1   | 2   | 1   | 18  | 20  | 18  | 172  | 206  | 173  | 1698  | 1832  | 1709  |
| DryLoc         | 56  | 57  | 56  | 57  | 58  | 57  | 64   | 65   | 64   | 108   | 110   | 109   |
| Grace          | 30  | 32  | 31  | 31  | 32  | 31  | 35   | 37   | 35   | 69    | 72    | 69    |
| LightInject    | 48  | 50  | 48  | 48  | 50  | 48  | 51   | 53   | 51   | 80    | 82    | 80    |
| Ninject        | 29  | 34  | 30  | 214 | 239 | 219 | 2013 | 2426 | 2063 | 20440 | 22891 | 20858 |
| NiquIoCPartial | 4   | 5   | 5   | 8   | 9   | 8   | 41   | 57   | 42   | 365   | 406   | 368   |
| NiquIoCFull    | 16  | 16  | 16  | 16  | 17  | 16  | 18   | 19   | 18   | 37    | 39    | 38    |
| SimpleInjector | 28  | 28  | 28  | 28  | 29  | 28  | 32   | 34   | 32   | 61    | 62    | 61    |
| StructureMap   | 33  | 34  | 33  | 40  | 46  | 40  | 112  | 123  | 113  | 797   | 915   | 805   |
| Unity          | 27  | 28  | 27  | 46  | 47  | 46  | 226  | 263  | 228  | 2016  | 2305  | 2032  |
| Windsor        | 3   | 5   | 3   | 37  | 46  | 37  | 355  | 435  | 362  | 3543  | 4824  | 3637  |

Tab. 10: Wyniki testów dla operacji TransientSingleton dla Przypadku testowego B

Jeśli chodzi o wzrost czasów w stosunku do przypadku testowego A, to sytuacja wygląda niemal identycznie jak dla testu Transient - jest on ponad trzykrotny.

Dla 1 operacji podium wygląda jednak inaczej niż dla testu A - miejscami zamieniły się

NiquLocPartial i Windsor. Na pierwszy miejscu pozostaje Autofac. Różnice czasów dla pozostałych rozwiązań są już dużo bardziej zauważalne. Znajdujący się tuż za podium NiquLocFull ma wynik ponad trzy razy słabszy niż NiquLocPartial, który to podium zamyka. Kolejne miejsca to: Unity, SimpleInjector, Ninject, Grace, StructureMap, LightInject i na końcu DryLoc.

Gry mamy 10 operacji pierwsze dwa miejsca zajmują NiquLocPartial i NiquLocFull. Tuż za nimi znajduje się Autofac. SimpleInjector oraz Grace zanotowały niewielki wzrost czasów i zajmują kolejne dwa miejsca. Nieduży wzrostu czasów zaliczyły również LightInject i DryLoc, ale ich czasy są wciąż duże w stosunku do pozostałych rozwiązań i zajmuje dwa przedostatnie miejsca - ustępując jedynie Ninject. Miejsca od 6 do 8 przypadły: Windsor, StructureMap oraz Unity.

Przy 100 i 1000 operacjach nie ma już żadnych zmian w rankingu względem testu A. Dla pierwszego przypadku początek rankingu wygląda następująco: NiquLocFull, SimpleInjector i Grace. Zaraz za podium znajduje się NiquLocPartial. LightInject oraz DryLoc dzięki niedużemu wzrostowi czasów znalazły się na miejscach piątym i szóstym. Są to ostatnie zadowolające rozwiązania. Kolejne miejsca zajmują: StructureMap, Autofac, Unity, Windsor oraz Ninject.

Dla drugiego przypadku pierwsze trzy miejsca są jak wyżej. Natomiast o jedno miejsce awansowały LightInject oraz Grace. NiquLocPartial spadł na miejsce szóste i dołączył do grupy rozwiązań z bardzo dużymi czasami. Reszta rankingu nie uległa zmianie.

#### 4.2.6 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         | 246 | 250 | 247 | 246 | 250 | 247 | 246 | 248 | 247 | 246  | 249 | 247 |
| Grace          | 13  | 13  | 13  | 13  | 13  | 13  | 13  | 14  | 13  | 13   | 14  | 13  |
| LightInject    | 409 | 419 | 412 | 409 | 432 | 413 | 409 | 416 | 412 | 409  | 420 | 413 |
| Ninject        | 9   | 10  | 9   | 9   | 10  | 9   | 9   | 10  | 10  | 12   | 13  | 13  |
| NiquLocPartial | 4   | 5   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4    | 5   | 4   |
| NiquLocFull    | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8   | 8    | 8   | 8   |
| SimpleInjector | 23  | 23  | 23  | 23  | 23  | 23  | 23  | 23  | 23  | 23   | 24  | 23  |
| StructureMap   | 32  | 34  | 33  | 32  | 38  | 33  | 33  | 35  | 33  | 33   | 34  | 33  |
| Unity          | 23  | 25  | 23  | 23  | 24  | 23  | 23  | 24  | 23  | 24   | 25  | 24  |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |

Tab. 11: Wyniki testów dla operacji PerThread dla Przypadku testowego B

Podobnie jak było dla przypadku testowego A, tutaj również nie wszystkie rozwiązania mają czasy zbliżone do Singleton.

Są to te same rozwiązania co dla testu A, czyli: SimpleInjector, DryLoc i LightInject. Warto jednak zaznaczyć, że dla SimpleInjector wzrost czasów jest kilkukrotny, a dla pozostałych dwóch rozwiązań kilkudziesięciokrotny.

Tak samo jak w przypadku Singleton czołówka wygląda następująco: Autofac, Wind-

sor i NiquIoCPartial. Kolejne miejsca zajmują: NiquIoCFull, Ninject i Grace. Najslabiej poradziły sobie rozwiązania LightInject i DryIoc. W środku tabeli pod względem czasów znalazły się SimpleInjector, Unity i StrcutreMap. W odróżnieniu od przypadku testowego A, tutaj nie było żadnych zmian w rankingu i wszystkie rozwiązania zajmują te same miejsca dla dowolnej liczby operacji.

#### 4.2.7 Wyniki dla FactoryMethod

| 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   | 1    | 1   | 1   |
| DryIoc         | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| Grace          | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 1   | 0   |
| LightInject    | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| Ninject        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 1   | 6    | 6   | 6   |
| NiquIoCPartial | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| NiquIoCFull    | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| SimpleInjector | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1    | 1   | 1   |
| StructureMap   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1    | 2   | 1   |
| Unity          | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 2    | 2   | 2   |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1    | 2   | 1   |

Tab. 12: Wyniki testów dla operacji FactoryMethod dla Przypadku testowego B

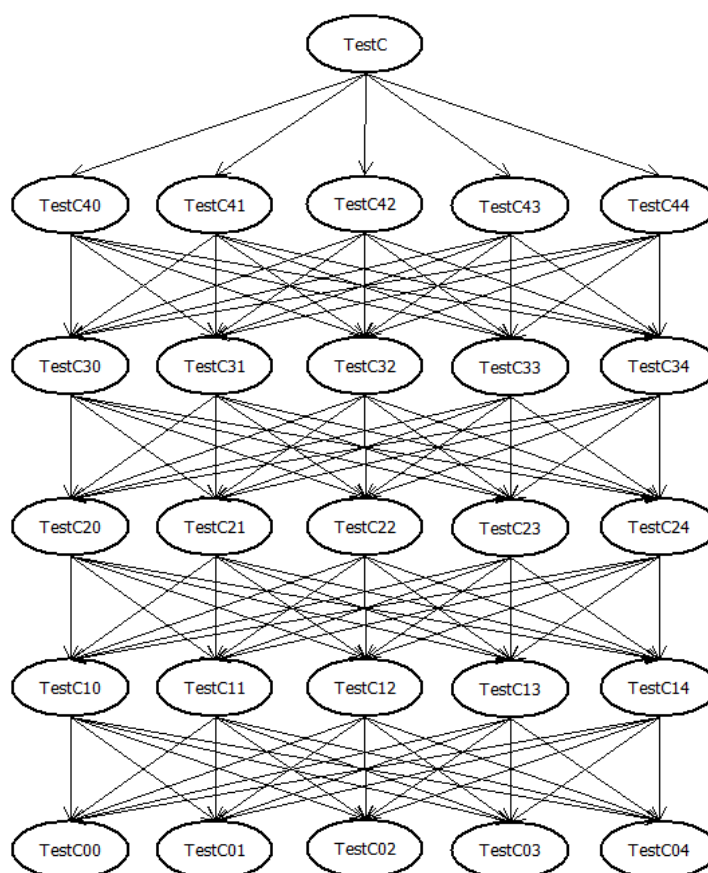
Dla tego przypadku testowego trochę więcej rozwiązań miało czasy 1 ms lub większe. Wyniki na poziomie 0 ms dla każdej liczby powtórzeń zanotowały: DryIoc, LightInject, NiquIoCPartial oraz NiquIoCFull. Trochę gorzej od nich poradził sobie Grace. Następne miejsca przypadły: Autofac, SimpleInjector, StructureMap i Windsor. Czasy około dwa razy gorsze od nich osiągnął Unity. Wszystkie z tych 6 rozwiązań miałyby czasy powyżej 0 ms dopiero dla 1000 operacji. Ninject tym razem również już przy 100 operacjach zaczął mieć wyniki na poziomie 1 ms. Gdy liczba operacji została zwiększona do 1000, to jego czasy wzrosły sześciokrotnie.

### 4.3 Przypadek testowy C

#### 4.3.1 Opis

Ten przypadek testowy znacząco różni się od dwóch poprzednich. Mamy tutaj 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 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, które w nazwie mają pierwszy numer o 1 mniejszy, czyli są to obiekty typów od "TestC30" do "TestC34". Dla tych i

kolejnych typów zasad z konstruktorami wygląda tak samo. Na końcu dochodzimy do typów od "TestC00" do "TestC04", które mają konstruktor bezparametrowy. Rys. 23 przedstawia graf zależności typów dla tego przypadku testowego.



Rys. 23: Graf zależności dla testu C.

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

- 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 oraz 3 125 obiektów typów od TestC00 do TestC04 - co daje w sumie 3 906 obiektów.

#### 4.3.2 Wyniki dla Register

| Test           | Singleton | Transient | TransientSingleton | PerThread |
|----------------|-----------|-----------|--------------------|-----------|
| Autofac        | 0         | 0         | 0                  | 0         |
| DryLoc         | 0         | 0         | 0                  | 0         |
| Grace          | 0         | 0         | 0                  | 0         |
| LightInject    | 0         | 0         | 1                  | 0         |
| Ninject        | 0         | 0         | 0                  | 0         |
| NiquIoCPartial | 0         | 0         | 0                  | 0         |
| NiquIoCFull    | 0         | 0         | 0                  | 0         |
| SimpleInjector | 0         | 0         | 0                  | 0         |
| StructureMap   | 1         | 1         | 1                  | 1         |
| Unity          | 0         | 0         | 0                  | 0         |
| Windsor        | 0         | 0         | 0                  | 0         |

Tab. 13: Wyniki testów dla operacji Register dla Przypadku testowego C

Tym razem wszystkie rozwiązania za wyjątkiem LightInject i StructureMap mają zawsze czasy zbliżone do 0. LightInject dla testu TransientSingleton ma czas zbliżony do 1 ms, a StructureMap dla wszystkich testów.

#### 4.3.3 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         | 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   | 5   | 5   | 5   | 5   | 5   | 5   | 5   | 5   | 5    | 5   | 5   |
| SimpleInjector | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4   | 4    | 4   | 4   |
| StructureMap   | 22  | 23  | 22  | 22  | 23  | 22  | 22  | 23  | 22  | 23   | 24  | 23  |
| Unity          | 15  | 16  | 15  | 15  | 16  | 15  | 15  | 16  | 15  | 16   | 17  | 16  |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |

Tab. 14: Wyniki testów dla operacji Singleton dla Przypadku testowego C

Pomimo zupełnie innego przypadku testowego, dla testu Singleton, wciąż najlepiej radzą sobie Autofac i Windsor. Ich czasy nawet dla 1000 operacji są wciąż mniejsze niż 1 milisekunda. Dodatkowo tutaj również podium zamyka NiquIoCPartial. Kolejne miejsca zajmują LightInject i SimpleInjector. Trochę gorzej od nich radzą sobie DryIoC, Grace i NiquIoCFull. Ostatnie trzy miejsca należą do Ninject, Unity i StructureMap. Zatem ranking wygląda tutaj identycznie jak dla poprzednich dwóch testów dla tego samego przypadku.

#### 4.3.4 Wyniki dla Transient

| Ilość          | 1   |     |     | 10  |     |     | 100  |      |      | 1000  |       |       |
|----------------|-----|-----|-----|-----|-----|-----|------|------|------|-------|-------|-------|
|                | min | max | avg | min | max | avg | min  | max  | avg  | min   | max   | avg   |
| Autofac        | 2   | 2   | 2   | 23  | 25  | 24  | 229  | 250  | 231  | 2274  | 2419  | 2288  |
| DryIoC         | 39  | 41  | 39  | 40  | 41  | 40  | 47   | 48   | 47   | 98    | 100   | 99    |
| Grace          | 61  | 63  | 61  | 62  | 65  | 62  | 73   | 74   | 73   | 153   | 163   | 155   |
| LightInject    | 36  | 37  | 36  | 36  | 39  | 37  | 43   | 49   | 44   | 86    | 88    | 86    |
| Ninject        | 38  | 42  | 39  | 345 | 375 | 353 | 3430 | 4043 | 3511 | 36072 | 42914 | 37642 |
| NiquIoCPartial | 3   | 4   | 4   | 10  | 11  | 10  | 72   | 80   | 74   | 683   | 781   | 690   |
| NiquIoCFull    | 31  | 33  | 31  | 31  | 33  | 32  | 38   | 41   | 38   | 81    | 83    | 82    |
| SimpleInjector | 40  | 42  | 41  | 41  | 43  | 41  | 49   | 51   | 49   | 115   | 117   | 116   |
| StructureMap   | 26  | 27  | 26  | 40  | 42  | 40  | 178  | 207  | 181  | 1520  | 1810  | 1540  |
| Unity          | 18  | 19  | 19  | 47  | 51  | 47  | 315  | 369  | 318  | 2995  | 3359  | 3015  |
| Windsor        | 7   | 8   | 7   | 61  | 82  | 62  | 594  | 769  | 608  | 5898  | 7708  | 6037  |

Tab. 15: Wyniki testów dla operacji Transient dla Przypadku testowego C

Gdy mamy 1 operację najniższe czasy uzyskały Autofac i NiquIoCPartial. Tuż za nimi znalazł się Windsor. Kolejne rozwiązania mają już dużo większe czasy. Na czwartym miejscu znalazł się Unity, dalej StructureMap i NiquIoCFull. Następne miejsca należą kolejno do LightInject, DryIoC, Ninject i SimpleInjector. Najgorzej poradził sobie tutaj Grace.

Przy 10 operacjach pierwsze dwa miejsca uległy zamianie. Tym razem najniższe czasy uzyskał NiquIoCPartial. Podium zamyka NiquIoCFull, który awansował o 3 miejsca. Niewiele za nim uplasował się LightInject. Kolejne pozycje w rankingu należą do DryIoC, StructureMap oraz SimpleInjector, które uzyskały zbliżone do siebie czasy. Znaczący wzrost czasów zanotowały natomiast Unity i Windsor, które znalazły się na miejscach 8 i 10. Z niewielkim wzrostem czasu w stosunku do 1 operacji, między nimi znalazł się Grace. Ranking zamyka Ninject.

Dla 100 operacji na prowadzenie wysunęły się NiquIoCFull oraz LightInject. Trochę słabiej poradziły sobie DryIoC i SimpleInjector, jednak ich czasy są również zadowalające. Następne miejsca z zauważalną różnicą czasów w stosunku do czołówki zajęły Grace oraz NiquIoCPartial. Miejsce 7 w rankingu zajął StructureMap z czasem ponad dwukrotnie większym niż poprzednik. Kolejne miejsca należą do: Autofac, Unity, Windsor i Ninject.

Test dla 1000 operacji nie spowodował żadnych zmian w rankingu, jednakże czasy dla

NiquIoCPartial przestały być akceptowalne - wzrosły one ponad dziewięciokrotnie, a dla wyższych miejsc wzrost był maksymalnie trzykrotny. Pozostałe rozwiązania miały wzrost zbliżony do wzorstu NiquIoCPartial.

#### 4.3.5 Wyniki dla TransientSingleton

| Ilość          | 1   |     |     | 10  |     |     | 100  |      |      | 1000  |       |       |
|----------------|-----|-----|-----|-----|-----|-----|------|------|------|-------|-------|-------|
|                | min | max | avg | min | max | avg | min  | max  | avg  | min   | max   | avg   |
| Autofac        | 1   | 2   | 2   | 19  | 20  | 20  | 184  | 198  | 187  | 1810  | 1898  | 1831  |
| DryLoc         | 58  | 70  | 58  | 59  | 62  | 59  | 65   | 68   | 65   | 105   | 107   | 106   |
| Grace          | 23  | 24  | 23  | 23  | 24  | 23  | 25   | 26   | 25   | 44    | 45    | 44    |
| LightInject    | 52  | 54  | 52  | 52  | 55  | 53  | 55   | 57   | 56   | 82    | 85    | 83    |
| Ninject        | 25  | 30  | 26  | 199 | 231 | 203 | 1903 | 2364 | 1951 | 19137 | 22595 | 19626 |
| NiquIoCPartial | 3   | 3   | 3   | 6   | 7   | 6   | 38   | 46   | 39   | 347   | 359   | 349   |
| NiquIoCFull    | 12  | 13  | 12  | 12  | 13  | 12  | 13   | 14   | 14   | 25    | 26    | 25    |
| SimpleInjector | 21  | 22  | 21  | 21  | 22  | 21  | 23   | 24   | 24   | 44    | 45    | 44    |
| StructureMap   | 23  | 23  | 23  | 30  | 31  | 30  | 80   | 84   | 80   | 549   | 593   | 553   |
| Unity          | 18  | 18  | 18  | 39  | 41  | 40  | 242  | 258  | 243  | 2258  | 2428  | 2272  |
| Windsor        | 3   | 3   | 3   | 34  | 41  | 35  | 338  | 400  | 343  | 3368  | 4115  | 3429  |

Tab. 16: Wyniki testów dla operacji TransientSingleton dla Przypadku testowego C

Dla 1 operacji tylko 3 rozwiązania uzyskały wynik jednocyfrowy. Są to kolejno: Autofac, NiquIoCPartial i Windsor. Pozostałe rozwiązania czasy mają dużo słabsze. Najniższy uzyskał NiquIoCPartial, a dalej za nim Unity i SimpleInjector. Kolejne miejsca należą do StructureMap, Grace i Ninject. Na końcu rankingu znalazły się LightInject oraz DryLoc.

Przy 10 operacjach sytuacja lekko uległa zmianie. Pierwsze dwa miejsca zajmują NiquIoCPartial i NiquIoCFull. Kawalek za nimi są Autofac i SimpleInjector, które dzieli niewielka różnica. Trochę gorszy czas od nich uzyskał Grace. Reszta rozwiązań dość znacząco odstaje od czołówki. Ostatnie 6 miejsc to: StructureMap, Windsor, Unity, LightInject, DryLoc oraz na końcu Ninject.

Gdy mamy 100 operacji na prowadzenie wysunął się NiquIoCFull, a poza pierwszą trójkę wypadł NiquIoCPartial. Miejsce drugie i trzecie należą do SimpleInjector oraz Grace. LightInject i DryLoc zanotowały niewielki wzrost czasów i przy dużym wzroście dla pozostałych rozwiązań, znalazły się na miejscach 5 i 6. Do grona najsłabszych dołączył natomiast Autofac. Zajął on miejsce pomiędzy StructureMap, a Unity. Ostatnia pozycja nie uległa zmianie i należy do Ninject, przed którym znalazł się Windsor.

Również i dla tego testu przy 1000 operacjach nie było dużych zmian w rankingu. Wszystkie rozwiązania zachowały swoje miejsca za wyjątkiem NiquIoCPartial, który spadł na miejsce 6.

#### 4.3.6 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         | 157 | 162 | 158 | 157 | 160 | 158 | 157 | 160 | 158 | 158  | 161 | 158 |
| Grace          | 8   | 9   | 8   | 8   | 9   | 8   | 8   | 9   | 8   | 8    | 9   | 8   |
| LightInject    | 525 | 556 | 529 | 525 | 551 | 530 | 525 | 558 | 530 | 525  | 550 | 529 |
| 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 | 16  | 16  | 16  | 16  | 16  | 16  | 16  | 16  | 16  | 16   | 18  | 16  |
| StructureMap   | 22  | 23  | 22  | 22  | 23  | 22  | 22  | 23  | 22  | 23   | 73  | 24  |
| Unity          | 15  | 16  | 15  | 15  | 16  | 15  | 15  | 16  | 15  | 16   | 17  | 16  |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |

Tab. 17: Wyniki testów dla operacji PerThread dla Przypadku testowego C

Podobnie jak dla przypadków testowych A i B, tutaj również SimpleInjector zaliczył lekki wzrost czasów w stosunku do testu Singleton. DryLoc oraz LightInject zaliczyły bardzo duże wzrosty i to one zamykają ranking. Przed nimi znalazł się StructureMap, a miejsce wyżej wcześniej wspomniany SimpleInjector. Podobne czasy do niego zanotował Unity, który uplasował się na miejscu 7. Idąc w rankingu ku górze, to przy 1, 10 i 100 operacjach lepiej radził Ninject, a przy 1000 operacjach Grace. Te dwa rozwiązania znalazły się na dwóch kolejnych miejscach. Tuż za podium jest NiquIoCFull, a pierwsze trzy miejsca bez zmian dla tego testu, czyli: Autofac, Windsor, NiquIoCPartial.

#### 4.3.7 Wyniki dla FactoryMethod

| 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    | 1   | 0   |
| DryLoc         | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| Grace          | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| LightInject    | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| Ninject        | 0   | 0   | 0   | 0   | 0   | 0   | 1   | 1   | 1   | 5    | 7   | 5   |
| NiquIoCPartial | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| NiquIoCFull    | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 0   | 0   |
| SimpleInjector | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0    | 1   | 1   |
| StructureMap   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1    | 1   | 1   |
| Unity          | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1    | 2   | 1   |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 0   | 1    | 1   | 1   |

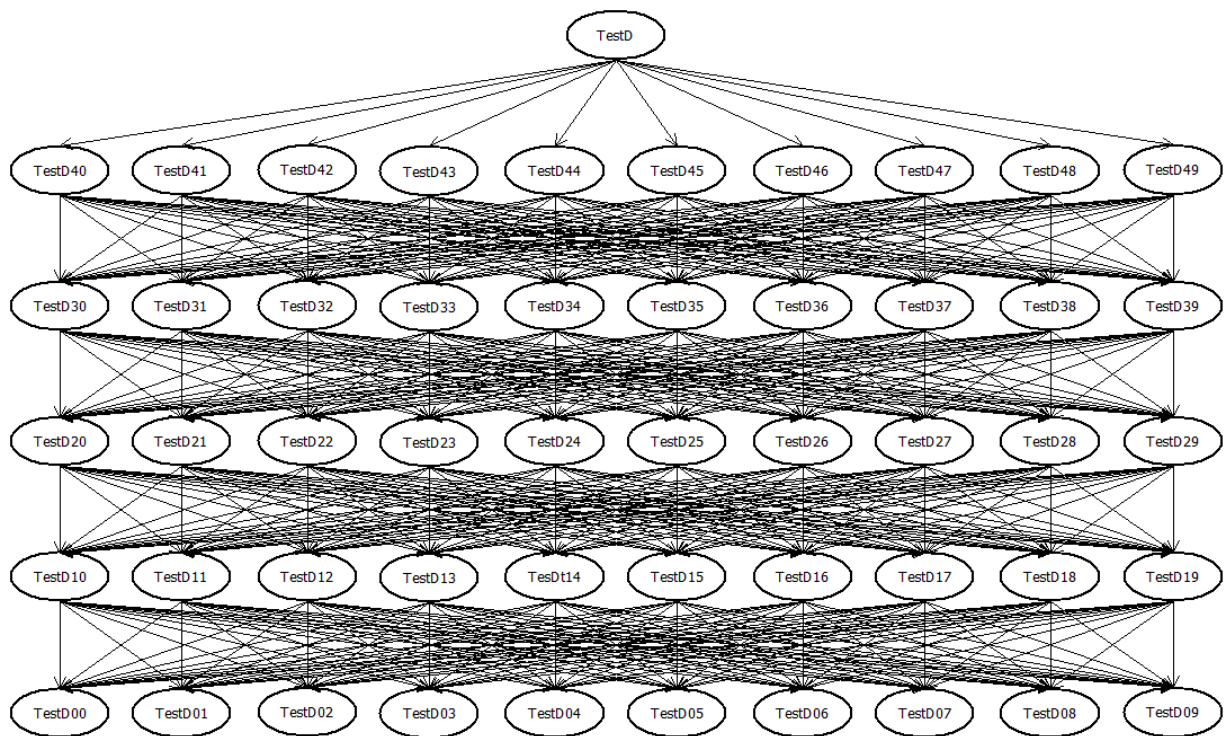
Tab. 18: Wyniki testów dla operacji FactoryMethod dla Przypadku testowego C

W tym przypadku 6 rozwiązań zanotowało wszystkie średnie czasy na poziomie 0 ms. Są to: DryLoc, Grace, LightInject, NiquIoCPartial, NiquIoCFull i Autofac, który czasem dla 1000 operacji osiągał 1 ms. Dalej znalazł się SimpleInjector, który czasami dla 1000 powtórzeń miał czasy 0 ms, ale jego średni czas to 1 ms. Następne miejsca przypadły StructureMapp i Windsor. Przedostatni w rankingu jest Unity, który czasami dla 1000 operacji uzyskiwał czasy nawet około 2 ms. Ostatnie miejsce tym razem również zajmuje Ninject i sytuacja wygląda niemal identycznie jak w pozostałych dwóch przypadkach testowych dla tego testu - już przy 100 powtórzeniach osiągnął on czasy na poziomie 1 ms, a gdy mamy 1000 operacji, to czasy średnie wynoszą około 5 ms.

## 4.4 Przypadek testowy D

### 4.4.1 Opis

Podobnie jak przypadek testowy B jest analogiczny z przypadkiem A, tak ten przypadek, jest zbliżony do przypadku C. Jednakże pokrewieństwo jest inne. Mamy tutaj tyle sama poziomów, co dla C, ale na każdym poziomie (poza pierwszym) obiektów jest dwa razy więcej. Zatem w tym teście mamy zdefiniowanych 51 typów. Tutaj konstruktor bezparametrowy ma 10 typów, a pozostałe 41 ma konstruktor z dziesięcioma parametrami. Typ główny to "TestD" i przyjmuje on w konstruktorze 10 obiektów, kolejno następujących typów: "TestD40", "TestD41", "TestD42", "TestD43", "TestD44", "TestD45", "TestD46", "TestD47", "TestD48", "TestD49". Jest więc to sytuacja niemal identyczna jak dla poprzedniego przypadku. Dla pozostałych typów jest podobnie i każdy z nich w konstruktorze przyjmuje 10 obiektów o typach z pierwszym numerem o 1 mniejszym (obiekty typów od "TestD40" do "TestD49", przyjmują w konstruktorze obiekty typów od "TestD30" do "TestD39" itd.). Ostatnie 10 typów, czyli typy od "TestD00" do "TestD09", mają konstruktor bezparametrowy. Graf zależności dla tego przypadku testowego został przedstawiony na Rys. 24.



Rys. 24: Graf zależności dla testu D.

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,
- 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 Register

| Test           | Singleton | Transient | TransientSingleton | PerThread |
|----------------|-----------|-----------|--------------------|-----------|
| Autofac        | 0         | 0         | 0                  | 0         |
| DryLoc         | 0         | 0         | 0                  | 0         |
| Grace          | 0         | 0         | 0                  | 0         |
| LightInject    | 0         | 0         | 0                  | 0         |
| Ninject        | 0         | 0         | 0                  | 0         |
| NiquIoCPartial | 0         | 0         | 0                  | 0         |
| NiquIoCFull    | 0         | 0         | 0                  | 0         |
| SimpleInjector | 0         | 0         | 0                  | 0         |
| StructureMap   | 4         | 2         | 1                  | 4         |
| Unity          | 0         | 0         | 0                  | 0         |
| Windsor        | 1         | 1         | 1                  | 2         |

Tab. 19: Wyniki testów dla operacji Register dla Przypadku testowego D

Dla tego przypadku testowego również wszystkie rozwiązania za wyjątkiem StructureMap i Windsor mają czasy zbliżone do 0 ms. StructureMap radzi sobie zdecydowanie najgorzej. Zarówno dla testu Singleton jak i PerThread osiągnął czas 4 ms. Dla Transient zanotował 2 ms, a dla TransientSingleton 1 ms. Windsor natomiast dla trzech pierwszych testów (Singleton, Transient oraz TransientSingleton) ma czas zbliżony do 1 ms, a dla ostatniego testu (PerThread) osiąga 2 ms.

#### 4.4.3 Wyniki dla Singleton

| Ilość          | 1   |     |     | 10  |     |     |
|----------------|-----|-----|-----|-----|-----|-----|
|                | min | max | avg | min | max | avg |
| Autofac        | 0   | 0   | 0   | 0   | 0   | 0   |
| DryLoc         | 10  | 10  | 10  | 10  | 10  | 10  |
| Grace          | 16  | 17  | 16  | 16  | 17  | 16  |
| LightInject    | 14  | 14  | 14  | 14  | 15  | 14  |
| Ninject        | 21  | 22  | 21  | 21  | 22  | 21  |
| NiquIoCPartial | 8   | 9   | 9   | 8   | 9   | 8   |
| NiquIoCFull    | 18  | 19  | 18  | 18  | 19  | 18  |
| SimpleInjector | 11  | 12  | 11  | 11  | 12  | 11  |
| StructureMap   | 49  | 50  | 49  | 49  | 50  | 49  |
| Unity          | 52  | 53  | 52  | 52  | 54  | 52  |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   |

Tab. 20: Wyniki testów dla operacji Singleton dla Przypadku testowego D

W porównaniu do poprzednich przypadków testowych, różnice czasów dla poszczególnych rozwiązań, są tutaj dużo bardziej zauważalne. Co więcej wyniki dla 1 jak i 10 operacji są praktycznie takie same. Ponownie najlepiej dla tego testu radzą sobie Autofac i

Windsor. Kolejne miejsca zajmują NiquIoCPartial, DryLoc oraz SimpleInjector. Różnice czasów pomiędzy tymi trzema rozwiązaniami są niewielkie (około 1 milisekunda). Na miejscu 6 znalazł się LightInject, a zaraz za nim Grace. Trochę słabiej od nich poradził sobie NiquIoCFull. Grupę rozwiązań ze średnimi czasami zamyka Ninject. Ostatnie dwa wyniki, z bardzo dużą różnicą w stosunku do pozostałych rozwiązań, uzyskały StructureMap i Unity.

#### 4.4.4 Wyniki dla Transient

| Ilość          | 1    |      |      | 10    |       |       |
|----------------|------|------|------|-------|-------|-------|
|                | min  | max  | avg  | min   | max   | avg   |
| Autofac        | 136  | 160  | 147  | 1161  | 1279  | 1188  |
| DryLoc         | 979  | 996  | 982  | 1013  | 1025  | 1015  |
| Grace          | 1413 | 1472 | 1426 | 1465  | 1558  | 1495  |
| LightInject    | 797  | 831  | 812  | 841   | 878   | 853   |
| Ninject        | 1016 | 1179 | 1039 | 10072 | 11599 | 10307 |
| NiquIoCPartial | 36   | 38   | 37   | 285   | 291   | 287   |
| NiquIoCFull    | 566  | 576  | 569  | 619   | 661   | 631   |
| SimpleInjector | 177  | 179  | 178  | 205   | 212   | 206   |
| StructureMap   | 102  | 110  | 103  | 586   | 603   | 591   |
| Unity          | 150  | 200  | 152  | 1073  | 1183  | 1080  |
| Windsor        | 177  | 261  | 182  | 1797  | 2421  | 1827  |

Tab. 21: Wyniki testów dla operacji Transient dla Przypadku testowego D

W przeciwieństwie do poprzedniego testu, tutaj różnica w czasach dla 1 i 10 operacji jest bardzo duża. Dla niektórych rozwiązań wzrost jest 10-krotny, a dla niektórych o kilka procent.

Gdy mamy 1 operację zauważalnie najlepiej radzi sobie NiquIoCPartial. Osiągnął on czasy ponad 3 razy mniejszy, niż StructureMap, który zajął drugie miejsce. Na kolejnych pozycjach z podobnymi czasami znalazły się: Autofac, Unity, SimpleInjector oraz Windsor. Dość duży czas zanotował NiquIoCFull i osiągnął miejsce 7. Dalej w rankingu jest LightInject. Ostatnie trzy miejsca zajęły: DryLoc, Ninject oraz Grace.

Przy 10 operacjach ranking bardzo się zmienia. Na prowadzenie wysunął się SimpleInjector, a na drugie miejsce spadł NiquIoCPartial. Podium zamyka StructureMap. Nieco słabszy czas od niego uzyskał NiquIoCFull, jednak zanotował on wzrost czasu o około 10%, a StructureMap ten wzrost miał ponad 5-krotny. Na kolejnych dwóch miejscach, również z niedużym wzrostem, znalazły się LightInject oraz DryLoc. Miejsce 8 zajął Autofac. Mały wzrost miał także Grace, ale bardzo duży czas dla 1 operacji spowodował, że wyprzedził jedynie Windsor i Ninject, które zajęły dwie ostatnie pozycje.



#### 4.4.5 Wyniki dla TransientSingleton

| Ilość          | 1    |      |      | 10   |      |      |
|----------------|------|------|------|------|------|------|
|                | min  | max  | avg  | min  | max  | avg  |
| Autofac        | 77   | 84   | 78   | 893  | 964  | 910  |
| DryLoc         | 905  | 922  | 910  | 936  | 983  | 942  |
| Grace          | 374  | 386  | 376  | 381  | 396  | 383  |
| LightInject    | 1285 | 1358 | 1310 | 1297 | 1360 | 1316 |
| Ninject        | 503  | 572  | 511  | 4732 | 5620 | 4818 |
| NiquIoCPartial | 17   | 18   | 17   | 100  | 104  | 101  |
| NiquIoCFull    | 144  | 151  | 145  | 151  | 159  | 152  |
| SimpleInjector | 85   | 87   | 86   | 91   | 93   | 91   |
| StructureMap   | 60   | 69   | 61   | 165  | 173  | 167  |
| Unity          | 115  | 140  | 116  | 683  | 771  | 689  |
| Windsor        | 85   | 102  | 87   | 852  | 965  | 864  |

Tab. 22: Wyniki testów dla operacji TransientSingleton dla Przypadku testowego D

Wyniki dla tego testu bardzo przypominają poprzednie rezultaty.

Dla 1 operacji z bardzo dużą przewagą na prowadzeniu jest NiquIoCPartial. Na podium znalazły się również StructureMap i Autofac. Kolejne dwa miejsca z podobnymi czasami zajmują SimpleInjector oraz Windsor. Dalej jest Unity, a kawałek za nim NiquIoCFull. Kolejne rozwiązania mają już bardzo dużo różnice czasów w stosunku do reszty. Miejsce 8 zajął Grace, a ranking zamykają: Ninject, DryLoc i LightInject.

Gdy mamy 10 operacji ranking ulega sporym zmianom. Trochę lepiej niż NiquIoCPartial radzi sobie SimpleInjector i to on znalazł się na prowadzeniu. Podium zamyka NiquIoCFull. Niedużo gorszy wynik od niego uzyskał StructureMap. Z lekkim wzrostem czasu na miejsce 5 awansował Grace. Pozostałe rozwiązania to już spore większe rezultaty. Najpierw jest Unity, a za nim z podobnymi wynikami znalazły się: Windsor, Autofac i DryLoc. Ostatnie dwa miejsca przypadły LightInject i Ninject.

#### 4.4.6 Wyniki dla PerThread

| Ilość          | 1    |      |      | 10   |      |      |
|----------------|------|------|------|------|------|------|
|                | min  | max  | avg  | min  | max  | avg  |
| Autofac        | 0    | 0    | 0    | 0    | 0    | 0    |
| DryIoc         | 1042 | 1051 | 1045 | 1042 | 1051 | 1044 |
| Grace          | 28   | 28   | 28   | 28   | 28   | 28   |
| LightInject    | -1   | -1   | -1   | -1   | -1   | -1   |
| Ninject        | 21   | 22   | 21   | 21   | 22   | 21   |
| NiquIoCPartial | 8    | 9    | 9    | 8    | 10   | 9    |
| NiquIoCFull    | 18   | 18   | 18   | 18   | 18   | 18   |
| SimpleInjector | 46   | 48   | 46   | 46   | 48   | 46   |
| StructureMap   | 49   | 51   | 49   | 49   | 51   | 49   |
| Unity          | 52   | 53   | 52   | 52   | 53   | 52   |
| Windsor        | 0    | 0    | 0    | 0    | 0    | 0    |

Tab. 23: Wyniki testów dla operacji PerThread dla Przypadku testowego D

Dla tego testu największym rozczarowaniem okazał się LightInject. Osiągnął on nieporównywalnie większe czasy niż cała reszta. Bardzo duże wyniki zanotował również DryIoc.

Najlepiej poradziły sobie Autofac i Windsor. NiquIoCPartial znalazł się na 3 miejscu, a dalej, z czasem dwukrotnie większym, NiquIoCFull. Trochę gorszy wynik od niego uzyskał Ninject. Kolejne miejsce zajął Grace. Ostatnia trójka (nie licząc wcześniej wspomnianych DryIoc oraz LightInject), to: SimpleInjector, StructureMap i Unity.

#### 4.4.7 Wyniki dla FactoryMethod

| Ilość          | 1   |     |     | 10  |     |     |
|----------------|-----|-----|-----|-----|-----|-----|
|                | min | max | avg | min | max | avg |
| Autofac        | 0   | 0   | 0   | 0   | 0   | 0   |
| DryIoc         | 0   | 0   | 0   | 0   | 0   | 0   |
| Grace          | 0   | 0   | 0   | 0   | 0   | 0   |
| LightInject    | 0   | 0   | 0   | 0   | 0   | 0   |
| Ninject        | 0   | 0   | 0   | 0   | 0   | 0   |
| NiquIoCPartial | 0   | 0   | 0   | 0   | 0   | 0   |
| NiquIoCFull    | 0   | 0   | 0   | 0   | 0   | 0   |
| SimpleInjector | 0   | 0   | 0   | 0   | 0   | 0   |
| StructureMap   | 0   | 0   | 0   | 0   | 0   | 0   |
| Unity          | 0   | 0   | 0   | 0   | 0   | 0   |
| Windsor        | 0   | 0   | 0   | 0   | 0   | 0   |

Tab. 24: Wyniki testów dla operacji FactoryMethod dla Przypadku testowego D

Tym razem wszystkie rozwiązania mają czasy na poziomie 0 ms.

## 5 Podsumowanie

Zaproponowane przeze mnie rozwiązania bardzo dobrze realizują postawione we wstępie pracy cele. Dla wszystkich przypadków testowych, dla każdego z rodzaju rejestracji jedno z moich rozwiązań radziło sobie najlepiej albo niewiele odbiegało od najlepszego rozwiązania (zawsze któreś z moich rozwiązań było w pierwszej trójce, a czasami nawet oba). Możliwość mieszania użyć zaprezentowanych przeze mnie rozwiązań (w jednym projekcie można korzystać z obu rozwiązań niezależnie) sprawia, że NiquIoC jest najwydajniejszą implementacją wzorca wstrzykiwania zależności dla złożonych grafów zależności.

Dodatkowo wyniki moich testów pokazują, że rozwiązanie Ninject jest najmniej wydajne i nie zaleca się go używać przy bardzo rozbudowanych grafach. Jeśli chodzi natomiast o rozwiązania, które radzą sobie dobrze, to nie ma drugiego takiego (poza moim), które zawsze dawałoby zadowalające rezultaty. SimpleInjector dość dobrze radzi sobie dla testów Singleton, Transient i TransientSingleton, ale osiąga słabe rezultaty dla PerThread i FactoryMethod. LightInject z kolei jest rozwiązaniem, które wraz ze wzrostem liczby operacji ma najmniejszy wzrost czasu. Jednakże w wielu przypadkach dość duże czasy dla małej liczby operacji powodują, że wyniki dla wielu operacji są bardzo duże. Takim pośrednim rozwiązaniem jest natomiast Grace. To rozwiązanie ma zarówno nieduże wzrosty czasów jak i nieduże wyniki dla małej ilości operacji, jednak mimo wszystko są one dużo słabsze niż dla NiquIoC.

### 5.1 Kontynuacja projektu

Zaprezentowane przeze mnie rozwiązania są bardzo wydajne, ale mają swoje ograniczenia. Jednym z nich jest między innymi konieczność rejestracji wszystkich typów. Tę funkcjonalność można by rozszerzyć, aby nie trzeba było rejestrować klas. Algorytm sam rejestrowałby klasy gdyby okazywały się potrzebne - większość z rozwiązań dostarcza taką funkcjonalność. Kolejnym rozszerzeniem mogłoby być automatyczna rejestracja wszystkich typów z danego assembly. Osobom wykorzystującym moje rozwiązanie zaoszczędziłoby to sporo czasu na rejestrowanie wszystkich typów. Dodatkowo dodając kolejny typ do danego assembly, nie trzeba byłoby się przejmować zarejestrowaniem go. Jeszcze jednym dodatkiem wydaje się danie możliwości rejestracji wielu implementacji danego interfejsu. To jest również często spotykana funkcjonalność. Na koniec warto również wspomnieć o metodzie BuildUp dla NiquIoCFull - w mojej pracy tego zabrakło, a to również mogłoby być jednym z kolejnych rozszerzeń.

## Spis rysunków

|    |   |    |
|----|---|----|
| 1  | Przykładowa klasa z atrybutem DependencyConstructor . . . . .                   | 7  |
| 2  | Przykładowa klasa z atrybutem DependencyMethod . . . . .                        | 7  |
| 3  | Przykładowa klasa z atrybutem DependencyProperty . . . . .                      | 8  |
| 4  | Interfejs IContainer . . . . .  | 10 |
| 5  | Interfejs IContainerMember . . . . .  | 11 |
| 6  | Interfejs IObjectLifetimeManager . . . . .                                      | 12 |
| 7  | Klasa ContainerMember . . . . .   | 15 |
| 8  | Metoda Resolve klasy PartialEmitFunction . . . . .                              | 16 |
| 9  | Metoda GetObject klasy PartialEmitFunction . . . . .                            | 16 |
| 10 | Metoda CreateInstanceFunction klasy PartialEmitFunction . . . . .               | 17 |
| 11 | Metoda CreateObjectFunction klasy PartialEmitFunction . . . . .                 | 18 |
| 12 | Metoda Resolve klasy FullEmitFunction . . . . .                                 | 18 |
| 13 | Metoda GetObject klasy FullEmitFunction . . . . .                               | 19 |
| 14 | Metoda CreateInstanceFunction klasy FullEmitFunction . . . . .                  | 20 |
| 15 | Metoda ValidateTypesCache klasy FullEmitFunction . . . . .                      | 20 |
| 16 | Metoda CreateObjectFunction klasy FullEmitFunction . . . . .                    | 21 |
| 17 | Metoda CreateObjectFunctionPrivate klasy FullEmitFunction . . . . .             | 22 |
| 18 | Metoda IsTransient klasy FullEmitFunction . . . . .                             | 22 |
| 19 | Metoda IsSingleton klasy FullEmitFunction . . . . .                             | 23 |
| 20 | Metoda AddObjectCreatedByObjectLifetimeManager klasy FullEmitFunction . . . . . | 24 |
| 21 | Graf zależności dla testu A. . . . .  | 27 |
| 22 | Graf zależności dla testu B. . . . .  | 33 |
| 23 | Graf zależności dla testu C. . . . .  | 39 |
| 24 | Graf zależności dla testu D. . . . .  | 45 |

## Spis tabel

|    |  |    |
|----|--|----|
| 1  | Wyniki testów dla operacji Register dla Przypadku testowego A . . . . .  | 28 |
| 2  | Wyniki testów dla operacji Singleton dla Przypadku testowego A . . . . . | 29 |
| 3  | Wyniki testów dla operacji Transient dla Przypadku testowego A . . . . . | 29 |
| 4  | Wyniki testów dla operacji TransientSingleton dla Przypadku testowego A  | 30 |
| 5  | Wyniki testów dla operacji PerThread dla Przypadku testowego A . . . .   | 31 |
| 6  | Wyniki testów dla operacji FactoryMethod dla Przypadku testowego A       | 32 |
| 7  | Wyniki testów dla operacji Register dla Przypadku testowego B . . . . .  | 34 |
| 8  | Wyniki testów dla operacji Singleton dla Przypadku testowego B . . . . . | 35 |
| 9  | Wyniki testów dla operacji Transient dla Przypadku testowego B . . . . . | 35 |
| 10 | Wyniki testów dla operacji TransientSingleton dla Przypadku testowego B  | 36 |
| 11 | Wyniki testów dla operacji PerThread dla Przypadku testowego B . . . . . | 37 |
| 12 | Wyniki testów dla operacji FactoryMethod dla Przypadku testowego B .     | 38 |
| 13 | Wyniki testów dla operacji Register dla Przypadku testowego C . . . . .  | 40 |
| 14 | Wyniki testów dla operacji Singleton dla Przypadku testowego C . . . . . | 40 |
| 15 | Wyniki testów dla operacji Transient dla Przypadku testowego C . . . . . | 41 |
| 16 | Wyniki testów dla operacji TransientSingleton dla Przypadku testowego C  | 42 |
| 17 | Wyniki testów dla operacji PerThread dla Przypadku testowego C . . . .   | 43 |
| 18 | Wyniki testów dla operacji FactoryMethod dla Przypadku testowego C .     | 43 |
| 19 | Wyniki testów dla operacji Register dla Przypadku testowego D . . . . .  | 46 |
| 20 | Wyniki testów dla operacji Singleton dla Przypadku testowego D . . . . . | 46 |
| 21 | Wyniki testów dla operacji Transient dla Przypadku testowego D . . . . . | 47 |
| 22 | Wyniki testów dla operacji TransientSingleton dla Przypadku testowego D  | 48 |
| 23 | Wyniki testów dla operacji PerThread dla Przypadku testowego D . . . .   | 49 |
| 24 | Wyniki testów dla operacji FactoryMethod dla Przypadku testowego D       | 49 |

## Literatura

- [1] Ian Griffiths, Matthew Adams, Jesse Liberty, C#. Programowanie. Wydanie VI, 2012
- [2] Mark Seemann, Dependency Injection in .NET, 2012
- [3] Serge Lidin, Expert .NET 2.0 IL Assembler, 2006
- [4] Robert C. Martin, Czysty kod. Podręcznik dobrego programisty, 2014