## 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

# Spis treści

1	Wst	ęp		3
	1.1	Cel pr	acy	3
	1.2	Układ	l pracy	3
2	Wst	rzykiw	ranie zależnosci	4
	2.1	Wstęp	)	4
	2.2	Imple	mentacje przemysłowe	4
3	Imp	lement	tacja	6
	3.1	Środo	wisko pracy	6
	3.2	Micro	soft Intermediate Language	6
	3.3	Reflec	tion.Emit	7
	3.4	Opis .		7
		3.4.1	Register	7
		3.4.2	Resolve	8
		3.4.3	BuildUp	8
	3.5	Rozwi	iązanie	9
		3.5.1	Rozwiązanie 1 - PartialEmitFunction	9
		3.5.2	Rozwiązanie 2 - FullEmitFunction	10
4	Test	y wyda	ajnościowe	12
		4.0.1	Register as Singleton	12
		4.0.2	Register as Transient	12
		4.0.3	Register as PerThread	12
	4.1	Przyp	adek testowy A	12
		4.1.1	Opis	12
		4.1.2	Wyniki dla Singleton	14
		4.1.3	Wyniki dla Transient	15

5	Pods	sumow	anie	24
		4.3.4	Wyniki dla PerThread	23
		4.3.3	Wyniki dla Transient	22
		4.3.2	Wyniki dla Singleton	21
		4.3.1	Opis	19
	4.3	Przyp	adek testowy C	19
		4.2.4	Wyniki dla PerThread	19
		4.2.3	Wyniki dla Transient	18
		4.2.2	Wyniki dla Singleton	18
		4.2.1	Opis	16
	4.2	Przypa	adek testowy B	16
		4.1.4	Wyniki dla PerThread	16

## 1 Wstęp

## 1.1 Cel pracy

Wstrzykiwanie zależnośc 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 funkcjolanosci z przestrzeni nazw Reflection.Emit. W tej pracy zostaną przedstawione dwa rozwiązania.

## 1.2 Układ pracy

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

## 2 Wstrzykiwanie zależnosci

#### 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 zależnóści nie jestem celem samym w sobie, raczej jest to środek do celu. Ostatecznie celem większości technik programowania jest dostarczenie jak najwydajniej działającego oprogramowania. Jednym z aspektów tego jest napisanie utrzymywalnego kodu.

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

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

Wstrzykiwanie zależności może odbywać się na 3 sposoby

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

Dodać jak działa/jak wygląda każde z tych wstrzykiwań?

## 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.palmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison). Dane zostały wzięte z dnia 21-02-2017. 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 też 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 2015 Comunnity
- ReSharper
- Dia

Testy zostały wykonane we frameworku XUnity, który jest wbudowany w Visual Studio.

Rozwiązanie opiera się na użyciu operacji z przestrzeni nazw System.Reflection.Emit.

## 3.2 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.

#### 3.3 Reflection.Emit

Przestrzeń nazw Reflection.Emit pozwala ona 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 wczeniej zdefiniowanych operacji IL.

### 3.4 Opis

Aplikacja składa się z 1 projektu i 4 projektów na potrzeby testów. Rozwiązanie jest skomplikowana i aby mieć pewność, że działa w pełni dobrze zostało stworzone ponad 1000 testów jednostkowych.

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

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

Pierwsze cztery metody służą do rejestracji typów do kontenera, piąta - Resolve, do zwracana obiektów wczeniej zarejestrowanego typu, a szczósta do uzupełnienia istniejącej istancji obiektu z wykorzystaniem wstrzykiwania zależności przez metodę i właściwość.

#### 3.4.1 Register

W pierwszej metodzie możemy zarejestrować zwykłe klasy. W drugiej interfejsy i klasy, które implementują dany interfejs. W trzeciej metodzie rejestrujemy klasę jako fabrykę obiektów - funkcję, która ma nam zwrócić pożądany obiekt. W czwartej natomiast możemy zarejestrować konkretną instancję danej klasy. W mojej implementacji kontenera każdy typ może być zarejestrowany tylko raz - ponowna rejestracja tego samego typu nadpisuje istniejącą rejestrację.

Każda z tych czterech pierwszy metod zwraca interfejs IContainerMember, który umożliwia nam zarejestrowanie danej klasy lub interfejsu z określonym menadżerem czasu

życia (podanie implementacji interfejsu IObjectLifetimeManager). Jest to po to, ponieważ dla różnych przypadków biznesowych możemy potrzebować, aby obiekt danego typu miał konretny czas życia. Interfejs IContainerMember daje nam możliwość ustawienia czasu życia na:

- Singleton
- Transient
- PerThread
- PerHttpContext
- Custom

Opcje: Singleton, Transient, PerThread i PerHttpContext to wbudowane implementacje interfejsu IObjectLifetimeManager. Custom, to opcja na podanie przez użytkownika jego własnej implementacji tego interfejsu. W mojej implementacji kontenera każdy typ domyślnie ma czas życia Transient.

Interfejs IObjectLifetimeManager zawiera w sobie następujące metody:

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

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

#### 3.4.2 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.

#### 3.4.3 BuildUp

Szósta metoda to taki dodatek - gdy mamy stworzony obiekt, ale nie jest w pełni uzupełniony, wtedy możemy go zbudować (używająć odpowiednio PartialEmitFunction

lub FullEmitFunction). Z tą metodą są powiązane bezpośrednio dwa pojęcia - wstrzykiwanie przez metodę i wstrzykiwanie przez właściwość. Do tego celu zostały stworzone dwa atrybuty:

- DependencyMethod (dla metod)
- DependencyProperty (dla propercji)

Podczas operacji BuildUp uzupełniane są wywoływane wszystie metody i właściwości, które mają te atrybuty. BuildUp jest również wykonywany podczas operacji Resolve.

Warto tutaj odnotować, że ze względu na szczegóły implementacyjne tylko jedno z moich rozwiązań wspiera operację BuildUp - PartialEmitFunction. W FullEmitFunction ta funckonalność nie została zaimplementowana. Jest to spowodowane skomplikowaniem FullEmitFunction i małą potrzebą biznesową używania operacji BuildUp.

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

- 1. Konstruktor z atrybutem DependencyConstrutor
- 2. Konstrutkro z największą liczbą parametrów

Jeśli jest kilka konstrutkrów z atrybutem DependencyConstrutor albo kilka z największą liczbą parametrów, to rzucany jest wyjątek.

## 3.5 Rozwiązanie

Aby kontener działał wydajnie dla złożonych grafów zależności, należy jak najwięcej informacji przechowywać w cache i należy to robić mądrze. W tym celum wykorzystałem Reflection. Emit, aby zapamiętać ciągu operacji niezbędnych do stworzenia obiektu nowej klasy (na potrzeby metody Resolve). Wykorzystałem to na dwa sposoby, które zostały opisane poniżej.

#### 3.5.1 Rozwiązanie 1 - PartialEmitFunction

W pierwszym rozwiązanu, które nazwałem PartialEmitFunction tworzę delegata z wykorzystaniem Reflection.Emit. Delegat ten jako parametr przyjmuje listę obiektów, które są potrzebne do stworzenia obiektu danej klasy wykorzystując odpowiedni kon-

struktor. Jeśli kontener do stworzenia obiektu danej klasy wybrał konstruktor bezparametrowy, to do takiego delegata trafi pusta lista. Jeśli natomiast został wybrany konstruktor, który w parametrze przynmuje obiekty jakiś typów, to lista tych obiektów w odpowiedniej kolejności zostanie przekazana do delegata (np. obiekt, który chcemy stworzyć w konstrutkorze który został wybrany potrzebuje obiektu klasy A i B, to do delegata zostanie przekazana lista zawierająca obiekt klasy A i B w kolejności takiej, jakiej są one zdefiniowane z konstruktorze).

Pseudokod tego rozwiązana wygląda następująco:

- 1. Dla każdego argumentu umieść ten argument na stosie
- 2. Na stosie umieść konstruktor docelowego typu
- 3. Wywołaj konstrutkor i stworzony obiekt umięść na szczycie stosu
- 4. Zwróć obiekt ze szczytu stosu

To rozwiążanie nazwałem "Partial", ponieważ tylko część operacji niezbędnych do stworzenia obiektu jest zakodowanych w języku IL (pobranie argumentów i stworzenie obiektu). W tym rozwiążaniu musimy wcześniej stworzyć obiekty, które są wymagane prezz konstruktor docelowego obiektu.

To rozwiązanie powinno lepiej się sprawdzać w sytuacjach, gdy wierzchołki w grafie zależnoci czesto się powtarzają i w aplikacji nie wykonujemy zbyć często operacji "Resolve".

#### 3.5.2 Rozwiązanie 2 - FullEmitFunction

W drugim rozwiązaniu, które nazwałem FullEmitFunction tworzę delegata bezparametrowego. To rozwiązanie samo tworzy wszystkie obiekty, które są mu potrzebne do stworzenia docelowego obiektu. Działa on rekurencyjnie i najpierw na stosie umieszcza wszystkie operacje niezbędne do stworzenia wszystkich "podobiektów" (obiektów wymaganych w konstruktorze docelowego obiektu), a następnie tworzy docelowy obiekt. Pseudokod tego rozwiązania wygląda następująco:

- 1. Czy konstruktor danego typu potrzebuje jakiś argument?
  - (a) Tak Dla każdego parametru konstruktora wywołaj rekurencyjnie funkcję z argumentem wejściowym jako typ danego parametru.
  - (b) Nie Idź dalej.
- 2. Na stosie umieść konstruktor docelowego typu
- 3. Wywołaj konstrutkor i stworzony obiekt umięść na szczycie stosu
- 4. Zwróć obiekt ze szczytu stosu

Jak łatwo zauważyć, to rozwiązanie jest pełne ("Full"), ponieważ tworzenie wszystkich niezbędnych obiektów jest zakodowane w języku IL w jednej metodzie.

To rozwiązanie powinno lepiej się sprawdzić w sytuacjach, gdy wierzchołki w grafie zależności rzadko się powtarzają i w aplikacji dużo razy wykonujemy operację "Resolve".

## 4 Testy wydajnościowe

Do przeprowadzania testów wydajnościowych stworzyłem osobną aplikację w której zaimplementowałem 3 przypadki testowe (przypadek testowy A, przypadek testowy B i przypadek testowy C). Każdy z przypadków testowych sprawdza czas wykoniania operacji "Resolve" dla różnych rodzajów rejestracji (operacji Register). Każdy z przypadków testowych wykonuje testy dla następujący rejestracji:

- Register as Singleton
- Register as Transient
- Register as PerThread (dla niektórych przemysłowych implementacji PerScope)

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

#### 4.0.1 Register as Singleton

Każdy typ jest zarejestrowany jako "Singleton", czyli obiekt jest tworzony raz i potem cały czas jest zwracany ten sam obiekt.

#### 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 PerThread

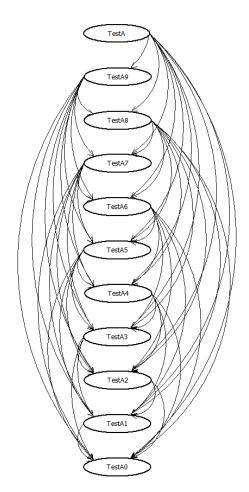
Każdy typ jest zarejestrowany jako "PerThread", czyli obiekt jest tworzony raz dla każdego wątku, a następnie w obrębie danego wątku jest on cały czas zwracany.

## 4.1 Przypadek testowy A

## 4.1.1 Opis

W tym teście mamy zdefiniowanych 11 typów i każdy z nich przymuje w konstruktorze od jeden mniej parametr mniej niż typ poprzedni (czyli przyjmują one kolejno od 10 do 0 parametrów w konstruktorze). Typem głównym, a zarazem typem o największej

licznie parametrów, jest typ "TestA". Przyjmuje on w konstruktorze 10 parametrów, kolejno następujących typów: "TestA0", "TestA1", "TestA2", "TestA3", "TestA4", "TestA5", "TestA6", "TestA7", "TestA8", "TestA9". Każdy z tych 10 typów w konstruktorze przyjmuje tyle obiekt, jaki ma numerek w nazwie (czyli obiekt typu "TestA0" ma konstruktore bezparametrowy, obiekt typy "TestA1" ma konstruktor z jednym parametrem; i tak dalej aż do typu "TestA9", który ma konstruktor z dziewięcioma parametrami). Każdy z tych typów jako parametry w konstruktorze przyjmuje kolejne obiekty typów z niższym numerkiem (czyli obiekt typu "TestA1" w konstruktorze przyjmuje parametr typu "TestA0", obiekt typu "TestA2" przymuje w konstruktorze obiekty typu "TestA0" i "TestA1"; i tak dalej aż do typu "TestA9", który w konstruktorze przyjmuje parametry z typami od "TestA0" do "TestA8"). Graf zależności poszczególnych typów został przedstawiony na Rys. 1.



Rys. 1: Graf zależności dla testu A.

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

• TestA0 - 1 obiekt,

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

Ilość		1			10			100			1000	
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	0	0	0	0	0	0	0	0	0	0	0	0
DryIoc	2	3	2	2	3	2	2	2	2	2	2	2
Grace	2	2	2	2	3	2	2	3	2	2	4	3
LightInject	2	2	2	2	2	2	2	2	2	2	2	2
Ninject	3	3	3	3	3	3	3	3	3	6	6	6
NiquIoCPartial	1	1	1	1	1	1	1	1	1	1	1	1
NiquIoCFull	2	2	2	2	2	2	2	2	2	2	2	2
SimpleInjector	2	2	2	2	2	2	2	2	2	2	2	2
StructureMap	9	9	9	9	9	9	9	9	9	10	10	10
Unity	7	7	7	7	7	7	7	7	7	8	8	8
Windsor	0	0	0	0	0	0	0	0	0	0	0	0

Najlepiej poradziły sobie najpopularniejsze rozwiązania - Windsor i Aurofac. Rozwiązanie NiquIoCPartial uplasowało się na 3 miejscu. Najsłabiej poradził sobie rozwiązania StructureMap, Unity oraz Ninject.

#### 4.1.3 Wyniki dla Transient

Ilość		1			10			100			1000	
	min	max	avg	min	max	avg	min	max	avg	min	max	avg
Autofac	0	0	0	6	7	6	61	70	65	615	687	632
DryIoc	14	15	15	15	16	15	16	18	17	30	31	30
Grace	16	18	16	16	17	16	19	20	19	39	40	40
LightInject	10	10	10	10	10	10	11	12	11	19	19	19
Ninject	10	11	10	86	90	88	858	951	873	8673	9035	8812
NiquIoCPartial	1	1	1	3	3	3	19	19	19	172	174	172
NiquIoCFull	8	8	8	8	9	8	9	10	9	18	18	18
SimpleInjector	13	13	13	13	14	13	15	15	15	28	29	29
StructureMap	10	10	10	13	14	14	52	54	52	415	420	417
Unity	8	8	8	15	16	15	84	86	84	767	820	773
Windsor	1	1	1	16	18	17	155	169	159	1528	1602	1548

Gdy mamy tylko 1 operację najlepiej radzi sobie Autofac, a zaraz za nim NiquIoCPartial i Windsor. Najsłabiej Grace, DryIoc i SimplyInjector.

Dla 10 operacji sytuacja lekko zaczyna się 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, dużo 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).

Gdy mamy 100 operacji na prowadzenie wysuneły się mniej popularne rozwiązania, które prawdopodobniej mają najlepsze cachowanie. Na pierwszym miejscu jest Niqu-IoCFull, a kolejne miejsca to LightInject i SimpleInjector. Do grona najsłabszym (Ninject, Windsor, Unity) w tym przypadku dołączył również Autofac, który z drugiego miejsca spadł na ósme.

Dla przypadku z 1000 operacji do czołówki dołączył DryIoc (czasy zbliżone do SimpleInjector). Na pierwszym miejscu wciąż pozostaje NiquIoCFull, a zaraz za nim LightInject i SimpleInjector. Pozostałe rozwiązania mają czasy od kilka do kilkanaście razy gorsze.

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

#### 4.1.4 Wyniki dla PerThread

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

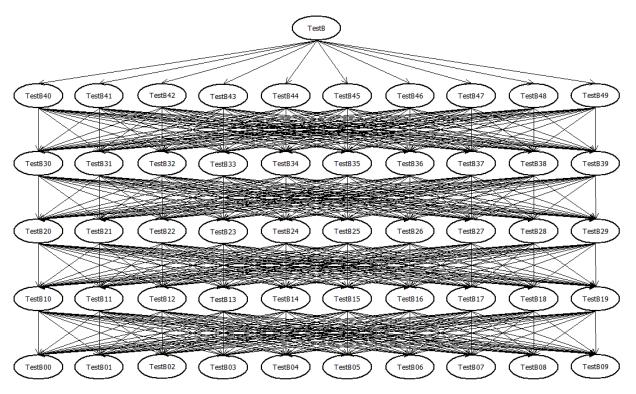
Czasy dla tego przypadku powinny być zbliżone do czasów dla Singleton, ponieważ wszystko było uruchamiane w jednym wątku. Niestety częć rozwiązań sobie z tym nie poradziła. Czasy zbliżone do czasów dla Singletona posiadają: Autofac, Grace, Ninject, NiquIoCPartial, NiquIoCFull, StructureMap, Unity i Windsor,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

Jeśli chodzi o najlepsze rozwiązania, to wyglądają one tak samo jak dla Singleton - Windso,i Aurofac i NiquIoCPartial. Najsłabiej poradził sobie rozwiązania LightInject i DryIoc. Także w drugiej części tabeli (pod względem czasów) znalazły się Unity, SimpleInjector i StructureMap.

## 4.2 Przypadek testowy B

#### 4.2.1 Opis

W tym teście mamy zdefiniowanych 51 typów. 10 z tych typów ma konstruktor bezparametrowy, a pozostałe 41 ma konstuktor z dziesięcioma parametrami. Typem głównym jest typ "TestB". Obiekt tego typy w konstruktorze przymuje 10 innych obiektów, kolejno następujących typów: "TestB40", "TestB41", "TestB42", "TestB43", "TestB44", "TestB45", "TestB46", "TestB46", "TestB47", "TestB48", "TestB49". Każdy z tych 10 typów w konstruktorze przyjmuje 10 obiektów o takich samych typach, ale z pierwszym numerkiem o 1 mniejszym (czyli obiekty typów od "TestB40" do "TestB49", przyjmują w konstruktorze obiekty typów od "TestB30" do "TestB39"). Dla typów od "TestB30" do "TestB39" zasad z konstruktorami wygląda tak samo.Na końcu dochodzimy do typów od "TestB00" do "TestB09", które mają konstruktor bezparametrowy. Rys. 2 przedstawia graf zależności typów dla tego przypadku testowego.



Rys. 2: Graf zależności dla testu B.

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

- typy od TestB00 do TestB09 1 obiekt,
- typy od TestB10 do TestB19 11 obiektów (obiekt danego typu plus 10 obiektów typów od TestB00 do TestB09),
- typy od TestB20 do TestB29 111 obiektów (obiekt danego typu plus 10 obiektów typów od TestB10 do TestB19),
- typy od TestB30 do TestB39 1 111 obiektów,
- typy od TestB40 do TestB49 11 111 obiektów,
- TestB 111 111 obiektów.

Zatem tworząc obiekt typu TestB, tworzymy: 1 obiekt typu TestB, 10 obiektów typów od TestB40 do TestB49, 100 obiektów typów od TestB30 do TestB39, 1 000 obiektów typów od TestB20 do TestB29, 10 000 obiektów typów od TestB10 do TestB19, 100 000 obiektów typów od TestB00 do TestB00 - co daje w sumie 111 111 obiektów..

#### 4.2.2 Wyniki dla Singleton

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

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

#### 4.2.3 Wyniki dla Transient

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

Dla tego testu rozbierzność czasów jest dość duża.

Gdy mamy tylko 1 operację najlepiej radzi sobie NiquIoCPartial. Czasy pozostały są znacząco większe. Najsłabiej poradziły sobie Grace, Ninject, DryIoc i LightInject. Dla 10 operacji znacząco najlepiej radzi sobie SimplyInjector i NiquIoCPartial. Czołówkę zamykają StrucutreMap i NiquIoCFull. Pozostałe rozwiązania miały podobne, dużo słabsze czasy. Wyjątkiem jest ponownie Ninject, którego czasy są kilkukrotnie większe. W tym przypadku najmniejszy wzorst czasu miały kolejno rozwiązania: DryIoc, SimpleInjector, LightInject, Grace oraz NiquIoCFull.

#### 4.2.4 Wyniki dla PerThread

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

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

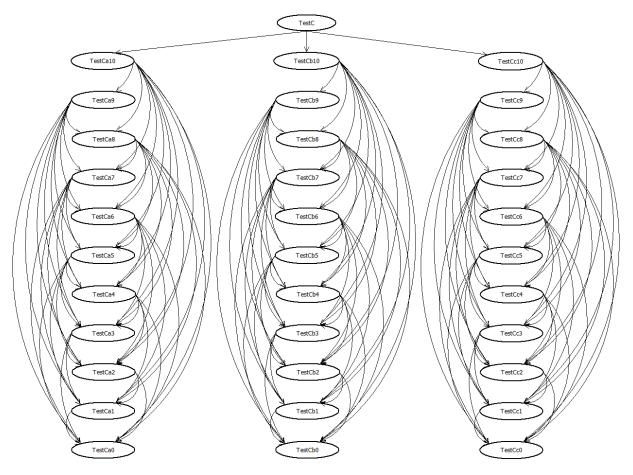
Zarówno dla 1 jak i 10 powtórzeń najlepiej radzi sobie Autofac i Windsor. Podium ponownie zamyka NiquIoCPartial.

Najgorzej poradził sobie tutaj LightInject, który trwał ponad 20 minut (stąd w rozwiążaniu czas "-1"). DryIoc również poradził sobie bardzo słabo - czasy ponad 20 raz gorsze niż rozwiążanie na miejscu o 1 wyższym (Unity).

## 4.3 Przypadek testowy C

#### 4.3.1 Opis

Ten test jest bardzo podobny do przypadku testowego A, tylko dochodzi nam 1 dodatkowy poziom, który wygląda trochę inaczej. W głównym obiekcie "TestC" konstruktor przyjmuje 3 parametry następujących typów: "TestCa10", "TestCb10", "TestCc10". Każdy z tych 3 typów odpowiada typowi "TestA", więc przyjmuje on w konstruktorze 10 parametrów. Dla "TestCa10" są to parametry typów od "TestCa0" do "TestCa9", dla "TestCb10" są to parametry typów od "TestCb9", a dla "TestCc10" są to parametry typów od "TestCc0" do "TestCc9". Zależności tych typów wyglądają tak samo, jak dla typów z przypadku testowego A - rys. 3 przedstawia te zależności.



Rys. 3: Graf zależności dla testu C.

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

- TestCa0 1 obiekt,
- TestCa1 2 obiekty (obiekt typu TestCa1 i obiekt typu TestCa0),
- TestCa2 4 obiekty (obiekt typu TestCa2, obiekt typu TestCa1 2 obiekty, obiekt typu TestCa0 1 obiekt),
- TestCa3 8 obiektów (obiekt typu TestCa3, obiekt typu TestCa2 4 obiekty, obiekt typu TestCa1, obiekt typu TestCa0),
- TestCa4 16 obiektów,
- TestCa5 32 obiektów,
- TestCa6 64 obiektów,
- TestCa7 128 obiektów,

- TestCa8 256 obiektów,
- TestCa9 512 obiektów,
- TestCa10 1 024 obiektów,
- ...(dla TestCb i TestCc sytuacja wygląda dokłądnie tak samo jak dla TestCa),
- TestC 3 073 obiektów.

Zatem tworząc obiekt typu TestC, tworzymy: 1 obiekt typu TestC, 1 obiekt typu TestCa10, TestCb10 i TestCc10,1 obiekt typu TestCa9, TestCb9 i TestCc9, 2 obiekty typu TestCa8, TestCb8 i TestCc8, 4 obiekty typu TestCa7, TestCb7 i TestCc7, 8 obiektów typu TestCa6, TestCb6 i TestCc6, 16 obiektów typu TestCa5, TestCb5 i TestCc5, 32 obiektów typu TestCa4, TestCb4 i TestCc4, 64 obiektów typu TestCa3, TestCb3 i TestCc3, 128 obiektów typu TestCa2, TestCb2 i TestCc2, 256 obiektów typu TestCa1, TestCb1 i TestCc1, 512 obiektów typu TestCa0, TestCb0 i TestCc0 - co daje w sumie 3 073 obiektów.

#### 4.3.2 Wyniki dla Singleton

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

Dla tego przypadku testowego również najlepiej poradziły sobie najpopularniejsze rozwiązania - Aurofac i Windso. Kontener NiquIoCPartial uplasował się tutaj również na 3 miejscu. Najsłabiej poradził sobie rozwiązania StructureMap, Unity oraz Ninject.

#### 4.3.3 Wyniki dla Transient

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

Gdy mamy tylko 1 operację najlepiej radzi sobie Autofac, a zaraz za nim NiquIoCPartial i Windsor. Najsłabiej DryIoc i SimplyInjector.

Gdy mamy 10 operacji znacząco lepiej niż reszta radzi sobie NiquIoCPartial - czasy ponad dwa mniejsze niż dla zajmującego drugie Autofac. Kolejne miejsca należą do NiquIoCFull i LightInject. Pozostałe rozwiązania miałby podobne, trochę słabsze czasy. Najgorzej poradził sobie Ninject, który czas miał ponad 5 razy większy niż rozwiązanie z przedostatnim czasem (Grace).

Tutaj również gdy mamy 100 operacji na prowadzenie wysuneły się mniej popularne rozwiązania. Kolejne miejsca to: NiquIoCFull, LightInject i SimpleInjector. Najsłabiej w tym przypadku poradziłysobie Ninject, Windsor, Unity i Autofac. Grace w 9 miejsca od końca awansował na 6, a NiquIoCPartial z pierwszego spadł na 5.

Dla przypadku z 1000 operacji do czołówki dołączył DryIoc i Grace - czasy trochę słabsze niż SimpleInjector. Pierwsze trzy miejsca nie uległu zmianie.

Tutaj najmniejszy wzorst czasu miały rozwiążania: NiquIoCFull, LightInject, SimpleInjector, DryIoc oraz Grace.

### 4.3.4 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	247	260	252	247	257	250	247	271	255	256	272	263
Grace	13	13	13	13	13	13	13	13	13	13	13	13
LightInject	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Ninject	9	9	9	9	9	9	9	10	9	12	13	12
NiquIoCPartial	4	4	4	4	4	4	4	4	4	4	4	4
NiquIoCFull	8	8	8	8	8	8	8	8	8	8	8	8
SimpleInjector	23	23	23	23	23	23	23	23	23	23	23	23
StructureMap	30	30	30	30	30	30	30	30	30	32	33	33
Unity	23	23	23	23	23	23	23	23	23	24	24	24
Windsor	0	0	0	0	0	0	0	0	0	0	0	0

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

Tak samo jak w przypadku Signleton czołówka wygląda następująco: Aurofac, Windsor i NiquIoCPartial. Kolejne miejsca to NiquIoCFull, Ninject i Grace. Najsłabiej poradził sobie rozwiążania LightInject i DryIoc.

# 5 Podsumowanie

<parę słów na koniec>

# Literatura

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