

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Kierunek: Informatyka

Adrian Mularczyk

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

Praca wykonana pod kierunkiem dr. Wiktora Zychli

Wrocław, 2016

Streszczenie

Implementacja wydajnego wzorca wstrzykiwania zależności dla złożonych grafów zależności

Wstrzykiwanie zależności jest to zbiór zasad projektowania oprogramowania, które umożliwiają luźne powiązania. 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 środowisku .NET z wykorzystaniem przestrzeni nazw Reflection.Emit oraz jest w pełni przetestowana. W pracy zaproponowano dwa równorzędne, niezależne podejścia, oba porównano pod kątem wydajnościowym. Porównanie obejmuje również 5 najbardziej popularnych i 4 najszybsze rozwiązania. Końcowe wyniki wykazały, że z obecnie dostępnych rozwiązań na rynku, zaprezentowane w niniejszej pracy radzi sobie najlepiej.

Abstract

Implementing an efficient dependency injection pattern for complex dependency graphs

Injection Dependency is a set of software design principles that allow for loose binding. This makes the code more extensible and easier to maintain. The purpose of this work was to create such an implementation that would work efficiently for complex dependency graphs. The application was written in a .NET environment using the Reflection.Emit namespace and is fully tested. The paper proposes two equivalent, independent approaches which are compared for performance. The comparison also includes the 5 most popular and 4 fastest solutions. The final results show that from solutions available nowadays on the market, the best solutions are these presented in this paper.

Spis treści

1	Wstęp	7
1.1	Cel pracy	7
1.2	Układ pracy	7
2	Odwrócenie zależności	8
2.1	SOLID	8
2.1.1	Przykład dla zasady DIP	8
2.2	Kontenery wstrzykiwania zależności	10
2.2.1	Przykład	10
3	Wstrzykiwanie zależności	13
3.1	Wstęp	13
3.1.1	Wstrzykiwanie przez konstruktor	13
3.1.2	Wstrzykiwanie przez metodę	13
3.1.3	Wstrzykiwanie przez właściwość	14
3.2	Implementacje przemysłowe	14
4	Implementacja	16
4.1	Środowisko pracy	16
4.2	Wstęp	16
4.2.1	Common Intermediate Language	17
4.2.2	Reflection.Emit	18
4.3	Opis	18
4.3.1	Register	20
4.3.2	Resolve	22
4.3.3	BuildUp	22
4.4	Rozwiązanie	23
4.4.1	Krok pierwszy	24

4.4.2	Rozwiązanie 1 - PartialEmitFunction	25
4.4.3	Rozwiązanie 2 - FullEmitFunction	28
4.4.4	Przykład	35
4.5	Porównanie z istniejącymi implementacjami	40
5	Testy wydajnościowe	41
5.0.1	Register as Singleton	41
5.0.2	Register as Transient	41
5.0.3	Register as TransientSingleton	42
5.0.4	Register as PerThread	42
5.0.5	Register as FactoryMethod	42
5.1	Przypadek testowy A	42
5.1.1	Opis	42
5.1.2	Wyniki Resolve dla Singleton	44
5.1.3	Wyniki Resolve dla Transient	45
5.1.4	Wyniki Resolve dla TransientSingleton	47
5.1.5	Wyniki Resolve dla PerThread	49
5.1.6	Wyniki Resolve dla FactoryMethod	50
5.2	Przypadek testowy B	52
5.2.1	Opis	52
5.2.2	Wyniki Resolve dla Singleton	54
5.2.3	Wyniki Resolve dla Transient	55
5.2.4	Wyniki Resolve dla TransientSingleton	57
5.2.5	Wyniki Resolve dla PerThread	59
5.2.6	Wyniki Resolve dla FactoryMethod	60
5.3	Przypadek testowy C	62
5.3.1	Opis	62
5.3.2	Wyniki Resolve dla Singleton	64
5.3.3	Wyniki Resolve dla Transient	65

5.3.4	Wyniki Resolve dla TransientSingleton	67
5.3.5	Wyniki Resolve dla PerThread	69
5.3.6	Wyniki Resolve dla FactoryMethod	70
5.4	Przypadek testowy D	72
5.4.1	Opis	72
5.4.2	Wyniki Resolve dla Singleton	73
5.4.3	Wyniki Resolve dla Transient	74
5.4.4	Wyniki Resolve dla TransientSingleton	75
5.4.5	Wyniki Resolve dla PerThread	76
5.4.6	Wyniki Resolve dla FactoryMethod	77
6	Podsumowanie	79
6.1	Kontynuacja projektu	79

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 na platformie .NET są **Unity** i **Ninject**. Niestety większość rozwiązań skupia się na zapewnieniu jak największej liczby funkcjonalności, co ma negatywne skutki dla wydajności. O ile przy małej liczbie zależności te czasy nie są zbyt duże, to wraz ze wzrostem zależności wydajność zauważalnie spada.

Celem niniejszej pracy magisterskiej jest stworzenie takiej implementacji tego wzorca, która będzie wydajna dla złożonych grafów zależności, ale również będzie dostarczać niezbędnej funkcjonalności. Do tego celu zostaną wykorzystane mechanizmy z przestrzeni nazw **Reflection.Emit**. W tej pracy zostaną przedstawione dwa rozwiązania tego problemu.

1.2 Układ pracy

Poza wstępem i podsumowaniem praca składa się jeszcze z czterech rozdziałów. Pierwszy z nich opisuje idee stojące za wstrzykiwaniem zależności. W kolejnym znajduje się opis teoretyczny tego, czym jest wstrzykiwanie zależności. Następny rozdział został poświęcony mojej implementacji tego wzorca. Ostatni z rozdziałów skupia się na testach wydajnościowych, w których są porównywane rozwiązania przedstawione w tej pracy, z kilkoma najbardziej popularnym i kilkoma najwydajniejszymi rozwiązaniami.

2 Odwrócenie zależności

2.1 SOLID

Na przestrzeni lat powstało bardzo dużo projektów. Część z nich była łatwiejsza w utrzymaniu, część trudniejsza. Analiza tych projektów pozwoliła zauważyć, że są pewne zasady, które powodują, że projekty rozwija się łatwiej. Te zasady zostały połączone w zbiory zasad. Najbardziej popularnych i powszechnie stosowanym zbiorem zasad jest SOLID [5]:

- SRP (Single responsibility principle) - klasa powinna mieć tylko jedną odpowiedzialność (nigdy nie powinien istnieć więcej niż jeden powód do modyfikacji klasy).
- OCP (Open/closed principle) - klasy powinny być otwarte na rozszerzenia i zamknięte na modyfikacje.
- LSP (Liskov substitution principle) - funkcje które używają klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów.
- ISP (Interface segregation principle) - wiele dedykowanych interfejsów jest lepsze niż jeden ogólny.
- DIP (Dependency inversion principle) - wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych. Zależności między nimi powinny wynikać z abstrakcji.

Niniejsza praca w dużej mierze skupia się na rozwiązaniu dla zasady DIP - Dependency inversion principle.

2.1.1 Przykład dla zasady DIP

Rys. 1 przedstawia przykładową klasę bez zastosowania zasady DIP.


```

public class Foo
{
    public void DoSomeWork()
    {
        //do some work
    }
}

public class Bar
{
    private readonly Foo _foo;

    public Bar()
    {
        _foo = new Foo();
    }

    public void DoSomething()
    {
        _foo.DoSomeWork();
    }
}

```

Rys. 1: Przykładowa klasa bez zasady DIP

Rys. 2 przedstawia tę samą klasę po zastosowaniu zasady DIP.

```

public interface IFoo
{
    void DoSomeWork();
}

public class Foo : IFoo
{
    public void DoSomeWork()
    {
        //do some work
    }
}

public class Bar
{
    private readonly IFoo _foo;

    public Bar(IFoo foo)
    {
        _foo = foo;
    }

    public void DoSomething()
    {
        _foo.DoSomeWork();
    }
}

```

Rys. 2: Przykładowa klasa z zasadą DIP

Łatwo zauważyć, że teraz klasy **Bar** nie interesuje implementacja klasy **Foo**. Spodziewa się jedynie, że otrzyma implementację interfejsu **IFoo**, który dostarcza metody **DoSomething**. Dzięki takiemu podejściu w łatwy sposób można podmienić implementację interfejsu **IFoo** w klasie **Bar**. Zatem tworząc ciało metody **DoSomething** w klasie **Bar** nie trzeba się przejmować implementacją metody **DoSomething**, tylko należy rozważyć jej "kontrakt", zobowiązanie do realizacji założonej funkcjonalności.

Kolejną zaletą takiego podejścia jest to, że teraz w łatwy sposób można przetestować metodę **DoSomething** - można stworzyć przykładową implementację interfejsu **IFoo** lub użyć tzw. obiektu zastępczego (ang. mock).

2.2 Kontenery wstrzykiwania zależności

Aby łatwiej zastosować zasadę DIP można wesprzeć się tzw. kontenerem wstrzykiwania zależności (ang. Dependency Injection Container). Jest to obiekt, który przechowuje mapę, w której abstrakcje (interfejsy, klasy abstrakcyjne) mają przyporządkowane implementacje (klasy implementujące interfejsy lub dziedziczące z klas abstrakcyjnych).

Kontenery dostarczają nam kilku funkcjonalności. Jedną z nich jest oczywiście możliwość zdefiniowania tego jakiej instancje jakiej klasy należy zwrócić w miejsce konkretnego interfejsu. Drugą, również ważną funkcjonalnością, jest tworzenie (na podstawie przechowywanego mapowania) instancji obiektów konkretnej klasy lub implementujących określony interfejs. Niektóre kontenery dostarczają również mechanizmy pozwalające rozszerzyć tworzenie nowych instancji o elementy programowania aspektowego.

2.2.1 Przykład

Założmy, że mamy dwie klasy: **A** i **B**, zdefiniowane jak na Rys. 3.

```
public class A
{
    public A(B b) { }
}

public class B
{
    public B() { }
```

Rys. 3: Definicja przykładowych klas A i B

Rejestracja tych klas w kontenerze wygląda następująco:

```
container.Register<A>();  
container.Register<B>();
```

Obiekty tych klas (przy pomocy kontenera) można uzyskać w następujący sposób:

```
container.Resolve<B>();  
container.Resolve<A>();
```

Stworzenie obiektu klasy **B** przy pomocy operacji **Resolve** jest proste, ponieważ ta klasa nie ma żadnych zależności. Dla obiektu klasy **A** ta operacja jest już bardziej skomplikowana, ponieważ wymaga przejrzenia struktury kodu i odkrycia, że do wygenerowania obiektu klasy **A** wymagany jest obiekt klasy **B**.

Operacja **Resolve** może się jeszcze bardziej skomplikować, gdy zostanie dodana polityka czasu życia. Jeśli klasa **B** zostanie zarejestrowana jako singleton:

```
container.Register<B>(new SharedLifetimePolicy());
```

To każdy nowo tworzony obiekt klasy **A** powinien otrzymać w konstruktorze dokładnie ten sam obiekt klasy **B**.

Operacja **Register** zaczyna się robić praktyczna, gdy dla klasy **B** dodamy abstrakcję:

```
container.Register<IB, B>(new SharedLifetimePolicy());
```

Należy poprawić również kod klas **A** i **B** (Rys. 4).

```
public class A  
{  
    public A(IB b) { }  
}  
  
public interface IB  
{  
}  
  
public class B : IB  
{  
}
```

Rys. 4: Nowa definicja przykładowych klas A i B

Możliwość rejestracji implementacji dla wskazanej abstrakcji jest kluczem do spełnienia postulatów reguły DIP przez kontener. Kontener otrzymuje wtedy pełną wiedzę i odpowiedzialność za koordynowanie rozwiązywania zależności. Rekonfiguracja za-

leżności wymaga tylko i wyłącznie zmiany konfiguracji, którą zna tylko kontener.

Dzięki takiemu podejściu programowanie rozległej aplikacji staje się "projektowaniem jej architektury" - określaniem miękkich zależności między klasami (przez abstrakcje) i komponowaniem ostatecznego kształtu aplikacji (wskazywaniem jak implementowane są abstrakcje) w miejscu konfiguracji kontenera.

Warto dodać, że wiele kontenerów wspiera konfigurację deklaratywną (w postaci plików np. XML) co powoduje, że aplikację można przekonfigurować bez potrzeby jej kompilacji, wyłącznie przez zmianę konfiguracji (mapowania abstrakcji na implementacje) w plikach konfiguracyjnych.

3 Wstrzykiwanie zależności

3.1 Wstęp

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.

3.1.1 Wstrzykiwanie przez konstruktor

Jest to główny i najbardziej popularny sposób wstrzykiwania zależności. Niektóre klasy mają więcej niż jeden konstruktor. Atrybut **DependencyConstructor** przydaje się wtedy do oznaczenia, który z nich ma zostać wybrany przy tworzeniu nowego obiektu. Przykład klasy z atrybutem **DependencyConstructor** przedstawia Rys. 5.

```
public class SampleClass
{
    public SampleClass()
    {
    }

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

    public EmptyClass EmptyClass { get; }
}
```

Rys. 5: Przykładowa klasa z atrybutem DependencyConstructor

3.1.2 Wstrzykiwanie przez metodę

Ten sposób wstrzykiwanie z reguły odbywa się albo poprzez oznaczenie metody (przez którą chcemy wstrzyknąć zależności) odpowiednim atrybutem, albo przez wskazanie odpowiedniej metody podczas rejestracji. Przykład klasy z oznaczeniem metody odpowiednim atrybutem przedstawia Rys. 6.

```

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

    public EmptyClass EmptyClass { get; private set; }
}

```

Rys. 6: Przykładowa klasa z atrybutem DependencyMethod

3.1.3 Wstrzykiwanie przez właściwość

Dla tego sposobu podobnie jak dla wstrzykiwania przez metodę, 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. 7.

```

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

```

Rys. 7: Przykładowa klasa z atrybutem DependencyProperty

3.2 Implementacje przemysłowe

Na rynku jest wiele implementacji wstrzykiwania zależności. Poniżej przedstawiono kilka najbardziej popularnych (według ilości pobrań z NuGet) oraz kilka najszybszych (według rankingu na blogu Daniela Palme [6]).

Dane pochodzą z dnia 21-02-2017. W nawiasie znajduje się wersja implementacji, która została użyta w testach (najnowsza na dany 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).

4 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 niniejsza praca napisana w języku LaTeX. Dodatkowo załączone są wszystkie obrazki zawarte w niniejszej pracy.

4.1 Środowisko pracy

Praca 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

4.2 Wstęp

Na początku chciałbym pokrótce opisać dwie rzeczy, które są istotne dla rozwiązania przedstawionego w niniejszej pracy. Pierwszą z nich jest **Common Intermediate Language**, a drugą przestrzeń nazw **Reflection.Emit**.

4.2.1 Common Intermediate Language

Common Intermediate Language - CIL (w skrócie IL) to język pośredni do którego jest kompilowany kod C#. Język ten pozwala na komunikację między aplikacjami napisanymi na platformie .NET, a systemem operacyjnym.

CIL jest językiem w całości opartym na stosie. Dodatkowo jest w nim możliwość zdefiniowania zmiennych lokalnych, a także wczytania argumentów. Dostępnymi operacjami są: pobranie wartości ze stosu, umieszczenie wartości na stosie, operacje arytmetyczne oraz wywołanie funkcji.

Rys. 8 przedstawia funkcję **Abs** napisaną w języku C#, która zwraca wartość bezwzględną z danej liczby.

```
public int Abs(int x)
{
    if (x >= 0)
        return x;
    else
        return -x;
}
```

Rys. 8: Przykładowa metoda Abs

Taka funkcja w języku CIL wyglądałaby następująco:

```
.method public hidebysig instance int32 Abs(int32 x) cil managed
{
    // wczytaj pierwszy argument funkcji i umieść go na szczycie stosu
    IL_0000: ldarg.1
    // umieść 0 na szczycie stosu
    IL_0001: ldc.i4.0
    // jeśli pierwsza wartość na szycie stosu jest większa od drugiej, idź do operacji IL_0006
    IL_0002: blt.s IL_0006
    // wczytaj pierwszy argument funkcji i umieść go na szczycie stosu
    IL_0004: ldarg.1
    // zwróć wartość ze szczytu stosu
    IL_0005: ret
    // wczytaj pierwszy argument funkcji i umieść go na szczycie stosu
    IL_0006: ldarg.1
    // zmień znak na przeciwny wartości na szczycie stosu
    IL_0007: neg
    // zwróć wartość ze szczytu stosu
    IL_0008: ret
}
```

4.2.2 Reflection.Emit

Przestrzeń nazw **Reflection.Emit** pozwala w kodzie programu, w sposób dynamiczny, na utworzenie listy operacji w języku IL, a następnie zapamiętanie ciągu tych operacji jako delegata. Za każdym razem, gdy delegat zostanie wywołany, wykona się ciąg wcześniej zdefiniowanych operacji IL.

Rys. 9 przedstawia funkcję, która stworzy delegata dla funkcji z Rys. 8.

```
public Func<int, int> CreateDelegateForAbs()
{
    var dm = new DynamicMethod("Emit", typeof(int), new[] { typeof(int) }, true);
    var ilgen = dm.GetILGenerator();

    var lower = ilgen.DefineLabel();
    ilgen.Emit(OpCodes.Ldarg_0);
    ilgen.Emit(OpCodes.Ldc_I4_0);
    ilgen.Emit(OpCodes.Blt_S, lower);
    ilgen.Emit(OpCodes.Ldarg_0);
    ilgen.Emit(OpCodes.Ret);
    ilgen.MarkLabel(lower);
    ilgen.Emit(OpCodes.Ldarg_0);
    ilgen.Emit(OpCodes.Neg);
    ilgen.Emit(OpCodes.Ret);

    return (Func<int, int>)dm.CreateDelegate(typeof(Func<int, int>));
}
```

Rys. 9: Przykład stworzenia delegata dla metody Abs

Przykład użycia tego delegata został przedstawiony na Rys. 10.

```
var abs = CreateDelegateForAbs();
var result = abs.Invoke(-2);
```

Rys. 10: Przykład użycia delegata dla metody Abs

4.3 Opis

Aplikacja składa się z 1 głównego projektu i 8 projektów na potrzeby testów. Rozwiązanie jest stosunkowo złożone. Aby mieć pewność, że działa w pełni dobrze, zostało stworzone ponad 1250 testów jednostkowych. Pokrycie kodu testami wynosi ponad 97%.

W wykonanej implementacji został stworzony interfejs **IContainer**, który składa się z interfejsów zawierających niezbędne operacje, jakie powinny się znaleźć w każdym kontenerze (Rys. 11).

```
public interface IContainer : IContainerRegister, IContainerResolve, IContainerBuildUp
{
}
```

Rys. 11: Interfejs IContainer

Pierwszy z tych interfejsów to **IContainerRegister** (Rys. 12). Zawiera on metody służące do zarejestrowania typów w kontenerze.

```
public interface IContainerRegister
{
    IContainerMember RegisterType<T>()
        where T : class;

    IContainerMember RegisterType(Type type);

    IContainerMember RegisterType<TFrom, TTo>()
        where TTo : TFrom;

    IContainerMember RegisterType(Type typeFrom, Type typeTo);

    IContainerMember RegisterType<T>(Func<IContainerResolve, T> objectFactory)
        where T : class;

    IContainerMember RegisterType(Type type, Func<IContainerResolve, object> objectFactory);

    IContainerMember RegisterInstance<T>(T instance);
}
```

Rys. 12: Interfejs IContainerRegister

Drugi z nich to **IContainerResolve** (Rys. 13). Składa się on z metod odpowiedzialnych za tworzenie i zwracanie obiektów wcześniej zarejestrowanych typów.

```

public interface IContainerResolve
{
    T Resolve<T>();

    T Resolve<T>(ResolveKind resolveKind);

    object Resolve(Type type);

    object Resolve(Type type, ResolveKind resolveKind);
}

```

Rys. 13: Interfejs IContainerResolve

Ostatni z tych interfejsów to **IContainerBuildUp** (Rys. 14). Jego metody służą do uzupełnienia istniejącej instancji obiektu z wykorzystaniem wstrzykiwania zależności przez metodę i właściwość. Są to operacje opcjonalne i nie każde przemysłowe rozwiązanie je zawiera.

```

public interface IContainerBuildUp
{
    void BuildUp<T>(T instance);

    void BuildUp<T>(T instance, ResolveKind resolveKind);
}

```

Rys. 14: Interfejs IContainerBuildUp

Poniżej znajduje się szczegółowy opis metod z każdego z powyższych interfejsów.

4.3.1 Register

W pierwszej i drugiej metodzie interfejsu **IContainerRegister** można zarejestrować zwykłe klasy. W trzeciej i czwartej interfejsy oraz klasy, które implementują dany interfejs lub klasy i klasy po nich dziedziczące. W piątej i szóstej metodzie można zarejestrować klasę jako fabrykę obiektów - funkcję, która ma zwrócić pożądany obiekt. W siódmej (ostatniej) natomiast można zarejestrować konkretną instancję danego typu.

W rozwiązaniu przedstawionym w tej pracy każdy typ może być zarejestrowany tylko raz - ponowna rejestracja tego samego typu nadpisuje istniejącą rejestrację.

Każda z tych siedmiu metod rejestracji zwraca interfejs **IContainerMember** (Rys. 15), który umożliwia zarejestrowanie danego typu z określonym menadżerem czasu życia (czyli implementacją interfejsu **IObjectLifetimeManager** - Rys. 16). Celem takiego podejścia jest umożliwienie określenia polityki czasu życia dla nowo tworzonych instancji, ponieważ dla różnych przypadków biznesowych może istnieć potrzeba, aby obiekt danego typu miał konkretny czas życia.

```

public interface IContainerMember
{
    void AsSingleton();

    void AsTransient();

    void AsPerThread();

    void AsPerHttpContext();

    void AsCustomObjectLifetimeManager(IObjectLifetimeManager objectLifetimeManager);
}

```

Rys. 15: Interfejs IContainerMember

Pierwsze cztery metody interfejsu **IContainerMember**, to wbudowane implementacje interfejsu **IObjectLifetimeManager**. Piąta metoda dostarcza możliwość podania przez użytkownika jego własnej implementacji tego interfejsu. Domyślną polityką czasu życia nowo tworzonych instancji jest polityka **Transient**, czyli zwracanie nowej instancji obiektu za każdym razem, kiedy kontener rozwiązuje zależność.

Wyjaśnienie nazw polityk 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.

W interfejsie **IObjectLifetimeManager** właściwość **ObjectFactory** służy do ustawienia funkcji, która zwraca obiekt. Metoda **GetInstance** służy do pobrania obiektu.

W zależności od implementacji tego interfejsu, 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).

```

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

    object GetInstance();
}

```

Rys. 16: Interfejs IObjectLifetimeManager

4.3.2 Resolve

Metody interfejsu **IContainerResolve** są głównymi operacjami. **Register** można nazywać sercem kontenera, a **Resolve** mózgiem. Każda z metod tego interfejsu odpowiada za stworzenie i zwrócenie obiektu odpowiedniego typu.

W niniejszej pracy zaproponowano dwa rozwiązania - **PartialEmitFunction** i **FullEmitFunction** (różnice między nimi zostaną opisane w dalszej części pracy). W związku z tym metody rejestracji jako opcjonalny parametr przyjmują wartość enumeracji określającej wybraną politykę - **ResolveKind** (dzięki temu w przyszłości może być ona w łatwy sposób rozszerzona o kolejne rozwiązania). W interfejsie znajdują się również metody, które nie przyjmują tego parametru. Zostały one dodane na wypadek, gdy ktoś będzie zawsze korzystał tylko z jednego z rozwiązań (może ustawić je jako domyślne np. w konstruktorze).

4.3.3 BuildUp

Interfejs **IContainerBuildUp** to zwyczajowy, użyteczny dodatek do kontenera. Gdy mamy utworzony obiekt, ale nie jest on w pełni uzupełniony, to można go rozbudować (używając metody z odpowiednią wartością enumeracji **ResolveKind** lub tej z wartością domyślną). Z metodami tego interfejsu są powiązane bezpośrednio dwa pojęcia - wstrzykiwanie przez metodę i wstrzykiwanie przez właściwość. Do tego celu w niniejszym rozwiązaniu utworzono dwa atrybuty:

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

Podczas operacji **BuildUp** wywoływane są wszystkie metody i uzupełniane są wszystkie właściwości, które mają powyżej wymienione atrybuty. Opisywana operacja jest również wykonywana podczas operacji **Resolve**.

Warto odnotować, że ze względu na szczegóły implementacyjne, tylko jedno z niniejszych 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 jej dodania.

W aplikacji istnieje również atrybut **DependencyConstructor**. Można go użyć podczas definicji konstruktora danej klasy. Obiekt każdej klasy jest tworzony przy użyciu konstruktora. Klasa może mieć kilka konstruktorów. W niniejszym rozwiązaniu utworzono 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 wybierz odpowiedni 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.

4.4 Rozwiązanie

Utworzenie instancji 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 o tym, jakich typów obiekty są potrzebne do stworzenia danego obiektu.
- Stworzenie tych pomocniczych obiektów.
- Stworzenie docelowego obiektu.

W przypadku rozbudowanych grafów zależności zdarza się, że niektóre typy się powtarzają. Zatem pewne informacje można 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.

W niniejszej implementacji utworzono dwie strategie, które realizują wyżej opisane założenia. Pierwszy krok jest taki sam dla obu rozwiązań. Jest to uzyskanie informacji jakich typów obiekty są potrzebne do stworzenia danego obiektu. Natomiast kolejne kroki już się różnią. W pierwszym rozwiązaniu (które nazwano **PartialEmitFunction**) całym proces tworzenia nowego obiektu został rozbity na mniejsze części (docelowy obiekt jest tworzony po kawałku). Każda taka osobna część jest zapisywana w pamięci podręcznej. W drugim rozwiązaniu w pamięci podręcznej jest zapisana tylko jedna operacja. Zawiera ona listę wszystkich kroków, które są niezbędne do stworzenia docelowego obiektu. Finalnie docelowy obiekt jest tworzony przy pomocy jednej operacji (kroki drugi i trzeci są połączone). To rozwiązanie nazwano **FullEmitFunction**.

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

4.4.1 Krok pierwszy

Na początku algorytm znajduje odpowiedni konstruktor, przy pomocy którego ma zostać stworzony nowy obiekt. Jeśli robi to po raz pierwszy dla danego typu, to informacje o tym konstruktorze zapisuje w pamięci podręcznej. Do tego celu zostaje użyta struktura danych słownik, gdzie kluczem jest typ obiektu, a wartością obiekt klasy **ContainerMember** (Rys. 17), w którym przechowujemy wszystkie zapamiętane informacje dla danego typu. Następnym krokiem jest rekurencyjne wywołanie opisywanej operacji dla wszystkich typów, których obiekty są niezbędne do stworzenia obiektu docelowego typu.

W wyżej opisanej operacji jest kilka wyjątków - są nimi typy zarejestrowane jako **Instance** albo **FactoryObject**. Dla pierwszego przypadku nie istnieje potrzeba uzyskiwania informacji o tym jak stworzyć obiekt takiego typu. Taki obiekt jest już stworzony i zostaje on po prostu wykorzystany. Dla drugiego przypadku obiekt tworzony jest przy użyciu wcześniej zdefiniowanej przez użytkownika funkcji, której wywołanie spowoduje stworzenie obiektu oczekiwanego typu.

```
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. 17: Klasa ContainerMember

Klasa **ContainerMember** przechowuje:

- Informacje o konstruktorze, 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.
- Czy 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 menedżerze cyklu życia.

4.4.2 Rozwiązanie 1 - PartialEmitFunction

Cały algorytm jest zawarty w metodzie **Resolve** (Rys. 18). Sama ta metoda jest dość krótka, ale wywołuje ona kolejne (dłuższe) metody.

Docelowy obiekt jest tworzony przy pomocy funkcji. Na początku następuje sprawdzenie, czy już wcześniej została utworzona taka funkcja. Jeśli tak, będzie ona zapisana w klasie **ContainerMember** we właściwości **ObjectLifetimeManager**. Wywołanie funkcji kończy działanie algorytmu. Jeśli funkcja nie została wcześniej utworzona, to następuje jej utworzenie.

Wyżej wspomniana funkcja nie przyjmuje żadnych parametrów, a jej typem wynikowym jest obiekt. Ciało tej funkcji jest bardzo proste - ma ona wykonać metodę i zwrócić jej rezultat, którym jest 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. 18: Metoda Resolve klasy PartialEmitFunction

Metoda **GetObject** (Rys. 19) jest trochę bardziej rozbudowana. Na początku są pobierane informacje dotyczące parametrów konstruktora danego typu. Następnie dla każdego z tych parametrów jest tworzony obiekt przy pomocy funkcji **Resolve** (opisanej wyżej). W tym przypadku wykorzystano rekurencję. Gdy obiekty dla wszystkich parametrów konstruktora zostały utworzone, ich lista jest przekazywana do metody **CreateInstanceFunction**, która zwraca instancję obiektu oczekiwanego typu. Na koniec jest wywoływana funkcja **afterObjectCreate** i docelowy obiekt zostaje zwrócony.

```
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. 19: Metoda **GetObject** klasy **PartialEmitFunction**

Na początku metody **CreateInstanceFunction** (Rys. 20) występuje sprawdzenie, czy funkcja potrafiąca zwrócić obiekt docelowego typu została utworzona.

Jeśli wynik jest pozytywny, to przy pomocy funkcji jest tworzony i zwracany obiekt. Funkcja jako argument przyjmuje listę obiektów w kolejności zgodnej z listą parametrów konstruktora.

Jeśli wynik jest negatywny, to przy pomocy metody **CreateObjectFunction** wymagana funkcja jest tworzona. W kolejnym kroku jest ona zapisywana w pamięci podręcznej (również w strukturze słownik, której kluczem jest typ, a wartością jest funkcja). Dalej dzieje się to samo, co w sytuacji, gdy funkcja jest już utworzona.

```

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. 20: Metoda CreateInstanceFunction klasy PartialEmitFunction

CreateObjectFunction (Rys. 21) jest najbardziej zaawansowaną metodą w tym algorytmie. To w niej wykorzystano metody z przestrzeni nazw **Reflection.Emit**.

Na początku są pobierane informacje o konstruktorze. Następnie zostaje utworzony obiekt **DynamicMethod** i z niego jest pobierany **ILGenerator**. W nim będzie przechowywana lista kroków niezbędnych do utworzenia docelowego obiektu.

Dla każdego z parametrów konstruktora są wykonane 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. Pobierz typ parametru.
5. Dodaj do listy kroków operację, która zrzutuje obiekt ze szczytu stosu na odpowiedni typ.

Po wykonaniu kroków z listy na stosie będą znajdowały się wszystkie obiekty, które są wymagane przez konstruktor do utworzenia danego typu. Do listy kroków należy dodać jeszcze dwie operacje. Pierwsza z nich ma za zadanie stworzyć obiekt przy pomocy danego konstruktora i umieścić go na szczycie stosu. Druga ma na celu zwrócić obiekt ze szczytu stosu.

Wszystkie kroki niezbędne do stworzenia nowego elementu są zapisane w zmiennej typu **DynamicMethod**. Na końcu z tej zmiennej należy utworzyć, a następnie zwrócić delegata, który jako parametr będzie przyjmował tablicę obiektów (lista obiektów zgodna z parametrami konstruktora) i zwracał obiekt (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++)
    {
        // Step 1
        ilgen.Emit(OpCodes.Ldarg_0);
        // Step 2
        EmitHelper.EmitIntOntoStack(ilgen, i);
        // Step 3
        ilgen.Emit(OpCodes.Ldelem_Ref);
        // Step 4
        var paramType = parameters[i].ParameterType;
        // Step 5
        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. 21: Metoda CreateObjectFunction klasy PartialEmitFunction

4.4.3 Rozwiązanie 2 - FullEmitFunction

Podobnie jak dla **PartialEmitFunction** cały algorytm zawarty jest w metodzie **Resolve** (Rys. 22). Wygląda ona identycznie jak w rozwiązaniu 1 - docelowy obiekt jest tworzony przy pomocy funkcji, która woła w sobie metodę **GetObject**. Jeśli funkcja nie jest zapamiętana w pamięci, to zostaje ona stworzona i zapisana. Na końcu jest ona wywoływana.

```

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. 22: Metoda Resolve klasy FullEmitFunction

Metoda **GetObject** (Rys. 23) wygląda trochę inaczej. Nie pobiera ona żadnych dodatkowych informacji, tylko od razu wywołuje metodę **CreateInstanceFunction**, która zwraca instancję obiektu oczekiwanego typu. Na koniec również jest wywoływana funkcja **afterObjectCreate** i zostaje zwrócony docelowy obiekt.

```

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

    return obj;
}

```

Rys. 23: Metoda GetObject klasy FullEmitFunction

CreateInstanceFunction (Rys. 24) ma jeden dodatkowy krok względem rozwiązania 1. Na początku również następuje sprawdzenie czy została wcześniej utworzona funkcja, która umie zwrócić obiekt docelowego typu. Jeśli nie, to przy pomocy metody **CreateObjectFunction** taka funkcja zostaje stworzona, a następnie jest zapisywana w pamięci podręcznej (również w strukturze słownika, którego kluczem jest typ, a wartością jest typ pomocniczy **FullEmitFunctionResult**, który ma tylko jedną właściwość - **Result** typu funkcja). Następnym krokiem jest walidacja zapisanych danych dla typów przy pomocy metody **ValidateTypesCache**. Na końcu funkcji obiekt jest tworzony, a następnie zwracany. 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 hash danego typu, a wartością typ).

Najpierw zostanie opisana metoda **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. 24: Metoda CreateInstanceFunction klasy FullEmitFunction

W pierwszym kroku metoda **ValidateTypesCache** sprawdza (Rys. 25), 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 z zarejestrowanych typów zostaje utworzony słownik, w którym kluczem jest hash typu, a wartością typ. Potem następuje aktualizacja liczby 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. 25: Metoda ValidateTypesCache klasy FullEmitFunction

Metoda **CreateObjectFunction** (Rys. 26) jest dużo bardziej rozbudowana niż w rozwiązaniu 1. W niej również są wykorzystane metody z przestrzeni nazw **Reflection.Emit**.

Na początku jest tworzony obiekt typu **DynamicMethod** i z niego jest pobierany **ILGenerator**. W nim będzie przechowana lista kroków niezbędnych do utworzenia docelowego obiektu.

Dla każdego z parametrów konstruktora jest wywoływana metoda **CreateObjectFunctionPrivate**, która uzupełnia listę kroków o tworzenie pośrednich obiektów. To w tym miejscu jest główna różnica między oboma rozwiązaniami - w pierwszym rozwiązaniu

w liście kroków nie istniała potrzeba utworzenia pośrednich obiektów, ponieważ były one przekazywane jako parametr. W tym rozwiązaniu lista kroków zawiera również niezbędne kroki do utworzenia wszystkich obiektów pośrednich (obiektów wymaganych przez konstruktor).

Dalsze operacje są takie same jak w rozwiązaniu 1. Do listy kroków jest dodana operację, która stworzy obiekty przy pomocy danego konstruktora i umieści go na szczycie stosu. Następnie jest dodana operację, który zwróci nam obiekt ze szczytu stosu.

Na końcu ze zmiennej typu **DynamicMethod** tworzony jest delegata, a następnie jest on zwracany. Delegat będzie przyjmował dwa parametry - oba typu **Dictionary** (słownik). Jeden przechowujący informacje na temat typu, 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. 26: Metoda CreateObjectFunction klasy FullEmitFunction

W metodzie **CreateObjectFunctionPrivate** (Rys. 27) są trzy przypadki. W pierwszych dwóch przypadkach pomijane są 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. 27: Metoda CreateObjectFunctionPrivate klasy FullEmitFunction

W pierwszym przypadku typ został zarejestrowany jako **Transient** - Rys. 28. Dla każdego z parametrów konstruktora jest wywoływana rekurencyjnie metoda **CreateObjectFunctionPrivate**, a na koniec do listy kroków jest dodawana operacja, 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. 28: Metoda IsTransient klasy FullEmitFunction

W drugim przypadku typ musi być singletonem (został on zarejestrowany jako **Singleton**, **PerThread** lub **PerHttpContext**) - Rys. 29. Na początku następuje sprawdzenie czy już wcześniej algorytm natrafił na ten typ. Jeśli tak, to z pamięci podręcznej jest pobierana zmienna lokalna dla danego typu, a następnie do listy kroków jest dodawana operacja, która doda na szczyt stosu obiekt z tej zmiennej. Jeśli nie, to najpierw zostaje stworzona nowa zmienna lokalna dla danego typu, a potem jest ona zapisywana w pamięci podręcznej (wykorzystany jest do tego słownik, gdzie kluczem jest typ, a wartością obiekt typu **LocalBuilder**). Następnie jest wywoływana metoda **AddObjectCreatedByObjectLifetimeManager**. Na końcu są dodawane 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. Ma to na celu zapamiętanie danego obiektu i pozostawienie go na szczycie stosu. Dzięki takiemu zabiegowi, gdy drugi raz dany obiekt będzie potrzebny (obiekt jest singletonem, więc za każdym razem będziemy chcieli mieć ten sam obiekt - w kontekście tworzenia danego typu, czyli jednej operacji **Resolve**), to z zapisanej zmiennej zostanie on dodany na szczyt stosu.

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

Rys. 29: Metoda IsSingleton klasy FullEmitFunction

Metoda **AddObjectCreatedByObjectLifetimeManager** (Rys. 30), 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 hash 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 przechowujący informacje 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, który został utworzony przy pomocy odpowiedniego menedżera 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. 30: Metoda AddObjectCreatedByObjectLifetimeManager klasy FullEmitFunction

Trzeci przypadek zachodzi, gdy typ został zarejestrowany jako **Instance**, **FactoryObject** lub przy pomocy własnego menedżera czasu życia (własnej implementacji interfejsu **IObjectLifetimeManager**). W tej sytuacji zostaje wywołana tylko metoda **AddObjectCreatedByObjectLifetimeManager**.

4.4.4 Przykład

Mamy klasę **A**, która w konstruktorze przyjmuje obiekty klas **B** i **C**. Klasa **B** w konstruktorze przyjmuje obiekt klasy **C**, a klasa **C** ma konstruktor bezparametrowy. Tę sytuację przedstawia Rys. 31.

```
public class A
{
    public A(B b, C c)
    {
        B = b;
        C = c;
    }

    public B B { get; }
    public C C { get; }
}

public class B
{
    public B(C c)
    {
        C = c;
    }

    public C C { get; }
}

public class C
{
    public C()
    {
    }
}
```

Rys. 31: Przykładowe klasy A, B i C

Dla rozwiązania **PartialEmitFunction** algorytm zakłada, że zostały wcześniej stworzone obiekty wymagane w konstruktorze klasy **A** (w tym przypadku obiekty **B** i **C**). Delegat, w którym będzie zapisana lista niezbędnych operacji do stworzenia obiektu klasy **A** ma jeden parametr. Jest nim tablica obiektów. Dla tego przypadku wygląda ona następująco:

[B, C]

Natomiast lista operacji IL prezentuje się w sposób następujący:

```
// na stosie zostanie umieszczona referencja do pierwszego argumentu aktualnej funkcji,
// czyli do tablicy zawierającej B i C.
ldarg_0
```

```
// na stos zostanie dodana liczba 0.
```

```
ldc_i4_0
```

```
// ze stosu zostanie ściągnięta liczba oraz tablica, a na jego szczycie zostanie umiesz-  
czony obiekt pod indeksem 0 z tablicy - w tym przypadku obiekt B.
```

```
ldelem_ref
```

```
// obiekt ze szczytu stosu jest rzutowany na typ B (wcześniej był typu object).
```

```
castclass typeof(B)
```

```
// na stosie zostanie umieszczona referencja do pierwszego argumentu aktualnej funkcji,  
czyli do tablicy zawierającej B i C.
```

```
ldarg_0
```

```
// na stos zostanie dodana liczba 1.
```

```
ldc_i4_1
```

```
// ze stosu zostanie ściągnięta liczba oraz tablica, a na jego szczycie zostanie umiesz-  
czony obiekt pod indeksem 1 z tablicy - w tym przypadku obiekt C.
```

```
ldelem_ref
```

```
// obiekt ze szczytu stosu jest rzutowany na typ C.
```

```
castclass typeof(C)
```

```
// ze stosu zostają zdjęte wszystkie obiekty wymagane przez konstruktor klasy A (w  
tym przypadku obiekty B i C - muszą być one w odpowiedniej kolejności na stosie), a  
na jego szczycie jest umieszczany nowo utworzony obiekt A.
```

```
newobj .ctor A
```

```
// obiekt ze szczytu stosu (obiekt A) zostaje zwrócony.
```

```
ret
```

Obiekty **B** i **C** są tworzone na takiej samej zasadzie jak obiekt klasy **A**. Zatem w tym przypadku będziemy mieli 3 listy z operacjami IL - jedna dla obiektu **C**, jedna dla obiektu **B** i jedna dla obiektu **A**.

Niezależnie od tego jak klasy **A**, **B** i **C** zostały zarejestrowane, zawsze uzyskamy dokładnie te same 3 listy operacji.

Dla rozwiązania **FullEmitFunction** jest to trochę bardziej rozbudowane. Delegat z listą operacji IL ma dwa parametry. Pierwszym z nich jest słownik zawierający typ obiektu i ContainerMember dla danego obiektu, a drugim słownik zawierający hash danego obiektu ContainerMember i typ obiektu. Dla opisywanego przypadku argumenty tej funkcji wyglądają następująco:

```
{ {typeof(A), ContainerMember}, {typeof(B), ContainerMember},  
  {typeof(C), ContainerMember} }  
{ {1, typeof(A)}, {2, typeof(B)}, {3, typeof(C)} }
```

Natomiast lista operacji IL zależy od tego jak klasy zostały zarejestrowane. Jeśli wszystkie klasy zostały zarejestrowane jako **Transient**, to lista operacji jest krótka:

```
// na stosie zostanie umieszczony nowo utworzony obiekt C.  
newobj .ctor C  
  
// ze stosu zostanie zdjęty obiekt C i w jego miejsce pojawi się nowo utworzony obiekt B.  
newobj .ctor B  
  
// na szczycie stosu zostanie umieszczony nowo utworzony obiekt C.  
newobj .ctor C  
  
// ze szczytu stosu zostają zdjęte obiekty B i C, a w ich miejsce pojawi się nowo utworzony obiekt A.  
newobj .ctor A  
  
// obiekt ze szczytu stosu (obiekt A) zostaje zwrócony.  
ret
```

Jeśli wszystkie klasy zostały zarejestrowane jako **Singleton**, **PerThread** lub **PerHttpContext**, to lista operacji prezentuje się w sposób następujący:

```
// na stosie zostanie umieszczona referencja do drugiego argumentu aktualnej funkcji,  
czyli do słownika zawierającego hash danego obiektu ContainerMember i typ obiektu.  
ldarg_1  
  
// na stos zostanie dodana liczba równa hashowi obiektu ContainerMember dla klasy B (w tym wypadku 2).  
ldc_i4 2  
  
// ze stosu zostają pobrane dwa elementy (słownik oraz liczba), a następnie zostaje  
wywołana metoda GetItem na słowniku, której parametrem jest liczba. Wynik tej ope-  
racji (typ B) zostanie umieszczony na szczycie stosu.
```

```

call "get_Item"

// obiekt ze szczytu stosu zostaje zapisany w zmiennej variable_Type
stloc variable_Type

// na stosie zostanie umieszczona referencja do drugiego argumentu aktualnej funkcji,
czyli do słownika zawierającego typ obiektu i ContainerMember dla danego obiektu.
ldarg_0

// obiekt ze zmiennej variable_Type zostaje umieszczony na szczycie stosu
ldloc variable_Type

// ze stosu zostają pobrane dwa elementy (słownik oraz typ), a następnie zostaje wy-
wołana metoda GetItem na słowniku, której parametrem jest typ. Wynik tej operacji
(obiekt ContainerMember dla typu B) zostanie umieszczony na szczycie stosu.
call "get_Item"

// ze stosu zostaje pobrany element i jest na nim wywołana metoda GetObjectLife-
timeManager. Wynik tej operacji (odpowiedni obiekt ObjectLifetimeManager) zostanie
umieszczony na szczycie stosu.
call "getObjectLifetimeManager"

// ze stosu zostaje pobrany element i jest na nim wywołana metoda GetInstance. Wy-
nik tej operacji (obiekt typu B) zostanie umieszczony na szczycie stosu.
call "GetInstance"

// obiekt ze szczytu stosu jest rzutowany na typ B.
castclass

// obiekt ze szczytu stosu zostaje zapisany w zmiennej variable_B
stloc variable_B

// obiekt ze zmiennej variable_B zostaje umieszczony na szczycie stosu
ldloc variable_B

ldarg_1

// na stos zostanie dodana liczba równa hashowi obiektu ContainerMember dla klasy
C (w tym wypadku 3).
ldc_i4 3

// ze stosu zostają pobrane dwa elementy (słownik oraz liczba), a następnie zostaje
wywołana metoda GetItem na słowniku, której parametrem jest liczba. Wynik tej ope-
racji (typ C) zostanie umieszczony na szczycie stosu.
call "get_Item"

```

```

// obiekt ze szczytu stosu zostaje zapisany w zmiennej variable_Type
stdloc variable_Type

ldarg_0

// obiekt ze zmiennej variable_Type zostaje umieszczony na szczycie stosu
ldloc variable_Type

// ze stosu zostają pobrane dwa elementy (słownik oraz typ), a następnie zostaje wy-
wołana metoda GetItem na słowniku, której parametrem jest typ. Wynik tej operacji
(obiekt ContainerMember dla typu C) zostanie umieszczony na szczycie stosu.
call "get_Item"

// ze stosu zostaje pobrany element i jest na nim wywołana metoda GetObjectLife-
timeManager. Wynik tej operacji (odpowiedni obiekt ObjectLifetimeManager) zostanie
umieszczony na szczycie stosu.
call "get_ObjectLifetimeManager"

// ze stosu zostaje pobrany element i jest na nim wywołana metoda GetInstance. Wy-
nik tej operacji (obiekt typu C) zostanie umieszczony na szczycie stosu.
call "GetInstance"

// obiekt ze szczytu stosu jest rzutowany na typ C.
castclass

// obiekt ze szczytu stosu zostaje zapisany w zmiennej variable_C
stloc variable_C

// obiekt ze zmiennej variable_C zostaje umieszczony na szczycie stosu
ldloc variable_C

// ze szczytu stosu zostają zdjęte obiekty B i C, a w ich miejsce pojawi się nowo utwo-
rzony obiekt A
newobj .ctor A

// obiekt ze szczytu stosu (obiekt A) zostaje zwrócony
ret

```

Jeśli typ został zarejestrowany jako **Instance**, **FactoryObject** albo z własną implementacją interfejsu **IObjectLifetimeManager**, to lista kroków wygląda podobnie jak dla **Singleton**, z tą różnicą, że nie ma kroków z zapisywaniem obiektu do zmiennej lokalnej (w tym przypadku brak kroków 11, 12, 23 i 24).

4.5 Porównanie z istniejącymi implementacjami

Rozwiązanie przedstawione w tej pracy zostało zainspirowane artykułem przedstawiającym rozwiązanie **Hiro** [7].

Idea **Hiro** było najpierw zbudowanie mapy wszystkich zależności, a następnie zapamiętaniu tej mapy i wykorzystaniu za każdym razem, gdy obiekt danego typu będzie tworzony. W tym rozwiązaniu nie użyto refleksji, a mapa zależności jest zapisana w dodatkowym assembly tworzonym dynamicznie.

Niestety to rozwiązanie nie wyszło z fazy proof-of-concept, a aktualna wersja dostępna w repozytorium autora nie działała i od kilku lat nie jest aktualizowana, stąd brak porównania z nim.

Rozwiązaniem najbardziej podobnym do rozwiązania przedstawionego w tej pracy wydaje się rozwiązanie **DryLoc**. To rozwiązanie również wykorzystuje **Reflection.Emit**, jednakże działa ono trochę inaczej. W tym rozwiązaniu najpierw tworzone jest drzewo (**Expression**) zależności dla każdego typu, a dopiero później na jego podstawie tworzony jest delegat z wykorzystaniem **Reflection.Emit**. W związku z tym stworzenie pierwszego obiektu zajmuje dużo więcej czasu niż w przypadku **NiquoCFull**, ale potem stworzenie każdego kolejnego obiektu zajmuje tego czasu trochę mniej. Niestety ten narzut dla pierwszego wywołania sprawia, że nawet przy tworzeniu wiele razy tego samego obiektu **DryLoc** osiąga większe czasy niż **NiquoCFull**. Dodatkowo dla rejestracji typu **PerThread**, to rozwiązanie nie tworzy delegata z wykorzystaniem **Reflection.Emit**. Jest to powodem dużych wyników dla testów z wykorzystaniem tego typu rejestracji.

Jeśli chodzi o najwolniejsze rozwiązania, to one prawdopodobnie nie optymalizują kompozycji grafu zależności, tylko przechodzą go za każdym razem. Takie przechodzenie nie wydaje się konieczne, ponieważ tak długo jak kod nie jest rekompilowany, to struktura zależności nie ulega zmianie.

5 Testy wydajnościowe

Do przeprowadzania testów wydajnościowych przygotowano osobną aplikację, w której zostały utworzone 4 przypadki testowe:

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

Dla każdego z przypadków sprawdzany jest czas wykonania operacji **Resolve** dla różnych rodzajów rejestracji. 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. Wyniki dla operacji **Register** zostały pominięte, ponieważ są one prawie zawsze mniejsze niż 1 ms. 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.

Rozwiązania zaprezentowane w tej pracy to **NiquIoCPartial** i **NiquIoCFull**. Wyniki w tabelach dla obu tych rozwiązań zostały pogrubione.

5.0.1 Register as Singleton

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

5.0.2 Register as Transient

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

5.0.3 Register as TransientSingleton

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

5.0.4 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.

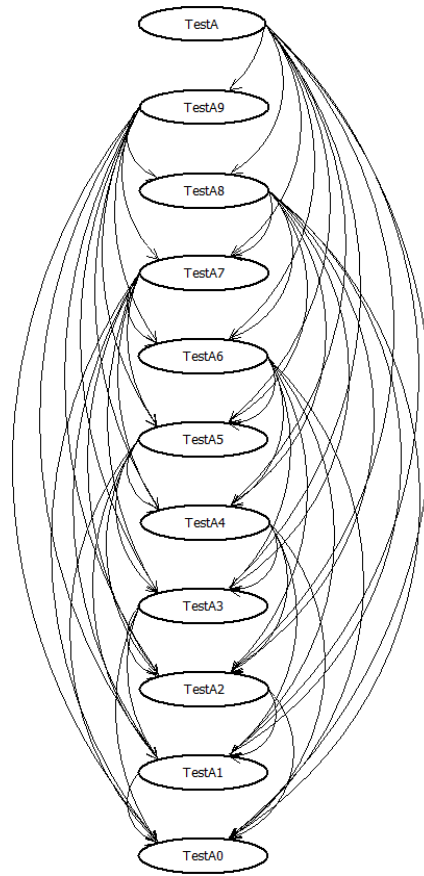
5.0.5 Register as FactoryMethod

Każdy typ jest zarejestrowany jako **FactoryMethod**, czyli jako funkcja, która zwraca obiekt danego typu. Ciało tej funkcji wygląda tak, że oczekiwany typ jest tworzony z użyciem konstruktora (za pomocą **new**), a wszystkie parametry wymagane przez ten konstruktor są tworzone przy pomocy operacji **Resolve**.

5.1 Przypadek testowy A

5.1.1 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, ile wynosi liczba w jego nazwie (czyli obiekt typu **TestA0** ma konstruktor 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 przyjmują obiekty typów z niższymi liczbami w nazwie (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. 32.



Rys. 32: Graf zależności dla testu A.

Łatwo z niego wywnioskować, że tworząc obiekty poszczególnych typów liczba tworzonych obiektów rośnie wykładniczo:

- **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 główny obiekt typu **TestA**, zostaje utworzony: 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.

5.1.2 Wyniki Resolve dla Singleton

Liczba	1		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	1	1	1
DryIoc	2	2	2
Grace	2	2	2
LightInject	2	2	2
NiquIoCFull	2	2	2
SimpleInjector	2	2	2
Ninject	3	3	3
Unity	7	8	7
StructureMap	9	10	9

Tab. 1: Wyniki testów dla 1 powtórzenia dla operacji Singleton dla Przypadku testowego A

Czasy dla wszystkich rozwiązań są nieduże. Jest to spowodowane tym, że gdy wszystkie klasy są zarejestrowane jako **Singleton**, to jest tworzonych niewiele nowych obiektów.

Należy jednak wspomnieć, że najslabsze wyniki osiągnęły **Unity** i **StructureMap**, których czasy są zauważalnie większe niż pozostałych rozwiązań.

Liczba	1000		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	1	1	1
DryLoc	2	2	2
LightInject	2	2	2
NiquIoCFull	2	2	2
SimpleInjector	2	2	2
Grace	2	3	2
Ninject	6	7	6
Unity	8	8	8
StructureMap	10	11	10

Tab. 2: Wyniki testów dla 1000 powtórzeń dla operacji Singleton dla Przypadku testowego A

Gdy klasa jest zarejestrowana jako **Singleton**, to wraz ze wzrostem liczby tworzonych obiektów, nie powinien wzrastać czas ich tworzenia. We wszystkich rozwiązaniach, za wyjątkiem **Ninject**, czasy dla 1000 operacji, są zbliżone do czasów dla 1 operacji. **Ninject** natomiast ma czasy około dwa razy większe.

5.1.3 Wyniki Resolve dla Transient

Liczba	1		
	min	max	avg
Autofac	0	0	0
NiquIoCPartial	1	1	1
Windsor	1	2	1
NiquIoCFull	8	9	8
Unity	8	9	8
LightInject	10	10	10
StructureMap	10	10	10
Ninject	10	12	11
SimpleInjector	13	14	13
DryLoc	14	14	14
Grace	15	16	15

Tab. 3: Wyniki testów dla 1 powtórzenia dla operacji Transient dla Przypadku testowego A

W tym wypadku najlepsze wyniki osiągnęły **Autofac**, **NiquIoCPartial** i **Windsor**, które uzyskały zbliżone rezultaty. Pozostałe rozwiązania uzyskały dużo słabsze, o kilka razy większe czasy.

Liczba	10		
	min	max	avg
NiquIoCPartial	3	3	3
Autofac	6	7	6
NiquIoCFull	8	9	8
LightInject	10	10	10
SimpleInjector	13	14	14
StructureMap	13	14	14
DryIoc	14	15	15
Grace	15	16	16
Unity	16	17	16
Windsor	16	21	16
Ninject	88	95	90

Tab. 4: Wyniki testów dla 10 powtórzeń dla operacji Transient dla Przypadku testowego A

Najmniejszy czas uzyskał **NiquIoCPartial**, ale jest on niestety kilkukrotnie większy niż dla 1 powtórzenia. Co więcej, wszystkie najpopularniejsze rozwiązania zanotowały spory wzrost czasów. Z drugiej strony wszystkie najszybsze rozwiązania oraz **NiquIoCFull** zanotowały podobne czasy co dla 1 powtórzenia.

Liczba	100		
	min	max	avg
NiquIoCFull	9	10	9
LightInject	11	12	11
SimpleInjector	15	15	15
DryIoc	16	16	16
Grace	18	19	18
NiquIoCPartial	19	22	19
StructureMap	53	55	54
Autofac	58	63	59
Unity	88	91	88
Windsor	152	192	155
Ninject	864	1035	882

Tab. 5: Wyniki testów dla 100 powtórzeń dla operacji Transient dla Przypadku testowego A

Wraz ze wzrostem liczby powtórzeń coraz słabsze wyniki osiągają **NiquIoCPartial** oraz najpopularniejsze rozwiązania. Wszystkie notują spory wzrost czasów. Dla **NiquIoCFull** i najszybszych rozwiązań ten wzrost jest nieznaczny.

Liczba	1000		
	min	max	avg
NiquIoCFull	18	19	18
LightInject	19	20	19
SimpleInjector	28	30	29
DryLoc	29	30	29
Grace	37	38	37
NiquIoCPartial	171	198	173
StructureMap	414	457	417
Autofac	584	609	587
Unity	803	961	813
Windsor	1511	1799	1529
Ninject	8745	9610	8934

Tab. 6: Wyniki testów dla 1000 powtórzeń dla operacji Transient dla Przypadku testowego A

Test dla 1000 powtórzeń potwierdził, że czasy dla **NiquIoCPartial** oraz najpopularniejszych rozwiązań rosną liniowo wraz ze wzrostem liczby powtórzeń. Czasy dla **NiquIoCFull** oraz najszybszych rozwiązań również wzrosły, ale zaledwie dwukrotnie w stosunku do dziesięciokrotnego wzrostu liczby powtórzeń.

5.1.4 Wyniki Resolve dla TransientSingleton

Liczba	1		
	min	max	avg
Autofac	0	0	0
NiquIoCPartial	1	1	1
Windsor	1	2	1
NiquIoCFull	5	6	5
Unity	8	9	8
SimpleInjector	9	9	9
Grace	9	10	10
Ninject	9	10	10
StructureMap	9	10	10
LightInject	14	15	14
DryLoc	18	19	18

Tab. 7: Wyniki testów dla 1 powtórzenia dla operacji TransientSingleton dla Przypadku testowego A

Dla 1 powtórzenia najmniejsze czasy uzyskały **NiquIoCPartial** oraz najpopularniejsze rozwiązania.

Liczba	10		
	min	max	avg
NiquIoCPartial	2	3	2
NiquIoCFull	5	6	5
Autofac	6	7	6
SimpleInjector	9	10	9
Grace	10	10	10
StructureMap	12	12	12
Windsor	12	14	12
LightInject	14	14	14
Unity	14	15	14
DryLoc	19	19	19
Ninject	65	82	67

Tab. 8: Wyniki testów dla 10 powtórzeń dla operacji TransientSingleton dla Przypadku testowego A

Gdy jest 10 powtórzeń czasy wszystkich rozwiązań wydają się być do siebie zbliżone. Wyjątkiem są tylko dwa rozwiązania: **NiquIoCPartial** osiąga wyniki trochę lepsze niż pozostałe rozwiązania, a **Ninject** trochę gorsze.

Liczba	100		
	min	max	avg
NiquIoCFull	6	6	6
SimpleInjector	10	11	10
Grace	10	11	11
NiquIoCPartial	14	15	14
LightInject	14	15	15
DryLoc	20	21	20
StructureMap	35	40	36
Autofac	53	62	54
Unity	73	82	74
Windsor	117	140	119
Ninject	626	743	641

Tab. 9: Wyniki testów dla 100 powtórzeń dla operacji TransientSingleton dla Przypadku testowego A

W tym teście również gdy liczba powtórzeń wzrosła do 100, to najlepsze czasy zaczęły uzyskiwać **NiquIoCFull** i najszybsze rozwiązania. **NiquIoCPartial** oraz najpopularniejsze rozwiązania zanotowały kilkukrotny wzrost czasów.

Liczba	1000		
	min	max	avg
NiquIoCFull	10	12	11
SimpleInjector	17	19	17
Grace	18	19	18
LightInject	20	21	20
DryIoc	31	32	32
NiquIoCPartial	120	123	121
StructureMap	252	306	256
Autofac	531	561	535
Unity	659	753	663
Windsor	1161	1302	1177
Ninject	6297	7218	6463

Tab. 10: Wyniki testów dla 1000 powtórzeń dla operacji TransientSingleton dla Przypadku testowego A

Przy 1000 powtórzeń **NiquIoCFull** oraz wszystkie najszybsze rozwiązania uzyskały czasy o rząd wielkości mniejsze niż **NiquIoCPartial** i najpopularniejsze rozwiązania.

5.1.5 Wyniki Resolve dla PerThread

Liczba	1		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	1	1	1
NiquIoCFull	2	2	2
Ninject	3	3	3
Grace	4	4	4
Unity	7	8	7
SimpleInjector	7	8	8
StructureMap	9	10	9
LightInject	50	53	50
DryIoc	71	73	71

Tab. 11: Wyniki testów dla 1 powtórzenia dla operacji PerThread dla Przypadku testowego A

Z racji tego, że wszystkie obiekty są tworzone w jednym wątku czasy powinny być zbliżone do wyników operacji resolve dla **Singleton**. Za wyjątkiem dwóch rozwiązań - **LightInject** i **DryIoc**, wszystkie pozostałe uzyskały zbliżone czasy. Te dwa rozwiązania najwyraźniej nie radzą sobie dla tego typu rejestracji.

Liczba	1000		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	1	1	1
NiquIoCFull	2	2	2
Grace	4	4	4
Ninject	6	7	6
SimpleInjector	8	8	8
Unity	8	9	8
StructureMap	10	10	10
LightInject	50	52	51
DryIoc	72	74	72

Tab. 12: Wyniki testów dla 1000 powtórzeń dla operacji PerThread dla Przypadku testowego A

Podobnie jak w przypadku resolve dla **Singleton**, tutaj również wzrost liczby powtórzeń nie miał wpływu na wzrost czasu.

5.1.6 Wyniki Resolve dla FactoryMethod

Liczba	1		
	min	max	avg
Autofac	0	0	0
DryIoc	0	0	0
LightInject	0	0	0
NiquIoCPartial	0	0	0
NiquIoCFull	0	0	0
SimpleInjector	1	1	1
Unity	1	1	1
Windsor	1	1	1
Grace	2	2	2
StructureMap	7	7	7
Ninject	8	10	9

Tab. 13: Wyniki testów dla 1 powtórzenia dla operacji FactoryMethod dla Przypadku testowego A

Dla tego testu dla 1 powtórzenia wszystkie rozwiązania za wyjątkiem **StructureMap** i **Ninject** uzyskały zbliżone rezultaty. Te dwa rozwiązania osiągnęły wyniki słabsze niż reszta.

Liczba	10		
	min	max	avg
DryLoc	0	0	0
LightInject	1	1	1
NiquIoCPartial	1	1	1
NiquIoCFull	1	1	1
SimpleInjector	2	2	2
Grace	3	5	3
Autofac	5	8	6
Windsor	10	11	10
StructureMap	11	11	11
Unity	14	15	14
Ninject	53	60	55

Tab. 14: Wyniki testów dla 10 powtórzeń dla operacji FactoryMethod dla Przypadku testowego A

Wraz ze wzrostem liczby powtórzeń tym razem również wszystkie najpopularniejsze rozwiązania zanotowały liniowy wzrost czasów. Oba rozwiązania przedstawione w tej pracy oraz wszystkie najszybsze rozwiązania uzyskały zbliżone wyniki.

Liczba	100		
	min	max	avg
DryLoc	6	6	6
LightInject	6	8	7
NiquIoCFull	11	12	12
SimpleInjector	11	12	12
NiquIoCPartial	12	12	12
Grace	12	13	12
Autofac	49	57	50
StructureMap	52	53	52
Windsor	95	97	95
Unity	141	168	143
Ninject	521	597	531

Tab. 15: Wyniki testów dla 100 powtórzeń dla operacji FactoryMethod dla Przypadku testowego A

Gdy liczba powtórzeń wzrosła do 100, to wszystkie rozwiązania zanotowały liniowy wzrost czasów.

Liczba	1000		
	min	max	avg
LightInject	60	65	62
DryIoc	63	65	64
SimpleInjector	99	103	100
Grace	103	107	104
NiquIoCFull	113	115	114
NiquIoCPartial	118	121	120
StructureMap	431	447	435
Autofac	482	490	484
Windsor	926	1012	938
Unity	1407	1532	1423
Ninject	5298	5469	5339

Tab. 16: Wyniki testów dla 1000 powtórzeń dla operacji FactoryMethod dla Przypadku testowego A

Dla 1000 powtórzeń liniowy wzrost utrzymał się dla wszystkich rozwiązań. Rozwiązania zaprezentowane w tej pracy dla tego testu osiągnęły trochę słabsze wyniki niż wszystkie najszybsze rozwiązania (czasy prawie dwukrotnie słabsze niż najlepszy wśród najszybszych rozwiązań **LightInject**), ale dużo lepsze niż wszystkie najpopularniejsze rozwiązania (czasy prawie czterokrotnie mniejsze niż najlepszy wśród najpopularniejszych rozwiązań **StructureMap**).

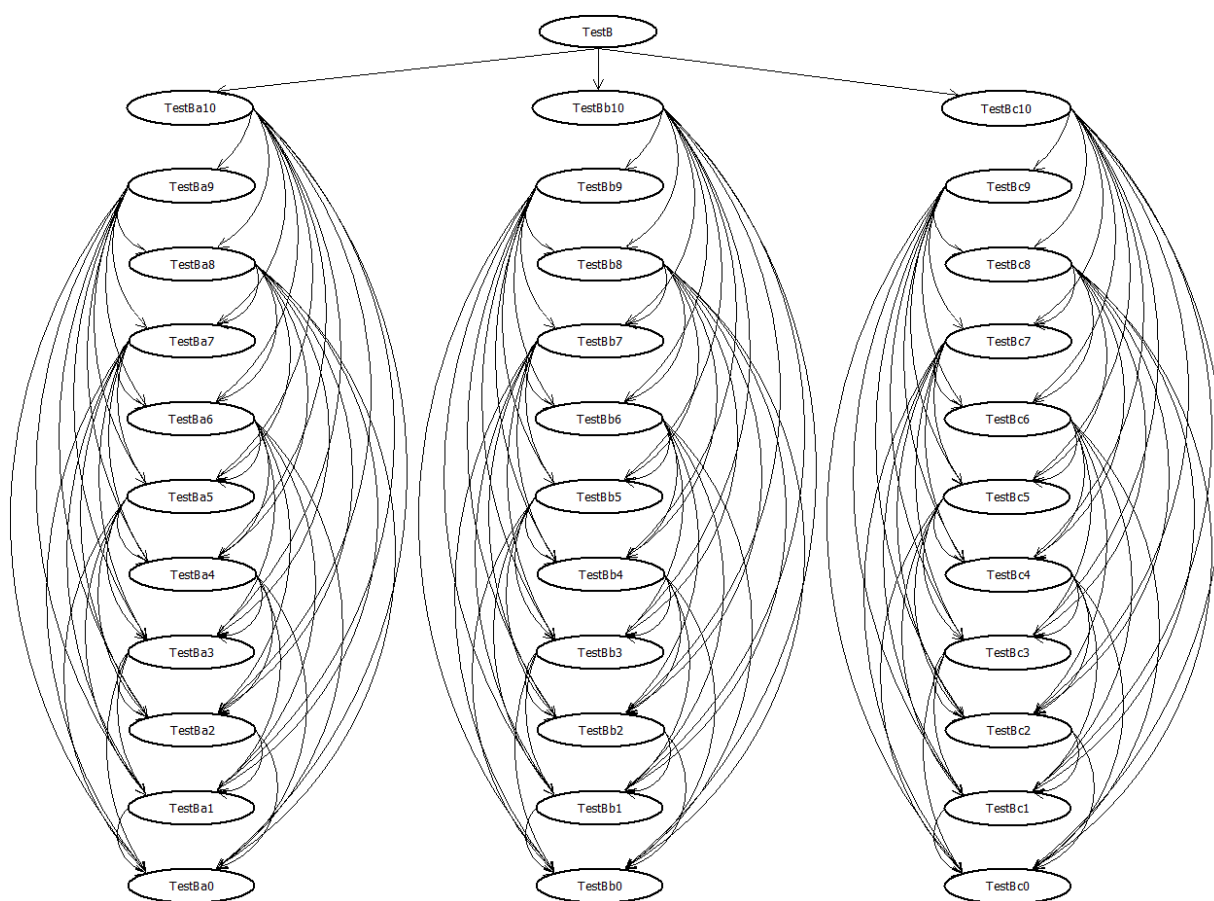
5.2 Przypadek testowy B

5.2.1 Opis

Ten test jest bardzo podobny do przypadku testowego A. Dochodzi tutaj 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. 33 przedstawia te zależności.

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

- **TestBa0** - 1 obiekt,
- **TestBa1** - 2 obiekty (obiekt typu **TestBa1** i obiekt typu **TestBa0**),



Rys. 33: 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**, zostaje utworzony: 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.

5.2.2 Wyniki Resolve dla Singleton

Liczba	1		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	4	4	4
SimpleInjector	6	6	6
LightInject	6	7	6
DryLoc	8	8	8
Grace	8	8	8
NiquIoCFull	8	9	8
Ninject	9	10	9
Unity	23	24	23
StructureMap	32	33	33

Tab. 17: Wyniki testów dla 1 powtórzenia dla operacji Singleton dla Przypadku testowego B

Czasy dla wszystkich rozwiązań wzrosły trzykrotnie w stosunku do przypadku testowego A. **Autofac** i **Windsor** osiągnęły najlepsze wyniki, jednakże dopiero dla tego przypadku testowego ich czasy są dużo mniejsze niż pozostałych rozwiązań. Z drugiej strony wyniki dla **Unity** oraz **StructureMap** dużo bardziej odstają od reszty.

Liczba	1000		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	4	4	4
SimpleInjector	6	6	6
LightInject	6	7	6
Grace	8	8	8
NiquIoCFull	8	8	8
DryLoc	8	9	8
Ninject	12	13	12
Unity	24	25	24
StructureMap	33	34	33

Tab. 18: Wyniki testów dla 1000 powtórzeń dla operacji Singleton dla Przypadku testowego B

Podobnie jak dla przypadku testowego A, tutaj również wraz ze wzrostem liczby powtórzeń nie było znaczącego wzrostu czasu dla żadnego z rozwiązań.

5.2.3 Wyniki Resolve dla Transient

Liczba	1		
	min	max	avg
Autofac	2	2	2
NiquIoCPartial	5	5	5
Windsor	6	7	6
NiquIoCFull	26	27	26
Unity	28	29	28
LightInject	31	33	32
StructureMap	33	34	33
Ninject	36	38	36
SimpleInjector	40	41	40
DryLoc	43	44	43
Grace	50	59	51

Tab. 19: Wyniki testów dla 1 powtórzenia dla operacji Transient dla Przypadku testowego B

W tym teście dla 1 powtórzenia **Autofac**, **NiquIoCPartial** oraz **Windsor** zanotowały czasy o rząd wielkości mniejsze niż pozostałe rozwiązania. Dodatkowo najpopularniejsze rozwiązania osiągnęły niższe czasy niż najszybsze rozwiązania.

Liczba	10		
	min	max	avg
NiquIoCPartial	10	10	10
Autofac	21	25	21
NiquIoCFull	26	27	26
LightInject	32	33	32
SimpleInjector	40	42	41
DryLoc	44	45	44
StructureMap	45	46	45
Unity	49	50	49
Windsor	49	69	50
Grace	51	55	52
Ninject	276	337	284

Tab. 20: Wyniki testów dla 10 powtórzeń dla operacji Transient dla Przypadku testowego B

Wraz ze wzrostem liczby powtórzeń wszystkie najpopularniejsze rozwiązania zanotowały wzrost około 10-krotny. **NiquIoCPartial** zanotował wzrost 2-krotny. **NiquIoCFull** oraz najszybsze rozwiązania nie zanotowały wzrostu czasów.

Liczba	100		
	min	max	avg
NiquIoCFull	32	33	32
LightInject	37	38	37
SimpleInjector	47	48	47
DryLoc	51	52	51
Grace	57	59	58
NiquIoCPartial	59	79	61
StructureMap	167	175	168
Autofac	189	201	191
Unity	260	271	262
Windsor	476	610	484
Ninject	2733	3595	2790

Tab. 21: Wyniki testów dla 100 powtórzeń dla operacji Transient dla Przypadku testowego B

Gdy liczba powtórzeń wzrosła do 100, **NiquIoCPartial** zanotował 5-krotny wzrost. Najpopularniejsze rozwiązania zanotowały stały, około 10-krotny wzrosty. **NiquIoCFull** oraz wszystkie najszybsze rozwiązania osiągnęły nieduży wzrost, jednakże wszystkie osiągnęły o rząd wielkości lepsze wyniki niż najpopularniejsze rozwiązania.

Liczba	1000		
	min	max	avg
NiquIoCFull	66	68	66
LightInject	70	73	71
SimpleInjector	95	97	96
DryLoc	97	100	98
Grace	122	128	124
NiquIoCPartial	543	633	550
StructureMap	1333	1565	1346
Autofac	1890	2044	1903
Unity	2355	2419	2366
Windsor	4726	5665	4799
Ninject	28220	29976	28720

Tab. 22: Wyniki testów dla 1000 powtórzeń dla operacji Transient dla Przypadku testowego B

NiquIoCFull oraz wszystkie najszybsze rozwiązania zanotowały około 2-krotny wzrost czasów, **NiquIoCPartial** około 5-krotny, a wszystkie najpopularniejsze rozwiązania utrzymały około 10-krotny wzrost czasów.

5.2.4 Wyniki Resolve dla TransientSingleton

Liczba	1		
	min	max	avg
Autofac	1	2	1
Windsor	3	5	3
NiquIoCPartial	4	5	5
NiquIoCFull	16	16	16
Unity	27	28	27
SimpleInjector	28	28	28
Ninject	29	34	30
Grace	30	32	31
StructureMap	33	34	33
LightInject	48	50	48
DryLoc	56	57	56

Tab. 23: Wyniki testów dla 1 powtórzenia dla operacji TransientSingleton dla Przypadku testowego B

Wyniki dla tego testu są zbliżone do wyników dla **Transient**. Dla 1 powtórzenia najpopularniejsze rozwiązania osiągają lepsze wyniki niż najszybsze rozwiązania. Oba rozwiązania zaprezentowane w tej pracy posiadają czasy zbliżone do najpopularniejszy rozwiązań.

Liczba	10		
	min	max	avg
NiquIoCPartial	8	9	8
NiquIoCFull	16	17	16
Autofac	18	20	18
SimpleInjector	28	29	28
Grace	31	32	31
Windsor	37	46	37
StructureMap	40	46	40
Unity	46	47	46
LightInject	48	50	48
DryLoc	57	58	57
Ninject	214	239	219

Tab. 24: Wyniki testów dla 10 powtórzeń dla operacji TransientSingleton dla Przypadku testowego B

Dla 10 powtórzeń zarówno **NiquIoCPartial** jak i **NiquIoCFull** osiągnęły najmniejsze czasy. Są one jednak zbliżone do pozostały rozwiązań. Wyjątkiem jest jedynie **Ninject**, który osiągnął czasy o rząd wielkości większe niż reszta.

Liczba	100		
	min	max	avg
NiquIoCFull	18	19	18
SimpleInjector	32	34	32
Grace	35	37	35
NiquIoCPartial	41	57	42
LightInject	51	53	51
DryLoc	64	65	64
StructureMap	112	123	113
Autofac	172	206	173
Unity	226	263	228
Windsor	355	435	362
Ninject	2013	2426	2063

Tab. 25: Wyniki testów dla 100 powtórzeń dla operacji TransientSingleton dla Przypadku testowego B

Gdy liczba powtórzeń wzrosła do 100, **NiquIoCFull** oraz najszybsze rozwiązania zanotowały niewielki wzrost czasów. Z drugiej strony **NiquIoCPartial** oraz wszystkie najpopularniejsze rozwiązania zanotowały kilkukrotny wzrost.

Liczba	1000		
	min	max	avg
NiquIoCFull	37	39	38
SimpleInjector	61	62	61
Grace	69	72	69
LightInject	80	82	80
DryLoc	108	110	109
NiquIoCPartial	365	406	368
StructureMap	797	915	805
Autofac	1698	1832	1709
Unity	2016	2305	2032
Windsor	3543	4824	3637
Ninject	20440	22891	20858

Tab. 26: Wyniki testów dla 1000 powtórzeń dla operacji TransientSingleton dla Przypadku testowego B

Podobnie jak dla testu dla **Transient**, **NiquIoCPartial** i najpopularniejsze rozwiązania utrzymały liniowy, około 10-krotny wzrost. Dla pozostałych rozwiązań ten wzrost był około 2-krotny.

5.2.5 Wyniki Resolve dla PerThread

Liczba	1		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	4	5	4
NiquIoCFull	8	8	8
Ninject	9	10	9
Grace	13	13	13
SimpleInjector	23	23	23
Unity	23	25	23
StructureMap	32	34	33
DryLoc	246	250	247
LightInject	409	419	412

Tab. 27: Wyniki testów dla 1 powtórzenia dla operacji PerThread dla Przypadku testowego B

Dla tego testu sytuacja wygląda identycznie jak dla przypadku testowego A - **DryLoc** oraz **LightInject** nie poradziły sobie z rejestracją **PerThread** i zanotowały czas o rząd wielkości większe niż pozostałe rozwiązania.

Wyraźnie najlepsze wyniki osiągnęły **Autofac** i **Windsor**, a dalej **NiquIoCPartial** oraz **NiquIoCFull**. Czasy nieco większe osiągnęły również **Ninject**, a także **Grace**.

Liczba	1000		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	4	5	4
NiquIoCFull	8	8	8
Ninject	12	13	13
Grace	13	14	13
SimpleInjector	23	24	23
Unity	24	25	24
StructureMap	33	34	33
DryLoc	246	249	247
LightInject	409	420	413

Tab. 28: Wyniki testów dla 1000 powtórzeń dla operacji PerThread dla Przypadku testowego B

Gdy liczba powtórzeń wzrosła do 1000 żadne z rozwiązań nie zanotowało wzrostu czasów.

5.2.6 Wyniki Resolve dla FactoryMethod

Liczba	1		
	min	max	avg
DryLoc	0	0	0
NiquIoCPartial	0	0	0
NiquIoCFull	0	0	0
Autofac	1	1	1
LightInject	2	4	2
SimpleInjector	4	4	4
Windsor	4	5	4
Unity	4	6	4
Grace	7	9	7
Ninject	23	25	24
StructureMap	25	25	25

Tab. 29: Wyniki testów dla 1 powtórzenia dla operacji FactoryMethod dla Przypadku testowego B

Dla 1 powtórzenia wszystkie rozwiązania za wyjątkiem **Ninject** i **StructureMap** mają zbliżone, nieduże czasy. Te dwa rozwiązania zanotowały czasy o rząd wielkości większe niż pozostałe.

Liczba	10		
	min	max	avg
DryIoc	1	2	1
NiquIoCFull	3	4	3
LightInject	4	4	4
NiquIoCPartial	4	4	4
SimpleInjector	7	7	7
Grace	10	10	10
Autofac	16	17	16
Windsor	33	36	33
StructureMap	37	37	37
Unity	44	45	44
Ninject	166	182	170

Tab. 30: Wyniki testów dla 10 powtórzeń dla operacji FactoryMethod dla Przypadku testowego B

Najszybsze rozwiązania zanotowały około 2-krotny wzrost czasów, a z kolei dla **NiquIoCPartial** oraz najpopularniejszych rozwiązań ten wzrost było około 10-krotny. **NiquIoCFull** osiągnął kilkukrotny wzrost czasów, ale poradził sobie słabiej tylko od **DryIoc**.

Liczba	100		
	min	max	avg
DryIoc	19	19	19
LightInject	21	22	21
NiquIoCFull	35	37	35
SimpleInjector	35	37	36
NiquIoCPartial	37	39	38
Grace	38	39	39
Autofac	154	156	155
StructureMap	159	162	160
Windsor	298	321	301
Unity	436	443	439
Ninject	1601	1670	1622

Tab. 31: Wyniki testów dla 100 powtórzeń dla operacji FactoryMethod dla Przypadku testowego B

Wraz ze wzrostem powtórzeń do 100, większość rozwiązań zanotowała około 10-krotny wzrost czasów. Wyjątkami są **LightInject**, **SimpleInjector**, **Grace** i **StructureMap**, które zanotował około 5-krotny wzrost czasów.

Liczba	1000		
	min	max	avg
LightInject	180	186	182
DryIoC	185	188	186
SimpleInjector	305	311	308
Grace	312	319	315
NiquIoCFull	344	373	353
NiquIoCPartial	364	373	368
StructureMap	1347	1385	1359
Autofac	1545	1573	1550
Windsor	2939	3226	2971
Unity	4355	4492	4373
Ninject	16788	17602	16963

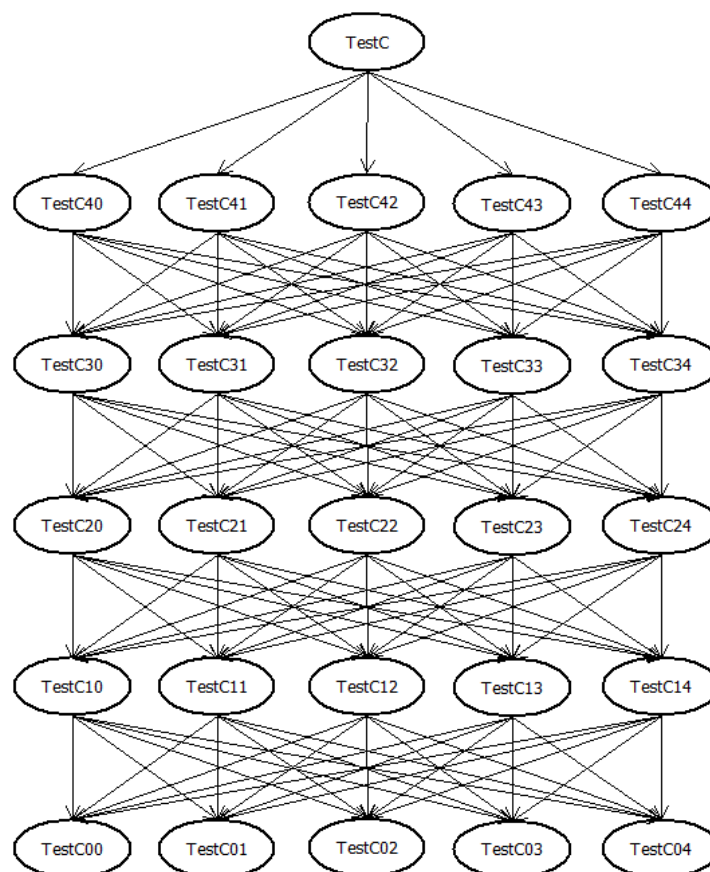
Tab. 32: Wyniki testów dla 1000 powtórzeń dla operacji FactoryMethod dla Przypadku testowego B

Gdy liczba powtórzeń wzrosła do 1000, wszystkie rozwiązania zanotowały około 9-krotny lub 10-krotny wzrost. Najmniejsze czasy zanotowały **LightInject** oraz **DryIoC**.

5.3 Przypadek testowy C

5.3.1 Opis

Ten przypadek testowy znacząco różni się od dwóch poprzednich. Jest 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 pierwszą cyfrę o 1 mniejszą, czyli są to obiekty typów od **TestC30** do **TestC34**. Dla tych i kolejnych typów zasada z konstruktorami wygląda tak samo. Na końcu są typy od **TestC00** do **TestC04**, które mają konstruktor bezparametrowy. Rys. 34 przedstawia graf zależności typów dla tego przypadku testowego.



Rys. 34: Graf zależności dla testu C.

Łatwo wywnioskować, że tworząc obiekty poszczególnych typów, liczba 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**, zostaje utworzony: 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.

5.3.2 Wyniki Resolve dla Singleton

Liczba	1		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	3	3	3
LightInject	4	4	4
SimpleInjector	4	4	4
DryIoc	5	5	5
Grace	5	5	5
NiquIoCFull	5	5	5
Ninject	6	6	6
Unity	15	16	15
StructureMap	22	23	22

Tab. 33: Wyniki testów dla 1 powtórzenia dla operacji Singleton dla Przypadku testowego C

Najlepsze wyniki osiągnął **Autofac** i **Windsor**, a najslabiej **Unity** oraz **StructureMap**. Pozostałe rozwiązania zanotowały zbliżone czasy.

Liczba	1000		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	3	3	3
LightInject	4	4	4
SimpleInjector	4	4	4
DryIoc	5	5	5
Grace	5	5	5
NiquIoCFull	5	5	5
Ninject	9	10	9
Unity	16	17	16
StructureMap	23	24	23

Tab. 34: Wyniki testów dla 1000 powtórzeń dla operacji Singleton dla Przypadku testowego C

Dla tego przypadku testowego sytuacja dla tego testu wygląda identycznie jak dla dwóch poprzednich przypadków testowych. Wraz ze wzrostem powtórzeń, czasy pozostają na tym samym poziomie, a tabela z wynikami jest zbliżona do tabeli dla przypadku testowego A i B.

5.3.3 Wyniki Resolve dla Transient

Liczba	1		
	min	max	avg
Autofac	2	2	2
NiquIoCPartial	3	4	4
Windsor	7	8	7
Unity	18	19	19
StructureMap	26	27	26
NiquIoCFull	31	33	31
LightInject	36	37	36
DryLoc	39	41	39
Ninject	38	42	39
SimpleInjector	40	42	41
Grace	61	63	61

Tab. 35: Wyniki testów dla 1 powtórzenia dla operacji Transient dla Przypadku testowego C

Dla 1 powtórzenia najlepsze wyniki osiągają najpopularniejsze rozwiązania oraz **NiquIoCPartial**. Dalej jest **NiquIoCFull**, a za nim wszystkie najszybsze rozwiązania. Najniższe czasy osiągnęły **Autofac** i **NiquIoCPartial**.

Liczba	10		
	min	max	avg
NiquIoCPartial	10	11	10
Autofac	23	25	24
NiquIoCFull	31	33	32
LightInject	36	39	37
DryLoc	40	41	40
StructureMap	40	42	40
SimpleInjector	41	43	41
Unity	47	51	47
Grace	62	65	62
Windsor	61	82	62
Ninject	345	375	353

Tab. 36: Wyniki testów dla 10 powtórzeń dla operacji Transient dla Przypadku testowego C

Tym razem również wszystkie najpopularniejsze rozwiązania zanotowały około 10-krotny wzrost, **NiquIoCPartial** około 2-krotny wzrost, a **NiquIoCFull** oraz wszystkie najszybsze rozwiązania osiągnęły podobne czasy co dla 1 powtórzenia.

Liczba	100		
	min	max	avg
NiquIoCFull	38	41	38
LightInject	43	49	44
DryIoc	47	48	47
SimpleInjector	49	51	49
Grace	73	74	73
NiquIoCPartial	72	80	74
StructureMap	178	207	181
Autofac	229	250	231
Unity	315	369	318
Windsor	594	769	608
Ninject	3430	4043	3511

Tab. 37: Wyniki testów dla 100 powtórzeń dla operacji Transient dla Przypadku testowego C

Pomimo że ten przypadek testowy znacząco różni się od poprzednich dwóch, to stosunek wzrostów czasów, wraz ze wzrostem powtórzeń dla tego testu wydaje się być taki sam. **NiquIoCFull** oraz najszybsze rozwiązania notują niewielki wzrost i ich czasy są na tym samym poziomie co dla 10 powtórzeń, a pozostałe rozwiązania mają wyniki około 9-10 razy większe.

Liczba	1000		
	min	max	avg
NiquIoCFull	81	83	82
LightInject	86	88	86
DryIoc	98	100	99
SimpleInjector	115	117	116
Grace	153	163	155
NiquIoCPartial	683	781	690
StructureMap	1520	1810	1540
Autofac	2274	2419	2288
Unity	2995	3359	3015
Windsor	5898	7708	6037
Ninject	36072	42914	37642

Tab. 38: Wyniki testów dla 1000 powtórzeń dla operacji Transient dla Przypadku testowego C

Tendencja wzrostów się utrzymała i **NiquIoCFull** osiągnął lepsze wyniki niż wszystkie najszybsze rozwiązania, a **NiquIoCPartial** lepsze niż wszystkie najpopularniejsze rozwiązania.

5.3.4 Wyniki Resolve dla TransientSingleton

Liczba	1		
	min	max	avg
Autofac	1	2	2
NiquIoCPartial	3	3	3
Windsor	3	3	3
NiquIoCFull	12	13	12
Unity	18	18	18
SimpleInjector	21	22	21
StructureMap	23	23	23
Grace	23	24	23
Ninject	25	30	26
LightInject	52	54	52
DryIoC	58	70	58

Tab. 39: Wyniki testów dla 1 powtórzenia dla operacji TransientSingleton dla Przypadku testowego C

Dla 1 powtórzenia najlepiej radzą sobie **Autofac**, **NiquIoCPartial** i **Windsor**. Pozostałe rozwiązania zanotowały czasy od kilku, do kilkunastu razy większe. Najsłabsze wyniki osiągnęły **LightInject** oraz **DryIoC**.

Liczba	10		
	min	max	avg
NiquIoCPartial	6	7	6
NiquIoCFull	12	13	12
Autofac	19	20	20
SimpleInjector	21	22	21
Grace	23	24	23
StructureMap	30	31	30
Windsor	34	41	35
Unity	39	41	40
LightInject	52	55	53
DryIoC	59	62	59
Ninject	199	231	203

Tab. 40: Wyniki testów dla 10 powtórzeń dla operacji TransientSingleton dla Przypadku testowego C

Gdy jest 10 powtórzeń zarówno **NiquIoCFull** jak i wszystkie najszybsze rozwiązania nie zanotowały wzrostu czasów. **NiquIoCPartial**, **Unity** oraz **StructureMap** zanotowały wzrost około 2-krotny, a **Autofac**, **Windsor** oraz **Ninject** około 10-krotny. Mimo że dla **LightInject** i **DryIoC** czasy nie wzrosły, to i tak dla tylu powtórzeń osiągnęły one jedne z najwyższych wyników.

Liczba	100		
	min	max	avg
NiquIoCFull	13	14	14
SimpleInjector	23	24	24
Grace	25	26	25
NiquIoCPartial	38	46	39
LightInject	55	57	56
DryIoc	65	68	65
StructureMap	80	84	80
Autofac	184	198	187
Unity	242	258	243
Windsor	338	400	343
Ninject	1903	2364	1951

Tab. 41: Wyniki testów dla 100 powtórzeń dla operacji TransientSingleton dla Przypadku testowego C

Przy 100 powtórzeniach wzrost czasów wygląda identycznie jak dla testu **Transient** - **NiquIoCFull** oraz najszybsze rozwiązania mają niewielki, a pozostałe rozwiązania kilkukrotny. **LightInject** i **DryIoC** tym razem zanotowały czasy lepsze niż wszystkie najpopularniejsze rozwiązania, ale wciąż słabsze niż **NiquIoCPartial**.

Liczba	1000		
	min	max	avg
NiquIoCFull	25	26	25
Grace	44	45	44
SimpleInjector	44	45	44
LightInject	82	85	83
DryIoc	105	107	106
NiquIoCPartial	347	359	349
StructureMap	549	593	553
Autofac	1810	1898	1831
Unity	2258	2428	2272
Windsor	3368	4115	3429
Ninject	19137	22595	19626

Tab. 42: Wyniki testów dla 1000 powtórzeń dla operacji TransientSingleton dla Przypadku testowego C

Dla 1000 powtórzeń najmniejszy czas zanotował **NiquIoCFull**, a dalej wszystkie najszybsze rozwiązania. **NiquIoCPartial** dla tego przypadku testowego również znalazł się za wszystkimi najszybszymi rozwiązaniami oraz przed wszystkimi najpopularniejszymi rozwiązaniami.

5.3.5 Wyniki Resolve dla PerThread

Liczba	1		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	3	3	3
NiquIoCFull	5	5	5
Ninject	6	7	6
Grace	8	9	8
Unity	15	16	15
SimpleInjector	16	16	16
StructureMap	22	23	22
DryIoc	157	162	158
LightInject	525	556	529

Tab. 43: Wyniki testów dla 1 powtórzenia dla operacji PerThread dla Przypadku testowego C

Dla tego przypadku testowego wyniki są podobne jak w poprzednich przypadkach testowych - najlepsze wyniki osiągnęły **Autofac** i **Windsor**, następnie dwa rozwiązania zaprezentowane w tej pracy, a dalej pozostałe rozwiązania. Najwyższe czasy uzyskały **DryIoc** oraz **LightInject**, które nie radzą sobie z rejestracją **PerThread**.

Liczba	1000		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	3	3	3
NiquIoCFull	5	5	5
Grace	8	9	8
Ninject	9	10	9
Unity	16	17	16
SimpleInjector	16	18	16
StructureMap	23	73	24
DryIoc	158	161	158
LightInject	525	550	529

Tab. 44: Wyniki testów dla 1000 powtórzeń dla operacji PerThread dla Przypadku testowego C

Test dla 1000 powtórzeń pokazał, że dla żadnego rozwiązania (dla tego testu), wraz ze wzrostem liczby powtórzeń, nie wzrasta czas.

5.3.6 Wyniki Resolve dla FactoryMethod

Liczba	1		
	min	max	avg
DryLoc	0	0	0
NiquIoCPartial	0	0	0
NiquIoCFull	0	0	0
Autofac	2	2	2
LightInject	2	2	2
SimpleInjector	3	4	3
Windsor	4	5	4
Grace	6	6	6
Unity	6	7	6
StructureMap	18	20	18
Ninject	26	28	27

Tab. 45: Wyniki testów dla 1 powtórzenia dla operacji FactoryMethod dla Przypadku testowego C

Gdy jest 1 powtórzenie najlepiej radzą sobie **NiquIoCPartial**, **NiquIoCFull** oraz najszybsze rozwiązania. Najpopularniejsze rozwiązania zanotowały czasy nieznacznie (**Autofac**, **Windsor**, **Unity**) lub znacznie (**StructureMap**, **Ninject**) słabsze.

Liczba	10		
	min	max	avg
DryLoc	2	2	2
NiquIoCFull	4	4	4
LightInject	4	5	4
NiquIoCPartial	5	5	5
SimpleInjector	7	7	7
Grace	9	9	9
Autofac	19	21	19
StructureMap	34	35	34
Windsor	38	41	39
Unity	56	58	56
Ninject	209	220	213

Tab. 46: Wyniki testów dla 10 powtórzeń dla operacji FactoryMethod dla Przypadku testowego C

Wraz ze wzrostem liczby powtórzeń, czasy dla wszystkich rozwiązań wzrosły - dla rozwiązań najszybszych był to wzrost około 2-krotny, a dla pozostałych około 10-krotny.

Liczba	100		
	min	max	avg
DryIoC	24	25	24
LightInject	24	27	25
Grace	42	44	42
NiquIoCFull	42	45	43
SimpleInjector	43	45	44
NiquIoCPartial	45	47	46
StructureMap	177	198	180
Autofac	186	197	189
Windsor	366	378	369
Unity	555	645	564
Ninject	2110	2474	2152

Tab. 47: Wyniki testów dla 100 powtórzeń dla operacji FactoryMethod dla Przypadku testowego C

Gdy jest 100 powtórzeń wyraźnie wyższe czasy osiągnęły najpopularniejsze rozwiązania. Najniższe czasy zanotowały **DryIoC** oraz **LightInject**, a pozostałe rozwiązania (**Grace**, **NiquIoCFull**, **SimpleInjector** i **NiquIoCPartial**) uzyskały podobne wyniki.

Liczba	1000		
	min	max	avg
LightInject	215	223	218
DryIoC	232	236	234
Grace	357	369	361
SimpleInjector	380	389	385
NiquIoCFull	419	440	424
NiquIoCPartial	446	479	451
StructureMap	1594	1748	1623
Autofac	1867	1979	1880
Windsor	3650	4077	3721
Unity	5561	5645	5577
Ninject	22141	23030	22398

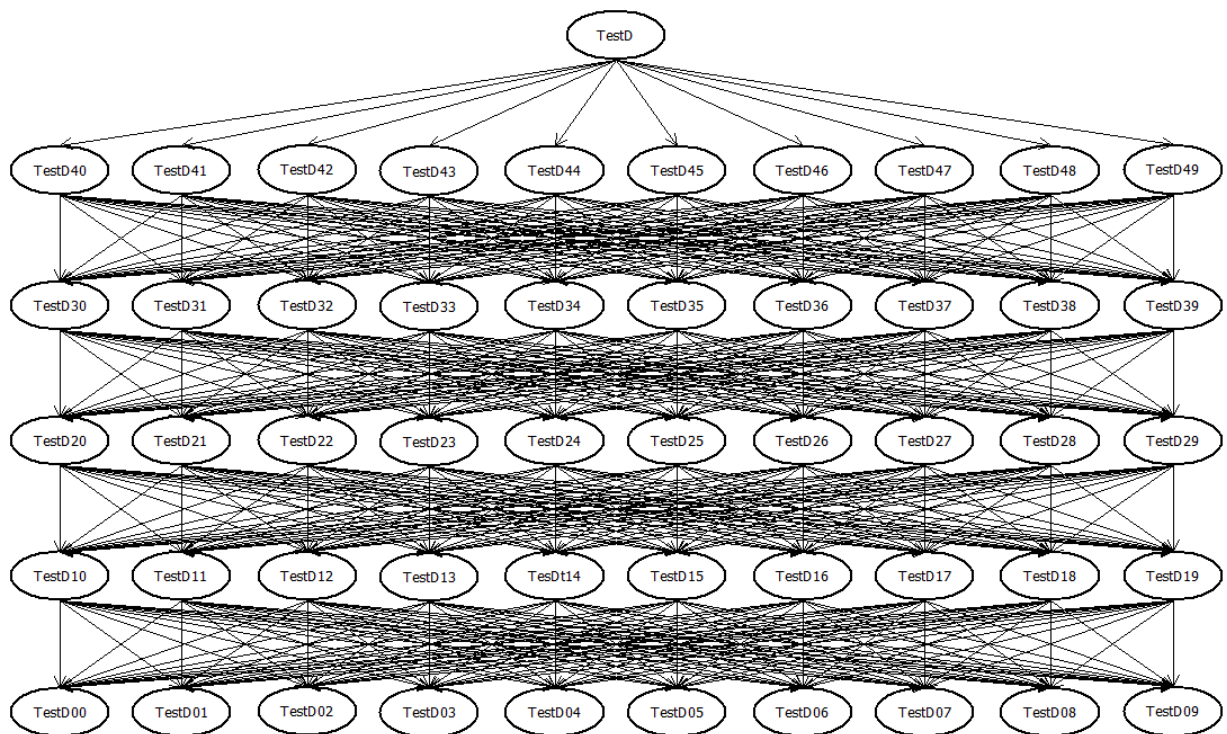
Tab. 48: Wyniki testów dla 1000 powtórzeń dla operacji FactoryMethod dla Przypadku testowego C

Przy 1000 powtórzeniach wszystkie rozwiązania zanotowały liniowy, około 10-krotny wzrost. Jednakże najmniejszy przyrost czasu zanotował **LightInject** i to on osiągnął najlepszy czas.

5.4 Przypadek testowy D

5.4.1 Opis

Podobnie jak przypadek testowy B jest analogiczny z przypadkiem A, tak ten przypadek, jest zbliżony do przypadku C. Jest tutaj tyle samo 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 pierwszą cyfrą o 1 mniejszą (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. 35.



Rys. 35: Graf zależności dla testu D.

Łatwo wywnioskować, że tworząc obiekty poszczególnych typów liczba 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**, zostaje utworzony: 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.

5.4.2 Wyniki Resolve dla Singleton

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

Tab. 49: Wyniki testów dla 1 powtórzenia dla operacji Singleton dla Przypadku testowego D

Dla tego testu tabela z wynikami bardzo przypomina tabelę dla przypadku testowego C, jednakże czasy są około 2-3 razy większe. Po raz kolejny dla rejestracji **Singleton** najlepsze wyniki osiągają **Autofac** i **Windsor**, a najslabsze **StructureMap** i **Unity**.

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

Tab. 50: Wyniki testów dla 10 powtórzeń dla operacji Singleton dla Przypadku testowego D

Jak można się było spodziewać, tym razem również wzrost liczby powtórzeń nie miał wpływu na wzrost wyników.

5.4.3 Wyniki Resolve dla Transient

Liczba	1		
	min	max	avg
NiquIoCPartial	36	38	37
StructureMap	102	110	103
Autofac	136	160	147
Unity	150	200	152
SimpleInjector	177	179	178
Windsor	177	261	182
NiquIoCFull	566	576	569
LightInject	797	831	812
DryLoc	979	996	982
Ninject	1016	1179	1039
Grace	1413	1472	1426

Tab. 51: Wyniki testów dla 1 powtórzenia dla operacji Transient dla Przypadku testowego D

Test dla tego przypadku testowego pokazał dużo większe różnice czasów niż dla poprzednich przypadków testowych. Dla 1 powtórzenia wyraźniej najlepszy czas zanotował **NiquIoCPartial**, który osiągnął czas o rząd wielkości niższy niż kolejne rozwiązanie. Średnie czasy zanotowały **StructureMap**, **Autofac**, **Unity** oraz **Windsor**. Pozostałe rozwiązania razem z **NiquIoCFull** osiągnęły bardzo wysokie wyniki.

Liczba	10		
	min	max	avg
SimpleInjector	205	212	206
NiquIoCPartial	285	291	287
StructureMap	586	603	591
NiquIoCFull	619	661	631
LightInject	841	878	853
DryIoc	1013	1025	1015
Unity	1073	1183	1080
Autofac	1161	1279	1188
Grace	1465	1558	1495
Windsor	1797	2421	1827
Ninject	10072	11599	10307

Tab. 52: Wyniki testów dla 10 powtórzeń dla operacji Transient dla Przypadku testowego D

Wzrost liczby powtórzeń do 10 pokazał, że zarówno **NiquIoCFull** jak i prawie wszystkie najszybsze rozwiązania zanotowały niewielki wzrost. Wyjątkiem jest tutaj **DryIoC**, który osiągnął 10-krotny przyrost czasu. Jednakże bardzo wysokie wyniki już dla 1 powtórzenia spowodowały, że z najszybszy rozwiązań jedynie **SimpleInjector** ma nie-duży czas. **NiquIoCPartial** pomimo 9-krotnego wzrostu czasu osiągnął drugi z najlepszych wyników. Wszystkie najpopularniejsze rozwiązania zanotowały wzrost około 10-krotny.

5.4.4 Wyniki Resolve dla TransientSingleton

Liczba	1		
	min	max	avg
NiquIoCPartial	17	18	17
StructureMap	60	69	61
Autofac	77	84	78
SimpleInjector	85	87	86
Windsor	85	102	87
Unity	115	140	116
NiquIoCFull	144	151	145
Grace	374	386	376
Ninject	503	572	511
DryIoc	905	922	910
LightInject	1285	1358	1310

Tab. 53: Wyniki testów dla 1 powtórzenia dla operacji TransientSingleton dla Przypadku testowego D

Dla tego testu ponownie dla 1 powtórzenia najmniejszy czas osiągnął **NiquIoCPartial**. Z pozostałych rozwiązań, to jedynie **Grace**, **Ninject**, **DryIoC** i **LightInject** zanotowały bardzo duże wyniki.

Liczba	10		
	min	max	avg
SimpleInjector	91	93	91
NiquIoCPartial	100	104	101
NiquIoCFull	151	159	152
StructureMap	165	173	167
Grace	381	396	383
Unity	683	771	689
Windsor	852	965	864
Autofac	893	964	910
DryIoc	936	983	942
LightInject	1297	1360	1316
Ninject	4732	5620	4818

Tab. 54: Wyniki testów dla 10 powtórzeń dla operacji TransientSingleton dla Przypadku testowego D

Gdy liczba operacji wzrosła do 10, to jak można się było spodziewać - **NiquIoCPartial** zanotował około 5-krotny wzrost, a najpopularniejsze rozwiązania zanotowały około 10-krotny wzrost. **NiquIoCFull** oraz najszybsze rozwiązania osiągnęły czasy zbliżone do czasów dla 1 powtórzenia. Najmniejszy czas uzyskał **SimpleInjector**, a trochę większe rezultaty zanotowały **NiquIoCPartial**, **NiquIoCFull** i **StructureMap**. Pozostałe rozwiązania osiągnęły zauważalnie większe czasy - od kilku do nawet kilkunastu razy.

5.4.5 Wyniki Resolve dla PerThread

Liczba	1		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	8	9	9
NiquIoCFull	18	18	18
Ninject	21	22	21
Grace	28	28	28
SimpleInjector	46	48	46
StructureMap	49	51	49
Unity	52	53	52
DryIoc	1042	1051	1045
LightInject	762748	795038	773697

Tab. 55: Wyniki testów dla 1 powtórzenia dla operacji PerThread dla Przypadku testowego D

Wyniki dla tego testu są podobne jak w pozostałych przypadkach testowych. Jedy-
nym wyjątkiem jest **LightInject**, który najwyraźniej nie poradził sobie z tak dużą liczbą
obiektów dla tego typu rejestracji i zanotował czas o 5 rzędów wielkości większy, niż
rozwiązania z najmniejszymi czasami - **Autofac**, **Windsor** i **NiquIoCPartial**.

Liczba	10		
	min	max	avg
Autofac	0	0	0
Windsor	0	0	0
NiquIoCPartial	8	10	9
NiquIoCFull	18	18	18
Ninject	21	22	21
Grace	28	28	28
SimpleInjector	46	48	46
StructureMap	49	51	49
Unity	52	53	52
DryLoc	1042	1051	1044
LightInject	763101	781163	774677

Tab. 56: Wyniki testów dla 10 powtórzeń dla operacji PerThread dla Przypadku testowego D

Przy 10 powtórzeniach nic się nie zmieniło i wszystkie wyniki są zbliżone do wyników
dla 1 powtórzenia.

5.4.6 Wyniki Resolve dla FactoryMethod

Liczba	1		
	min	max	avg
DryLoc	9	10	10
LightInject	12	13	12
NiquIoCPartial	15	17	15
SimpleInjector	20	20	20
NiquIoCFull	20	21	20
Grace	24	25	24
Autofac	67	80	68
StructureMap	90	93	91
Windsor	127	136	129
Unity	173	181	174
Ninject	654	748	674

Tab. 57: Wyniki testów dla 1 powtórzenia dla operacji FactoryMethod dla Przypadku
testowego D

NiquIoCPartial, **NiquIoCFull** oraz wszystkie najszybsze rozwiązania zanotowały zbli-
żone wyniki. Rozwiązania najpopularniejsze osiągnęły czasy od kilku, do kilkunastu

razy większe. Najlepszy wynik uzyskał **DryIoC**.

Liczba	10		
	min	max	avg
DryIoC	78	81	79
LightInject	79	83	80
SimpleInjector	128	131	129
Grace	127	149	130
NiquIoCPartial	164	170	166
NiquIoCFull	185	191	188
StructureMap	592	610	597
Autofac	671	799	685
Windsor	1227	1284	1238
Unity	1744	1839	1769
Ninject	6206	6515	6299

Tab. 58: Wyniki testów dla 10 powtórzeń dla operacji FactoryMethod dla Przypadku testowego D

Wzrost liczby powtórzeń do 10 spowodował wzrost czasów dla wszystkich rozwiązań. Najmniejszy wzrost czasów zanotowały najszybsze rozwiązania, jednak wciąż był on co najmniej 6-krotny. **NiquIoCPartial** i **NiquIoCFull** podobnie jak wszystkie najpopularniejsze rozwiązania zanotowały wzrost około 10-krotny. Jednakże nieduże czasy przy 1 powtórzeniu dla rozwiązań zaprezentowanych w tej pracy spowodowały, że mimo wszystko osiągnęły one znacząco lepsze wyniki niż wszystkie najpopularniejsze rozwiązania. Najlepsze czasy uzyskały wszystkie najszybsze rozwiązania, a najmniej z tych czasów osiągnęły **DryIoC** i **LightInject**.

6 Podsumowanie

Zaproponowane rozwiązania bardzo dobrze realizują postawione we wstępie pracy cele. Dla wszystkich przypadków testowych, dla każdego z rodzaju rejestracji, jedno z przedstawionych rozwiązań osiągało najlepsze wyniki albo odbiegały one niewiele od najlepszych wyników. Zawsze któreś z przedstawionych rozwiązań było w pierwszej trójce rozwiązań z najlepszymi czasami, a czasami nawet oba. Możliwość mieszanienia użyć zaprezentowanych rozwiązań (w jednym projekcie można korzystać z obu rozwiązań niezależnie) sprawia, że **NiquIoC** jest najbardziej wydajną implementacją wzorca wstrzykiwania zależności dla złożonych grafów zależności.

Dodatkowo wyniki testów pokazują, że rozwiązanie **Ninject** jest najmniej wydajne i nie zaleca się go używać przy bardzo rozbudowanych grafach. Jeśli chodzi natomiast o rozwiązanie, które osiąga dobre wyniki, to nie ma drugiego takiego (poza zaprezentowanymi w tej pracy), które zawsze dawałoby zadowalające rezultaty. **SimpleInjector** osiąga dość dobre wyniki 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 wzrostu czasu. Jednakże w wielu przypadkach dość duże czasy dla małej liczby operacji powodują, że wyniki dla wielu operacji są mimo wszystko bardzo duże. 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 gorsze niż dla **NiquIoC**.

6.1 Kontynuacja projektu

Zaprezentowane 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 dodanie automatycznej rejestracji wszystkich typów z danego assembly. Osobom wykorzystującym niniejsze rozwiązanie zaoszczędziłoby to sporo czasu (nie musiałyby rejestrować wszystkich typów). Dodatkowo podczas dodawania kolejnego typu do danego assembly, nie trzeba byłoby go ręcznie rejestrować. Jeszcze jednym dodatkiem wydaje się dodanie możliwości rejestracji wielu implementacji danego interfejsu. To jest również często spotykana funkcjonalność. Na koniec warto również wspomnieć o metodzie **BuildUp** dla **NiquIoCFull** - w niniejszej pracy tego zabrakło, a to również mogłoby być jednym z kolejnych rozszerzeń.

Spis rysunków

1	Przykładowa klasa bez zasady DIP	9
2	Przykładowa klasa z zasadą DIP	9
3	Definicja przykładowych klas A i B	10
4	Nowa definicja przykładowych klas A i B	11
5	Przykładowa klasa z atrybutem DependencyConstructor	13
6	Przykładowa klasa z atrybutem DependencyMethod	14
7	Przykładowa klasa z atrybutem DependencyProperty	14
8	Przykładowa metoda Abs	17
9	Przykład stworzenia delegata dla metody Abs	18
10	Przykład użycia delegata dla metody Abs	18
11	Interfejs IContainer	19
12	Interfejs IContainerRegister	19
13	Interfejs IContainerResolve	20
14	Interfejs IContainerBuildUp	20
15	Interfejs IContainerMember	21
16	Interfejs IObjectLifetimeManager	21
17	Klasa ContainerMember	24
18	Metoda Resolve klasy PartialEmitFunction	25
19	Metoda GetObject klasy PartialEmitFunction	26
20	Metoda CreateInstanceFunction klasy PartialEmitFunction	27
21	Metoda CreateObjectFunction klasy PartialEmitFunction	28
22	Metoda Resolve klasy FullEmitFunction	29
23	Metoda GetObject klasy FullEmitFunction	29
24	Metoda CreateInstanceFunction klasy FullEmitFunction	30
25	Metoda ValidateTypesCache klasy FullEmitFunction	30
26	Metoda CreateObjectFunction klasy FullEmitFunction	31
27	Metoda CreateObjectFunctionPrivate klasy FullEmitFunction	32

28	Metoda IsTransient klasy FullEmitFunction	32
29	Metoda IsSingleton klasy FullEmitFunction	33
30	Metoda AddObjectCreatedByObjectLifetimeManager klasy FullEmitFunction	34
31	Przykładowe klasy A, B i C	35
32	Graf zależności dla testu A.	43
33	Graf zależności dla testu B.	53
34	Graf zależności dla testu C.	63
35	Graf zależności dla testu D.	72

Spis tabel

1	Wyniki testów dla 1 powtórzenia dla operacji Singleton dla Przypadku testowego A	44
2	Wyniki testów dla 1000 powtórzeń dla operacji Singleton dla Przypadku testowego A	45
3	Wyniki testów dla 1 powtórzenia dla operacji Transient dla Przypadku testowego A	45
4	Wyniki testów dla 10 powtórzeń dla operacji Transient dla Przypadku testowego A	46
5	Wyniki testów dla 100 powtórzeń dla operacji Transient dla Przypadku testowego A	46
6	Wyniki testów dla 1000 powtórzeń dla operacji Transient dla Przypadku testowego A	47
7	Wyniki testów dla 1 powtórzenia dla operacji TransientSingleton dla Przypadku testowego A	47
8	Wyniki testów dla 10 powtórzeń dla operacji TransientSingleton dla Przypadku testowego A	48
9	Wyniki testów dla 100 powtórzeń dla operacji TransientSingleton dla Przypadku testowego A	48
10	Wyniki testów dla 1000 powtórzeń dla operacji TransientSingleton dla Przypadku testowego A	49

11	Wyniki testów dla 1 powtórzenia dla operacji PerThread dla Przypadku testowego A	49
12	Wyniki testów dla 1000 powtórzeń dla operacji PerThread dla Przypadku testowego A	50
13	Wyniki testów dla 1 powtórzenia dla operacji FactoryMethod dla Przypadku testowego A	50
14	Wyniki testów dla 10 powtórzeń dla operacji FactoryMethod dla Przypadku testowego A	51
15	Wyniki testów dla 100 powtórzeń dla operacji FactoryMethod dla Przypadku testowego A	51
16	Wyniki testów dla 1000 powtórzeń dla operacji FactoryMethod dla Przypadku testowego A	52
17	Wyniki testów dla 1 powtórzenia dla operacji Singleton dla Przypadku testowego B	54
18	Wyniki testów dla 1000 powtórzeń dla operacji Singleton dla Przypadku testowego B	55
19	Wyniki testów dla 1 powtórzenia dla operacji Transient dla Przypadku testowego B	55
20	Wyniki testów dla 10 powtórzeń dla operacji Transient dla Przypadku testowego B	56
21	Wyniki testów dla 100 powtórzeń dla operacji Transient dla Przypadku testowego B	56
22	Wyniki testów dla 1000 powtórzeń dla operacji Transient dla Przypadku testowego B	57
23	Wyniki testów dla 1 powtórzenia dla operacji TransientSingleton dla Przypadku testowego B	57
24	Wyniki testów dla 10 powtórzeń dla operacji TransientSingleton dla Przypadku testowego B	58
25	Wyniki testów dla 100 powtórzeń dla operacji TransientSingleton dla Przypadku testowego B	58
26	Wyniki testów dla 1000 powtórzeń dla operacji TransientSingleton dla Przypadku testowego B	59
27	Wyniki testów dla 1 powtórzenia dla operacji PerThread dla Przypadku testowego B	59

28	Wyniki testów dla 1000 powtórzeń dla operacji PerThread dla Przypadku testowego B	60
29	Wyniki testów dla 1 powtórzenia dla operacji FactoryMethod dla Przypadku testowego B	60
30	Wyniki testów dla 10 powtórzeń dla operacji FactoryMethod dla Przypadku testowego B	61
31	Wyniki testów dla 100 powtórzeń dla operacji FactoryMethod dla Przypadku testowego B	61
32	Wyniki testów dla 1000 powtórzeń dla operacji FactoryMethod dla Przypadku testowego B	62
33	Wyniki testów dla 1 powtórzenia dla operacji Singleton dla Przypadku testowego C	64
34	Wyniki testów dla 1000 powtórzeń dla operacji Singleton dla Przypadku testowego C	64
35	Wyniki testów dla 1 powtórzenia dla operacji Transient dla Przypadku testowego C	65
36	Wyniki testów dla 10 powtórzeń dla operacji Transient dla Przypadku testowego C	65
37	Wyniki testów dla 100 powtórzeń dla operacji Transient dla Przypadku testowego C	66
38	Wyniki testów dla 1000 powtórzeń dla operacji Transient dla Przypadku testowego C	66
39	Wyniki testów dla 1 powtórzenia dla operacji TransientSingleton dla Przypadku testowego C	67
40	Wyniki testów dla 10 powtórzeń dla operacji TransientSingleton dla Przypadku testowego C	67
41	Wyniki testów dla 100 powtórzeń dla operacji TransientSingleton dla Przypadku testowego C	68
42	Wyniki testów dla 1000 powtórzeń dla operacji TransientSingleton dla Przypadku testowego C	68
43	Wyniki testów dla 1 powtórzenia dla operacji PerThread dla Przypadku testowego C	69
44	Wyniki testów dla 1000 powtórzeń dla operacji PerThread dla Przypadku testowego C	69

45	Wyniki testów dla 1 powtórzenia dla operacji FactoryMethod dla Przypadku testowego C	70
46	Wyniki testów dla 10 powtórzeń dla operacji FactoryMethod dla Przypadku testowego C	70
47	Wyniki testów dla 100 powtórzeń dla operacji FactoryMethod dla Przypadku testowego C	71
48	Wyniki testów dla 1000 powtórzeń dla operacji FactoryMethod dla Przypadku testowego C	71
49	Wyniki testów dla 1 powtórzenia dla operacji Singleton dla Przypadku testowego D	73
50	Wyniki testów dla 10 powtórzeń dla operacji Singleton dla Przypadku testowego D	74
51	Wyniki testów dla 1 powtórzenia dla operacji Transient dla Przypadku testowego D	74
52	Wyniki testów dla 10 powtórzeń dla operacji Transient dla Przypadku testowego D	75
53	Wyniki testów dla 1 powtórzenia dla operacji TransientSingleton dla Przypadku testowego D	75
54	Wyniki testów dla 10 powtórzeń dla operacji TransientSingleton dla Przypadku testowego D	76
55	Wyniki testów dla 1 powtórzenia dla operacji PerThread dla Przypadku testowego D	76
56	Wyniki testów dla 10 powtórzeń dla operacji PerThread dla Przypadku testowego D	77
57	Wyniki testów dla 1 powtórzenia dla operacji FactoryMethod dla Przypadku testowego D	77
58	Wyniki testów dla 10 powtórzeń dla operacji FactoryMethod dla Przypadku testowego D	78

Literatura

- [1] Ian Griffiths, Matthew Adams, Jesse Liberty, C#. Programowanie, Wydanie VI, Helion, 2012
- [2] Mark Seemann, Dependency Injection in .NET, Manning, 2012
- [3] Serge Lidin, Expert .NET 2.0 IL Assembler, Apress, 2006
- [4] Robert C. Martin, Czysty kod. Podręcznik dobrego programisty, Helion, 2014
- [5] [https://en.wikipedia.org/wiki/SOLID_\(object-oriented_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))
- [6] Daniel Palme, IoC Container Benchmark - Performance comparison
<http://www.palmmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison>
- [7] Philip Laureano, Introducing Hiro, the World's Fastest IOC Container, Part I: Design Diary
<https://www.codeproject.com/Articles/35345/Introducing-Hiro-the-Worlds-Fastest-IOC-Container-.aspx>