Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Kierunek: Informatyka

Adrian Mularczyk

Stworzenie wydajnego wzorca wstrzykiwania zależności dla złożonych grafów zależności

Praca wykonana pod kierunkiem dr. Wiktora Zychli

Wrocław, 2016

Spis treści

1 Wstęp

1.1 Cel pracy

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

2 Wstrzykiwanie zależnosci

Jest to zbiór zasad projektowania oprogramowania i wzorców, które pozwalają nam rozwijać luźno powiązany kod [1].
Jakiemu celowi ma służyć wstrzykiwanie zależności? Wstrzykiwanie zależnści nie jestem celem samym w sobie, raczej jest to środek do celu. Ostatecznie celem większości technik programowania jest dostarczenie jak najwydajniej działającego oprogramowania. Jednym z aspektów tego jest napisanie utrzymywalnego kodu.
O ile nie pisze się prototypu lub aplikacji, które nigdy nie mają kolejnych wersji (kończą się na wersji 1), to wkrótce będzie trzeba zająć się utrzymaniem i rozwijaniem istniejącego kodu. Aby być w stanie pracować wydajnie z takim kodem bazowym, musi on być jak najlepiej utrzymywalny.
Wstrzykiwanie zależności jest niczym więcej niż techniką, która umożliwia luźne powiązania, a luźne powiązania sprawiają, że kod jest rozszerzalny i łatwy w utrzymaniu. [1]
Wstrzykiwanie zależności może odbywać się na 3 sposoby:
- wstrzykiwanie przez konstruktor (główna i najbardziej popularna);
- wstrzykiwanie przez metodę;
- wstrzykiwanie przez propercję.

3 Implementacja

Stworzyłem interfejs IConatiner, który zawiera w sobie metody niezbędne w każdym kontenerze:
- IContainerMember RegisterType<T>() where T : class;
- IContainerMember RegisterType<TFrom, TTo>() where TTo : TFrom;
- IContainerMember RegisterType<T>(Func<object> objectFactory) where T : class;
- IContainerMember RegisterInstance<T>(T instance);
- T Resolve<T>(ResolveKind resolveKind);
- void BuildUp<T>(T instance, ResolveKind resolveKind).

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

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

W mojej implementacji kontenera każdy typ domyślnie ma czas życia Transient.
Interfejs IObjectLifetimeManager zawiera w sobie następujące metody:
- Func<object> ObjectFactory get; set;
- object GetInstance().

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

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

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

W aplikacji istnieje również atrybut DependencyConstrutor. Można go użyć przy definicji konstruktora danej klasy. Obiekty danej klasy tworzy się przy użyciu konstruktora. Dana klasa może mieć kilka konstruktorów. W mojej implementacji stworzyłem logikę wyboru konstruktora przy pomocy którego ma zostać stworzony obiekt. Jeśli jest kilka konstrutkrów, odpowiedni jest wybierany w następującej kolejności:
- Konstruktor z atrybutem DependencyConstrutor;
- Konstrutkro z największą liczbą parametrów.

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

3.1 Microsoft Intermediate Language

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

3.2 Reflection.Emit

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

3.3 Rozwiązanie

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

3.4 Rozwiązanie 1 - PartialEmitFunction

W pierwszym rozwiązanu, które nazwałem PartialEmitFunction tworzę delegata z wykorzystaniem Reflection.Emit. Delegat ten jako parametr przyjmuje listę obiektów, które są potrzebne do stworzenia obiektu danej klasy wykorzystując odpowiedni konstruktor. Jeśli kontener do stworzenia obiektu danej klasy wybrał konstruktor bezparametrowy, to do takiego delegata trafi pusta lista. Jeśli natomiast został wybrany konstruktor, który w parametrze np. potrzebuje obiektu klasy A i B, to do delegata zostanie przekazana lista zawierająca obiekt klasy A i B w kolejności takiej, jakiej są one zdefiniowane z konstruktorze.
Pseudokod tego rozwiązana wygląda następująco:
1. Dla każdego argumentu umieść ten argument na stosie.
2. Na stosie umieść konstruktor docelowego typu.
3. Wywołaj konstrutkor i stworzony obiekt umięść na szczycie stosu.
4. Zwróć obiekt ze szczytu stosu.
To rozwiązanie nazwał "Partial", ponieważ tylko część operacji niezbędnych do stworzenia obiektu jest zakodowanych w języku IL (pobranie argumentów i stworzenie obiektu). W tym rozwiązaniu musimy wcześniej stworzyć obiekty, które są potrzebne do stworzenia docelowego obiektu.
Zalety tego rozwiązania:

Wady tego rozwiązania:

3.5 Rozwiązanie 2 - FullEmitFunction

W drugim rozwiążaniu, które nazwałem FullEmitFunction tworzę delegata bezparametrowego. Sam tworzy on wszystkie obiekty, które są mu potrzebne do stworzenia docelowego obiektu. Działa on rekurencyjnie i najpierw na stosie umieszcza wszystkie operacje niezbędne do stworzenia obiektu danego typu, a następnie tworzy docelowy obiekt.
Pseudokod tego rozwiązania wygląda następująco: 1. Czy konstruktor danego typu potrzebuje jakiś argument?
1b. Tak - Dla każdego parametru konstruktora wywołaj rekurencyjnie funkcję z argumentem wejściowym jako typ parametru.
2. Na stosie umieść konstruktor docelowego typu.
3. Wywołaj konstrutkor i stworzony obiekt umięść na szczycie stosu.
4. Zwróć obiekt ze szczytu stosu.
Jak łatwo zauważyć, to rozwiązanie jest pełne ("Full"), ponieważ wszystkie niezbędne obiekty zostaną stworzonę przy użyciu IL w jednej metodzie.
Zalety tego rozwiążania:
Wady tego rozwiązania:

4 Testy wydajnościowe

<wyniki testów i ich opis>

5 Podsumowanie

<parę słów na koniec>

Literatura

[1] Dependency Injection in .NET, Mark Seemann (str. 4-5)

[2] Expert .NET 2.0 IL Assembler, Serge Lidin (str. 3-7)