Projektopgave1: MyIoC

Fag:
Akademiuddannelsen i informationsteknologi
Videregående Programmering

Lavet af:
Klaus Mark

Dato:
20. marts 2016

Indholdsfortegnelse

Indledning

Jeg kender principperne bag ”Inversion Of Control” og ”Dependency injection”, og jeg bruger det on del. Jeg ender ofte i situationer hvor jeg skal konstruere objekter med mange afhængigheder og det er besværligt, grimt og gavner ikke absolut ikke læsevenligheden. Jo flere lag af abstraktioner, jo flere afhængigheder og jo flere unit tests, jo flere afhængigheder.

Et værktøj til at afhjælpe dette, kan være at bruge en IoC container. En IoC container er et stykke kode der kan holde styr på alle afhængigheder og når der er behov for det, konstruere de konkrete instanser. Jeg har aldrig brugt en IoC container men jeg har hørt lidt om hvad den kan og gør. Den IoC container jeg har hørt om, er CastleWindsor. Derfor skal min opgave være at undersøge og afprøve CastleWindsor og derefter lave min egen simple version.

Dependency Injection / Inversion of Control

En klasse skal have et ansvar og kun et ansvar, der skal så at sige kun være en grund til at ændre den. En person-klasse skal for eksempel ikke vide hvordan den gemme sig selv, ej heller hvilken konkret klasse der kan. Før interfaces var løsningen nedavning. En klasse kunne kræve en klasse der nedarver fra en abstrakt personGemmer-klasse i sin constructor og bruge den. I dag vil man blot kræve en klasse der implementere et interface, men princippet er præcist det samme. Det jeg lige har beskrevet er faktisk det der hedder Dependency Injection, man skubber så at sige afhængigheder ind i klassen, som derfor ikke skal forholde sig til hvilke klasser den får eller hvordan man laver dem. Dette ville nemlig bryde med at en klasse kun har et ansvar, for det at konstruere andre klasser er et ansvar. Dependency injection er en måde hvorpå vi kan fravige andre ansvar, så som at vide noget som helt om hvordan man gemmer sig selv, eller hvem der kan, man kan med andre ord sige at man omvender noget styring (inversion of control).

Forundersøgelser

Jeg startede med at prøve at bruge CastleWindsor, for at se hvordan en IoC rent faktisk fungere. Jeg startede med at lave 3 interfaces

```
public interface ISayHi
{
    void Speak();
}
```

```
public interface ISayBye
{
    void Speak();
}
```

```
public interface IConsoleAbstractions
{
    void WriteLine(string line);
}
```

Og så lavede jeg 3 implementeringer af disse interfaces:

```
public class SayHi : ISayHi
{
    private readonly IConsoleAbstractions _consoleAbstractions;

    public SayHi(IConsoleAbstractions consoleAbstractions)
    {
        _consoleAbstractions = consoleAbstractions;
    }

    public void Speak()
    {
        _consoleAbstractions.WriteLine("Hi there!");
    }
}
```

```
public class SayBye : ISayBye
{
    private readonly IConsoleAbstractions _consoleAbstractions;

    public SayBye(IConsoleAbstractions consoleAbstractions)
    {
        _consoleAbstractions = consoleAbstractions;
    }

    public void Speak()
    {
        _consoleAbstractions.WriteLine("Bye now...");
    }
}
```

```
public class ConsoleAbstractions : IConsoleAbstractions
{
    public void WriteLine(string line)
    {
        Console.WriteLine(line);
    }
}
```

Samt en klasse der bruger alle 3 interfaces.

```
public class SayHiAndBye
{
    private readonly IConsoleAbstractions _consoleAbstractions;
    private readonly ISayHi _sayHi;
    private readonly ISayBye _sayBye;

    public SayHiAndBye(IConsoleAbstractions consoleAbstractions, ISayHi sayHi, ISayBye sayBye)
    {
        _consoleAbstractions = consoleAbstractions;
        _sayHi = sayHi;
        _sayBye = sayBye;
    }

    public void Speak()
    {
        _consoleAbstractions.WriteLine($"I'm {nameof(SayHiAndBye)} and now i'm asking stuff to speak");
        _sayHi.Speak();
        _sayBye.Speak();
    }
}
```

Læg mærke til at SayHi, SayBye og SayHiAndBye alle bruger en IConsoleAbstractions, hvilket ”tvinger” IoC’en til at være recursiv.

På denne måde testede jeg hvordan CastleWindsor gjorde:

```
public class SayHiAndByeWithCastleWindsorAsIoC
{
    public static void GoAhead()
    {
        var ioc = new WindsorContainer();
        ioc.Register(Component.For<SayHiAndBye>());
        ioc.Register(Component.For<ISayHi>().ImplementedBy<SayHi>());
        ioc.Register(Component.For<ISayBye>().ImplementedBy<SayBye>());
        ioc.Register(Component.For<IConsoleAbstractions>().ImplementedBy<ConsoleAbstractions>());

        var sayHiAndBye = ioc.Resolve<SayHiAndBye>();

        sayHiAndBye.Speak();
    }
}
```

Ideen bag min løsning

Grundlæggende forestillede jeg mig at jeg med en generisk Register<TInterface, TImplementation> var i stand til at får typerne ud, uden at brugeren af mit API skulle bruge typeof() eller måske endda give mig instanser af implementationerne. En af funktionerne i en IoC container er også at konstruere de afhængigheder der er brug for, når de skal bruges. Jeg forstillede mig også at jeg kunne holde de to typer, altså den registrerede afhængighed og den dertilhørende implementation i en Dictionay<Type, Type>.

Det ville også virkede fint, indtil det gik op for mig at denne løsning ville give en ny instans af en afhængighed hver gang. Dette anså jeg som et problem, da man vel som udgangspunkt vil være i stand til at vælge om samme instans skal bruges eller om en ny skal laves hver gang. For at løse dette ville jeg introducere jeg en ”Implementation”-klasse, der skulle holde styr på typen, hvorvidt denne implementation altid skal give samme instans, samt en variabel til at holde en reference til instansen, hvis den skal bruges.

```
internal class Implementation
{
    public Type TypeOfImplementation { get; set; }
    public object InstanceOfImplementation { get; set; }
    public bool UseSingleInstance { get; set; }
}
```

Sidste overvejelse der kom med, var muligheden for at give implementationen parametre, forestil dig en klasse der er afhængig af ICard, men som skal vide hvor mange kort en hånd består af. Dette ville jeg løse ved også at holde disse i Implemenation-klassen.

```
internal class Implementation
{
    public Type TypeOfImplementation { get; set; }
    public object InstanceOfImplementation { get; set; }
    public bool UseSingleInstance { get; set; }
    public ImplementationParameter[] Parameters { get; set; }
}
```

Min løsning

Jeg holder data om hvilke afhængigheder der hænger sammen med hvilke implementationer i en Dictonary. Denne konstrueres i klassens constructor (dog ser det ikke helt sådan ud, men det er et andet tema)

```
private readonly Dictionary<Type, Implementation> _registrations = new Dictionary<Type, Implementation>();
```

Afhængigheder registreres med Register<>(); der er en række forskellige men her ses den de alle bruger

```
public void Register<TInterface, TImplementation>(bool useSingleInstance, params ImplementationParameter[] parameters)
{
    var typeOfInterface = typeof (TInterface);
    var typeOfImplementation = typeof (TImplementation);

    CheckIfImplementationImplementsInterface(typeOfInterface, typeOfImplementation);

    var implementation = new Implementation
    {
        UseSingleInstance = useSingleInstance,
        TypeOfImplementation = typeOfImplementation,
        Parameters = parameters ?? new ImplementationParameter[0]
    };
    _registrations.Add(typeOfInterface, implementation);
}
```

Der sker ikke så meget. Først tjekkes om implementationen rent faktisk er en implementation af den afhængighed den registreres for og derefter tilføjes en ny instans af Implementation til Dictonary. Da et array er en reference, kan den være null, hvorfor jeg bruger ?? til at smide et tomt array ind hvis den der kalder metoden giver et null i parameters. En lille bemærkning til dette er at jeg på en måde tillader brugen af null i min kode, hvilket er en ”code smell” og derfor burde jeg måske smide en exception i stedet.

Her er et revideret eksempel på brugen af Register taget fra mit UsageExample project

```
ioc.Register<IDie, Die>(false, new ImplementationParameter { ParameterName = "numberOfSides", Parameter = 10 });
ioc.Register<IDieRoller, DieCup>();
ioc.Register<IRandom, RandomWrapper>();
ioc.Register<IØvelsesLogger, ØvelsesConsoleLogger>();
```

Det absolut sværeste at lave var Resolv metoden. Ideen er at man kalder den generiske version med den afhængighed man gerne vil have ud. Resolv skal så finde implementationen og lave en instans af denne. Dog hvis implementationen selv har afhængigheder skal Resolv kalde sig selv med disse afhængigheder hvilket vil få metoden til recursivt at finde alle afhængigheder indtil disse ikke selv har afhængigheder.

```
private object Resolv(Type type)
{
    var implementation = _registrations[type];
    if (implementation.UseSingleInstance && implementation.InstanceOfImplementation != null)
        return implementation.InstanceOfImplementation;

    var constructor = GetConstructorICanConstruct(type);
    var parameters = ConstructParametersForConstructor(type, constructor).ToArray();
    var instanceOfType = constructor.Invoke(parameters);

    if (implementation.UseSingleInstance) implementation.InstanceOfImplementation = instanceOfType;
    return instanceOfType;
}
```

Man kan ikke lige se at den kalder sig selv, for det sker gennem ConstructParametersForConstructor. Først tjekker metoden om der altid skal returneres samme instans og hvis der skal, har den så allerede instansen får så kan den blot returnere denne. Herefter leder vi efter den første constructor som vi kan

constructe. Beder ConstructParametersForConstructor om at returnere instanser for alle de parameter constructoren kræver og i den rækkefølge de kræves. Derefter kaldes constructoren med parametrene. Dette giver os en instans af vores implementation. Hvis denne implementation altid skal give samme instans, gemmes denne instans og til sidst returneres instansen.

Konklusion

Selv om at min IoC container er meget simpel, synes jeg at den dækker mine specifikationer. Der er absolut plads til forbedringer. En del steder burde jeg nok smide en exception i stedet for at lade noget andet gøre det. Et eksempel kunne være denne linje i Resolv

```
var implementation = _registrations[type];
```

Hvis en bruger at mit API forsøger at resolv en type der ikke finde bliver der smid en KeyNotFoundException, da de ikke ved eller bør vide noget om hvordan jeg implementere min IoC container bør denne exception nok ikke komme ud.

Min IoC container er ikke trådsikker, så den skal kun bruges fra samme tråd. Jeg tjekker heller ikke på om den bliver brugt af multible tråde. Dette vil helt sikkert blive et problem, hvis man ikke er opmærksom på det.

Selv om at MyIoC ikke er så lang, gemmer der sig ganske givet et par klasser der burde blive taget ud af den. Dette ville helt sikkert også give den et mere præcist udtryk og da indholdet i MyIoC er noget langhåret ville dette nok være en ganske god ting at få gjort.

Selv om at jeg har skrevet nogle tests, kunne der godt testes mere og specielt i ”sad path”, vil jeg tro.