Dependency Injection

Loosely Coupled Components

Een van de belangrijkste kenmerken van een goed OO design is "separation of concerns". Ik wil dat de componenten in mijn applicaties zo onafhankelijk mogelijk zijn en zo weinig mogelijk onderlinge afhankelijkheden hebben.

In een ideale situatie weet een component niets over een andere componenten. Ze communiceren alleen via abstracte klassen en interfaces. Dit staat bekend als losse koppeling, en het maakt het testen en aanpassen van toepassingen gemakkelijker.

Als ik een component genaamd MyEmailSender wil maken die e-mails zal versturen, zal ik een interface maken die alle publieke functies definieert die nodig zijn om een e-mail te sturen. Ik zal deze interface IEmailSender noemen.

Elk ander onderdeel van mijn applicatie dat een e-mail moet versturen kan nu gebruik maken van deze interface. In het onderstaande voorbeeld zal de PasswordResetHelper verwijzing naar de methoden in de IEmailSender interface. Er is dus geen rechtstreekse afhankelijkheid tussen PasswordResetHelper en MyEmailSender.

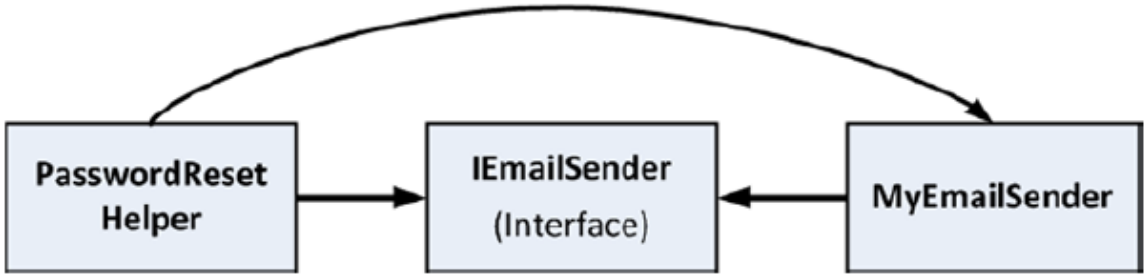
Door de introductie van IEmailSender, zorg ik ervoor dat er geen directe afhankelijkheid tussen PasswordResetHelper en MyEmailSender is. Ik kan MyEmailSender vervangen door een andere e-mail provider zonder wijzigingen in PasswordResetHelper te maken.

Dependency Injection

Interfaces helpen om componenten los te koppelen, maar ik heb nog steeds een probleem: Ik moet gebruik maken van het new keyword zoals weergegeven in onderstaande code:

```
public class PasswordResetHelper
{
 public void ResetPassword()
 {
  IEmailSender mySender = new MyEmailSender();
  //...call interface methods to configure e-mail details...
  mySender.SendEmail();
 }
}
```

Dit ondermijnt mijn doel MyEmailSender te kunnen vervangen zonder PasswordResetHelper te moeten veranderen. De PasswordResetHelper klasse configureert en verzend e-mail via de IEmailSender interface, maar om een object te maken dat deze interface implementeert moet ik een instantie van MyEmailSender creëren. In feite heb ik het nog erger gemaakt voor mezelf omdat PasswordResetHelper nu afhankelijk is van de klasse MyEmailSender en de IEmailSender interface zoals weergegeven in onderstaande afbeelding.



Wat ik nodig heb is een manier om objecten die een interface implementeren te krijgen zonder dat object rechtstreeks aan te maken. De oplossing voor dit probleem is dependency injection (DI), ook bekend als Inversion of Control (IOC). DI is een ontwerp patroon dat zorgt voor een losse koppeling. DI is een zéér belangrijk concept dat centraal staat in professionele ontwikkeling en het kan heel wat verwarring veroorzaken als je dit niet goed onder de knie hebt.

Dependencies Declareren

Het DI patroon bestaat uit 2 onderdelen. Eerst verwijder ik afhankelijkheden op concrete klassen uit mijn component. In dit geval PasswordResetHelper. Ik doe dit door het creëren van een constructor die implementaties aanvaardt van de interfaces ik nodig heb als argumenten.

```
public class PasswordResetHelper
{
 private IEmailSender emailSender;

 public PasswordResetHelper(IEmailSender emailSenderParam)
 {
  emailSender = emailSenderParam;
 }

 public void ResetPassword()
```

```
{
 // ...call interface methods to configure e-mail details...
 emailSender.SendEmail();
 }
}
```

De constructor van de PasswordResetHelper klasse heeft nu een dependency op de IEmailSender interface, wat betekent dat deze niet kan worden gemaakt en gebruikt, tenzij het een object dat de IEmailSender implementeert ontvangt. De PasswordResetHelper klasse heeft niet langer enige kennis van MyEmailSender, ze is alleen afhankelijk van de IEmailSender interface. Kortom, de PassworsResetHelper weet niet meer hoe de IEmailSender-interface wordt geïmplementeerd.

Dependencies Injecteren

Het tweede deel van het DI patroon is het injecteren van de door de PasswordResetHelper klasse aangegeven afhankelijkheden. Dit betekent dat ik moet bepalen welke klasse die de IEmailSender-interface implementeert ik ga gebruiken. Daarna maak ik een instantie van die klasse om ze vervolgens door te geven als argument aan de PasswordResetHelper constructeur.

De afhankelijkheden worden at runtime geïnjecteerd in de PasswordResetHelper. Er is geen compile-time afhankelijkheid tussen PasswordResetHelper en elke klasse die de interface implementeert.

Omdat de afhankelijkheden at runtime worden behandeld, kan ik tijdens het uitvoeren van de applicatie beslissen welke interface implementaties gebruikt worden. Ik kan kiezen tussen verschillende e-mailproviders, zo lang deze maar de IEmailSender interface implementeren.

Dependency Containers

Ik heb mijn afhankelijkheids probleem opgelost, maar hoe kan ik een instantie maken van de concrete implementatie van de interfaces zonder ergens anders in de applicatie afhankelijkheden te creëren? Ik heb nog steeds iets als onderstaande code nodig:

```
...
IEmailSender sender = new MyEmailSender();
helper = new PasswordResetHelper(sender);
...
```

De oplossing is een dependency injection container, ook bekend als IOC container. Dit is een component die bemiddelt tussen de afhankelijkheden die een klasse als PasswordResetHelper heeft en de klassen die kunnen worden gebruikt om deze afhankelijkheden op te vangen, zoals MyEmailSender. Ik registreer de set van interfaces of abstracte klassen waar mijn applicatie gebruik van maakt in de DI container, en geef aan welke klassen moeten worden gebruikt om aan deze afhankelijkheden te voldoen. Ik zal de IEmailSender registreren in de container en aangeven dat een instantie van MyEmailSender moet worden gemaakt wanneer een implementatie van IEmailSender vereist is. Als ik een PasswordResetHelper object nodig heb, vraag ik de DI container een voor mij te maken. Deze weet dat de PasswordResetHelper een afhankelijkheid op de IEmailSender-interface heeft en weet dat ik heb aangegeven dat ik de MyEmailSender klasse wil gebruiken als implementatie van die interface. De DI container zet deze twee stukken informatie bij elkaar, maakt een MyEmailSender instantie en gebruikt het als een argument voor de PasswordResetHelper constructor.

Er zijn veel open source implementaties van DI containers beschikbaar. Degene die wij zullen gebruiken is Ninject.

Dependency Injection By Hand

Laten we eens kijken naar dependency injection aan de hand van een eenvoudig voorbeeld. Laten we de basis leggen voor een spel waar nobele krijgers tegen elkaar zullen strijden. Om te beginnen hebben we een wapen nodig om onze strijders te bewapenen.

```
class Sword
{
 public void Hit(string target)
 {
  Console.WriteLine("Chopped {0} clean in half", target);
 }
}
```

Nu creëren wa een klasse voor onze strijders. Om zijn vijanden aan te vallen, heeft de krijger een methode Attack() nodig. Wanneer deze methode wordt angeroepen, moet hij de Sword klasse gebruiken om zijn tegenstander te raken.

```
class Samurai
{
 readonly Sword sword;

 public Samurai()
 {
  this.sword = new Sword();
 }

 public void Attack(string target)
 {
  this.sword.Hit(target);
 }
}
```

Nu kunnen we onze Samurai creëren en ten strijde trekken!

```
class Program
{
 public static void Main()
 {
  var warrior = new Samurai();
  warrior.Attack("the evildoers");
 }
}
```

Dit werkt prima, maar wat als we onze Samurai willen bewapenen met een ander wapen? Aangezien het zwaard wordt gemaakt in de constructor van de Samurai klasse moeten we de implementatie van de klasse aanpassen om deze wijziging door te voeren.

In dit voorbeeld is de Samurai klasse tightly coupled met de Sword klasse. Om sterke koppeling tussen klassen te voorkomen kunnen we interfaces gebruiken. Laten we een interface creëren om een wapen in ons spel te vertegenwoordigen.

```
interface IWeapon {  void Hit(string target); }
```

Dan kan onze Sword klasse deze interface implementeren:

```
class Sword : IWeapon
{
 public void Hit(string target)
 {
  Console.WriteLine("Chopped {0} clean in half", target);
 }
}
```

En we kunnen onze Samurai klasse veranderen:

```
class Samurai
{
 readonly IWeapon weapon;

 public Samurai()
 {
  this.weapon = new Sword();
 }

 public void Attack(string target)
 {
  this.weapon.Hit(target);
 }
}
```

Nu kan onze Samurai met verschillende wapens bewapend worden. Maar wacht! Het zwaard is nog steeds gemaakt in de constructor van Samurai. Samurai is dus nog steeds nauw verbonden met Sword.

Gelukkig is er een eenvoudige oplossing. In plaats de Sword klasse in de constructeur van Samurai aan te maken kunnen we deze meegeven als parameter van de constructor.

```
class Samurai
{
 readonly IWeapon weapon;

 public Samurai(IWeapon weapon)
 {
  this.weapon = weapon;
```

```
}

public void Attack(string target)
{
 this.weapon.Hit(target);
}
}
```

Nu kunnen we het zwaard via de constructor van Samurai injecteren. Dit is een voorbeeld van dependency injection (in het bijzonder, constructor injectie). Laten we een ander wapen creëren dat onze Samurai zou kunnen gebruiken:

```
class Shuriken : IWeapon
{
 public void Hit(string target)
 {
  Console.WriteLine("Pierced {0}'s armor", target);
 }
}
```

Nu kunnen we een heel leger aanmaken

```
class Program
{
 public static void Main()
 {
  var warrior1 = new Samurai(new Shuriken());
  var warrior2 = new Samurai(new Sword());
  warrior1.Attack("the evildoers");
  warrior2.Attack("the evildoers");
```

```
}
}
```

Dit geeft onderstaande output:
"Pierced the evildoers armor."
"Chopped the evildoers clean in half."

Dit is dependency injection by hand. Elke keer dat je een Samurai wil maken, moet je eerst een implementatie van IWeapon aanmaken en vervolgens doorgeven aan de constructor van Samurai. We kunnen nu de Samurai met andere wapens laten strijden zonder dat we de code van deze klasse moeten aapassen.

Dependency Injection met Ninject

In plaats van tijd te spenderen aan het creëren en het connecteren van objecten met de hand kunnen we beter een dependency container als Ninject gebruiken om dit werk voor ons te doen.

Hoe Ninject instanties maakt voor u

Ninject roept een van de constructos van je type op, net zoals je dit zou doen met Dependency Injection by hand. Ninject kijken naar de beschikbare publieke constructors en kiest degene met de meeste parameters die hij kan resolven.

Hoe Ninject beslist welk type te gebruiken

Het enige dat je hoeft te doen Ninject laten weten welke concrete klasse gebruikt om de interface te voldoen. We bekijken het volgende voorbeeld

```
IKernel kernel = new StandardKernel();
var samurai = kernel.Get<Samurai>();
```

In het bovenstaande geval zal Ninject de enige constructor die gedefinieerd is gebruiken. Vervolgens probeert Ninject een instantie te resolven per argumenten de constructeur. In dit geval is er slechts één argument te resolven: IWeapon. Maar wacht eens even ... IWeapon is een interface. Je kan geen instantie van een interface maken, dus hoe weet Ninject welke implementatie van IWeapon te injecteren? Dit wordt opgelost door het type bindings. Een type binding is een mapping tussen een servicetype (een interface of abstracte klasse) en een implementatietype. Bindings worden meestal uitgedrukt via fluten interface van Ninject. om onze Samurai te bewapenen met een zwaard, kunnen we volgende bindend maken:

```
Bind<IWeapon>().To<Sword>();
```

Deze binding geeft aan dat wanneer Ninject een afhankelijkheid van IWeapon moet ophalen, het Sword zal injecteren. Dit proces is recursief. Dit wil zeggen dat indien Sword zelf afhankelijkheden heeft, de ook worden geresolved voordat de construtor van Samurai wordt aangeroepen. Op deze manier kan Ninject een hele ketting van dendencies ophalen. We noemen dit een Dependency Chain.