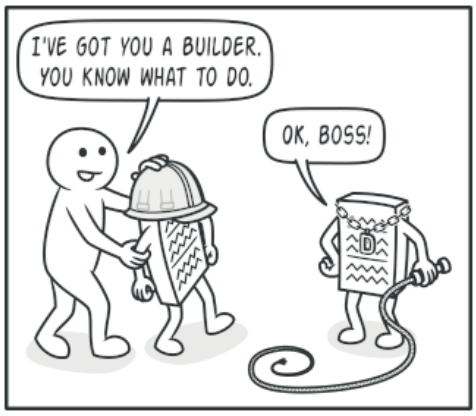
Builder.md

Let's Build(er)

We kunnen het niet genoeg herhalen: ook dit patroon maakt gebruik van interfaces. Interface laat toe heel wat knappe software the schrijven!

Het Builder Design Pattern is een 'Creation' pattern. Dat wil zeggen dat het een *best practice* is voor het aanmaken van objecten. Het Builder pattern is vooral geschikt voor het creeeren van gecompliceerde objecten. In deze walkthrough maken we een *maaltijd Builder*. We beginnen deze workshop simpel met een eenvoudige .NET Core Console applicatie.



Je kan in Visual Studio 2019 kiezen voor File, New, Project, Visual C#, .NET Core en dan Console Application .NET Core of ...

1: Maak op een locatie naar keuze een folder aan en noem deze 'Builder'.

2: Open Command Prompt (Windows Search -> 'cmd' -> enter) en navigeer naar de betreffende folder.

3: Type nu onderstaande code en druk vervolgens op enter om een .NET Core Console applicatie aan te maken.

```
dotnet new
```

4: Indien je voorheen niet Visual Studio hebt gebruikt maar Visual Studio Code met de Command Prompt dan dien je nu eerst een restore van de packages toe doen middels:

```
dotnet restore
```

Het basis Console project staat nu. Dus laten we beginnen met het Builder Pattern. De maaltijd die we gaan 'builden' bestaat uit items (zoals hamburgers en cola) die natuurlijk moet worden ingepakt in een verpakking (bijvoorbeeld een flesje of een doosje).

5: We beginnen met het definieren van de Interfaces. Maak in het project een nieuwe folder 'Interfaces' aan.

6: Maak een nieuwe Interface class aan en noem deze *IPacking.cs*

7: De *IPacking* interface bevat een enkele `string` property met de naam 'Pack' en een public 'getter'.

```
namespace Builder.Interfaces
{
    public interface IPacking
    {
        string Pack { get; }
    }
}
```

Het *IPacking.cs* bestand mag je nu weer sluiten.

8: Creëer in dezelfde interface folder een nieuwe Interface class en noem deze 'IItem.cs'.

9: In de `public interface IItem` schrijven we een drietal properties met een public 'getter'. * `string Name { get; }` * `IPacking Packing { get; }` * `float Price { get; }`

```
namespace Builder.Interfaces
{
    public interface IItem
    {
        string Name { get; }

        IPacking Packing { get; }

        float Price { get; }
    }
}
```

We gaan nu de Interface concreet maken.

10: Maak in de root van het project een nieuwe folder aan en noem deze 'Models'.

11: Maak in de 'Models' folder een nieuwe class aan en noem deze 'Bottle.cs'.

12: De 'Bottle.cs' class implementeert de `IPacking` interface. Hiervoor dien je nog wel even naar de 'Interfaces' folder te refereren.

Pro-tip: Wanneer je de interface benoemt in je class, krijg je standaard een rode kringel onder de interface naam. Ga met je muis op de rode kringel staan en druk vervolgens op `CTRL + .` (Control + punt), Selecteer in het context menu dat verschijnt: 'Implement Interface' en het meeste werk wordt direct voor je gedaan.

13: Implementeer de 'Pack' property en laat deze de tekst 'Bottle' retourneren.

```
using Builder.Interfaces;

namespace Builder.Models
{
    public class Bottle : IPacking
    {
        public string Pack
        {
            get { return "Bottle"; }
        }
    }
}
```

14: Sluit nu het 'Bottle.cs' bestand.

15: Dezelfde stappen (11 t/m 14) herhalen we voor een 'Box'. Het nieuwe bestand noem je nu dus 'Box.cs' en de IPacking interface implementeert 'Pack' welke de tekst 'Box' retourneert.

```
using Builder.Interfaces;

namespace Builder.Models
{
    public class Box : IPacking
    {
        public string Pack
        {
            get { return "Box"; }
        }
    }
}
```

Nu we een `Box` en een `Bottle` hebben kunnen we ons voedsel ergens in bewaren. Tijd voor het maken van de *voedsel* classes.

16: We beginnen met het maken van een nieuwe `abstract` class met de naam Burger.cs, nog steeds in de 'Models' folder.

17: Burger implementeert de `IItem` interface.

18: De `Name` en `Price` properties implementeren we hier nog niet, dat doen we in de burger implementaties, dus zetten we ze door als `abstract` properties met een `public get`.

19: De `Packing` property laten we een `new Box();` retourneren in de public `get`.

De Burger.cs code zou er ongeveer zou uit moeten zien:

```
using Builder.Interfaces;

namespace Builder.Models
{
    public abstract class Burger : IItem
    {
        public abstract string Name { get; }

        public IPacking Packing
        {
            get { return new Box(); }
        }

        public abstract float Price { get; }
    }
}
```

Het bestand 'Burger.cs' is nu gereed en mag je sluiten.

We maken nu een tweetal typen 'Burgers' aan. Namelijk een Hamburger en een ChickenBurger.

20: Nog steeds in de 'Models' folder: Maak een nieuwe class aan met de naam 'Hamburger'.

21: `Hamburger` implementeert (logischerwijs) de `Burger` class die we eerder hebben aangemaakt.

22: Implementeer de 'overgebleven' properties `Name` en `Price`. (CTRL + .).

23: Zorg er voor dat `Name` de waarde "Hamburger" retourneert in de getter.

24: Voor `Price` mag je `2.80f` retourneren (nee, de `f` staat niet voor de ouderwetsche Neerlandische Frank maar voor `float`)

Een voorbeeld van de `Hamburger` class met de sexy nieuwe alternatieve manier van de 'get return' implementaties.

```
namespace Builder.Models
{
    public class Hamburger : Burger
    {
        public override string Name => "Hamburger";

        public override float Price => 2.80f;
    }
}
```

25: Tijd voor de `ChickenBurger` maak in de 'Models' folder het bestand 'ChickenBurger.cs' aan.

26: Herhaal de bovenstaande stappen 21 t/m 24 waarbij je in dit geval `Hamburger` vervangt voor `ChickenBurger` en pas je de prijs aan naar `2.50f`.

De `ChickenBurger` class:

```
namespace Builder.Models
{
   public class ChickenBurger : Burger
   {
       public override string Name => "ChickenBurger";

       public override float Price => 2.50f;
   }
}
```

Bij ons diner hebben we ook wat drinken nodig. Gelukkig hebben we al een 'Bottle' om mee te beginnen.

27: Maak in de 'Models' folder een nieuwe class aan en noem deze 'Drink'.

28: `Drink` implementeert `IItem` en logischerwijs de bijbehorende properties. `Name` en `Price` definieer je `abstract` die zetten we pas in de concrete implementatie van de `Drink` class.

```
        public abstract string Name { get; }
        public abstract float Price { get; }
```

29: De `Packing` property dient een `IPacking` implementatie terug te geven. Hiervoor hebben we eerder een `Bottle` class aangemaakt.

```
        public IPacking Packing => new Bottle();
```

Dat was het voor de `Drink` class. Door naar de concrete implementatie van de drankjes.

30: Maak, wederom in de 'Models' folder, een nieuwe class aan genaamd `Cola` .

31: Je raadt het al, `Cola` erft over van `Drink` en dient dus ook de bijbehorende properties te 'overriden' en in te vullen. * `Name` retourneert `"Cola"` * `Price` retourneert `1.50f`

```
namespace Builder.Models
{
    public class Cola : Drink
    {
        public override string Name => "Cola";
        public override float Price => 1.50f;
    }
}
```

32: Hetzelfde trucje herhalen we voor `Water` . Een nieuwe class in de 'Models' folder. * `Name` retourneert `"Water"` * `Price` retourneert `1f`

```
namespace Builder.Models
{
    public class Water : Drink
    {
        public override string Name => "Water";
        public override float Price => 1f;
    }
}
```

Nu we al de losse items hebben kunnen we eindelijk een maaltijd bereiden. Eerst moeten we alleen nog even definieren wat een maaltijd precies is.

33: Maak in de, 'Models' folder een nieuwe class aan genaamd 'Meal.cs'.

34: Een maaltijd bestaat uit meerdere componenten. Hiervoor schrijven we een `private` property genaamd 'items' welke bestaat uit een `List'. Instantieer deze ook direct.

```
using Builder.Interfaces;
using System.Collections.Generic;

namespace Builder.Models
{
    public class Meal
    {
        private List<IItem> items = new List<IItem>();
    }
}
```

35: Aan deze lijst willen we natuurlijk items toevoegen dit doen we door een `public` functie 'AddItem' te maken die een enkele `IItem` als parameter ontvangt.

36: De `IItem` parameter stoppen we vervolgens in de 'items' lijst.

```
        public void AddItem(IItem itemToAdd)
        {
            items.Add(itemToAdd);
        }
```

37: Van de maaltijd willen we ook de totaal prijs weten. Schrijf een functie `GetCosts()` welke een `float` retourneert.

38: In de `GetCosts` functie tel je de prijs van al de items in de lijst bij elkaar op en retourneer je het resultaat. Lang leve `Linq`, dit kunnen we in een enkele regel schrijven:

```
        public float GetCosts()
        {
            return items.Sum(i => i.Price);
        }
```

39: Om de 'Meal' class af te ronden schrijven we nog een laatste functie die voor elk `item` in de lijst laat zien wat de naam, de verpakking en de prijs is.

De uiteindelijke `Meal` class zou er nu ongeveer zo uit moeten zien:

```
using Builder.Interfaces;
using System;
using System.Collections.Generic;
using System.Linq;

namespace Builder.Models
{
    public class Meal
    {
        private List<IItem> items = new List<IItem>();

        public void AddItem(IItem itemToAdd)
        {
            items.Add(itemToAdd);
        }

        public float GetCosts()
        {
            return items.Sum(i => i.Price);
        }

        public void ShowItems()
        {
            foreach (var item in items)
            {
                Console.WriteLine("Item: {0}, Packing: {1}, Price {2}", item.Name, item.Packing.Pack, item.Price.ToString("C"));
            }
        }
    }
}
```

Het echte werk...

We kijken naar het Builder Pattern.

40: In de root van de applicatie maak je een nieuwe class genaamd 'MealBuilder'. Deze doet eigenlijk precies wat de naam zegt, hij bouwt de maaltijd op.

41: In de `MealBuilder` class hebben we een tweetal functies: `PrepareChickenMeal()` en `PrepareBurgerMeal()`. Beiden retourneren, je hebt het wederom correct, een `Meal`.

42: We beginnen in de `PrepareChickenMeal` functie. Hierin maken we een nieuwe instance van de de `Meal` class aan. Hiervoor dien je wel nog een `using` te leggen naar de 'Models' folder.

43: Vervolgens voegen we aan de `meal` een drankje (`Water`) en een hapje (`ChickenBurger`) toe middels de `AddItem()` functie die we eerder hebben geschreven.

44: Logischerwijs retourneren we de gehele maaltijd (`meal`) onderin de `PrepareChickenMeal` functie.

45: Dezelfde stappen herhalen we voor de `PrepareBurgerMeal` functie. Deze bestaat uit een `Cola` en een `Hamburger` .

De `MealBuilder` class is gereed als deze er ongeveer zo uit ziet:

```
using Builder.Models;

namespace Builder
{
    public class MealBuilder
    {
        public Meal PrepareChickenMeal()
        {
            var meal = new Meal();
            meal.AddItem(new Water());
            meal.AddItem(new ChickenBurger());

            return meal;
        }

        public Meal PrepareBurgerMeal()
        {
            var meal = new Meal();
            meal.AddItem(new Hamburger());
            meal.AddItem(new Cola());

            return meal;
        }

    }
}
```

46: Alle openstaande bestanden mag je nu afsluiten voor het laatste deel van deze Builder Pattern workshop. We gaan voor deze laatste stap terug naar onze `Main` functie in de `Program` class.

47: Bovenin de `Main` functie instantieren we de `MealBuilder` met de naam 'builder'.

48: Onderin de `Main` functie schrijven we een regel die voorkomt dat de applicatie zometeen direct afsluit wanneer je hem runt. Dit doe je met `Console.ReadLine();`

49: Tussen de `ReadLine()` functie en de `builder` property maken een hamburger maaltijd aan:

```
    var burgerMeal = builder.PrepareBurgerMeal();
```

50: Uiteraard zijn we benieuwd wat er dan zoal in deze `burgerMeal` zit en wat de totaalprijs is:

```
    Console.WriteLine("BurgerMeal: ");
    burgerMeal.ShowItems();
    Console.WriteLine("Costs: {0}", burgerMeal.GetCosts().ToString("C"));
```

51: De laatste maaltijd spreekt voor zich. Tijd voor de `ChickenMeal` implementatie.

```
    var chickenMeal = builder.PrepareChickenMeal();
    Console.WriteLine("ChickenMeal: ");
    chickenMeal.ShowItems();
    Console.WriteLine("Costs: {0}", chickenMeal.GetCosts().ToString("C"));
```

Het uiteindelijke Program bestand is niets meer dan:

```
using System;

namespace Builder
{
    public class Program
    {
```

```
        public static void Main(string[] args)
        {
            var builder = new MealBuilder();

            var burgerMeal = builder.PrepareBurgerMeal();
            Console.WriteLine("BurgerMeal: ");
            burgerMeal.ShowItems();
            Console.WriteLine("Costs: {0}", burgerMeal.GetCosts().ToString("C"));

            var chickenMeal = builder.PrepareChickenMeal();
            Console.WriteLine("ChickenMeal: ");
            chickenMeal.ShowItems();
            Console.WriteLine("Costs: {0}", chickenMeal.GetCosts().ToString("C"));

            Console.ReadLine();
        }
    }
}
```

52: Start nu de applicatie door op 'F5' te drukken in Visual Studio of door onderstaande functie te typen in Command Prompt

```
dotnet run
```

Wat we zien is dat een Builder eigenlijk niet meer is dan een soort van functie die het genereren van sub-items voor zijn rekening neemt. Je doet een enkele call naar de builder implementatie, in dit geval `PrepareBurgerMeal` of `PrepareChickenMeal` en de builder zorgt er voor dat alle onderdelen bij elkaar in het pakketje worden gedaan. Eigenlijk net als de kassamadam of -mijnheer bij McDonalds!