

## OO basics

### Bierwinkel

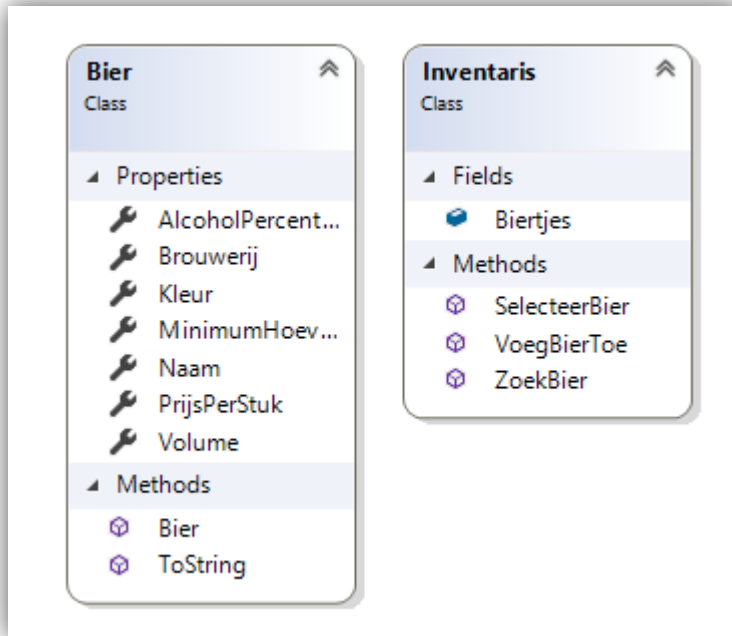
Onze klant “Johnny” heeft een bierwinkel en om zijn eigen klanten zo goed mogelijk te helpen roept hij onze hulp in. We gaan dus software schrijven voor Johnny’s bierwinkel. In deze winkel bieden we een aantal soorten bier aan die de klant kan selecteren aan de hand van een aantal kenmerken; zo kan de klant een bier selecteren op basis van zijn kleur (blond, amber en bruin), de brouwerij vanwaar het bier afkomstig is, de inhoudsmaat (25 cl, 33cl, ...) en het alcoholpercentage.

Elk bier heeft natuurlijk ook een unieke naam en – het zou anders geen winkel zijn – een prijs per stuk natuurlijk. Sommige bieren verkopen we apart, andere in setjes van 4, 6 of 8. En voor een aantal is de levering per 12 of 24.



Wanneer een klant naar de winkel komt moeten we in staat zijn om het juiste bier te vinden voor deze klant. We hebben dus een inventaris nodig van welke bieren er te koop zijn in onze winkel.

We gaan aan de slag en ons eerste ontwerp ziet er als volgt uit :



We hebben een klasse `Bier` met al de eigenschappen die zijn gevraagd en we hebben ook een `Inventaris` klasse waar we een lijst van bieren bijhouden. Er kunnen nieuwe bieren toegevoegd worden – we kijken namelijk vooruit en voorzien dat we ons assortiment gaan uitbreiden, en we voorzien ook dat de klant een bier kan opzoeken op basis van een aantal kenmerken.

Onze klasse bier ziet er dan als volgt uit :

```
public class Bier
{
    2 references
    public Bier(double prijsPerStuk, string naam, string kleur, string brouwerij,
        double volume, double alcoholPercentage, int minimumHoeveelheid)
    {
        PrijsPerStuk = prijsPerStuk;
        Naam = naam;
        Kleur = kleur;
        Brouwerij = brouwerij;
        Volume = volume;
        AlcoholPercentage = alcoholPercentage;
        MinimumHoeveelheid = minimumHoeveelheid;
    }
    1 reference
    public double PrijsPerStuk { get; set; }
    2 references
    public string Naam { get; set; }
    4 references
    public string Kleur { get; set; }
    5 references
    public string Brouwerij { get; set; }
    4 references
    public double Volume { get; set; }
    4 references
    public double AlcoholPercentage { get; set; }
    1 reference
    public int MinimumHoeveelheid { get; set; }
}
```

We hebben een constructor waar we alle parameters instellen en voor elk kenmerk van ons bier hebben we een property voorzien.

De inventaris klasse hebben we als volgt geschreven :

```
public class Inventaris
{
    public List<Bier> Biertjes = new List<Bier>();
    4 references
    public void VoegBierToe(double prijsPerStuk, string naam, string kleur, string brouwerij,
        double volume, double alcoholPercentage, int minimumHoeveelheid)
    {
        Bier bier = new Bier(prijsPerStuk, naam, kleur, brouwerij, volume, alcoholPercentage, minimumHoeveelheid);
        Biertjes.Add(bier);
    }
    1 reference
    public Bier SelecteerBier(string naam) {...}
    1 reference
    public Bier ZoekBier(Bier bier) {...}
}
```

We voorzien een List voor de bieren bij te houden. Wanneer we een bier wensen toe te voegen aan onze lijst, maken we op basis van de meegegeven waarden een nieuwe instantie bier aan en voegen deze toe aan de lijst.

Voor het selecteren van een bier op basis van zijn naam, hebben we de volgende eenvoudige methode gemaakt. We doorlopen onze lijst en wanneer we een bier vinden met de naam die we zoeken geven we dit resultaat terug.

```
public Bier SelecteerBier(string naam)
{
    foreach(Bier bier in Biertjes)
    {
        if (bier.Naam == naam) return bier;
    }
    return null;
}
```

Het opzoeken van een bier is een stuk ingewikkelder en is te zien in onderstaand codefragment.

```
public Bier ZoekBier(Bier bier)
{
    foreach (Bier b in Biertjes)
    {
        if (bier.Kleur.Length > 0 && bier.Kleur != b.Kleur) continue;
        if (bier.Brouwerij != null && bier.Brouwerij.Length > 0 && bier.Brouwerij != b.Brouwerij) continue;
        if (bier.Volume > 0 && bier.Volume != b.Volume) continue;
        if (bier.AlcoholPercentage >= 0 && bier.AlcoholPercentage != b.AlcoholPercentage) continue;
        return b;
    }
    return null;
}
```

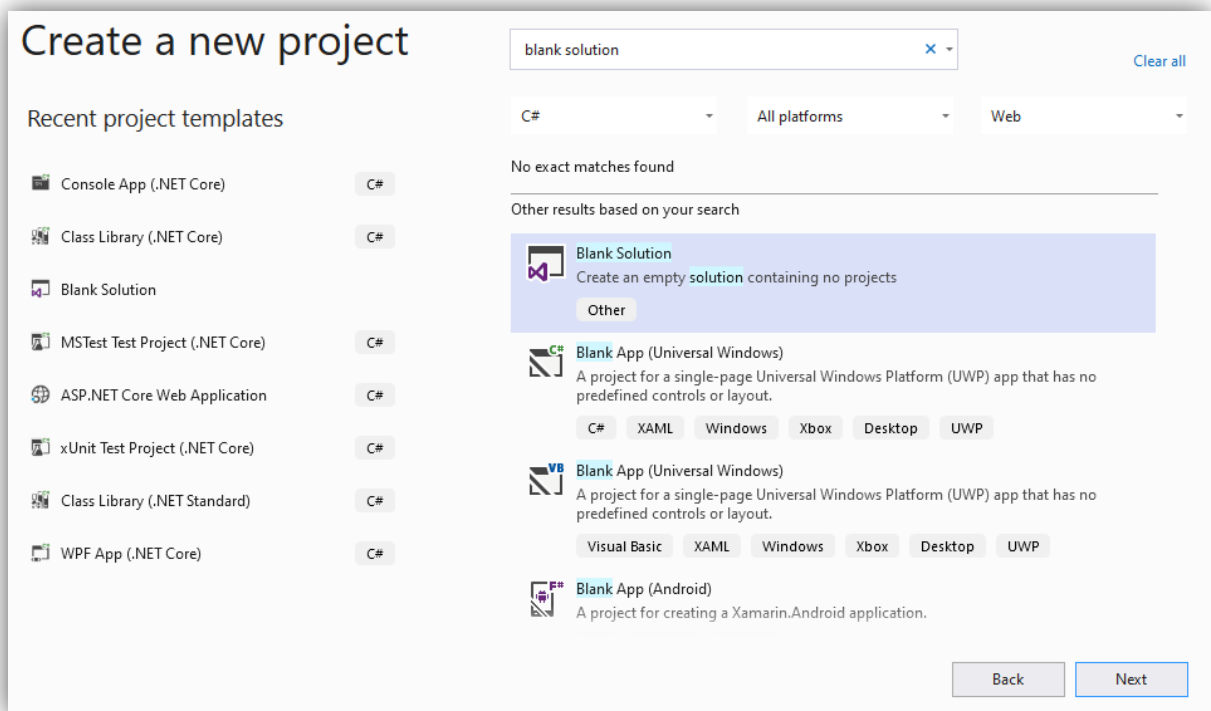
Ook hier doorlopen we de lijst van beschikbare bieren en kijken of dit bier voldoet aan de gestelde zoekvoorwaarden. We controleren telkens eerst of de parameter is ingevuld, als er bijvoorbeeld geen kleur is meegegeven (lengte is 0) dan controleren we dit ook niet. Van zodra een bier voldoet aan de gevraagde eigenschappen geven we het resultaat terug.

Dan wordt het nu tijd om onze software eens uit te proberen !

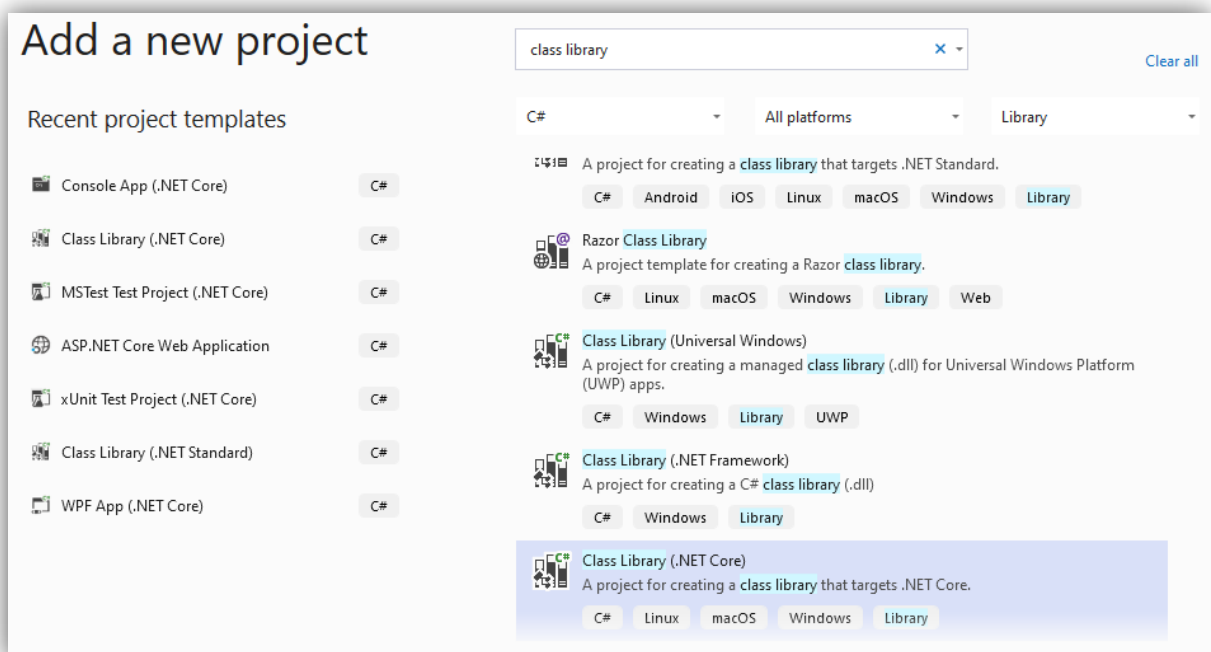
Eerste test run.

Maar daarvoor gaan we eerst kort beschrijven op welke manier we gebruik gaan maken van Visual Studio en hoe we onze code gaan organiseren. In de eerste plaats maken we onderscheid tussen de klassen die de business-regels implementeren en de applicatie die we gaan gebruiken om deze uit te voeren.

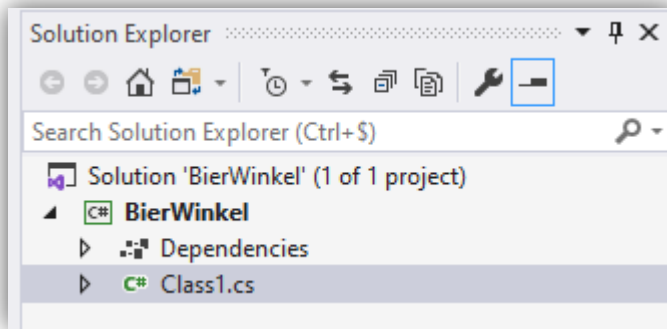
We starten met het aanmaken van een 'Blank solution' in Visual Studio. We starten VS op en maken een nieuwe 'solution' aan.



Daarna voegen we een project toe aan de solution en kiezen voor een “class library (.NET Core)”.

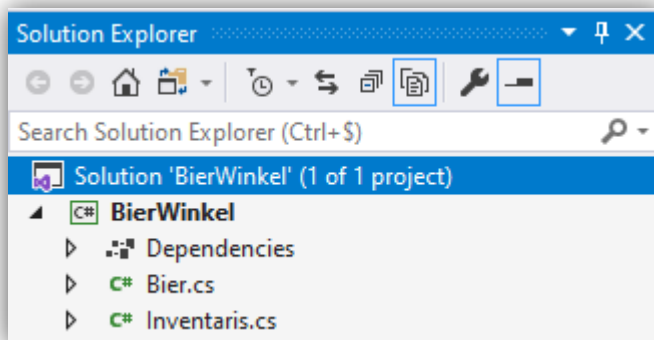


In de solution explorer van VS ziet het er dan als volgt uit :

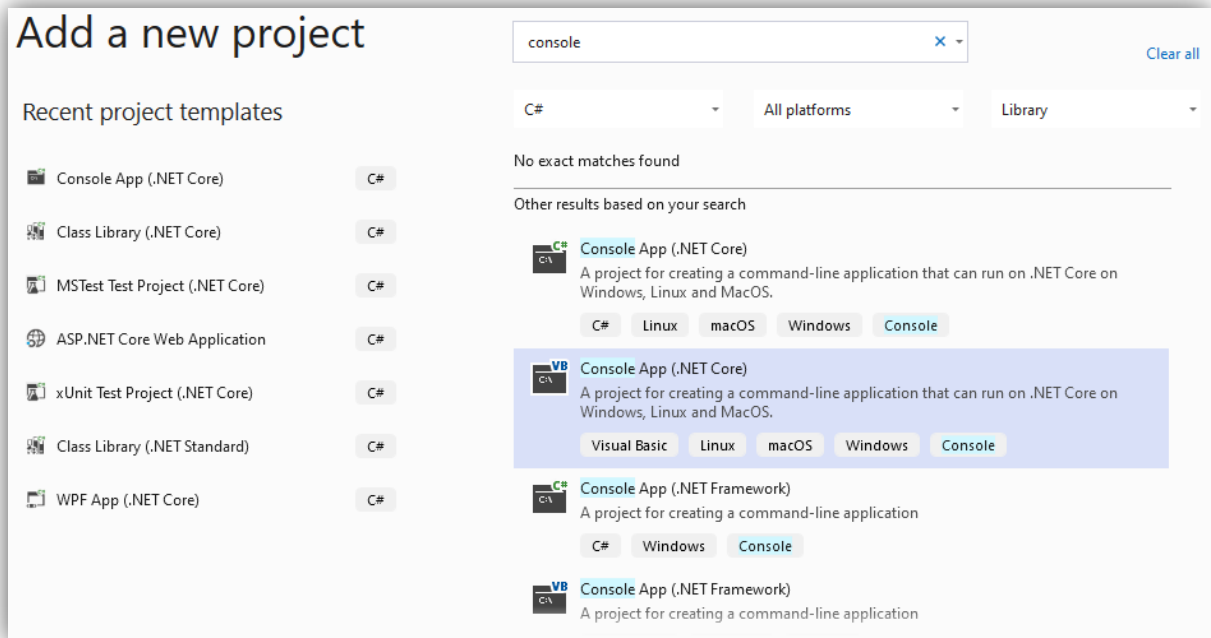


Standaard wordt er een Class1.cs bestand aangemaakt, maar dat gaan we niet gebruiken dus deze klasse mag worden verwijderd.

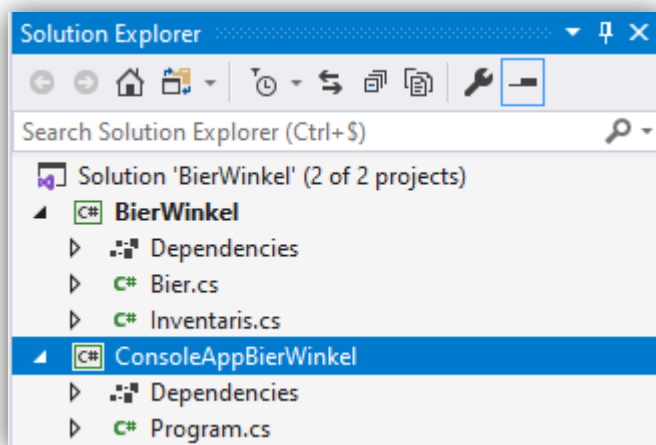
De volgende stap is het aanmaken van onze klassen Bier en Inventaris in het project BierWinkel. De code voor deze klassen hebben we reeds meegegeven. Onze solution ziet er nu als volgt uit :



De volgende stap is nu het maken van een applicatie zodat we onze klassen kunnen uitproberen. Dat doen we door een nieuw project aan te maken die gebruik zal maken van de klassen uit onze bibliotheek (class library). We kiezen voorlopig voor een console applicatie (later zullen we opteren om een mooiere user interface te maken door gebruik te maken van WPF – Windows Presentation Foundation).



Het resultaat is dan een solution met daarin 2 projecten (onze class library met de business logica en een applicatie – console app – die er gebruik van zal maken).



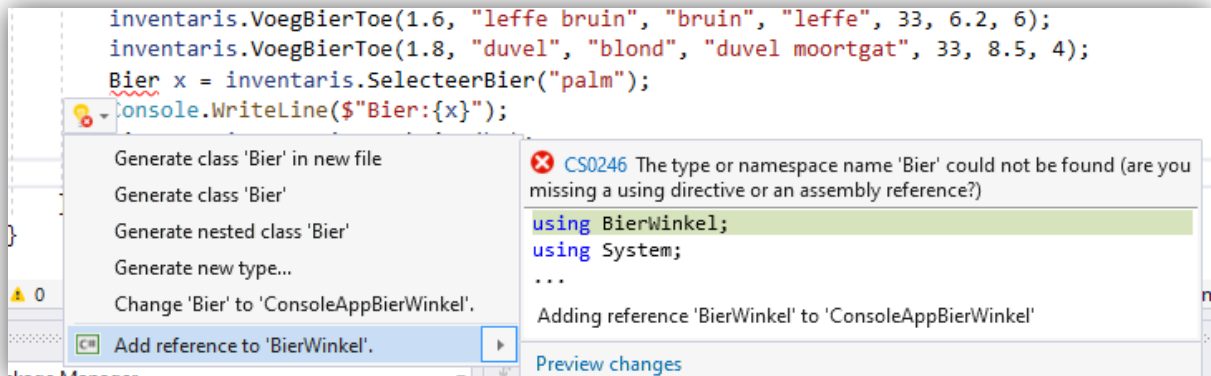
Om onze applicatie uit te testen voeren we een aantal biertjes toe aan onze inventaris. Een beperkt aanbod voorlopig, maar zeker niet duur !

```

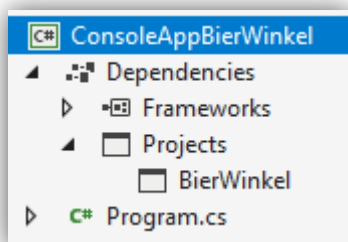
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
        Bier b1 = new Bier(1.05, "palm", "Amber", "palm", 25, 5.2, 6);
        Inventaris inventaris = new Inventaris();
        inventaris.VoegBierToe(1.05, "palm", "amber", "palm", 25, 5.2, 6);
        inventaris.VoegBierToe(1.25, "rodenbach classic", "bruin", "palm", 25, 5.2, 6);
        inventaris.VoegBierToe(1.6, "leffe bruin", "bruin", "leffe", 33, 6.2, 6);
        inventaris.VoegBierToe(1.8, "duvel", "blond", "duvel moortgat", 33, 8.5, 4);
        Bier x = inventaris.SelecteerBier("palm");
        Console.WriteLine($"Bier:{x}");
        Bier y = inventaris.ZoekBier(b1);
        Console.WriteLine($"Bier:{y}");
    }
}

```

We testen ook de SelecteerBier en ZoekBier methodes uit van de inventaris-klasse. Maar vooraleer we dat kunnen doen, moeten we eerst nog een probleempje oplossen. Onze applicatie kent namelijk de klassen Bier en Inventaris niet (aangegeven met een rode lijn onder de types). VS maakt het ons echter gemakkelijk en suggereert een aantal oplossingen. We kiezen voor het toevoegen van een referentie naar onze class library (BierWinkel) waar de klassen zijn in ondergebracht.



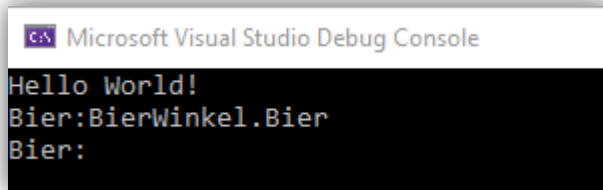
Na het toevoegen van de referentie zien we in de solution explorer bij de dependencies van de console app onze link staan naar de BierWinkel (bij projects).



Kunnen we eindelijk een eerste test run uitvoeren !



Voeren we de applicatie uit dan verschijnt het volgende in de output console :

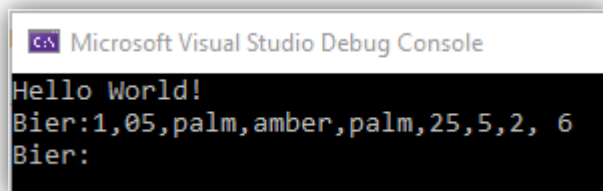


```
Microsoft Visual Studio Debug Console
Hello World!
Bier:BierWinkel.Bier
Bier:
```

Niet echt wat we in gedachte hadden. We zouden in onze Console.WriteLine methode elke property van Bier apart kunnen meegeven, maar dan moeten we dat telkens als er iets verandert aan onze klasse Bier onze applicatie aanpassen en dat willen we vermijden. Het tonen (afdrukken) van de inhoud van de klasse Bier willen we overlaten aan de klasse Bier zelf (-> voorbeeld van goed ontwerp). Daarom voegen we het volgende stukje code toe in de klasse Bier.

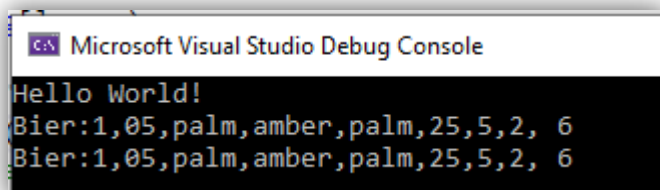
```
public override string ToString()
{
    return $"{PrijsPerStuk},{Naam},{Kleur},{Brouwerij},{Volume},{AlcoholPercentage}, {MinimumHoeveelheid}";
}
```

Uitleg hoe dit precies werkt leggen we uit in één van de volgende hoofdstukken, voorlopig is het voldoende om dit gewoon toe te passen. Het resultaat wordt nu :



```
Microsoft Visual Studio Debug Console
Hello World!
Bier:1,05,palm,amber,palm,25,5,2, 6
Bier:
```

Nog niet helemaal wat we willen, want onze zoekmethode geeft geen resultaat terug. Op die manier zal “Johnny” niet tevreden zijn natuurlijk. Kijken we nog eens goed naar de code in onze console app en dan merken we dat we voor het zoeken “Amber” met een hoofdletter hebben geschreven en in onze inventaris zit dit zonder hoofdletter. Dit kunnen we eenvoudig oplossen door in onze zoekmethode alle strings om te zetten naar lower case. Probleem opgelost, het resultaat is nu :



```
Microsoft Visual Studio Debug Console
Hello World!
Bier:1,05,palm,amber,palm,25,5,2, 6
Bier:1,05,palm,amber,palm,25,5,2, 6
```

Eindelijk kunnen we gaan verkopen.

## Great Software

Onze applicatie werkt ! Geweldige software !

Wait a second... I hate to butt in, but what does "great software" mean? That's sort of a vague term to be throwing around, isn't it?

**Good question... and there are lots of different answers:**

**The customer-friendly programmer says:**

"Great software always does what the customer wants it to. So even if customers think of new ways to use the software, it doesn't break or give them unexpected results."

**The object-oriented programmer says:**

"Great software is code that is object-oriented. So there's not a bunch of duplicate code, and each object pretty much controls its own behavior. It's also easy to extend because your design is really solid and flexible."

**The design-guru programmer says:**

"Great software is when you use tried-and-true design patterns and principles. You've kept your objects loosely coupled, and your code open for extension but closed for modification. That also helps make the code more reusable, so you don't have to rework everything to use parts of your application over and over again."

*Handwritten notes:*

- This approach is all about making sure the customer is happy with what their app does.
- This design-focused approach optimizes code for extension and reuse, and takes advantages of design patterns and proven OO techniques.
- Good OO programmers are always looking for ways to make their code more flexible.
- Not sure about what all that means? It's OK- you'll learn about all these things in the upcoming chapters.

(screenshot uit *Head First Object-Oriented Analysis & Design*, Brett D. McLaughlin, Garry Pollice & David West – Uitgeverij O'Reilly)

**Aan welke voorwaarden moet goede software nu voldoen ?**

We stellen hierbij drie eenvoudige stappen voor die we zullen volgen.

Stap 1 : zorg ervoor dat de software doet wat de klant wenst dat die doet.

Stap 2 : pas basis OO (object oriëntatie) principes toe voor een flexibele oplossing.

Stap 3 : streef naar een onderhoudbaar en herbruikbaar ontwerp.

### En hoever staan we met ons huidig ontwerp ? Stap 1 is afgerond ?

Of toch niet helemaal. Onze zoekmethode stopt wanneer er één bier is gevonden dat voldoet aan de voorwaarden. Het zou natuurlijk beter zijn als “Johnny” zijn klanten een lijst van bieren kan aanbieden die voldoen, betere service en wellicht meer verkoop. Dus dat gaan we moeten aanpassen.

### Wat kunnen we nog verbeteren ?

Wel, we hebben gezien dat onze eerste versie van de zoekmethode afhankelijk was van het gebruik van hoofdletters. Maar er zijn nog meer valkuilen aanwezig, het gebruik van strings voor de kleur, doubles voor volume en int voor het minimum aantal zijn potentiële bronnen van foute invoer. Die zouden we echter kunnen oplossen door het gebruik van vaste waarden, in c# zijn dat enums.

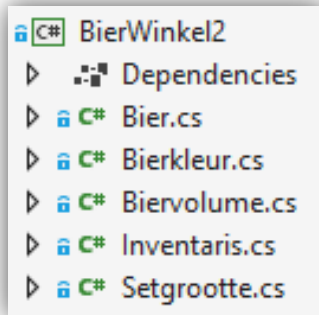
We voeren dus de volgende enums toe aan ons project.

```
public enum Bierkleur
{
    Blond, Bruin, Amber
}
```

```
public enum Biervolume
{
    c125, c133, c150, c175, c1100, c1150
}
```

```
public enum Setgrootte
{
    een, vier, zes, acht, twaalf, vierentwintig
}
```

Aangezien we onze code op een duidelijke manier wensen te organiseren zodat we snel kunnen vinden wat we zoeken, plaatsen we de verschillende enums elk in een eigen bestand. Ons project ziet er nu als volgt uit. *(Opmerking in het screenshot noemt het project BierWinkel2 – dit is enkel omdat ik met verschillende versies werk – dus het cijfertje kan je gerust negeren.)*



### Hoe kunnen we onze software nog verbeteren ?

We kunnen ervoor zorgen dat er enkel Bieren worden aangemaakt die voldoen aan onze eisen. Een alcoholpercentage kleiner dan 0 zullen we niet toestaan, evenals een biernaam die leeg is of enkel spaties bevat of een prijs die kleiner of gelijk aan 0 is.

We passen de constructor van onze Bier-klasse als volgt aan :

```
public Bier(double prijsPerStuk, string naam, Bierkleur? kleur, string brouwerij,
    Biervolume? volume, double? alcoholPercentage, Setgrootte minimumHoeveelheid)
{
    if (prijsPerStuk <= 0) throw new Exception("prijs moet groter zijn dan 0");
    PrijsPerStuk = prijsPerStuk;
    if (string.IsNullOrEmpty(naam)) throw new Exception("naam mag niet leeg zijn");
    Naam = naam;
    Kleur = kleur;
    Brouwerij = brouwerij;
    Volume = volume;
    if (alcoholPercentage < 0) throw new Exception("percentage kan niet kleiner zijn dan 0");
    AlcoholPercentage = alcoholPercentage;
    MinimumHoeveelheid = minimumHoeveelheid;
}
```

*Opmerking : Exceptions dienen om aan te geven dat iets is fout gelopen – we behandelen exceptions later in de cursus.*

Verder valt er nog op te merken dat er gekozen is om voor kleur, volume en alcoholpercentage een Nullable type te kiezen. Dit laat ons toe om bij het zoeken de voorwaarden te vereenvoudigen.

Aangezien we nu een aantal controles doen bij het aanmaken van een bier, moeten we nu ook beletten dat een gebruiker van onze library zomaar waarden kan aanpassen en de controles omzeilt. Daarom maken we de properties in onze klasse wel publiekelijk raadpleegbaar, maar niet meer aanpasbaar (private setters).

```

public double PrijsPerStuk { get; private set; }
3 references
public string Naam { get; private set; }
5 references
public Bierkleur? Kleur { get; private set; }
6 references
public string Brouwerij { get; private set; }
5 references
public Biervolume? Volume { get; private set; }
5 references
public double? AlcoholPercentage { get; private set; }
2 references
public Setgrootte MinimumHoeveelheid { get; private set; }

```

Op deze manier sluiten we uit dat er waarden worden ingegeven die we niet wensen.

Nu we een aantal aanpassingen hebben doorgevoerd om onze software veiliger te maken, is het tijd om de inventaris-klasse aan te pakken. We hadden reeds opgemerkt dat we in staat moeten zijn om een lijst van bieren terug te geven bij een zoekopdracht. Dit vereist slechts een kleine aanpassing, telkens we de lus doorlopen van de verschillende bieren voegen we het bier toe aan de lijst van gevonden bieren als het voldoet aan de voorwaarden.

Een tweede aanpassing die we doen is het optimaliseren van de SelecteerBier methode. In een vorige versie doorliepen we telkens de lijst van bieren en controleerden of de naam overeenstemde met het te zoeken bier. Dit gaan we veranderen door gebruik te maken van een dictionary waarbij we als sleutel de naam van het bier gaan gebruiken. Onze SelecteerBier methode is nu veel efficiënter.

Een derde aanpassing is de VoegBierToe methode. Hier voorzien we nu een controle om te vermijden dat we meerdere keren hetzelfde bier zouden opnemen in onze inventaris.

```

public class Inventaris
{
    private Dictionary<string,Bier> Biertjes = new Dictionary<string, Bier>();
    4 references
    public void VoegBierToe(double prijsPerStuk, string naam, Bierkleur kleur, string brouwerij,
        Biervolume volume, double alcoholPercentage, Setgrootte minimumHoeveelheid)
    {
        Bier bier = new Bier(prijsPerStuk,naam,kleur,brouwerij,volume,alcoholPercentage,minimumHoeveelheid);
        if (!Biertjes.ContainsKey(naam)) Biertjes.Add(bier.Naam,bier);
    }
    1 reference
    public Bier SelecteerBier(string naam)
    {
        if (Biertjes.ContainsKey(naam)) return Biertjes[naam];
        return null;
    }
    1 reference
    public List<Bier> ZoekBier(Bier bier)
    {
        List<Bier> gevondenBiertjes = new List<Bier>();
        foreach (Bier b in Biertjes.Values)
        {
            if (bier.Kleur !=null && bier.Kleur != b.Kleur) continue;
            if (bier.Brouwerij != null && bier.Brouwerij.Length > 0 && bier.Brouwerij.ToLower() != b.Brouwerij.ToLower())
            if (bier.Volume !=null && bier.Volume != b.Volume) continue;
            if (bier.AlcoholPercentage !=null && bier.AlcoholPercentage != b.AlcoholPercentage) continue;
            gevondenBiertjes.Add(b);
        }
        return gevondenBiertjes;
    }
}

```

*Het herschrijven van code waarbij de functionaliteit hetzelfde blijft maar de uitvoering, onderhoudbaarheid en flexibiliteit verbeteren noemen we **refactoring**.*

Onze software ziet er nu al een stuk beter uit en voldoet aan de eisen van onze klant.

Dan bekijken we nogmaals stap 2 (pas basis OO principes toe voor een flexibele oplossing). Hier kijken we vooral naar zaken zoals 'duplicate code' (dubbele codefragmenten) en 'bad class design'.

Wat ons opvalt is dat bij het zoeken van een bier in onze inventaris we een Bier-object meegeven, maar dat is feitelijk niet correct. We gebruiken namelijk slechts een aantal properties van deze klasse. De naam van het bier, de prijs en de hoeveelheid zijn parameters die niet worden gebruikt bij het zoeken. Een object met enkel de criteria (parameters) die nodig zijn voor het zoeken lijkt hierbij de aangewezen oplossing, we zullen deze klasse Bierspecificatie noemen.

### **Mismatched object type.**

*Een object dient enkel voor het uitvoeren van een bepaalde taak of het implementeren van een bepaald concept. Het uitvoeren van taken die niet het echte doel zijn van een object moeten we vermijden. En dat is precies wat hier aan de hand is, we gebruiken een object Bier om de selectiecriteria mee te geven aan de zoek-functie in de inventaris. Dit is duidelijk niet het doel van de klasse Bier.*

*Een aantal punten waar we moeten op letten als we deze problemen wensen op te sporen zijn :*

1. Objecten moeten doen wat hun naam suggereert. Voorbeeld een object *Trein* kan methoden hebben zoals *Rijden*, *Loskoppelen*, ... maar methoden zoals *VerkoopTicket* horen duidelijk niet bij dit object.
2. Elk object behoort tot één enkel concept. Een object moet één duidelijke taak hebben.
3. Objecten met ongebruikte eigenschappen (zoals in ons voorbeeld) wijzen meestal op een combinatie van taken/concepten.

De ZoekBier methode in de inventaris passen we dan als volgt aan :

```
public List<Bier> ZoekBier(Bierspecificatie bier)
{
    List<Bier> gevondenBiertjes = new List<Bier>();
    foreach (Bier b in Biertjes.Values)
    {
        if (bier.Kleur != null && bier.Kleur != b.Bierspecificatie.Kleur) continue;
        if (bier.Brouwerij != null && bier.Brouwerij.Length > 0 &&
            bier.Brouwerij.ToLower() != b.Bierspecificatie.Brouwerij.ToLower()) continue;
        if (bier.Volume != null && bier.Volume != b.Bierspecificatie.Volume) continue;
        if (bier.AlcoholPercentage != null && bier.AlcoholPercentage != b.Bierspecificatie.AlcoholPercentage) continue;
        gevondenBiertjes.Add(b);
    }
    return gevondenBiertjes;
}
```

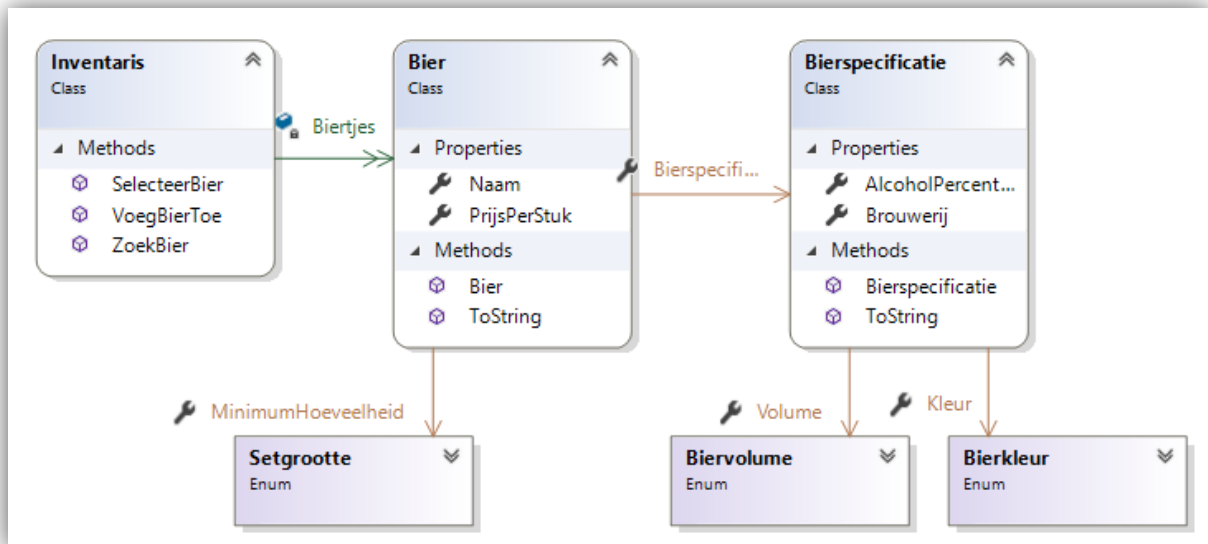
In de Bier klasse vervangen we nu de properties die in Bierspecificatie zijn opgenomen (en vermijden we dus dubbele code).

```
public class Bier
{
    1 reference
    public Bier(double prijsPerStuk, string naam, Bierkleur? kleur, string brouwerij,
        Biervolume? volume, double? alcoholPercentage, Setgrootte minimumHoeveelheid)
    {
        if (prijsPerStuk <= 0) throw new Exception("prijs moet groter zijn dan 0");
        PrijsPerStuk = prijsPerStuk;
        if (string.IsNullOrEmpty(naam)) throw new Exception("naam mag niet leeg zijn");
        Naam = naam;
        this.Bierspecificatie=new Bierspecificatie(kleur,brouwerij,volume,alcoholPercentage);
        MinimumHoeveelheid = minimumHoeveelheid;
    }
    6 references
    public Bierspecificatie Bierspecificatie { get; private set; }
    2 references
    public double PrijsPerStuk { get; private set; }
    3 references
    public string Naam { get; private set; }
    2 references
    public Setgrootte MinimumHoeveelheid { get; private set; }
    - references
    public override string ToString()...
}
```

## Encapsulation

*Inkapselen laat ons enerzijds toe om de interne werking van een deel van de applicatie te verbergen, maar maakt anderzijds ook duidelijk wat dat deel doet. Binnen OO refereert Encapsulation naar het combineren van zowel data als de methoden die op deze data inwerken binnen een geheel. Dit wordt gerealiseerd door middel van klassen.*

Het klassendiagram komt er dan als volgt uit te zien:



### Is dit nu goed ontworpen software ?

Stap 3 stelt dat we moeten streven naar een onderhoudbaar en herbruikbaar ontwerp. Voldoet onze oplossing hieraan ? We komen het snel te weten, want “Johnny” wil nog extra informatie toevoegen aan een biertje. Om tegemoet te komen aan buitenlandse klanten, wordt het assortiment uitgebreid en moet het ook mogelijk zijn om te selecteren op basis van het land waar het bier werd gebrouwen. Dit betekent een extra property **HerkomstLand**, die we als string zullen opslaan.

### Wat is nu de impact van deze verandering in onze code ?

In de constructor van de klasse **Bier** zijn alle properties opgenomen om een instantie van een object te initialiseren. Een extra property betekent dus een aanpassing in de klasse **Bier** en dat is niet echt wat we willen. Onze klasse **Bier** zou niet moeten worden aangepast als er een wijziging is bij de **Bierspecificatie**. Daarom zullen we de constructor wijzigen zodat we in de toekomst deze klasse niet meer moeten aanpassen.

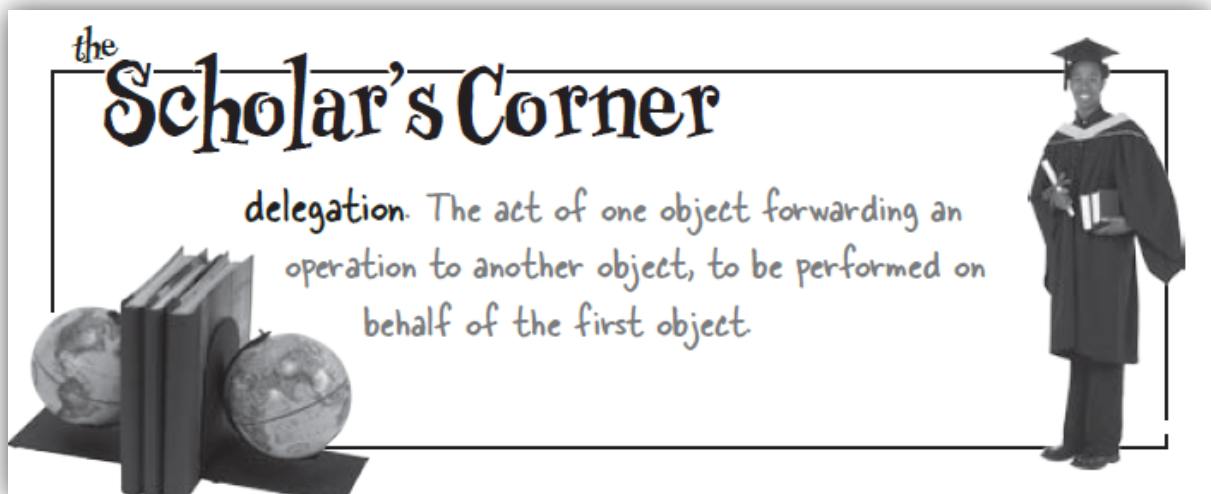
```
public Bier(double prijsPerStuk, string naam, Bierspecificatie bierspecificatie, Setgrootte minimumHoeveelheid)
{
    if (prijsPerStuk <= 0) throw new Exception("prijs moet groter zijn dan 0");
    PrijsPerStuk = prijsPerStuk;
    if (string.IsNullOrEmpty(naam)) throw new Exception("naam mag niet leeg zijn");
    Naam = naam;
    this.Bierspecificatie=bierspecificatie;
    MinimumHoeveelheid = minimumHoeveelheid;
}
```



Ook de klasse Inventaris draagt de gevolgen van de uitbreiding en is dus ook geen toonbeeld van onderhoudbaarheid. Om het probleem op te lossen passen we de methode VoegBierToe aan door een variabele van het type Bierspecificatie toe te voegen in plaats van de aparte properties. Op deze manier moeten we bij een volgende uitbreiding deze methode niet meer aanpassen.

```
public void VoegBierToe(double prijsPerStuk, string naam, Bierspecificatie bierspecificatie,
    Setgrootte minimumHoeveelheid)
{
    Bier bier = new Bier(prijsPerStuk, naam, bierspecificatie, minimumHoeveelheid);
    if (!Biertjes.ContainsKey(naam)) Biertjes.Add(bier.Naam, bier);
}
```

Rest ons nog de ZoekBier methode. Elke aanpassing aan Bierspecificatie maakt dat we deze methode moeten aanpassen. Om de koppeling tussen beide klassen losser te maken kunnen we gebruik maken van een principe dat we delegeren noemen.



(screenshot uit *Head First Object-Oriented Analysis & Design*, Brett D. McLaughlin, Garry Pollice & David West – Uitgeverij O'Reilly)

Het vergelijken van 2 specificaties laten we nu over aan de Bierspecificatie klasse zelf, waar we een methode VoldoetAanSpecificatie voorzien die ons een antwoord geeft op de vraag of een bier voldoet aan de vraag. We kunnen deze methode dan als volgt gaan gebruiken in de ZoekBier methode in de inventaris klasse.

```

public List<Bier> ZoekBier(Bierspecificatie bier)
{
    List<Bier> gevondenBiertjes = new List<Bier>();
    foreach (Bier b in Biertjes.Values)
    {
        if (b.Bierspecificatie.VoldoetAanSpecificatie(bier)) gevondenBiertjes.Add(b);
    }
    return gevondenBiertjes;
}

```

De methode zelf kan als volgt worden geschreven.

```

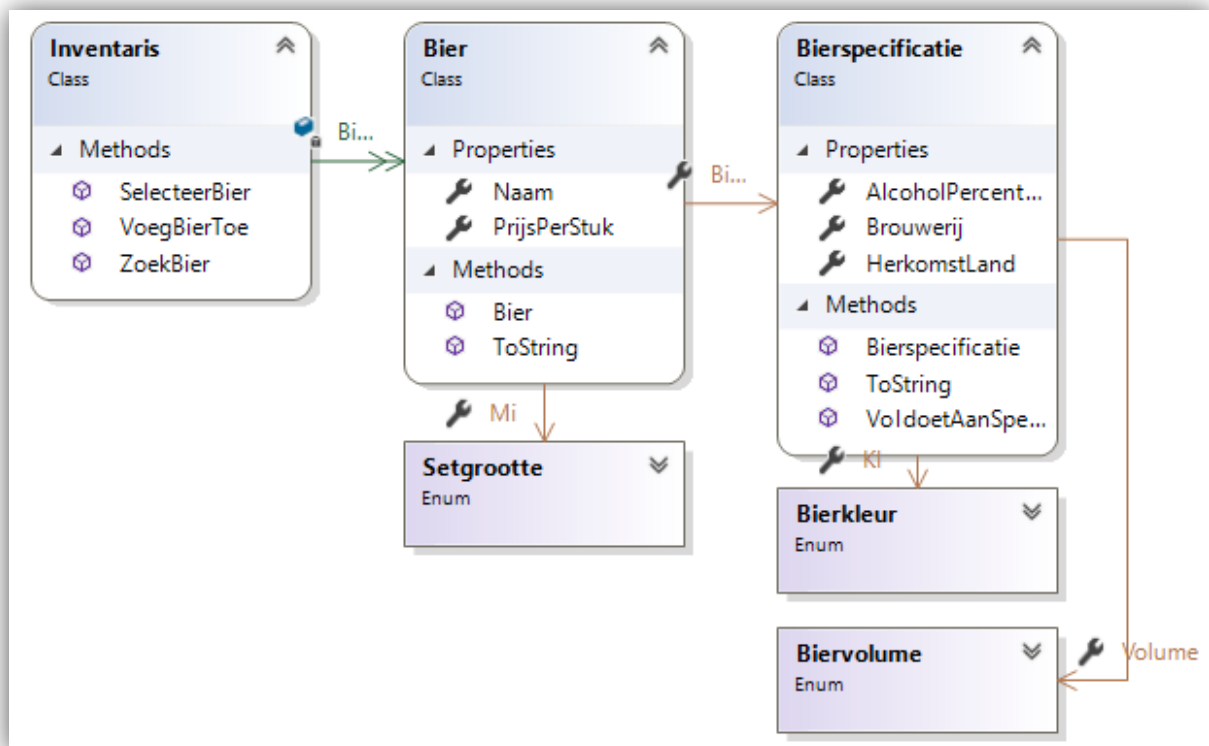
public bool VoldoetAanSpecificatie(Bierspecificatie spec)
{
    bool ok = true;
    if (spec.Kleur != null && spec.Kleur != this.Kleur) ok=false;
    if (spec.Brouwerij != null && spec.Brouwerij.Length > 0
        && spec.Brouwerij.ToLower() != this.Brouwerij.ToLower()) ok = false;
    if (spec.Volume != null && spec.Volume != this.Volume) ok = false;
    if (spec.AlcoholPercentage != null && spec.AlcoholPercentage != this.AlcoholPercentage) ok = false;
    if (spec.HerkomstLand != null && spec.HerkomstLand != this.HerkomstLand) ok = false;
    return ok;
}

```

Deze aanpassingen maken onze software nu veel meer geschikt voor uitbreiding en onderhoud. Een uitbreiding van de bierspecificaties heeft nu geen invloed meer op de klassen Bier en inventaris waardoor we een beter ontwerp ter beschikking hebben.

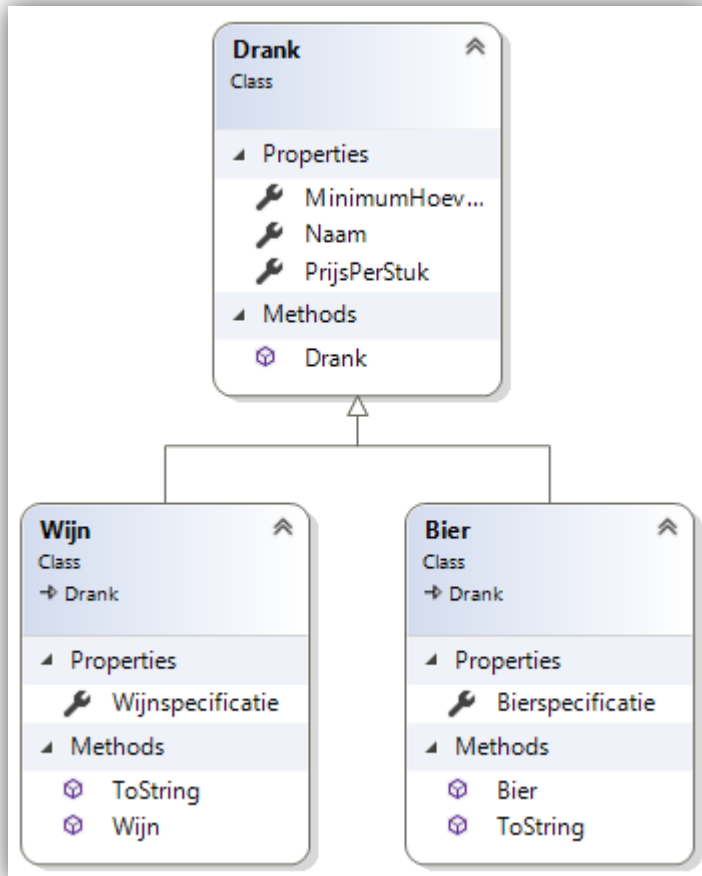
## DrankWinkel

Het gaat goed met Johnny's bierwinkel en uitbreiding komt eraan. Naast bier gaat de winkel nu ook wijn verkopen. Even kijken hoe goed onze software is ontworpen zodat uitbreidingen mogelijk zijn. Ons huidige ontwerp ziet er als volgt uit :



En nu wensen we ook wijn te verkopen. Er zijn natuurlijk heel wat overeenkomsten tussen Bier en Wijn (vanuit een software standpunt gezien !). De kleur is nu niet meer blond, bruin of amber maar wordt rood, wit en rosé. Voor de rest zijn de properties wel herbruikbaar.

Volgen we dezelfde logica als voor de Bierwinkel dat zullen we een klasse Wijn moeten aanmaken en een klasse wijnspecificatie. De klassen Bier en Wijn bevatten dus veel gemeenschappelijke properties en om duplicate code te vermijden gaan we gebruik maken van overerving. Daarbij gaan we een klasse maken die de 'ouder' is van de klassen Bier en Wijn en die de gemeenschappelijke zaken zal bevatten.



De klasse Bier ziet er dan als volgt uit :

```

public class Bier : Drank
{
    0 references
    public Bier(double prijsPerStuk, string naam, Bierspecificatie bierspecificatie, Setgrootte minimumHoeveelheid)
        : base(prijsPerStuk, naam, minimumHoeveelheid)
    {
        Bierspecificatie = bierspecificatie;
    }
    1 reference
    public Bierspecificatie Bierspecificatie { get; private set; }
    - references
    public override string ToString()...
}
  
```

Via de syntax

```

public class Bier : Drank
  
```

geven we aan dat de klasse Bier overerft van de klasse Drank en dus alle properties/methoden van de klasse Drank zijn ook beschikbaar voor de klasse Bier. Voor de klasse Wijn doen we nu net hetzelfde.

De klasse Drank zelf bevat nu de gemeenschappelijke eigenschappen en is als volgt geschreven :

```

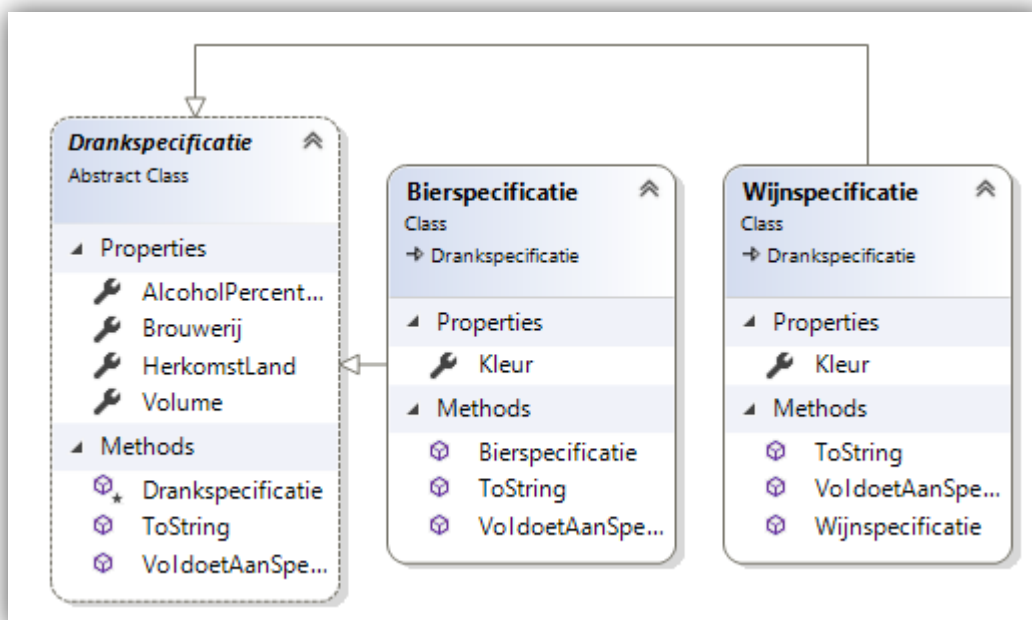
public class Drank
{
    3 references
    public Drank(double prijsPerStuk, string naam, Setgrootte minimumHoeveelheid)
    {
        if (prijsPerStuk <= 0) throw new Exception("prijs moet groter zijn dan 0");
        PrijsPerStuk = prijsPerStuk;
        if (string.IsNullOrEmpty(naam)) throw new Exception("naam mag niet leeg zijn");
        Naam = naam;
        MinimumHoeveelheid = minimumHoeveelheid;
    }

    2 references
    public double PrijsPerStuk { get; private set; }
    3 references
    public string Naam { get; private set; }
    2 references
    public Setgrootte MinimumHoeveelheid { get; private set; }

    25 references
    public override string ToString()...
}

```

Ook voor de klassen Wijnspecificatie en Bierspecificatie zijn er veel gemeenschappelijke eigenschappen en kunnen we dus eveneens overerving gaan toepassen. Enkel de property kleur is verschillend bij beiden en de methode VoldoetAanSpecificatie. Ons aangepast model heeft dan de volgende vorm :



Rest ons enkel nog de klasse inventaris. Overerving maakt het mogelijk dat we geen aparte lijsten moeten bijhouden met bieren en wijnen, we kunnen één gemeenschappelijke lijst van dranken bijhouden. Ook het toevoegen van een drank kan worden gegeneraliseerd. Voor het zoeken naar een drank moeten we el nog onderscheid maken of we gebruik maken van een Bierspecificatie of een Wijnspecificatie.

```

public class Inventaris
{
    private Dictionary<string, Drank> Dranken = new Dictionary<string, Drank>();
    0 references
    public void VoegDrankToe(double prijsPerStuk, string naam, Drankspecificatie drankspecificatie,
        Setgrootte minimumHoeveelheid) {...}
    0 references
    public Drank SelecteerDrank(string naam) {...}
    0 references
    public List<Drank> ZoekDrank(Wijnspecificatie drank) {...}
    0 references
    public List<Drank> ZoekDrank(Bierspecificatie drank) {...}
}

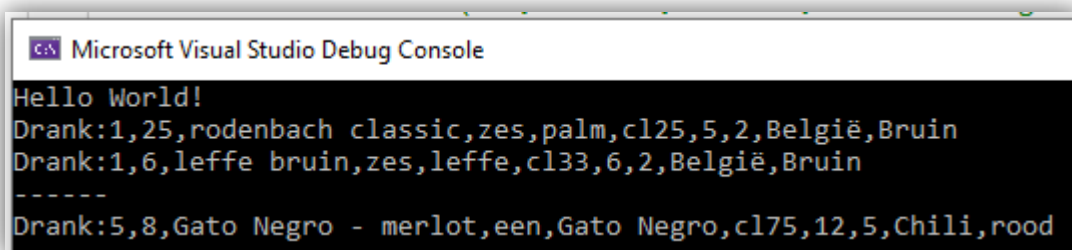
```

Het zou natuurlijk veel handiger zijn als er maar één zoekmethode zou zijn die bijvoorbeeld als volgt zou werken :

```

Bierspecificatie bierspec = new Bierspecificatie(Bierkleur.Bruin, null, null, null, null);
List<Drank> ly = inventaris.ZoekDrank(bierspec);
foreach (Drank y in ly)
    Console.WriteLine($"Drank:{y}");
Console.WriteLine("-----");
Wijnspecificatie wijnspec = new Wijnspecificatie(Wijnkleur.rood, null, null, null, "Chili");
List<Drank> lz = inventaris.ZoekDrank(wijnspec);
foreach (Drank y in lz)
    Console.WriteLine($"Drank:{y}");

```



```

Microsoft Visual Studio Debug Console
Hello World!
Drank:1,25,rodenbach classic,zes,palm,cl25,5,2,België,Bruin
Drank:1,6,leffe bruin,zes,leffe,cl33,6,2,België,Bruin
-----
Drank:5,8,Gato Negro - merlot,een,Gato Negro,cl75,12,5,Chili,rood

```

**Hoe moeten we dit nu realiseren ?**

De methode die we zouden willen in onze inventarisklasse ziet er als volgt uit :

```

public List<Drank> ZoekDrank(Drankspecificatie drank)
{
    List<Drank> gevondenDranken = new List<Drank>();
    foreach (Drank d in Dranken.Values)
    {
        if (d.Drankspecificatie.VoldoetAanSpecificatie(drank)) gevondenDranken.Add(d);
    }
    return gevondenDranken;
}

```

Als parameter geven we een Drankspecificatie mee (in plaats van een Wijn- of Bierspecificatie) en we hebben een methode VoldoetAanSpecificatie nodig die via de property Drankspecificatie de correcte methode oproept in de klasse Bierspecificatie of Wijnspecificatie (afhankelijk van welk type drank we hebben).

### **Polymorphisme**

*Polymorfisme hangt nauw samen met overerving. Wanneer een klasse (sub class) overerft van een andere klasse (parent class) dan laat polymorfisme toe dat de subklasse de rol overneemt van de ouder/super klasse. In ons voorbeeld hebben we een Drankspecificatie klasse als ouder klasse, maar de Wijn- en Bierspecificatieklassen nemen de rol over en hun specifieke methodes worden uitgevoerd in plaats van deze uit de ouder klasse.*

Dit vraagt natuurlijk enkele aanpassingen. De Drankspecificatie klasse bevat een methode VoldoetAanSpecificatie die controleert of er aan de overkoepelende voorwaarden is voldaan.

```
public virtual bool VoldoetAanSpecificatie(Drankspecificatie spec)
{
    bool ok = true;
    if (spec.Brouwerij != null && spec.Brouwerij.Length > 0
        && spec.Brouwerij.ToLower() != this.Brouwerij.ToLower()) ok = false;
    if (spec.Volume != null && spec.Volume != this.Volume) ok = false;
    if (spec.AlcoholPercentage != null && spec.AlcoholPercentage != this.AlcoholPercentage) ok = false;
    if (spec.HerkomstLand != null && spec.HerkomstLand != this.HerkomstLand) ok = false;
    return ok;
}
```

In de Wijn- en Bierspecificatie klassen gaan we deze methode overschrijven en aanvullen met de specifieke voorwaarden die horen bij de betreffende klasse. Vandaar dat we in de ouderklasse (Drankspecificatie) het keyword **virtual** gebruiken. Op die manier geven we aan dat deze methode zal worden overschreven in de afgeleide klassen.

De specifieke implementatie ziet er dan als volgt uit voor de Wijnspecificatie klasse. Hier gebruiken we het keyword **override** om aan te geven dat we een bepaalde methode uit een ouderklasse overschrijven.

```
public override bool VoldoetAanSpecificatie(Drankspecificatie spec)
{
    if (!base.VoldoetAanSpecificatie(spec)) return false;
    if (spec.GetType() != typeof(Wijnspecificatie)) return false;
    if (((Wijnspecificatie)spec).Kleur != null && ((Wijnspecificatie)spec).Kleur != this.Kleur)
        return false;
    return true;
}
```

Bekijken we de code even in detail dan zien we dat we eerst de code uit de basis (ouder) klasse aanroepen, daarna controleren we of het type van de parameter spec wel van het type Wijnspecificatie is (indien dit een Bierspecificatie zou zijn – of bij verdere uitbreiding nog iets anders – dan voeren we de test niet uit). In de volgende code regel controleren we dan de specifieke

properties van de Wienspecificatie. Opmerking : hier is het dan wel nodig om onze spec variabele te casten naar een Wijnspecificatie, anders kunnen we de property Kleur niet bereiken.

Nog een laatste aanpassing die nodig is om dit te laten werken is het opnemen van een property Drankspecificatie in de klasse Drank. Indien we dit niet zouden doen, dan is het noodzakelijk dat we in onze zoekmethode in de klasse inventaris toch steeds moeten onderscheid maken tussen Wijn of Bier.

De klasse Drank komt er dan als volgt uit te zien :

```
public class Drank
{
    3 references
    public Drank(double prijsPerStuk, string naam, Setgrootte minimumHoeveelheid, Drankspecificatie drankspecificatie) {...}

    2 references
    public double PrijsPerStuk { get; private set; }
    3 references
    public string Naam { get; private set; }
    2 references
    public Setgrootte MinimumHoeveelheid { get; private set; }
    3 references
    public Drankspecificatie Drankspecificatie { get; private set; }

    - references
    public override string ToString() {...}
}
```

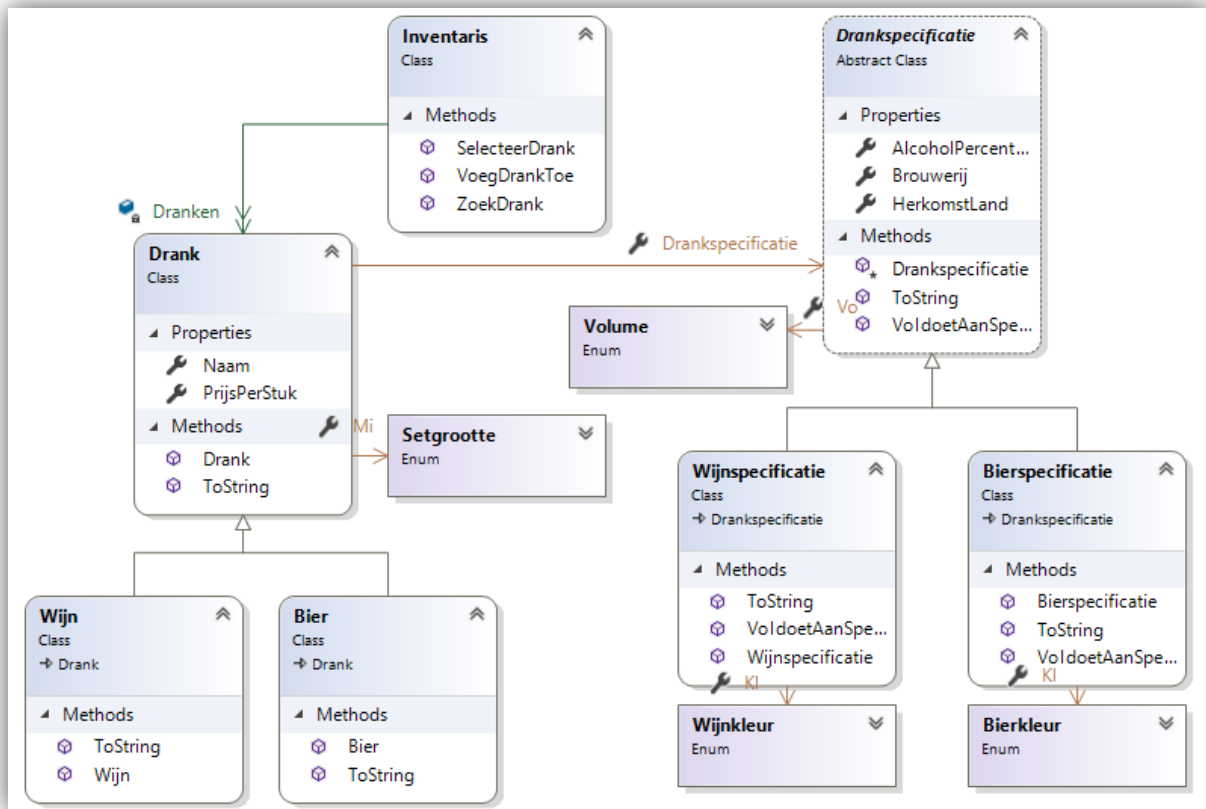
En de subklassen Bier en Wijn bevatten enkel nog een constructor.

```
public class Bier : Drank
{
    0 references
    public Bier(double prijsPerStuk, string naam, Bierspecificatie bierspecificatie, Setgrootte minimumHoeveelheid)
        : base(prijsPerStuk, naam, minimumHoeveelheid, bierspecificatie)
    {
    }
    - references
    public override string ToString() {...}
}
```

```
public class Wijn : Drank
{
    0 references
    public Wijn(double prijsPerStuk, string naam, Wijnspecificatie wijnspecificatie, Setgrootte minimumHoeveelheid)
        : base(prijsPerStuk, naam, minimumHoeveelheid, wijnspecificatie)
    {
    }
    25 references
    public override string ToString() {...}
}
```

Het finale ontwerp ziet er uiteindelijk als volgt uit :





O ja, er zijn natuurlijk nog wat esthetische aanpassingen nodig. Zo veranderen we de naam van Biervolume naar volume (anders wordt het nogal warrig voor de wijndrinkers). En onze applicatie zelf dopen we om tot drankwinkel (opent perspectieven voor nog verdere uitbreidingen).

***Cheers !***