

1.0.0

## **H17: polymorfisme en interfaces**

# Polymorfisme

**Polymorfisme** oftewel "meerdere vormen" is een programmeertechniek waarbij je code met één algemene noemer op meerdere manieren implementeert. Hierdoor kan je algemene code schrijven die verschillende specifieke effecten kan vertonen naargelang de configuratie. Samen met encapsulatie, abstractie en overerving is polymorfisme een vierde belangrijke eigenschap van object georiënteerd programmeren.



Nog eens samengevat:

- encapsulatie betekent dat je de gebruiker niet laat sleutelen aan de interne werking van data (bv. door access modifiers te gebruiken)
- overerving betekent dat je code uitspaart door code van een andere klasse te "recycleren"
- abstractie betekent dat je data zo algemeen mogelijk behandelt en irrelevante verschillen tussen data verbergt (bv. door variabelen met het type van een ouderklasse te declareren)
- polymorfisme betekent dat je werkt met een algemene voorstelling van data, maar dat de eigenlijke data bij uitvoering van het programma specifiek kan zijn en specifiek kan werken; het is eigenlijk de keerzijde van abstractie

We tonen de werking van polymorfisme aan de hand van een voorbeeld:

## Polymorfisme in de praktijk: Dieren

Een voorbeeld maakt veel duidelijk. Stel dat we een aantal Dier-gerelateerde klassen hebben die allemaal op hun eigen manier een geluid voortbrengen. We hanteren de klasse `Dier`:

```
abstract class Dier {  
    public abstract string MaakGeluid();  
}
```

Twee child-klassen:

```

class Paard : Dier {
    public override string MaakGeluid() {
        return "Hinnikhinnik";
    }
    // deze methode is alleen voor paarden
    public void Galoppeer() {
        Console.WriteLine("klipklop");
    }
}
class Varken : Dier {
    public override string MaakGeluid() {
        return "Oinkoink";
    }
}

```

Dankzij polymorfisme kunnen we nu elders objecten van Paard en Varken in een Dier bewaren, maar toch hun eigen geluid laten reproduceren:

```

Dier eenDier = new Varken();
Dier anderDier = new Paard();
Console.WriteLine(eenDier.MaakGeluid()); //Oinkoink
Console.WriteLine(anderDier.MaakGeluid()); //Hinnikhinnik

```

Het is belangrijk te beseffen dat eenDier en anderDier van het type Dier zijn en dus enkel die dingen kunnen die in Dier beschreven staan. Dit is het gevolg van de abstractie van beide soorten dieren tot één algemeen datatype Dier. Je kan eenDier en anderDier wel declareren als Varken en Paard respectievelijk, maar dat doe je best alleen als je hiermee methodes wil oproepen die enkel in Varken of Paard aanwezig zijn. Dit gaat bijvoorbeeld niet:

```

// werkt niet omdat anderDier gedeclareerd is als type Dier
anderDier.Galoppeer();

```

De declaratie vertelt de compiler dat hij alleen moet onthouden dat anderDier alles kan wat in Dier beschreven staat. Galoppeer valt daar niet onder. **De waarde is tijdens de uitvoering van type Paard, maar de variabele is van type Dier. Tijdens de compilatiefase wordt alleen gebruik gemaakt van de informatie die in de types van de variabelen staat.**

Als je die methode toch wil kunnen gebruiken, moet je de declaratie anders schrijven:

```

Paard anderDier = new Paard();
// geen probleem
// zowel de variabele als de waarde zijn van type Paard
anderDier.Galoppeer();

```

Datastructuren zoals arrays, lijsten, dictionaries,... bieden heel veel mogelijkheden in combinatie met abstractie en polymorfisme. Je kan een lijst van de basis-klasse maken en deze vullen met allerlei objecten van de basis-klasse **én de child-klassen**.

Een voorbeeld:

```
List<Dier> zoo = new List<Dier>();
zoo.Add(new Varken());
zoo.Add(new Paard());
foreach(Dier dier in zoo){
    Console.WriteLine(dier.MaakGeluid());
}
```

Deze code is algemener dan wat je zou kunnen doen met twee aparte lijsten. Zo kan je hier paarden en varkens door elkaar voorstellen. Er hangt wel een prijskaartje aan: zonder cast kan je hier geen gebruik maken van `Galoppeer`.

# Polymorfisme in de praktijk

## ✓ Kennisclip

Beeld je in dat je een klasse `President` hebt met een methode `RunTheCountry` (voorbeeld van [StackOverflow](#)). De president heeft toegang tot tal van adviseurs die hem kunnen helpen (inzake militair, binnenlands beleid, economie). Zonder de voordelen van polymorfisme zou de klasse `President` er zo kunnen uitzien, **slechte manier**:

```
public class President
{
    MilitaryMinister general = new MilitaryMinister();
    ForeignSecretary secretary = new ForeignSecretary();
    HealthOfficial doctor = new HealthOfficial();

    public void RunTheCountry()
    {
        // people walk into the Presidents office and he tells them what to do
        // depending on who they are.
        general.IncreaseTroopNumbers();
        general.ImproveSecurity();
        general.PayContractors();

        secretary.StallNegotiations();
        secretary.LowBallFigure();
        secretary.FireDemocraticallyElectedIraqiLeaderBecauseIDontLikeHim();

        doctor.IncreasePremiums();
        doctor.AddPreexistingConditions();
    }
}
```

De `MilitaryMinister` zou er zo kunnen uitzien:

```

class MilitaryMinister
{
    public void IncreaseTroopNumbers()
    {
        //..
    }
    public void ImproveSecurity()
    {
        //..
    }
    etc
}

```

De `HealthOfficial` -klasse heeft dan weer heel andere publieke methoden. En die `ForeignSecretary` ook weer totaal andere.

Je merkt dat de president (of de programmeur van deze klasse) aardig wat specifieke kennis moet hebben van de vele verschillende departementen van het land. De verschillende ministers kunnen zelf geen initiatief nemen en moeten alle instructies voorgekauwd krijgen. Bovenstaande code is dus zeer slecht. Telkens er zaken binnen het takenpakket van een bepaalde minister wijzigen moet dit ook in de klasse `President` aangepast worden.

Dankzij polymorfisme kunnen we dit alles veel mooier oplossen:

1. We maken **abstractie** van de taken van alle ministers. We bekijken het dus algemener. We zeggen: "elke minister heeft bepaalde eigen taken en het interesseert ons niet hoe deze worden uitgevoerd."
2. We verplichten alle adviseurs dat ze overerven van de abstracte klasse `Advisor` die maar 1 abstracte methode heeft `Advise` :

```

abstract class Advisor
{
    abstract public void Advise();
}

class MilitaryMinister:Advisor
{
    public override void Advise()
    {
        increaseTroopNumbers();
        improveSecurity();
        payContractors();
    }
    private void increaseTroopNumbers(){ ... }
    private void improveSecurity(){ ... }
    private void payContractors(){ ... }
}

class ForeignSecretary:Advisor
{
    //...
}

class HealthOfficial:Advisor
{
    //...
}

```

Het leven van de president wordt veel makkelijker:

```

public class President
{
    public void RunTheCountry()
    {
        Advisor general = new MilitaryAdvisor();
        Advisor secretary = new ForeignSecretary();
        Advisor doctor = new HealthOfficial();
        general.Advise(); // # Petraeus says send 100,000 troops to Fallujah
        secretary.Advise(); // # she says negotiate trade deal with Iran
        doctor.Advise(); // # they say we need to spend $50 billion on ObamaCare
    }
}

```

We kunnen ook nog meer flexibiliteit bieden door één lijst adviseurs op te stellen, zodat er op een bepaald moment meer of minder adviseurs kunnen zijn:



```
public class President
{
    public void RunTheCountry()
    {

        List<Advisor> allMinisters= new List<Advisor>();
        allMinisters.Add(new MilitaryAdvisor());
        allMinisters.Add(new ForeignSecretary());
        allMinisters.Add(new HealthOfficial());

        // Ask advice from each:
        foreach (Advisor minister in allMinisters)
        {
            minister.Advise();
        }
    }
}
```

De president moet nu gewoon de juiste adviseurs kunnen kiezen. We kunnen veranderen hoe elk van deze adviseurs zijn taak vervult, zonder de code van `President` aan te passen. We kunnen ook makkelijk nieuwe types adviseurs toevoegen (bv. voor landbouw, cyberbeveiliging,...).

# Interfaces



## Betekenenissen

Het woord "interface" heeft meerdere betekenissen:

- de totale verzameling methodes en properties van een bepaalde klasse
  - hierin wordt soms onderscheid gemaakt tussen de publieke interface en de totale interface
- een *language construct* dat toestaat vast te leggen dat bepaalde methodes of properties deel uitmaken van de publieke interface in de eerste zin van het woord

Je moet beide betekenissen begrijpen. De eerste is meer een abstract concept, de tweede kan je programmeren in C# en staat toe nog meer polymorfisme toe te passen.

⚠ Deze pagina heeft niet echt iets te maken met "grafische user interface". Ook daar wil "interface" zeggen "wat je kan gebruiken", maar verder is er geen verband.

Een interface als *language construct* is een garantie dat bepaalde methodes en properties geïmplementeerd zijn door een bepaalde klasse. Dit vertelt ons niets over hoe deze methodes en properties geïmplementeerd zijn.

## Interfaces in C#

Volgende code toont hoe we een interface implementeren voor data die we naar een CSV-file willen kunnen wegschrijven en die we terug willen kunnen uitlezen.

```
interface ICSVSerializable
{
    string ToCsv();

    string Separator
    {
        get;
        set;
    }
}
```

Enkele opmerkingen:

- Het woord `class` wordt niet gebruikt, in de plaats daarvan gebruiken we `interface`.
- Het is een vrij algemene afspraak om interfaces met een `I` te laten starten in hun naamgeving
- Methoden en properties gaan niet vooraf van `public`: interfaces zijn van nature net publiek, dus alle methoden en properties van de interface zijn dat bijgevolg ook.
- Er wordt geen code/implementatie gegeven: iedere methode eindigt ogenblikkelijk met een puntkomma.

Als we deze interface nu koppelen aan een klasse, **moeten** we deze methodes implementeren.

**i** Een interface is een beschrijving hoe een component een andere component kan gebruiken, zonder te zeggen hoe dit moet gebeuren. De interface is met andere woorden 100% scheiding tussen de methode/Property-signatuur en de eigenlijke implementatie ervan.

**i** Dit lijkt wel heel erg op een afgewaterde abstracte klasse? Ja, maar er zijn goede redenen om interfaces te gebruiken. De simpelste: je mag maar van één klasse erven, maar je mag zo veel interfaces implementeren als je wil. Als je wil weten waarom, zie [hier](#). De uitleg gaat over Java maar kan rechtstreeks toegepast worden op C#.

## Regels voor interfaces

- Je kan geen constructor declareren
- Je kan geen access specificeren (`public`, `protected`, etc): alles is public
- Een interface kan niet overerven van een klasse, wel van een andere interface. In dat geval moet een implementatie de velden en properties van de ouderinterface en de kindinterface voorzien.
- Je hoeft geen `override` te schrijven wanneer je een methode of property implementeert.

## Toepassing interfaces

Volgende code toont hoe we kunnen aangeven dat een klasse `Student` serialisatie van en naar CSV voorziet:

```

class Student : ICSVSerializable
{
    private string naam;
    private byte leeftijd;
    private string separator;

    public string Separator {
        get {
            return this.separator;
        }
        set {
            this.separator = value;
        }
    }

    public Student(string naam, byte leeftijd) {
        this.naam = naam;
        this.leeftijd = leeftijd;
        this.Separator = ";";
    }

    public string ToCsv() {
        return $"Student{Separator}{this.naam}{Separator}{this.leeftijd}";
    }
}

```

Een nuttige toepassing van deze interface zou een data dump kunnen zijn. Als we een lijst bijhouden van alle `ICSVSerializable` objecten, kunnen we deze in één beweging exporteren. De code die deze export uitvoert mag dezelfde zijn voor allerlei verschillende soorten data: studenten, huisdieren, voedingsproducten,... Maakt niet uit!

## Meerdere interfaces

Het is toegelaten meerdere interfaces te implementeren. Volgende code staat toe een `Student` op te slaan op schijf door hem te serialiseren als CSV of XML, naargelang je voorkeur.

```

interface IXMLSerializable
{
    string ToXml();
}

class Student : ICSVSerializable, IXMLSerializable
{
    private string naam;
    private byte leeftijd;
    private string separator;
    public string Separator {
        get {
            return this.separator;
        }
        set {
            this.separator = value;
        }
    }

    public Student(string naam, byte leeftijd) {
        this.naam = naam;
        this.leeftijd = leeftijd;
    }

    public string ToCsv() {
        return $"Student{this.Separator}{this.naam}{this.Separator}{this.leeftijd}";
    }

    public string ToXml() {
        return $"<Student>
            <naam>{this.naam}</naam>
            <leeftijd>{this.leeftijd}</leeftijd>
        </Student>";
    }
}

```

## Interfaces vs. overerving

Je kan je hier terecht afvragen of we interfaces wel nodig hebben, aangezien we abstractie en polymorfisme ook kunnen bereiken met behulp van (al dan niet abstracte) klassen. Maar er zijn goede redenen voor het bestaan van interfaces:

- Om technische redenen kan een klasse maar één ouderklasse hebben. Een klasse kan wel om het even welk aantal interfaces implementeren. Er zijn weliswaar talen waarin je meerdere ouderklassen kan hebben, maar daarin is een object altijd "iets meer" ouderklasse 1 dan ouderklasse 2 en dat kan heel onoverzichtelijk worden.
- Er is een betekenisverschil: een kindklasse **is** een specifiek type van de ouderklasse. Een klasse die een interface implementeert **kan** gewoon bepaalde zaken. Dit is een beetje een kwestie van interpretatie, maar als vuistregel leidt dit tot betere ontwerpen dan lukraak gebruik van overerving.
- Er is een praktisch verschil: bij overerving wil je **veel** overnemen van de ouderklasse, want zo bespaar je code. Interfaces wil je juist **beperkt** houden: je moet elk van je interfaces volledig implementeren.

Deze puntjes samen leiden ook tot een tweede vuistregel voor het gebruik van interfaces: kan je wel (veel) code besparen met overerving? Bijvoorbeeld bij `ToCsv` zal de methode voor elk object verschillen en win je dus niets met een ouderklasse tegenover een interface, terwijl je wel zorgt dat je geen andere ouderklasse meer kan hebben.

# Losse koppeling

## "Losse koppeling"

In een objectgeoriënteerd programma zijn objecten de onderdelen waaruit je programma bestaat. Eén kenmerk van een goed ontworpen programma is dat deze onderdelen vlot inwisselbaar zijn voor andere onderdelen die dezelfde taak vervullen, maar op een andere manier. Anders gezegd: ze hebben dezelfde **interface** (nu in de "algemene" betekenis), maar een andere **implementatie**. Als dit vlot gaat, spreken we over een "losse koppeling" ("loose coupling") van de onderdelen van het programma. Interfaces (in de "language construct"-betekenis) spelen hier een belangrijke rol in in C#.

### Voorbeeld uit het echte leven: stopcontacten

Je gebruikt elke dag dezelfde het principe van losse koppeling. Je kan je elektrische tandenborstel, de lader van je GSM, de lader van je laptop, je desktop,... allemaal aansluiten op elk stopcontact in je huis. Dat komt omdat ze allemaal afgestemd zijn op West-Europese stopcontacten.

Dit is niet vanzelfsprekend: een laptop trekt bijvoorbeeld veel meer stroom dan een elektrische tandenborstel. Daarom wordt je laptop geleverd met een transformator ("stroomblok"). In principe zouden we gebouwen kunnen maken met aparte stopcontacten voor laptops zonder stroomblok, voor elektrische tandenborstels, enzovoort. Dan hadden we geen stroomblokken meer nodig, **maar dan kon je elk toestel maar op sommige stopcontacten aansluiten**. De toestellen zouden iets makkelijker te maken zijn, maar ze zouden niet meer inwisselbaar zijn, omdat ze allemaal een andere aansluiting zouden hebben.

Door een stroomblok toe te voegen, zorgen we dat elk toestel dezelfde "interface" heeft, zelfs wanneer de implementatie verschillend is.

### Voorbeeld in code: `IEnumerable<T>`

Als je een klasse schrijft waaraan je een reeks elementen, bijvoorbeeld van de klasse `Student`, wil linken, dan heb je een heleboel keuzes. Enkele opties zijn:

- `List<Student>`
- `ImmutableList<Student>`
- `HashSet<Student>`
- `LinkedList<Student>` (deze ken je waarschijnlijk nog niet)

Elke optie heeft haar voor- en nadelen. `List` is vrij algemeen bruikbaar. `ImmutableList` kan veiligere code opleveren. `HashSet` zorgt er automatisch voor dat je geen dubbels in je verzameling plaatst. `LinkedList` kan snellere code opleveren als je vooral elementen vooraan of achteraan in de lijst toevoegt.

Je kan twee dingen doen:

- op voorhand één lijsttype kiezen en je programma overal afstemmen op dat lijsttype
- een zo eenvoudig mogelijke interface bepalen die je in je programma nodig hebt en daar mee werken

De eerste optie is zoals stopcontacten voor elk type apparaat ontwerpen. De tweede is zoals vastleggen hoe het stopcontact er uit ziet en dan compatibele apparaten kiezen. In code:

### Optie 1

⚠ Dit gaat over studenten en klasgroepen enz. maar hoort niet bij SchoolAdmin!

```
public class KlasGroep {
    private List<Student> studenten = new List<Student>();
    public List<Student> Studenten {
        get {
            return this.studenten;
        }
    }
    // nog methoden
}
```

Code die objecten van `KlasGroep` aanmaakt, bijvoorbeeld `School`, zal nu misschien veronderstellen dat `Studenten` altijd een `List<Student>` is. Ze zal misschien zelf variabelen declareren van type `List<Student>`. Dit maakt dat de code niet los gekoppeld is, maar sterk gekoppeld. Als je iets aanpast in `KlasGroep`, moet je ook aanpassingen doen in `School`.

### Optie 2

```
public class KlasGroep {
    private IEnumerable<Student> studenten = new List<Student>();
    public IEnumerable<Student> Studenten {
        get {
            return this.studenten;
        }
    }
    // nog methoden
}
```


De `IEnumerable` interface geeft aan dat iets een opsomming van elementen is. `List` is een soort opsomming, maar `ImmutableList` is dat ook, `HashSet` is dat ook en `LinkedList` ook. Nu is `KlasGroep` zo algemeen mogelijk. Er worden studenten bijgehouden en deze kunnen worden opgesomd. Code van `School` zal niet meer veronderstellen dat ze gebruik mag maken van methodes van `List`. Dat zal misschien wat meer code vragen, maar het voordeel is dat we nu de implementatie van `KlasGroep` kunnen veranderen zonder effect op `School`. Bijvoorbeeld:



```
public class KlasGroep {  
    // andere implementatie, zelfde interface  
    private IEnumerable<Student> studenten = new LinkedList<Student>();  
    public IEnumerable<Student> Studenten {  
        get {  
            return this.studenten;  
        }  
    }  
    // not methoden  
}
```

Om deze reden zal je soms gevraagd worden ergens een interface te gebruiken, terwijl een gewone klasse op het eerste zicht net zo goed zou volstaan. Sommige softwareprojecten gaan hier heel ver in en gebruiken voor zowat alle velden interfaces in plaats van klassen. Dat maakt het bijvoorbeeld heel makkelijk productiecode om te wisselen voor testcode.

# Labo

 Onderstaande oefeningen maak je oproepbaar via een klasse `Polymorfisme` met een methode `ToonSubmenu`.

## h17-autoconstructeur

### Functionele analyse

Omdat we aan de vooravond staan van de transitie van klassieke aandrijvingen naar meer milieubewuste aandrijvingen van auto's heeft een autoconstructeur beslist om zijn assemblage software te herwerken.

In de huidige software wordt gebruik gemaakt van een superklasse motor met twee subklassen bezinemotor en dieselmotor. Dit wordt nu uitgebreid.

Er zijn verschillende nieuwe motoren op de markt: elektrische, CNG, waterstof,...

Niet alleen de motor is verschillend maar ook de periferie (omgeving) van de motor is erg verschillend. Denk bv. aan de brandstofvoorziening (voor benzine, diesel, elektrisch).

### Technische analyse

Maak een interface `IAandrijving` die de volgende methoden en properties ondersteunt:

```
void EnergieToevoegen(); // Het vroegere tanken  
  
void Vertragen(int kmPerUurPerSeconde, int doelsnelheid);  
  
void Versnellen(int kmPerUurPerSeconde, int doelsnelheid);
```

Maak een klasse voor de volgende types aandrijvingen die de interface `IAandrijving` implementeren

```
AandrijvingElektrisch  
  
AandrijvingBenzine  
  
AandrijvingCNG
```

De implementatie van de methodes is steeds

```
Console.WriteLine("<Naam van de methode> - <Type aandrijving>");
```

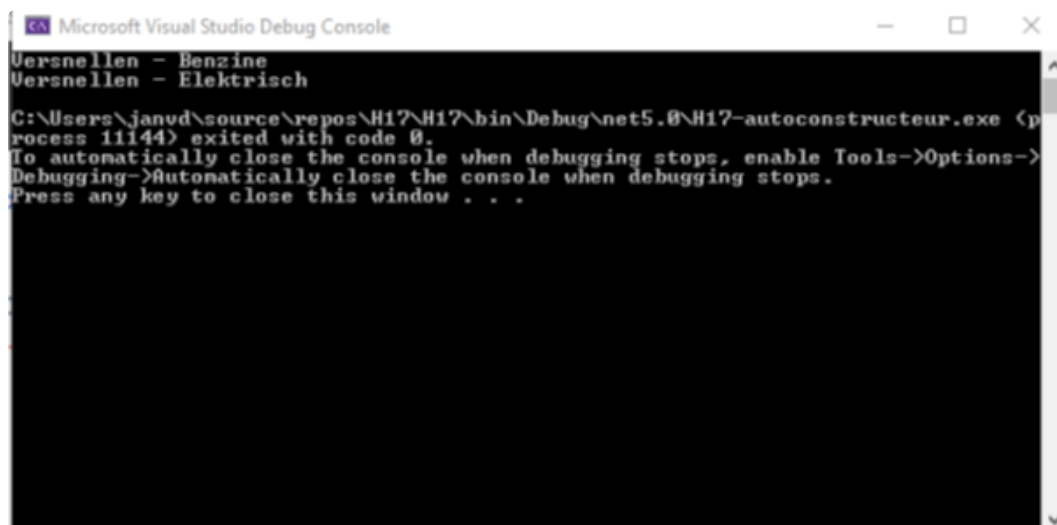
Bijvoorbeeld:

```
Console.WriteLine("Versnellen - Benzine");
```

Maak een klasse `Auto` met public property (van het type `string`) `AutoType` en een public property `Aandrijving`. Zorg er voor dat elk type van aandrijving kan toegevoegd worden aan een object van type `Auto`.

Instantieer een auto met benzine aandrijving. Doe dat door de aandrijving als parameter in de constructor mee te geven. De constructor van de auto heeft als signatuur dus `public Auto (string autoType, ? aandrijving)`. Het `?` moet je zelf invullen. Laat de auto versnellen. Bouw die auto nu om naar een elektrische aandrijving. Laat de auto opnieuw versnellen. Doe dit allemaal in een methode `DemonstreerAandrijving`.

### Voorbeeldinteractie



```
Microsoft Visual Studio Debug Console
Vernsellen - Benzine
Vernsellen - Elektrisch
C:\Users\janvd\source\repos\H17\H17\bin\Debug\net5.0\H17-autoconstructeur.exe (p
rocess 11144) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->
Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

## h17-grootkeuken

### Functionele analyse

Je wordt gevraagd om een grootkeuken van een studentenrestaurant te automatiseren.

In de keuken staan een zestal ketels die bepaalde functies wel of niet hebben. De functies zijn:

- `Verwarmen(int doelTemperatuur)`
- `Afkoelen(int doelTemperatuur)`
- `StoomVerwarmen(int doelTemperatuur)`
- `WaterDosereren(int hoeveelheid)`

Er bestaan 4 types van ketels

1. `Ketel` (dit is een abstracte klasse zonder enige functionaliteit)
2. Stoomketel (met functionaliteit `StoomVerwarmen`, `Afkoelen`, `WaterDosereren`) klasse `StoomKetel`
3. Gewone ketel zonder doseren (met functionaliteit `Verwarmen`) klasse `KetelZonderDosereren`
4. Gewone ketel met dosering (met functionaliteit `Verwarmen`, `WaterDosereren`) klasse `KetelMetDosereren`

## Technische analyse

Creëer 4 interfaces voor de verschillende functies: `IVerwarmen`, `IAfkoelen`, `IStoomVerwarmen`, `IWaterDosereren`). In de interface vinden we steeds de functie als methode terug (met return type `void`).

Creëer de 3 types van ketels. Gebruik overerving om gemeenschappelijke properties (inhoud type `int` en temperatuur type `int`) te implementeren. De constructors van de drie types ketels krijgen de inhoud als parameter mee. De temperatuur blijft op de defaultwaarde staan.

Demonstreer je code door 6 ketels te instantiëren. Zet de code hiervoor in `DemonstreerGrootkeuken`:

Ketel 1 : Stoomketel met capaciteit 300l

Ketel 2 : Stoomketel met capaciteit 300l

Ketel 3 : Ketel zonder doseren met capaciteit 150l

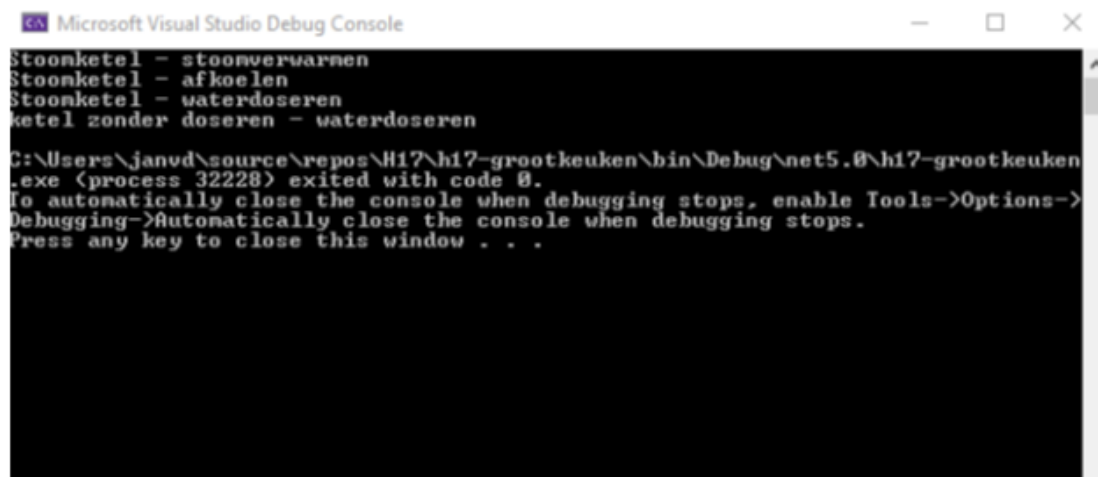
Ketel 4 : Ketel zonder doseren met capaciteit 300l

Ketel 5 : Ketel met doseren met capaciteit 200l

Ketel 6 : Ketel met doseren met capaciteit 150l

Verwarm ketel 1 tot 100 graden Celcius... zie interactie:

## Voorbeeldinteractie



```
Microsoft Visual Studio Debug Console
Stoomketel - stoomverwarmen
Stoomketel - afkoelen
Stoomketel - waterdosereren
ketel zonder doseren - waterdosereren

C:\Users\janvd\source\repos\H17\h17-grootkeuken\bin\Debug\net5.0\h17-grootkeuken.exe (process 32228) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

## h17-Rooster-stap1

### Functionele analyse

We schrijven een kalender. Hierop kunnen we verschillende zaken plaatsen: afspraken en taken. Beide werken anders, maar beide nemen wel een zekere hoeveelheid tijd in beslag.

### Technische analyse

- Schrijf twee klassen, `Afspraak` en `Taak`
- Voor een afspraak moet je volgende zaken bijhouden:
  - de tijd om je naar de afspraak te verplaatsen (een `TimeSpan`)
  - de tijd om terug te komen (een `TimeSpan`)
  - de duur van de afspraak (een `TimeSpan`)
  - een omschrijving (een `string`)
- Voor een taak moet je volgende zaken bijhouden:
  - de werktijd die je nodig zal hebben (een `TimeSpan`)
  - een omschrijving (een `string`)

Beide hebben constructoren die de hierboven genoemde parameters in volgorde bevatten.

Zowel afspraken als taken zijn roosterbaar op een kalender. Dit maak je mogelijk door hen allebei te voorzien van de `IRoosterbaar` interface. Deze omvat:

- een read-only property `Tijdsduur` die een `TimeSpan` teruggeeft
  - voor een afspraak is dit de som van de verplaatsingstijd en de duur van de afspraak
  - voor een taak is dit gewoon de duur van de taak
- een read-only property `Omschrijving` die een `string` teruggeeft
  - voor een taak kan je gewoon de bestaande property gebruiken
  - voor een afspraak geef je de bestaande omschrijving, gevolgd door de tekst (inclusief verplaatsing)

Test uit met volgende code, die je `DemonstreerIRoosterbaar` noemt (in de klasse voor dit labo).

```
IRoosterbaar blok1 = new Afspraak(new TimeSpan(0,20,0),new TimeSpan(1,0,0),new TimeSpan(0,20,0));
IRoosterbaar blok2 = new Taak(new TimeSpan(2,0,0),"dagelijkse oefeningen OOP");
System.Console.WriteLine($"Totale kalendertijd: {(blok1.Tijdsduur + blok2.Tijdsduur).Hours}u
```

### Voorbeeldinteractie

## h17-Rooster-stap2

### Functionele analyse

We willen onze taken en afspraken nu echt kunnen inplannen op een interactieve kalender.

### Technische analyse

- Schrijf een klasse `Kalender`. Een kalender heeft een naam en koppelt tijdstippen aan roosterbare gebeurtenissen. Enkel de naam wordt meegegeven bij constructie. Voor de koppeling gebruik je een `Dictionary<DateTime, IRoosterbaar>`.
- Een `Kalender` heeft een methode `VoegToe`. Deze vraagt eerst om wat voor gebeurtenis het gaat (`Taak` of `Afspraak`) en vraagt dan om alle properties van dit type object. Daarna vraagt ze: "Wanneer moet dit geroosterd worden"? Ten slotte wordt het roosterbare object geassocieerd met dit tijdstip. **Let op:** voor een afspraak vraag je wanneer **de afspraak zelf** geroosterd moet worden, maar rooster je vanaf het moment dat je moet vertrekken naar de afspraak.

### Voorbeelduitvoering

Schrijf zelf een methode `DemonstreerKalender1`. Deze vraagt maakt een kalender met naam "DemonstratieKalender" en vraagt de gebruiker objecten toe te voegen tot hij niet meer wil doorgaan. Daarna wordt de inhoud van de kalender getoond.

```

Om wat voor object gaat het?
1. Afspraak
2. Taak
1
Omschrijving?
Tandarts
Aantal minuten verplaatsing?
20
Aantal minuten afspraak zelf?
40
Aantal minuten om terug te keren?
20
Wanneer moet dit geroosterd worden?
20/05/2021 1:00 PM
Wil je nog een item toevoegen?
ja
Om wat voor object gaat het?
1. Afspraak
2. Taak
2
Omschrijving?
Projectwerk
Aantal minuten werk?
120
Wanneer moet dit geroosterd worden?
20/05/2021 5:00 PM
Wil je nog een item toevoegen?
nee
5/20/2021 12:40:00 PM:Tandarts
5/20/2021 5:00:00 PM:Projectwerk

```


## h17-Rooster-stap3

### Functionele analyse

Onze code is te sterk gekoppeld. Om Kalender te schrijven, hebben we code moeten schrijven om beide soorten objecten in te lezen. Als we nog meer tijdsblokken willen inbouwen (bijvoorbeeld `QualityTime`), moeten we `Kalender` verder uitbreiden.

### Technische analyse

Voorzie `Taak` en `Afspraak` van een constructor zonder parameters. Voorzie de interface `IRoosterbaar` van een methode `Initialiseer` en van een methode `RoosterOm(DateTime referentiepunt)`. De methode `Initialiseer` vraagt alle gegevens voor een object van dat type en stelt ze in. De methode `RoosterOm` bepaalt uit het referentiepunt wanneer de kalender moet worden ingeblokt.

 We werken met een constructor zonder parameters, gevolgd door initialisatie, omdat we geen statische methoden kunnen toevoegen aan een interface. Er zijn elegantere oplossingen, maar we willen niet te ver afwijken van de koers.

### Voorbeeldinteractie

Deze ziet eruit zoals hierboven, maar de demonstratiecode is nu:

```

public void VoegToeLosgekoppeld() {
    System.Console.WriteLine("Om wat voor object gaat het?");
    System.Console.WriteLine("1. Afspraak");
    System.Console.WriteLine("2. Taak");
    IRoosterbaar item;
    DateTime begin;
    int antwoord = Convert.ToInt32(Console.ReadLine());
    if (antwoord == 1) {
        item = new Afspraak();
    }
    else {
        item = new Taak();
    }
    item.Initialiseer();
    System.Console.WriteLine("Wanneer moet dit geroosterd worden?");
    begin = Convert.ToDateTime(Console.ReadLine(), new CultureInfo("nl-BE"));
    this.Rooster[item.RoosterOm(begin)] = item;
}

```

Merk op dat je maar een heel kleine aanpassing zou moeten doen om `Kalender` uit te breiden met bijvoorbeeld `QualityTime`. Die klasse zou door iemand anders geschreven mogen worden.

## SchoolAdmin project: sorteren volgens criteria

### Functionele analyse

We willen graag de data in ons systeem gesorteerd weergeven. We willen de gebruiker de keuze geven om te sorteren op verschillende manieren. Dit ben je ongetwijfeld gewoon van op webwinkels waar je kan sorteren volgens prijs, productnaam,...

### Technische analyse



- Om dit klaar te spelen, heb je een klasse nodig die de `IComparer<T>` interface implementeert. Deze interface bestaat al. Je hoeft hem niet te schrijven. Je moet hem alleen implementeren.
- Bijvoorbeeld, om studenten op naam te sorteren, kan je een `StudentenVolgensNaamComparer` schrijven die `IComparer<Student>` implementeert.
- Deze interface bevat één methode `Compare(T, T)`. Deze vergelijkt twee objecten van één type.
  - Als het eerste argument voor het tweede gesorteerd moet worden, geeft de methode een negatief getal terug.
  - Als het eerste argument na het tweede gesorteerd moet worden, geeft de methode een positief getal terug.
  - Als het niet uitmaakt, geeft ze 0 terug.
- Door een instantie van een `IComparer` als argument mee te geven aan `Sort`, kan je sorteren op basis van de implementatie van `Compare`.
- Voeg hiermee volgende functionaliteit toe aan je systeem:
  - Een methode `Student.ToonStudenten` die je kan oproepen vanaf het keuzemenu
    - Bij het tonen van studenten, moet de gebruiker kunnen kiezen om ze te tonen in stijgende of dalende alfabetische volgorde.
    - `null` zou niet mogen voorkomen, maar je mag dit altijd vooraan zetten in om het even welke lijst objecten
  - Een methode `Cursus.ToonCursussen` die je kan oproepen vanaf het keuzemenu
    - Bij het tonen van cursussen, moet de gebruiker kunnen kiezen om ze te tonen volgens cursusnaam van A naar Z of volgens oplopend aantal studiepunten.
    - Voorzie een `ToString` die de titel van de cursus toont, gevolgd door het aantal studiepunten tussen haakjes om te controleren of alles werkt
    - `null` zou niet mogen voorkomen, maar je mag dit altijd vooraan zetten in om het even welke lijst objecten

## Voorbeeldinteractie

```

Wat wil je doen?
1. DemonstreerStudenten uitvoeren
2. DemonstreerCursussen uitvoeren
3. DemonstreerStudentUitTekstFormaat uitvoeren
4. DemonstreerStudieProgramma uitvoeren
5. DemonstreerAdministratiefPersoneel uitvoeren
6. DemonstreerLectoren uitvoeren
7. Student toevoegen
8. Cursus toevoegen
9. VakInschrijving toevoegen
10. Inschrijvingsgegevens tonen
11. Studenten tonen
12. Cursussen tonen
8
Titel van de cursus?
Programmeren
Aantal studiepunten?
9
Wat wil je doen?
1. DemonstreerStudenten uitvoeren
2. DemonstreerCursussen uitvoeren
3. DemonstreerStudentUitTekstFormaat uitvoeren
4. DemonstreerStudieProgramma uitvoeren
5. DemonstreerAdministratiefPersoneel uitvoeren
6. DemonstreerLectoren uitvoeren
7. Student toevoegen
8. Cursus toevoegen
9. VakInschrijving toevoegen
10. Inschrijvingsgegevens tonen
11. Studenten tonen
12. Cursussen tonen
8
Titel van de cursus?
Databanken
Aantal studiepunten?
3
Wat wil je doen?
1. DemonstreerStudenten uitvoeren
2. DemonstreerCursussen uitvoeren
3. DemonstreerStudentUitTekstFormaat uitvoeren
4. DemonstreerStudieProgramma uitvoeren
5. DemonstreerAdministratiefPersoneel uitvoeren
6. DemonstreerLectoren uitvoeren
7. Student toevoegen
8. Cursus toevoegen
9. VakInschrijving toevoegen
10. Inschrijvingsgegevens tonen
11. Studenten tonen
12. Cursussen tonen
12
In welke volgorde wil je cursussen tonen?
1. Stijgend alfabetisch
2. Volgens studiepunten
2
Databanken (3)
Programmeren (9)

```

## SchoolAdmin project: data export naar CSV

### Functionele analyse

We zouden graag alle entiteiten in ons systeem in één beweging kunnen exporteren naar CSV-formaat. Zo kunnen we makkelijk heel ons systeem voorzien van een backup zonder al te veel code. We zullen dit hier doen voor enkele entiteitstypes, maar niet allemaal, om ons niet te verliezen in de details.

## Technische analyse

Schrijf een interface `ICSVSerializable`. Deze bevat één objectmethode zonder parameters, namelijk `ToCSV`. Het return type is `string`.

Deze interface wordt geïmplementeerd door `Persoon` en door `Cursus`. Voor elk object tonen we steeds eerst de naam van de klasse waartoe het object behoort, gevolgd door puntkomma, gevolgd door het Id van het object.

Voor een persoon tonen we daarna (met telkens puntkomma's tussen):

- de naam tussen dubbele aanhalingstekens
- de geboortedatum

Voor personeel tonen we verder voor elke taak:

- de omschrijving van die taak tussen dubbele aanhalingstekens
- de hoeveelheid werk die in die taak kruipt

Voor lectoren tonen we ook per cursus:

- het Id van de cursus
- het aantal uren voor die cursus


Voor studenten tonen we ook per entry in het dossier:

- de datum
- de tekst tussen dubbele aanhalingstekens

Voor cursussen tonen we ten slotte ook de titel tussen aanhalingstekens en het aantal studiepunten.

Om zeker te zijn dat een datum op elke machine op dezelfde manier wordt voorgesteld, mag je hem zo omzetten naar een `string: Geboortedatum.ToString(new CultureInfo("nl-BE"))`

Dit garandeert dat de Vlaamse voorstellingswijze voor een datum wordt gebruikt.

 Tip: gebruik overerving om de gemeenschappelijke aspecten niet telkens opnieuw te schrijven. Je kan dit ofwel doen via `base.ToCSV` ofwel met een hulpmethode die de serialisatie van het gedeelte van de ouderklasse afhandelt. De eerste aanpak levert je minder methodes, de tweede kan voorkomen dat je vergeet de methode af te werken in de kindklassen.

## Voorbeeldinteractie

```
Wat wil je doen?  
1. DemonstreerStudenten uitvoeren  
2. DemonstreerCursussen uitvoeren  
...  
13. Alle exporteerbare data exporteren  
13  
Cursus;1;"Communicatie";3  
Cursus;2;"Programmeren";3  
Cursus;3;"Webtechnologie";6  
Cursus;4;"Databanken";5  
Student;1;"Said Aziz";3/01/2001 00:00:00  
Student;2;"Mieke Vermeulen";1/02/2000 00:00:00
```