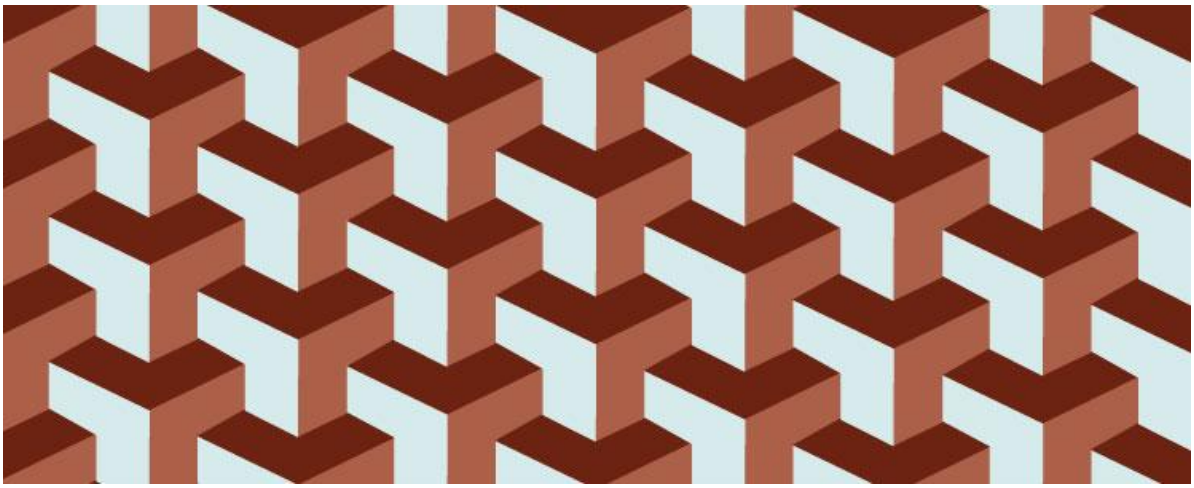
VDAB

samen sterk voor werk

C#

Design Patterns



Inhoudsopgave

1 INLEIDING

1.1 Doelstelling

In deze cursus leer je werken met design patterns. Dit zijn voorgedefinieerde oplossingen voor problemen die vaak voorkomen bij het schrijven van code.

1.2 Vereiste voorkennis

- C# programming fundamentals

1.3 Nodige software

- Visual Studio

1.4 Algemeen

Bij het schrijven van code komen regelmatig soortgelijke problemen voor. Erich Gamma, Richard Helm, Ralph Johnson en John Vlissides hebben deze problemen beschreven in het boek Design patterns. Ze hebben elk probleem een naam gegeven en ook een mooie oplossing toegevoegd.

Het geheel van een probleem met zijn oplossing heet een design pattern (ontwerppatroon).

Ze hebben de patronen ingedeeld in drie categorieën :

- Creational patterns :
  lossen problemen op bij het *maken* van objecten.
- Structural patterns :
  lossen problemen op bij het *samenstellen* van objecten.
- Behavioral patterns :
  lossen problemen op bij het *samenwerken* van objecten.

De code in het boek is geschreven in de programmeertaal C++.
Andere mensen hebben de design patterns ook geschreven in andere programmeertalen.

Je leert in deze cursus de meest gebruikte design patterns kennen.

1.5 Werkwijze

1. Je hebt een probleem in je code.
2. Je overloopt de design patterns en je zoekt het pattern dat overeenstemt met je probleem.
3. Je leest de voorbeeldoplossingscode van het pattern.
4. Je past deze code toe op je eigen probleem.

1.6 Principes

Bij design patterns worden enkele principes van goed programmeren gebruikt.
Je ziet hieronder deze principes.

1.6.1 Program to an interface, not an implementation

Het type van een variabele, van een parameter of het returntype van een method is beter een interface dan een class. Een variabele met als type een interface is *flexibeler* dan een variabele met als type een class. Een variabele met als type een interface kan verwijzen naar elk object dat de interface implementeert. Een variabele met als type een class kan enkel verwijzen naar een object van diezelfde class, of naar een derived class van die class.

1.6.2 Favor composition over inheritance

Composition (class A bevat een private variabele met als type een class B) is *flexibeler* dan inheritance (class B erft van class A). Inheritance kan niet wijzigen tijdens de uitvoering van je programma: je kan class B die erft van class A tijdens de uitvoering niet laten erven van een class C.

Als class A een private variabele bevat van het type B, kan deze variabele, als deze als type een interface heeft, eerst verwijzen naar een class C die de interface implementeert, en later verwijzen naar een class D die ook deze interface implementeert.

1.6.3 Loosely coupled design

Als een class A samenwerkt met een class B, probeer je deze classes zo te schrijven dat als je class A aanpast, je class B niet (of minimaal) moet aanpassen. Omgekeerd schrijf je class B zo dat als je class B aanpast, je class A niet (of minimaal) moet aanpassen.

1.6.4 Single responsibility

Een class heeft maar één reden tot verandering. Als een class twee verantwoordelijkheden heeft, zijn er twee mogelijke redenen om deze class te wijzigen. Als een class slechts één verantwoordelijkheid bevat, is er slechts één reden om deze class te wijzigen.

Slecht voorbeeld: een class stelt de werking van een scorebord van een spelletje voor én bepaalt hoe het scorebord wordt weggeschreven naar de harde schijf. Als de werking van het scorebord wijzigt moet je deze class wijzigen. Als je het scorebord niet meer naar een XML bestand wegschrijft, maar naar een database, moet je deze class ook wijzigen.

Verbeterd voorbeeld: één class stelt de werking van het scorebord voor. Je moet deze class slechts wijzigen als de werking van het scorebord wijzigt. Een andere class bepaalt hoe je het scorebord wegschrijft naar de harde schijf. Je moet deze class slechts wijzigen als je de manier van wegschrijven wijzigt. Je moet hierbij de eerste class (die de werking van het scorebord voorstelt) niet wijzigen. Elke class is klein en overzichtelijk.

1.6.5 A class should be open for extension and closed for modification

Als hetgeen van een class verwacht wordt uitbreidt, moet je deze class niet wijzigen, maar maak je een nieuwe afgeleide class van de oorspronkelijke class. Het is in deze afgeleide class dat je de uitbreiding schrijft. Gezien je de oorspronkelijke class niet moet wijzigen, kan je ook geen fouten introduceren in de oorspronkelijke class. Elke class blijft klein en overzichtelijk.

2 SINGLETON

2.1 Categorie

Creational design pattern

2.2 Probleem

Je gebruikt het singleton design pattern als van een class slechts één object bestaat in de werkelijkheid. Het singleton design patterns zorgt er voor…

- dat er maar één object kan bestaan in je code (en niet meerdere per ongeluk).
- dat dit ene object overal in je code gemakkelijk aanspreekbaar is.

2.3 Oplossing

Maak een nieuwe consoleapplicatie en voeg er een class Singleton aan toe. Deze class bevat een static member van het type Singleton (1). We voegen ook een constructor toe maar deze is private (2). Dit zorgt er voor dat je buiten de class geen Singleton-object(en) kan aanmaken met `new Singleton()`. Om een Singleton-object aan te roepen gebruik je de static method `GetInstance()` (3). Wanneer deze method voor het eerst wordt opgeroepen is de static member `_uniekeInstantie` leeg (4). We maken dan een instantie van Singleton aan door de private constructor op te roepen. Het aangemaakte Singleton wordt in de private static member `_uniekeInstantie` opgeslagen (5). Bij alle volgende oproepen van `GetInstance()` hoeven we geen nieuwe instantie meer aan te maken maar kunnen we de instantie uit de static member `_uniekeInstantie` gebruiken.

```
using System;

namespace SingletonDesignPattern
{
    public class Singleton
    {
        private static Singleton _uniekeInstantie;                          (1)

        private Singleton() { } // Private constructor !!                   (2)

        public static Singleton GetInstance()                               (3)
        {
            if (_uniekeInstantie == null)                                   (4)
            {
                _uniekeInstantie = new Singleton();                         (5)
            }
            return _uniekeInstantie;
        }

        public void ZegGoeiedag()
        {
            Console.WriteLine("Dag iedereen !");
        }
    }
}
```

We kunnen deze class nu gebruiken in het hoofdprogramma :

```
using System;

namespace SingletonDesignPattern
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            var singleton = Singleton.GetInstance();                    (1)
            singleton.ZegGoeiedag();                                    (2)

            var tweedeSingleton = Singleton.GetInstance();              (3)
            if (singleton == tweedeSingleton)                           (4)
                Console.WriteLine("Zelfde instanties");

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

(1) We maken een nieuw Singleton-object aan met de method GetInstance().
(2) We kunnen de method ZegGoeiedag() aanspreken.
(3) We proberen een tweede Singleton-object aan te maken.
(4) singleton en tweedeSingleton verwijzen allebei naar hetzelfde object.

2.4 Concreet voorbeeld

In de software van een auto heb je maar één object van het type motor. We zullen de motor starten en stoppen.

De code voor de motor :

```
public class Motor
{
    private static Motor _uniekeInstantie;

    private bool gestart;                                               (1)

    private Motor() { } // Private constructor !!

    public static Motor GetInstance()
    {
        if (_uniekeInstantie == null)
        {
            _uniekeInstantie = new Motor();
        }
        return _uniekeInstantie;
    }

    public void Start()
    {
        if (!gestart) {                                                 (2)
            gestart = true;
            Console.WriteLine("Motor werd gestart.");
        }
        else
            Console.WriteLine("Motor werd reeds gestart.");             (3)
    }

    public void Stop()
    {
        if (gestart) {
            gestart = false;
            Console.WriteLine("Motor werd gestopt.");
        }
        else
            Console.WriteLine("Motor werd reeds gestopt.");
    }

}
```

(1) We houden in een membervariabele bij of de motor gestart is of niet. Deze variabele wordt (door het bool-type) automatisch op false geïnitialiseerd.
(2) Wordt de method Start() opgeroepen dan controleren we of de motor uit staat. In dat geval wijzigen we de membervariabele. Anders verschijnt een foutboodschap.
(3) Op een gelijkaardige manier testen we de staat van de motor vooraleer we deze stoppen.

In het hoofdprogramma starten we de motor tot twee maal toe en stoppen we deze. Na de oproep van de Start-method van mijnMotor krijgen we de melding dat de motor reeds gestart is. Dit wijst erop dat deMotor en mijnMotor duidelijk naar dezelfde instantie verwijzen.

```
static void Main(string[] args)
{
    var deMotor = Motor.GetInstance();
    deMotor.Start();

    //allerlei andere code

    var mijnMotor = Motor.GetInstance();
    mijnMotor.Start();
    mijnMotor.Stop();

    Console.ReadLine();

}
```

2.5 Oefeningen

Maak de onderstaande oefeningen in de takenbundel :

 Singleton

 Singleton 2

3 SIMPLE FACTORY

3.1 Categorie

Creational design pattern: een fabriek (factory) *maakt* objecten

3.2 Probleem

Je moet op meerdere plaatsen in je code kiezen welk object je maakt uit een reeks van objecten. Als je deze code herhaalt, moet je deze code ook op meerdere plaatsen aanpassen bij fouten en bij uitbreidingen.

3.3 Voorbeeld

Je moet op meerdere plaatsen in je code kiezen of je een Tekst object, een Rekenblad object of een Presentatie object maakt, gebaseerd op een bestandsextensie (.docx, .xlsx, .pptx).

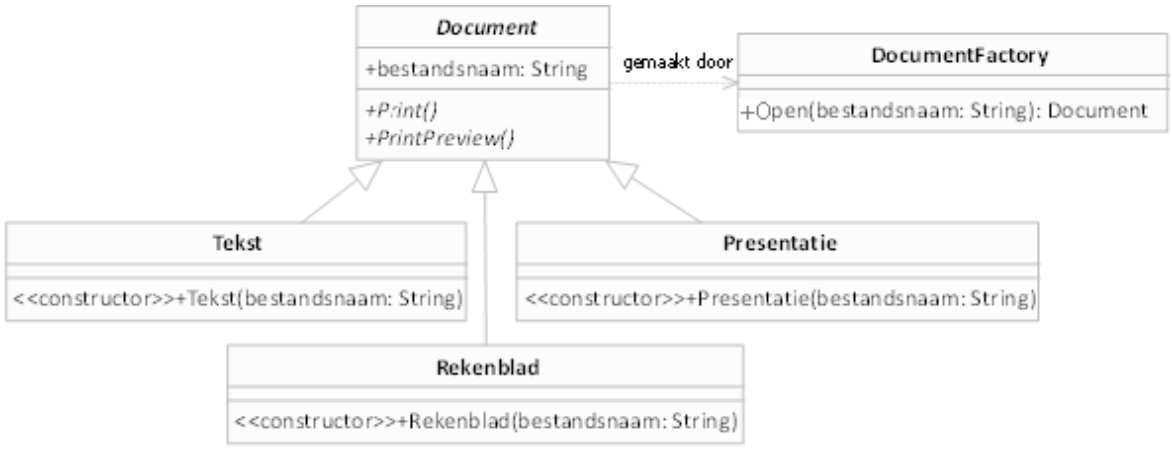
Het codefragment om de keuze te maken:

```
switch (extensie)
{
    case "docx":
        return new Tekst(bestandsnaam);
    case "xlsx":
        return new Rekenblad(bestandsnaam);
    case "pptx":
        return new Presentatie(bestandsnaam);
    default:
        throw new Exception("Verkeerde extensie");
}
```

Als je dit codefragment meerdere keren in je applicatie opneemt, moet je dit codefragment meerdere keren aanpassen als er nog een bestandstype (bvb. database) bijkomt.

3.4 Oplossing

Je schrijft de code die de beslissing neemt niet meerdere keren in je programma, maar één keer in een class (de factory) die je van overal in je programma kan oproepen.



De method open geeft een `Tekstverwerker`, `Rekenblad` of `Presentatie` object terug, naargelang de extensie van de bestandsnaam. Bemerk dat het returntype van de method `Document` is (niet `Tekst`, `Presentatie` of `Rekenblad`) om deze flexibiliteit te bekomen.

Je ziet in het voorbeeld dat de class een duidelijke naam heeft: DocumentFactory, m.a.w. een factory voor documenten. Dit is een 'best practice': zo ziet een programmeur onmiddellijk welk design pattern is gebruikt en waarvoor deze class dient.

```
public class DocumentFactory
{
    public Document Open(string bestandsnaam)
    {
        string extensie = bestandsnaam.Substring(bestandsnaam.Length - 4);
        switch (extensie)
        {
            case "docx":
                return new Tekst(bestandsnaam);
            case "xlsx":
                return new Rekenblad(bestandsnaam);
            case "pptx":
                return new Presentatie(bestandsnaam);
            default:
                throw new Exception("Verkeerde extensie.");
        }
    }
}


public abstract class Document
{
    public string Bestandsnaam { get; set; }
    protected Document(string bestandsnaam)
    {
        Bestandsnaam = bestandsnaam;
    }
    public abstract void Print();
    public abstract void PrintPreview();
}


public class Tekst : Document
{
    public Tekst(string bestandsnaam) : base(bestandsnaam) { }
    public override void Print()
    {
        Console.WriteLine("Een afdruk van een tekst");
    }
    public override void PrintPreview()
    {
        Console.WriteLine("Een afdrukvoorbeeld van een tekst");
    }
}


public class Rekenblad : Document
{
    public Rekenblad(string bestandsnaam) : base(bestandsnaam) { }
    public override void Print()
    {
        Console.WriteLine("Een afdruk van een rekenblad");
    }
    public override void PrintPreview()
    {
        Console.WriteLine("Een afdrukvoorbeeld van een rekenblad");
    }

}
```

```
public class Presentatie : Document
{
    public Presentatie(string bestandsnaam) : base(bestandsnaam) { }
    public override void Print()
    {
        Console.WriteLine("Een afdruk van een presentatie");
    }
    public override void PrintPreview()
    {
        Console.WriteLine("Een afdrukvoorbeeld van een presentatie");
    }
}
```

Je kan deze classes als volgt gebruiken:

```
static void Main(string[] args)
{
    DocumentFactory factory = new DocumentFactory();
    Document document = factory.Open("liedje.docx");
    document.PrintPreview();
    document.Print();
}
```

3.5 Factory als singleton

Je hebt maar één object van de factory nodig in je applicatie. Je kan daarom deze factory als een singleton schrijven. Dit is een voorbeeld van het combineren van design patterns. Dit combineren van design patterns wordt ook compound patterns genoemd.

```
public class DocumentFactory
{
    private static DocumentFactory deFactory;
    private DocumentFactory() { }
    public static DocumentFactory GetInstance()
    {
        if (deFactory == null)
        {
            deFactory = new DocumentFactory();
        }
        return deFactory;
    }
    public Document Open(string bestandsnaam)
    {
        string extensie = bestandsnaam.Substring(bestandsnaam.Length - 4);
        switch (extensie)
        {
            case "docx":
                return new Tekst(bestandsnaam);
            case "xlsx":
                return new Rekenblad(bestandsnaam);
            case "pptx":
                return new Presentatie(bestandsnaam);
            default:
                throw new Exception("Verkeerde extensie.");
        }
    }
}
```

Je gebruikt de classes nu als volgt:

```
static void Main(string[] args)
{
    DocumentFactory factory = DocumentFactory.GetInstance();
    Document document = factory.Open("liedje.docx");
    document.PrintPreview();
    document.Print();
}
```

3.6 Oefeningen

Maak de onderstaande oefening in de takenbundel :

Simple factory

4 BUILDER

4.1 Categorie

Creational design pattern

4.2 Probleem

Je moet veel parameters meegeven om een object aan te maken. Voorbeeld:

```
Inwoner inwoner = new Inwoner("Olivier", "Gerard", 1, 3, true, false);
```

Deze constructor heeft veel parameters, wat de leesbaarheid niet bevordert. Er wordt aangeraden dat een constructor maximum vier parameters heeft. De betekenis van de parameterwaarden is ook niet duidelijk: is de voornaam van de inwoner Olivier of Gerard ? Wat is de betekenis van de waarde 1, de waarde 3, de waarde true en de waarde false ? Je weet dit slechts als je de source of de documentatie bij het programma inziet.

En wat als je niet alle parameterwaarden kent? Je zou aan constructor overloading kunnen doen en een extra constructor maken met enkel de parameters voornaam en familienaam, ééntje met voornaam, familienaam en aantalKinderen, …

4.3 Oplossing

Het eerste probleem, de onduidelijkheid over de betekenis van de parameters zou je nog kunnen oplossen met *named parameters*.

```
Inwoner inwoner = new Inwoner(voornaam: "Olivier",
                              familienaam : "Gerard",
                              aantalKinderen : 1,
                              aantalKeerVerhuisd : 3,
                              gehuwd : true,
                              gescheiden : false);
```

Het Builder design pattern biedt ook een oplossing voor het 2de probleem.

Je gebruikt een Builder-class die een class stap voor stap opbouwt.

Eigenlijk heb je al eens met een Builder-class gewerkt : de StringBuilder.

Beschouw volgende code :

```
StringBuilder builder = new StringBuilder("builder-");          (1)
builder.Append("test");                                         (2)
builder.Replace('e', 'o').Remove(0, 1);                         (3)
String woord = builder.ToString();                              (4)
```

Je gebruikt hier een class StringBuilder om de stringwaarde van een String samen te stellen. Je doet dit eerst via een constructor (1). Daarna voeg je achteraan tekst toe via een Append-method (2). Tenslotte gebruik je de method Replace() om de ‘e’-s te vervangen door ‘o’-s en de method Remove() om het eerste teken weg te laten (3). Als je de documentatie van deze method er op zou naslaan dan zal je zien dat het resultaat van deze methods telkens opnieuw een StringBuilder-object is. Daardoor kan je aan method-chaining doen. In de laatste instructie gebruiken we een StringBuilder-method om een String-object te maken (4).

We passen deze manier van werken toe op ons voorbeeld. We maken een extra class `InwonerBuilder` die de class `Inwoner` stap per stap opbouwt en uiteindelijk een Inwoner class oplevert :

```
InwonerBuilder builder = new InwonerBuilder();
Inwoner inwoner = builder.metVoornaam("Olivier")
                         .metFamilienaam("Gerard")
                         .metAantalKinderen(1)
```

```
                        .metAantalKeerVerhuisd(3)
                        .metGehuwd(true)
                        .metGescheiden(false)
                        .maakInwoner();
```

Deze werkwijze met method chaining wordt ook een fluent (vloeiende) interface genoemd.

De class Inwoner :

```
public class Inwoner
{
    public string Voornaam { get; set; }
    public string Familienaam { get; set; }
    public int AantalKinderen { get; set; }
    public int AantalKeerVerhuisd { get; set; }
    public bool Gehuwd { get; set; }
    public bool Gescheiden { get; set; }

    public Inwoner(string voornaam, string familienaam, int aantalKinderen,
        int aantalKeerVerhuisd, bool gehuwd, bool gescheiden)
    {
        Voornaam = voornaam;
        Familienaam = familienaam;
        AantalKinderen = aantalKinderen;
        AantalKeerVerhuisd = aantalKeerVerhuisd;
        Gehuwd = gehuwd;
        Gescheiden = gescheiden;
    }

    public override String ToString()
    {
        return Voornaam + ' ' + Familienaam;
    }
}
```

De builder class :

```
public class InwonerBuilder
{
    public string Voornaam { get; set; }
    public string Familienaam { get; set; }
    public int AantalKinderen { get; set; }
    public int AantalKeerVerhuisd { get; set; }
    public bool Gehuwd { get; set; }
    public bool Gescheiden { get; set; }
    public InwonerBuilder metVoornaam(String voornaam)
    {
        Voornaam = voornaam;
        return this;
    }
    public InwonerBuilder metFamilienaam(String familienaam)
    {
        Familienaam = familienaam;
        return this;
    }
    public InwonerBuilder metAantalKinderen(int aantalKinderen)
    {
        AantalKinderen = aantalKinderen;
        return this;
    }
    public InwonerBuilder metAantalKeerVerhuisd(int aantalKeerVerhuisd)
    {
        AantalKeerVerhuisd = aantalKeerVerhuisd;
        return this;
    }
```

```
        public InwonerBuilder metGehuwd(bool gehuwd)
        {
            Gehuwd = gehuwd;
            return this;
        }
        public InwonerBuilder metGescheiden(bool gescheiden)
        {
            Gescheiden = gescheiden;
            return this;
        }
        public Inwoner maakInwoner()
        {
            return new Inwoner(Voornaam, Familienaam, AantalKinderen,
              AantalKeerVerhuisd, Gehuwd, Gescheiden);
        }
    }
```

(1) Iedere method van een builder geeft dezelfde builder terug. Dit laat de fluent interface toe: `.metVoornaam("Olivier").metFamilienaam("Gerard")`

Het hoofdprogramma :

```
        static void Main(string[] args)
        {
            InwonerBuilder builder = new InwonerBuilder();
            Inwoner inwoner = builder.metVoornaam("Olivier")
                                     .metFamilienaam("Gerard")
                                     .metAantalKinderen(1)
                                     .metAantalKeerVerhuisd(3)
                                     .metGehuwd(true)
                                     .metGescheiden(false)
                                     .maakInwoner();
            Console.WriteLine(inwoner);
        }
```

Je kan in de main nog altijd de onleesbare `Inwoner` constructor oproepen in plaats van de `InwonerBuilder` te gebruiken. Je lost dit met volgende stappen op:

1. Je kopieert in de source `InwonerBuilder` de regels `public class InwonerBuilder` en bijhorend code block op het klembord.
2. Je plakt de inhoud van het klembord voor de sluit accolade van de class `Inwoner`. `InwonerBuilder` wordt dus een nested class van `Inwoner`.
3. Je maakt de constructor van `Inwoner` private, zodat enkel de nested class `InwonerBuilder` deze kan oproepen.
4. Je wijzigt in de class Main twee keer `InwonerBuilder` naar `Inwoner.InwonerBuilder`.
5. Je verwijdert de source `InwonerBuilder.cs`

4.4 Oefeningen

Maak de onderstaande oefening in de takenbundel :

Builder

5 FACADE

5.1 Categorie

Structural Design Pattern

5.2 Probleem

De gebruiker van een verzameling classes moet een moeilijk algoritme schrijven om die classes aan te spreken. Dit patroon is zeker interessant als één ontwikkelaar de verzameling classes schreef en een andere ontwikkelaar deze moet gebruiken en er een moeilijk algoritme moet op schrijven.

5.3 Voorbeeld

Om een lening toe te kennen, moeten drie voorwaarden voldaan zijn:

- De persoon moet voldoende beroepsinkomsten hebben (€ 2500).
- Het saldo op de rekening van de persoon mag niet negatief zijn.
- De persoon mag geen andere leningen lopende hebben.

Zonder façade is het de *gebruiker* van de classes `Loon`, `Rekening` en `Leningen` die bepaalt dat deze drie voorwaarden samen moeten voldaan worden. Het façade design pattern verplaatst dit algoritme naar de binnenkant van een extra façade class (`LeningVerstrekker`).
De gebruiker roept vanaf dan een eenvoudige method op van die façade class.

5.4 Zonder façade

De class Beroepsinkomsten :

```
public class Beroepsinkomsten
{
    public decimal MaandWedde { get; }

    public Beroepsinkomsten(decimal maandWedde)
    {
        this.MaandWedde = maandWedde;
    }
}
```

De class Rekening :

```
public class Rekening
{
    private decimal saldo;
    public decimal Saldo
    {
        get { return saldo; }
        private set { saldo = value; }
    }
    public Rekening()
    {
        Saldo = decimal.Zero;
    }
    public void Storten(decimal bedrag)
    {
        Saldo += bedrag;
    }
}
```

De class Lening :

```
public class Lening
{
    public decimal Bedrag { get; }
```

```
        public Lening(decimal bedrag)
        {
            Bedrag = bedrag;
        }
    }
```

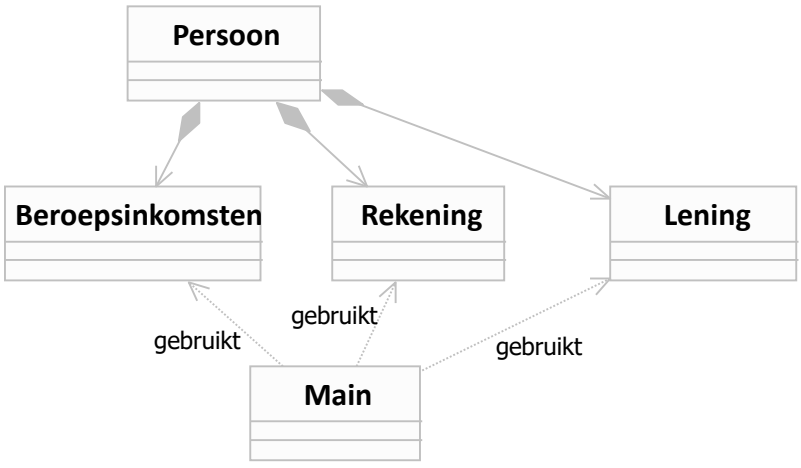
De class Persoon :

```
    public class Persoon
    {
        public Beroepsinkomsten Beroepsinkomsten { get; }
        public Rekening Rekening { get; }
        public List<Lening> Leningen { get; }

        public Persoon(Beroepsinkomsten beroepsinkomsten,
                       Rekening rekening)
        {
            Beroepsinkomsten = beroepsinkomsten;
            Rekening = rekening;
            Leningen = new List<Lening>();
        }

        public void AddLening(Lening lening)
        {
            Leningen.Add(lening);
        }
    }
}
```

Het hoofdprogramma :

```
static void Main(string[] args)
{
    Persoon persoon = new Persoon(
        new Beroepsinkomsten(3000m),
        new Rekening());
    if (persoon.Beroepsinkomsten.MaandWedde > 2500m &&
        persoon.Rekening.Saldo >= decimal.Zero &&
        persoon.Leningen.Count == 0)
        Console.WriteLine("Lening goedgekeurd");
    else
        Console.WriteLine("Lening afgekeurd");
}
```

De Main, gebruiker van de classes, bevat het moeilijk algoritme dat bepaalt of een rekening toegekend wordt.



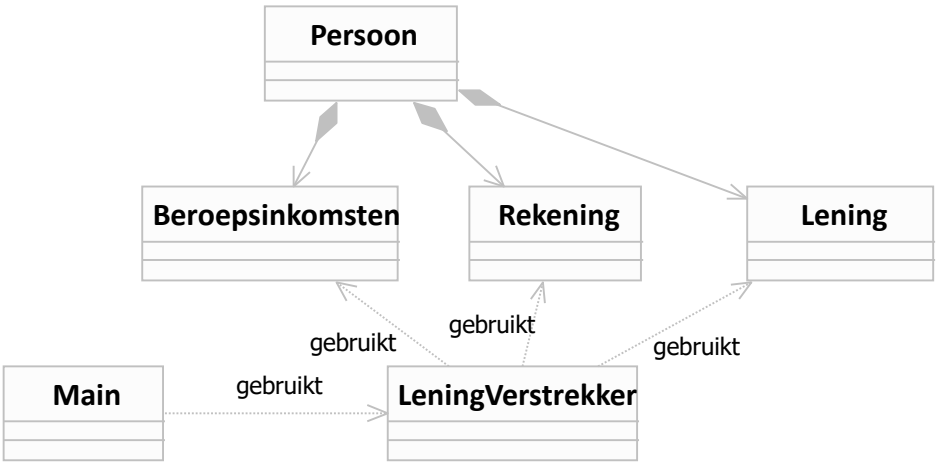
5.5 Oplossing met façade

Een class LeningVerstrekker speelt de rol van façade en heeft het moeilijk algoritme in zich :

```
public class LeningVerstrekker
{
    public bool IsLeningGoedgekeurd(Persoon persoon)
    {
        return persoon.Beroepsinkomsten.MaandWedde > 2500m &&
            persoon.Rekening.Saldo >= decimal.Zero &&
            persoon.Leningen.Count == 0;
    }
}
```

De class Main, gebruiker van de classes, wordt nu eenvoudig:

```
static void Main(string[] args)
{
    Persoon persoon = new Persoon(
        new Beroepsinkomsten(3000m),
        new Rekening());
    LeningVerstrekker verstrekker = new LeningVerstrekker();
    if (verstrekker.IsLeningGoedgekeurd(persoon))
        Console.WriteLine("Lening goedgekeurd");
    else
        Console.WriteLine("Lening afgekeurd");
}
```



5.6 Oefeningen

Maak de onderstaande oefening in de takenbundel :

Façade

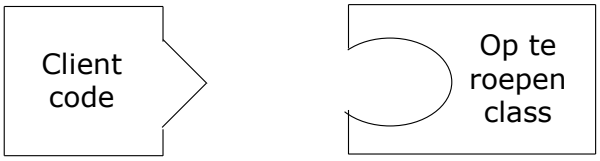
6 ADAPTER

6.1 Categorie

Structural design pattern

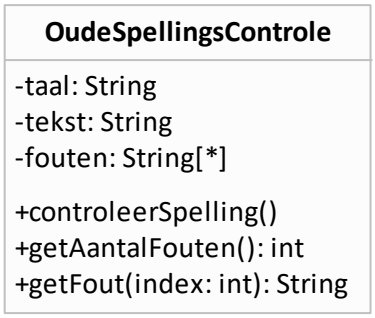
6.2 Probleem

Je wil een class aanspreken, maar die heeft niet de methods die jij verwacht aan te spreken.



6.3 Voorbeeld

Je gebruikte vroeger een spellingscontrole library. De aan te spreken class was als volgt:




```
public class OudeSpellingsControle
{
    public string Taal { get; set; }
    public string Tekst { get; set; }
    public string[] Fouten { get; set; }
    public void ControleerSpelling()
    {
        Fouten = new string[] { "onmidelijk", "paralelogram" };
    }
    public int GetAantalFouten()
    {
        return Fouten.Length;
    }
    public string GetFout(int index)
    {
        return Fouten[index];
    }
}
```

Je gebruikte deze library als volgt:

```
OudeSpellingsControle controle = new OudeSpellingsControle();
controle.Taal = "nl";
controle.Tekst = "Ik kom onmidelijk met een paralelogram.";
controle.ControleerSpelling();
int aantalFouten = controle.GetAantalFouten();
for (int index = 0; index != aantalFouten; index++)
{
    Console.WriteLine(controle.GetFout(index));
}
```

De firma die deze library onderhoudt gaat echter failliet.
Vanaf nu moet je een andere spellingscontrole library gebruiken.
De aan te spreken class heeft andere methods dan de oude library :

NieuweSpellingsControle
-taal: String -tekst: String
<<constructor>>+NieuweSpellingsControle(taal: String, tekst: String) +controleer():String[*]()

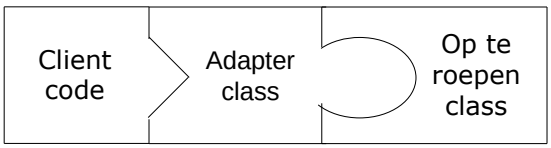
```
public class NieuweSpellingsControle
{
    public string Taal { get; set; }
    public string Tekst { get; set; }
    public NieuweSpellingsControle(string taal, string tekst)
    {
        Taal = taal;
        Tekst = tekst;
    }
    public string[] Controleer()
    {
        return new string[] { "onmidelijk", "paralelogram" };
    }
}
```

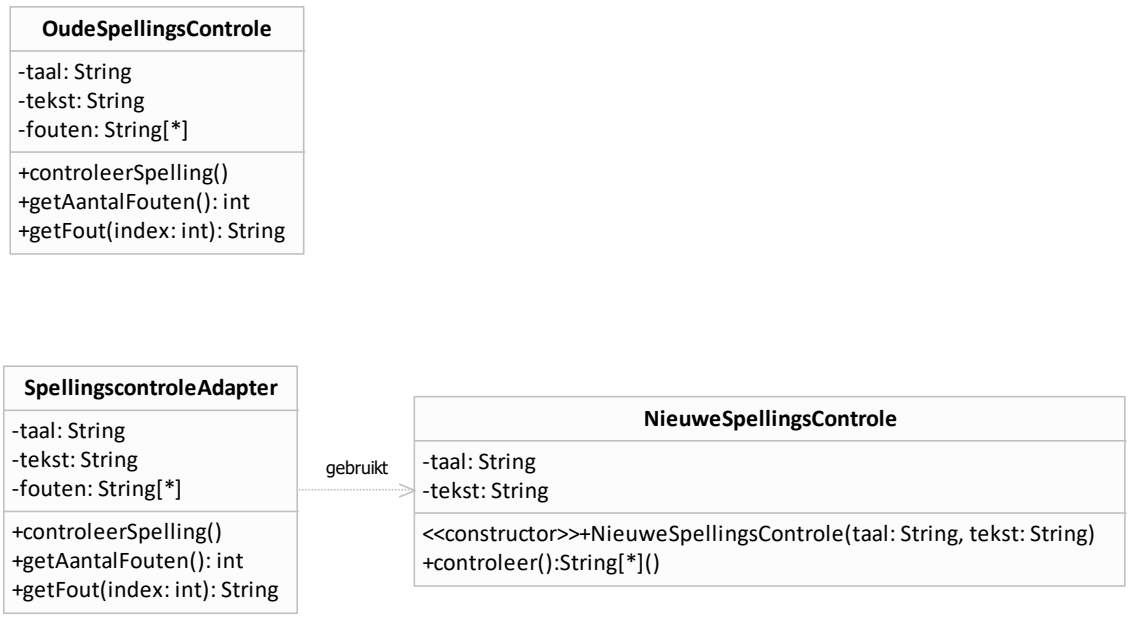
Zonder het adapter design pattern moet je overal in je code waar je de oude library opriep meerdere regels aanpassen om de nieuwe library op te roepen :

```
NieuweSpellingsControle controle2 = new NieuweSpellingsControle("nl",
    "Ik kom onmidelijk met een paralelogram.");
foreach (string fout in controle2.Controleer())
{
    Console.WriteLine(fout);
}
```

6.4 Oplossing

Je maakt een adapter class. Deze heeft dezelfde method declaraties als de oude library.
De adapter class vertaalt de oproepen van deze methods naar oproepen naar de nieuwe library






```
public class SpellingsControleAdapter
{
    public string Taal { get; set; }
    public string Tekst { get; set; }
    public string[] Fouten { get; set; }
    public void ControleerSpelling()
    {
        NieuweSpellingsControle controle =
            new NieuweSpellingsControle(Taal, Tekst);
        Fouten = controle.Controleer();
    }
    public int GetAantalFouten()
    {
        return Fouten.Length;
    }
    public string GetFout(int index)
    {
        return Fouten[index];
    }
}
```

Nu moet je op de plaatsen waar je de spellingscontrole gebruikt enkel de eerste regel aanpassen:

```
SpellingsControleAdapter controle = new SpellingsControleAdapter();
controle.Taal = "nl";
controle.Tekst = "Ik kom onmidelijk met een paralelogram.";
controle.ControleerSpelling();
int aantalFouten = controle.GetAantalFouten();
for (int index = 0; index != aantalFouten; index++)
{
    Console.WriteLine(controle.GetFout(index));
}
```

Adapters bestaan ook in de werkelijkheid. Wanneer de pinnen van een elektrisch apparaat niet overeenstemmen met de gaten in een stopcontact steek je tussen het elektrisch apparaat en het stopcontact een adapter om dit probleem op te lossen:



6.5 Oefeningen

Maak de onderstaande oefening in de takenbundel :

Adapter

7 COMPOSITE

7.1 Categorie

Structural design pattern

7.2 Probleem

Sommige gegevensverzamelingen hebben een boomstructuur. Hoe kan je op dezelfde manier de elementen benaderen die nog vertakkingen hebben als de elementen die geen vertakkingen hebben?

Een harde schijf is een voorbeeld van zo'n boomstructuur. De harde schijf bestaat uit een root directory. Deze bestaat op zijn beurt uit directory's en bestanden. Ieder van die directory's bestaat weer uit directory's en bestanden. Het probleem is: hoe kan je deze twee soorten (directory's en bestanden) benaderen om er eenzelfde handeling (bvb. verwijderen) op uit te voeren?
Als je een directory verwijdert, moet je alle bestanden in die directory verwijderen, maar ook alle subdirectory's. Om zo'n subdirectory te verwijderen, moet je ook daar alle subdirectory's en bestanden verwijderen, …

7.3 Voorbeeld

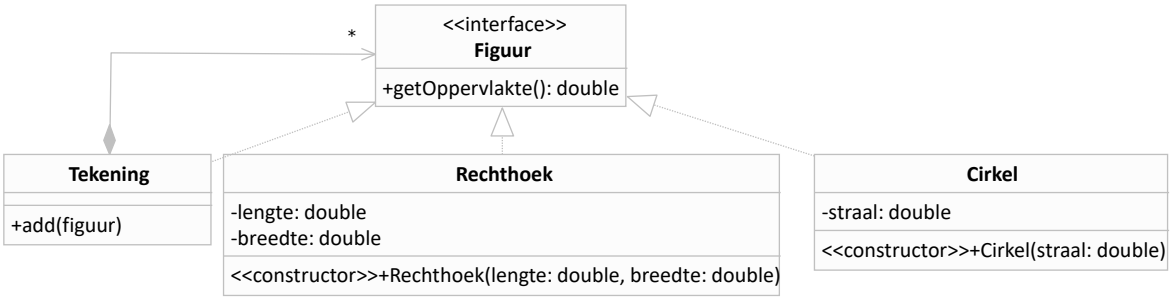
Een tekening in een cad-cam programma bestaat uit figuren (rechthoeken, cirkels, …).
Maar één van de figuren kan op zijn beurt weer een tekening zijn (tekening in tekening).
Deze tweede tekening bestaat terug uit figuren en eventuele tekeningen.

Om de totale oppervlakte te weten van alle figuren in de tekening, moet je de som maken van de figuren in die tekening plus de som van alle figuren in tekeningen van die tekening …

Het composite design pattern laat dit op een elegante manier toe.

7.4 Oplossing

Je benoemt het onderdeel dat vertakkingen heeft (tekening), en je maakt er een class van.
Je benoemt de onderdelen die geen vertakkingen hebben (rechthoek, cirkel) en je maakt er classes van. Al deze classes implementeren een gemeenschappelijke interface class (figuur).
Je voegt aan deze interface method declaraties toe voor alle methods die je vindt in alle subclasses.



Bemerk dat een tekening (via composition) terug uit figuren bestaat.

```
public interface IFiguur
{
    double GetOppervlakte();
}

public class Rechthoek : IFiguur
{
    public double Lengte { get; set; }
    public double Breedte { get; set; }
    public Rechthoek(double lengte, double breedte)
    {
        Lengte = lengte;
        Breedte = breedte;
    }
```

```
        public double GetOppervlakte()
        {
            return Lengte * Breedte;
        }
    }

    public class Cirkel : IFiguur
    {
        public double Straal { get; set; }
        public Cirkel(double straal)
        {
            Straal = straal;
        }
        public double GetOppervlakte()
        {
            return Straal * Straal * Math.PI;
        }
    }

    public class Tekening : IFiguur
    {
        public List<IFiguur> Figuren { get; set; } = new List<IFiguur>();
        public void Add(IFiguur figuur)
        {
            Figuren.Add(figuur);
        }
        public double GetOppervlakte()
        {
            double oppervlakte = 0;
            foreach (IFiguur figuur in Figuren)                                    (1)
                oppervlakte += figuur.GetOppervlakte();
            return oppervlakte;
        }
    }
```

(1) Je overloopt alle figuren in de huidige figuur. Je maakt daarbij geen onderscheid tussen tekeningen (figuren die op zich weer figuren bevatten), rechthoeken of cirkels.

Je gebruikt deze classes als volgt:

```
    static void Main(string[] args)
    {
        Tekening tekening = new Tekening();
        tekening.Add(new Rechthoek(2, 1));
        tekening.Add(new Cirkel(3));
        Tekening subTekening = new Tekening();
        subTekening.Add(new Rechthoek(3, 2));
        subTekening.Add(new Cirkel(4));
        tekening.Add(subTekening);
        Console.WriteLine(tekening.GetOppervlakte());
    }
```

7.5 Oefeningen

Maak de onderstaande oefening in de takenbundel :

Composite

8 DECORATOR

8.1 Categorie

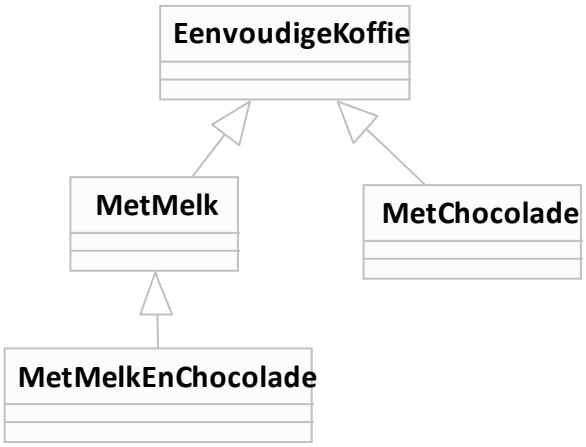
Structural design pattern

8.2 Probleem

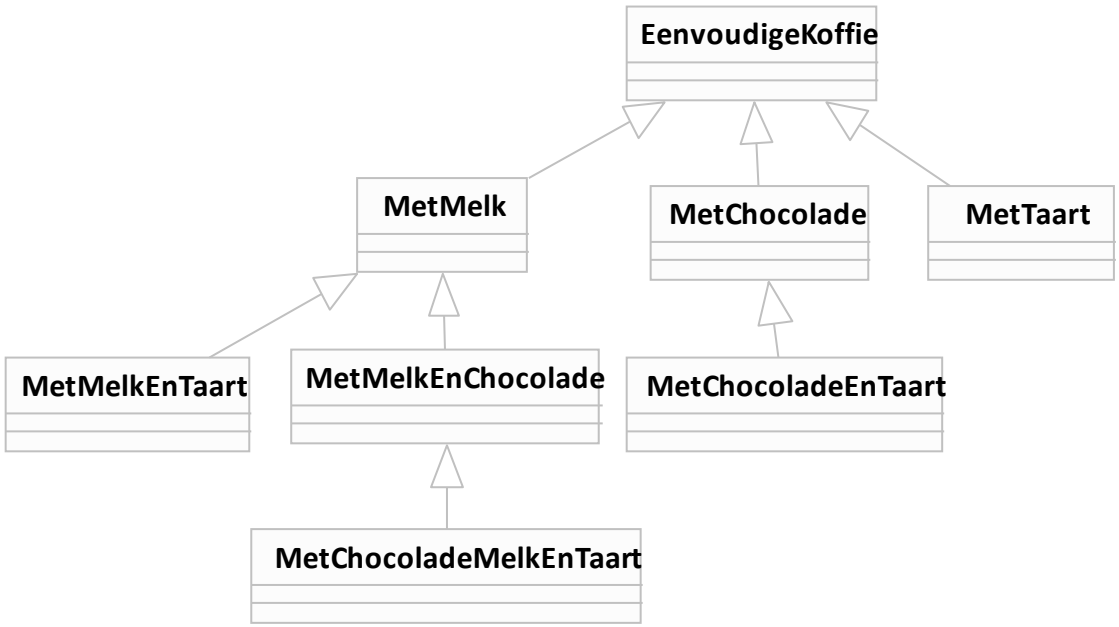
Hoe kan je, op een dynamische manier, verantwoordelijkheden toevoegen aan een class ?

Voorbeeld: een eenvoudige koffie heeft een prijs en een bereidingswijze. Je wil aan deze koffie eventueel melk toevoegen. Dit wijzigt de prijs en de bereidingswijze. Je kan aan een koffie ook chocolade toevoegen. Ook dit wijzigt de prijs en de bereidingswijze. Je kan aan een koffie ook én melk én chocolade toevoegen. Ook dit wijzigt de prijs en de bereidingswijze.

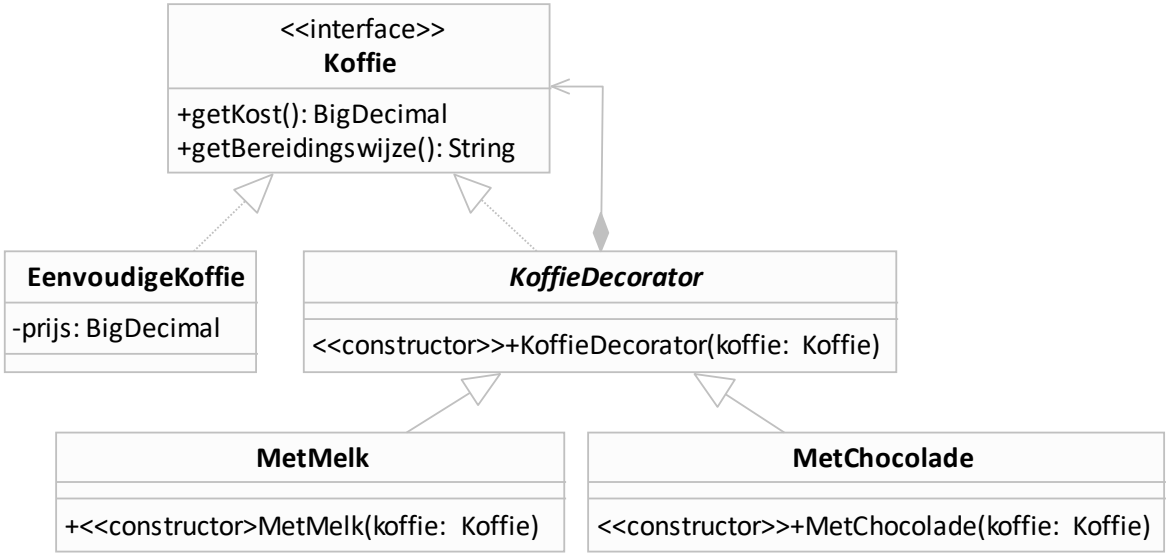
Als je dit probeert op te lossen met inheritance, krijg je veel classes:



Zeker als er daarna nog een taart kan aan toegevoegd worden, zou je nog meer classes krijgen:



8.3 Oplossing



Je vertrekt vanaf een EenvoudigeKoffie: `new EenvoudigeKoffie()`. Als de klant geen melk en geen chocolade wenst, heb je met dit object genoeg om de kost en de bereidingswijze te bepalen.

Als de klant melk wenst, "decoreer" je de EenvoudigeKoffie met MetMelk:
`new MetMelk(new EenvoudigeKoffie());`.
De kost wordt de kost van de koffie plus de kost van de melk.

Als de klant chocolade wenst, "decoreer" je de `EenvoudigeKoffie` met `MetChocolade`:
`new MetChocolade(new EenvoudigeKoffie());`.
De kost wordt de kost van de koffie plus de kost van de chocolade.

Als de klant melk én chocolade wenst, "decoreer" je de `EenvoudigeKoffie` met `MetMelk` én met `MetChocolade` : `new MetChocolade(new MetMelk(new EenvoudigeKoffie()));`.
De kost wordt de kost van de koffie plus de kost van de melk plus de kost van de chocolade.

Hetgeen `MetMelk` en `MetChocolade` gemeen hebben, plaats je in een gemeenschappelijke base class KoffieDecorator

Alle "decorators" (`MetMelk` en `MetChocolade`) en `EenvoudigeKoffie` implementeren de interface `Koffie`: je kan van alle "decorators" én van `EenvoudigeKoffie` de kost berekenen en de bereidingswijze bepalen.

Er is ook een composition (◆——>) tussen `KoffieDecorator` en `Koffie`: elke "decorator" onthoudt in zich de koffie die hij "decoreert".
Dit is een `EenvoudigeKoffie` bij `new MetMelk(new EenvoudigeKoffie());`.
Dit is een `MetMelk` bij `new MetChocolade(new MetMelk(new EenvoudigeKoffie()));`.

```
public interface IKoffie
{
    decimal GetKost();
    string GetBereidingswijze();
}

public class EenvoudigeKoffie : IKoffie
{
    public string GetBereidingswijze()
    {
        return "maal de koffiebonen, laat kokend water over het poeder lopen";
    }
```

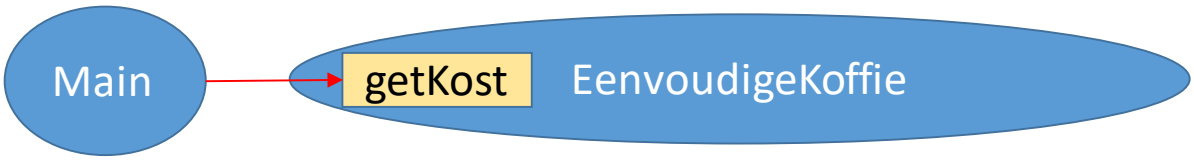
```
    public decimal GetKost()
    {
        return 3m;
    }
}

public abstract class KoffieDecorator : IKoffie
{
    protected IKoffie GedecoreerdeKoffie { get; set; }
    public KoffieDecorator(IKoffie koffie)
    {
        GedecoreerdeKoffie = koffie;
    }
    public abstract decimal GetKost();
    public abstract string GetBereidingswijze();
}

public class MetMelk : KoffieDecorator
{
    public MetMelk(IKoffie gedecoreerdeKoffie)                              (1)
        : base(gedecoreerdeKoffie)                                          (2)
    {
    }
    public override decimal GetKost()
    {
        return base.GedecoreerdeKoffie.GetKost() + 1m;                      (3)
    }
    public override string GetBereidingswijze()
    {
        return base.GedecoreerdeKoffie.GetBereidingswijze() +               (4)
            ", warm de melk, voeg de melk toe";
    }
}

public class MetChocolade : KoffieDecorator
{
    public MetChocolade(IKoffie gedecoreerdeKoffie)                         (5)
        : base(gedecoreerdeKoffie)
    {
    }
    public override decimal GetKost()
    {
        return base.GedecoreerdeKoffie.GetKost() + 2m;                      (6)
    }
    public override string GetBereidingswijze()
    {
        return base.GedecoreerdeKoffie.GetBereidingswijze() +               (7)
            ", schilfer de chocolade, voeg de schilfers toe";
    }
}

static void Main(string[] args)
{
    IKoffie eenvoudig = new EenvoudigeKoffie();
    Console.WriteLine("Eenvoudige koffie");
    Console.WriteLine("Kostprijs :  {0} euro", eenvoudig.GetKost());
    Console.WriteLine("Bereiding : " + eenvoudig.GetBereidingswijze());
    Console.WriteLine();
    IKoffie metMelk = new MetMelk(new EenvoudigeKoffie());
    Console.WriteLine("Koffie met melk");
    Console.WriteLine("Kostprijs :  {0} euro", metMelk.GetKost());
```

```
        Console.WriteLine("Bereiding : " + metMelk.GetBereidingswijze());
        Console.WriteLine();
        IKoffie metChocolade = new MetChocolade(new EenvoudigeKoffie());
        Console.WriteLine("Koffie met chocolade");
        Console.WriteLine("Kostprijs :  {0} euro", metChocolade.GetKost());
        Console.WriteLine("Bereiding : " + metChocolade.GetBereidingswijze());
        Console.WriteLine();
        IKoffie metMelkEnChocolade
            = new MetChocolade(new MetMelk(new EenvoudigeKoffie()));
        Console.WriteLine("Koffie met melk en chocolade");
        Console.WriteLine("Kostprijs :  {0} euro", metMelkEnChocolade.GetKost());
        Console.WriteLine("Bereiding : " + metMelkEnChocolade.GetBereidingswijze());
    }
```

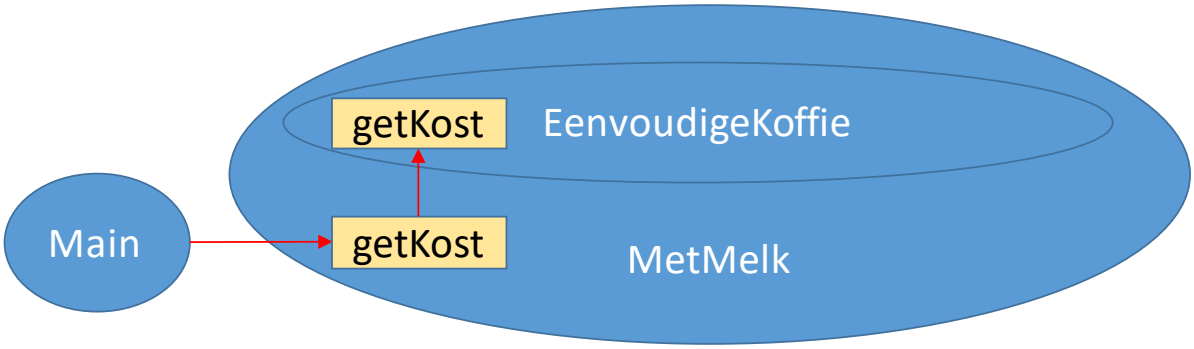
(1) `MetMelk` decoreert een `EenvoudigeKoffie` die (onder de gedaante van `Koffie`) als constructor parameter binnenkomt.
(2) Je geeft deze parameter door aan de base class constructor. Deze onthoudt de parameter in een protected variabele.
(3) De kost van `MetMelk` is de kost van de koffie die hij decoreert plus de kost van de melk zelf: één euro.
(4) De bereidingswize van `MetMelk` is de bereidingswijze van de koffie die hij decoreert plus de bereidingswijze van de melk zelf.
(5) `MetChocolade` decoreert een `EenvoudigeKoffie` of een `MetMelk` die (onder de gedaante van `Koffie`) als constructor parameter binnenkomt.
(6) De kost van `MetChocolade` is de kost van de koffie die hij decoreert plus de kost van de chocolade zelf: twee euro.
(7) De bereidingswize van `MetChocolade` is de bereidingswijze van de koffie die hij decoreert plus de bereidingswijze van de chocolade zelf.

Grafische voorstellingen:

Je kan de kost vragen van een EenvoudigeKoffie, zonder decorator:

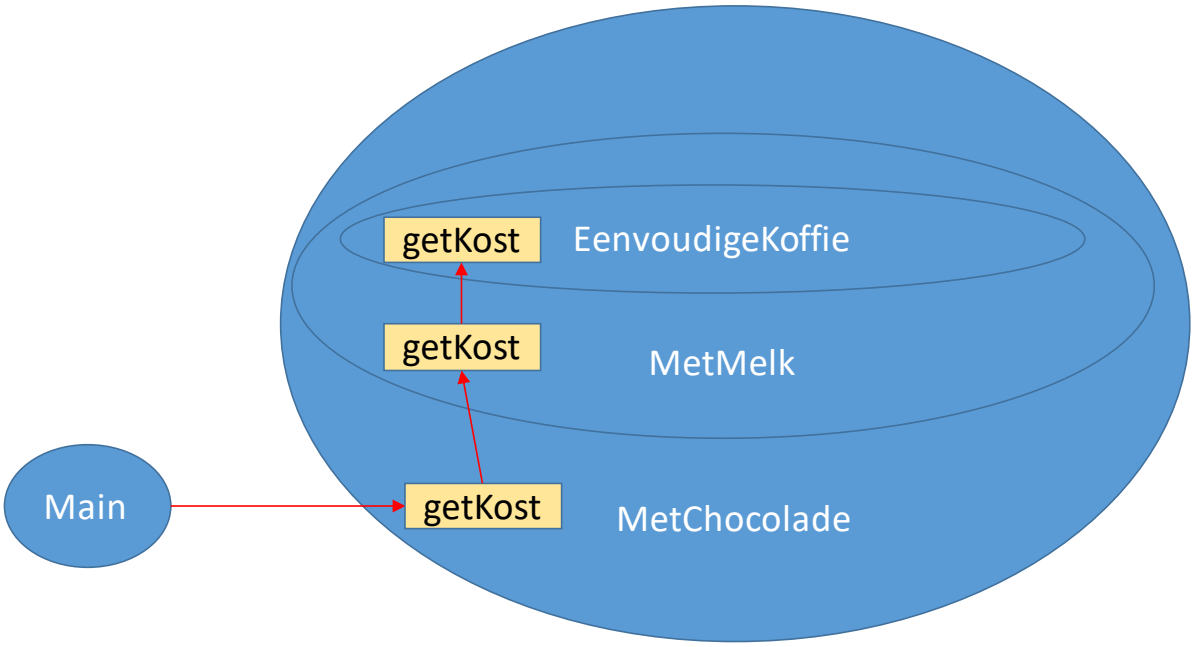


Je kan ook de kost vragen van een `EenvoudigeKoffie`, gedecoreerd met een `MetMelk`:



De `getKost` method van `MetMelk` roept dan de `getKost` method van `EenvoudigeKoffie` op.

Je kan ook de kost opvragen van een `EenvoudigeKoffie`, gedecoreerd met een `MetMelk` én met een `MetChocolade`:



De `getKost` method van `MetChocolade` roept dan de `getKost` method van `MetMelk` op.
Deze roept op zijn beurt de `getKost` van EenvoudigeKoffie op.

De decoratie is nu hard gecodeerd in de Main.
Je kan de decoratie ook dynamisch doen, bijvoorbeeld op basis van invoer van de gebruiker:

```
static void Main(string[] args)
{
    IKoffie koffie = new EenvoudigeKoffie();
    Console.WriteLine("Melk (j/n):");
    var invoer = Console.ReadLine();
    if (invoer.ToUpper().Equals("J"))
    {
        koffie = new MetMelk(koffie);
    }
    Console.WriteLine("Chocolade (j/n):");
    invoer = Console.ReadLine();
    if (invoer.ToUpper().Equals("J"))
    {
        koffie = new MetChocolade(koffie);
    }
    Console.WriteLine("Kostprijs :  {0} euro", koffie.GetKost());
    Console.WriteLine("Bereiding : " + koffie.GetBereidingswijze());
}
```

8.4 Oefeningen

Maak de onderstaande oefening in de takenbundel :

Decorator

9 STRATEGY

9.1 Categorie

Behavioral design pattern

9.2 Probleem

Je hebt voor een bepaald algoritme meerdere implementaties waaruit je moet kiezen.
Je wil deze implementaties niet in de class zelf uitwerken, maar in de *gebruiker* van de class (bvb. de class Main). Op die manier moet je de class niet wijzigen als er nog een algoritme bijkomt. Je volgt hiermee het principe *A class should be closed for modification*.

9.3 Voorbeeld

Een persoon heeft een voornaam, een familienaam en een titel. Er bestaan meerdere algoritmes om een briefhoofding voor de persoon te maken. Een eerste algoritme geeft een formele aanspreking (`Geachte heer Smits`). Een tweede algoritme geeft een informele aanspreking (`Dag Jean`). Je wil deze algoritmes niet hard coderen in de class Persoon. Zo kan je later nog algoritmes toevoegen.

9.4 Oplossing

Je beschrijft het algoritme in een delegate.

Je maakt een lambda die beantwoordt aan het delegate type en je geeft het mee als parameter van de method in de class Persoon die dit algoritme gebruikt.

```
namespace Strategy
{
    public delegate string Aanspreking(Persoon persoon);
    public class Persoon
    {
        public string Voornaam { get; set; }
        public string Familienaam { get; set; }
        public string Titel { get; set; }

        public Persoon(string voornaam, string familienaam, string titel)
        {
            Voornaam = voornaam;
            Familienaam = familienaam;
            Titel = titel;
        }

        public string MaakBriefHoofding(Aanspreking algoritme)
        {
            StringBuilder builder = new StringBuilder();
            builder.Append("Brussel,");
            DateTime vandaag = DateTime.Now;
            builder.Append(vandaag.ToString("d/M/yyyy"));
            builder.Append("\n");
            builder.Append(algoritme(this));
            builder.Append("\n");
            return builder.ToString();
        }
    }
}
```

```
namespace Strategy
{
    class Program
    {
        static void Main(string[] args)
        {
            Persoon[] personen = new Persoon[]  {
                new Persoon("Jean", "Smits", "heer"),
                new Persoon("Jeanine", "Desmet", "mevrouw")
            };

            // briefhoofdingen met informele aansprekingen:
            foreach (Persoon persoon in personen)
            {
                Console.WriteLine(persoon.MaakBriefHoofding((pers) =>
                    "Dag " + pers.Voornaam));
            }

            // briefhoofdingen met formele aansprekingen:
            foreach (Persoon persoon in personen)
            {
                Console.WriteLine(persoon.MaakBriefHoofding((pers) =>
                    "Geachte " + pers.Titel + ' ' + pers.Familienaam));
            }
        }
    }
}
```

Strategy: zie takenbundel

10 COLOFON

Domeinexpertisemanager: Jean Smits

Moduleverantwoordelijke: Hans Desmet

Medewerkers: Hans Desmet
Steven Lucas

Versie: 11/1/2018