Table of Contents

Software Engineering

Inleiding

Software ontwikkelaars worden geconfronteerd met ontwerpproblemen. Professionals zullen echter merken dat bepaalde soorten van ontwerpproblemen steeds terugkomen. Eénmaal je een probleem herkent als een variant van een probleem dat je vroeger al eens hebt opgelost, kan je gebruik maken van de inzichten die je al verworven hebt. Je ziet bepaalde patronen terugkeren.

Wat is nu precies een ontwerppatroon

> Een ontwerppatroon is een standaardoplossing voor een vaak voorkomend ontwerpprobleem. Deze patronen zijn belangrijk omdat ze je de moeite kunnen besparen om telkens opnieuw het warm water uit te vinden. Bovendien heeft elk patroon een eigen naam, wat ervoor zorgt dat het heel eenvoudig wordt om bepaalde complexe ideëen in een oogwenk te communiceren aan een andere programmeur.

GESCHIEDENIS VAN ONTWERPEN

Sinds het begin van het computertijdperk is probleem-oplossend denken ingrijpend veranderd.

PROGRAMMEREN: THE SEQUEL

In het begin programmeerden we met assembly, en was elk programma beperkt tot een honderdtal lijnen. Elke programmeur had zijn eigen stijl volgens intuitie.

PROGRAMMEREN: FLOW BASED DESIGN

Toen de complexiteit toenam, gingen meerdere programmeurs code reviews verrichten bij elkaar en merkte men al dat onderhoud en begrijpen van code niet voor de hand lag. Men trachtte normen op te leggen en ging flow charts maken om programmeurs een goed design te laten maken. Flow charts bleken ook nuttig om programma's eenvoudiger te begrijpen.

PROGRAMMEREN: GESTRUCTUREERD PROGRAMMEREN

Gestructuurd programmeren volgde in de jaren '70. Een gestructureerde code bestaat uit één enkel begin en afsluitpunt en daar tussen een set van modules. Gestructureerde programma's zijn makkelijker te lezen en te begrijpen, te onderhouden en vereisen minder ontwikkel-tijd.

PROGRAMMEREN: OBJECT-GEORIENTEERD DESIGN

Object-georienteerd programmeren gebeurt intuitief en identificeert natuurljike objecten (Hero, vijand, ...) die voorkomen in je probleem. Daarna worden relaties zoals composities, referenties, overerving bepaald. Dit resulteert in hergebruik van code, overzichtelijkere en makkelijk te onderhouden code.

VANDAAG

Door de toenemende concurrentie moet je als programmeur tegenwoordig zeer dynamisch zijn. Ook is de

gemiddelde levensduur van een product drastisch verlaagd. Organisaties moeten snel op marktveranderingen kunnen antwoorden. Ook worden business strategieën snel aangepast wat wil zeggen dat bijvoorbeeld een goed software design zeer belangrijk is om snel op deze veranderingen in te kunnen inspelen. Software moet snel ontwikkeld kunnen worden en staat dicht bij de klant (deze kan al vaak worden betrokken bij de ontwikkeling van gepersonaliseerde software (Agile Development)).

SOLID

S.O.L.I.D. zijn 5 principes die ons helpen om een goede software architectuur te schrijven (door Robert C. Martin)

- S : SRP (Single responsibility principle)
- O : OCP (Open closed principle)
- L : LSP (Liskov substitution principle)
- I : ISP (Interface segregation principle)
- D : DIP (Dependency inversion principle)

Single Responsibility Principle

> Een klasse heeft slechts 1 bestaandsreden en kan maar 1 reden hebben om te veranderen

Eigenschappen van SRP zijn:

- coupling
- cohesion

> Cohesion: wat een klasse zou moeten doen. Lage cohesie betekent dat een klasse verschillende zaken doet, en niet gefocust is op de taak die hij zou moeten doen. Terwijl hoge cohesie betekent dat een klasse doet wat hij moet doen, en maar 1 taak uitvoert. Probeer er voor te zorgen dat alle methoden in een klasse betrekking hebben tot 1 doel, maw er een hoge cohesie heerst.
>
> Coupling: Hangt een klasse van nog andere klassen af

Men streeft naar "low coupling" en "high cohesion"

Waarop letten?

Klassen mogen maar een beperkt aantal instantievariabelen hebben. De methoden van deze klasse moeten 1 of meerdere van deze variabelen manipuleren.

Wat bedoelen we met verantwoordelijkheid?

> Een reden tot verandering!

Een voorbeeld van cohesie

Een klasse met hoge cohesie:

```
class EmailMessage
{
    private string sendTo;
```

```
    private string subject;
    private string message;
    public EmailMessage(string to, string subject, string message)
    {
        this.sendTo = to;
        this.subject = subject;
        this.message = message;
    }
    public void SendMessage()
    {
        // send message using sendTo, subject and message
    }
}
```

Een voorbeeld van lage cohesie :

```
class EmailMessage
{
    private string sendTo;
    private string subject;
    private string message;
    private string username;
    public EmailMessage(string to, string subject, string message)
    {
        this.sendTo = to;
        this.subject = subject;
        this.message = message;
    }
    public void SendMessage()
    {
        // send message using sendTo, subject and message
    }
    public void Login(string username, string password)
    {
        this.username = username;
        // code to login
    }
}
```

De Login methode and username klasse variabele heeft niets te maken met de EmailMessage klasse's hoofddoel. Daarom zeggen we dat er een lage cohesie is dat we moeten proberen vermijden.

SRP voorbeeld

```
public class Werknemer
   {
       Database db;
       public Werknemer()
       {
           db = new Database();
       }
       void Insert(){

           try {

               string sql = "insert into werknemers(voornaam,achternaam,stad) values ('Tom', 'Peeters', 'Antwerpen')";
               db.query(sql);
           }
           catch(Exception e)
           {
               //Log error
               System.IO.File.WriteAllText(@"c:\Error.txt", e.ToString());
           }
       }

       void Delete()
```

```
        {

        }

        void Update()
        {

        }
    }
```

De werknemer klasse is nu verantwoordelijk voor CRUD operaties, maar ook voor het loggen van errors. Dus meer dan 1 verantwoordelijkehid. Indien we beslissen om niet meer naar een bestand te loggen, moeten we de klasse aanpassen.

Daarom is het beter om dit als volgt te coderen:

```
public class Werknemer
    {
        Database db;
        FileLogger logger;
        public Werknemer()
        {
            db = new Database();
            logger = new FileLogger();

        }
        void Insert(){

            try {

                string sql = "insert into werknemers(voornaam,achternaam,stad) values ('Tom', 'Peeters', 'Antwerpen')";
                db.query(sql);
            }
            catch(Exception e)
            {
                //Log error
                logger.Log(e.ToString());
            }
        }

    }
```

De klasse FileLogger:

```
public class FileLogger
   {
       public void Log(string error)
       {
           System.IO.File.WriteAllText(@"c:\Error.txt", e.ToString());
       }
   }
```

Single responsibility is niet enkel op klasse maar ook op method niveau.

Single Responsibility op method niveau

Probleemstelling

Er is je gevraagd om software te schrijven voor een online video shop. Het programma berekent en print de rekening van een klant bij onze online shop. Onderstaande paragraaf geeft ons de voorbeeldcode van het

programma. We zullen deze oplossing grondig analyseren en bekijken hoe we de code kunnen verbeteren. Aan het programma wordt meegegeven welke film de klant heeft gehuurd, en voor hoe lang. Daarna wordt de rekening gemaakt – afhankelijk van hoe lang de film gehuurd geweest is, en welk type film (nieuwe release, kinder, gewone). UML notatie:

![movie architectuur][moviearchitectuur.PNG]

Voorbeeld van de MAIN functie

(altijd goed om je architectuur uit te testen door in je main een voorbeeld applicatie te laten draaien)

```
static void Main(string[] args)
        {
            List<Customer> _list = new List<Customer>();

            Customer c = new Customer("Peeters");
            c.AddRental(new Rental(new Movie("Godfather", 0),3));
            _list.Add(c);

            Customer c2 = new Customer("Vandeperre");
            c2.AddRental(new Rental(new Movie("Lion King", 2),2));
            _list.Add(c2);


            Customer c3 = new Customer("Verlinden");
            c3.AddRental(new Rental(new Movie("Rundskop", 1),4));
            _list.Add(c3);


            Customer c4 = new Customer("Dams");
            c4.AddRental(new Rental(new Movie("Top Gun", 0),1));
            _list.Add(c4);

            foreach (Customer cust in _list)
            {
               Console.WriteLine( cust.Statement() );
            }

            Console.ReadLine();


        }
```

Movie klasse .. een simpele klasse

```
public class Movie
    {
        public  const int CHILDRENS = 2;
        public  const int REGULAR = 0;
        public  const int NEW_RELEASE = 1;



        public Movie(string title, int priceCode)
        {
            Title = title;
            PriceCode = priceCode;
         }

          public int PriceCode { get; set; }

   public string Title { get; set; }
    }
```

Rental klasse

Deze klasse stelt voor hoe lang een klant een bepaalde film gehuurd heeft.

```
public class Rental
    {
        private Movie _movie;

        public Rental(Movie movie, int daysRented)
        {
            _movie = movie;
            DaysRented = daysRented;
        }

public int DaysRented { get; set; }


        public Movie GetMovie()
        {
            return _movie;
        }
    }
```

Customer klasse

Deze klasse stelt de klant van de winkel voor

```
public class Customer
    {

        List<Rental> _rentals = new List<Rental>();

        public Customer(string name)
        {
            Name = name;
        }

        public void AddRental(Rental arg)
        {
            _rentals.Add(arg);
        }

        public string Name { get;  }

public string Statement()
        {
            double totalAmount = 0;

            string result = "Rental Record for " + Name + "\n";

            foreach (Rental r in _rentals)
            {
                double thisAmount = 0;
                switch( r.GetMovie().GetPriceCode() )
                {
                    case Movie.REGULAR:
                        thisAmount += 2;
                        if (r.GetDaysRented() > 2)
                        {
                            thisAmount += (r.GetDaysRented() - 2) * 1.5;
                        }
                        break;

                    case Movie.NEW_RELEASE:
                        thisAmount += r.GetDaysRented() * 3;
                        break;
```

```
                    case Movie.CHILDRENS:
                        thisAmount += 1.5;
                        if (r.GetDaysRented() > 3)
                        {
                            thisAmount += (r.GetDaysRented() - 3) * 1.2;
                        }
                        break;
                }



                //Show figures for this rental
result += "\t" + r.GetMovie().GetTitle() + "\t" + thisAmount.ToString() + "\n";
                totalAmount += thisAmount;
            }

            //Add footer lines
            result += "Amount owned is " + totalAmount.ToString() + "\n";

            return result;

        }
}
```

Analyse van onze architectuur

Voor een dergelijke (“simpele”) applicatie is design/architectuur niet zo belangrijk. We zien echter dat dit niet echt object georiënteerde code is, wat een invloed heeft op het gemak waarmee de toepassing kan uitgebreid en veranderd worden.

Enkele bemerkingen: de statement functie in onze Customer klasse is te lang en doet te veel. Veel zaken die we hier in doen, zouden naar andere klasses overgedragen moeten worden.

Ook al werkt ons programma (mooi geschreven code of lelijke code speelt echt geen rol voor een compiler), we moeten ons steeds het volgende afvragen: als in onze applicatie toevoegingen of veranderingen moeten aangebracht worden, moet er “iemand” zijn die dit kan klaar spelen, en een zwak gedesigned systeem is moeilijk te veranderen. Het vergt dan heel wat analysetijd van de programmeur om je programma te doorgronden.

Een voorbeeld van verandering: stel dat je klant vraagt om je rekening ook op een webpagina in HTML af te drukken. Welke impact heeft dit op je programma? Als we naar onze code kijken, merken we dat voor dergelijke vraagstelling het niet mogelijk is code te hergebruiken. Dus moeten we een nieuwe functie maken, die veel gedrag van de reeds bestaande statement functie kopieert. Op zich nog niet echt een probleem, want met wat copy-paste werk kan je de statement functie dupliceren en hernoemen naar htmlstatement() en de result string aanpassen met bijvoorbeeld: result+=”**”blabla**” ..

Maar bedenk eens wat je allemaal moet doen als één regel in het rekening maken verandert? Je moet zowel aanpassingen maken in de statement als de htmlstatement functie, wat gegarandeerd fouten (bugs) zal introduceren!

Nog een andere opmerking. Als de winkel beslist om de classificatie (gewone film, kinder, nieuwe release) te veranderen, maar nog niet zeker is hoe, kan het zijn dat ze je vragen de mogelijke ideeën uit te testen. Dat heeft dan ook een invloed op hoe kosten voor films en huurpunten worden berekend. Als professionele software ontwikkelaar in spe ga ik je reeds verwittigen dat dergelijke veranderingen heel regelmatig voorkomen!

De statement() functie is de plaats waar de veranderingen in classificatie en berekeningen gebeuren. Dus ook niet te vergeten consistente veranderingen te maken in de htmlstatement() functie. Als de berekeningsmethodes steeds complexer worden, zal het met ons design ook steeds moeilijker worden om deze veranderingen door te

voeren.

Wat nu volgt zijn voorstellen om onze software architectuur stap voor stap te veranderen totdat we object georiënteerde code hebben geschreven die ons in staat stelt dergelijke veranderingen op een “makkelijke” manier te realiseren.

Analyseren van de statement functie

Tracht steeds korte functies/methodes te schrijven. Tracht lange functies onder te verdelen in kleinere delen. Kleinere stukken code zijn veel eenvoudiger te onderhouden! Om een functie te verdelen tracht je bij elkaar horende blokken te vinden. Een goede manier is om naar lokale scope variabelen te zoeken. Bijvoorbeeld thisAmount en Rental r, waarbij r niet wordt veranderd, terwijl thisAmount wel. Elke variabele die niet wordt veranderd, kunnen we als argument doorgeven. Indien er variabelen zijn die wel worden veranderd kunnen we, indien er maar 1 is, deze terug retourneren.

We zoeken in onze statement() functie naar deze lijnen code:

```
switch( r.GetMovie().PriceCode )
                {
                    case Movie.REGULAR:
                        thisAmount += 2;
                        if (r.GetDaysRented() > 2)
                        {
                            thisAmount += (r.GetDaysRented() - 2) * 1.5;
                        }
                        break;

                    case Movie.NEW_RELEASE:
                        thisAmount += r.GetDaysRented() * 3;
                        break;

                    case Movie.CHILDRENS:
                        thisAmount += 1.5;
                        if (r.GetDaysRented() > 3)
                        {
                            thisAmount += (r.GetDaysRented() - 3) * 1.5;
                        }
                        break;
                }
```

En maken hiervoor een aparte functie:

```
private double AmountFor(Rental r)
        {
            double thisAmount=0;
            switch (r.GetMovie().GetPriceCode())
            {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if (r.GetDaysRented() > 2)
                    {
                        thisAmount += (r.GetDaysRented() - 2) * 1.5;
                    }
                    break;

                case Movie.NEW_RELEASE:
                    thisAmount += r.GetDaysRented() * 3;
                    break;

                case Movie.CHILDRENS:
                    thisAmount += 1.5;
```

```
                    if (r.GetDaysRented() > 3)
                    {
                        thisAmount += (r.GetDaysRented() - 3) * 1.5;
                    }
                    break;
            }
            return thisAmount;
        }
```

Terwijl we in de statement functie deze verandering maken:

```
foreach (Rental r in _rentals)
{

      double thisAmount = 0;
      thisAmount = AmountFor(r);
…
```

(zie volledige C# code - project SoftwareArchitectuur2)

Analyse van AmountFor functie

Als we naar onze nieuwe AmountFor(Rental r) functie kijken, valt het op dat we hier met Rental data werken, en eigenlijk geen data van de customer klasse gebruiken. In de meeste gevallen moeten functies/methodes in die klasse staan vanwaar ze data gebruiken, dus in dit geval van de Rental klasse.

```
public double GetCharge()
        {
            double result = 0;
            switch (GetMovie().GetPriceCode())
            {
                case Movie.REGULAR:
                    result += 2;
                    if (GetDaysRented() > 2)
                    {
                        result += (GetDaysRented() - 2) * 1.5;
                    }
                    break;

                case Movie.NEW_RELEASE:
                    result += GetDaysRented() * 3;
                    break;

                case Movie.CHILDRENS:
                    result += 1.5;
                    if (GetDaysRented() > 3)
                    {
                        result += (GetDaysRented() - 3) * 1;
                    }
                    break;
            }
            return result;
        }
```

Bij deze heb ik ook de naam van de functie veranderd in GetCharge(), omwille van de duidelijkheid. Tracht altijd naamgevingen te gebruiken die direct duidelijk maken wat je programmeert. Dus in de Customer klasse staat nu

```
public string Statement()
        {
```

```
            double totalAmount = 0;
            int frequentRenterPoints = 0;

            string result = "Rental Record for " + GetName() + "\n";

            foreach (Rental r in _rentals)
            {
                double thisAmount = 0;
                thisAmount += r.GetCharge();
…
```

Het klasse diagramma is nu veranderd naar:

![movie architectuur][moviearchitectuur2.PNG]

Als we terug naar de statement() functie kijken dan is de variabele thisAmount redundant, en veranderen we naar:

```
public string Statement()

        {
            double totalAmount = 0;
            int frequentRenterPoints = 0;

            string result = "Rental Record for " + GetName() + "\n";

            foreach (Rental r in _rentals)
            {


                //Show figures for this rental
  result += "\t" + r.GetMovie().GetTitle() + "\t" + r.GetCharge().ToString() + "\n";
                totalAmount += r.GetCharge();
            }

            //Add footer lines
            result += "Amount owned is " + totalAmount.ToString() + "\n";

            return result;
        }
```

Best is om tijdelijke variabelen te verwijderen, omdat je makkelijk vergeet waarvoor ze dienen. Je zou in bovenstaand geval toch kunnen kiezen voor een temporary variabele thisAmount, omdat de getCharge() tweemaal wordt opgeroepen dus tweemaal een berekening maakt, als we dan naar performantie kijken.

In de Customer klasse:

```
private double getTotalCharge()
        {
            double result = 0;

            foreach (Rental r in _rentals)
            {
                result += r.GetCharge();
            }

            return result;

        }
```

Met de statement functie als:

```
public string Statement()
        {
            string result = "Rental Record for " + GetName() + "\n";

            foreach (Rental r in _rentals)
            {

                //Show figures for this rental
result += "\t" + r.GetMovie().GetTitle() + "\t" +  r.GetCharge().ToString() + "\n";
            }

            //Add footer lines
            result += "Amount owned is " +getTotalCharge().ToString() + "\n";
result += "You earned " + getTotalFrequentRenterPoints().ToString() + "frequent renter points";

            return result;

        }
```

HTMLStatement() functie

In plaats van tekst te loggen wil ik mijn prijsberekening naar een HTML pagina schrijven. Dit is nu vrij simpel, en bij veranderingen in de prijsberekening moet ik de customer klasse niet meer aanpassen!

```
public string HtmlStatement()
        {
            string result = "<h1>Rental Record for " + GetName() + "</h1>";

            foreach (Rental r in _rentals)
            {
                //Show figures for this rental
              result += "<p>" + r.GetMovie().GetTitle() + "   " + r r.GetCharge().ToString() + "</br></p";
            }

            //Add footer lines
            result += "<p>Amount owned is " + getTotalCharge().ToString() + "</br>";
result += "You earned " + getTotalFrequentRenterPoints().ToString() + "frequent renter points</p>";

            return result;

}
```

Bij een verandering aan de berekening, of toevoeging van nieuwe types films worden de statement functies niet meer gewijzigd, waardoor we duidelijk meer onderhoudvriendelijke code hebben geschreven.

```
 public double GetCharge()
        {
            double thisAmount = 0;
            switch (GetMovie().GetPriceCode())
            {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if (GetDaysRented() > 2)
                    {
                        thisAmount += (GetDaysRented() - 2) * 1.5;
                    }
                    break;

                case Movie.NEW_RELEASE:
                    thisAmount += GetDaysRented() * 3;
                    break;
```

```
                case Movie.CHILDRENS:
                    thisAmount += 1.5;
                    if (GetDaysRented() > 3)
                    {
                        thisAmount += (GetDaysRented() - 3) * 1;
                    }
                    break;
            }
            return thisAmount;
        }
```

Het valt hier op dat we in de Rental klasse met een Movie object werken. Logischerwijze zou deze functie beter in de movie klasse staan. Het is een slecht idee om een switch te doen op een attribuut van een ander object!

We moeten dan wel het aantal huurdagen meegeven als parameter van deze nieuwe functie. Dus eigenlijk gebruikt deze functie 2 stukken data – type film, en aantal huurdagen. Waarom dan toch naar Movie klasse brengen, en daysRented meegeven als argument? Wel , de voorgestelde veranderingen gingen om type film (wat te doen als nieuw type wordt geïntroduceerd), daarom is het logisch om de type informatie zo compact mogelijk te bundelen (in 1 functie ipv 2 functies (als je het type zou doorgeven als parameter)).

De Rental klasse:

```
public double GetCharge()
        {
            return GetMovie().GetCharge(DaysRented);

        }
```

In de klasse Movie zit nu:

```
public double GetCharge(int daysRented)
        {
            double result = 0;
            switch (GetPriceCode())
            {
                case Movie.REGULAR:
                    result += 2;
                    if (daysRented > 2)
                    {
                        result += (daysRented - 2) * 1.5;
                    }
                    break;

                case Movie.NEW_RELEASE:
                    result += daysRented * 3;
                    break;

                case Movie.CHILDRENS:
                    result += 1.5;
                    if (daysRented > 3)
                    {
                        result += (daysRented - 3) * 1;
                    }
                    break;
            }
            return result;
        }
```

Open-Closed Principe (OCP)

Het open/closed principe stelt dat klasses of functies open moeten zijn voor uitbreiding, maar gesloten voor wijziging!

> Open for extension, closed for modification

Gesloten voor wijziging betekent dat het gedrag mag veranderd worden zonder de broncode aan te passen..

Het typische voorbeeld:

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

Nu bouwen we een applicatie die de oppervlakte van een collectie rechthoeken zal berekenen.

```
public class OppBerekenaar
   {

       public double Opp(Rechthoek[] shapes)
       {
           double opp = 0;
           foreach (var shape in shapes)
           {
               opp += shape.Width * shape.Height;
           }

           return opp;
       }
   }
```

En we schrijven ons testprogramma:

```
static void Main(string[] args)
       {

           Rechthoek rh1 = new Rechthoek() { Width = 49, Height = 30 };
           Rechthoek rh2 = new Rechthoek() { Width = 30, Height = 20 };
           Rechthoek rh3 = new Rechthoek() { Width = 22, Height = 10 };
           Rechthoek rh4 = new Rechthoek() { Width = 44, Height = 35 };

           Rechthoek[] rechthoeken = new Rechthoek[4];
           rechthoeken[0] = rh1;
           rechthoeken[1] = rh2;
           rechthoeken[2] = rh3;
           rechthoeken[3] = rh4;

           OppBerekenaar opb = new OppBerekenaar();
           double totaal = opb.Opp(rechthoeken);

           Console.WriteLine("totaal: "+ totaal);
           Console.ReadLine();
       }
```

De volgende vraag komt op: kunnen we het programma uitbreiden zodat we ook de oppervlakte van een cirkel

kunnen berekenen?

We passsen de code als volgt aan:

```
public double Opp(Object[] shapes)
        {
            double opp = 0;
            foreach (var shape in shapes)
            {
                if(shape is Rechthoek)
                    opp += ((Rechthoek)shape).Width * ((Rechthoek)shape).Height; //CAST to Rechthoek


                if (shape is Cirkel)
                    opp += ((Cirkel)shape).Straal * ((Cirkel)shape).Straal * Math.PI;  //CAST to cirkel
            }

            return opp;
        }
```

Wat later krijgen we de vraag om de OppBereken klasse uit te breiden zodat we ook de oppervlakte van driehoeken kunnen opnemen. Dit druist in tegen het principe "gesloten voor wijziging!"

OPC oplossing

> Maak gebruik van abstractie.

In .NET betekent abstractie : gebruik maken van interfaces, of abstracte klassen.

Wat is een interface?

Je hebt geleerd dat een klasse slechts van één klasse kan erven. Een klasse kan echter ook nog interfaces implementeren. Wanneer een klasse een interface implementeert sluit de klasse een contract **met de compiler dat de klasse zich zal gedragen volgens de interface. Concreet betekent dit dat in de klasse alle eigenschappen (properties) en methoden van de interface moet implementeren. Een interface bevat dus eigenlijk enkel een lijst van eigenschappen en methoden die nog geen concrete invulling hebben.**

Volgens WIKIPEDIA: Een interface in de programmeertaal als Java of C# is een soort abstracte klasse die een interface aanduidt die klassen kunnen implementeren. Een interface wordt aangeduid met het sleutelwoord interface en bevat alleen ongedefinieerde methoden.

Wat is een abstracte klasse?

In de informatica is een abstracte klasse een klasse die ongedefinieerde methoden kan bevatten. Deze methoden worden geïmplementeerd in een subklasse van de abstracte klasse. Het is niet mogelijk om een object te maken van abstracte klassen maar wel van niet-abstracte subklassen. Door middel van overerving is het wel mogelijk om de methoden die wel gedefinieerd zijn in de abstracte klasse te erven en in de subklassen te gebruiken.

> Een klasse kan meerdere interfaces implementeren maar alleen van één klasse (rechtstreeks) overerven. Een verschil met abstracte klassen is dat een abstracte klasse wel gedefinieerde methoden kan bevatten maar een interface bevat alleen ongedefinieerde methoden.

Om aan het OPC principe te voldoen moeten we als volgt te werk gaan:

We maken een basis klasse voor rechthoeken, cirkels, driehoeken, andere vormen, en deze definieert een abstracte methode om de oppervlakte te berekenen.

```
public abstract class Vorm
   {
       public abstract double Oppervlakte();
   }
```

De andere klassen leiden af van vorm:

```
public class Rechthoek: Vorm
   {
       public int Width { get; set; }
       public int Height { get; set; }

       public override double Oppervlakte()
       {
           return Width * Height;
       }
   }
```

```
public class Cirkel:Vorm
   {

       public int Straal { get; set; }

       public override double Oppervlakte()
       {
           return Straal * Straal * Math.PI;
       }
   }
```

De berekening gebeurt nu als volgt:

```
public class OppBerekenaar
   {

       public double Opp(Vorm[] shapes)
       {
           double opp = 0;
           foreach (var shape in shapes)
           {
               opp += shape.Oppervlakte();
           }

           return opp;
       }

   }
```

Op deze manier is de OppBerekenaar klasse gesloten voor wijziging, maar toch open voor uitbreiding!

In de praktijk

OPC zal je als ervaren programmeur sneller toepassen. Van bij de start van je ontwikkeling zal je niet altijd OPC

toepassen, en accepteer dat een klasse veranderd moet worden. Maar bij nog verandering, zorg je ervoor dat je naar het OPC principe refactort.

Liskov Substitution Design Principle

> Subtypes moeten vervangbaar zijn door hun super types (parent class).
>
> de IS-Arelatie zou vervangen moeten worden door IS-VERVANGBAAR DOOR

Als voorbeeld werken we met een klasse vierkant die overerft van Rechthoek. De klasse Rechthoek heeft eigenschappen als "width" en "height", en vierkant erft deze over. Maar als voor de klasse vierkant de width OF height gekend is, ken je de waarde van de andere ook. En dit is tegen het principe van Liskov.

```
public class Rechthoek
  {
      public virtual int Width { get; set; }
      public virtual int Height { get; set; }

      public int BerekenOpp()
      {
          return Width * Height;
      }
  }
```

De klasse Vierkant erft over van Rechthoek (maar is in programmeren een vierkant wel een rechthoek?) Een vierkant is een rechthoek met gelijke breedte en hoogte, en we kunnen de properties virtual maken in de klasse Rechthoek om dit te realiseren. Rare implementatie, niet? Maar kijk nu naar de client code..

```
public class Vierkant:Rechthoek
  {
      public override int Width
      {
          get
          {
              return base.Width;
          }

          set
          {
              base.Width = value;
              base.Height = value;
          }
      }

      public override int Height
      {
          get
          {
              return base.Height;
          }

          set
          {
              base.Height = value;
              base.Width = value;
          }
      }
  }
```

Client code:

```
static void Main(string[] args)
       {
           Rechthoek r = new Vierkant();

           r.Width = 5;
           r.Height = 10;

           Console.WriteLine(r.BerekenOpp());
           Console.ReadLine();

       }
```

De gebruiker weet dat r een Rechthoek is dus is hij in de veronderstelling dat hij de width en height kan aanpassen zoals in de parent klasse. Dit in acht genomen zal de gebruiker verrast zijn om 100 te zien ipv 50.

Oplossen van het LSP probleem

- Code dat niet **vervangbaar** is zorgt ervoor dat polymorfisme niet werkt
- Client code (en dit geval de Main) veronderstelt dat basis klassen kunnen vervangen worden door hun afgeleide klassen (Rechthoek r = new Vierkant())
- Het oplossen van LSP door switch cases zorgt voor een onderhoudsnachtmerrie!

```
public abstract class Shape
   {
       public abstract int BerekenOpp();
   }

   public class Rechthoek : Shape
   {
       public int Width { get; set; }
       public int Height { get; set; }
       public override int BerekenOpp()
       {
           return Width * Height;
       }
   }

   public class Vierkant : Shape
   {
       public int Side { get; set; }
       public override int BerekenOpp()
       {
           return Side * Side;
       }
   }

   public class OppBerekenaar
   {
       public List<Shape> shapes;
       public int BerekenOppervlakte()
       {
           shapes = new List<Shape>();
           shapes.Add(new Vierkant() { Side = 10 });
           shapes.Add(new Rechthoek(){ Width = 5, Height= 20 });

           int total = 0;
           foreach(Shape s in shapes)
           {
               total += s.BerekenOpp();
           }

           return total;
```

```
        }
    }
```

Een ander voorbeeld:

```
public interface ICar {
     void drive();
     void playRadio();
     void addLuggage();
}
```

Wat gebeurt er als we een Formule 1 auto hebben:

```
public class FormulaOneCar: ICar {
    public void drive() {
        //Code to make it go super fast
    }

    public void addLuggage() {
        throw new NotSupportedException("No room to carry luggage, sorry.");
    }

    public void playRadio() {
        throw new NotSupportedException("Too heavy, none included.");
    }
}
```

De interface dient als het contract, en moet je veronderstellen dat alle auto's dit gedrag hebben.

Dit is de essentie van het Liskov Substitution Principle.

Waarom is het schenden van LSP niet goed?

Gebruik van abstracte klassen betekent dat je in de toekomst makkelijk een subklasse kan toevoegen in de werkende, geteste code. Dit is de essentie van het open closed principe. Maar wanneer je subklassen gebruikt die niet volledig de interface (abstracte klasse) supporteren moet je in de bestaande code speciale gevallen gaan definiëren.

Bijvoorbeeld:

```
public void DoeIets(Bird b){

    if(b is Pinguin) {
        //Doe iets met de pinguin
    }
    else {
        //Doe iets anders
    }
}
```

Hoe LSP ontdekken

> Kom je code tegen zoals bovenstaande: if(b is Pinguin) ... else , is dit een teken dat je niet aan LSP voldoet,

> en bijgevolg je code flexibeler moet schrijven.
>
> Kom je code tegen waarbij methoden afgeleid van een basis klasse niet geïmplementeerd worden: vb.

```
public void addLuggage() {
       throw new NotSupportedException("No room to carry luggage, sorry.");
   }
```

Nog een indicatie tot Liskov schending

Zie je code als onderstaande, wil dit zeggen dat je niet voldoet aan het LSP principe:

```
public abstract class BaseClass
   {
       public abstract void Method1();

       public abstract void Method2();
   }

   public class ChildClass : BaseClass
   {
       public override void Method1()
       {
           throw new NotImplementedException();
       }

       public override void Method2()
       {
           //Doe Iets
       }
   }
```

Interface Segregation Principle (ISP)

> Client mogen niet afhankelijk zijn over interfaces die ze niet gebruiken

```
class SomeButton
   {
       private SomeController _controller;
       public void setController(SomeController controller) { }
   }

   class SomeWindow
   {
       private SomeController  _controller;
       public void setController(SomeController controller) { }
   }

   class SomeController
   {

       private SomeWindow _window;
       private SomeButton _okButton;
       private SomeButton _cancelButton;

       public void onButtonDown(SomeButton button) { }
       public void onButtonUp(SomeButton button) { }
       public void onWindowOpen(SomeWindow window) { }
       public void onWindowClose(SomeWindow window) { }
```

```
        public void onWindowMoved(SomeWindow window) { }
    }
```

De SomeController klasse handelt de click en window events van SomeButtons en SomeWindows. Het probleem is bovenstaande architectuur is dat zowel SomeButton als SomeWindow naar een object van SomeController refereren. Dus SomeButton kan de OnButton Down en Up uitvoeren van het controller object, maar ook de OnWindowOpen/Close en Move events van Window. Dit is het schenden van ISP. In SomeButton zou het controller object enkel de button events mogen aanspreken, terwijl bij SomeWindow deze enkel de Window events mag oproepen.

ISP violation oplossen

```
class SomeButton
   {
       private IButtonController _controller;
       public void setController(IButtonController controller) { }
   }

   interface IButtonController
   {
       void onButtonDown(SomeButton button);
       void onButtonUp(SomeButton button);
   }

   class SomeWindow
   {
       private IWindowController _controller;
       public void setController(IWindowController controller) { }
   }

   interface IWindowController
   {
        void onWindowOpen(SomeWindow window);
        void onWindowClose(SomeWindow window);
        void onWindowMoved(SomeWindow window);
   }

   class SomeController: IButtonController, IWindowController
   {

       private SomeWindow _window;
       private SomeButton _okButton;
       private SomeButton _cancelButton;

       public void onButtonDown(SomeButton button) { }
       public void onButtonUp(SomeButton button) { }
       public void onWindowOpen(SomeWindow window) { }
       public void onWindowClose(SomeWindow window) { }
       public void onWindowMoved(SomeWindow window) { }
   }
```

Hoe ISP detecteren?

1. Implementeer je sommige methodes van een interface niet, dan voldoe je niet aan ISP.

2. Als je klasse refereert naar een andere klasse, maar gebruikt in beperkte mate deze gerefereerde klasse, dan voldoe je niet aan ISP.

Dependency inversion principle

> High level modules mogen niet afhankelijk zijn van low level modules. Ze mogen enkel afhankelijk zijn van abstractie.

Wat zijn dependencies?

> Een dependency is iets dat misschien kan veranderen tijdens de levenscyclus van je code.

Stel je voor dat je muis, keyboard, monitor door de fabrikant aan je moederbord zijn gesoldeerd zijn. Met andere woorden, wanneer je een moederbord koopt heb je ook een muis, keyboard en monitor. Stel je voor je met één van de devices vervangen. Je beschadigt misschien het moederboard.

In programmeren is het net hetzelfde. Het dependency Inversion principe is een manier om plugs aan je code toe te voegen zodat de high level modules (moederbord) onafhankelijk is van de low level modules (muis). De low level modules kunnen later ontwikkeld worden en zouden makkelijk vervangbaar moeten zijn.

Een indicatie die veranderingen te weeg brengen is het "new" keyword.

Een praktische oplijsting van veel voorkomende dependencies:

- Third party library
- Database
- File system
- Web Service
- New keyword

Je zou er voor moeten zorgen dat constructors alle dependencies bevat die een klasse nodig heeft. Dit noemen we EXPLICIT DEPENDENCIES. In het andere geval noemen we het hidden dependencies.

Dependency injection is een techniek om een dependency (afhankelijkheid) te injecteren in een klasse wanneer deze klasse ze nodig heeft.

Een voorbeeld:

```
class EventLogWriter
{
    public void Write(string message)
    {
        //Write to event log here
    }
}

class Printer
{
    // Handle to EventLog writer to write to the logs
    EventLogWriter writer = null;

    // This function will be called when the app pool has problem
    public void Notify(string message)
    {
        if (writer == null)
        {
            writer = new EventLogWriter();
        }
        writer.Write(message);
    }
```

```
}
```

Op het eerste zicht is er niets mis met bovenstaande code. Maar eigenlijk schenden we DIP. De high level module 'Printer' is afhankelijk van de klasse EventLogWriter. Deze klasse noemen we een concrete klasse, en is dus geen abstracte klasse.

Het wordt duidelijker als we ook een email naar de IT admin willen sturen bij een bepaald probleem en niet enkel een log neerschrijven.

Als we een klasse schrijven voor het versturen van emails, moet de Printer klasse het juiste kunnen afhandelen, zonder dat wij concrete code implementeren. Dus nu is de Printer klasse afhankelijk van de EventLogWriter klasse, en hiervan willen we vanaf.

Onderstaande code laat zien hoe je verandering op een ondynamische manier injecteert:

```
class EventLogWriter
   {
       public void Write(string message)
       {
           //Write to event log here
       }
   }

   class EmailLogWriter
   {
       public void Send(string message)
       {
           //Send email
       }
   }

   class Printer
   {

       EventLogWriter writer = null;
       EmailLogWriter email = null;


       public void Notify(string message, string type)
       {
           if (type == "EventViewer")
           {
               if (writer == null)
               {
                   writer = new EventLogWriter();
               }
               writer.Write(message);
           }

           if (type == "email")
           {
               if (email == null)
               {
                   email = new EmailLogWriter();
               }
               email.Send(message);
           }
       }
   }
```

Dus onze printer klasse moet een instantie van al onze loggers bijhouden. Volgens het dependency inversion principe moeten we ons systeem ontkoppelen zodat de higher level modules, dus de Printer module afhankelijk is van een abstracte klasse of interface. Deze abstractie zal gemapt worden (polymorf gedrag) naar een concrete

klasse die de juiste actie zal ondernemen.

```
interface INotification
   {
       void Notify(string message);
   }

   class EventLogWriter:INotification
   {
       public void Notify(string message)
       {
       }

   }

   class EmailLogWriter:INotification
   {
       public void Notify(string message)
       {

       }


   }

   class Printer
   {

       INotification writer;

       public printer(INotification w){
           writer = w;
       }
       public void Notify(string message)
       {

           writer.Notify(message);
       }
   }
```

Op deze manier maken we de Printer klasse (High level module) onafhankelijk van de concrete log klassen.

Hoe kunnen we deze verder ontkoppelen zodat we bij het toevoegen van andere log klassen (vb. SMS logger), de printer klasse niet meer aangepast hoeft te worden?

Dit kan je implementeren door Dependency injectie

Dependency Injection

Dependency Injection betekent dat we een concrete implementatie in een klasse injecteren, met als doel om de koppeling tussen klassen te verminderen.

3 manieren voor Dependency injection:

- Constructor injection
- Method injection
- Property injection

Contructor injection

Constructor Injection

We geven het object van de concrete klasse mee met de constructor van de afhankelijke klasse.

```
class Printer
    {
        INotification writer;

        public Printer(INotification logger)
        {
            writer = logger;
        }


        public void Notify(string message)
        {
            writer.Notify(message);
        }
    }
```

```
     static void Main(string[] args)
        {

            EventLogWriter log = new EventLogWriter();
            Printer p = new Printer(log);
            p.Notify("dit is een test");

            Console.ReadLine();
        }
```

Method Injection

In constructor injection wordt de concrete klasse gedurende de volledige levenscyclus van de Printer gebruikt. Als je verschillende concrete klassen moet aanroepen, moet je deze in de methode zelf injecteren.

```
static void Main(string[] args)
        {

            EventLogWriter log = new EventLogWriter();
            Printer p = new Printer();
            p.Notify(log,"dit is een test");

            Console.ReadLine();
        }
```

Property Injection

We geven het object van de concrete klasse mee via een set property.

```
  class Printer
    {
        // Handle to EventLog writer to write to the logs

        public INotification writer { get; set; }
```

```
        // This function will be called when the app pool has problem
        public void Notify(INotification logger, string message)
        {
            writer = logger;
            writer.Notify(message);
        }
    }
```

```
static void Main(string[] args)
        {

            EventLogWriter log = new EventLogWriter();
            Printer p = new Printer();
            p.writer = log;
            p.Notify("dit is een test");

            Console.ReadLine();
        }
```

Een tweede voorbeeld

Bepaal de dependencies:

```
public class Order
    {
        public void Checkout(Cart cart, PaymentDetails paymentDetails, bool notifyCustomer)
        {
            if (paymentDetails.PaymentMethod == PaymentMethod.CreditCard)
            {
                ChargeCard(paymentDetails, cart);
            }

            ReserveInventory(cart);

            if (notifyCustomer)
            {
                NotifyCustomer(cart);
            }
        }

        public void NotifyCustomer(Cart cart)
        {
            string customerEmail = cart.CustomerEmail;
            if (!String.IsNullOrEmpty(customerEmail))
            {
                using (var message = new MailMessage("orders@somewhere.com", customerEmail))
                using (var client = new SmtpClient("localhost"))
                {
                    message.Subject = "Your order placed on " + DateTime.Now;
                    message.Body = "Your order details: \n " + cart;

                    try
                    {
                        client.Send(message);
                    }
                    catch (Exception ex)
                    {
                        Logger.Error("Problem sending notification email", ex);
                        throw;
                    }
                }
            }
        }
```

```
        public void ReserveInventory(Cart cart)
        {
            foreach (OrderItem item in cart.Items)
            {
                try
                {
                    var inventorySystem = new InventorySystem();
                    inventorySystem.Reserve(item.Sku, item.Quantity);
                }
                catch (InsufficientInventoryException ex)
                {
                    throw new OrderException("Insufficient inventory for item " + item.Sku, ex);
                }
                catch (Exception ex)
                {
                    throw new OrderException("Problem reserving inventory", ex);
                }
            }
        }

        public void ChargeCard(PaymentDetails paymentDetails, Cart cart)
        {
            using (var paymentGateway = new PaymentGateway())
            {
                try
                {
                    paymentGateway.Credentials = "account credentials";
                    paymentGateway.CardNumber = paymentDetails.CreditCardNumber;
                    paymentGateway.ExpiresMonth = paymentDetails.ExpiresMonth;
                    paymentGateway.ExpiresYear = paymentDetails.ExpiresYear;
                    paymentGateway.NameOnCard = paymentDetails.CardholderName;
                    paymentGateway.AmountToCharge = cart.TotalAmount;

                    paymentGateway.Charge();
                }
                catch (AvsMismatchException ex)
                {
                    throw new OrderException("The card gateway rejected the card based on the address provided.", ex);
                }
                catch (Exception ex)
                {
                    throw new OrderException("There was a problem with your card.", ex);
                }
            }
        }
    }
```

De dependencies opgelijst:

- MailMessage
- SmtpClient
- Inventory
- PaymentGateway

Hoe moeten we dit refactoren?

1. Dependencies in interfaces stoppen
2. Injecteer de implementatie van deze interface in de order klasse
3. Zorg voor Single Responsible principle

Het toepassen van dependency injection zorgt typisch voor heel wat interfaces die ergens moeten geinstantierd worden. Dit doen we meestal in de default constructor of in de Main (de startup routine van je applicatie)

De MailMessage en SmtpClient zorgt voor een eventuele verandering. Stel je wil later de klant niet via email een notificatie sturen, maar via facebook messenger, dan zal je deze code moeten aanpassen. Door dit in interface te

duwen, zal je veel flexibelere code schrijven:

```
public interface INotifyCustomer
   {
       void NotifyCustomer(Cart cart);
   }
```

```
  public class NotifyCustomerService : INotifyCustomer
   {
       /**
      * Method Notifies the customer via Email
      * @param cart a cart object to mail all cart details
      */
       public void NotifyCustomer(Cart cart)
       {
           string customerEmail = cart.CustomerEmail;
           if (!String.IsNullOrEmpty(customerEmail))
           {
               using (var message = new MailMessage("orders@somewhere.com", customerEmail))
               using (var client = new SmtpClient("localhost"))
               {
                   message.Subject = "Your order placed on " + DateTime.Now;
                   message.Body = "Your order details: \n " + cart;

                   try
                   {
                       client.Send(message);
                   }
                   catch (Exception ex)
                   {
                       Logger.Error("Problem sending notification email", ex);
                       throw;
                   }
               }
           }
       }
   }
```

In de order klasse zie je de flexibiliteit terugkomen:

```
public class Order
   {
       INotifyCustomer _notifier;

       public Order(INotifyCustomer notification)
       {
           _notifier = notification;

       }
       public void Checkout(Cart cart, PaymentDetails paymentDetails, bool notifyCustomer)
       {

           if (notifyCustomer)
           {
               _notifier.NotifyCustomer(cart);
           }
       }
```

Op deze manier moet de Order klasse niet weten of we een email notificatie sturen, en push notification voor mobile phone, een facebook message, ...

Strategy patroon

We gaan verder met het voorbeeld van Single Responsibility op method niveau.

De conditionele logica van PrijsCode veranderen met behulp van Polymorfisme

Overerving

Omdat we verschillende types films hebben, die elk een verschillende GetCharge() implementatie hebben, kan je beter gebruik maken van subclasses. Op die manier is het zeer onderhoudbaar om nieuwe types toe te voegen (OPC Principe). Indien je niet met subclasses werkt, zal je in verschillende functies aanpassingen moeten doen: dit leidt tot bugs en inefficiëntie..

Klassediagramma:

![klassediagram][klassediagram.PNG]

Op deze manier heb je het switch statement (getCharge() functie) in de klasse Movie niet meer nodig!

```
switch (GetPriceCode())
            {
                case Movie.REGULAR:
                    result += 2;
                    if (daysRented > 2)
                    {
                        result += (daysRented - 2) * 1.5;
                    }
                    break;

                case Movie.NEW_RELEASE:
                    result += daysRented * 3;
                    break;

                case Movie.CHILDRENS:
                    result += 1.5;
                    if (daysRented > 3)
                    {
                        result += (daysRented - 3) * 1.5;
                    }
                    break;
            }
```

Maar elke subklasse zal een implementatie maken van getCharge(). Voorbeeldcode:

```
public abstract class Movie
    {
        private string _title;

        public Movie(string title)
        {
            _title = title;
        }

        public string GetTitle()
        {
            return _title;
        }

        public abstract double GetCharge(int daysRented);
```

```
…


class NewReleaseMovie : Movie
    {
        public NewReleaseMovie(string title) : base(title) { }

        public override double GetCharge(int daysRented)
        {
            return  daysRented * 3;
        }
    }
```

Aan deze implementatie zit wel een zeer groot nadeel. Het is niet mogelijk om een film van classificatie te wisselen. Dus stel dat je de film “Rundskop” maakt als een NewReleaseMovie, dan is het later niet mogelijk om van deze film een RegularMovie te maken.

```
            Movie m2 = new NewReleaseMovie ("Rundskop");
```

Indien je nu van m2 een RegularMovie wil maken, zal je de variabele opnieuw moeten instantiëren, met als gevolg dat je alle originele data kwijt bent.

```
            m2 = new RegularMovie ("Rundskop");
```

Met andere woorden, onze objecten kunnen niet van klasse veranderen gedurende hun levenstijd. Er is wel een oplossing voor, namelijk gebruik maken van het strategy Patroon.

![strategy][strategy.PNG]

In plaats van te subklassen met Movie (RegularMovie, NewReleaseMovie, ChildrensMovie,) gaan we dit op een indirecte manier doen en van price subklassen maken. (zie klassediagramma hierboven). Op die manier kunnen we van een movie object steeds de prijs veranderen, zonder dat we het ganse movie object opnieuw moeten instantiëren.

Het strategy pattern is dus in staat om een status/ of om in realtime algoritme te veranderen (bijvoorbeeld prijs van een film veranderen na een week van New Release naar Regular) bij te houden. Door vele programmeurs wordt dit initieel via enum en switch/case structuur opgezet. Het state patroon wordt aangemaakt door de type code naar state klassen over te brengen. Daarna gaan we de conditionele logica (switch/case) naar de price klassen zetten, om tenslotte deze switch case te verwijderen.
De strategy klassen

```
public abstract class Price
    {
        public abstract double getCharge(int daysRented);

    }

class RegularPrice : Price
    {
        public override double getCharge(int daysRented)
        {

            double result = 2;
            if (daysRented > 3)
                result += (daysRented - 2) * 1.5;
```

```
            return result;
        }
    }

public  class ChildrensPrice : Price
    {

        public override double getCharge(int daysRented)
        {
            double result = 1.5;
            if (daysRented > 3)
                result += (daysRented - 3) * 1.5;

            return result;
        }
    }


    public class NewReleasePrice:Price
    {
        public override double getCharge(int daysRented)
        {
            return daysRented * 3;
        }

        public override int getFrequentRenterPoints(int daysRented)
        {
            return (daysRented > 1) ? 2 : 1;
        }
    }
```

We maken nu een abstracte klasse price met zijn subklassen. De getCharge() functie in de superklasse is abstract, zodat de afgeleide klassen verplicht zijn een eigen implementatie te geven aan deze functie. In de Movie klasse zal je in plaats van de int _priceCode een member van het type Price moeten definiëren. In runtime beslis je dan of deze member de constructor van NewReleasePrice, ChildrensPrice of RegularPrice oproept.

```
        private Price _priceCode;

        public Movie(string title, Price priceCode)
        {
            _title = title;
            _priceCode = priceCode;
        }

        public Price GetPriceCode()
        {
            return _priceCode;
        }

        public void SetPriceCode(Price arg)
        {
            //is dit juist? copyconstructor..?
            _priceCode = arg;
        }
```

De volgende stap is om de getCharge() functie in de klasse Movie over te dragen naar de juiste Price klasse. (de implementatie van getCharge() in de desbetreffende klassen kan je reeds zien op de vorige pagina). De getCharge() functie van de Movie klasse wordt dan:

```
public double GetCharge(int daysRented)
        {
            return _priceCode.getCharge(daysRented);
```

```
        }
```

Hierbij is ook de conditionele logica in de functie verdwenen (want deze is overgedragen naar de Price klassen).

We kunnen dit nu ook doen voor de getFrequentRenterPoints(int daysRented). We moeten deze functie in de superclasse 'Price' niet abstract maken maar geven er hier reeds een implementatie aan. We maken deze wel virtual zodat de afgeleide kunnen bepalen of ze er een eigen definitie aan geven.

```
public abstract class Price
    {
        public abstract double getCharge(int daysRented);

        public virtual int getFrequentRenterPoints(int daysRented)
        {
            return 1;
        }

    }
```

Enkel de NewReleasePrice geeft een eigen implementatie aan de getFrequentRenterPoints(int daysRented) functie:

```
public class NewReleasePrice:Price
    {
        public override double getCharge(int daysRented)
        {
            return daysRented * 3;
        }

        public override int getFrequentRenterPoints(int daysRented)
        {
            return (daysRented > 1) ? 2 : 1;
        }
    }
```

Dit strategy patroon invoegen gaf ons heel wat werk, maar de winst is dat ik prijzen gemakkelijk kan veranderen en nieuwe prijsklassen kan aanmaken zonder andere code te wijzigen. Want de rest van de implementatie weet niets van dit state pattern! Voor grote complexe projecten betekent dat heel wat winst.

Alle bovenstaande veranderingen moeten leiden tot makkelijker te onderhouden code. Een heel verschil met proceduraal programmeren, maar wanneer je dit onder knie hebt, zal je veel gemakkelijker tests kunnen schrijven en veranderingen implementeren!

> Eigenschap van het strategy patrooon: Het strategy patroon laat toe om objecten in real time van status te veranderen.