Inhoud

[Software engineering: software architectuur 2](#_Toc306344287)

[REFACTORING 2](#_Toc306344288)

[Voorbeeld van de MAIN functie 2](#_Toc306344289)

[Movie klasse .. een simpele klasse 3](#_Toc306344290)

[Rental klasse 4](#_Toc306344291)

[Customer klasse 4](#_Toc306344292)

[Analyse van onze architectuur 6](#_Toc306344293)

[Decompose statement functie 7](#_Toc306344294)

[Analyse van AmountFor functie 8](#_Toc306344295)

[Extract frequent huur punten 10](#_Toc306344296)

[HTMLStatement() functie 14](#_Toc306344297)

[De conditionele logica van PrijsCode veranderen met behulp van Polymorfisme 14](#_Toc306344298)

[Overerving 16](#_Toc306344299)

[State Pattern 18](#_Toc306344300)

[De state klassen 18](#_Toc306344301)

# Software engineering: software architectuur

## REFACTORING

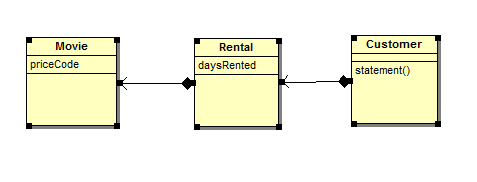
Probleemstelling:

Er is je gevraagd om software te schrijven voor een videotheek. Het programma berekent en print de rekening van een klant bij onze videotheek.

Onderstaande paragraaf geeft ons de voorbeeldcode van het programma. We zullen deze oplossing grondig analyseren en bekijken hoe we onze code kunnen verbeteren.

Aan het programma wordt meegegeven welke film de klant heeft gehuurd, en voor hoe lang. Daarna wordt de rekening gemaakt – afhankelijk hoe lang de film gehuurd geweest is, en welk type film (nieuwe release, kinder, gewone). Als extra functionaliteit geeft het programma huurpunten (vergelijkbaar met stempels op een klantenkaart), die verschillen al naargelang de film een nieuwe release is of niet.

UML notatie:



### 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;

private string \_title;

private int \_priceCode;

public Movie(string title, int priceCode)

{

\_title = title;

\_priceCode = priceCode;

}

public int GetPriceCode()

{

return \_priceCode;

}

public void SetPriceCode(int arg)

{

\_priceCode = arg;

}

public string GetTitle()

{

return \_title;

}

}

### Rental klasse

*Deze klasse stelt voor welke film en voor hoe lang een klant gehuurd heeft*

public class Rental

{

private Movie \_movie;

private int \_daysRented;

public Rental(Movie movie, int daysRented)

{

\_movie = movie;

\_daysRented = daysRented;

}

public int GetDaysRented()

{

return \_daysRented;

}

public Movie GetMovie()

{

return \_movie;

}

}

### Customer klasse

*Deze klasse stelt de klant van de winkel voor*

public class Customer

{

private string \_name;

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

public Customer(string name)

{

\_name = name;

}

public void AddRental(Rental arg)

{

\_rentals.Add(arg);

}

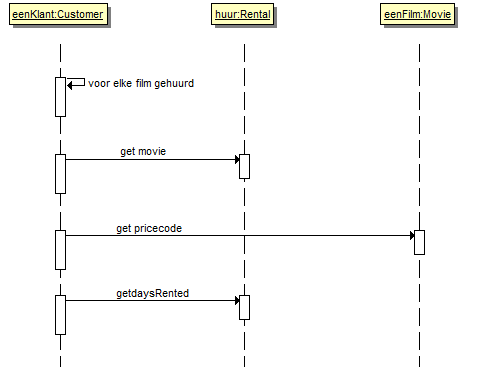
public string GetName()

{

return \_name;

}

Het sequentiële diagramma voor onze geschreven code als we de statement functie uitvoeren van onze klasse Customer is als volgt:



De functie statement (moet nog aan onze Customer klasse worden toegevoegd)

public string Statement()

{

double totalAmount = 0;

int frequentRenterPoints = 0;

string result = "Rental Record for " + GetName() + "\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) \* 1.5;

}

break;

}

//Add frequent renter points

frequentRenterPoints++;

//Add bonus for a two day new release rental

if (r.GetMovie().GetPriceCode() == Movie.NEW\_RELEASE && r.GetDaysRented() > 1)

{

frequentRenterPoints++;

}

//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";

result += "You earned " + frequentRenterPoints.ToString() + "frequent renter points";

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 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 in de mogelijkheid te voorzien 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 op 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+=”<b>”blabla</b>” ..

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 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.

## Decompose 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().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;

}

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) \* 1.5;

}

break;

}

return result;

}

Bij deze heb ik ook de naam van de functie veranderd in GetCharge(), omwille van 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)

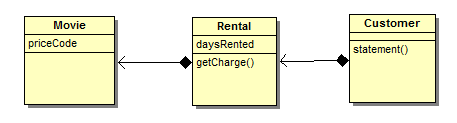
{

double thisAmount = 0;

thisAmount = r.GetCharge();

…

Het klasse diagramma is nu veranderd naar:



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)

{

//Add frequent renter points

frequentRenterPoints++;

//Add bonus for a two day new release rental

if (r.GetMovie().GetPriceCode() == Movie.NEW\_RELEASE && r.GetDaysRented() > 1)

{

frequentRenterPoints++;

}

//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";

result += "You earned " + frequentRenterPoints.ToString() + "frequent renter points";

return result;

}

Best is om temporary variabelen te verwijderen, omdat je makkelijk vergeet voor wat 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.

Nu kunnen we hetzelfde doen voor de huurpunten…

## Extract huur punten

De statement() functie van de klasse Customer wordt veranderd in:

public string Statement()

{

double totalAmount = 0;

int frequentRenterPoints = 0;

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

foreach (Rental r in \_rentals)

{

**frequentRenterPoints = r.getFrequentRenterPoints();**

//Show figures for this rental

result += "\t" + r.GetMovie().GetTitle() + "\t" + r.GetCharge().ToString() + "\n";

totalAmount += r.GetCharge();

}

In de klasse Rental wordt een nieuwe functie geïmplementeerd, namelijk getFrenquentRenterPoints() , omdat in de statement() functie gebruik gemaakt wordt van de Rental klasse:

public int getFrequentRenterPoints()

{

if (GetMovie().GetPriceCode() == Movie.NEW\_RELEASE && GetDaysRented() > 1)

{

return 2;

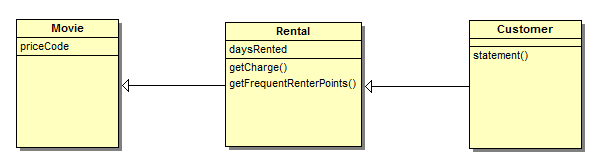
}

else

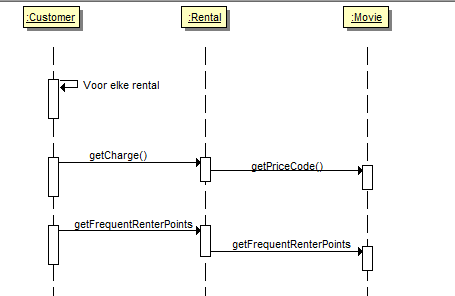
return 1;

}

Het nieuwe klassediagramma: (*GetFrequentRenterPoints() is veranderd*)



Dus ook het sequentiediagramma is aangepast:



We kunnen de statement functie nog kleiner maken, door aparte functies te voorzien om de eindafrekening en het totaal aantal huurpunten te berekenen. Nadeel hier is dat we de loop 3x moeten doorlopen (in de statement functie en de aparte getTotalCharge () en getTotalFrequentRenterPoints() functie. Omdat het de bedoeling is een “open/onderhoudbare” manier te vinden om onze rekening in HTML af te drukken, gaan we in dergelijk geval toch verder opdelen. Deze refactoring is maar mogelijk omdat het niet om zware berekeningen gaat en de loops dus geen performantieverlies geven. In het geval van zware berekeningen, is het beter om de getCharge() functie niet verder op te delen.

In de Customer klasse:

private double getTotalCharge()

{

double result = 0;

foreach (Rental r in \_rentals)

{

result += r.GetCharge();

}

return result;

}

private int getTotalFrequentRenterPoints()

{

int result=0;

foreach (Rental r in \_rentals)

{

result += r.getFrequentRenterPoints();

}

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

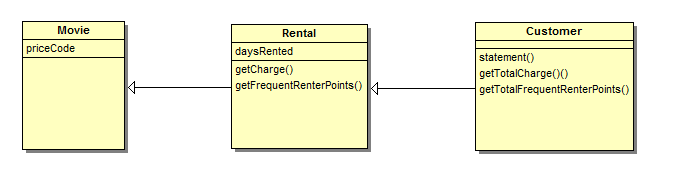
**result += "Amount owned is " +getTotalCharge().ToString() + "\n";**

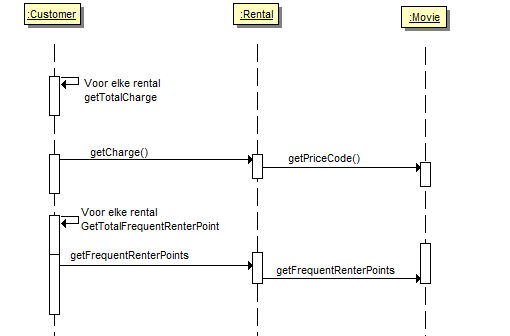
**result += "You earned " + getTotalFrequentRenterPoints().ToString() + "frequent renter points";**

return result;

}

Het klassediagramma en sequentiediagramma verloopt dus nu:





## HTMLStatement() functie

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() + " &nbsp; " + 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 verandering aan berekening, of toevoeging van nieuwe types films worden de statement functies niet meer gewijzigd, waardoor we duidelijk meer onderhoudvriendelijke code hebben geschreven.

## De conditionele logica van PrijsCode veranderen met behulp van Polymorfisme

Laat ons eens naar het switch statement kijken in de Rental klasse bij de functie getCharge() :

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) \* 1.5;

}

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 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) \* 1.5;

}

break;

}

return result;

}

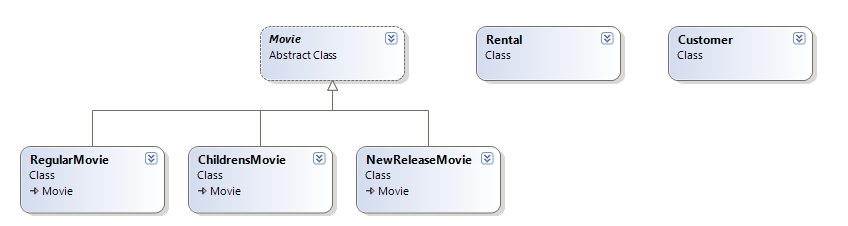
We kunnen nu ook hetzelfde doen voor de huurpunten berekening.

(TODO: uml schema + klassediagramma)

## Overerving

Daar 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. Indien je niet met subclasses werkt, zal je in verschillende functies aanpassingen moeten doen: dit leidt tot bugs en inefficiëntie..

Klassediagramma:



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");

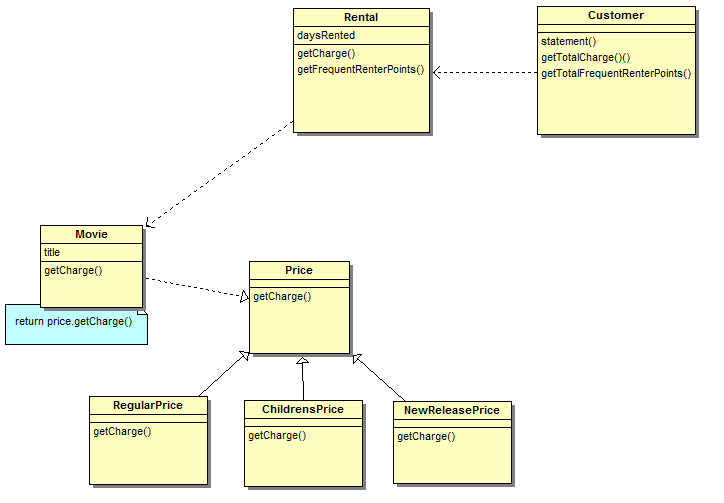
Indien je nu van m2 een RegularMovie wil maken, zal je de variabele opnieuw moeten instantiëren, met als gevolg dat je alle originele data kwijt bent.

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 **State Patroon**.

## State Pattern



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 state pattern is dus in staat om een status bij te houden (status geven, wijzigen). 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 state 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;

}

}

Een abstracte klasse price met zijn subklassen. We maken de getCharge() functie in de superklasse 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 klassen. (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 aangeven.

public abstract class Price

{

public abstract double getCharge(int daysRented);

public virtual int getFrequentRenterPoints(int daysRented)

{

return 1;

}

}

Enkel de NewReleasePrice geeft 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)

{

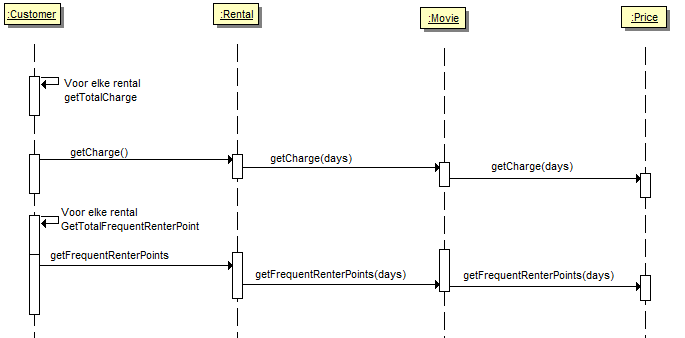
return (daysRented > 1) ? 2 : 1;

}

}

Dit state 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.

Hieronder geef ik nog een nieuwe sequentiële diagramma weer:



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!

Bedenk intussen dat het aanmaken van verschillende levels voor je project volledig in bovenstaand verhaal klopt!