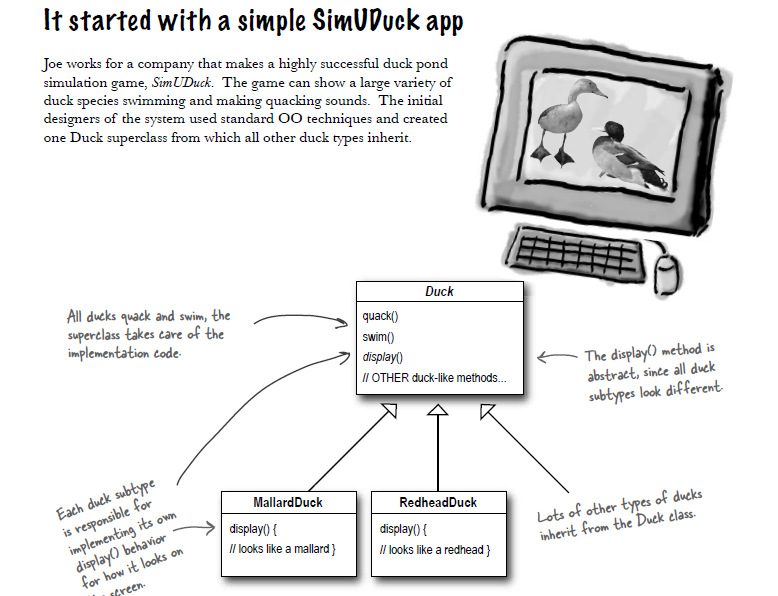
Design Patterns - Inleiding

**De SimUDuck applicatie**

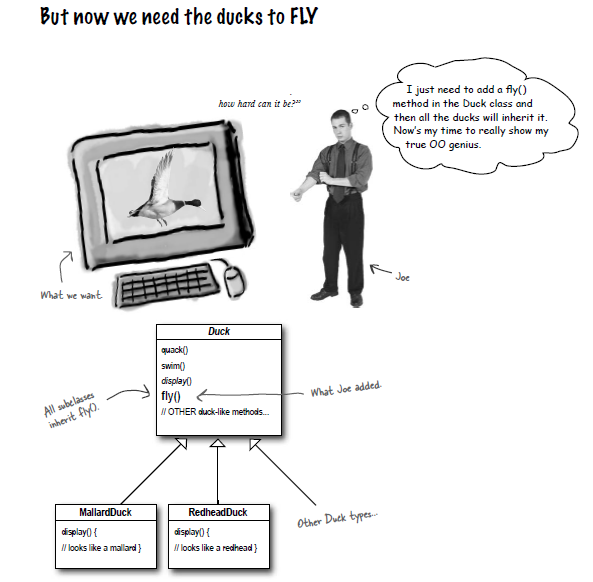
Je werkt voor een bedrijf dat veel succes heeft met een simulatiespel van een eendenvijver. *SimUDuck* . Het spel kan een groot aantal verschillende eendensoorten tonen die rondzwemmen en kwaken. De oorsponkelijke ontwerpers van het systeem gebruikten standaard-OO technieken en creëerden een superclass Duck waar alle andere eendentypen van afgeleid zijn.



Het afgelopen jaar heeft het bedrijf steeds meer last van concurrentie. Na een week lang brainstrormen denkt de bedrijfsleiding dat het tijd wordt voor een grote innovatie. Ze moeten echt iets geweldigs laten zien op de komende aandeelhoudersvergadering volgende week.

**Maar nu moeten de ze ook kunnen vliegen…**

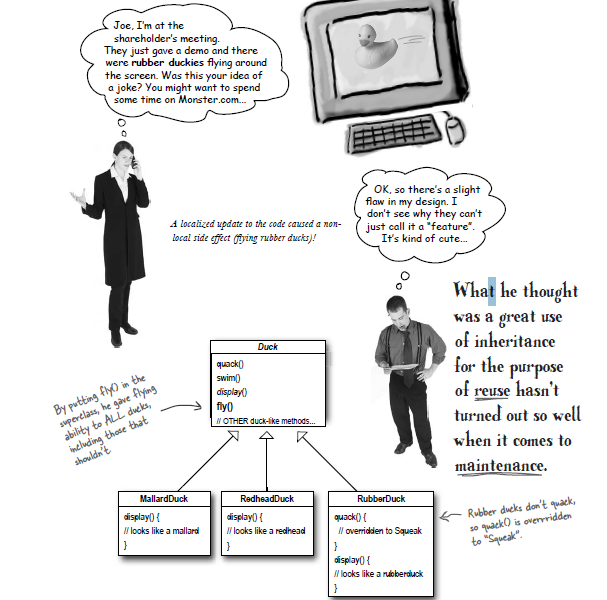
De bedrijfsleiding besloot dat vliegende eenden in de simulator precies hetgeen is dat kan opwegen ten concurrerende eenden spelletjes. En de manager vertelde dat het geen probleem zou zijn om dit in één week toe te voegen… Dat is maar een kleine toevoeging, dat kan toch niet moeilijk zijn?…



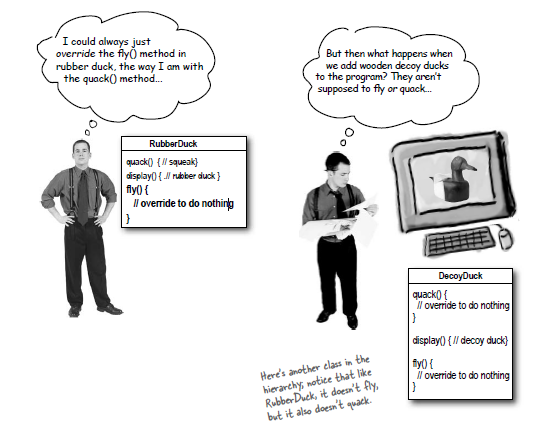
**Er liep iets helemaal verkeerd met de oplossing met overerving…**

*Wat is er gebeurd?*

De ontwikkelaar, Joe, had er niet aan gedacht dat ***niet alle*** subclasses van Duck mogen vliegen. Toen Joe **nieuw gedrag** toevoegde **via de methode Fly(),** voegde hij eveneens gedrag toe aan een **aantal subclasses** van Duck waarvoor **dit niet van toepassing** was. Het gevolg is dat nu niet-vliegende afgeleide objecten (zoals bad-eendjes) kunnen vliegen in het spelletje.



**De ontwikkelaar denkt nog eens na over de oplossing met overerving…**



We weten dat niet alle subclasses vlieg- of kwaakgedrag vertonen, dus is hier **overerving niet de juiste oplossing.** We willen dat we **zo weinig mogelijk onderhoud en herprogrammatie** nodig hebben bij **elke verandering of toevoeging** van gedrag van de eenden en eventuele nieuwe eend-soorten in het spel…

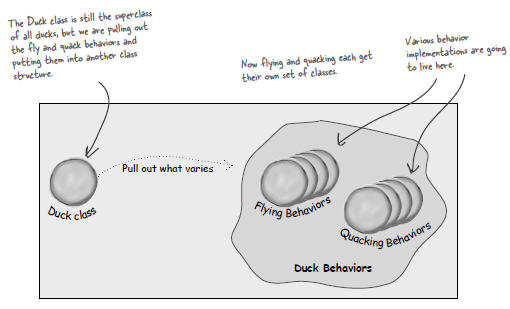
****

# Ontwerpprincipe 1 :

Bepaal de **aspecten** van je applicatie **die variëren en scheid deze af** van de aspecten die hetzelfde blijven.

We weten dat **Fly() en Quack()** delen van de Duck class zijn die **variëren per eenden-soort**.

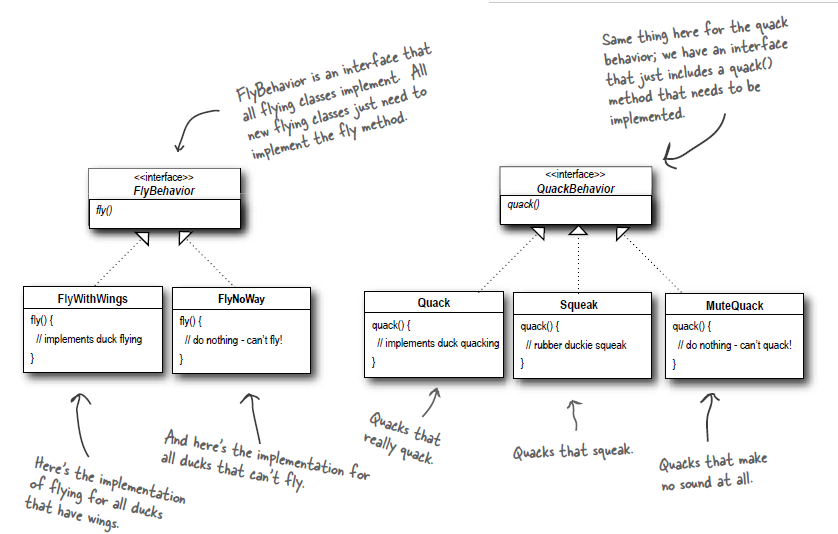
Om dit gedrag van de class Duck te scheiden, halen we beide methoden uit de class Duck en maken we een nieuwe reeks van interface met classes die dit gedrag vertegenwoordigen (voor zowel fly als quack gedrag).



Dus dan maken we een **interface IFlyBehaviour** voor fly-gedrag met **voor elke afzonderlijke manier van vliegen**

bv korte afstand, lange afstand of geen afstand vliegen (**Fly algoritme**), een **nieuwe class die deze interface en dus dit algoritme voor fly implementeert**.

En eveneens een interface voor kwaakgedrag **QuackBehaviour**, en classes die de verschillende manier van kwaken per eendensoort in een **algoritme** (**Quack**) implementeren (bv een bad-eendje maakt een piepend geluid en een lok-eend van een jager nog een ander geluid dan de doorsnee eend:



public interface IFlyBehavior

{

void Fly();

}

public class FlyWithWings : IFlyBehavior

{

public void Fly()

{

Console.WriteLine("Ik vlieg!!");

}

}

public class FlyNoWay : IFlyBehavior

{

public void Fly()

{

Console.WriteLine("I can't fly");

}

}

public interface IQuackBehavior

{

void Quack();

}

// NormalQuack genoemd om naamgevingsconfict te vermijden met naam van methode:

public class NormalQuack : IQuackBehavior

{

public void Quack()

{

Console.WriteLine("Normal Duck Quack");

}

}

public class MuteQuack : IQuackBehavior

{

public void Quack()

{

Console.WriteLine("<< stilte >>");

}

}

public class Squeak : IQuackBehavior

{

public void Quack()

{

Console.WriteLine("Piep Piep");

}

}

Met dit ontwerp kunnen andere objecttypen ons gedrag voor vliegen en kwaken hergebruiken omdat dit gedrag niet langer binnen de Duck class wordt geïmplementeeerd.

We kunnen ook **gemakkelijk nieuw gedrag toevoegen**, zonder de bestaande gedragsclasses te veranderen of aan te spreken.

*Bv. Fly() nieuw gedrag: algoritme voor eend met raket-aandrijving:*

public class FlyRocketPowered : IFlyBehavior

{

public void Fly()

{

Console.WriteLine("I'm flying at rocket-speed!");

}

}

**Integratie van het eendengedrag**

**Duck** class zal nu het **vlieg- en kwaakgedrag kunnen delegeren**, in plaats van gebruik te maken van methoden fly en quack die geïmplementeerd zijn in de class Duck of subclasses.

1. Eerst **voegen we 2 instantievariabelen (fields) toe aan de class Duck met namen flyBehavior en quackBehavior**. Deze hebben het type van de interfaces **IFlyBehavior en IQuackBehavior.**
2. We **verwijderen de methoden Fly() en Quack() uit de class Duck** (en alle subclasses van Duck), want we willen de implementatie van dit gedrag verplaatsen naar concrete classes die de interfaces IFlyBehavior en IQuackBehavior implementeren.
3. We voegen de methoden **PerformFly() en PerformQuack()** toe (deze vervangen dus Fly() en Quack() maar doen iets anders. Zij gaan de Fly() en Quack() **delegeren** via de interfaces..

public abstract class Duck

{

public IFlyBehavior FlyBehavior { get; set; }

public IQuackBehavior QuackBehavior { get; set; }

public abstract void Display();

public void PerformFly()

{

FlyBehavior.Fly();

}

public void PerformQuack()

{

QuackBehavior.Quack();

}

public void Swim() //Nieuw gedrag toevoegen is gemakkelijk

{

Console.WriteLine("Alle eenden zwemmen, ook bad-eendjes en lok-eenden!");

}

}

public class NormalDuck : Duck

{

public NormalDuck ()

{

QuackBehavior = new NormalQuack();

FlyBehavior = new FlyWithWings();

}

override public void Display()

{

Console.WriteLine("I'm a real duck");

}

}

public class DecoyDuck : Duck

{

public DecoyDuck()

{

QuackBehavior = new LoudQuack();

FlyBehavior = new FlyNoWay();

}

override public void Display()

{

Console.WriteLine("I'm a decoy duck");

}

}

public MallardDuck()

{

QuackBehavior = new NormalQuack();

FlyBehavior = new FlyWithWings();

}

override public void Display()

{

Console.WriteLine("I'm a real type of duck");

}

}

public class ModelDuck : Duck //rubber bad-eendje

{

public ModelDuck()

{

QuackBehavior = new Squeak();

FlyBehavior = new FlyNoWay();

}

override public void Display()

{

Console.WriteLine("I'm a model duck");

}

}

**Dan rest ons nog de klant (de code dat de ducks objecten aanmaakt en gedrag instructies aanroept (afhankelijk van de soort klasse waarvan een object wordt aangemaakt zal de Fly() en Quack() gedrag verschillen):**

static void Main(string[] args)

{

Duck mallard = new MallardDuck();

mallard.Display();

mallard.PerformQuack();

mallard.PerformFly();

Console.WriteLine("");

Duck model = new ModelDuck();

model.Display();

model.PerformFly();

model.FlyBehavior = new FlyRocketPowered();

model.PerformFly();

// Wait for user input

Console.ReadKey();

}

# Ontwerpprincipe 2 :

Programmeer naar een interface (of abstracte class), niet naar een implementatie (concrete class).

# Referenties:

Head First Design Patterns – O’ Reilly Media Inc - www.oreilly.com