Design Patterns

Gerwin van Dijken (gerwin.vandijken@inholland.nl)

Programma periode 1.4

01 (wk-15)	abstracte classes en interfaces
02 (wk-16)	Template Method pattern / Observer pattern
03 (wk-17)	MVC pattern
04 (wk-18)	*geen lessen (meivakantie)*
05 (wk-19)	Strategy pattern / Adapter pattern
06 (wk-20)	Singleton pattern / State pattern
07 (wk-21)	Factory patterns
08 (wk-22)	herhaling / proeftentamen

---

09 (wk-23)	tentamen (praktijk)
10 (wk-24)	*hertentamens (vakken periode 1.3)*
11 (wk-25)	*hertentamens (vakken periode 1.4)*

Strategy pattern

- Geeft de mogelijkheid om 'on-the-fly' een nieuwe strategie (algoritme/gedrag) in te zetten
- Dit (dynamisch in te zetten) gedrag is losgekoppeld van de class zelf

The Strategy Pattern (GoF): *'defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.'*

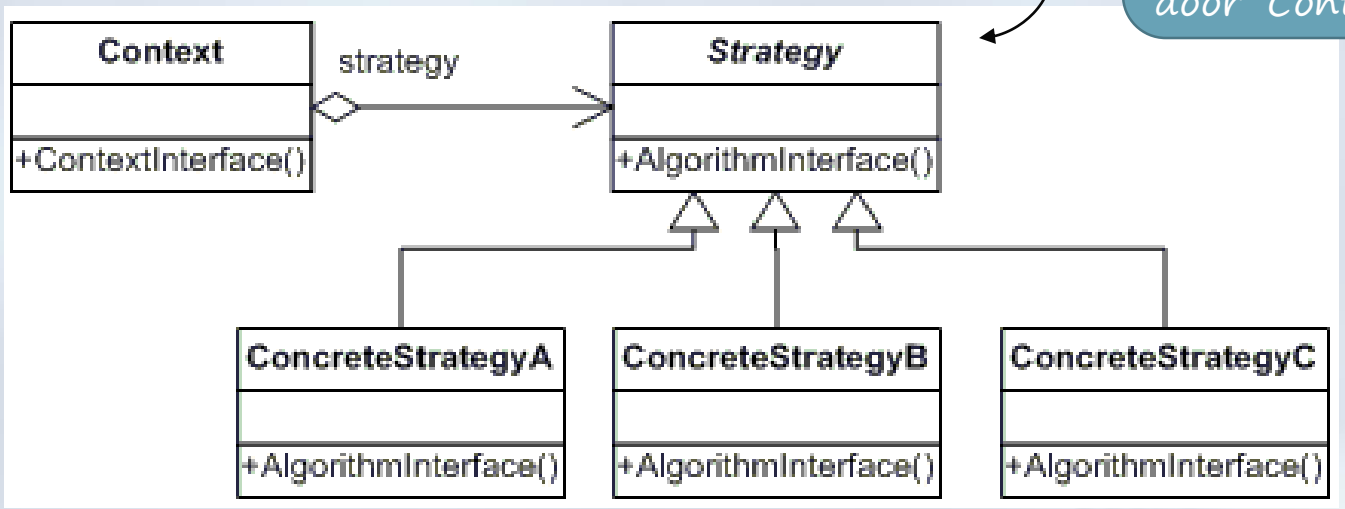
Strategy pattern - 3 onderdelen

- Strategy: de interface die definieert hoe het algoritme moet worden aangeroepen
- Concrete Strategy: de implementatie van de strategy
- Context: het object dat de Concrete Strategy bevat (via interface referentie)

Strategy pattern

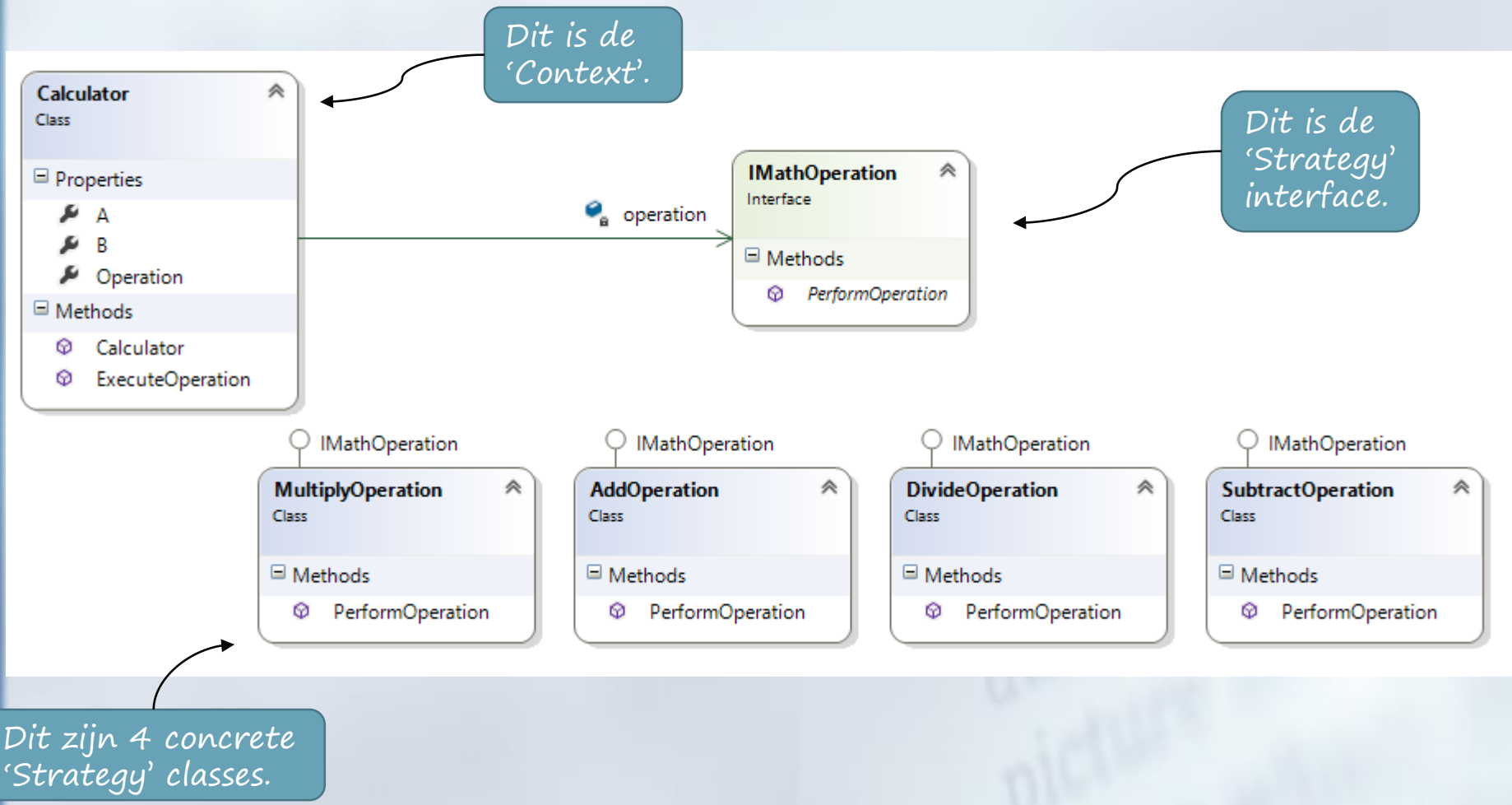
'Context' bevat een referentie naar een Strategy (interface) object.

'Strategy' definieert een interface die gebruikt wordt door 'Context'.



De ConcreteStrategy classes implementeren de interface (algoritme).

Strategy pattern – een voorbeeld



Strategy pattern – e

```
public interface IMathOperation
{
  double PerformOperation(
        double a, double b);
}
```

De 'Strategy' interface.

De 4 concrete 'Strategy' classes.

```
public class AddOperation : IMathOperation
{
  public double PerformOperation(double a, double b)
  {
    return a + b;
  }
}

public class SubtractOperation : IMathOperation
{
  public double PerformOperation(double a, double b)
  {
    return a - b;
  }
}

public class MultiplyOperation : IMathOperation
{
  public double PerformOperation(double a, double b)
  {
    return a * b;
  }
}

public class DivideOperation : IMathOperation
{
  public double PerformOperation(double a, double b)
  {
    return a / b;
  }
}
```

Strategy pattern – een voorbeeld

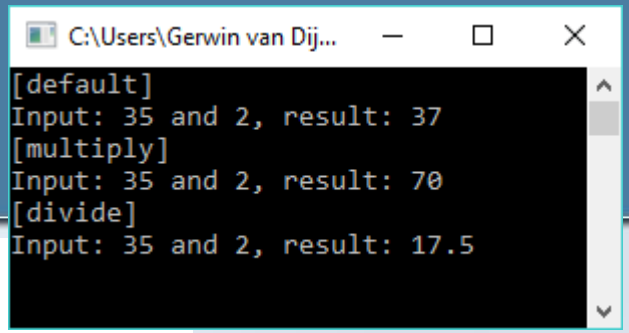
```
public class Calculator
{
  private IMathOperation operation;

  public double A { get; set; }
  public double B { get; set; }

  public IMathOperation Operation
  {
    get { return operation; }
    set { operation = value; }
  }

  // constructor
  public Calculator()
  {
    A = B = 0;

    // default behaviour
    operation = new AddOperation();
  }

  public double ExecuteOperation()
  {
    return operation.PerformOperation(A, B);
  }
}
```

Referentie naar 'Strategy' interface.

Met deze property 'Operation' kan het gedrag ingesteld worden.

Afhandeling gebeurt door het huidige concrete (operation) object.

Strategy pattern




```
void Start()
{
  Calculator calculator = new Calculator();
  calculator.A = 35;
  calculator.B = 2;

  Console.WriteLine("[default]");
  double result = calculator.ExecuteOperation();
  Console.WriteLine(String.Format("Input: {0} and {1}, result: {2}",
          calculator.A, calculator.B, result));

  Console.WriteLine("[multiply]");
  calculator.Operation = new MultiplyOperation();
  result = calculator.ExecuteOperation();
  Console.WriteLine(String.Format("Input: {0} and {1}, result: {2}",
          calculator.A, calculator.B, result));

  Console.WriteLine("[divide]");
  calculator.Operation = new DivideOperation();
  result = calculator.ExecuteOperation();
  Console.WriteLine(String.Format("Input: {0} and {1}, result: {2}",
          calculator.A, calculator.B, result));
}
```

Hier wordt de 'strategy' aangeroepen.

Door het wijzigen van de operatie ('on the fly'), wordt het gedrag v/d calculator gewijzigd.

Voordelen Strategy pattern

- 1. Je kunt ‘at runtime’ gedrag veranderen
  (kan niet bij afgeleide classes…)
- 2. Je kunt in één class meerdere soorten gedrag opnemen (en wijzigen)
  (multiple inheritance kan niet in C#...)

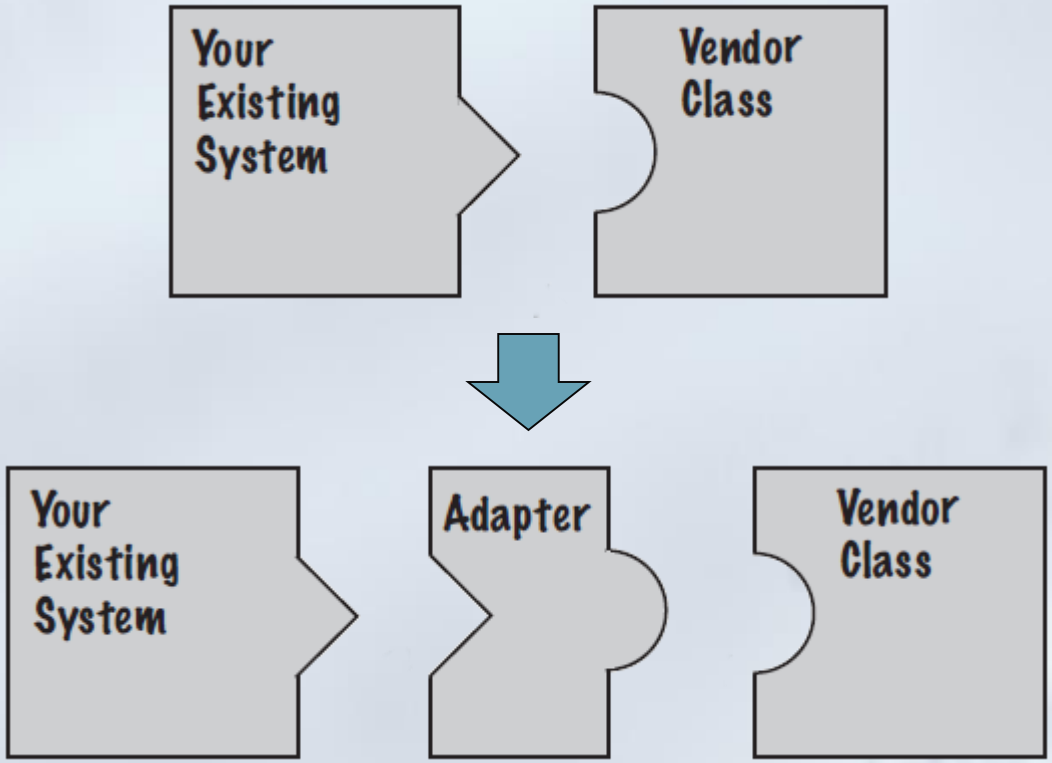
Adapter pattern

- Interface omzetten naar een andere interface, zodat 2 bestaande systemen met elkaar samen kunnen werken

The Adapter Pattern (GoF): *'converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.'*

Adapter pattern

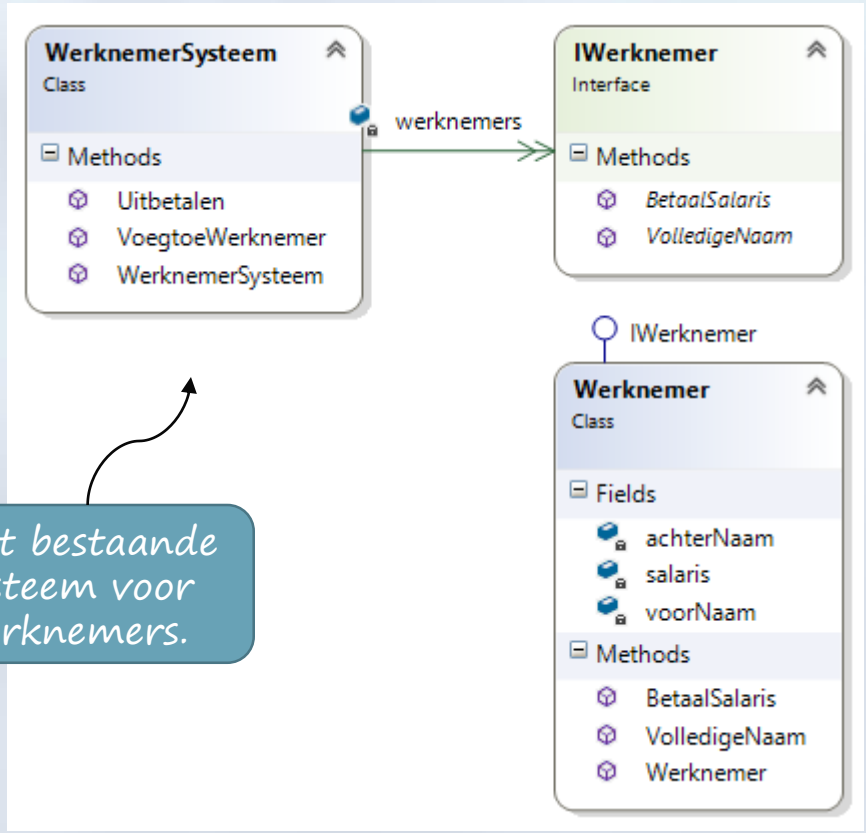
- Bestaande software met andere software laten werken

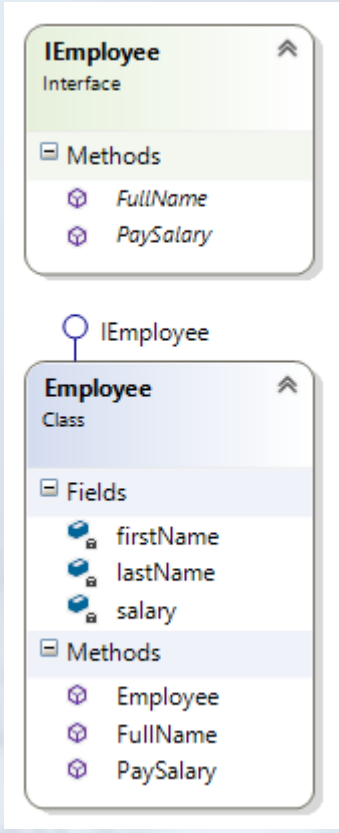


Adapter pattern – een voorbeeld

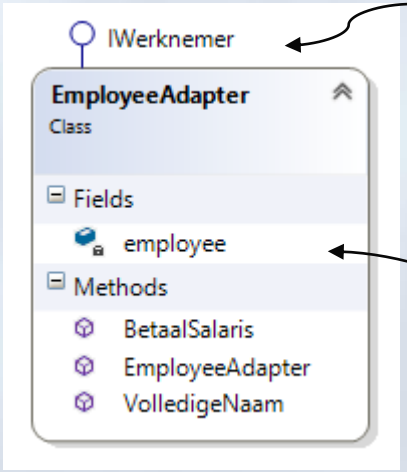
- Stel we hebben een werknemersysteem met daarin… werknemers (IWerknemer)
- Vervolgens moeten uit een ander bestaand systeem daar aan toegevoegd worden: employees (IEmployee)
- We willen de bestaande systemen niet aanpassen

Adapter pattern – een voorbeeld





Adapter pattern – een voorbeeld



De Adapter gedraagt zich als een IWerknemer...

...maar de akties worden door een IEmployee uitgevoerd.

IEmployee wordt meegegeven aan de constructor van de Adapter.

```
class EmployeeAdapter : IWerknemer
{
  private IEmployee employee;

  public EmployeeAdapter(IEmployee employee)
  {
    this.employee = employee;
  }

  public string VolledigeNaam()
  {
    return employee.FullName();
  }

  public void BetaalSalaris()
  {
    employee.PaySalary();
  }
}
```

Adapter pattern - main

```
void Start()
{
  IWerknemer werknemer1 = new Werknemer("Kees", "van Kralingen", 2500);
  IWerknemer werknemer2 = new Werknemer("Karel", "van Dijk", 2800);
  IWerknemer werknemer3 = new Werknemer("Pieter", "de Boer", 2200);

  WerknemerSysteem systeem = new WerknemerSysteem();
  systeem.VoegtoeWerknemer(werknemer1);
  systeem.VoegtoeWerknemer(werknemer2);
  systeem.VoegtoeWerknemer(werknemer3);

  // add employee using an Adapter
  Employee employee1 = new Employee("Sam", "Potter", 3000);
  //systeem.VoegtoeWerknemer(employee1);  // this does not compile...
  systeem.VoegtoeWerknemer(new EmployeeAdapter(employee1));

  systeem.Uitbetalen();
}
```

Hier gebruiken we de adapter, om het systeem met een IEmployee te kunnen laten werken (employee is vermomd als IWerknemer).

```
class WerknemerSysteem
{
  List<IWerknemer> werknemers = new List<IWerknemer>();

  // ...

  public void Uitbetalen()
  {
    foreach (IWerknemer werknemer in werknemers)
      werknemer.BetaalSalaris();
  }
}
```

Samenvattend

- Strategy Pattern: het delegeren van gedrag zodat deze uitwisselbaar is
- Adapter Pattern: omzetten van een nieuw interface naar een bestaande interface zodat bestaande systemen met elkaar kunnen samenwerken

Opdrachten

- Zie Moodle: ‘Week 4 opdrachten’