



# Design Patterns

Gerwin van Dijken ([gerwin.vandijken@inholland.nl](mailto: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)	<i>geen lessen (meivakantie)</i>
05 (wk-19)	Strategy pattern / Adapter pattern
06 (wk-20)	Singleton pattern / State pattern
07 (wk-21)	Factory patterns
08 (wk-22)	herhaling / proeftentamen
<hr/>	
09 (wk-23)	tentamen (praktijk)
10 (wk-24)	<i>hertentamens (vakken periode 1.3)</i>
11 (wk-25)	<i>hertentamens (vakken periode 1.4)</i>

# Template Method pattern

- *'Definieer het geraamte van een algoritme en laat specifieke stappen over aan subclasses. Deze subclasses kunnen dan bepaalde stappen van een algoritme herdefiniëren zonder de structuur van het algoritme te veranderen'*
- 'Standaard recept met enkele specifieke handelingen'
- Grote lijnen liggen vast (in base class), verschillen zitten in afgeleide classes

Template Method (GoF): *'Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.'*

# Koffie...



```
public class Koffie
```

```
{  
    public void BereidingsRecept() {  
        KookWater();  
        SchenkKoffieOp();  
        Inschenken();  
        SuikerEnMelkToevoegen();  
    }  
}
```

*Het recept/algoritme voor koffie.*

```
void KookWater() {  
    Console.WriteLine("Water koken");  
}
```

```
void SchenkKoffieOp() {  
    Console.WriteLine("Schenk koffie door filter");  
}
```

```
void Inschenken() {  
    Console.WriteLine("In mok schenken");  
}
```

```
void SuikerEnMelkToevoegen() {  
    Console.WriteLine("Suiker en melk toevoegen");  
}  
}
```

*Het algoritme (om koffie te zetten) bestaat uit 4 stappen/methoden, nl:*

- water koken;*
- koffie opschenken;*
- koffie inschenken;*
- suiker & melk toevoegen;*

# ... en thee



Het recept/algorithm  
voor thee.

Het algoritme (om thee te zetten)  
bestaat uit 4 stappen/methoden,  
nl:

- water koken;
- theezakje trekken;
- thee inschenken;
- citroen toevoegen;

```
public class Thee
{
    public void BereidingsRecept() {
        KookWater();
        TheezakjeTrekken();
        Inschenken();
        CitroenToevoegen();
    }

    void KookWater() {
        Console.WriteLine("Water koken");
    }

    void TheezakjeTrekken() {
        Console.WriteLine("Theezakje laten trekken");
    }

    void Inschenken() {
        Console.WriteLine("In mok schenken");
    }

    void CitroenToevoegen() {
        Console.WriteLine("Citroen toevoegen");
    }
}
```

# Koffie en thee, zoek de verschillen

```
public class Koffie
```

```
{  
    public void BereidingsRecept() {  
        KookWater();  
        SchenkKoffieOp();  
        Inschenken();  
        SuikerEnMelkToevoegen();  
    }  
}
```

```
void KookWater() {  
    Console.WriteLine("Water koken");  
}
```

```
void SchenkKoffieOp() {  
    Console.WriteLine("Schenk koffie door filter");  
}
```

```
void Inschenken() {  
    Console.WriteLine("In mok schenken");  
}
```

```
void SuikerEnMelkToevoegen() {  
    Console.WriteLine("Suiker en melk toevoegen");  
}
```

```
}
```

```
public class Thee
```

```
{  
    public void BereidingsRecept() {  
        KookWater();  
        TheezakjeTrekken();  
        Inschenken();  
        CitroenToevoegen();  
    }  
}
```

```
void KookWater() {  
    Console.WriteLine("Water koken");  
}
```

```
void TheezakjeTrekken() {  
    Console.WriteLine("Theezakje laten trekken");  
}
```

```
void Inschenken() {  
    Console.WriteLine("In mok schenken");  
}
```

```
void CitroenToevoegen() {  
    Console.WriteLine("Citroen toevoegen");  
}
```

```
}
```

*Veel code hetzelfde!  
Hoe kunnen we dit  
beter ontwerpen?!?*

Dit is de template methode, een vast algoritme.

Deze 2 abstracte methoden worden door afgeleide classes geïmplementeerd

```
public abstract class CafeineDrank
{
    public void BereidingsRecept() {
        KookWater();
        Brouwen();
        Inschenken();
        IngredientenToevoegen();
    }

    public void KookWater() {
        Console.WriteLine("Water koken");
    }

    public abstract void Brouwen();

    public void Inschenken() {
        Console.WriteLine("In mok schenken");
    }

    public abstract void IngredientenToevoegen();
}
```

Deze 2 concrete methoden worden door de base class zelf geïmplementeerd

```
public class Koffie : CafeineDrank
{
    public override void Brouwen() {
        Console.WriteLine("Schenk koffie door filter");
    }

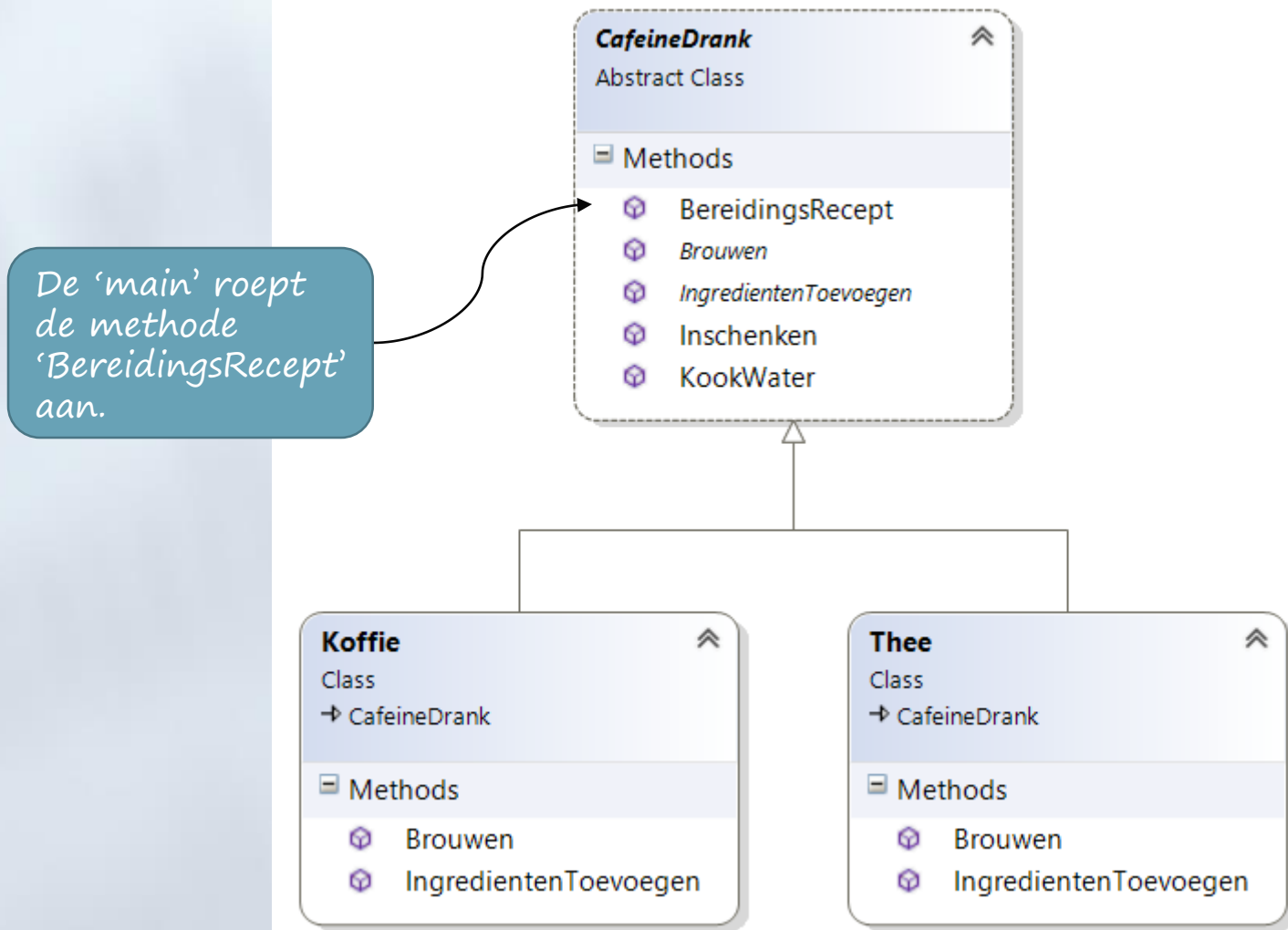
    public override void IngredientenToevoegen() {
        Console.WriteLine("Suiker en melk toevoegen");
    }
}
```

```
public class Thee : CafeineDrank
{
    public override void Brouwen() {
        Console.WriteLine("Theezakje laten trekken");
    }

    public override void IngredientenToevoegen() {
        Console.WriteLine("Citroen toevoegen");
    }
}
```



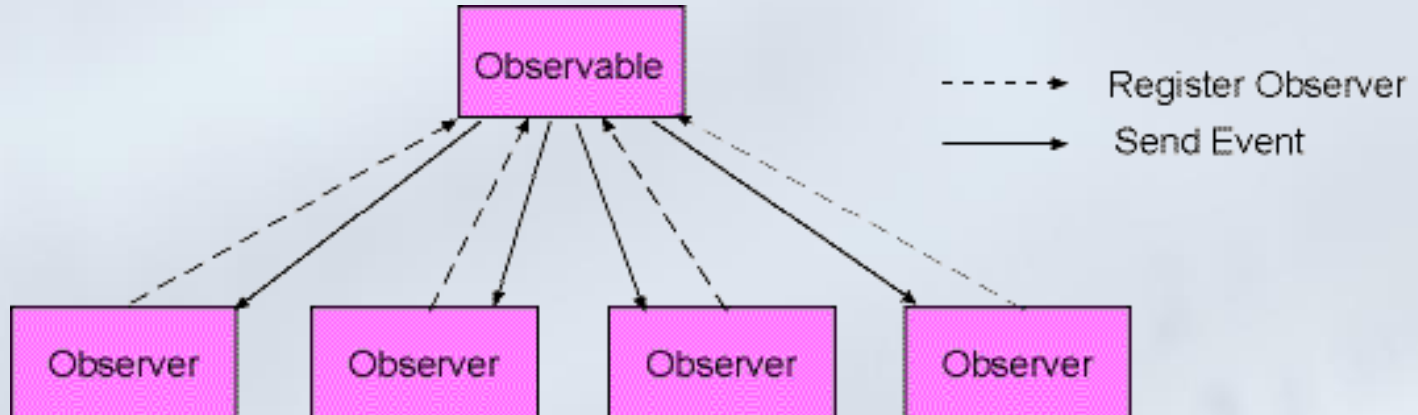
# Class diagram





# Observer pattern

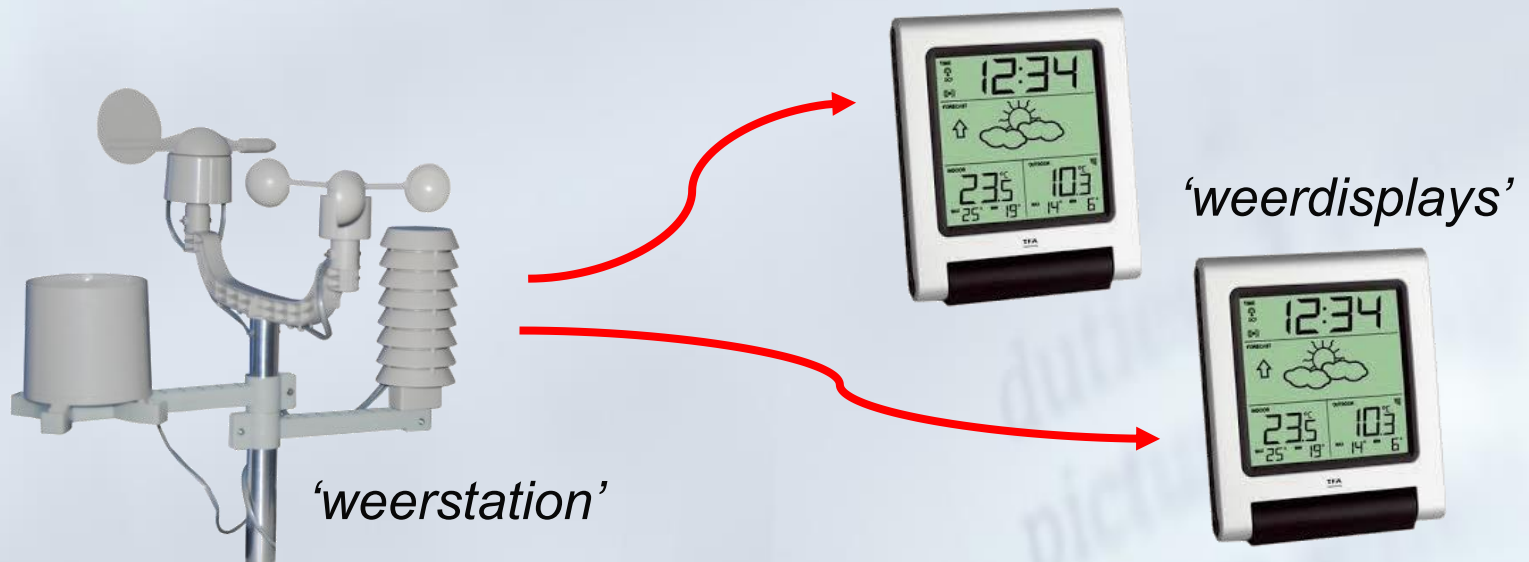
- Subject (Observable) → verstuurt updates
- Observers → krijgen elk een update



*Observer Method (GoF): 'Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.'*

# Voorbeeld applicatie

- Weerstation (Subject/Observable)
- Verschillende weerdisplays (Observers)
- De weerdisplays moeten geupdate worden als er nieuwe meetdata is



Dit is de informatie over het weer.

De weerdisplays zijn de observers; deze laten de weerinformatie zien.

Alle weerdisplays worden hier geupdate.

Wat is hier niet zo handig?!?

```
public class WeerStation
{
    public float Temperatuur { get; set; }
    public float Luchtvochtigheid { get; set; }
    public float Luchtdruk { get; set; }

    private WeerDisplay display1, display2;

    public WeerStation(WeerDisplay display1, WeerDisplay display2)
    {
        this.display1 = display1;
        this.display2 = display2;
    }

    public void MeasurementChanged() {
        // update gegevens (lees sensoren uit)
        LeesTemperatuur();
        LeesLuchtVochtigheid();
        LeesLuchtdruk();

        // update alle displays
        display1.Update(Temperatuur, Luchtvochtigheid, Luchtdruk);
        display2.Update(Temperatuur, Luchtvochtigheid, Luchtdruk);
    }

    // methode om de werkelijke waarde te lezen (sensoren)
    private void LeesTemperatuur() { }
    private void LeesLuchtVochtigheid() { }
    private void LeesLuchtdruk() { }
}
```




# Aantal observers variabel...

- We willen dat elk weerdisplay zich kan aanmelden voor updates (van WeerStation), en weer afmelden (subscribe / unsubscribe)  
(Observer: subscriber)
- WeerStation houdt een lijst bij van aanmelders (observers), en update elk weerdisplay bij een nieuwe meting  
(Subject: publisher)

De weerdisplays zijn nu variabel; er is nu een dynamische lijst.

Een weerdisplay kan zich aanmelden en weer afmelden.

Alle weerdisplays worden hier geupdate.



```
public class WeerStation
{
    public float Temperatuur { get; set; }
    public float Luchtvochtigheid { get; set; }
    public float Luchtdruk { get; set; }

    private List<WeerDisplay> weerDisplays = new List<WeerDisplay>();

    public void RegisterObserver(WeerDisplay display) {
        weerDisplays.Add(display);
    }

    public void RemoveObserver(WeerDisplay display) {
        weerDisplays.Remove(display);
    }

    public void NotifyObservers() {
        foreach (WeerDisplay display in this.weerDisplays) {
            display.Update(Temperatuur, Luchtvochtigheid, Luchtdruk);
        }
    }

    public void MeasurementChanged() {
        // update gegevens (lees sensoren uit)
        LeesTemperatuur();
        LeesLuchtVochtigheid();
        LeesLuchtdruk();

        // update alle displays
        NotifyObservers();
    }
}
```

# Weerdisplay



```
public class WeerDisplay
{
    private WeerStation weerStation;

    public WeerDisplay(WeerStation weerStation)
    {
        this.weerStation = weerStation;
        weerStation.RegisterObserver(this);
    }

    public void Update(float temperatuur, float luchtvochtigheid, float luchtdruk)
    {
        // toon de gegevens
        Console.WriteLine(String.Format("T: {0}, H: {1}, P: {2}",
            temperatuur, luchtvochtigheid, luchtdruk));
    }
}
```

Observer krijgt Subject/ Observable mee in de constructor, en registreert zichzelf.

Deze Update methode wordt aangeroepen vanuit WeerStation.

# Interfaces Observer pattern

- Voor Subject (WeerStation) wordt meestal een interface 'ISubject' gebruikt;



```
public interface ISubject
{
    void RegisterObserver(IObserver display);
    void RemoveObserver(IObserver display);
    void NotifyObservers();
}
```

- Voor Observer (WeerDisplay) wordt meestal een interface 'IObserver' gebruikt;

```
public interface IObserver
{
    void Update(float temperature, float luchtvochtigheid, float luchtdruk);
}
```





# Weerdisplay (IObserver)

WeerDisplay is niet meer gekoppeld aan een concrete class (WeerStation) maar aan een interface (ISubject).

Dit betekent dat een WeerStation vervangen kan worden door een andere 'ISubject'-object.

```
public class WeerDisplay : IObserver
{
    private ISubject weerStation;

    public WeerDisplay(ISubject weerStation)
    {
        this.weerStation = weerStation;
        weerStation.RegisterObserver(this);
    }

    public void Update(float temperatuur, float luchtvochtigheid, float luchtdruk)
    {
        // toon de gegevens
        Console.WriteLine(String.Format("T: {0}, H: {1}, P: {2}",
            temperatuur, luchtvochtigheid, luchtdruk));
    }
}
```



WeerStation is niet meer gekoppeld aan een concrete class (WeerDisplay) maar aan een interface (IObserver).

Dit betekent dat een WeerDisplay vervangen kan worden door een andere 'IObserver'-object.

```
public class WeerStation : ISubject
{
    public float Temperatuur { get; set; }
    public float Luchtvochtigheid { get; set; }
    public float Luchtdruk { get; set; }

    private List<IObserver> weerDisplays = new List<IObserver>();

    public void RegisterObserver(IObserver display) {
        weerDisplays.Add(display);
    }

    public void RemoveObserver(IObserver display) {
        weerDisplays.Remove(display);
    }

    public void NotifvObservers() {
        foreach (IObserver display in this.weerDisplays) {
            display.Update(Temperatuur, Luchtvochtigheid, Luchtdruk);
        }
    }

    public void MeasurementChanged() {
        // update gegevens (lees sensoren uit)
        LeesTemperatuur();
        LeesLuchtVochtigheid();
        LeesLuchtdruk();

        // update alle displays
        NotifyObservers();
    }
}
```



# Samenvattend

- Met de 'Template' pattern wordt een vast algoritme in de base class geplaatst, waarbij sommige (deel)stappen in afgeleide classes geïmplementeerd (moeten) worden
- De 'Observer' pattern zorgt voor een 'zwakke koppeling' tussen 'state' objects (subjects) en 'luisteraars' (observers)  
*(subject weet niet wie er allemaal luisteren...)*

# Opdrachten

- Zie Moodle: 'Week 2 opdrachten'