



Design Patterns

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

Program term 1.4

01 (wk-15)	abstract classes and interfaces
02 (wk-16)	Template Method pattern / Observer pattern
03 (wk-17)	MVC pattern
04 (wk-18)	<i>no classes</i>
05 (wk-19)	Strategy pattern / Adapter pattern
06 (wk-20)	Singleton pattern / State pattern
07 (wk-21)	Factory patterns
08 (wk-22)	repetition / practice exam
<hr/>	
09 (wk-23)	exam (<i>computer assignments</i>)
10 (wk-24)	<i>retakes (courses term 1.3)</i>
11 (wk-25)	<i>retakes (courses term 1.4)</i>

Template Method pattern

- *'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.'*
- 'Standard recipe with some specific steps'
- Global recipe/algorithm is always the same (in the base class), differences are implemented in the derived classes

Coffee ...



```
public class Coffee
```

```
{  
    public void PrepareRecipe() {  
        BoilWater();  
        BrewCoffeeGrinds();  
        PourInCup();  
        AddSugarAndMilk();  
    }  
}
```

The recipe/algorithm for making coffee.

```
    public void BoilWater() {  
        Console.WriteLine("Boiling water");  
    }
```

```
    public void BrewCoffeeGrinds() {  
        Console.WriteLine("Dripping coffee through filter");  
    }
```

```
    public void PourInCup() {  
        Console.WriteLine("Pouring into cup");  
    }
```

```
    public void AddSugarAndMilk() {  
        Console.WriteLine("Adding sugar and milk");  
    }  
}
```

The algorithm (to make coffee) has four steps/methods:

1. boil water;
2. brew coffee grinds;
3. pour coffee in cup;
4. add sugar and milk;

... and tea



The recipe/algorithm for making tea.

The algorithm (to make tea) has four steps/methods:

1. boil water;
2. steep teabag;
3. pour tea in cup;
4. Add lemon;

```
public class Tea
{
    public void PrepareRecipe() {
        BoilWater();
        SteepTeaBag();
        PourInCup();
        AddLemon();
    }

    public void BoilWater() {
        Console.WriteLine("Boiling water");
    }

    public void SteepTeaBag() {
        Console.WriteLine("Steeping the tea");
    }

    public void PourInCup() {
        Console.WriteLine("Pouring into cup");
    }

    public void AddLemon() {
        Console.WriteLine("Adding Lemon");
    }
}
```

Coffee and tea, find the differences...

*Duplicate code!
How can we
minimize this!?!?*

```
public class Coffee
```

```
{
```

```
    public void PrepareRecipe() {
```

```
        BoilWater();
```

```
        BrewCoffeeGrinds();
```

```
        PourInCup();
```

```
        AddSugarAndMilk();
```

```
    }
```

```
    public void BoilWater() {
```

```
        Console.WriteLine("Boiling water");
```

```
    }
```

```
    public void BrewCoffeeGrinds() {
```

```
        Console.WriteLine("Dripping coffee through filter");
```

```
    }
```

```
    public void PourInCup() {
```

```
        Console.WriteLine("Pouring into cup");
```

```
    }
```

```
    public void AddSugarAndMilk() {
```

```
        Console.WriteLine("Adding sugar and milk");
```

```
    }
```

```
}
```

```
public class Tea
```

```
{
```

```
    public void PrepareRecipe() {
```

```
        BoilWater();
```

```
        SteepTeaBag();
```

```
        PourInCup();
```

```
        AddLemon();
```

```
    }
```

```
    public void BoilWater() {
```

```
        Console.WriteLine("Boiling water");
```

```
    }
```

```
    public void SteepTeaBag() {
```

```
        Console.WriteLine("Steeping the tea");
```

```
    }
```

```
    public void PourInCup() {
```

```
        Console.WriteLine("Pouring into cup");
```

```
    }
```

```
    public void AddLemon() {
```

```
        Console.WriteLine("Adding Lemon");
```

```
    }
```

```
}
```

This is the template method, a fixed algorithm.

```
public abstract class CaffeineBeverage
{
    public void PrepareRecipe() {
        BoilWater();
        Brew();
        PourInCup();
        AddIngredients();
    }

    public void BoilWater() {
        Console.WriteLine("Boiling water");
    }

    public abstract void Brew();

    public void PourInCup() {
        Console.WriteLine("Pouring into cup");
    }

    public abstract void AddIngredients();
}
```

These 2 abstract methods will be implemented by derived classes.

These 2 concrete methods are implemented by the base class itself.

```
public class Coffee : CaffeineBeverage
{
    public override void Brew() {
        Console.WriteLine("Dripping coffee through filter");
    }

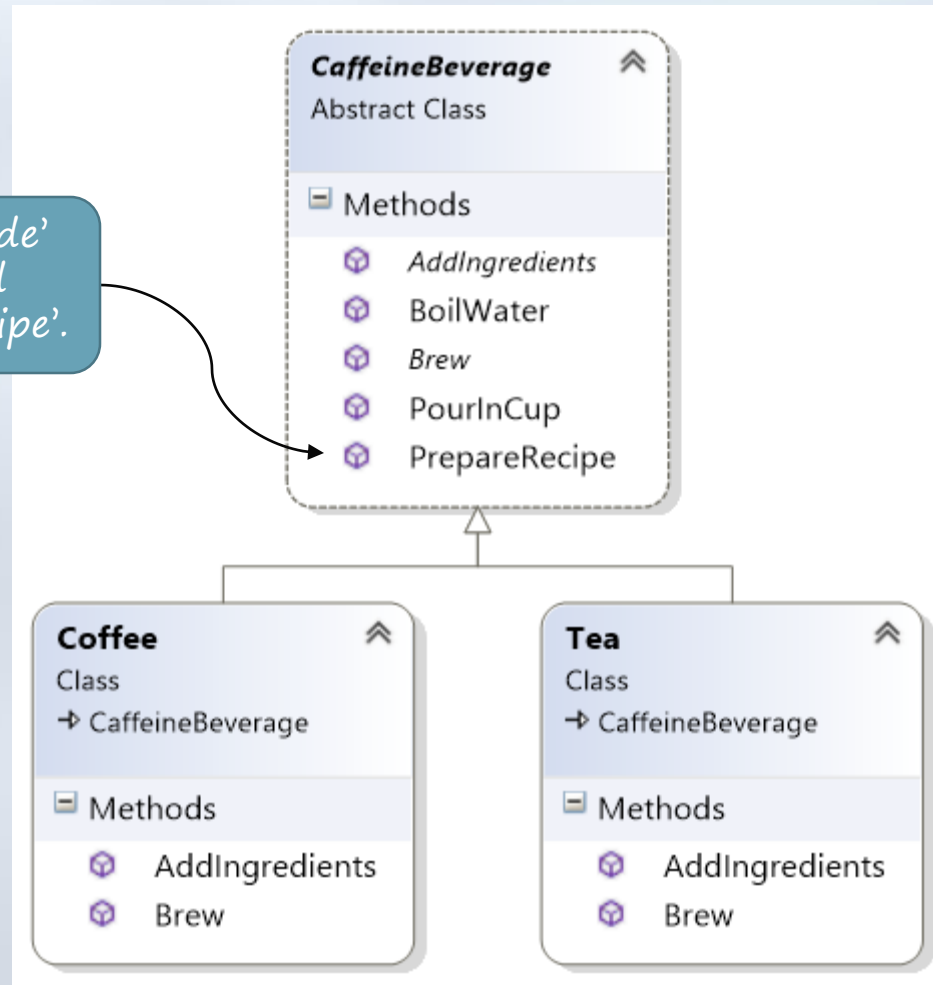
    public override void AddIngredients() {
        Console.WriteLine("Adding sugar and milk");
    }
}
```

```
public class Tea : CaffeineBeverage
{
    public override void Brew() {
        Console.WriteLine("Steeping the tea");
    }

    public override void AddIngredients() {
        Console.WriteLine("Adding Lemon");
    }
}
```

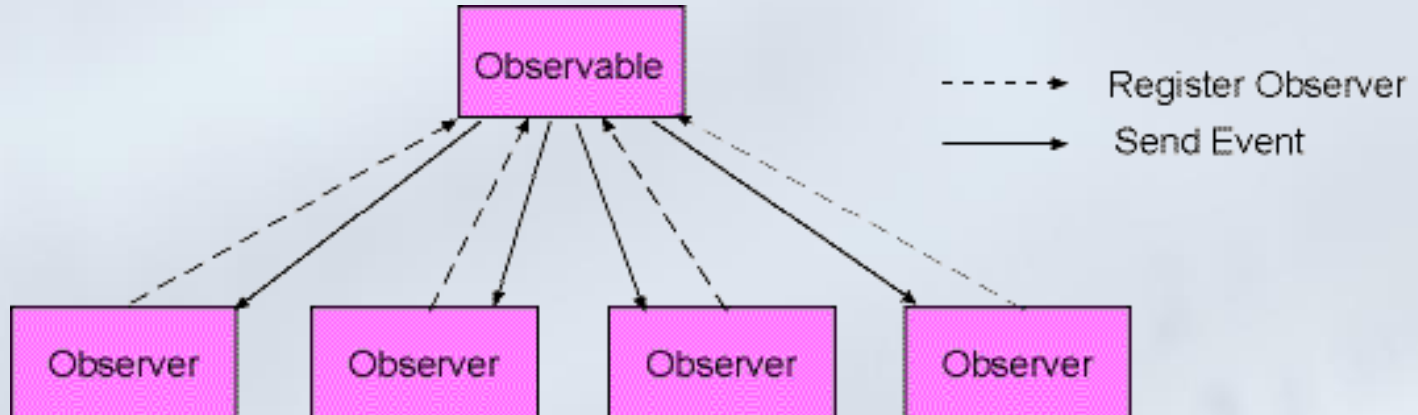
Class diagram

The 'user code' calls method 'PrepareRecipe'.



Observer pattern

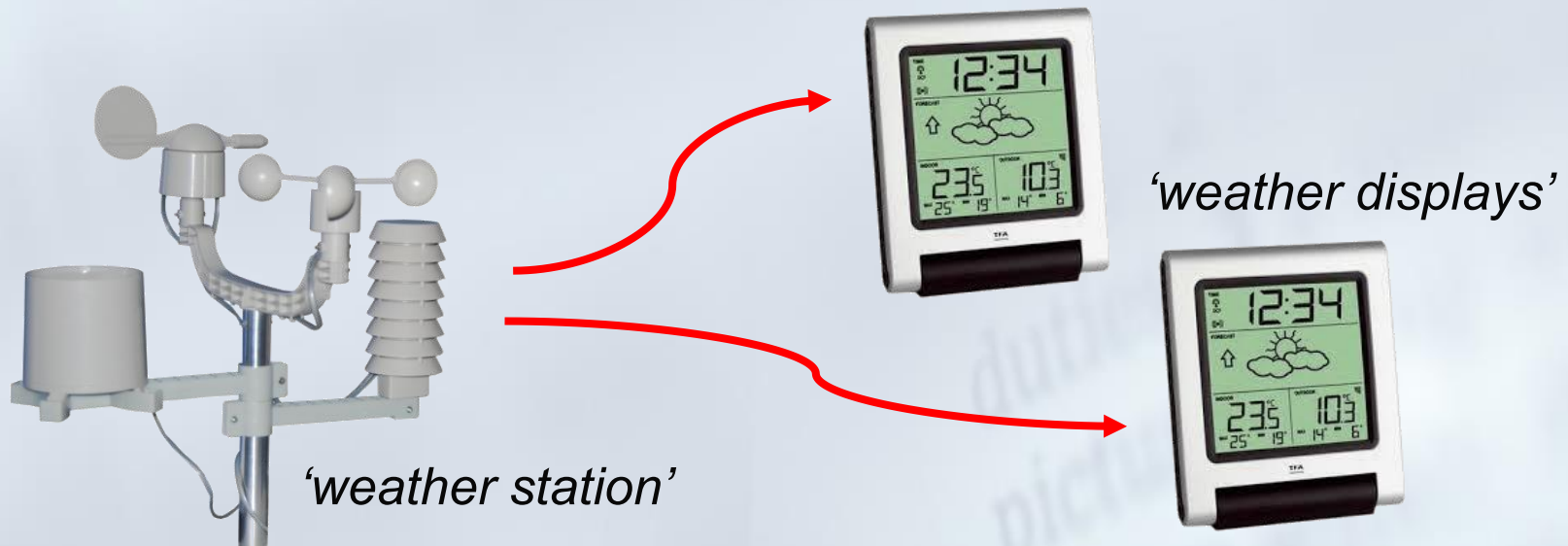
- Subject (Observable) → sends updates
- Observers → receives updates



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.'

Example application

- Weather station (Subject/Observable)
- Multiple weather displays (Observers)
- The weather displays must be updated when a new measurement has been performed





This is the information about the weather.

The weather displays are the observers; these will display the weather information.

All weather displays are being updated here.

What's wrong with this implementation?

```
public class WeatherStation
{
    public float Temperature { get; set; }
    public float Humidity { get; set; }
    public float Pressure { get; set; }

    private WeatherDisplay display1, display2;

    public WeatherStation(WeatherDisplay display1,
                          WeatherDisplay display2)
    {
        this.display1 = display1;
        this.display2 = display2;
    }

    public void MeasurementChanged() {
        // update weather information (read sensors)
        ReadTemperature();
        ReadHumidity();
        ReadPressure();

        // update all displays
        display1.Update(Temperature, Humidity, Pressure);
        display2.Update(Temperature, Humidity, Pressure);
    }

    // methods to read actual (sensor) values
    private void ReadTemperature() { }
    private void ReadHumidity() { }
    private void ReadPressure() { }
}
```

Make number of observers variable...

- Every weather display should be able to subscribe and unsubscribe itself for updates (to the weather station)
(Observer: subscriber)
- The weather station has a list of subscribers (observers), and updates each weather display when a new measurement has been performed
(Subject: publisher)

The weather displays are now variable; a dynamic list is used.

A weather display can subscribe/unsubscribe.

All weather displays are updated here.

```
public class WeatherStation
{
    public float Temperature { get; set; }
    public float Humidity { get; set; }
    public float Pressure { get; set; }

    private List<WeatherDisplay> weatherDisplays = new List<WeatherDisplay>();

    public void RegisterObserver(WeatherDisplay display) {
        weatherDisplays.Add(display);
    }

    public void RemoveObserver(WeatherDisplay display) {
        weatherDisplays.Remove(display);
    }

    public void NotifyObservers() {
        foreach (WeatherDisplay display in this.weatherDisplays) {
            display.Update(Temperature, Humidity, Pressure);
        }
    }

    public void MeasurementChanged() {
        // update weather information (read sensors)
        ReadTemperature();
        ReadHumidity();
        ReadPressure();

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



Weather display

Observable is passed via constructor, and 'this' Observer subscribes itself.

This Update method will be called by the WeatherStation.

```
public class WeatherDisplay
{
    private WeatherStation weatherStation;

    public WeatherDisplay(WeatherStation weatherStation)
    {
        this.weatherStation = weatherStation;
        weatherStation.RegisterObserver(this);
    }

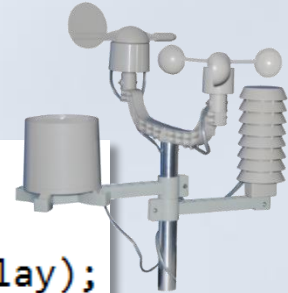
    public void Update(float temperature, float humidity, float pressure)
    {
        // display weather information
        Console.WriteLine(String.Format("T: {0}, H: {1}, P: {2}",
            temperature, humidity, pressure));
    }
}
```



Interfaces Observer pattern

- Usually an interface 'ISubject' is used for the Subject (WeatherStation)

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



- Usually an interface 'IObserver' is used for Observer (WeatherDisplay)

```
public interface IObserver
{
    void Update(float temperature, float humidity, float pressure);
}
```



Weather display (IObserver)

WeatherDisplay is no longer linked to an specific class (WeatherStation) but to an interface (ISubject).

This means that a WeatherStation can be replaced by another 'ISubject'-object.

```
public class WeatherDisplay : IObserver
{
    private ISubject weatherStation;

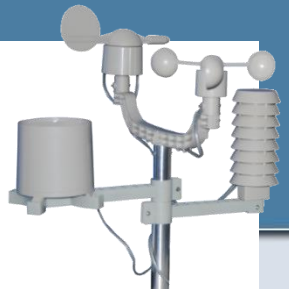
    public WeatherDisplay(ISubject weatherStation)
    {
        this.weatherStation = weatherStation;
        weatherStation.RegisterObserver(this);
    }

    public void Update(float temperature, float humidity, float pressure)
    {
        // display weather information
        Console.WriteLine(String.Format("T: {0}, H: {1}, P: {2}",
            temperature, humidity, pressure));
    }
}
```



WeatherStation is no longer linked to a specific class (WeatherDisplay) but to an interface (IObserver).

This means that a WeatherDisplay can be replaced by another 'IObserver'-object.



```
public class WeatherStation : ISubject
{
    public float Temperature { get; set; }
    public float Humidity { get; set; }
    public float Pressure { get; set; }

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

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

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

    public void NotifyObservers() {
        foreach (IObserver display in this.weatherDisplays) {
            display.Update(Temperature, Humidity, Pressure);
        }
    }

    public void MeasurementChanged() {
        // update weather information (read sensors)
        ReadTemperature();
        ReadHumidity();
        ReadPressure();

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

Summary

- The 'Template Method' pattern defines a fixed algorithm in the base class, with some steps undefined, these steps must be implemented in derived classes
- The 'Observer' pattern creates a 'weak coupling' between 'state' objects (subjects) and 'listeners' (observers)
(subject does not know who's listening...)

Assignments

- Moodle: 'Week 2 assignments'