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)*

Singleton pattern

The Singleton Pattern (GoF): *'ensures a class has only one instance, and provides a global point of access to it.'*

- Afdwingen dat er maar één instantie van een class aangemaakt kan worden
- De volgende code moet dus resulteren in hetzelfde object

```
static void Main(string[] args)
{
  OneOfAKind item1 = new OneOfAKind();
  OneOfAKind item2 = new OneOfAKind();

  // ...
}
```

(één object op de heap, bereikbaar door zowel item1 als item2)

Singleton pattern

- Hoe voorkom je dat er meer dan één object aangemaakt wordt?!?
- Via de constructor (new) maak je een object aan, dus…
- …maak de constructor private, maar dan…?
- zorg voor een static methode om een instantie te leveren

Singleton Pattern - voorbeeld

```
public class OneOfAKind
{
  private OneOfAKind()
  {
    // ...
  }
}
```

De constructor maken we private, zodat deze niet gebruikt kan worden.

```
static void Main(string[] args)
{
  OneOfAKind item1 = new OneOfAKind();
  OneOfAKind item2 = new OneOfAKind();

  // ...
}
```

OneOfAKind.OneOfAKind()

Error:
'SingletonPattern.OneOfAKind.OneOfAKind()' is inaccessible due to its protection level

Singleton Pattern - voorbeeld

```
public class OneOfAKind
{
  private static OneOfAKind uniqueInstance;

  // constructor is not available
  private OneOfAKind() { }

  // static method that delivers a unique instance
  public static OneOfAKind GetInstance()
  {
    if (uniqueInstance == null)
      uniqueInstance = new OneOfAKind();

    return uniqueInstance;
  }
}
```

Een static member bevat het unieke object.

Via een (public) static methode kan de (unieke) instantie opgevraagd worden.

Alleen als er nog geen instantie is, dan wordt deze aangemaakt.

```
class Program
{
  static void Main(string[] args)
  {
    OneOfAKind item1 = OneOfAKind.GetInstance();
    OneOfAKind item2 = OneOfAKind.GetInstance();

    if (item1.Equals(item2))
      Console.WriteLine("items zijn hetzelfde");
    else
      Console.WriteLine("items zijn niet hetzelfde");

    Console.ReadKey();
  }
}
```

Singleton Pattern - voorbeeld 2

Een concreet voorbeeld is bv 'Preferences' (voorkeuren) die door een gehele applicatie te benaderen moet zijn.

We willen zo'n object niet steeds moeten doorgeven.

```
public class Preferences
{
    private static Preferences uniqueInstance;
    private Dictionary<string, string> settings;

    // constructor is not available
    private Preferences()
    {
        settings = new Dictionary<string, string>();
    }

    // static method that delivers a unique instance
    public static Preferences GetInstance()
    {
        if (uniqueInstance == null)
            uniqueInstance = new Preferences();

        return uniqueInstance;
    }
```

Singleton Pattern - voorbeeld 2

```
    public static Preferences GetInstance() {
        if (uniqueInstance == null)
            uniqueInstance = new Preferences();

        return uniqueInstance;
    }

    public void AddSetting(string key, string value) {
        if (settings.ContainsKey(key))
            throw new Exception($"setting with key '{key}' already exists!!");
        settings.Add(key, value);
    }

    public string GetSetting(string key) {
        if (!settings.ContainsKey(key))
            throw new Exception($"no setting with key '{key}' available!!");
        return settings[key];
    }
}
```

Singleton Pattern - voorbeeld 2

```
void Start()
{
    try
    {
        //Preferences p = new Preferences();    // not possible
        Preferences p = Preferences.GetInstance();
        p.AddSetting("configfile", "config.txt");
        p.AddSetting("passwordfile", "passwords.txt");
        p.AddSetting("databasehost", "localhost");

        Preferences p2 = Preferences.GetInstance();
        Console.WriteLine(p2.GetSetting("databasehost"));
    }
    catch (Exception exp)
    {
        Console.WriteLine($"Exception occured: {exp.Message}");
    }

    Console.ReadKey();
}
```

Deze voorkeuren worden ergens in de applicatie aangepast…

…en elders (op meerdere plekken) ingelezen.

State pattern

The State Pattern (GoF): *'allows an object to alter its behavior when its internal state changes. The object will appear to change its class.'*

- Gedrag is afhankelijk van de huidige status van het object
- Bij de Strategy pattern wordt het gedrag door een externe partij gewijzigd, bij de State pattern wordt het intern aangepast

State pattern – een voorbeeld

- Kauwgomballen automaat
- Akties?

  → MuntInwerpen, MuntUitwerpen, HendelDraaien
- Toestanden?

  → GeenMuntAanwezig, MuntAanwezig,
  KauwgombalVerkocht, Uitverkocht

State pattern – een voorbeeld

- Hoe te implementeren?

We kunnen een enumeratie gebruiken voor de status.

```
class KauwgomballenAutomaat
{
  private enum AutomaatStatus { Uitverkocht, MuntAanwezig, GeenMuntAanwezig, KauwgombalVerkocht}
  private AutomaatStatus huidigeStatus;
  private uint aantalBallen;

  public KauwgomballenAutomaat(uint aantalBallen)
  {
    this.aantalBallen = aantalBallen;
    if (aantalBallen > 0)
      huidigeStatus = AutomaatStatus.GeenMuntAanwezig;
    else
      huidigeStatus = AutomaatStatus.Uitverkocht;
  }
```

De automaat krijgt een initiele (huidige) status.

State pattern – een voorbeeld

```
  public void MuntInwerpen()
  {
    if (huidigeStatus == AutomaatStatus.MuntAanwezig)
      Console.WriteLine("Je kunt niet nog een munt inwerpen.");
    else if (huidigeStatus == AutomaatStatus.Uitverkocht)
      Console.WriteLine("Je kunt geen munt inwerpen, er zijn geen kauwgomballen.");
    else if (huidigeStatus == AutomaatStatus.KauwgombalVerkocht)
      Console.WriteLine("Wacht even, er wordt reeds een kauwgombal uitgegeven.");
    else if (huidigeStatus == AutomaatStatus.GeenMuntAanwezig)
    {
      huidigeStatus = AutomaatStatus.MuntAanwezig;
      Console.WriteLine("U heeft een munt ingeworpen.");
    }
  }

  public void MuntUitwerpen()
  {
    // ...
  }

  public void HendelDraaien()
  {
    // ...
  }
}
```

In elke aktie/methode moet elke mogelijke toestand afgehandeld worden.

De mogelijke akties op de automaat worden als methoden geimplementeerd.

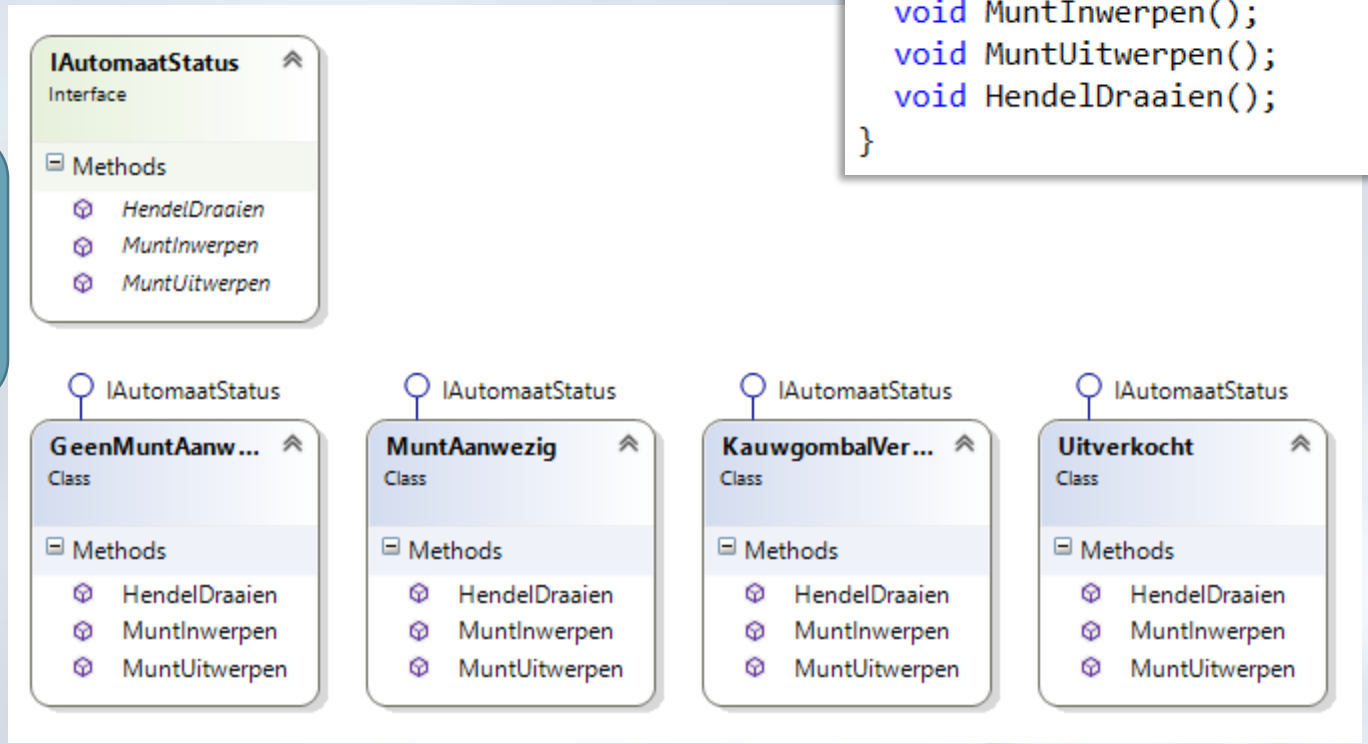
State pattern – een voorbeeld

- Maar wat nu als we een nieuwe toestand willen toevoegen? Dan moeten alle akties (methoden) aangepast worden…
- Wat ook kan: maak een 'Toestand' interface en maak voor elke toestand een class; deze toestand-classes bepalen het gedrag in die toestand

State pattern – een voorbeeld

- Toestand-interface bevat methode voor elke aktie/bewerking

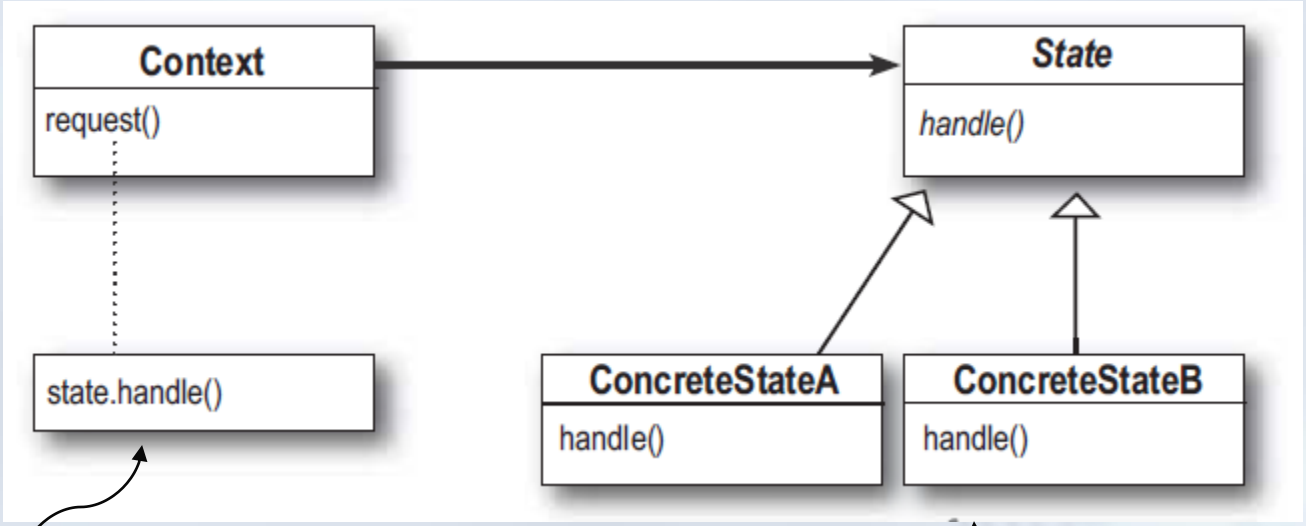
```
public interface IAutomaatStatus
{
  void MuntInwerpen();
  void MuntUitwerpen();
  void HendelDraaien();
}
```



Elk status-object handelt elke mogelijke aktie af.

State pattern

- Sterk gerelateerd aan Strategy pattern



Elke aktie wordt gedelegeerd naar een toestand-object.

De concrete classes implementeren elk (het gedrag van) een toestand.

State pattern – een voorbeeld

```
class KauwgomballenAutomaat
{
  private IAutomaatStatus geenMuntAanwezig;
  private IAutomaatStatus muntAanwezig;
  private IAutomaatStatus kauwgombalVerkocht;
  private IAutomaatStatus uitverkocht;

  private IAutomaatStatus huidigeStatus;
  private uint aantalBallen;

  public KauwgomballenAutomaat(uint aantalBallen)
  {
    geenMuntAanwezig = new GeenMuntAanwezig(this);
    muntAanwezig = new MuntAanwezig(this);
    kauwgombalVerkocht = new KauwgombalVerkocht(this);
    uitverkocht = new Uitverkocht(this);

    this.aantalBallen = aantalBallen;

    if (aantalBallen > 0)
      huidigeStatus = geenMuntAanwezig;
    else
      huidigeStatus = uitverkocht;
  }
```

Geen enumeraties meer maar aparte toestand-objecten.

Een van deze toestand-objecten is de huidige.

State pattern – een voorbeeld

```
  public void muntInwerpen()
  {
    huidigeStatus.MuntInwerpen();
  }

  public void muntUitwerpen()
  {
    huidigeStatus.MuntUitwerpen();
  }

  public void hendelDraaien()
  {
    huidigeStatus.HendelDraaien();
  }

  public void ZetStatus(IAutomaatStatus nieuweStatus)
  {
    huidigeStatus = nieuweStatus;
  }

  public IAutomaatStatus GeefMuntAanwezigStatus()
  {
    return muntAanwezig;
  }

  // ...
}
```

De akties worden doorgesluisd naar de het huidige toestand-object.

Deze methoden hebben we nodig om de toestand te veranderen (vanuit 1 v/d toestand-objecten).

State pattern – een voorbeeld

```
class GeenMuntAanwezig : IAutomaatStatus
{
  private KauwgomballenAutomaat automaat;

  public GeenMuntAanwezig(KauwgomballenAutomaat automaat)
  {
    this.automaat = automaat;
  }

  public void MuntInwerpen()
  {
    Console.WriteLine("U heeft een munt ingeworpen.");
    automaat.ZetStatus(automaat.GeefMuntAanwezigStatus());
  }

  public void MuntUitwerpen()
  {
    Console.WriteLine("U heeft geen munt ingeworpen.");
  }

  public void HendelDraaien()
  {
    Console.WriteLine("U moet eerst een munt inwerpen, om de hendel te kunnen draaien.");
  }
}
```

De automaat wordt meegegeven aan de constructor.

De toestand v/d automaat wordt gewijzigd vanuit de toestand-classes.

Samenvattend

- Singleton Pattern: afdwingen dat er met één unieke instantie van een class wordt gewerkt
- State Pattern: gedrag implementeren in aparte classes; gedrag is afhankelijk van de (huidige) status

Opdrachten

- Zie Moodle: ‘Week 5 opdrachten’