HET DECORATOR PATTERN

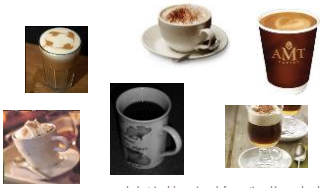
Veerle Ongenae, Wijnand Schepens

Objecten decoreren

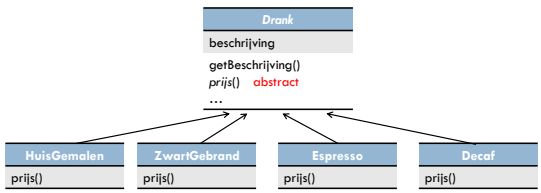
- Verantwoordelijkheden toevoegen aan objecten
  - Overerving ↔ decoreren
  - Uitbreiding at compile time ↔ uitbreiding at runtime
- Decorator Pattern

Voorbeeld

- Koffieshop
  - Verschillende soorten koffie: espresso, decafeiné, …
  - Diverse toevoegingen: melk, room, geklopte melk, soja, …



Klassen: basisschema

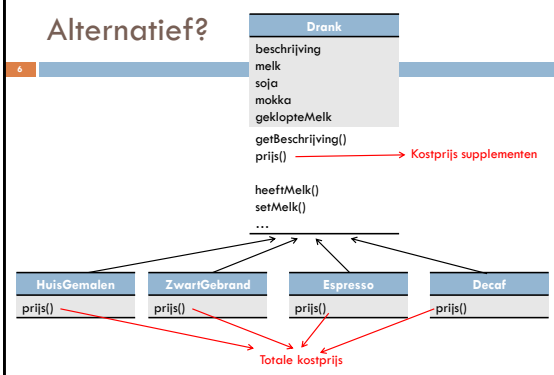


- Voor elk type koffie: aparte klasse
- Voor elk combinatie koffie+supplementen: aparte klasse?

Klassen

- Voor elk combinatie koffie+supplementen: aparte klasse?
  - Een explosie aan klassen
  - Nachtmerrie onderhoud
  - Welk ontwerpprincipes worden geschonden?
- Alternatief?
  - attributen voor elk supplement?

Alternatief?



Kostprijs

7

```
public class Drank {
    …
    public double prijs() {
        double extraKost = 0;
        if (heeftMelk())
            extraKost += melkKost;
        if (heeftSoja())
            extraKost += sojaKost;
        …
    }
    …
}
```

```
public class Decaf extends Drank {
    …
    public Decaf() {
        beschrijving="Heerlijke cafeïne"
            + " vrije koffie";
    }

    public double prijs() {
        return 1.40 + super.prijs();
    }
    …
}
```


Problemen ?

8

- Prijsveranderingen?
- Nieuwe toevoegingen?
- Nieuwe dranken?
- Twee dezelfde supplementen?
- …

Overerven van gedrag

9

- At runtime
  - Compositie en delegeren
  - Meerdere verantwoordelijkheden geven
  - Geen aanpassing van code, wel nieuwe code
- At compile time
  - Subklassen
  - Niet flexibel
  - Moeilijk onderhoudbaar
- Ontwerpprincipe: Klassen moeten gesloten zijn voor verandering en open voor uitbreiding

Open-gesloten principe

10

- Klassen moeten **open** zijn **voor uitbreiding,** maar **gesloten voor verandering.**
- Eenvoudig nieuw gedrag toevoegen
- Bestaande code NIET wijzigen
- Niet overal toepassen
  - Duur
  - Complexe code
  - Enkel op gebieden die meest kans hebben om te veranderen

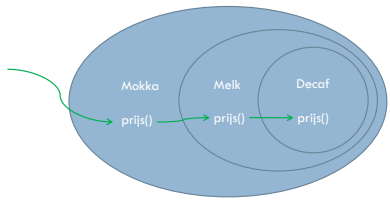
Stand van zaken

11

- Overerving → explosie van klassen
- Functionaliteit toevoegen aan basisklasse
  - Niet voor alle subklassen van toepassing
- Anders?
  - Drank 'decoreren' at runtime
  - Wrappers …

Objecten decoreren

12



Objecten decoreren

13

- Decorators hebben zelfde supertype
- Meerdere decorators om een object
- Decorator voegt eigen gedrag toe nadat
  - De rest van het werk wordt doorgegeven aan het object dat hij decoreert
- Decoratie kan op ieder moment

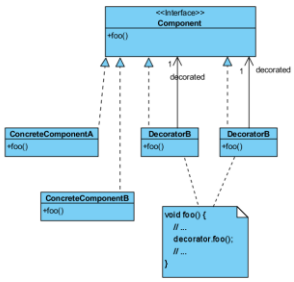
Decorator Pattern: definitie

14

- Het Decorator Pattern kent dynamisch extra verantwoordelijkheden toe aan een object.
- Decorators bieden een flexibel alternatief voor het gebruik van afgeleide klassen om functionaliteit uit te breiden.
- Ook Wrapper genoemd

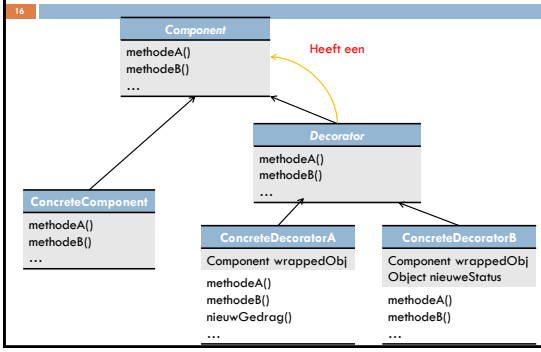
Decorator Pattern: definitie

15



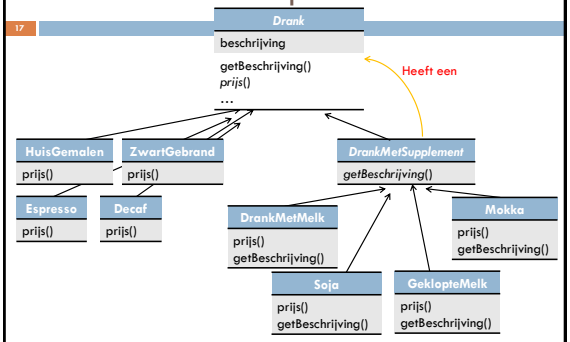
Decorator Pattern: definitie

16



Klassen: koffieshop

17



Overerving + compositie

18

- Overerving
  - Zelfde type (interface of abstracte klasse)
- Compositie
  - Toevoegen gedrag

Drank en Supplement: abstracte klassen

19

```
public abstract class Drank {
    String beschrijving = "Onbekende drank";

    public String getBeschrijving() {
        return beschrijving;
    }
    public abstract double prijs();
}

public abstract class Supplement extends Drank {
    public abstract String getBeschrijving();
}
```


Dranken

20

```
public class Espresso extends Drank {
    public Espresso () {
        beschrijving = "Espresso";
    }

    public double prijs() {
        return 1.99;
    }
}
```

```
public class Decaf extends Drank {
    public Decaf() {
        beschrijving = "Heerlijke cafeïne"
            + " vrije koffie";
    }

    public double prijs() {
        return 1.05;
    }
}
```


Supplementen

21

```
public class Melk extends Supplement {
    Drank drank;

    public Melk (Drank drank) {
        this.drank = drank;
    }
    public String getBeschrijving() {
        return drank.getBeschrijving() + ", melk";
    }
    public double prijs() {
        return drank.prijs()+ 0.10;
    }
```


Koffie opdienen

22

```
Drank drank = new Espresso();
System.out.println(drank.getBeschrijving() + " € " + drank.prijs());

Drank drank2 = new HuisGemalen();
drank2 = new Melk(drank2);
drank2 = new Melk(drank2);
System.out.println(drank.getBeschrijving() + " € " + drank.prijs());
```


Wanneer Decorator gebruiken?

23

- Dynamisch verantwoordelijkheden toevoegen aan objecten
  - Zonder invloed op andere objecten
- Verantwoordelijkheden moeten dynamisch ingetrokken kunnen worden
- Uitbreiding door afgeleide klassen is onpraktisch
  - Explosief aantal (veel combinaties mogelijk)
  - Klasse niet beschikbaar om af te leiden (verborgen, …)

Opmerkingen

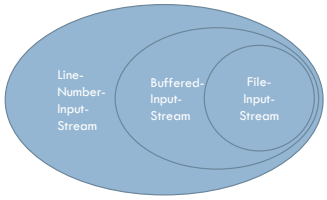
24

- Decorator ≠ component
  - Identiteit object is niet bruikbaar
- Veel kleine objecten
  - Moeilijker om leren
  - Moeilijker om te debuggen

Decorator Pattern in JDK

25

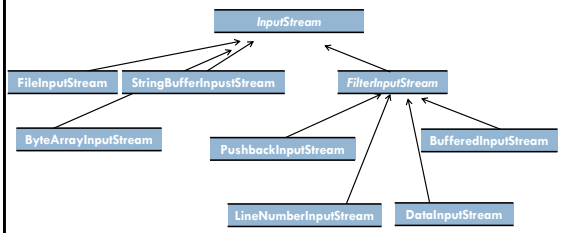
- Java I/O



- Java Swing
  - Samenstellen borders, …

Java I/O

26



Eigen I/O-decorator schrijven

27

```
public class LowerCaseInputStream extends FilterInputStream {
    public LowerCaseInputStream (InputStream in) {
        super(in);
    }
    public int read() throws IOException {
        int c = super.read();
        return (c==-1 ? c : Character.toLowerCase((char)c);
    }
    public int read(byte[] b, int offset, int len) throws IOException {
        int result = super.read(b,offset,len);
        for (int i = offset; i < offset+len; i++)
            b[i] = (byte) Character.toLowerCase((char)b[i]);
        return result;
    }
```


Voorbeeld met Reader

28

- nieuweiodecorator.LowerCaseReader
- nieuweiodecorator.Main

Samenvatting kennis

29

- OO-basis
- OO-principes
  - Isoleer wat verandert
  - Verkies compositie boven overerving
  - Programmeer naar een interface en niet naar een implementatie
  - Streef naar ontwerpen met een zwakke koppeling tussen interagerende objecten
  - Klassen moeten open zijn voor uitbreiding maar gesloten voor verandering
- OO-patterns
  - Strategy
  - Observer
  - Decorator: kent dynamisch extra verantwoordelijkheden aan een object toe. Decorators bieden een flexibel alternatief voor het gebruik van afgeleide klassen om de functionaliteit uit te breiden.