DESIGN PATTERNS: INLEIDING

Veerle Ongenae, Wijnand Schepens

Inhoud

- Praktische informatie
- Inleiding tot Design Patterns

Organisatie

- Theorie: donderdag 10u30-12u30 (!)
- Labo: (niet elke) woensdag 13u30-15u30:
  - 10/10
  - 24/10
  - 14/11
  - 28/11
  - 5/12: test
  - inhaalweek
- Puntenverdeling:
  - Mondeling examen in januari (70%)
  - Labotest (30%)

Vakinhoud

- OO, ontwerpprincipes, UML
- Strategy, Observer, Decorator, Factory, Singleton, Command, Adapter, Facade, Template Method, Iterator, Composite, State, Proxy, Compound, MVC (Hoofdstuk 1-13)
- Bridge, Builder, Chain fo Responsibility, Flyweight, Interpreter, Mediator, Memento, Prototype, Visitor (appendix boek)
- Spring (slides)
- AOP (slides)
- Extra's: testing, logging, multi-layer, … (slides)

Cursusmateriaal

- "**Train je hersens in DESIGN PATTERNS**", De Head First Methode, Freeman & Freeman
- **https://intranet.tiwi.be**
  - Slides theorie,
  - labo-opdrachten, …
- Extra:
  "Design Patterns, Elementen van herbruikbare objectgeoriënteerde software",
  Gamma, Helm, Johnson, Vlissides
  - Catalogus van design patterns
  - GoF (Gang of Four)

Inhoud

- Praktische informatie
- Inleiding tot Design Patterns

Inleiding tot Design Patterns

7

- Iemand heeft je probleem al eens opgelost
  - Hergebruik van kennis en ervaring
- Gebruik en voordelen Design Patterns
- Belangrijke OO-ontwerpprincipes
- Voorbeeld

Historiek

8

- 1977-1979:
  architecturaal concept (Christopher Alexander)
- 1994:
  "Design Patterns: Elements of Reusable Object-Oriented Software" (the Gang of Four)
- 2004:
  "Head First Design Patterns"
- patterns voor sotfware architecture, entreprise applications, service oriented architecture (SOA), …

Hergebruik

9

- Software-ontwikkeling
  - Tijdsintensief
  - Duur
- Doel
  - Eenmalig ontwerp en ontwikkeling
  - Veelvuldig hergebruiken
- Kost
  - 9 keer duurder dan code voor een prototype
    - Herbruikbaar, getest, gedocumenteerd, performant, bugvrij, …

Hergebruik

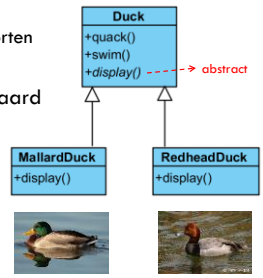
10

- Herbruikbare modules/bibliotheken
  - Bv. Java Collection Framework
- Componenten
  - Bouwblokken voor bepaalde standaarden
  - (Her)gebruikt in grotere systemen
  - Bv. servlets, beans, …
- Patronen: hergebruik van ontwerp
  - Iemand heeft je probleem al eens opgelost
    - Gelijkaardige situatie
    - Oplossingen ontwikkeld in de loop van de tijd
      - Streven naar flexibiliteit en hergebruik

Voorbeeld

11

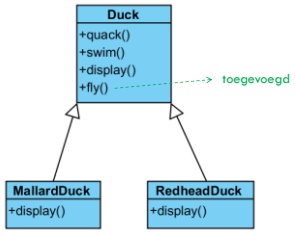
- Simulatie eendenvijver
  - Verschillende eendensoorten
  - Zwemmen en kwaken
- Gerealiseerd met standaard OO-technieken
  - Overerving
  - Polymorfisme
  - Abstractie



Aanpassing

12

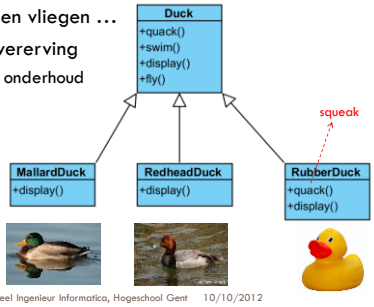
- Eenden moeten ook vliegen



Probleem met aanpassing

13

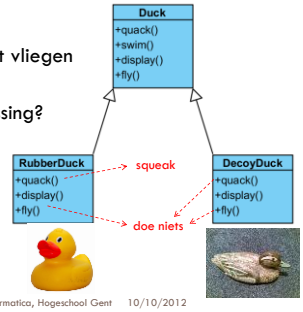
- Niet alle eenden vliegen …
- Hergebruik: overerving
  - Probleem met onderhoud (aanpassing)



Oplossing?

14

- fly() overschrijven
- Wat met eenden die niet vliegen en kwaken?
- Nadelen van deze oplossing?



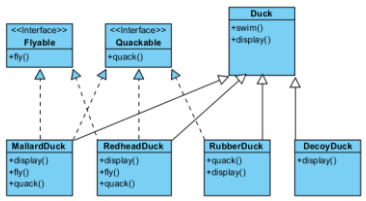
Nadelen gebruik overerving

15

- Nadelen gebruik overerving
  - Gedrag eenden beschrijven
- Duplicatie code in afgeleide klassen
- Verandering gedrag at runtime moeilijk/onmogelijk
- Gedrag alle eenden kennen is moeilijk
- Veranderingen kunnen onbedoeld andere eenden beïnvloeden

Interfaces?

16



Nadelen?

Oplossingen?

17

- Niet alle afgeleide klassen hebben vlieg- of kwaakgedrag
  - Geen overerving
- Interfaces
  - Geen hergebruik van code
    - Onderhoud = nachtmerrie
- Software zo bouwen
  - Veranderingen → minimale gevolgen
- Pas OO-softwareontwerpprincipes toe

Softwareontwikkeling ↔ verandering

18

- Applicatie moet na verloop van tijd (altijd) veranderen
- Ontwerpprincipe: "**Encapsulate what varies**"
  - Bepaal de aspecten van je applicatie die variëren en scheid deze van de aspecten die hetzelfde blijven
- Basis van vele design patterns
- Voordelen
  - Minder onbedoelde effecten na verandering
  - Grotere flexibliteit

Voorbeeld aanpassen

19

- Bepaal de aspecten van je applicatie die variëren en scheid deze van de aspecten die hetzelfde blijven
- kwaak() → verzameling klassen
  - Één voor elk "soort kwaken": kwaken, piepen, stil zijn, …
- vlieg() → verzameling klassen

Ontwerp eendengedrag

20

- Ontwerp verzameling klassen??
  - Flexibel
  - Gedrag toekennen
  - Eventueel gedrag dynamisch veranderen
- Ontwerpprincipe
  - Programmeer naar een interface, niet naar een implementatie
- Elke type gedrag = interface
  - VliegGedrag, KwaakGedrag, …
- Realisatie gedrag = implementatie interface
  - VliegenMetVleugels, NietVliegen, …
  - Kwaken, Piepen, Zwijgen, …
  - Klassen die een gedrag voorstellen

Ontwerp eendengedrag

21

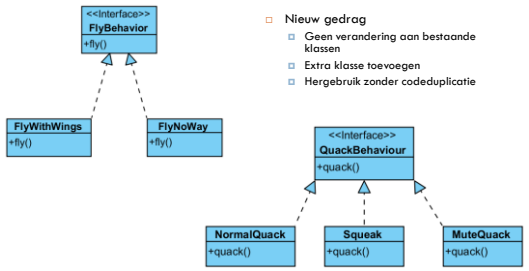
- Eerst
  - Concrete implementatie superklasse
  - Specifiek implementatie afgeleide klassen
  - Geen ruimte voor verandering

  Programmeren naar een implementatie
- Nieuw ontwerp
  - Gedrag voorgesteld door interface
  - Realisatie gedrag
    - Aparte klasse
  - Wel ruimte voor verandering

  Programmeren naar een interface

Implementatie eendengedrag

22



- Nieuw gedrag
  - Geen verandering aan bestaande klassen
  - Extra klasse toevoegen
  - Hergebruik zonder codeduplicatie

Intermezzo

23

- Ontwerpprincipe: "Programmeer naar een interface"
- Engels: "Program to an interface"
- kan ook abstracte klasse ipv interface zijn ("abstractie")
  => "Programmeren naar een supertype"

Voorbeeld

24

- Programmeren naar een implementatie:

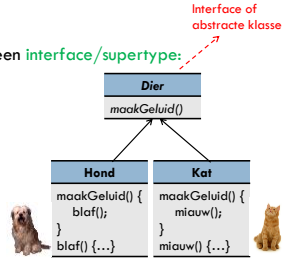
```
Hond hond = new Hond();
hond.blaf();
```

- Beter: programmeren naar een interface/supertype:

```
Dier dier = new Hond();
dier.maakGeluid();
```

- Of, nog beter:

```
Dier dier = getDier();
dier.maakGeluid();
```



Gedrag delegeren

25

```
class Duck {
    FlyBehavior   flyBehavior;
    QuackBehavior quackBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }
    public void performFly() {
        flyBehavior.fly();
    }

    // ...
}
```

attributen

delegeren

instellen

```
class MallardDuck extends Duck {
    public MallardDuck() {
        flyBehavior = new FlyWithWings();
        quackBehavior = new NormalQuack();
    }
    // ...
}
```


Gedrag dynamisch instellen

26

```
class Duck {
    FlyBehavior   flyBehavior;
    QuackBehavior quackBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }
    public void performFly() {
        flyBehavior.fly();
    }

    public void setFlyBehavior(FlyBehavior fb) {
        flyBehavior = fb;
    }
    // analoog voor setQuackBehavior...
}
```

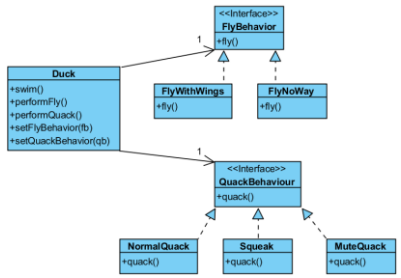
Setters maken het mogelijk gedrag dynamisch (= at runtime) in te stellen.

Voorbeeld:

```
duck.performFly();
duck.setFlyBehavior(
    new FlyRocketPowered());
duck.performFly();
```

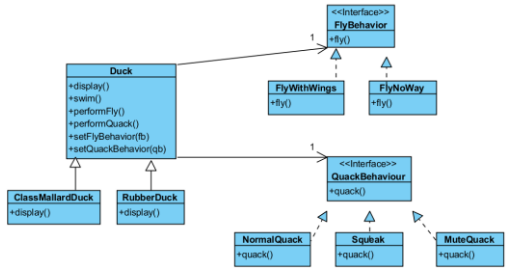

Samenvatting

27



Of, met abstracte display()

28



HEEFT-EEN is soms beter dan IS-EEN

29

- Ontwerpprincipe
  - Geef aan compositie de voorkeur boven overerving
- Eend krijgt gedrag door samenvoeging i.p.v. overerving
  - Meer flexibiliteit
    - Gedrag at runtime aanpassen
    - Hergebruik gedragcode
      - Vermijdt gedupliceerde code
      - Beter onderhoudbaar
    - Eenvoudig uitbreidbaar

Strategy Pattern

30

- Voorbeeld toepassing van Strategy Pattern
- Strategy Pattern
  - Definieert een familie algoritmen
  - Isoleert ze
  - Maakt ze uitwisselbaar
  - Maakt het mogelijk om
    - Algoritmen te veranderen
    - Los van de client die ze gebruikt
- Andere naam: Policy
- Scheiding object en zijn gedrag

Gebruik Strategy Pattern

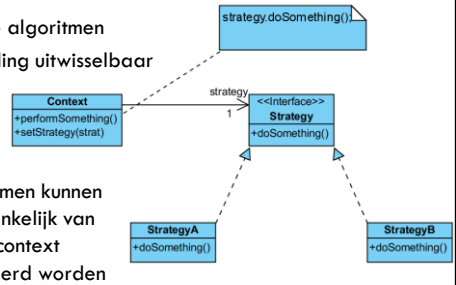
31

- Veel gerelateerde klassen die alleen verschillen in hun gedrag
  - Vb. eenden
- Nood aan verschillende varianten van een algoritme
  - Vb. sorteringsalgoritme
- Een algoritme gebruikt gegevens (of structuren) waar clients niets over horen te weten.
  - Vb. prijsberekeningen, berekeningen inkomstbelastingen, …
- Een klasse veel gedrag definieert en dit gedrag zijn reacties bepaalt.
  - Vb. vechtspel: uitrusting bepaalt vechtstijl

Strategy Pattern

32

- Familie algoritmen
- Onderling uitwisselbaar



- Algoritmen kunnen onafhankelijk van client/context veranderd worden

Design Patterns

33

- Goede OO-systemen
  - Herbruikbaar
  - Uitbreidbaar
  - Onderhoudbaar
- Patterns = bewezen OO-ervaring
- Patterns
  - Geen code
  - Algemene structuur

Waarom Design Patterns?

34

- Gemeenschappelijk woordenschat met andere programmeurs
  - Eenvoudige communicatie
  - Minder woorden
- Denken op pattern-niveau i.p.v. op object-niveau
- GoF-boek beschrijft patterns
  - Naam
  - Probleem
  - Oplossing
  - Gevolgen

Gebruik patterns

35

- Uit je hoofd leren
- Herkennen waar je ze kan toepassen
- Hergebruik van ervaring i.p.v. hergebruik van code

Samenvatting kennis

36

- OO-basis
  - Abstractie
  - Inkapseling
  - Polymorfisme
  - Overerving
- OO-principes
  - Isoleer wat verandert
  - Verkies compositie boven overerving
  - Programmeer naar een interface en niet naar een implementatie
- OO-patterns
  - Strategy: definieert een familie algoritmen, kapselt ieder algoritme in en maak deze uitwisselbaar. Strategy laat toe algoritmen onafhankelijk van de clients die ze gebruiken te veranderen.