Stockholms universitet

Programmering 2

Föreläsning 2: Arv och delegering

Isak Samsten, VT22

Stockholms universitet

Arv

- Arv tillåter oss alltså att dela beteenden genom att objekt "är-en/ett"
  - Exempel:
    - `Katt är-ett Djur`
    - `Ko är-ett Djur`
    - `ArrayList är-en AbstractCollection`

Stockholms universitet

Exempel

- Änder
  - Kan simma
  - Kan kvacka
  - Kan visas på skärmen
- Tre typer: Gräsand, Skedand och Plastanka
- Krav kan ev. ändras

Stockholms universitet

Första versionen

- Det fungerar som vi vill!
- Men, nytt krav: det visar sig att änder kan flyga!

Stockholms universitet

Flyga

- En ny metod i `And`-interfacet
  - Flyg
- Exempel

Stockholms universitet

Flyga: första versionen

- Änder kan nu flyga
- Men vad gör vi med Plastanka?
  - Lämnar implementationen blank: de kan inte flyga
- **Exempel**

Stockholms universitet

Flyga: fler klasser?

- En ny anka: Träanka
  - Kan inte heller flyga
  - Kan inte kvacka
- Lämna både flyga och kvacka tomt?
- Fler subklasser?
  - Duplicera beteendet "kan inte flyga"
  - Eller göra det till standard (och duplicera "kan flyga"-beteendet)

Stockholms universitet

Vilka nackdelar har arv här?

- [A] Koden dupliceras mellan subklasser
- [B] Förändringar av beteende under körning försvåras
- [C] Svårt att få kännedom om samtliga beteenden
- [D] Änder kan inte flyga och kvacka samtidigt
- [E] Förändringar kan oavsiktligt påverka andra änder

Rätt svar: A, B, C, E

Stockholms universitet

Kan vi lösa det med interfaces?

- Två interfaces:
  - `Flygande`
  - `Kvackande`
- Exempel

Stockholms universitet

Är det en bra lösning?

- Delvis:
  - Vi kan ha metoder såsom:

```
public void resa(Flygbar flygbar) {
    // Flyg någonstans..
}
```

Stockholms universitet

```
class And {}

interface Flygande {}

interface Kvackande {}

class FlygAnd extends And implements Flygande {}

class FlygKvackAnd extends FlygandeAnd implements Kvackande {}
```

Stockholms universitet

Är det en bra lösning?

- Löser vi grundproblemet?
  - Koden kommer fortfarande dupliceras…
  - Vi kan inte ändra beteende under körning (om så efterfrågas)
  - Ju fler subklasser desto fler potentiella beteenden - svårt att få en överblick
  - Vi kan inte ha objekt av samma klass men med olika beteenden

Är arv svaret?

Vi vet att **inte alla** änder kan flyga eller kvacka

Så arv är kanske **inte svaret** här!

Stockholms universitet

Interfaces löser en del av problemet

- Med hjälp av `Flygande` och `Kvackande` löser vi en del av problemet:
  - Änder som inte ska flyga eller kvacka kan inte kvacka eller flyga
- Men:
  - Vi kan inte återanvända kod
  - Lösningen är svår att underhålla:
    - Fler än ett beteende bland flygande änder?

Stockholms universitet

Designprincip 1

Identifiera de aspekter av programmet som varierar och separera dem från de som är konstanta

- Om vi har aspekter som ändras (med nya krav) så har vi beteenden som ska separeras från de som är konstanta

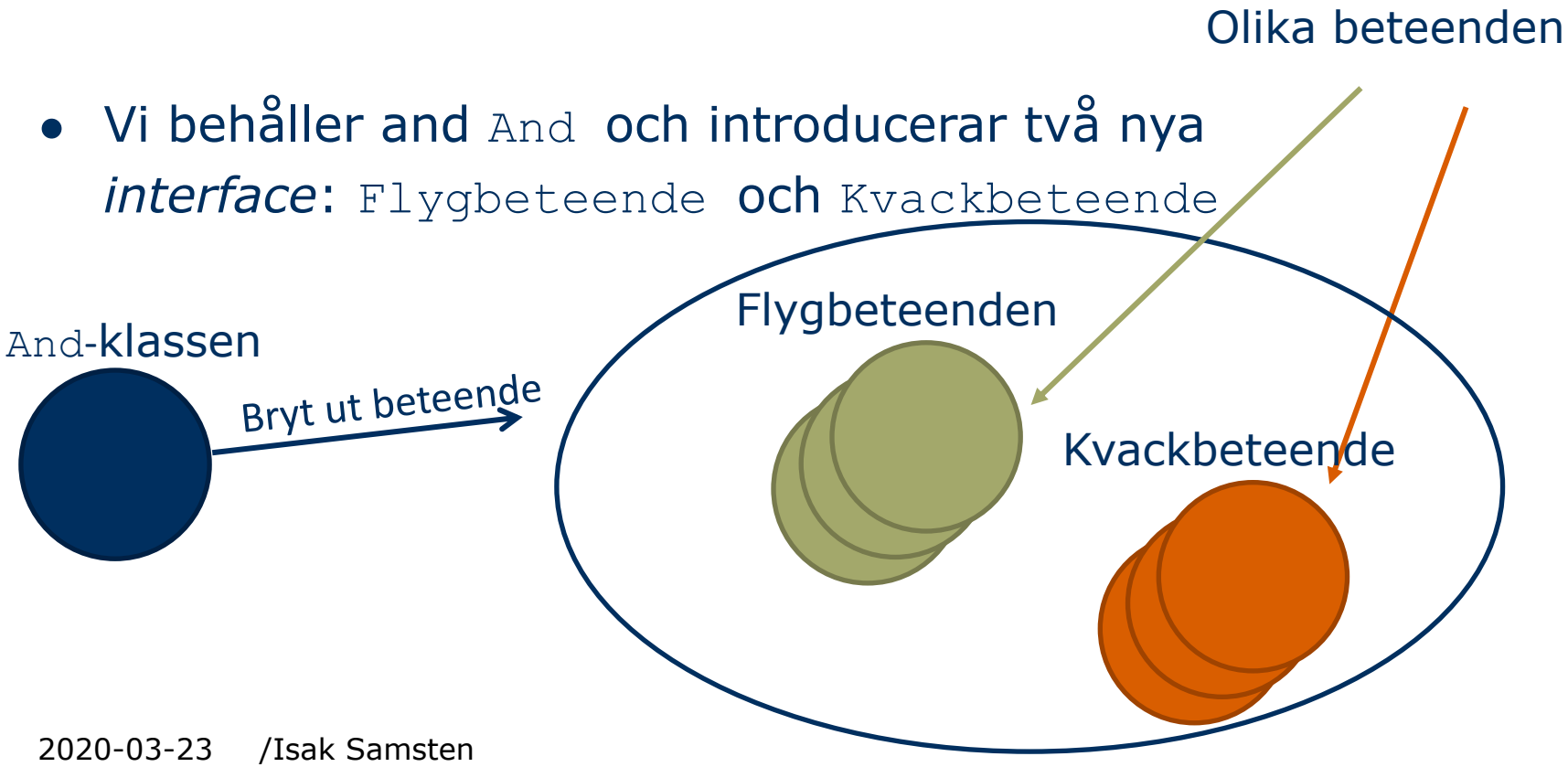
Stockholms universitet

Delegering med arv

- Vi vet att änder har två delar av implementation som varierar: flygande och kvackande
- Så vår lösning blir att "bryta ut" det som ändras och behålla det som är samma

Nya klasser!

- Vi behåller and `And` och introducerar två nya *interface*: `Flygbeteende` och `Kvackbeteende`



Stockholms universitet

Hur ska vi lösa det?

- Vi vet att vi vill kunna *tilldela* beteenden till änder
- Och hålla implementationen flexibel
- Som exempel: vi vill skapa en viss **And** med ett **särskilt** flygbeteende
  - Och varför inte tillåta att beteenden ändras när vi ändå håller på?
  - Med andra ord: vi vill ha setters och getters som kan ändra beteendet!

Stockholms universitet

Designprincip 2

Programmera mot ett interface inte mot en implementation

- Inte att förväxla med *interface* i Java
- Vi ska nyttja *polymorfism* så att vi i så stor utsträckning som möjligt programmerar mot en (abstrakt) supertyp istället för en (konkret) implementation

```
class Djur {
    abstract void låt();
}

class Hund {
    void låt() { skäll(); }
    void skäll() { … }
}

class Katt{
    void låt() { mjau(); }
    void mjau() { … }
}
```

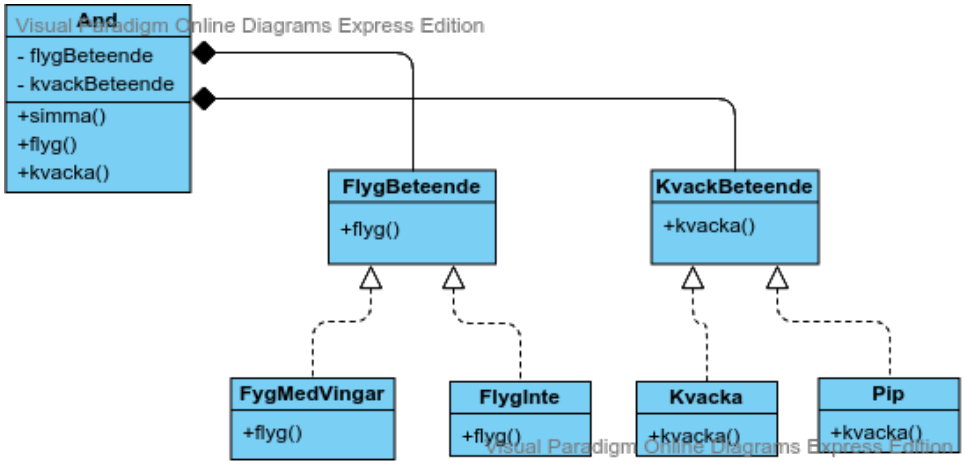
Stockholms universitet

Dåligt:

```
Hund h = new Hund();
h.skäll();
```

Bra:

```
Djur djur = getDjur();
djur.låt()
```


Visual Paradigm Online Diagrams Express Edition
And
- flygBeteende
- kvackBeteende
+simma()
+flyg()
+kvacka()
FlygBeteende
+flyg()
KvackBeteende
+kvacka()
FygMedVingar
+flyg()
FlygInte
+flyg()
Kvacka
+kvacka()
Pip
+kvacka()
Visual Paradigm Online Diagrams Express Edition

Stockholms universitet

Vår design

- Vi kan återanvända flygbeteendet för andra klasser: kanske Fladdermus eller Fågel
- När vi behöver lägga till andra beteenden behöver vi inte ändra i existerande klasser
- Beteendena är inte längre "dolda" i våra And-klasser

Stockholms universitet

Gör klart exemplet!

2020-03-23 /Isak Samsten

Stockholms universitet

Vad har vi åstadkommit?

- Vi kan ha olika änder (med olika utseenden)
  - Dessa kan ha olika beteenden och dynamiskt ändra dem
- Vi kan tänka på Flygbeteende och Kvackbeteende som familjer med *algoritmer* som kan ändras och utökas

Stockholms universitet

Delegering

- Det betyder att vi har brutit ut "*är-en*"-relationer och ersatt dessa med "*har-en*"-relation
  - Varje **And** *har-ett* Flygbeteende och *har-ett* Kvackbeteende
- När vi samordnar två klasser på det här sättet har vi komposition (eng. composition)
  - Det här är en **mycket** viktig objektorienterad princip!

Stockholms universitet

Designprincip 3

Välj komposition framför arv

- Genom att skapa system med hjälp av komposition ger oss större flexibilitet
- Det tillåter inte bara att vi "kapslar in" beteenden i egna klasser
  - Det tillåter oss också att **ändra beteende under körning**

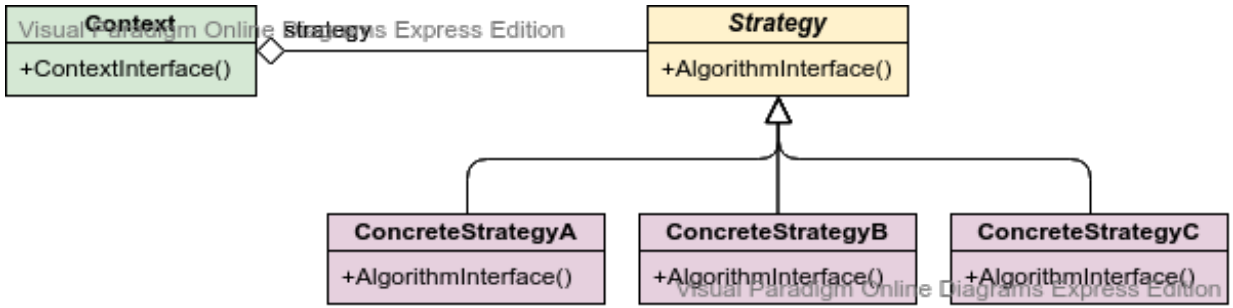
Stockholms universitet

Designmönster

- Vi har lärt oss vårt första mönster:

  The Strategy pattern

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.


Context
+ContextInterface()
Strategy
Strategy
+AlgorithmInterface()
ConcreteStrategyA
+AlgorithmInterface()
ConcreteStrategyB
+AlgorithmInterface()
ConcreteStrategyC
+AlgorithmInterface()
Visual Paradigm Online Diagrams Express Edition

Stockholms universitet

Beteenden är ju samma för många änder

- Vi skapar en massa beteendeinstanser
- Men de gör alla samma sak
- Har samma tillstånd
- Onödigt!

Stockholms universitet

Singleton pattern

- Endast en instans per beteende
- I Java:
  - Dölj klassens konstruktor
  - Definiera en statisk operation för att hämta den enda instansen av objektet

Stockholms universitet

Designmönster

- Vi har lärt oss vårt andra mönster:

Singleton pattern