Design

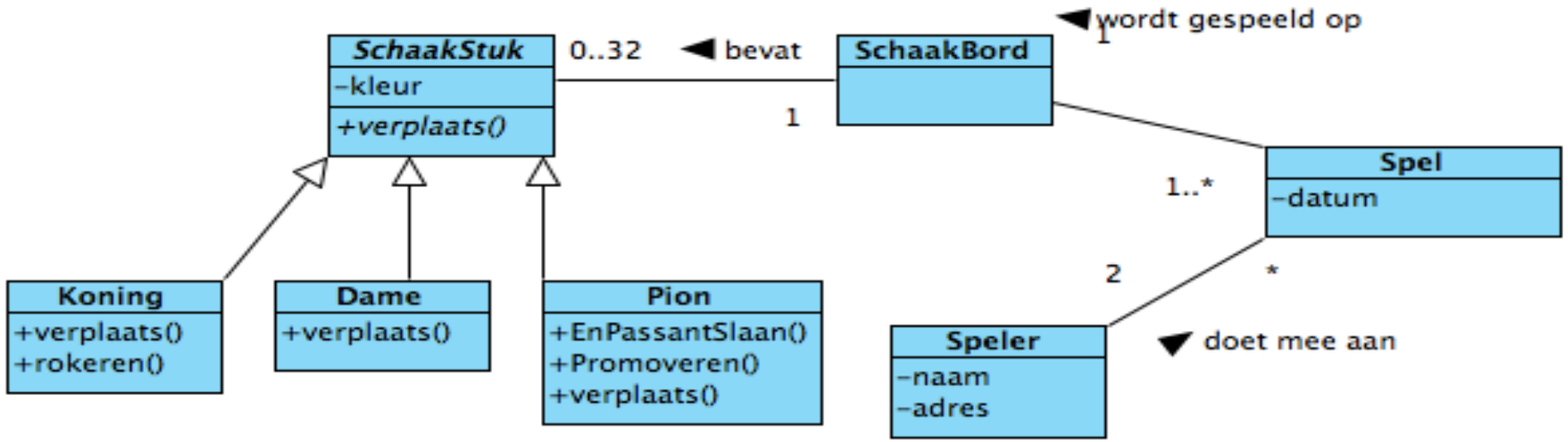
Programmeertechnieken, Tim Cocx

Universiteit
Leiden
The Netherlands

Analysis Klassendiagram

- Een eerste schets van de opbouw van het systeem
- Alle gebruikte termen komen uit de bedrijfspraktijk
- Beperkt Model
  - Operaties (alleen uit bedrijfspraktijk, geen interne methoden)
  - Overerving (alleen als hiërarchie bestaat in bedrijfspraktijk)
- Doel: gebruik bij correct krijgen van precieze Requirements
- Output: gebruikt bij een *design klassendiagram* en / of database klassendiagram

Analysis vs. Design Klassendiagram



Alle Klassen zijn voor de business (klant / gebruiker) herkenbaar. De overerving is ook intuïtief

Kwaliteit van ontwerp

- Een systeemontwerp
  - Is onderdeel van een ontwikkelingscyclus (WS 1)
  - Dient als input bij het maken van ‘code’
- Het is belangrijk dat al bij het ontwerp kwalitatief goede keuzes worden gemaakt.

Ontwerp

Kwaliteitsdoelen

- Software wordt niet 1 keer geschreven en daarna nooit meer veranderd (zoals een boek)
- Software wordt uitgebreid, aangepast, geupdate, onderhouden, gecorrigeerd, geport.
- Dit wordt gedaan door verschillende mensen gedaan op verschillende momenten
  - Vaak veel mensen
  - Vaak over een lange tijd
- Dit gegeven moet worden ondersteund.

Kwaliteitsdoelen

- *Begrijpelijkheid* (*understandability*):
  - Iedere vakman moet binnen afzienbare tijd begrijpen hoe het systeem in elkaar zit
- *Uitbreidbaarheid* (*Expandability*):
  - Het systeem moet makkelijk voorzien kunnen worden van nieuwe functionaliteit
- *Aanpasbaarheid* (*adaptability*):
  - Het systeem moet makkelijk aangepast kunnen worden aan veranderde omstandigheden

Kwaliteitscriteria: Compleetheid

- Een Klasse verricht alle taken die de naam doet vermoeden
  - Naamgeving is dus belangrijk!
- Dit heet *compleetheid* (*Completeness*).
  - Doel: hoog

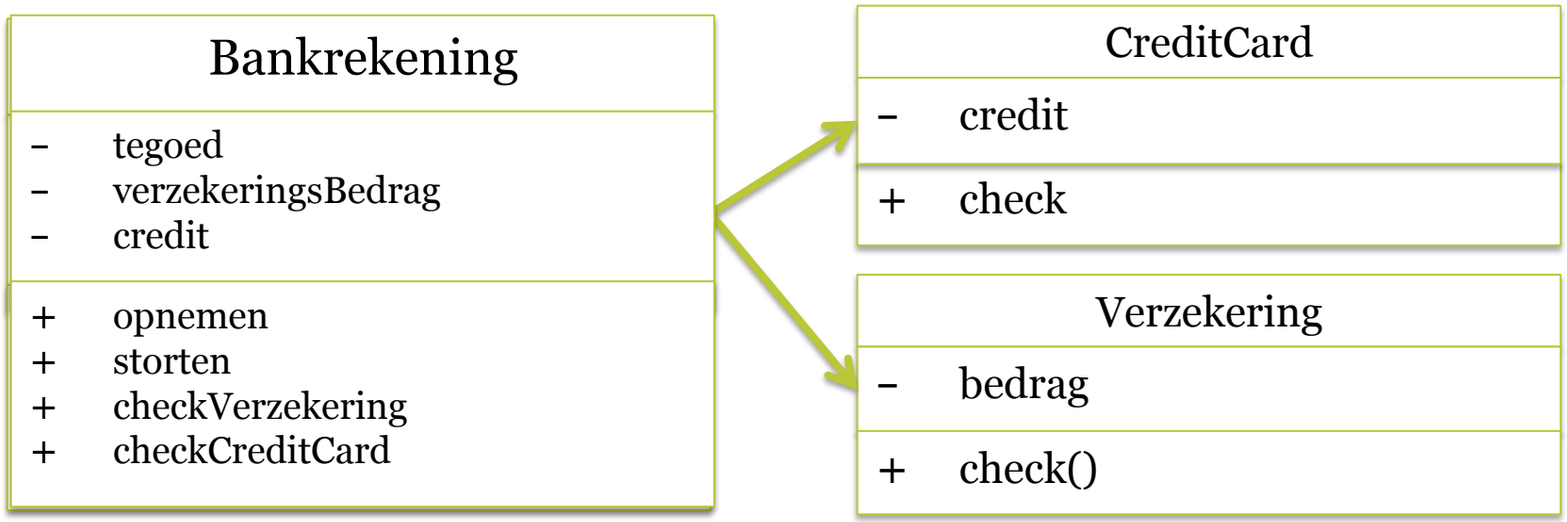
Bankrekening
– tegoed
+ opnemen + storten

Kwaliteitscriteria: Uitsluitendheid

- Een klasse verricht **alleen** de taken die de naam doet vermoeden.
  - Naamgeving is dus belangrijk
- Dit heet *uitsluitendheid* (*sufficiency)*
  - *Doel hoog*

Doelen
Begrijpelijkheid
Uitbreidbaarheid
Aanpasbaarheid

Compleetheid
Uitsluitendheid
Criteria



Kwaliteitscriteria: Primitiefheid

- Een klasse biedt slechts een enkele manier om iets te doen
- Dit heet *primitiefheid* (*primitiveness*).
  - Doel: hoog

Doelen
Begrijpelijkheid
Uitbreidbaarheid
Aanpasbaarheid

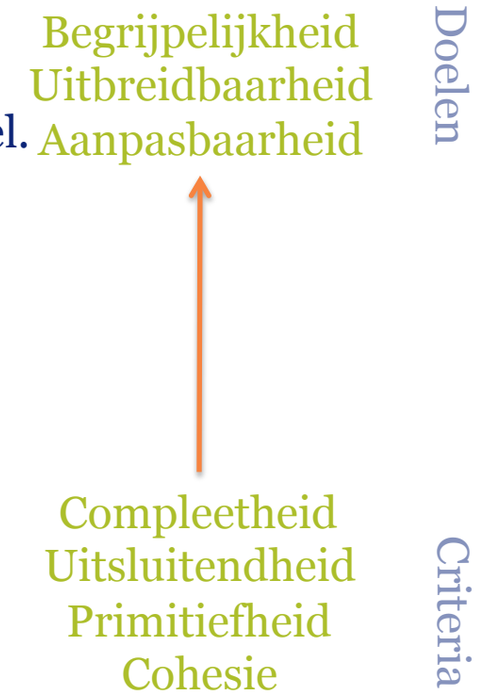
Criteria
Compleetheid
Uitsluitendheid
Primitiefheid

Bankrekening
– tegoed
+ opnemen + storten

X

Kwaliteitscriteria: Cohesie

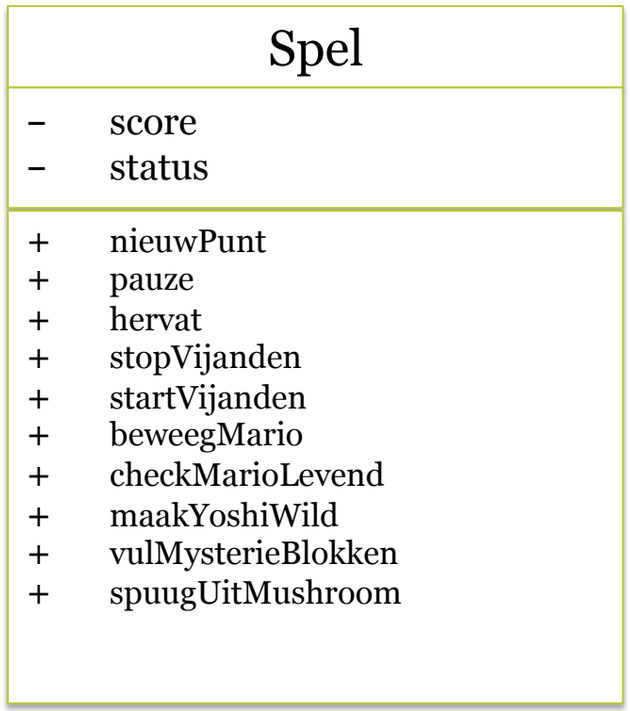
- Een Klasse modelleert slechts 1 logische taak en alle methoden ondersteunen dat doel.
  - Zo'n klasse is makkelijk te begrijpen en hergebruiken
- Dit heet *cohesie* (*cohesion*)
  - Doel: hoog
- Ook bij operaties kan je spreken van cohesie



Kwaliteitscriteria: cohesie

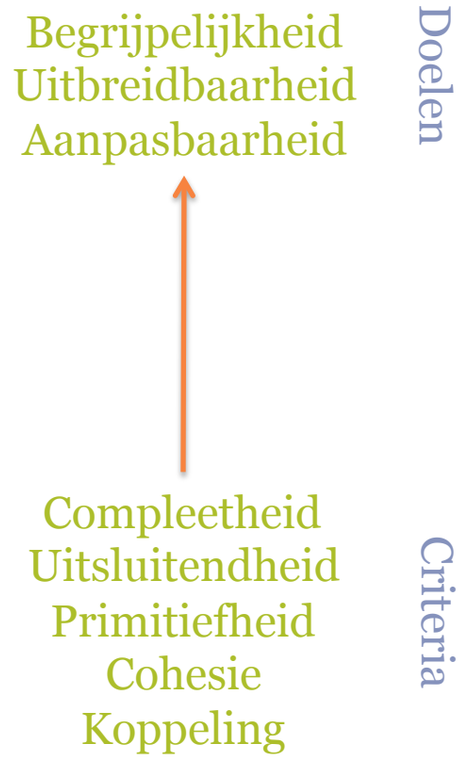
- Ultiem voorbeeld van lage cohesie:
  - Een klasse regelt feitelijk alles, alle andere klassen hebben weinig verantwoordelijkheid
  - '*god-class*'

- Indicatie van lage cohesie:
  - Grote classes
  - Het diagram is een 'ster'
  - Veel public methoden
    - Er moet toch gecommuniceerd worden…
  - Acties en namen van andere klassen in operatie-namen



Kwaliteitscriteria: Koppeling

- Een klasse heeft zo weinig mogelijk ‘kennis’ nodig van andere klassen
  - Dit is gebruik van methoden
- Als klassen veel van elkaar weten (en dus gebruiken) moeten bij aanpassing van 1 klasse veel andere ook worden aangepast.
- Dit heet *koppeling* (*coupling*)
  - Doel **laag**



Kwaliteitscriteria: Koppeling

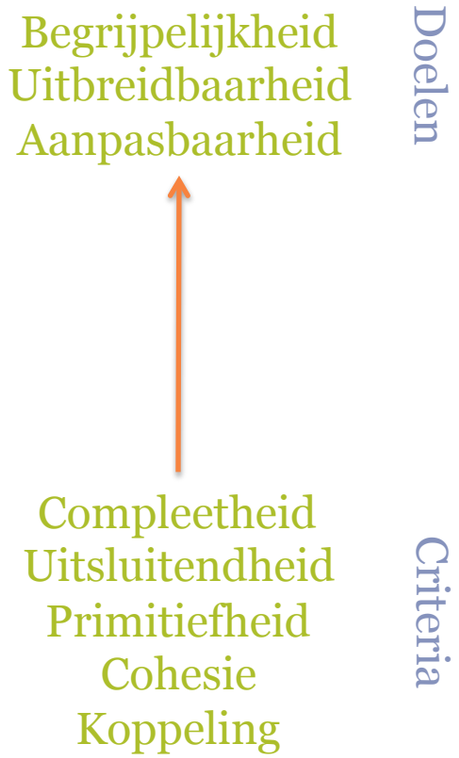
- Indicaties van koppeling:
  - Veel associaties

Kennelijk gebruiken de klassen elkaar

  - Operaties als:

```
get<AndereKlasse>()
```

Kennelijk geeft deze klasse kennis van een klasse door naar een andere klasse



Kwaliteitscriteria

- Bij een eenvoudig systeem zijn de kwaliteitscriteria relatief makkelijk te behalen.
- Bij complexere systemen wordt dit lastiger
  - Bij een grafische interface:

Hier gebeurt input (knoppen)

Hier gebeurt output (bv. PopUp)

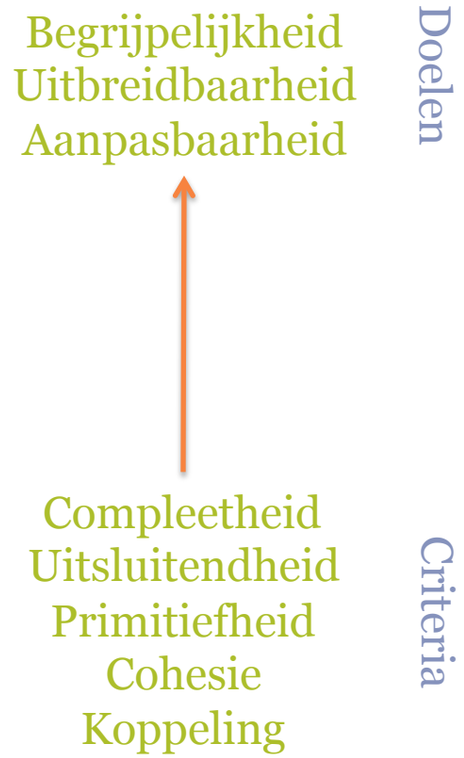
Koppeling naar andere klassen ligt op de loer

  - Bij een spel:

Een spel kan pauzeren

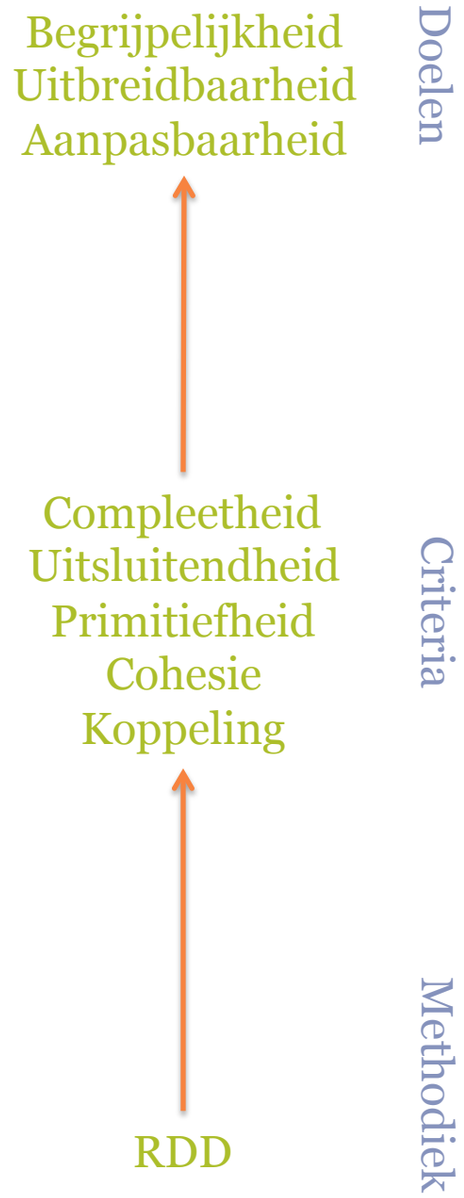
Een spel moet resultaten verwerken (gewonnen / verloren)

‘God-class’ lig op de loer (=lage cohesie)



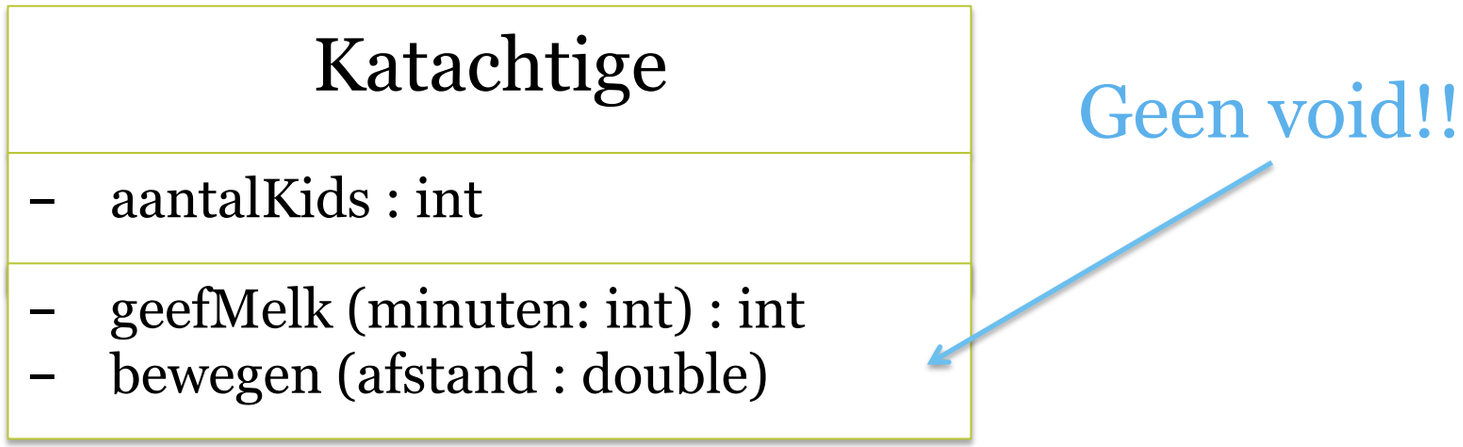
Kwaliteitsmethodiek

- Aan de criteria kan je het beste voldoen door je steeds af te vragen:
  - Heb ik de *verantwoordelijkheden* goed verdeeld?
  - Is dit inderdaad de *verantwoordelijkheid* van deze klasse?
- Dit heet *Responsibility Driven Design* (*RDD*).



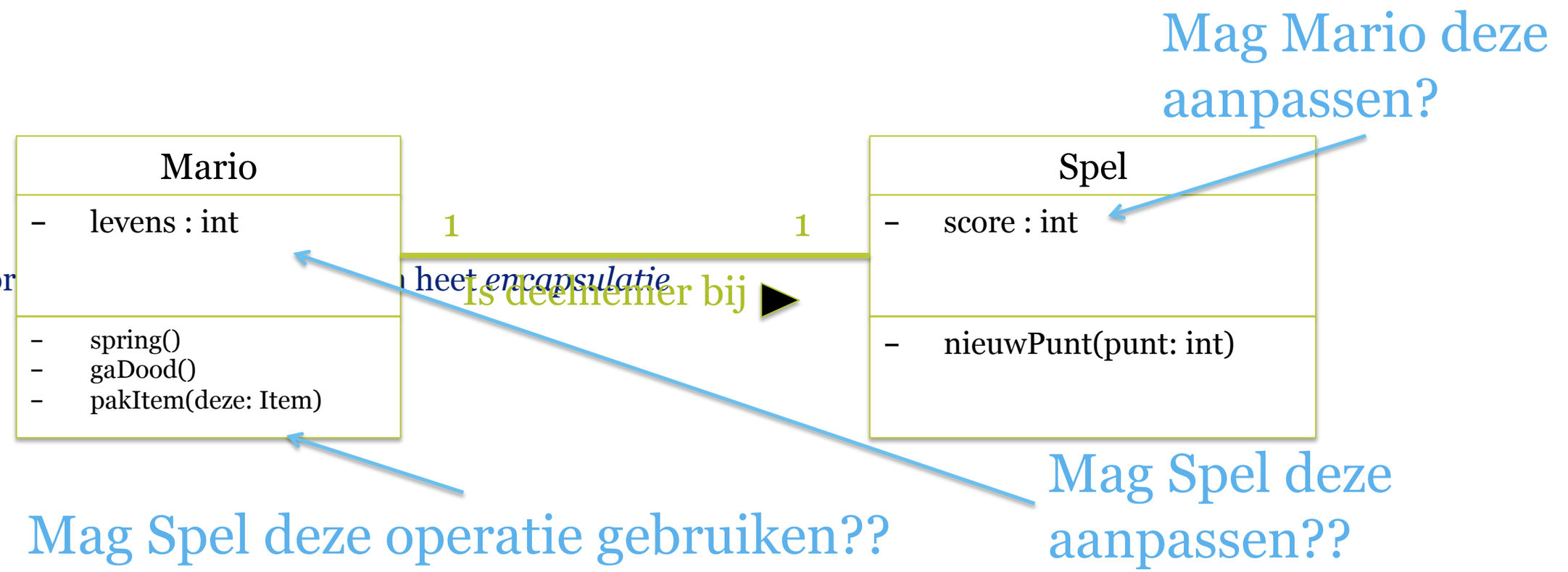
Types: UML syntax

- In de meeste programmeertalen hebben attributen, parameters (variabelen), en operaties een bepaald type.
  - Deze leg je bij het ontwerp vaak al vast…
  - … en zie je dus al in het klassendiagram
  - De syntax is (operatie/ variabele)naam : type



Encapsulatie

- Als klassen met elkaar een associatie hebben kunnen ze elkaars attributen aanpassen en elkaars operaties gebruiken.



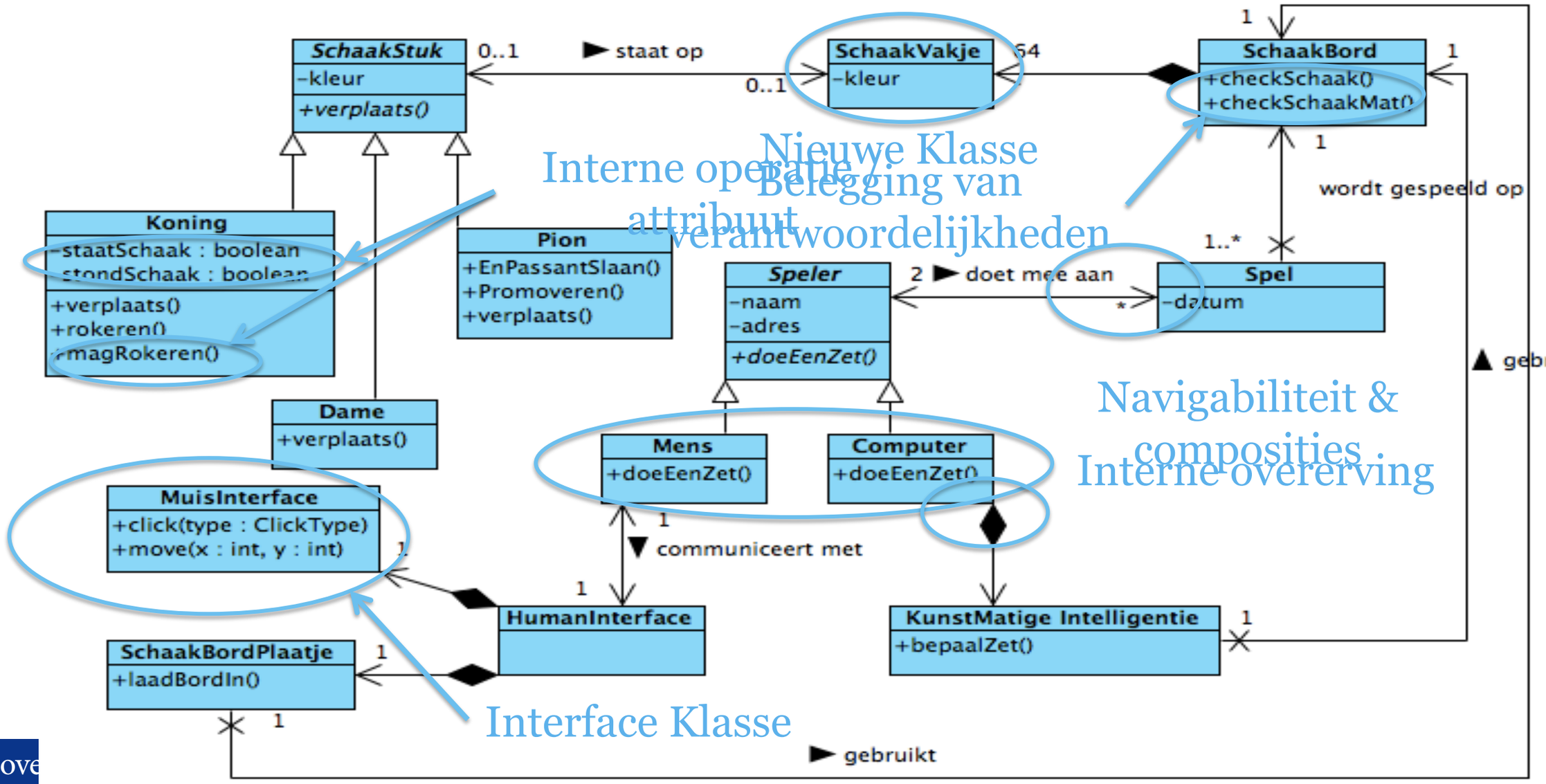
- Het pr heet *encapsulatie*

Encapsulatie: UML syntax

- Een attribuut of operatie is *public*:
  - Het element kan door elke geassocieerde klasse aangeroepen of aangepast worden
  - Syntax '+'
- Een attribuut of operatie is *private*:
  - Het element kan alleen door operaties van de eigen klasse aangeroepen of aangepast worden
  - Syntax '-'
- Een attribuut of operatie is *protected*:
  - Het element kan door operaties van de eigen klasse, de kind-klassen (en klassen in het package) aangeroepen of aangepast worden
  - Syntax: '#'
- Regel: zoveel mogelijk private
  - In ieder geval alle attributen
- Een klasse zonder 'public's' is onnuttig.

Encapsulatie: UML syntax

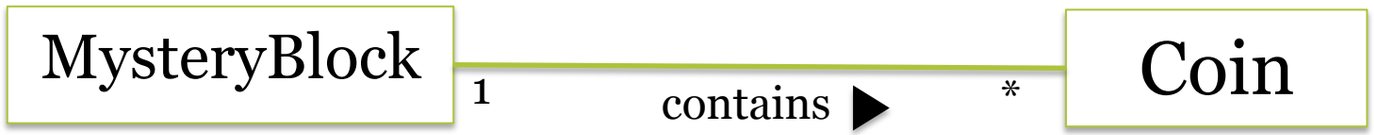
Analysis vs. Design Klassendiagram



Bewaren van modellen?

Strategie	Gevolgen
Analysis model verandert in Design model	1. Je hoeft maar 1 model te onderhouden 2. Je raakt de oorspronkelijk business communicatie kwijt
Analysis model verandert in Design model en gebruikt een tool om analyse model terug te krijgen	1. Je hoeft maar 1 model te onderhouden 2. Het analyse model is mogelijk van lage kwaliteit
Maak een kopie van het analyse model en verfijn dat naar een design model	1. Analysis model is mogelijk out-of-date 2. Beide diagrammen zijn aanwezig en van goede kwaliteit
Start en onderhoud 2 losse modellen	1. Je moet beide modellen onderhouden 2. Beide diagrammen zijn aanwezig en van goede kwaliteit

Constraints



- A MysteryBlock contains multiple Coins
- A Coin is contained by 1 MysteryBlock
- But…
  - Is the Coin interested in who contains it??
  - Is the MysteryBlock interested in what it contains?

Navigability: UML syntax

- An association has a direction, which we call *navigability*.
  - Which class has knowledge of the other class in an association.
- If you 'know' the other class you can call methods of the other class.
  - This is different from reading direction!
  - Denoted by: arrowheads and crosses.

The MysteryBlock has knowledge of the Coin
*The Coin has **no** knowledge of the MysteryBlock*

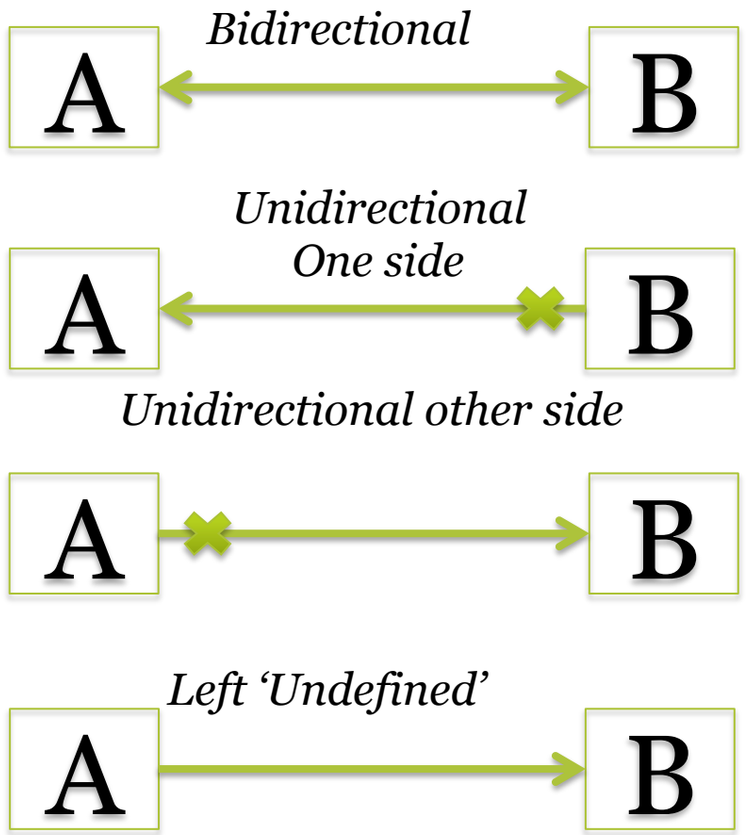
Navigability: UML syntax

- There are multiple options:
  - Bidirectional
  - Unidirectional
  - Undefined

But that one is not allowed

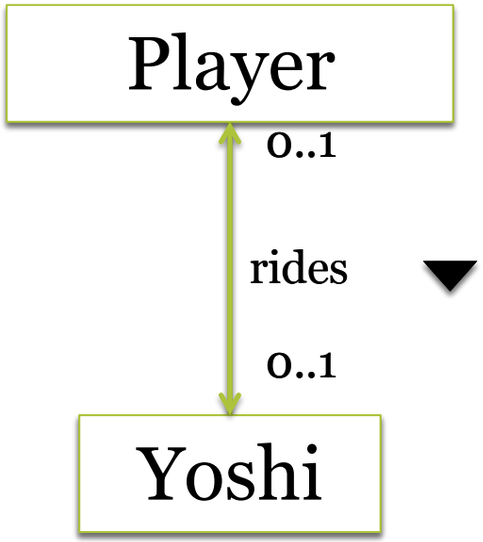
- Rules
  - At least one side is navigable (otherwise, why the association?)
  - As less navigability as possible

(lecture 4)



Case 1: Mario

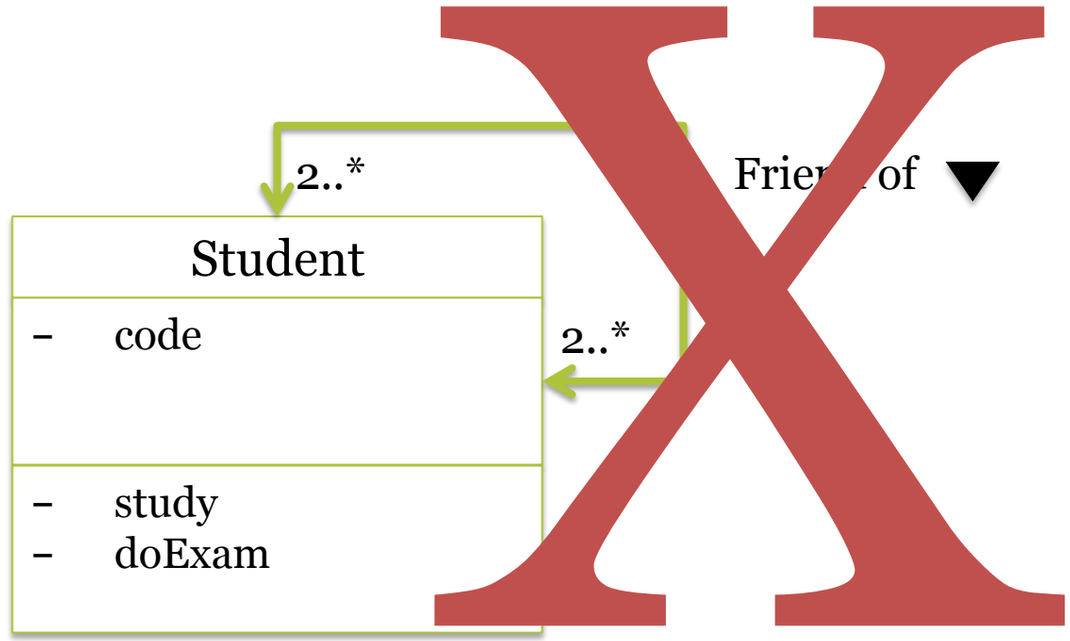
- Provide the navigability:



Navigability unary association

- What is the navigability of 'friend of'
  - You're always mutual friends

So bidirectional?

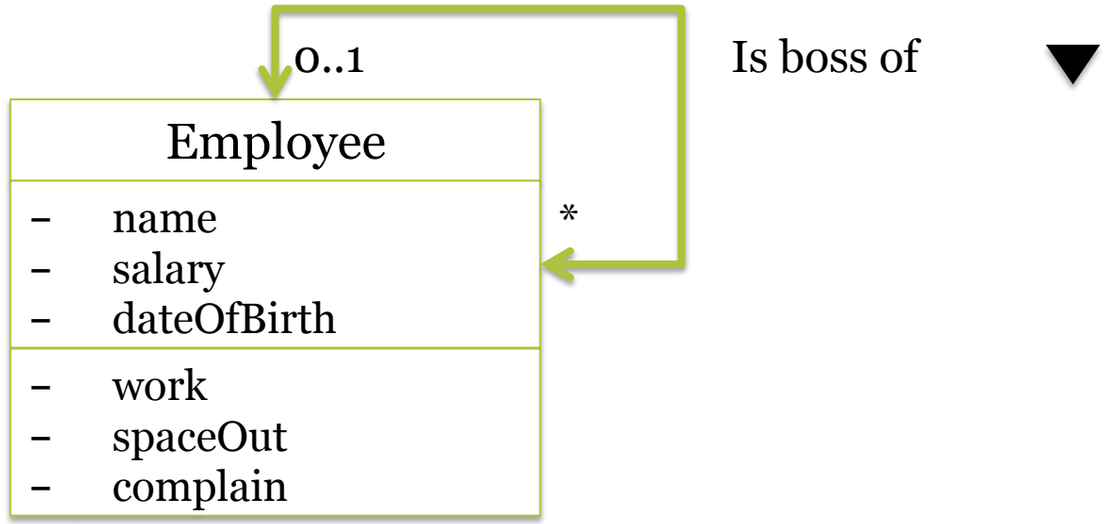


- Answer: NO, Unidirectional
  - Otherwise 'store 2 times'
    - 'I have you as friend'
    - 'you have me as friend'
- Rule: In unary associations based on *equality* (friends, neighbors, brother-sister, etc.) **always unidirectional**

Navigability unary association

- What is the navigability of 'is boss of'?
  - *Unequal* relationship

So unidirectional?

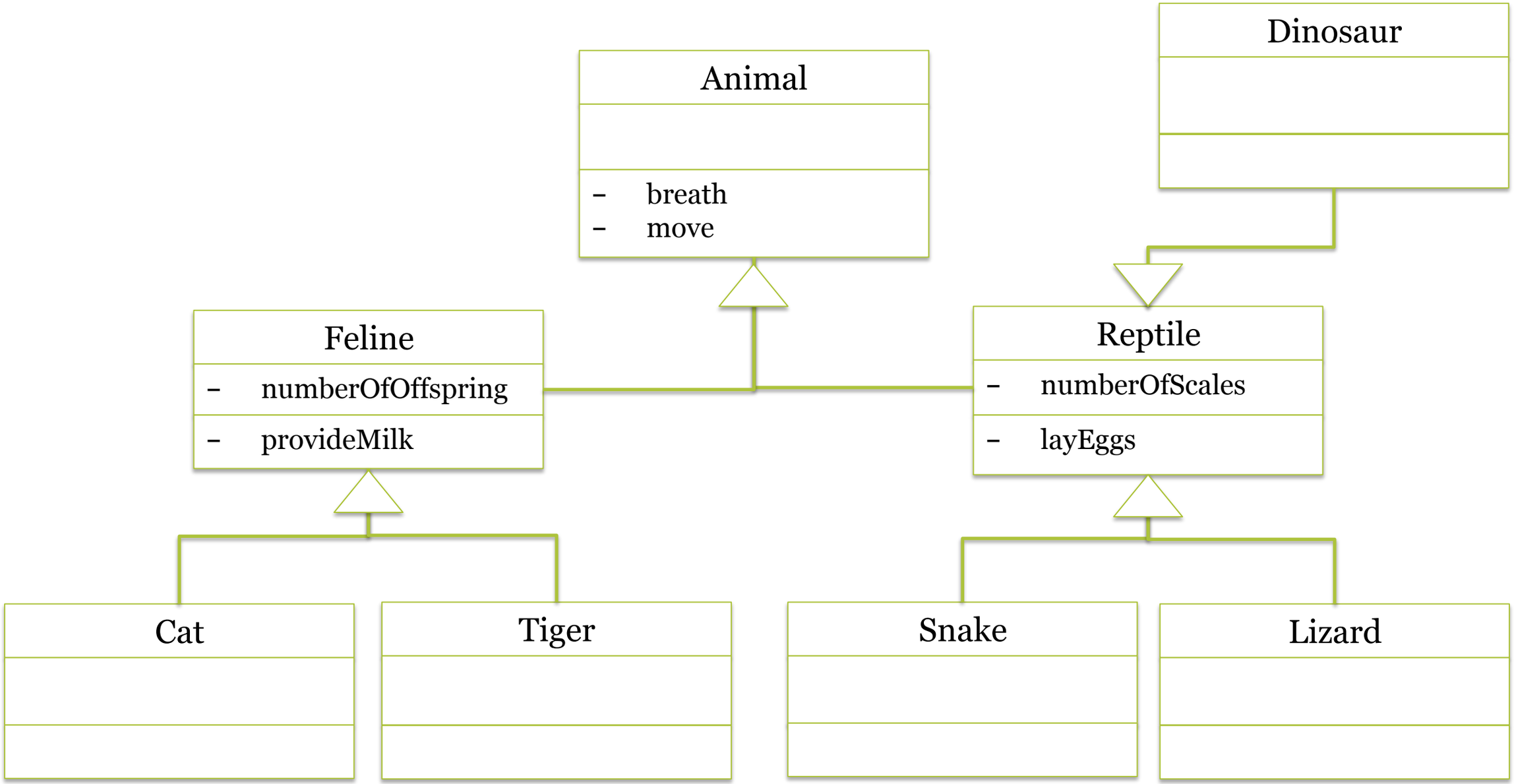


- Answer: NO, both can happen
  - If you know your boss and your boss knows its subordinates: bidirectional
  - If 1 of you doesn't know: unidirectional
  - Rulein an *unequal* relationship: both are possible

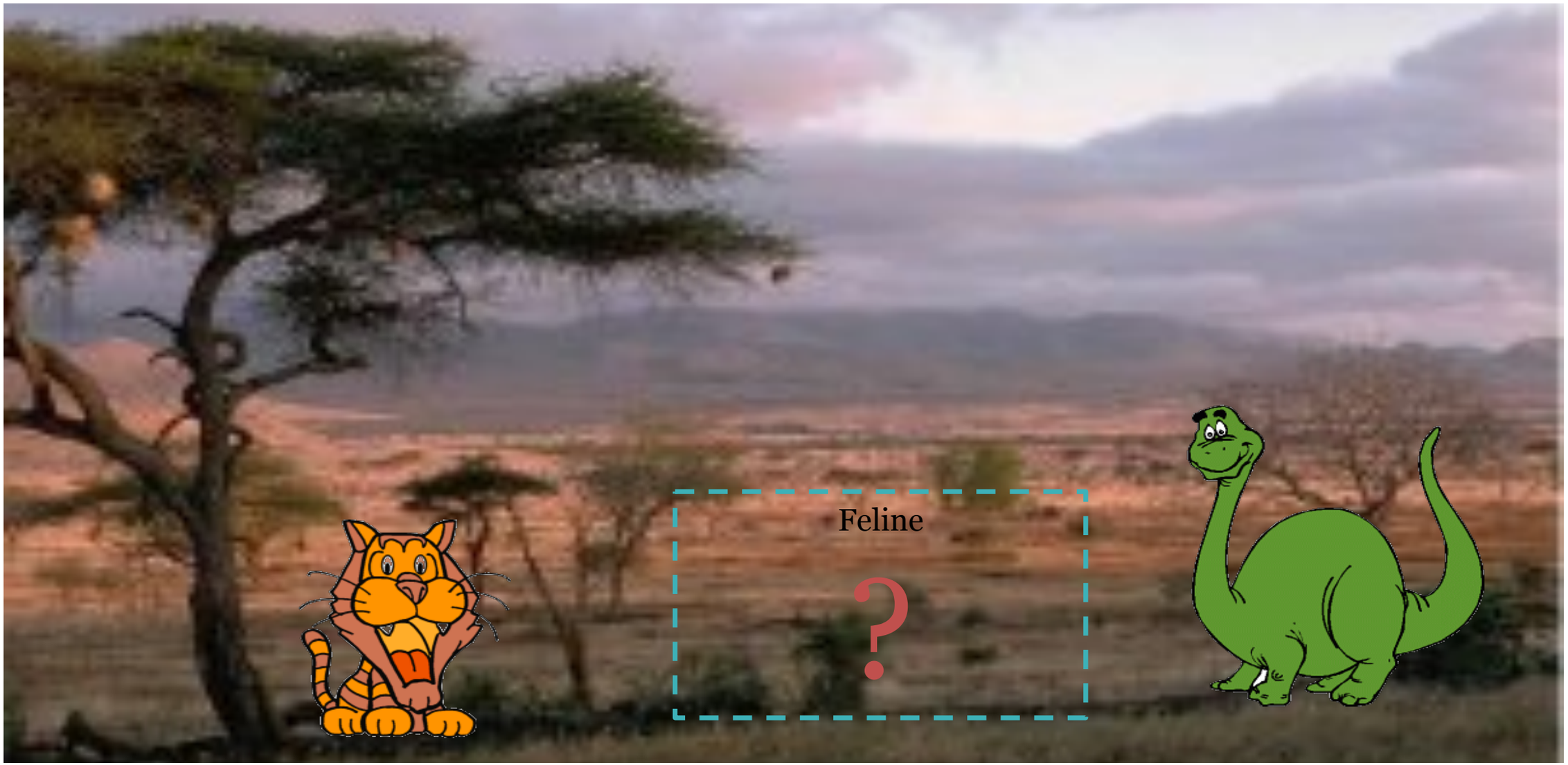
Case 5: Animal Kingdom

All animals are able to move and breath. Animals are either feline or reptiles. All reptiles have a number of scales, lay eggs, all felines have a number of offspring and provide milk. Reptiles are either a snake, a lizard or a dinosaur. The feline family consists of tigers and domesticated cats

Case 5: Animal Kingdom



Case 5: Animal Kingdom



Abstract Class: UML Syntax

- Some classes are not meant to make objects of
  - They are just a convenient tool to model similarity
- This is an *abstract* class
  - In UML: class name in *italics*
  - You cannot create an object of an abstract class

Feline
– numberOfOffspring
– provideMilk

Case 5: Animal Kingdom (extension)

Different animals move completely different. Reptiles usually move ‘close to the ground’, except snakes that crawl. Felines move ‘light on their feet’.

Implementation of methods

- We already saw an example where analysis text provides context for the possible contents of an attribute.
- This also happens for methods where a description is proided **how** the method should be implemented (think chess moves)
  - In super class: inheritance
  - In super class: abstract method
  - Also in child class: polymorphism

Abstract Method: UML Syntax

- "Different animals move completely different"
  - The superclass 'knows' all animals can move
  - The super class has no idea hoe the **implementation** is.

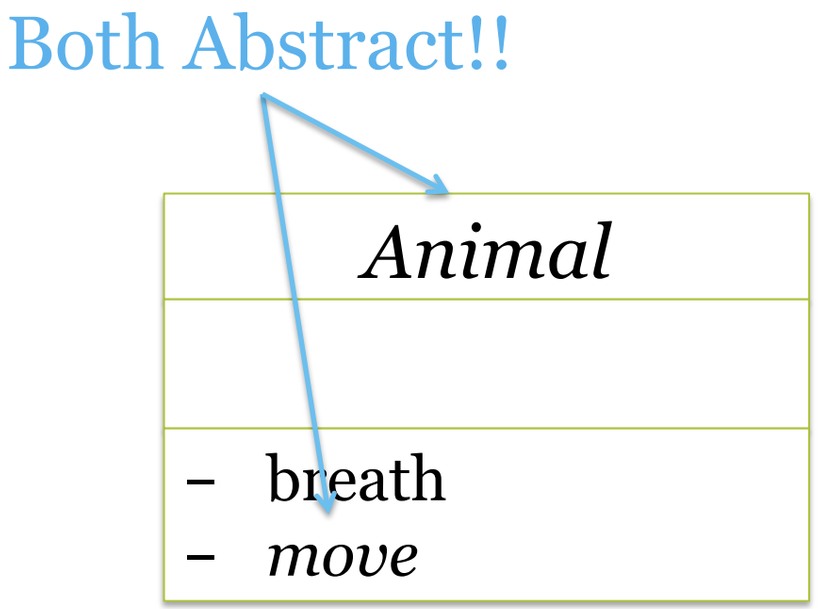
Since all sub classes have a 'completely different' implementation.

- This is an *Abstract Method*.
  - Every sub class should contain this method

Implented, ór

Pass it on to its own sub classes (again abstract)

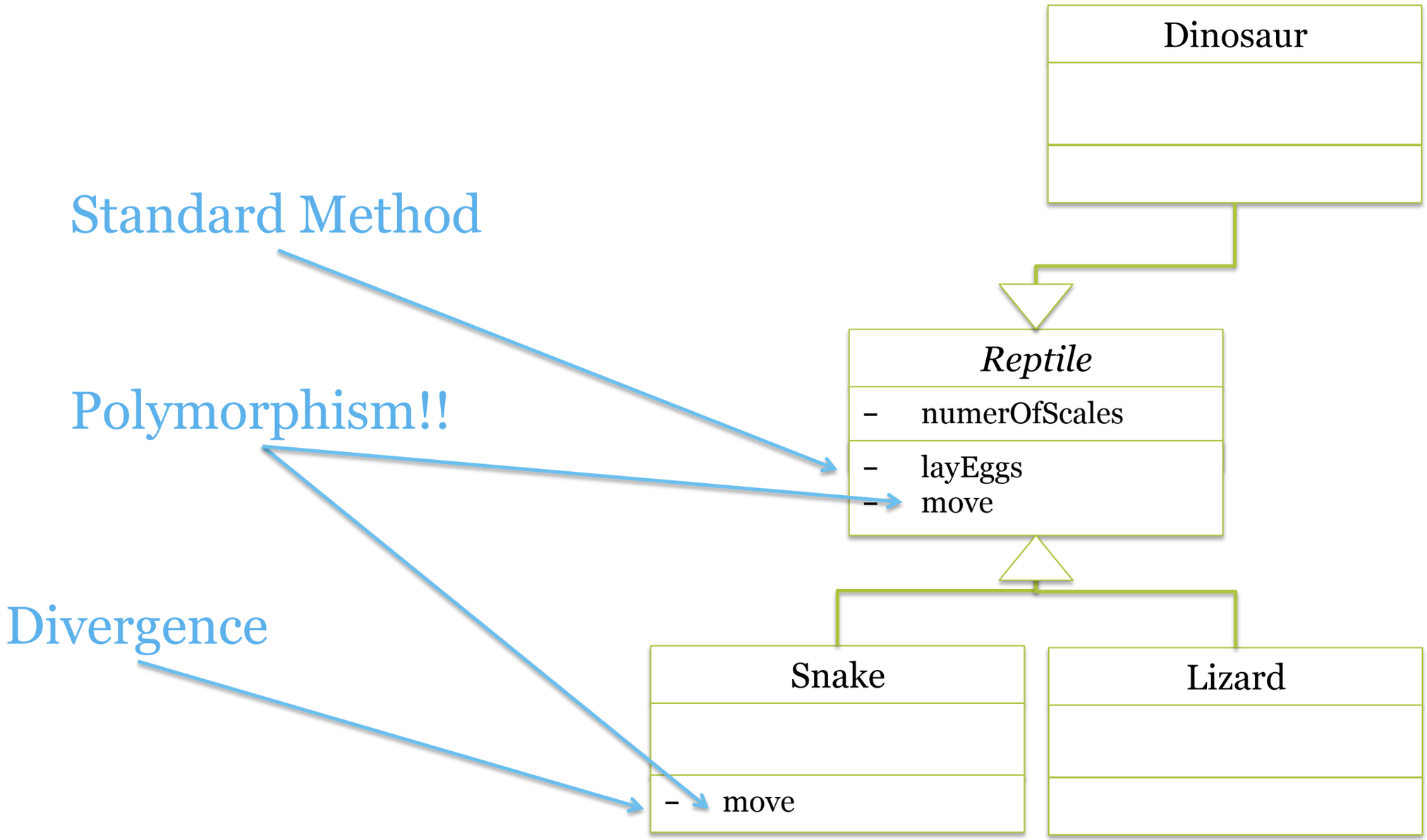
  - Can only exist in an abstract class
  - UML: method name in *italics*



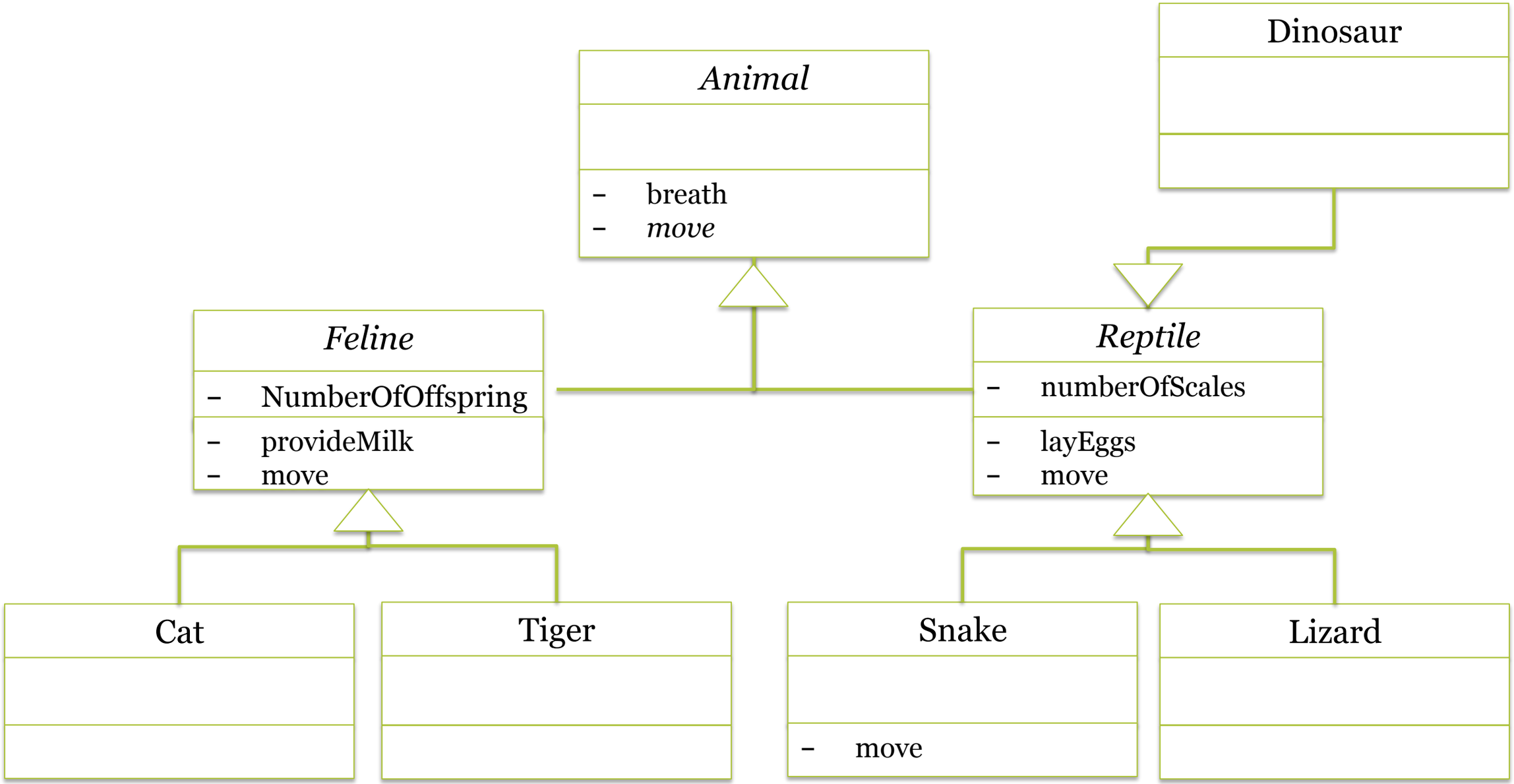
Polymorphism

- “Reptiles usually move ‘close to the ground’, except snakes that crawl.”
- If the **majority** of sub classes have the same implementation of a certain method, you place that implementation in the super class.
  - Divergent sub classes overwrite that method with an own variant
  - This is called *polymorphism*
  - UML: just have a method with the same name in the sub class

Polymorphism



Case 5: Animal Kingdom



Abstraction

- Sometimes you want to
  - Offer a simple interface to other ‘components’

With less methods

This is good for understandability

  - Handout a task, but don’t care what class performs that task
- Possible solution: abstract class

Abstraction

- But! Sometimes you want
  - Explicitly forbid other ‘components’ to see some functionality
  - To have different implementations
- And…
  - Sometimes there is no ‘natural’ inheritance.
- Solution: *interface*

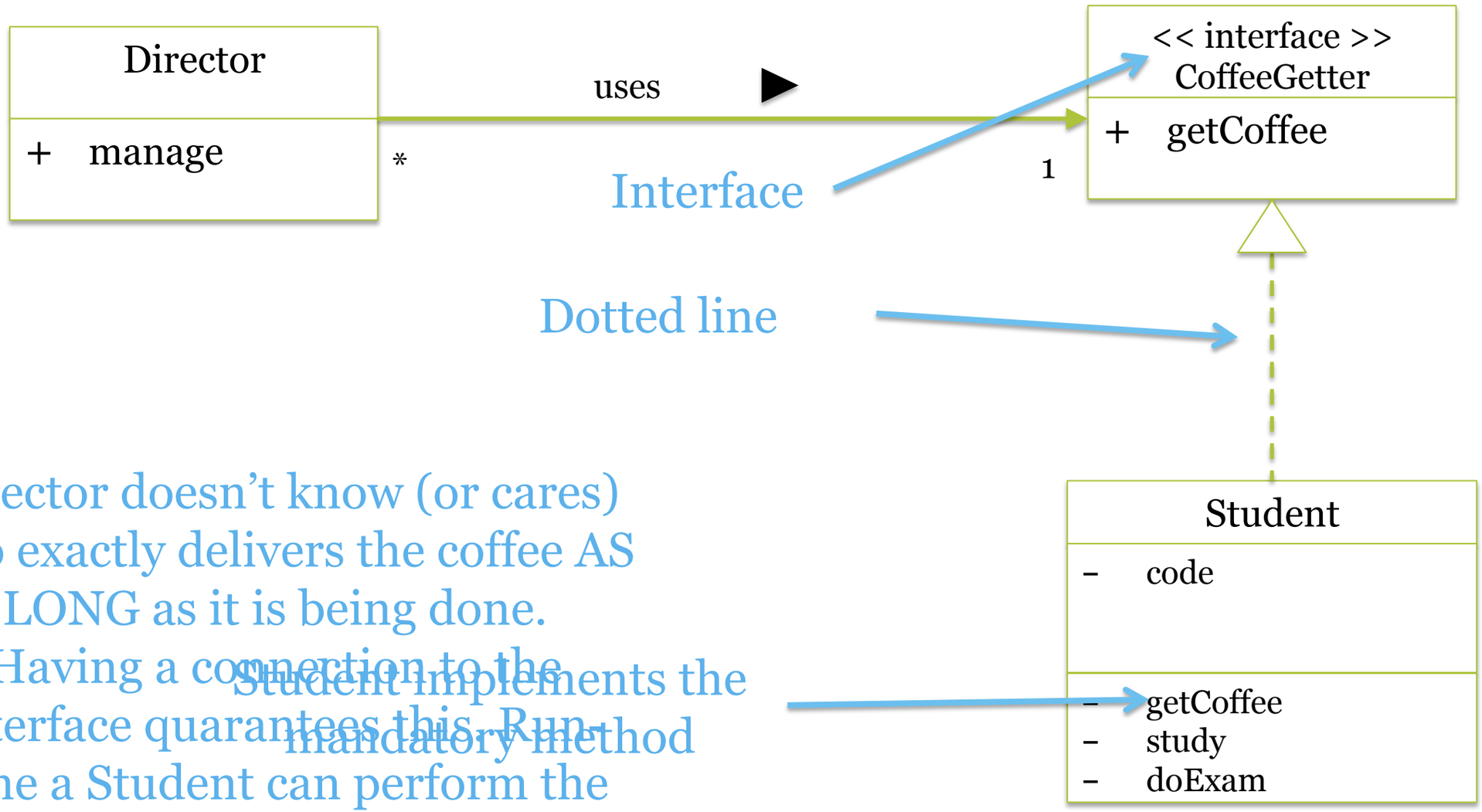
Interface

An *interface* is collection of methods ‘to be implemented’ by other classes

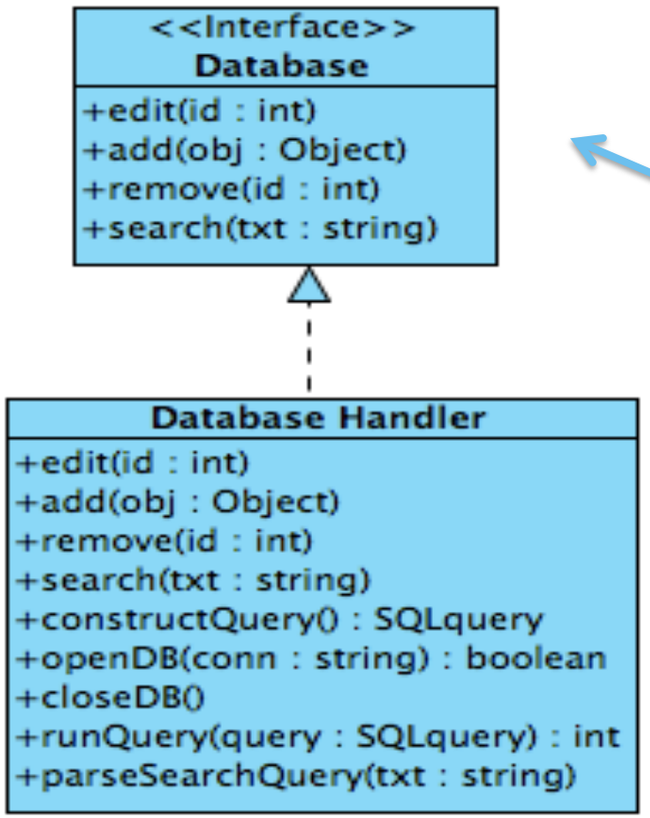
- These classes can be sub classes of another class
- An interface has no method implementation
- An interface has no attributes
- One cannot make an object of an interface
- A class that *implements* an interface must provide implementation for each method in it (or pass it on to sub classes)

- An interface offers functionality to other classes without specifying which classes will fulfill the role
  - An interface can be seen as a diploma; you don’t care who exactly does the job, as long as he / she is qualified.

Interface: UML Syntax

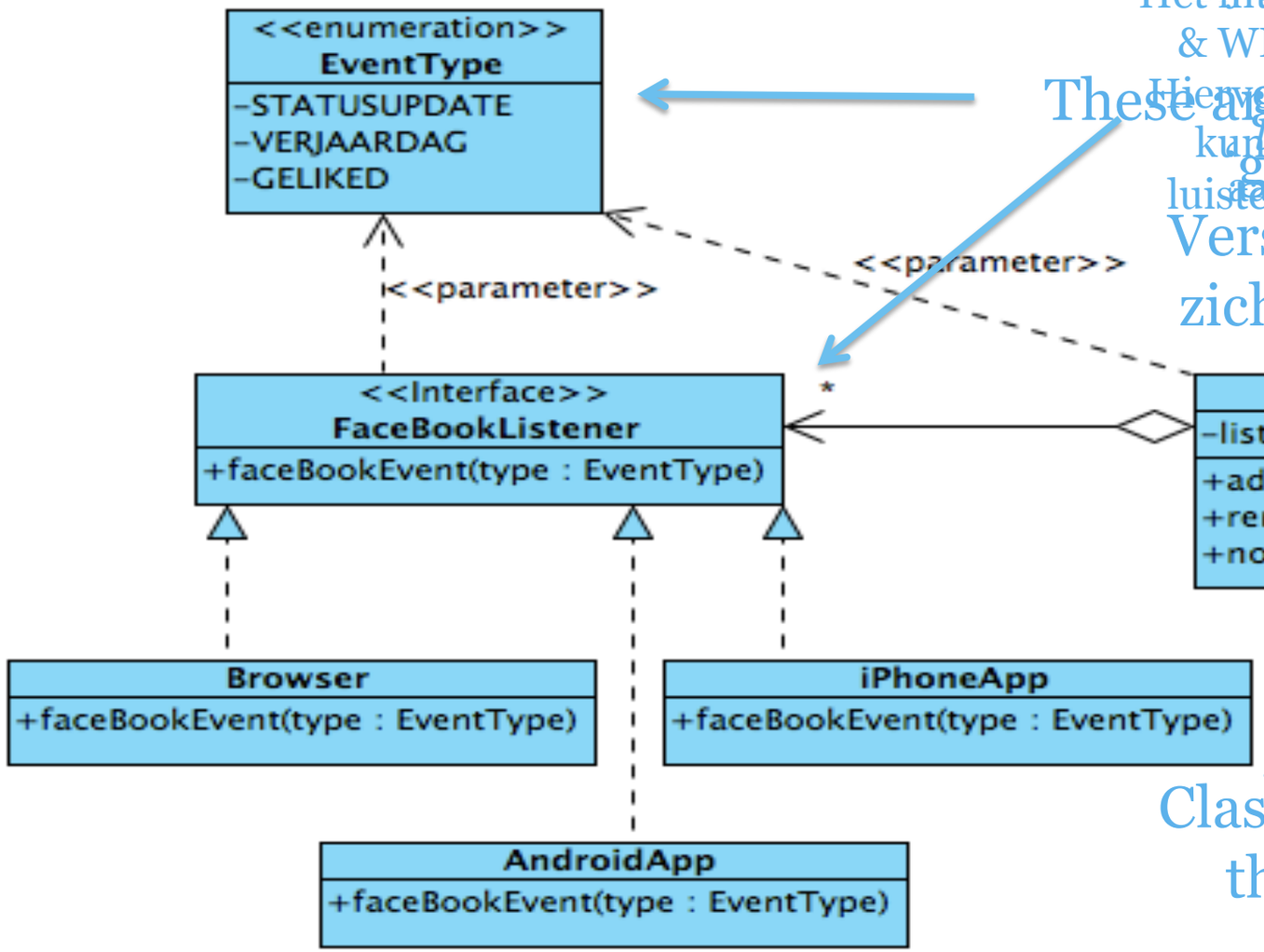


Example application for interface



Users of our database library can only access this functionality. The internal mechanics are (deliberately) unknown to them

Example application for interface: Observer pattern



Een Facebook account

These are all possible events

gebeurtenissen (events)

Verschillende klassen kunnen zich aanmelden om hierop te reageren

NB: in reality this system doesn't exist; its component would run on different devices.

Classes can enlist or remove themselves as 'listener'

It is just an illustration

Next up...

- Continue with assignment 3