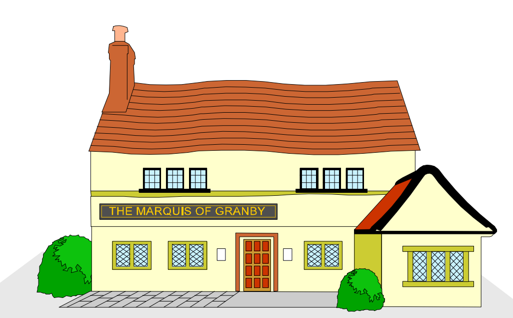
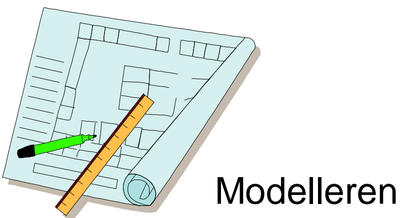
Modelleren



Werkelijkheid



Modelleren

TH Rijswijk

© 2007 Harry Broeders

Modeleren

Waarvan maken we een model?

- Analyse
  - Maak een model van de te automatiseren werkelijkheid of van het op te lossen probleem
    - Domeinkennis = structuur
    - Functionele eisen = gedrag
  - Wat?
- Ontwerp
  - Maak een model van het programma of van de oplossing
  - Hoe?
- Implementatie
  - Codeer het programma
  - Waarmee?

Bij OOA+OOD+OOP werken we steeds aan hetzelfde model (seamless development)

TH Rijswijk

UML

overzicht les 14 t/m 18

14 **UML inleiding**
H1 en H2

15 **UML Klasse- en objectdiagrammen (1)**
H4.1 t/m 4.3

16 **UML Klasse- en objectdiagrammen (2) en Use-case-diagram**
H4.4.1, 4.4.2, 4.4.5 t/m 4.4.9, 4.4.17 en H8.1 t/m 8.6

17 **UML Sequence- en collaboratiediagrammen**
H10.1 t/m 10.4
UML Toestands- en Activiteitsdiagrammen
H12.1 t/m 12.3, H15.1 t/m 15.3

18 **UML Klassediagrammen (3)**
H4.4.4, 4.4.10 en 4.4.12

TH Rijswijk

Together

ControlCenter 6.2

Borland
Excellence Endures™

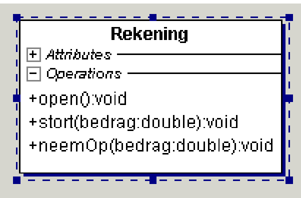
DESIGN DEVELOP

Borland® Together®
ControlCenter®

- tekenen van UML diagrammen
- omzetten UML naar C++ (of Java, C# of Visual Basic)
- omzetten C++ naar UML
- beschikbaar in lokaal 413 en 417

TH Rijswijk

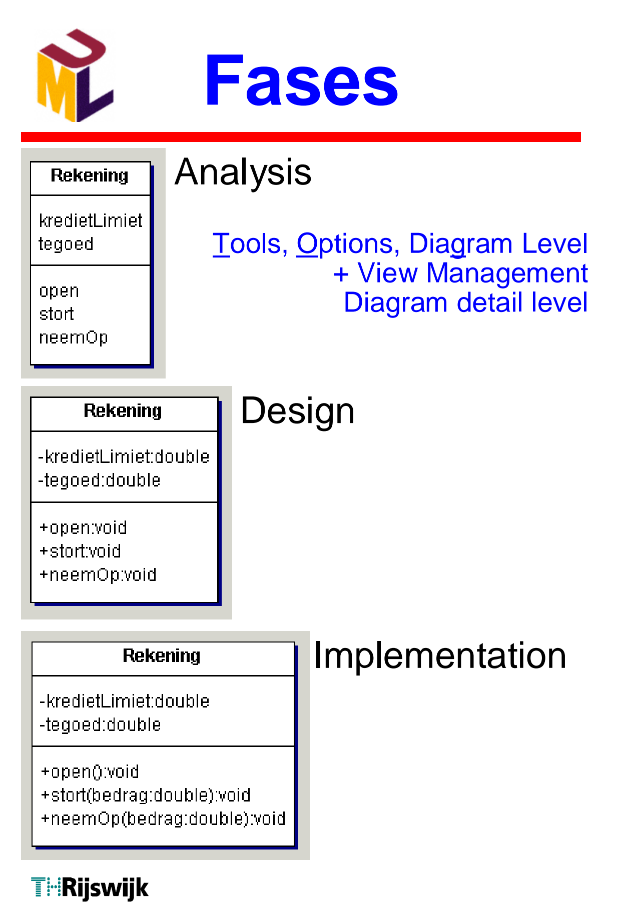
Klasse-diagram



- Attributes
  - =datamembers
- Operations
  - =messages
  - =memberfunctions

In Together kunnen attributes en operations “verborgen” worden.
Objectdiagrammen worden in Together **niet** ondersteund.

© 2007 Harry Broeders



Code generatie

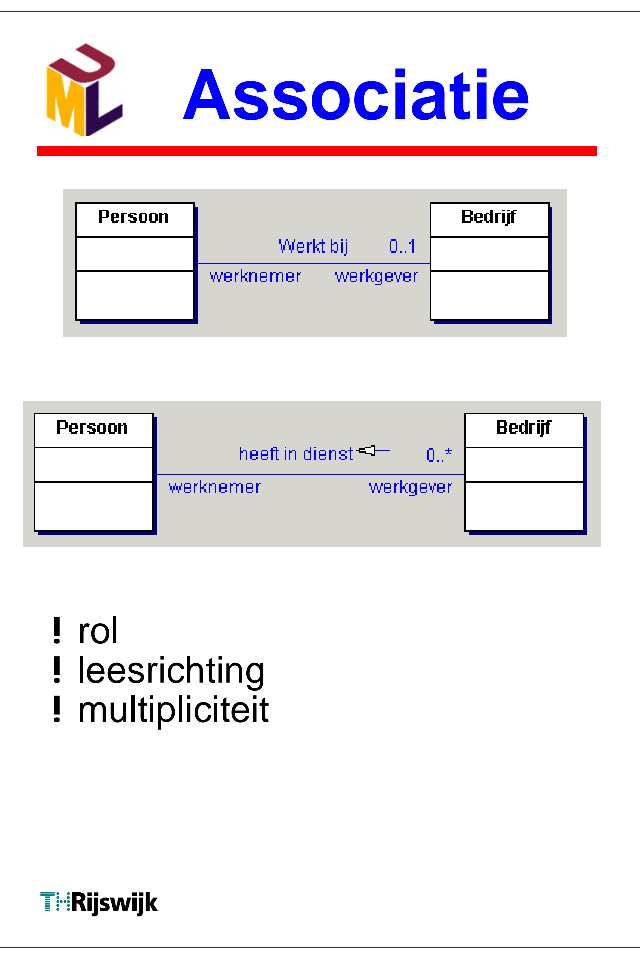
Rekening.h

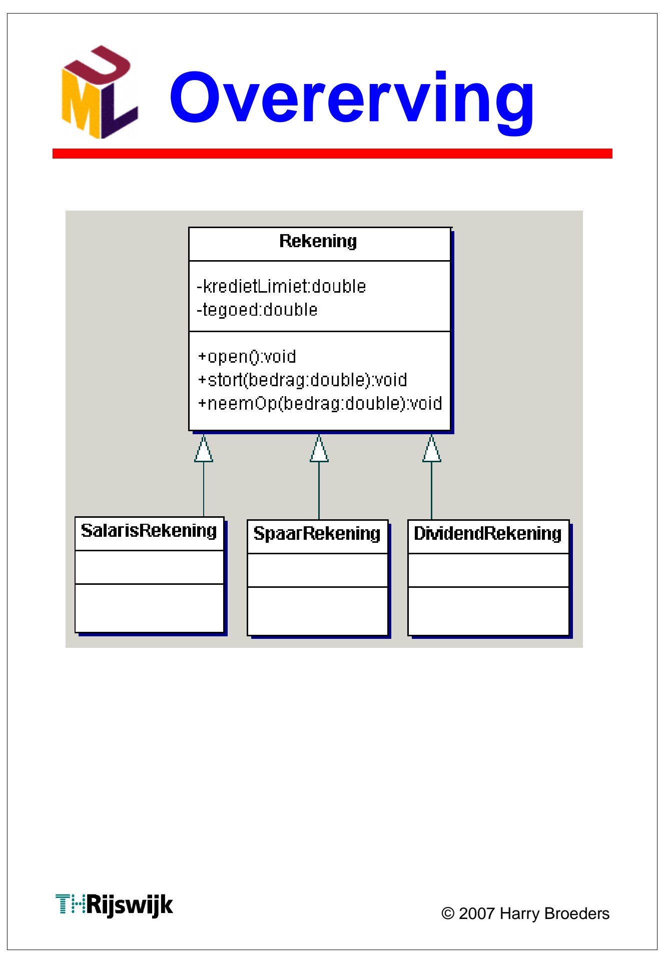
```
/* Generated by Together */
#ifndef REKENING_H
#define REKENING_H
class Rekening {
public:
    void open();
    void stort(double bedrag);
    void neemOp(double bedrag);
private:
    double kredietLimiet;
    double tegoed;
};
#endif //REKENING_H
```

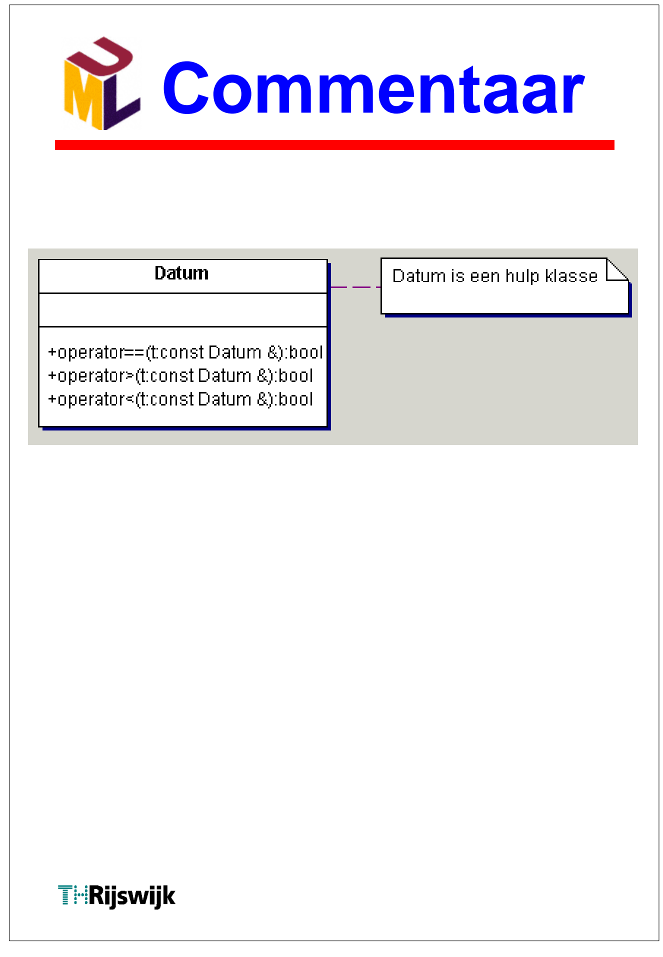
Rekening.cpp

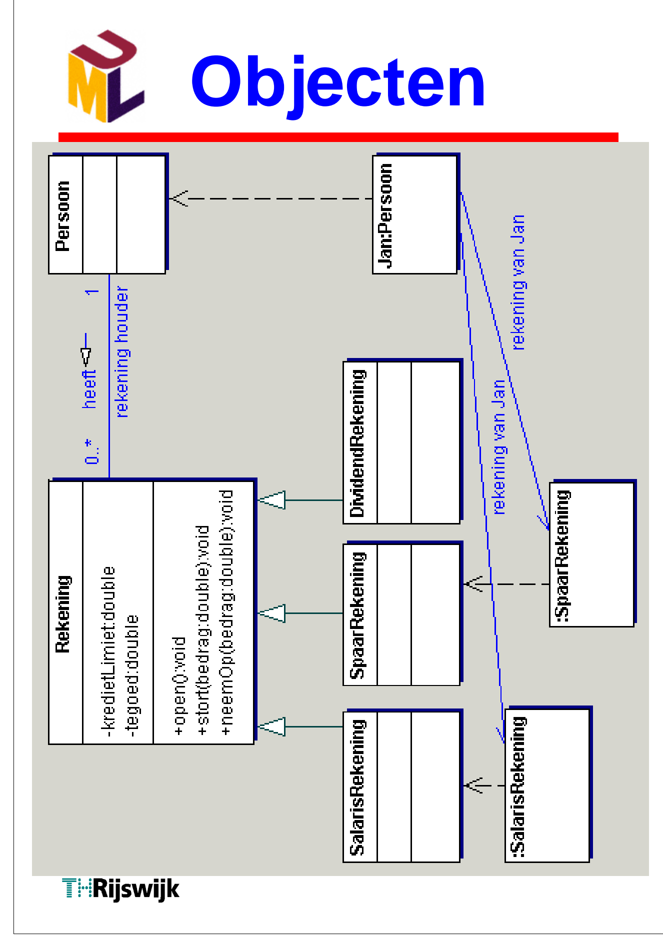
```
/* Generated by Together */
#include "Rekening.h"

void Rekening::open(){ }
void Rekening::stort(double bedrag){ }
void Rekening::neemOp(double bedrag){ }
```

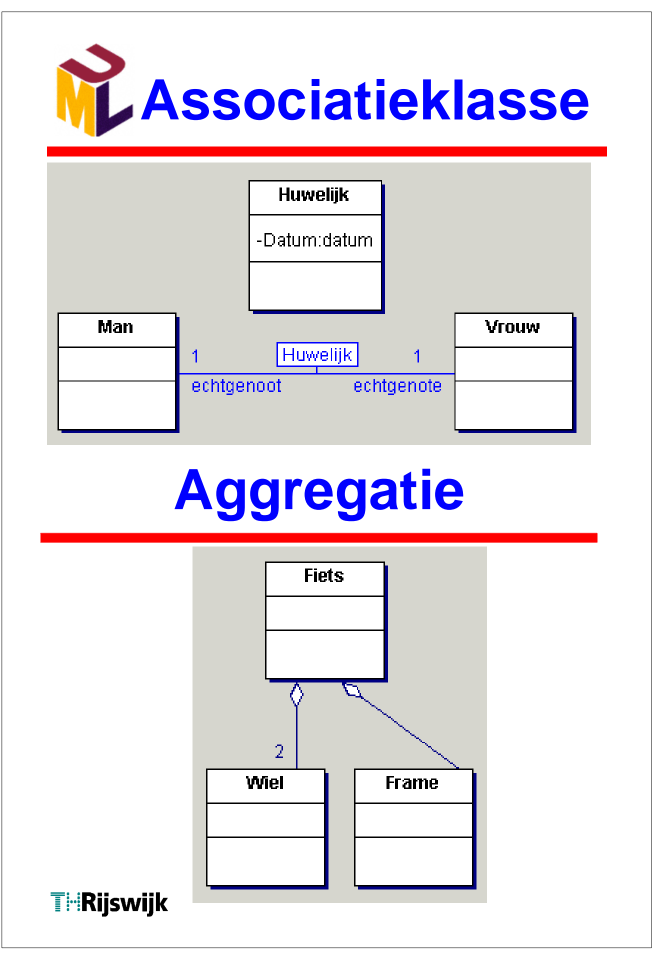

© 2007 Harry Broeders




Overerving
Rekening
-kredietLimiet:double
-tegoed:double
+open():void
+stort(bedrag:double):void
+neemOp(bedrag:double):void
SalarisRekening
SpaarRekening
DividendRekening
TH Rijswijk
© 2007 Harry Broeders

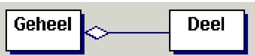

Commentaar
Datum
+operator==(t:const Datum &):bool
+operator>(t:const Datum &):bool
+operator<(t:const Datum &):bool
Datum is een hulp klasse
TH Rijswijk


Objecten
Persoon
Jan:Persoon
rekening van Jan
1
heeft
rekening houder
0..*
DividendRekening
rekening van Jan
:SpaarRekening
Rekening
-kredietLimiet:double
-tegoed:double
+open():void
+stort(bedrag:double):void
+neemOp(bedrag:double):void
SpaarRekening
SalarisRekening
:SalarisRekening
TH Rijswijk

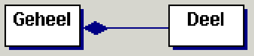

Associatieklasse
Huwelijk
-Datum:datum
Man
Vrouw
1
Huwelijk
1
echtgenoot
echtgenote
Aggregatie
Fiets
2
Wiel
Frame
TH Rijswijk

Associaties

speciale soorten



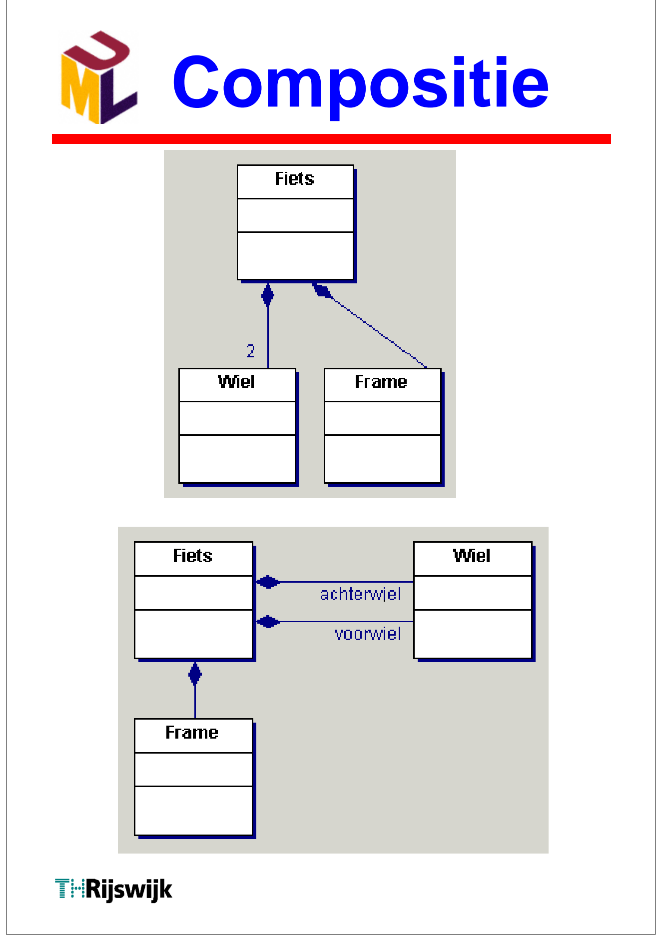
! Aggregratie
  " “heeft een” relatie.
  " vaag gedefinieerd



! Compositie
  " “bevat een” relatie
  " onderdeel behoort (op 1 bepaald moment) maar bij 1 geheel
  " levensduur deel <= levensduur geheel (lifetime dependency)

© 2007 Harry Broeders

Compositie



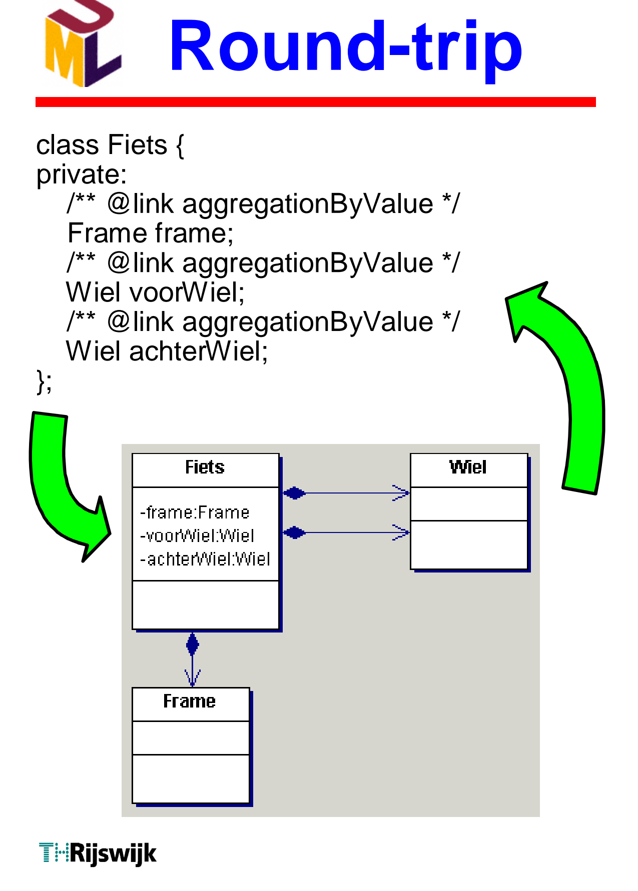
Code generatie

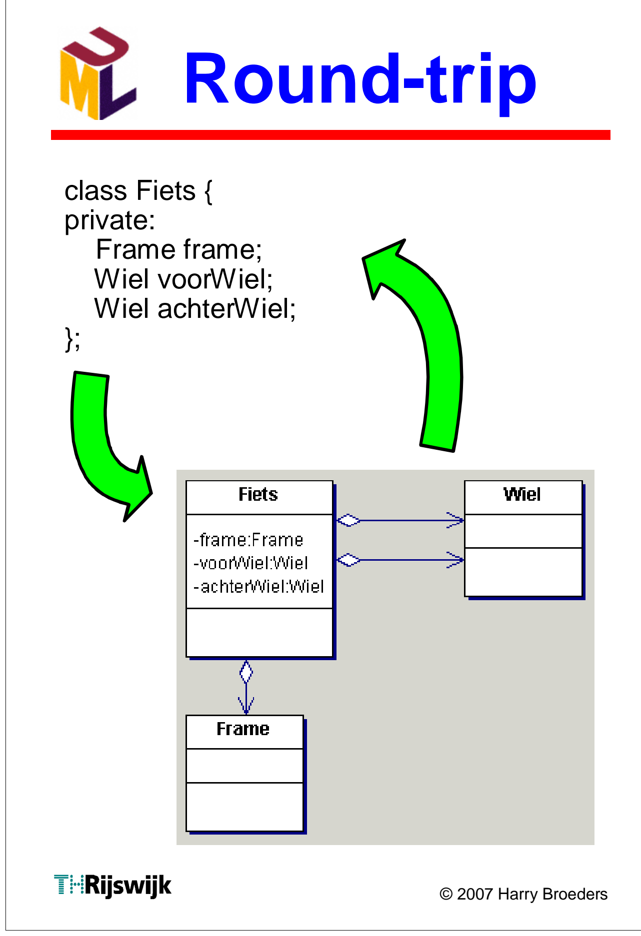
```
/* Generated by Together */

#ifndef FIETS_H
#define FIETS_H
#include "Wiel.h"
#include "Frame.h"

class Fiets {
private:
    /** @link aggregationByValue */
    Frame lnkFrame;
    /** @link aggregationByValue
     * @supplierRole voorwiel*/
    Wiel lnkWiel;
    /** @link aggregationByValue
     * @supplierRole achterwiel*/
    Wiel lnkWiel1;
};
#endif //FIETS_H
```


© 2007 Harry Broeders

Round-trip

```
class Fiets {
private:
    /** @link aggregationByValue */
    Frame frame;
    /** @link aggregationByValue */
    Wiel voorWiel;
    /** @link aggregationByValue */
    Wiel achterWiel;
};
```





Aggregratie

Kennel — 0..* — *Hond*

Tekkel, SintBernard

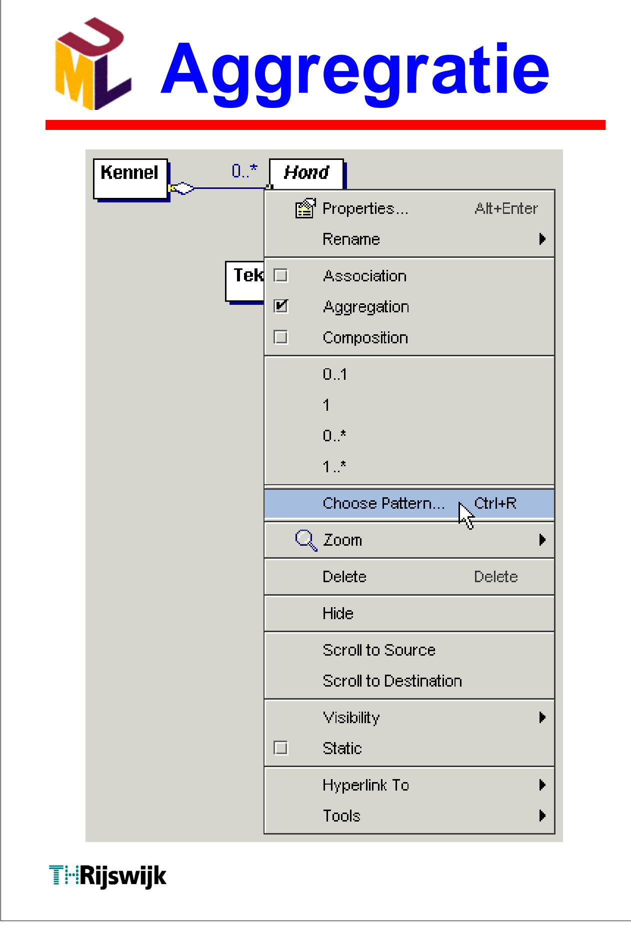
```
class Kennel {
private:
    /** @link aggregation
     * @supplierCardinality 0..**/
    Hond lnkHond;
};
```

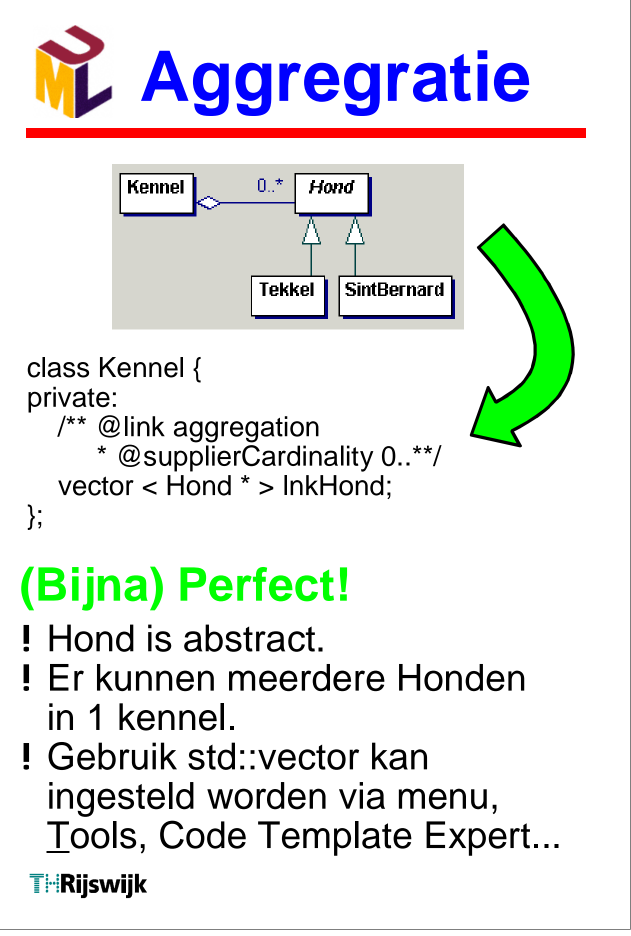
Slaat nergens op!

! Hond is abstract!

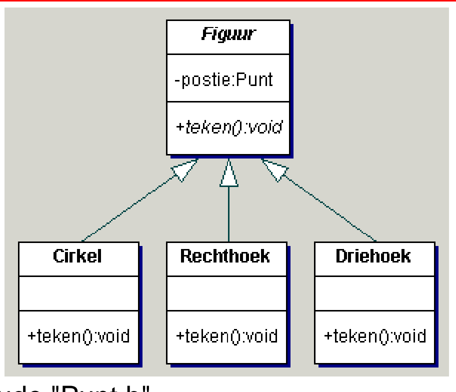
! Er moeten meerdere Honden in 1 kennel kunnen.

THRijswijk





Abstractie



Figuur.h

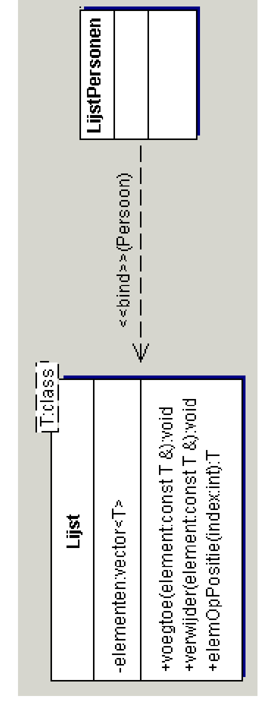
```
#include "Punt.h"
class Figuur {
public:
    virtual void teken()=0;
private:
    Punt postie;
};
```

Cirkel.h

```
#include "Figuur.h"
class Cirkel : public Figuur {
public:
    void teken();
};
```

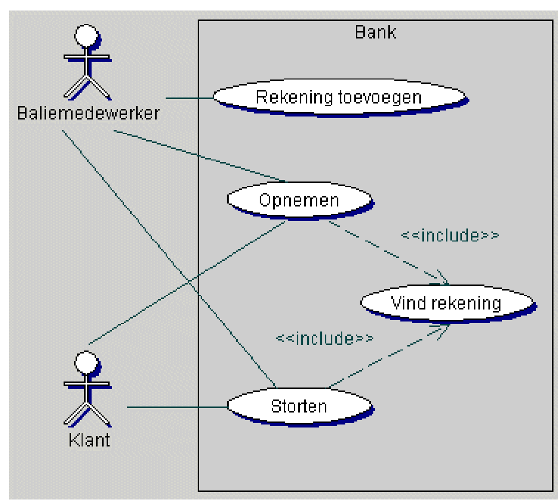

© 2007 Harry Broeders

Template




```
template < class T >
class Lijst {
public:
    void voegtoe(const T& element);
    void verwijder(const T& element);
    T elemOpPositie(int index);
private:
    vector<T> elementen;
};
```


Use-case-diagram



Senario

Use-case beschrijving

UseCase
Rekening toevoegen

preconditions:

BalieMedewerker heeft beschiking over NAW-gegevens van de klant. Klant kan zich legitimeren.

postconditions:

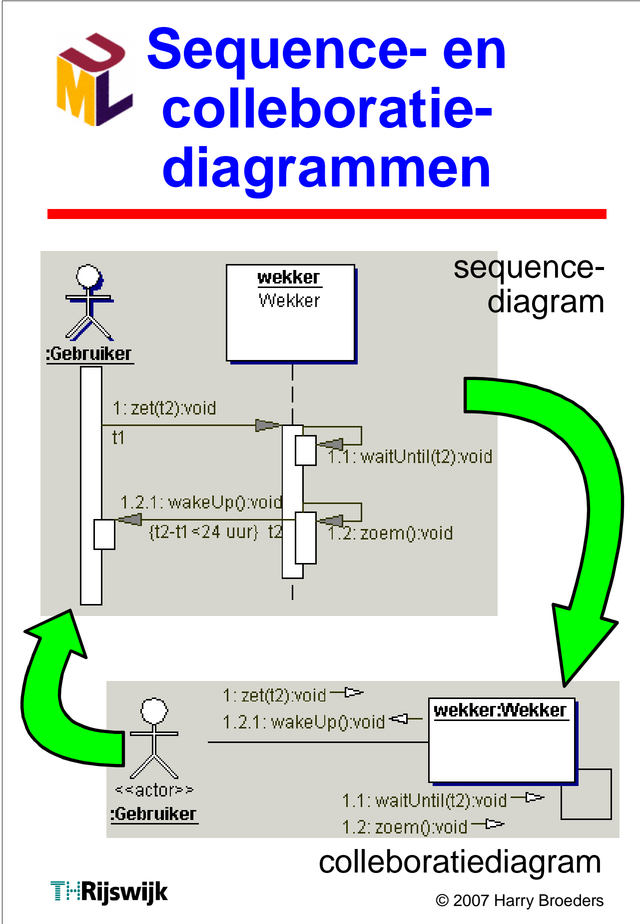
Klant heeft minstens 1 rekening

normalFlow:

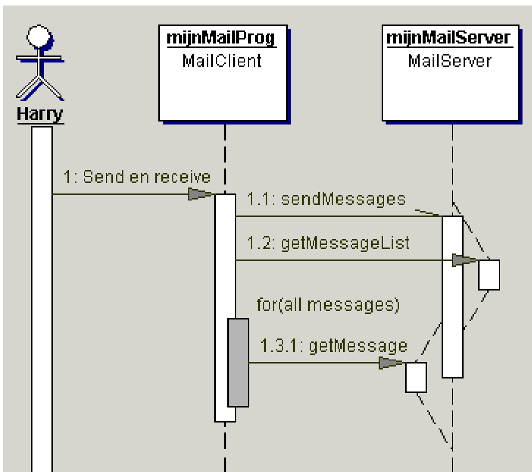
(1) De baliemedewerker maakt aan het systeem bekend dat een nieuwe rekening aangemaakt moet worden en geeft de NAW-gegevens van de klant. Als de klant een bedrijf is dan wordt ook het Kamer van Koophandelnummer ingevuld. (2) Het systeem checked of de klant al bekend is. Is dit het geval dan wordt de klantrekeningen gecontroleerd op rood staan. Als de klant rood staat dan treedt een uitzondering op. (3) Het systeem maakt het nieuwe rekeningnummer aan de baliemedewerker bekend.

alternateFlow:

[Rood staan] Als een rekening van de klant rood staat dan wordt hiervan door het systeem melding gegeven. De baliemedewerker kan dan naar de use-case Storten overgaan om de klant de gelegenheid te geven het tekort aan te vullen. Als het saldo is aangevuld, dan wordt de rest van de use-case uitgevoerd.

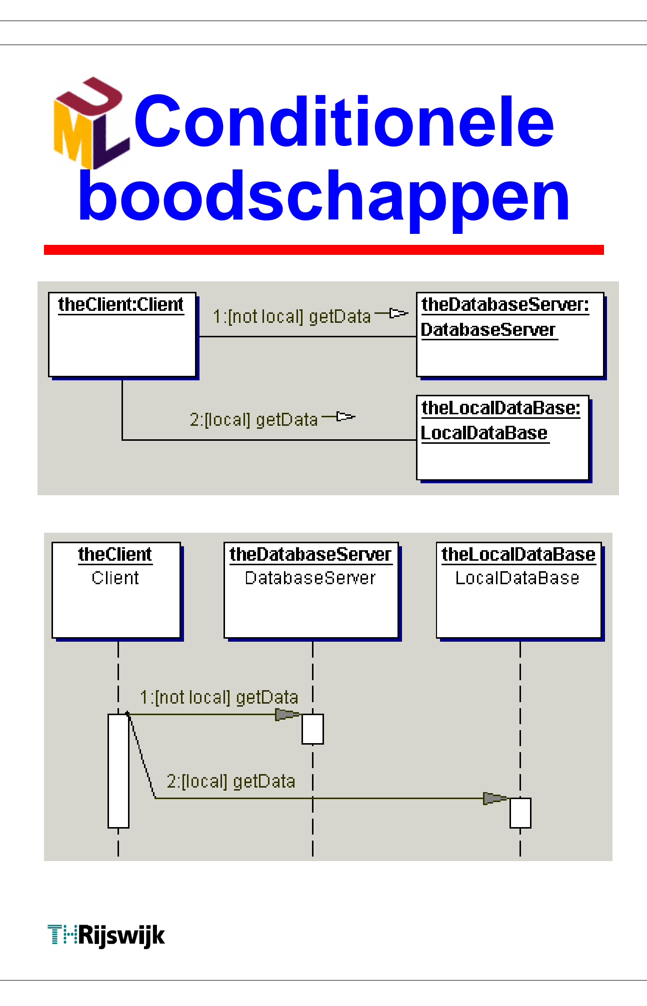


Synchroon en Asynchroon

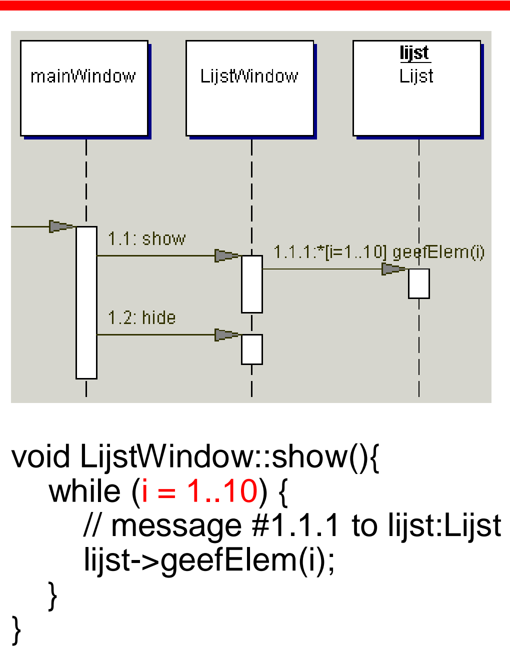


Together 6 gebruikt nog de “oude” notatie voor asynchrone boodschappen.

THRijswijk



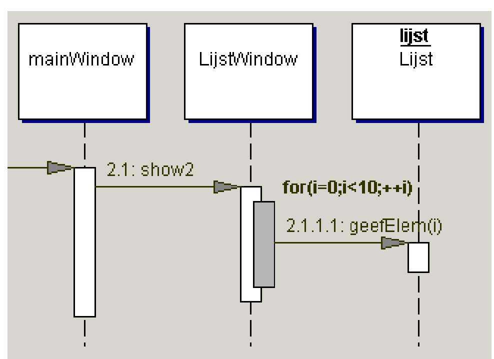
Iteratie van boodschappen




```
void LijstWindow::show(){
    while (i = 1..10) {
        // message #1.1.1 to lijst:Lijst
        lijst->geefElem(i);
    }
}
```

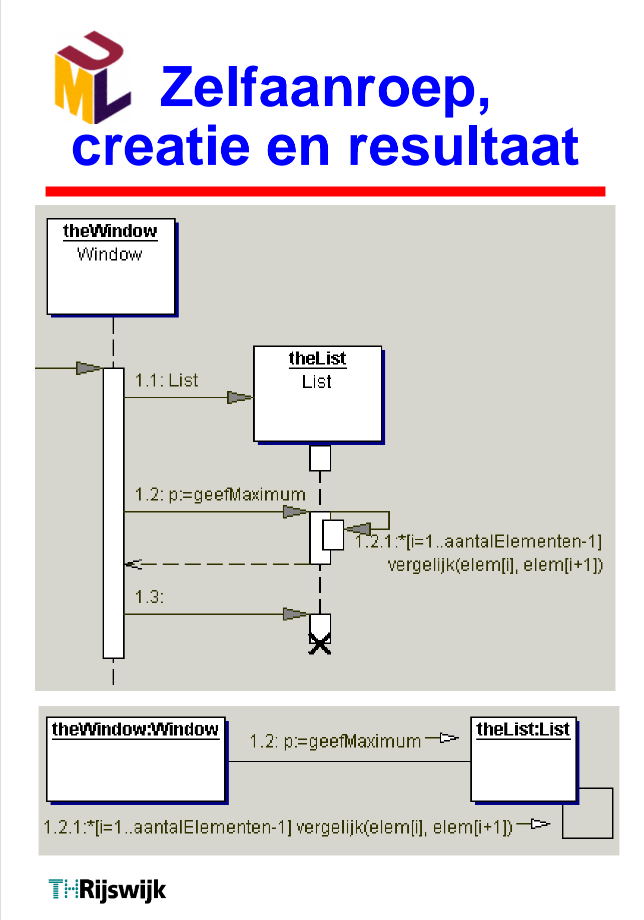
THRijswijk

Iteratie van boodschappen

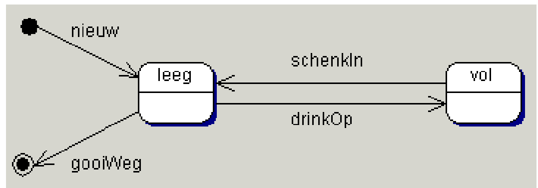



```
void LijstWindow::show2(){
    for (i = 0; i < 10; ++i) {
        // message #2.1.1.1 to lijst:Lijst
        lijst->geefElem(i);
    }
}
```

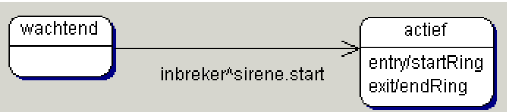

Zelfaanroep, creatie en resultaat



Toestands-diagram

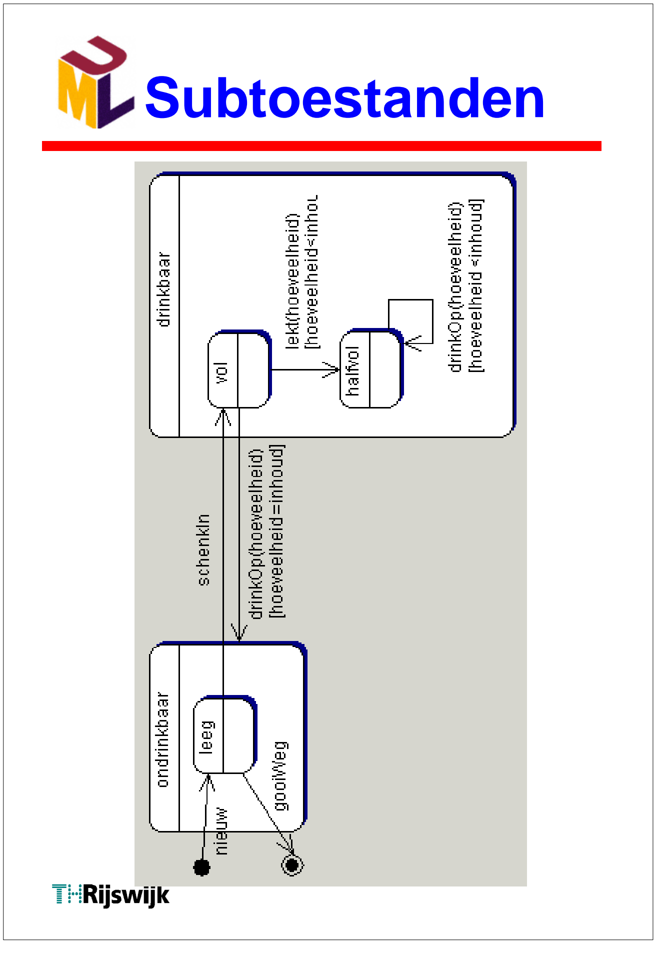


Toestandsdiagram bij de klasse Beker

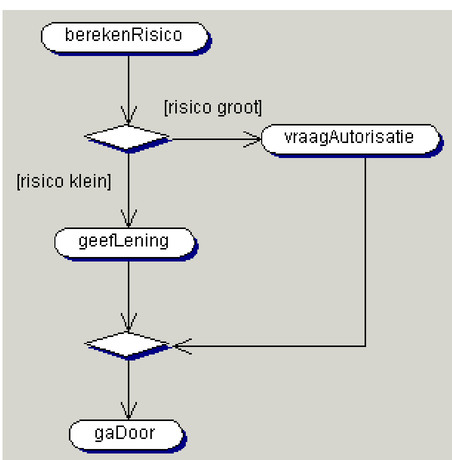


Toestandsdiagram met acties

Subtoestanden



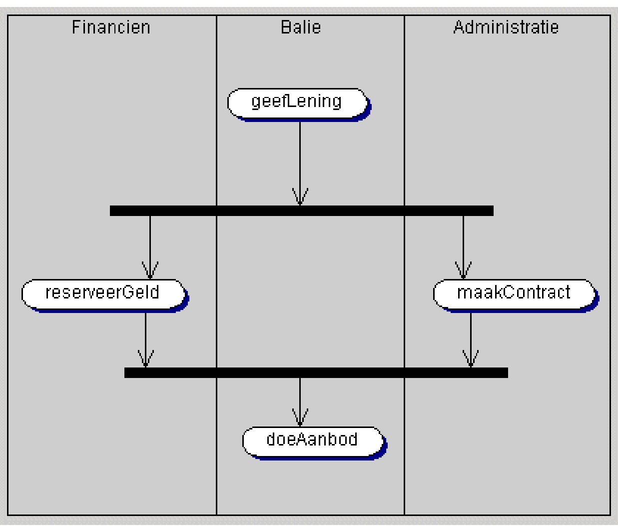
Activiteiten-diagrammen



- Gebruik van activiteitendiagram:
  - Workflow (hoog niveau)
  - Algorithmen
    - Operaties
    - Use-cases

© 2007 Harry Broeders

Splitsing, synchronisatie en swimlanes



Wat ik jullie nog niet vertelde over C++

- Static memberfuncties en static datamembers.
  - Zie TICPPV1 H10 Static members in C++
- Private (en protected) inheritance.
  - Zie TICPPV1 H14 Private inheritance
- Multiple inheritance.
  - Zie TICPPV2 H9

© 2007 Harry Broeders

Memberfuncties en datamembers

Elk object heeft zijn eigen datamembers terwijl de memberfuncties door alle objecten van een bepaalde class "gedeeld" worden.

```
class Hond {
public:
    Hond(const string& n);
    void blaf() const;
private:
    string naam;
};

Hond::Hond(const string& n): naam(n) {
}

void Hond::blaf() const {
    cout<<naam<<" zegt: WOEF"<<endl;
}
```

Memberfuncties en datamembers

Stel nu dat we bij willen houden hoeveel objecten van de class Hond er op een bepaald moment bestaan dan zouden we dit als volgt kunnen doen:

```
int aantalHonden=0; //dit is een globale variabele

class Hond {
public:
    Hond(const string& n);
    ~Hond();
    void blaf() const;
private:
    string naam;
};

Hond::Hond(const string n): naam(n) {
    ++aantalHonden;
}
Hond::~Hond() { --aantalHonden; }
```

static

Een static datamember is een onderdeel van de class en wordt door alle objecten van de class gedeeld.

```
class Hond {
public:
    Hond(const string& n);
    ~Hond();
    void blaf() const;
    static int aantal();
private:
    string naam;
    static int aantalHonden;
};

int Hond::aantalHonden=0;

Hond::Hond(const string& n): naam(n) {
    ++aantalHonden;
}
Hond::~Hond() { --aantalHonden; }
int Hond::aantal() { return aantalHonden; }
void Hond::blaf() const {
    cout<<naam<<" zegt: WOEF"<<endl; }
```

Hond
-naam:string -aantalHonden:int
+Hond(n:const string &) +~Hond() +blaf():void +aantal():int

UML boek: H4.4.4

Static memberfuncties

- Twee manieren van aanroepen:
  - via een object van de class:
    object_naam.member_functie_naam(parameters)
    Voorbeeld: cout<<h1.aantal()<<endl;
  - direct via de classnaam:
    class_naam::member_functie_naam(parameters)
    Voorbeeld: cout<<Hond::aantal()<<endl;
- Beperkingen t.o.v. een gewone memberfunctie:
  - Een static memberfunctie heeft **geen** receiver (ook niet als hij via een object aangeroepen wordt).
  - Een static memberfunctie heeft dus **geen** this pointer.
  - Een static memberfunctie kan dus **geen** "gewone" memberfuncties aanroepen en ook **geen** "gewone" datamembers gebruiken.

© 2007 Harry Broeders

Java

Static memberfuncties en static datamembers.

- Java heeft net als C++ ook static methods (= static memberfuncties) en static fields (=static datamembers).
- In Java worden static methods meer gebruikt dan in C++ omdat in Java geen "losse" functies gedefinieerd kunnen worden.
- In plaats van een "losse" functie wordt dan een static method gebruikt, zo is bijvoorbeeld de main() functie uit C en C++ in Java een static method.

© 2007 Harry Broeders

Private (en protected) inheritance

- Naast public inheritance kent C++ ook private en protected inheritance.
- Bij private inheritance zijn alle overerfde datamembers en memberfuncties uit de base class private in de derived class.
- De private derived class is **niet** polymorf met zijn private base class.
- De "is een" relatie is bij private inheritance dan ook niet geldig.
- Private inheritance = implemenation inheritance.
- Public inheritance = implementation inheritance + interface inheritance.

Java

~~Private (en protected) inheritance.~~

- Java kent in tegenstelling tot C++ geen implementation inheritance.
- Inheritance is in Java altijd public (en nooit private of protected) en wordt met het keyword **extends** aangegeven.
- Zowel de implementatie als de interface van de base class wordt in dit geval overerft.
- Java kent echter in tegenstelling tot C++ wel aparte syntax voor interface inheritance.

Java interfaces

- In Java bestaat de mogelijkheid om alleen de interface van een type te specificeren zonder de implementatie te definiëren. (**interface**)
- Een class kan dan deze interface overerven, dit wordt met het keyword **implements** aangegeven.
- In C++ kan dit d.m.v. een pure ABC (Abstract Base Class).
- In C++ is hier echter geen speciale syntax voor nodig. Zie multiple inheritance.

Hond interface

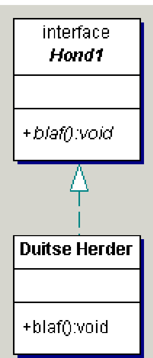
```
interface Hond {
   void blaf();
   //...
}
```

```
public class Tekkel implements
Hond {
   public void blaf() {
      System.out.println("Kef kef");
   }
}
```


© 2007 Harry Broeders

Hond interface

```
/** @interface */
class Hond1 {
public:
    virtual void blaf() = 0;
};

class Duitse_Herder :
                  public Hond1 {
public:
    virtual void blaf();
};
```



UML boek: H4.4.12

Multiple inheritance

Het overerven van meerdere base classes **tegelijk** wordt multiple inheritance genoemd.
Een bekend voorbeeld is de class AmfibieVoertuig die overerft van zowel de class Vaartuig als van de class Voertuig:

```
class AmfibieVoertuig: public Vaartuig,
    public Voertuig {
// ...
};
```

Voertuig
Vaartuig
AmfibieVoertuig

UML boek: H4.4.10

Hoewel multiple inheritance conceptueel erg leuk is, treden er in de praktijk veel problemen bij op. Vooral bij **multiple implementation inheritance**.

Java

~~Multiple inheritance.~~

- Java ondersteund **geen** multiple implementation inheritance. Je kunt maar van **één** class overerven (extends).
- Java ondersteund **wel** multiple interface inheritance. Je kunt van **meerdere** interfaces overerven (implements).