VDAB
TRAINING & OPLEIDING

Vlaamse Dienst voor Arbeidsbemiddeling en Beroepsopleiding

OBJECT ORIËNTATIE

Inhoudsopgave

1 INLEIDING

1.1 Vereiste voorkennis

Gestructureerd programmeren

1.2 Doel van de cursus

Deze cursus heeft als doel een inleiding te bieden tot object oriëntatie bij softwareontwikkeling.

Deze cursus heeft niet als doel één van de object georiënteerde programmeertalen aan te leren. Hiervoor bestaan andere cursussen en boeken. Deze kan je doornemen na deze cursus.

In deze cursus worden bepaalde concepten getoond met schema's uit de taal UML (Unified Modeling Language). Deze cursus heeft echter niet als doel UML volledig te beschrijven. Hiervoor bestaan andere cursussen en boeken.

2 OBJECT ORIËNTATIE T.O.V. GESTRUCTUREERD PROGRAMMEREN

2.1 Algemeen

Veel moderne programmeertalen (C++, Java, Visual Basic.net, C#, PHP, ...) zijn gebaseerd op object oriëntatie. De meeste ietwat oudere programmeertalen (C, Pascal, Basic, ...) zijn gebaseerd op gestructureerd programmeren. Bij object oriëntatie maak je een applicatie op een andere manier dan bij gestructureerd programmeren.

In dit hoofdstuk lees je een algemene inleiding tot object oriëntatie, en waarom deze manier van werken de voorkeur krijgt ten opzichte van gestructureerd programmeren.

2.2 Gestructureerd programmeren

Bij gestructureerd programmeren bestudeer je de te programmeren opdracht. Daarna ontwerp je een verzameling routines die de taken van deze opdracht kunnen uitvoeren.

Als een routine te groot wordt, splits je ze op in deelroutines. Dit opsplitsen heet functionele decompositie. Je gaat door met routines op te splitsen tot ze klein genoeg zijn om uit te coderen en te begrijpen.

De meeste van de routines hebben ook data nodig waarmee de routines werken. Deze data wordt tijdelijk bijgehouden in het interne geheugen en meestal opgeslagen in een bestand (tekstbestand, database, ...)

Hoe die data er in het interne geheugen uitziet wordt beschreven in een apart onderdeel van je programma: een structuur.

In het volgende voorbeeld wordt in de programmeertaal C een kleur beschreven in een structuur als een samenstelling van een rode, groen en blauwe intensiteit. De intensiteiten worden beschreven als gehele getallen (int)

```
struct Kleur {
  int rood;
  int groen;
  int blauw;
};
```

Opmerking: Het is niet de bedoeling in deze cursus de programmeertaal C te leren.

Typisch voor gestructureerd programmeren is de scheiding van code (in routines) ten opzichte van data (in structuren).

Een groter voorbeeld is de automatisering van een opleidingsinstelling.
Het programma beheert:

- de informatie over studenten
- de informatie over lesgevers
- de informatie over de cursussen van de opleidingsinstelling
- de informatie over welke studenten welke cursussen volgen.

Bij gestructureerd programmeren kunnen onder andere volgende routines voorkomen om de opleidingsinstelling te automatiseren:

- StudentToevoegen
- StudentInschrijvenVoorCursus
- ExamenResultatenVanStudent
- StudentVerwijderen

De data van het programma wordt beschreven in volgende structuren:

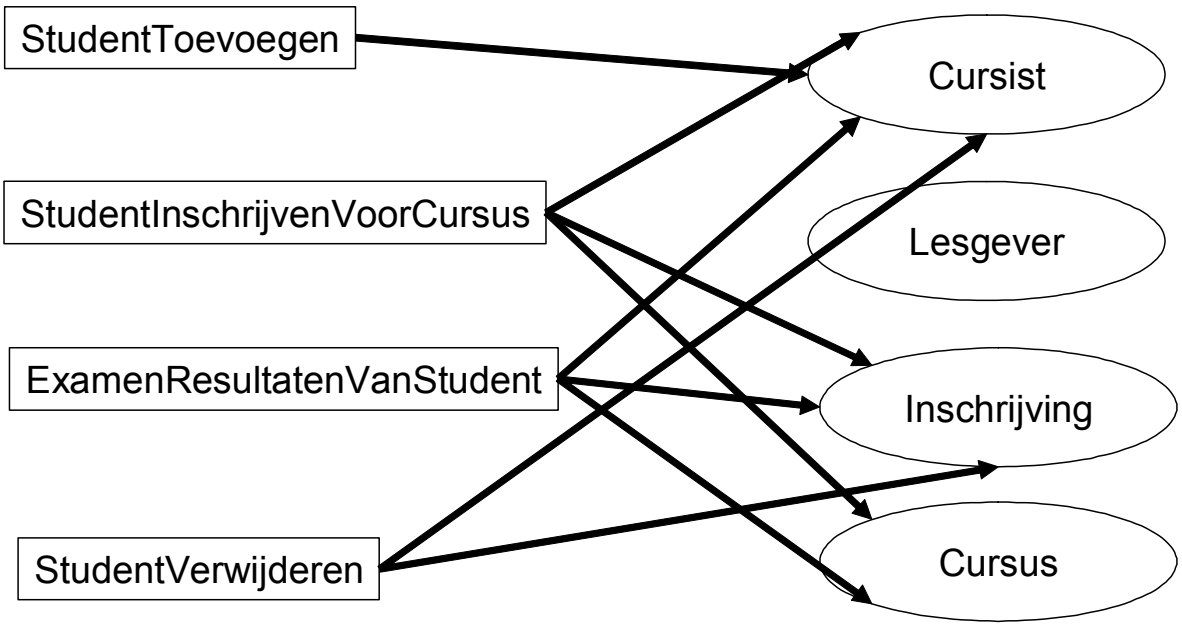
- Cursist
  Deze structuur bevat bvb. cursistnr., naam, voornaam, woonplaats
- Lesgever
  Deze structuur bevat bvb. lesgevernr, naam, voornaam, telefoonnummer
- Cursus
  Deze structuur bevat bvb. cursuscode, cursusnaam
- Inschrijving
  Deze structuur bevat cursistnr., cursuscode, prijs en examenresultaat

Opmerking: Om de structuren vast te leggen in een relationele database, moet je ook de nodige tabellen definiëren in die database:
Cursist, Lesgever, Cursus, Inschrijving.
Per tabel moet je ook de nodige kolommen definiëren.

De routines werken in op de structuren:

- De routine StudentToevoegen zal een nieuwe structuur Cursist aanmaken.
- De routine ExamenResultatenVanStudent zal informatie ophalen uit een Cursist structuur, een Inschrijving structuur en een Cursus structuur.

Het volgende schema toont het verband tussen de routines en de structuren waarop die routines inwerken:



Naarmate het programma groeit, wordt het verband tussen de routines en de structuren complexer en complexer. Daardoor wordt het programma moeilijk te onderhouden.

We bekijken dit met een voorbeeld. De prijs van een inschrijving voor een cursus wijzigt voortdurend, afhankelijk van interne en externe factoren.

Voorbeelden van interne factoren:

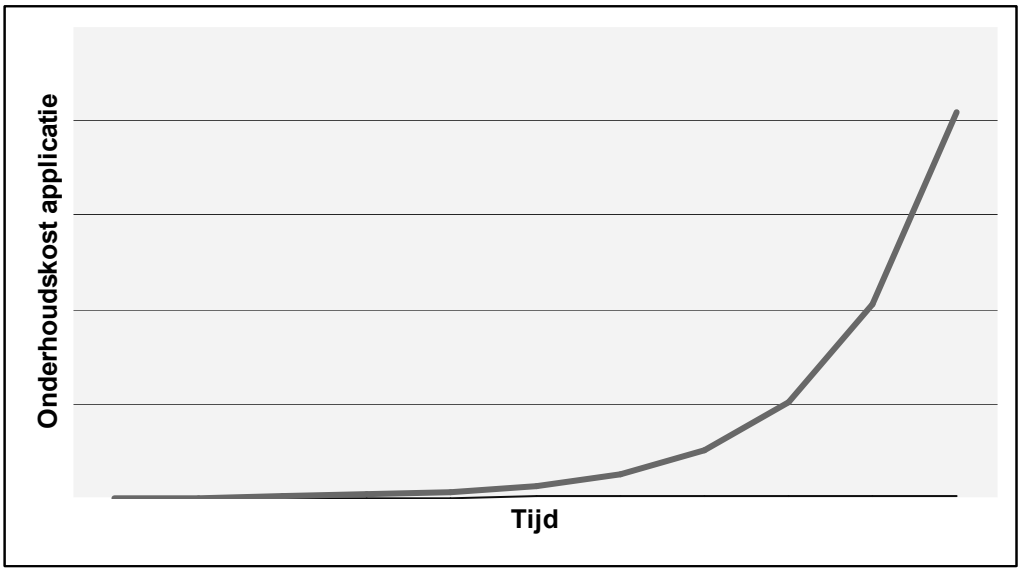
- Een firma die veel mensen tegelijk inschrijft voor een cursus, krijgt een goedkopere prijs dan een firma die maar één persoon inschrijft.
- Een firma die vroeger al veel mensen ingeschreven heeft, krijgt een goedkopere prijs dan een firma die voor het eerst mensen inschrijft.
- In juli en augustus (komkommermaanden) geven we een korting op alle inschrijvingen, om meer mensen naar een cursus te lokken.

Voorbeelden van externe factoren:

- Sinds een aantal jaren krijgt een firma die mensen inschrijft voor bepaalde cursussen van de overheid korting via opleidingscheques.
- Enkele jaren daarna beslist de overheid een soortgelijke korting toe te kennen aan werknemers die op eigen initiatief een cursus volgen.

Gezien veel routines de prijs van een inschrijving kunnen instellen en wijzigen wordt het moeilijk de prijs nog juist te berekenen. Bij middengrote tot grote applicaties wordt dit nog een groter probleem, gezien je dan met meerdere applicatieontwikkelaars samenwerkt, en ieder van deze ontwikkelaars routines kan hebben die de prijs van een inschrijving wijzigt.

Naarmate de applicatie groeit, stijgt de duurtijd om een aanpassing te doen exponentieel. Gezien bij softwareontwikkeling de menselijke kost (hoeveel uren duurt het om iets te verwezenlijken) de belangrijkste kost is, stijgt dus ook de onderhoudskost van de applicatie exponentieel:



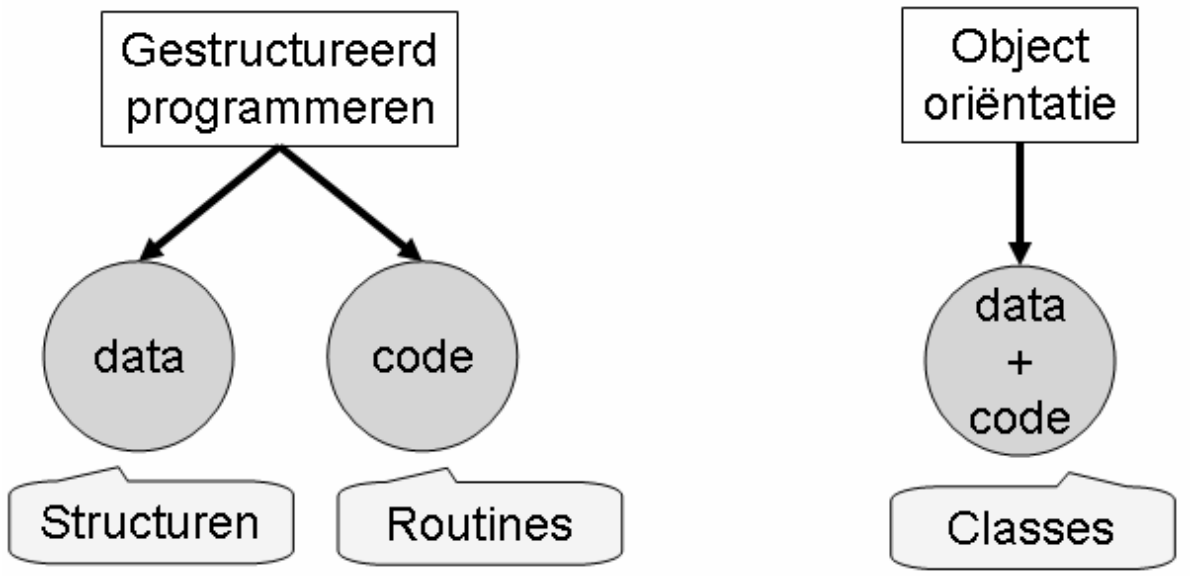
Als bij de aanpassingen nog eens fouten gemaakt worden, zijn de kosten niet enkel kosten van applicatieontwikkeling, maar ook kosten die de firma heeft met opgelopen schade (verkeerde inschrijvingen die geannuleerd worden, laattijdige betalingen van facturen, ...)

2.3 Object oriëntatie

2.3.1 Algemeen

Object oriëntatie probeert dit probleem op te lossen door code en data samen te brengen in één geheel. Zo een geheel heet een class.

In het volgende overzicht zie je bij gestructureerd programmeren de scheiding van code en data en bij object oriëntatie de fusie van code en data.



Bij het voorbeeld van de opleidingsinstelling kunnen volgende classes voorkomen:

Cursist	Lesgever	Cursus	Inschrijving
cursistNr voornaam naam woonplaats	lesgeverNr voornaam naam telefoonnummer	cursusCode naam	cursistNr cursusCode prijs examenresultaat
toevoegen() verwijderen()	toevoegen() verwijderen()	toevoegen() verwijderen()	toevoegen() verwijderen() opvragenExamenResultaat()

Classes worden voorgesteld door rechthoeken met drie compartimenten:

- bovenste compartiment: naam van de class
- middelste compartiment: data van de class
- onderste compartiment: routines van de class

De class Cursist beschrijft de data van een cursist (cursistNr, voornaam, ...) én de routines die op de cursist data inwerken (toevoegen en verwijderen).

De class Lesgever beschrijft de data van de lesgever én de routines die op de data van de lesgever inwerken.

De class Cursus beschrijft de data van de cursus én de routines die op de data van de cursus inwerken.

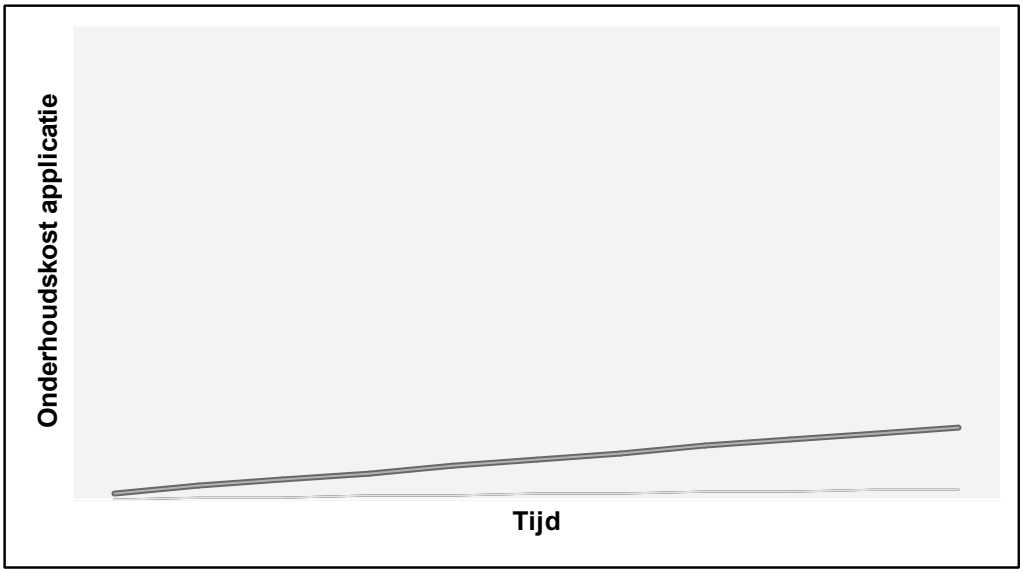
De class Inschrijving beschrijft de data van een inschrijving én de routines die op de data van een inschrijving inwerken.

2.3.2 Object oriëntatie als kostenbesparing

Enkel de routines van de class Inschrijving kunnen inwerken op de data van de class Inschrijving. Routines buiten de class Inschrijving kunnen de data van de class Inschrijving niet bijwerken.

Nu is het eenvoudiger om de steeds wijzigende prijzenpolitiek van de opleidingsinstelling ook in de applicatie toe te passen. Enkel de routine toevoegen van de class Inschrijving kan de prijs van een inschrijving instellen. Dus moet enkel deze routine aangepast worden bij een nieuwe prijzenpolitiek.

Naarmate de applicatie groeit, stijgt de duurtijd om een aanpassing te doen niet meer exponentieel, maar lineair. Gezien bij softwareontwikkeling de menselijke kost (hoeveel uren duurt het om iets te verwezenlijken) de belangrijkste kost is, stijgt dus ook de onderhoudskost van de applicatie minder snel:



Gezien je minder routines moet aanpassen is de kans op fouten ook kleiner. De kosten die de firma heeft met opgelopen schade (laattijdige leveringen van goederen, laattijdige betalingen van facturen, ...) zijn dus ook kleiner.

Opmerking: Het principe dat de routines en data die bij elkaar horen verzameld zijn in één class en enkel de routines van een class de data van de class mogen zien en wijzigen is een belangrijk principe in object oriëntatie en heet inkapseling (Engels: encapsulation).

2.3.3 Object oriëntatie als modellering van de werkelijkheid

Los van het voordeel dat je bij object oriëntatie gemakkelijker wijzigingen kunt aanbrengen dan bij gestructureerd programmeren, is object oriëntatie ook een betere voorstelling van de werkelijkheid in je code.

In de werkelijkheid denk je ook niet gescheiden over enerzijds de data van de dingen en anderzijds de handelingen die je op de dingen doet. Je denkt niet gescheiden over enerzijds het rekeningnummer en saldo van je spaarrekening en anderzijds de handelingen die je op die spaarrekening doet (storten, afhalen).

Bij gestructureerd programmeren wordt die "artificiële" scheiding opgedrongen. Bij object oriëntatie kan je in één class Spaarrekening de data van de spaarrekening (rekeningnummer, saldo) én de handelingen van de spaarrekening (routines storten en afhalen) beschrijven.

Dat object oriëntatie de werkelijkheid beter voorstelt dan gestructureerd programmeren wordt in technische termen uitgedrukt als:
Object oriëntatie is een betere abstractie van de werkelijkheid dan gestructureerd programmeren.

2.3.4 Object oriëntatie en hergebruik van softwareonderdelen

Object oriëntatie maakt het gemakkelijker concepten opnieuw te gebruiken in meerdere programma's. Als je in meerdere programma's met een spaarrekening moet werken, kan je de class Spaarrekening opnieuw gebruiken. Bij ieder hergebruik heb je én de data én de handelingen van een spaarrekening ter beschikking, want data en handelingen zitten samen in één class.

Bij gestructureerd programmeren moet je bij hergebruik van het concept spaarrekening enerzijds de data en anderzijds de bijbehorende routines opzoeken in het oorspronkelijke programma, want die zijn van elkaar gescheiden.

2.4 OOA en OOP

Bij object oriëntatie wordt aandacht besteed aan analyse én aan programmeren.

Als je analyse doet met toepassing van object oriëntatie spreekt men over OOA: Object Oriented Analysis. De meest gebruikte taal om je analyse met object oriëntatie uit te drukken is UML: The Unified Modeling Language (www.uml.org).

Als je object oriëntatie toepast bij het programmeren spreekt men over OOP: Object Oriented Programming.

Enkele populaire programmeertalen die object oriëntatie ondersteunen:

- c++ programmeertaal ontwikkeld door Bjarne Stroustrup. De programmeertaal wordt gecontroleerd door een ANSI comité.
- Java programmeertaal ontwikkeld door de firma Sun
- Visual Basic.net programmeertaal ontwikkeld door de firma Microsoft.
- C# programmeertaal ontwikkeld door de firma Microsoft.
- PHP programmeertaal uit de open-source beweging

3 CLASS

3.1 Algemeen

Een class (Nederlands: klasse) is een weergave van een begrip uit de te automatiseren werkelijkheid.

Een class beschrijft iets dat een zelfstandig bestaan leidt in de werkelijkheid.

Het is belangrijk de werkelijkheid zo volledig mogelijk in de code te beschrijven. Pas dan maak je software die bij de werkelijkheid past. Het is dus belangrijk alle begrippen uit de te automatiseren werkelijkheid als classes te beschrijven.

Tastbare (fysieke) begrippen uit de werkelijkheid kan je gemakkelijk detecteren. Als je een bibliotheek automatiseert, ga je eens rondwandelen in die bibliotheek om de werkelijkheid van die bibliotheek te leren kennen. Bij het rondwandelen ontdek je zeker volgende tastbare begrippen: boek, lezer. Deze begrippen zal je in je applicatie uitwerken als de classes Boek en Lezer.

Moeilijker te detecteren zijn abstracte begrippen in de te automatiseren werkelijkheid. In een bibliotheek is het begrip genre ook belangrijk. Je kunt het begrip genre echter niet fysiek vastnemen. Je leert het wel kennen in gesprekken met medewerkers van de bibliotheek en in documenten over de bibliotheek. Een dergelijk abstract begrip leidt uiteindelijk ook tot een class (Genre genoemd) in je applicatie.

Bepaalde begrippen zijn tastbaar (fysiek) maar zal je in de werkelijkheid niet fysiek detecteren. In een bibliotheek is het begrip auteur een fysiek begrip: je zou een auteur kunnen aanraken. De kans dat ik echter een auteur in de bibliotheek ontmoet is klein. Om dit begrip te detecteren zal je moeten terugvallen op gesprekken met bibliotheekmedewerkers, documenten over de bibliotheek of de studie van andere fysieke begrippen (in een boek wordt naar een auteur verwezen). Uiteindelijk zal het begrip auteur ook leiden tot een class (Auteur genoemd) in je applicatie.

Van één begrip vind je in de werkelijkheid meestal meerdere voorkomens. Van het begrip boek vind je in een bibliotheek duizenden voorkomens: je vindt duizenden boeken. Van het begrip lezer zie je in de bibliotheek tientallen voorkomens: in de bibliotheek wandelen tientallen lezers rond.

Je moet een begrip maar één keer als een class in je applicatie beschrijven, ongeacht het aantal voorkomens van dat begrip. Je moet dus het begrip boek maar één keer als class Boek beschrijven, hoewel er duizenden boeken bestaan. In die class beschrijf je dat een boek een ISBN nummer heeft, een titel, ... Dit geldt voor ieder van de duizenden boeken in de bibliotheek. Zo ook moet je het begrip lezer maar één keer als class lezer beschrijven, hoewel je tientallen Lezers ziet rondwandelen in de bibliotheek. In die ene class beschrijf je dat een lezer onder andere een voornaam en een familienaam heeft. Deze beschrijving geldt voor ieder van de lezers uit de bibliotheek.

In een class geef je de beschreven eigenschappen geen waarde. Je beschrijft dat een boek een ISBN nummer heeft, maar je vult geen concreet ISBN nummer in. Als je dit wel zou doen, zou de class Boek enkel dienen als beschrijving van het ene boek met dat ISBN nummer.

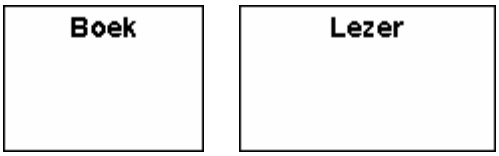
Je moet een class zo schrijven dat de beschrijving bruikbaar is voor alle boeken van de bibliotheek.

Er wordt dan ook gezegd dat een class een matrijs of sjabloon is van één of meerdere gelijkaardige voorkomens. De class Boek beschrijft het sjabloon van alle boeken van de bibliotheek.

In het hoofdstuk Object zal je zien hoe je op basis van een class de concrete voorkomens in je applicatie beschrijft.

In UML teken je een class als een rechthoek met daarin de naam van de class. De naam van de class staat in vetjes boven in de rechthoek.

Voorbeeld: de classes Boek en Lezer:



4 ATTRIBUUT

4.1 Algemeen

Een Attribuut is informatie die bij een begrip uit de werkelijkheid hoort.

Je beschrijft een attribuut in de class die het begrip uit de werkelijkheid beschrijft. Een synoniem voor attribuut is eigenschap (Engels: property).

Van een lezer uit de bibliotheek wil je zeker volgende attributen bijhouden: naam, voornaam, woonplaats, geboortedatum.

In je applicatie voeg je aan de class Lezer de attributen naam, voornaam, woonplaats en geboortedatum toe.

In UML schrijf je de attributen in de rechthoek van hun class in een compartiment onder de naam van de class:

4.2 Datatypes voor attributen

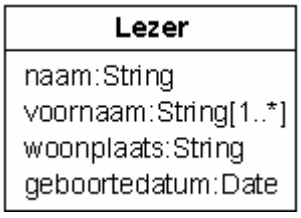
Meestal zal je ook per attribuut beschrijven welk datatype je gebruikt om dit attribuut bij te houden. Bij de attributen naam, voornaam en woonplaats kan je het datatype String gebruiken, bij het attribuut geboortedatum het datatype Date.

In UML geef je een attribuut een datatype door de naam van het attribuut te volgen door een dubbel punt en dan het datatype:

4.3 Meervoudig attribuut

Als een eigenschap meerdere keren voorkomt in een begrip heet dit een meervoudig attribuut. Een lezer kan bijvoorbeeld meerdere voornamen hebben. Dan is het attribuut voornaam een meervoudig attribuut.

In UML kan je een attribuut als meervoudig aanduiden: op het attribuut voornaam volgt [1..*]. Dit wil zeggen dat één lezer één of meerdere voornamen heeft.



4.4 Attribuut ten opzichte van class

Een class stelt een begrip uit de werkelijkheid voor. Je kunt veel attributen van een class ook aanzien als begrippen. Naam (attribuut van de class Lezer) is evengoed een begrip als lezer. Wil dit nu zeggen dat je van naam ook een class moet maken?

Bij deze twijfel geldt de volgende vuistregel: als je van een attribuut enkel de waarde wil bijhouden, moet je voor dit attribuut geen aparte class maken. Als je op een attribuut naast de waarde ook andere functionaliteit wil uitvoeren, maak je voor dit attribuut een aparte class.

Een voorbeeld van een attribuut dat je ook in een aparte class kan beschrijven is het attribuut emailadres van de class Lezer. Van dit attribuut wil je meer dan enkel de waarde kennen. Je wilt ook controleren of het emailadres correct geschreven is (een serie tekens, gevolgd door een @, gevolgd door een serie tekens, gevolgd door een punt, gevolgd door een serie tekens). Misschien wil je wel controleren of dit emailadres wel echt op het Internet voorkomt. Wegens deze extra functionaliteit maak je voor het attribuut emailadres een extra aparte class: EMailAdres.

In deze class zullen routines voorkomen die controleren of het emailadres correct geschreven is en of het echt voorkomt op het Internet.

Om te beschrijven dat een lezer een emailadres heeft, zal je een associatie leggen tussen de class Lezer en de class Emailadres. Dit komt aan bod in het hoofdstuk associaties tussen Classes.

Bij sommige beginnende object oriëntatie ontwikkelaars slaat nu de schrik om het hart dat ze bij een grote applicatie veel classes zullen hebben. Een vuistregel van object oriëntatie is echter dat je beter veel classes hebt die elk hun eigen stuk van de werkelijkheid beschrijven dan weinig classes waarbij één class een groot stuk van de werkelijkheid probeert te beschrijven.

Elke class is verantwoordelijk voor één deeltje van de te automatiseren werkelijkheid. Dit wordt in technische termen uitgedrukt met het woord *responsability* (verantwoordelijkheid).

Het voordeel van het opzetten van een systeem als een verzameling classes die elk een deel van het werk doen is dat je achteraf gemakkelijk aanpassingen en correcties kan doen aan het systeem.

Als in de applicatie voor de opleidingsinstelling de inschrijvingen gratis worden, weet je dat je voor deze aanpassing enkel moet werken in de code van de class Inschrijving. Als er het programma een fout veroorzaakt bij het tonen van de inhoud van een cursus, weet je dat je deze fout moet opzoeken in de code van de class Cursus.

Je kunt dit idee van verantwoordelijkheid vergelijken met de werking van het menselijke lichaam. In het menselijke lichaam heeft ieder orgaan ook een duidelijke deelverantwoordelijkheid in het geheel:

- Het hart zorgt voor het stromen van het bloed.
- De longen zorgen voor de zuurstoftoevoer naar het bloed.
- De nieren verwijderen de vuile stoffen uit het bloed.

Als een chirurg een ingreep moet doen kan hij aan de hand van de fout (bvb. geen polslag) weten welk orgaan hij moet herstellen (het hart) en welke organen hij niet moet nazien (bvb. nieren)

5 OBJECT

5.1 Algemeen

Een object is één voorkomen van een begrip uit de werkelijkheid.

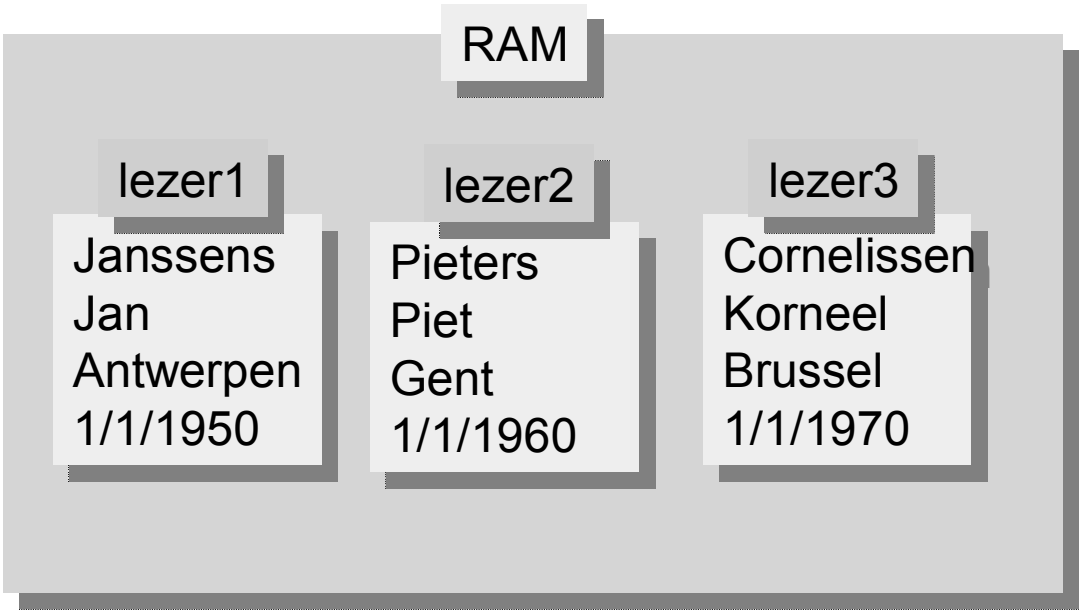
Als je het begrip lezer van de bibliotheek in een class Lezer beschreven hebt en je wil drie lezers in je software voorstellen, zijn dit drie objecten van het type Lezer.

Een synoniem voor object is instantie (Engels: instance).
De drie lezers zijn dus drie instanties van de class Lezer.

Waar een class een abstractie is van een begrip, is een object een concretisering van dat begrip met concrete waarden voor de attributen van de class die het begrip beschrijft.

Per object van een class houdt de software de attribuutwaarden bij in het interne geheugen (RAM).

De software voorziet voor ieder van de drie objecten ruimte in het interne geheugen voor de attributen naam, voornaam, woonplaats en geboortedatum:



5.2 Voorbeelden van aanmaak objecten

5.2.1 Een nieuwe lezer registreren in een bibliotheek

Een Lezer object wordt aangemaakt in het onderdeel van de applicatie waarmee de medewerkers van de bibliotheek nieuwe lezers toevoegen:

Als de medewerker op de knop OK klikt, maakt de applicatie een nieuw object van het type Lezer. Het attribuut voornaam van dit object wordt ingevuld met de inhoud van het eerste tekstvak. Het attribuut naam wordt ingevuld met de inhoud van het tweede tekstvak. Het attribuut woonplaats wordt ingevuld met de inhoud van het derde tekstvak. Het attribuut geboortedatum wordt ingevuld met de datum gekozen in de kalender.

Gezien objecten zich bevinden in het interne geheugen, blijft hun inhoud slechts behouden zolang de applicatie in uitvoering is. Om het Lezer object op langere tijd te bewaren, geeft de applicatie het object door aan een routine die het Lezer object toevoegt als een nieuw record in een table van een relationele database.

5.2.2 De gegevens van een boek afbeelden in een bibliotheek

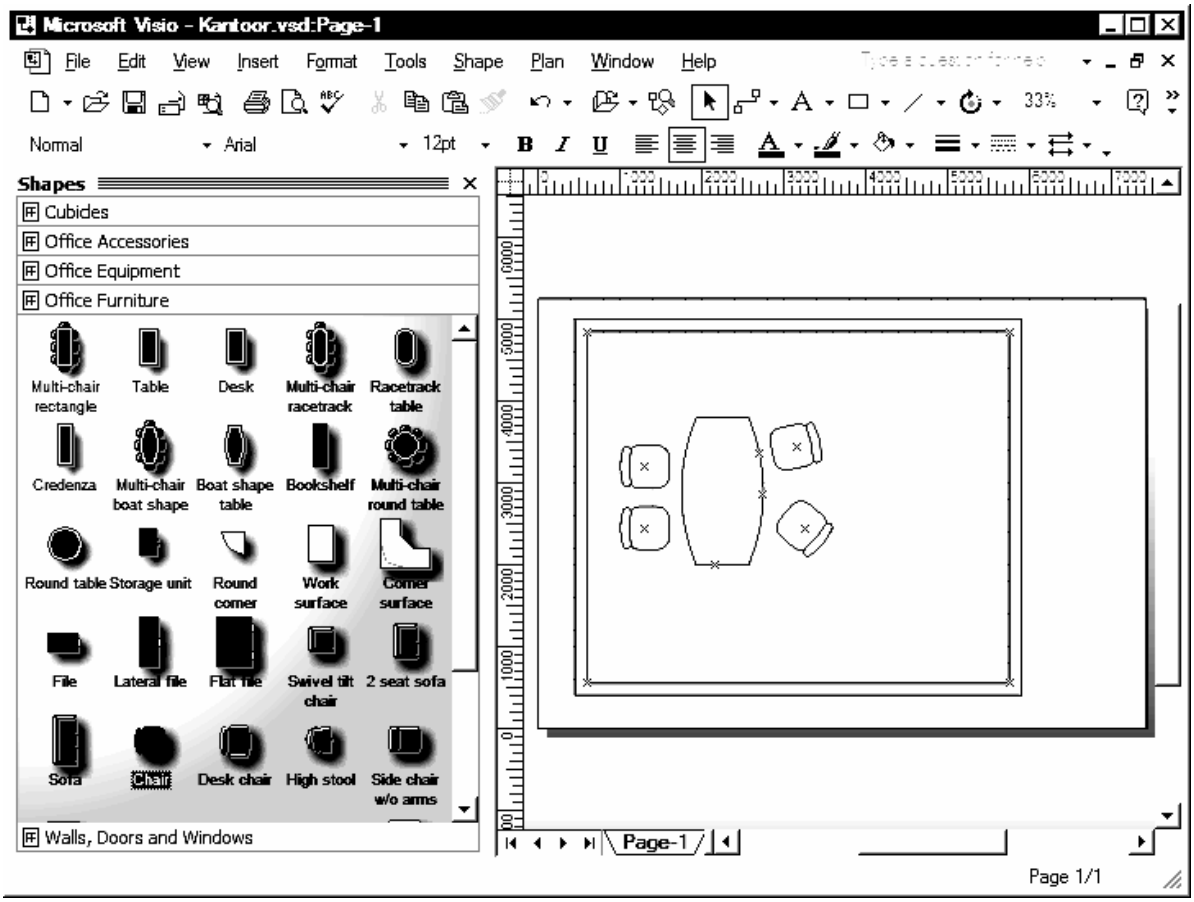
Ook bij het afbeelden van gegevens uit een relationele database op het scherm of op de printer, wordt een object aangemaakt.

Eerst geeft de gebruiker via een zoekoperatie aan welk gegeven hij uit de relationele database wil zien. Een routine zoekt het record in de relationele database. De applicatie maakt in het interne geheugen een nieuw object en vult de attributen van dit object in met de inhoud van het gevonden record. De gebruiker ziet in elementen van een dialoogvenster de inhoud van de attributen van het object.

In het volgende voorbeeld tikt de gebruiker een ISBN nummer van een boek en tikt daarna op de knop Zoeken. Dit boek wordt opgezocht in de relationele database. De applicatie maakt een nieuw Boek object. Daarna vult de applicatie de attributen van dit Boek object in met de inhoud van het gevonden record. Daarna toont de applicatie deze attributen als elementen van een dialoogvenster:

5.2.3 Een tekenprogramma

Met het volgende tekenprogramma maakt de gebruiker een opstelling van meubilair (tafels, stoelen, kasten, ...) en toebehoren (computers, printers, telefoons) van een kantoor.



Op de rechterhelft van de applicatie kan de gebruiker een kantoor samenstellen door kantooronderdelen uit de linkerhelft van de applicatie naar de rechterhelft te slepen. Als deze applicatie met object oriëntatie is opgebouwd, is ieder onderdeel

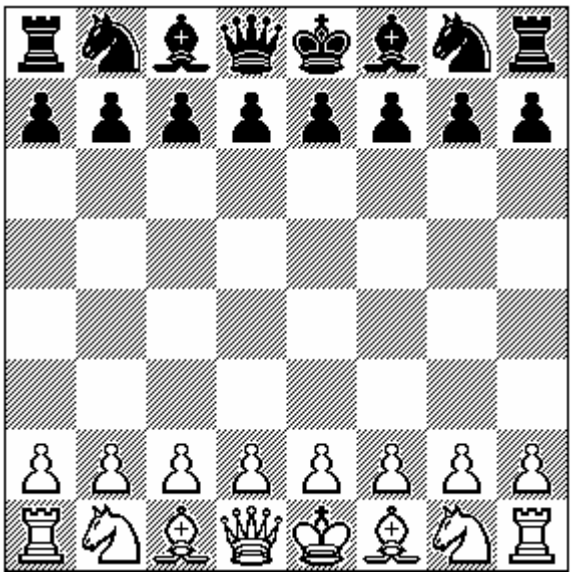
van een kantoor beschreven als een class. Je vindt dus de class Tafel, de class Stoel, de class Telefoon, ...

Ieder keer de gebruiker een kantooronderdeel van de linkerhelft naar de rechterhelft sleept, wordt een object van de juiste class aangemaakt. Als de gebruiker een tafel naar de rechterhelft sleept, wordt een object van de class Tafel aangemaakt.

Uiteindelijk is een kantoor een verzameling van allerlei objecten: objecten van het type Tafel, objecten van het type Stoel, ...

5.2.4 Een schaakspel

Een schaakspel is een spel met verschillende begrippen: pion, toren, loper, paard, koning, koningin:



Als je het schaakspel met object oriëntatie als softwareapplicatie maakt, zal je voor ieder van die begrippen een class maken. Deze classes hebben onder andere volgende attributen: kleur (zwart of wit), positie op het bord, ...

Als de gebruiker een nieuw spel begint worden volgende objecten aangemaakt;

- Vier objecten van de class Toren (twee witte, twee zwarte).
- Vier objecten van de class Loper (twee witte, twee zwarte).
- Vier objecten van de class Paard (twee witte, twee zwarte).
- Twee objecten van de class Koning (één witte, één zwarte).
- Twee objecten van de class Koningin (één witte, één zwarte).
- Zestien objecten van de class Pion (acht witte, acht zwarte).

5.3 Attributen met class bereik

Normaal heeft ieder object van een class zijn eigen geheugenruimte voor de attributen van dat object.

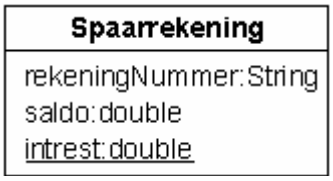
Je kunt ook een attribuut definiëren als een attribuut met class bereik. Dan is er voor dat attribuut maar één keer geheugenruimte, ongeacht het aantal objecten van die class. Alle objecten van de class delen de waarde van een dergelijk attribuut, het is een gedeeld attribuut.

Je gebruikt een attribuut met class bereik als de inhoud van het attribuut hetzelfde is voor alle objecten van een class.

Een voorbeeld is het intrestpercentage van de class Spaarrekening. Alle spaarrekeningen van een bank gebruiken hetzelfde intrestpercentage. Dan is het zinloos dat het attribuut intrestpercentage evenveel keer in het geheugen voorkomt als er zich Spaarrekening objecten in het interne geheugen bevinden. Dit zou zelfs nadelig zijn als het intrestpercentage wijzigt. Je zou deze wijziging bij alle Spaarrekening objecten in het interne geheugen moeten doorvoeren.

Als je het intrestpercentage als een attribuut met class bereik definieert, wordt voor dit attribuut maar één keer geheugenruimte voorzien, ongeacht het aantal Spaarrekening objecten in het interne geheugen.

In UML wordt een attribuut met class bereik onderstreept:



6 METHOD

6.1 Algemeen

Tot nu bevat een class enkel attributen. Dit is slechts de helft van het verhaal. Even belangrijk is de beschrijving van methods (handelingen) in de class.

Een method is een actie die een object van een class kan uitvoeren.
Een synoniem voor method is operatie (Engels: operation).

De method wordt beschreven als een routine in de class.

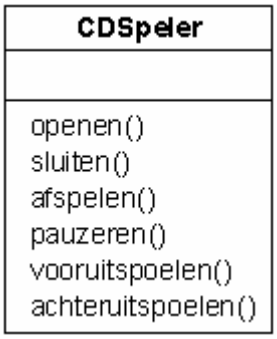
Bij uitvoering van de applicatie bevindt de code van een method zich maar één keer in het geheugen, ongeacht het aantal objecten van een class.

Als het geheugen van een class drie objecten bevat, bevat het geheugen maar één keer de code van de methods van die class.

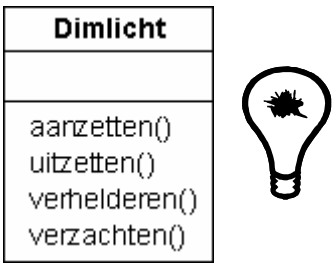
Een manier om de methods te vinden die je in een class dient uit te werken, is te kijken welke werkwoorden je in de werkelijkheid associeert met het begrip dat de class voorstelt.

Als je met het begrip Cd-speler de werkwoorden openen, sluiten, afspelen, pauzeren, vooruitspoelen en achteruitspoelen associeert, worden die werkwoorden ook methods van de class CDSpeler.

In UML worden de methods van een class voorgesteld als een extra compartiment onder het compartiment met attributen:



Een ander voorbeeld: een dimlicht met zijn eigen typische methods:



6.2 Method parameters

Als je bij het uitvoeren van een method een veranderlijk gegeven meegeeft aan die method, beschrijf je dit als een parameter van die method.

Bij het programmeren wordt dit een parameter van de routine die de method voorstelt. Iedere programmeertaal kan een routine voorzien van parameters.

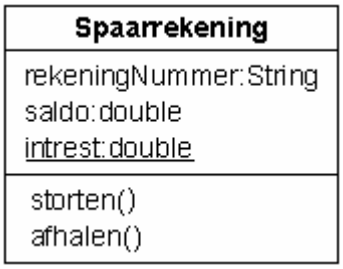
De class Spaarrekening bevat de methods storten (geld storten op de bankrekening) en afhalen (geld afhalen van de bankrekening).

De method storten wordt in de loop van de tijd meerdere keren uitgevoerd op een Spaarrekening object. Bij iedere uitvoering van de method storten kan het bedrag dat je stort verschillen.

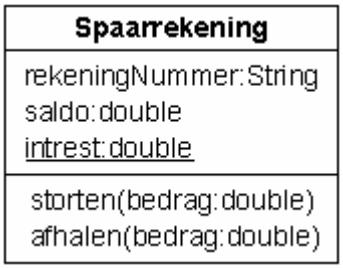
Ook bij de method afhalen kan het bedrag dat je afhaalt verschillen bij iedere uitvoering van de method afhalen.

Dit bedrag wordt dan een parameter van de method. De method storten krijgt dus een parameter bedrag én de method afhalen krijgt een parameter bedrag.

In UML worden de parameters beschreven tussen de ronde haakjes die volgen op de namen van de methods:



De parameters krijgen ook een datatype (tekst, getal, datum, ...).
In UML beschrijf je het datatype van een parameter na de naam van de parameter. Je scheidt de naam van de parameter en zijn datatype met een dubbele punt:

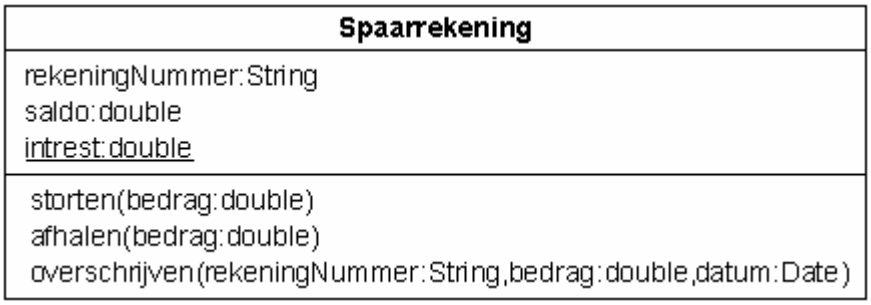


Een method kan meer dan één parameter hebben.

Overschrijven is ook een handeling die je op een spaarrekening uitvoert.
Bij deze handeling heb je drie veranderlijke waarden:

- Het rekeningnummer van de rekening waarnaar je overschrijft.
- Het bedrag dat je overschrijft.
- De datum waarop de overschrijving moet gebeuren.

Deze drie veranderlijke waarden worden drie parameters van de method overschrijven:



6.3 Method resultaatwaarde

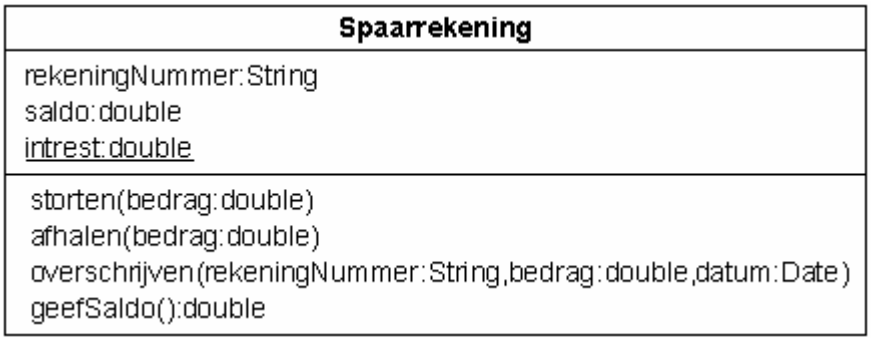
Sommige methods geven na het uitvoeren een resultaatwaarde terug aan de code die de method opgeroepen heeft.

Deze resultaatwaarde kan tekst zijn, een getal, een datum, een logische waarde (waar/onwaar), een object ...

Een method kan slechts één resultaatwaarde hebben.

Je kunt bijvoorbeeld de class Spaarrekening uitbreiden met een method geefSaldo. Deze method geeft aan de oproeper een getal terug met het huidige saldo van een Spaarrekening object.

In UML volgt bij een method met een resultaatwaarde na de ronde haakjes een dubbele punt en dan het soort gegeven (tekst, getal, ...) dat die method teruggeeft:



Je ziet dat de method geefSaldo gevolgd wordt door een dubbele punt en dan het woord double.

6.4 Waar zijn we ?

Tot nu hebben we het volgende geleerd:

- De wereld zit vol met dingen. Die dingen heten objecten.
- Soortgelijke objecten behoren tot een class.
- Objecten hebben attributen (data) en methods (functies).
  Deze attributen en methods worden in de class beschreven.
  Opmerking: De verzameling attributen en methods van een class heten de members van die class.
- Van één class kunnen er meerdere objecten zijn. Ieder object heeft dan zijn eigen attribuutwaarden. (Mijn sportwagen heeft als waarde voor het attribuut kleur groen, jouw sportwagen heeft als waarde voor het attribuut kleur rood).

7 DATA HIDING

7.1 Algemeen

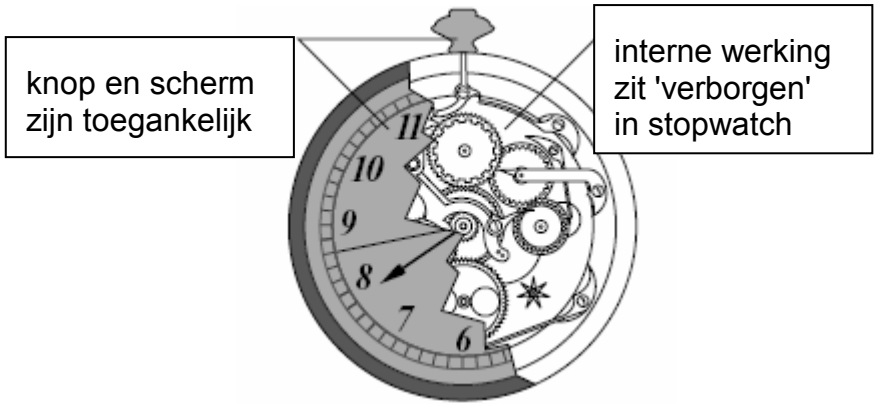
Bij OOP is een belangrijk idee dat je moet weten wat een class voor je kan doen, maar niet hoe de class dit doet.

Je moet dus weten welke methods je op een object van een class kan uitvoeren, maar je moet niet weten hoe de methods hun handeling uitvoeren.

Dit idee heeft belangrijke voordelen:

- Als een programmeur niet weet hoe een class zijn methods uitvoert, kan hij deze kennis niet misbruiken om de class verkeerde dingen te laten doen.
- Als een programmeur niet weet hoe een class zijn methods uitvoert, kan je de werking van deze methods wijzigen (verbeteren, optimaliseren), zonder dat de programmeurs die de class gebruiken van deze wijziging last hebben.
- De programmeur die de class gebruikt wordt niet geconfronteerde met de soms interne complexiteit van de class code.

Ook in de echte wereld wordt de werking van complexe dingen weggemoffeld.
Op een stopwatch heb je een knop om de stopwatch te starten of te stoppen en een scherm waarop je de tijd afleest. Een gebruiker van de stopwatch hoeft enkel deze knop en het scherm te gebruiken. De interne werking van de stopwatch is voor de gebruiker niet belangrijk, wel voor de maker van de stopwatch:



Het is gemakkelijk de code van de methods te verbergen. Als maker van een class geef je enkel de gecompileerde versie van de class aan andere programmeurs, niet de source (bronbestand).

Opdat de andere programmeurs zouden weten welke methods de class bevat, welke parameters deze methods verwachten, en welke resultaatwaarden de methods teruggeven geef je ook documentatie aan de andere programmeurs. Deze documentatie kan in gedrukte vorm zijn, een verzameling HTML documenten, een helpbestand, ...

Naast methods bevat een class ook attributen. Het is belangrijk dat je als maker van de class toegang hebt tot deze attributen, maar niet de programmeurs die de class gaan gebruiken.

Anders gaan deze programmeurs misschien verkeerde waarden in de attributen invullen (bvb. een negatief saldo in een Spaarrekening object).
Als deze programmeurs toegang hebben tot de attributen, kan je ook niets meer wijzigen aan deze attributen, zonder hun code te breken.

7.2 Voorbeeld: Wachtrij

Een voorbeeld is een wachtrij voor een wachtzaal van een dokter.
Je moet op een wachtrij twee handelingen kunnen uitvoeren:

- Aan een lijst patiëntnamen toevoegen die in de wachtzaal aankomen.
- Uit de lijst de volgende patiëntnaam halen die van de wachtzaal naar de dokter mag gaan.

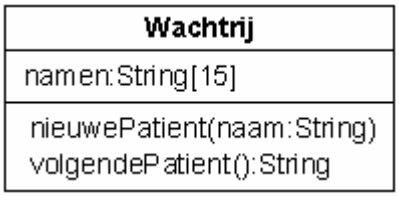
Bij object oriëntatie beschrijf je de wachtrij als een class.

Deze class heeft een meervoudig attribuut waarin je de namen van de patiënten in de wachtzaal bijhoudt. Dit kan een String array van 15 elementen zijn.

De class heeft een method nieuwePatient, waarmee je een patiënt die in de wachtzaal aankomt, toevoegt aan de wachtrij. Deze method heeft een String parameter met de naam van de nieuw aangekomen patiënt.

De class heeft een tweede method volgendePatient, waarmee de dokter de naam van de patiënt kan opvragen die van de wachtzaal naar de dokter mag komen. Deze naam is de resultaatwaarde van de method. Deze naam mag ook uit de wachtrij verdwijnen.

In UML ziet de class er als volgt uit:



De code van de method nieuwePatient ziet er als volgt uit:

```
nieuwePatient(naam)
index := 0
zolang namen[index] <> ""
    index := index + 1
namen[index] := naam
```

In de code doorlopen we de array tot we een lege plaats vinden. Op deze lege plaats vullen we de naam van de nieuw aangekomen patiënt in.

Opmerking: het volgnummer van het eerste element is in de meeste programmeertalen nul.

De code van de method volgendePatient ziet er als volgt uit:

```
volgendePatient
naam := namen[0]
voor index := 0 tm 13
    namen[index]=namen[index + 1]
:=naam
```

Opmerking: de laatste opdracht geeft de inhoud van de variabele naam terug aan de code die de method volgendePatient opgeroepen heeft.

De volgende patiënt die naar de dokter mag vinden we in het eerste element van de array. We onthouden deze naam in een variabele *naam*. Daarna schuiven we alle volgende elementen één element naar voor in de array. Zo verdwijnt de naam die eerste was uit de wachtrij en wordt de naam die tweede was in de wachtrij de eerste in de wachtrij. Op het einde van de method geven we de naam terug aan de code die de method opgeroepen heeft.

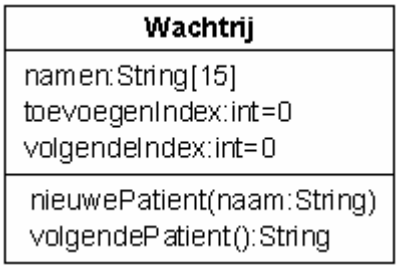
Als de applicatiecode naast toegang tot de twee methods ook toegang heeft tot de array (*namen*), kan je de code niet meer optimaliseren. Misschien gaat sommige applicatiecode er van uit dat het eerste element van de array altijd de volgende patiënt bevat die naar de dokter mag gaan.

Volgende optimalisatie maakt de twee routines sneller, maar de volgende patiënt die naar de dokter mag gaan, bevindt zich nog zelden in het eerste element van de array.

Je voegt aan de class een attribuut toe, waarin je bijhoudt op welk volgnummer de volgende aangekomen patiënt in de array mag geplaatst worden.

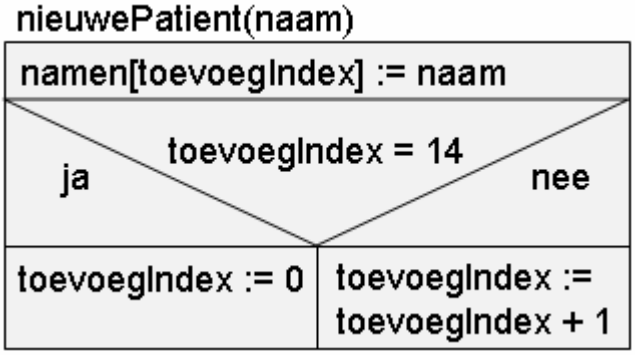
Je voegt aan de class een tweede attribuut toe, waarin je bijhoudt op welk volgnummer de volgende patiënt uit de array mag gelezen worden die van de wachtzaal naar de dokter mag.

In UML ziet de tweede versie van de class er als volgt uit:

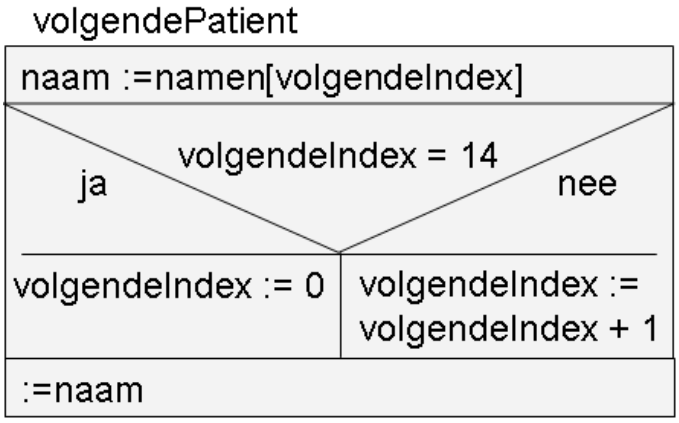


De attributen toevoegenIndex en volgendeIndex krijgen als beginwaarde de waarde nul. In UML wordt dit uitgedrukt met =0 op het einde van het attribuut.

De nieuwe versie van de method nieuwePatient heeft volgende snellere code:



De nieuwe versie van de method volgendePatient heeft ook snellere code:



Je kunt de class echter maar met gerust hart op deze manier optimaliseren als je zeker bent dat enkel deze twee methods de array kunnen benaderen en geen andere code. Gelukkig kan dit bij object oriëntatie. Dit principe heet *data hiding*: je verbergt de attributen voor de code buiten de class.

De code buiten de class ziet enkel de methods van de class.

7.3 Public - Private

De members van een class die zichtbaar zijn voor de code buiten de class heten de public members.

De members van een class die enkel zichtbaar zijn voor de code binnen de class heten de private members.

In UML kan je de public members van de private members onderscheiden:

- public members worden voorafgegaan door een plus teken.
- private members worden voorafgegaan door een min teken.

In de class Spaarrekening maak je van de attributen private members en van de methods public members:

Spaarrekening
-rekeningNummer:String -saldo:double -intrest:double
+storten(bedrag:double):void +afhalen(bedrag:double):void +overschrijven(rekeningNummer:String,bedrag:double,datum:Date) +geefSaldo():double

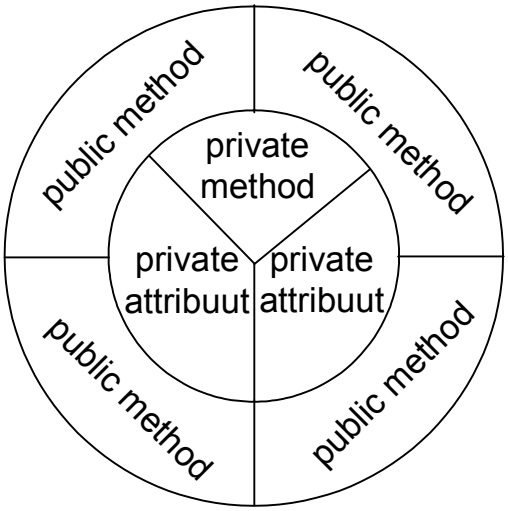
Ook in de class Wachtrij maak je van de attributen private members en van de methods public members:

Wachtrij
-namen:String[15] -toevoegenIndex:int=0 -volgendeIndex:int=0
+nieuwePatient(naam:String) +volgendePatient():String

Opmerking: sommige methods zijn enkel hulpmethods die opgeroepen worden door andere methods van de class, maar die niet moeten opgeroepen worden door programmeurs buiten de class. Ook zulke hulpmethods kan je verbergen voor de code buiten de class.

De public methods zijn toegankelijk voor code buiten de class, de private attributen en private methods niet.

Een object wordt dan ook regelmatig voorgesteld als een bol, waarbij de buitenwereld enkel toegang heeft tot de buitenste schil:



8 MESSAGE PASSING

8.1 Algemeen

Message passing is het berichtenverkeer tussen objecten. Een applicatie bestaat uit veel objecten. Deze objecten sturen berichten naar elkaar. Als object X een bericht wil sturen naar object Y, roept object X een method op van het object Y.

Object Y onderneemt actie op het binnengekomen bericht: Object Y voert de method uit die object X opgeroepen heeft.

Object Y kan optioneel een antwoord geven aan object X. Dit is het geval als de method een returnwaarde bevat. Het antwoord van het object Y aan het object X is de returnwaarde van de method.

Een boodschap is een verzoek van een object X (de zender) aan object Y (de ontvanger) om een method uit te voeren.

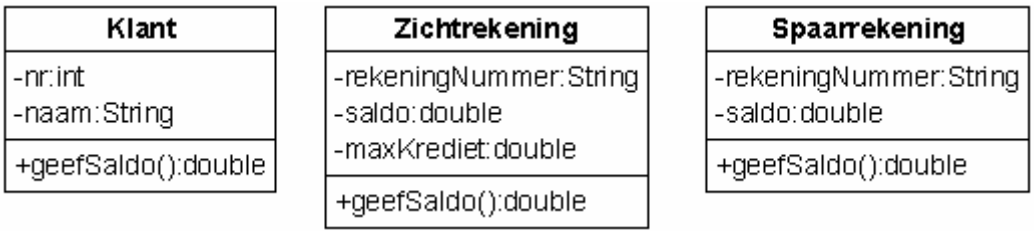
Bij een bank toepassing heb je een class Klant. Deze class bevat een method geefSaldo() die het bedrag berekent die een klant bij de bank geplaatst heeft.

Deze method kan op zich het saldo niet berekenen. Hij moet het saldo opvragen van de zichtrekening van de klant. Daarna moet hij het saldo opvragen van de spaarrekening van de klant. Daarna telt hij deze twee saldo's bij elkaar op. Deze som is het saldo van de klant.

Om het saldo te bepalen van de zichtrekening van de klant roept de method geefSaldo() van een Klant object de method geefSaldo() van een Zichtrekening object op. Er gaat dus een bericht van een Klant object naar een Zichtrekening object. Het Zichtrekening object berekent het saldo van de zichtrekening in zijn method geefSaldo() en geeft een antwoord terug aan het Klant object. Dit antwoord bevat het saldo van de zichtrekening.

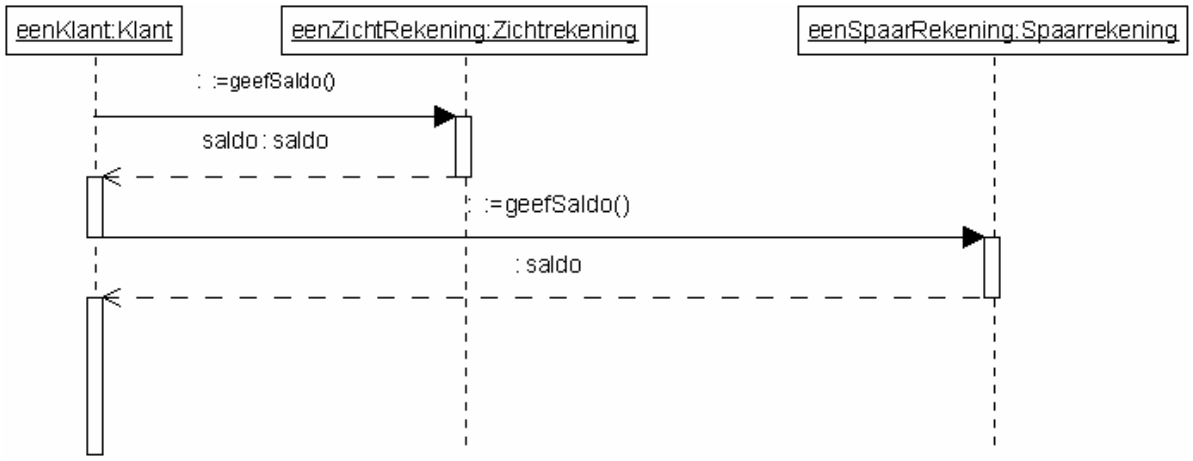
Om het saldo te bepalen van de spaarrekening van de klant roept de method geefSaldo() van een Klant object de method geefSaldo() van een Spaarrekening object op. Er gaat dus een bericht van een Klant object naar een Spaarrekening object. Het Spaarrekening object berekent het saldo van de spaarrekening in zijn method geefSaldo() en geeft een antwoord terug aan het Klant object. Dit antwoord bevat het saldo van de spaarrekening.

De classes die berichten naar elkaar sturen kunnen er in UML zo uitzien:



Een bericht tussen twee objecten wordt in UML voorgesteld als een pijl die gaat van het object die het bericht verstuurt naar het object dat het bericht ontvangt. Een antwoord op het bericht wordt voorgesteld als een pijl die de andere richting uitgaat. De lijn van de antwoordpijl is een onderbroken lijn.

Het berichtenverkeer tussen een Klant object, een Zichtrekening object en een Spaarrekening object kan er in UML als volgt uitzien:



Bovenaan zie je drie rechthoeken die objecten voorstellen:

- een object met de naam eenKlant, met als type de class Klant.
- een object met de naam eenZichtrekening, met als type de class Zichtrekening.
- een object met de naam eenSpaarrekening, met als type de class Spaarrekening.

De bovenste pijl is een bericht van het object eenKlant aan het object eenZichtrekening. Bij dit bericht roept het object eenKlant de method geefSaldo() op van het object eenZichtrekening.

De tweede pijl (met de onderbroken lijn) is het antwoord op het eerste bericht. Dit antwoord bevat het berekende saldo.

De derde pijl is een bericht van het object eenKlant aan het object eenSpaarrekening. Bij dit bericht roept het object eenKlant de method geefSaldo() op van het object eenSpaarrekening.

De vierde pijl (met de onderbroken lijn) is het antwoord op het tweede bericht. Dit antwoord bevat het berekende saldo.

9 ASSOCIATIES TUSSEN CLASSES

9.1 Algemeen

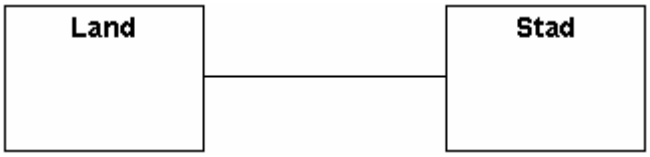
Associaties tussen classes beschrijven verbanden tussen classes.

Gezien er in de echte wereld verbanden bestaan tussen begrippen, is het belangrijk dat we deze verbanden in OOP ook kunnen uitdrukken als associaties tussen classes.

Associaties beschrijven structurele verbanden. Structureel betekent dat een object van de ene class gedurende ruime tijd verbonden is met een object van een andere class. Er is bijvoorbeeld een associatie tussen een klant en een auto als een klant een auto koopt in een garage. Deze associatie bestaat gedurende ruime tijd. Er is geen associatie tussen een klant en een tijdschrift als een klant een tijdschrift koopt in een krantenwinkel.

In de echte wereld is er bijvoorbeeld een verband tussen het begrip land en stad: één land bevat één of meerdere steden. In OOP stel je de begrippen land en stad voor met de classes Land en Stad en beschrijf je het verband tussen deze twee classes als een associatie.

In UML wordt de associatie tussen twee classes voorgesteld als een lijn tussen deze classes. Voorbeeld met de classes Land en Stad:



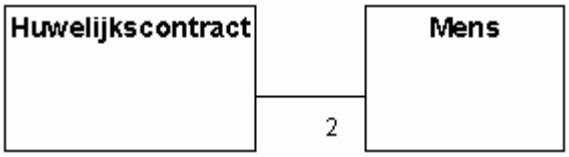
9.2 Multipliciteit

Als je een associatie beschrijft tussen twee classes is meestal ook de multipliciteit belangrijk: hoeveel objecten van de ene class behoren bij één object van de andere class.

9.2.1 Multipliciteit als vast getal

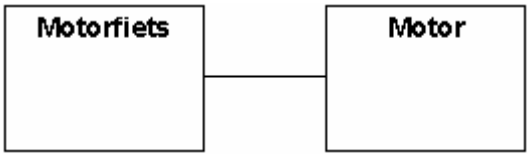
Het aantal objecten van de ene class dat hoort bij één object van de andere class kan een vast aantal zijn. Voorbeeld: twee mensen tekenen één huwelijkscontract:

In UML zie je bij de associatielijn tussen de twee classes een getal dat de multipliciteit aangeeft. In het voorbeeld staat het getal 2 bij de associatielijn aan de kant van de class Mens. Aan de kant van de class Huwelijkscontract staat geen getal. Dan wordt de standaard multipliciteit 1 verondersteld.



9.2.2 Één op één associatie

Bij één object van de ene class hoort één object van de andere class. Voorbeeld: één motorfiets heeft één motor:



Gezien aan beide kanten van de associatie geen getal staat, wordt de standaardwaarde (1) verondersteld.

9.2.3 Associatie met maximaal één object

Bij één object van de ene class hoort maximaal één object van de andere class. Dit wil dus zeggen dat bij één object van de éne class 0 of 1 objecten van de andere class horen.

Voorbeeld: één mens kan maximaal één Belgisch rijbewijs hebben. Dit wil zeggen: het kan zijn dat één mens geen Belgisch rijbewijs heeft, het kan zijn dat één mens één Belgisch rijbewijs heeft, maar het kan niet zijn dat één mens meerdere Belgische rijbewijzen heeft.

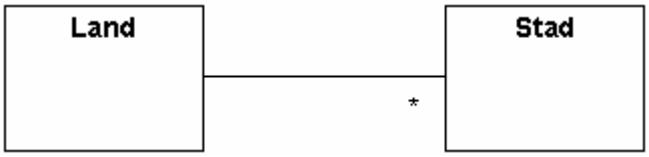
In UML wordt "maximaal één" voorgesteld als 0..1:

9.2.4 Associatie met een variabel aantal objecten.

Bij één object van de ene class horen x aantal objecten van de andere class, waarbij x ligt tussen een minimum en een onbepaald maximum.

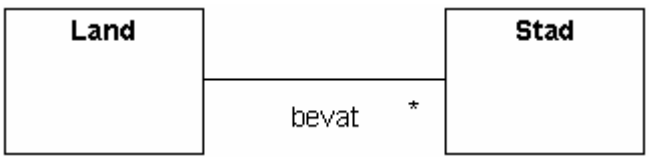
Voorbeeld: één land bevat nul of meerdere steden. Je kunt moeilijk een getal plakken op het aantal steden per land.

Een variabel aantal wordt in UML voorgesteld met een sterretje:



9.3 Benoemde associatie

Om de associatie te verduidelijken, kan je de associatie een naam geven in je analyse. Je schrijft de naam van de associatie bij de associatielijn.



9.4 Rol

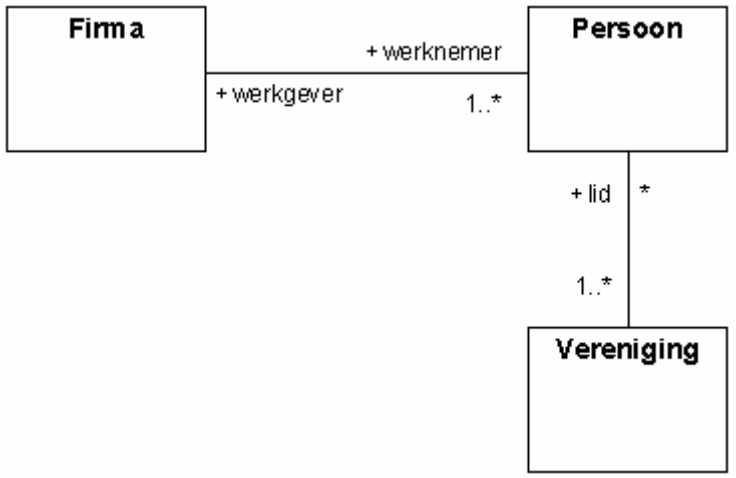
Een object kan in verschillende associaties verschillende rollen spelen.

Er is een associatie tussen een persoon en een firma. Een persoon werkt bij een firma. De persoon speelt bij die associatie de rol van werknemer. De firma speelt bij die associatie de rol van werkgever.

Er is een associatie tussen een persoon en een vereniging. Een persoon is lid van de vereniging. De persoon speelt bij die associatie de rol van lid.

Dezelfde persoon speelt dus één keer de rol van werknemer en één keer de rol van lid.

In UML schrijf je de rol bij de class, bij het vertrekpunt van de associatielijn waarin objecten van die class de rol spelen:

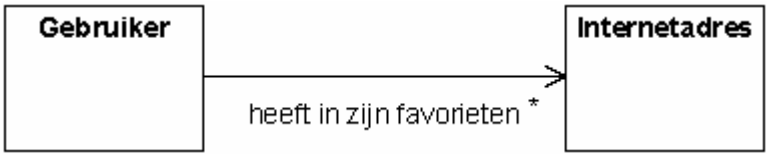


9.5 Associaties met richting

Bij een gewone associatie kennen beide objecten aan de einden van de associatie elkaar. Een land object weet welke stad objecten er bij zich horen én een stad object weet bij welk land object het hoort.

Soms is deze kennis niet wederzijds. Een gebruiker heeft bepaalde internetadressen als favoriete adressen. Een gebruiker weet welke internetadressen hij als favoriete adressen heeft. De internetadressen weten echter niet welke gebruikers hen als favoriet adres hebben.

Als een associatie enkel in één richting kennis bevat over welk object aan de andere kant van de associatie bij een object hoort, geef je dit in UML aan door de associatie een pijl te geven die de richting van de kennis aangeeft:



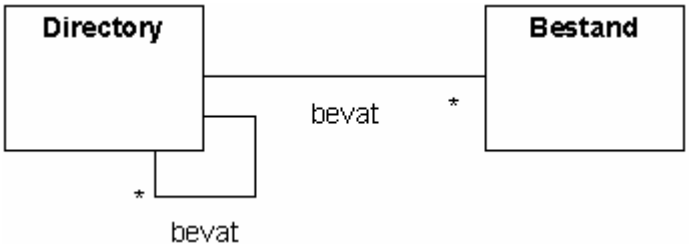
9.6 Reflexieve associaties

Een reflexieve associatie is een associatie van een object van een class naar een object van dezelfde class.

In een bestandssysteem van een computer heb je objecten van de class Directory en objecten van de class Bestand.

Er is een gewone associatie tussen de class Directory en de class Bestand: één directory bevat nul of meerdere bestanden.

Er is een reflexieve associatie tussen de class Directory en de class Directory; één directory kan op zich nul of meerdere directories bevatten:



9.7 Associatieclasses

Een Associatieclass is een class die sterk verbonden is aan de associatie tussen twee andere classes:

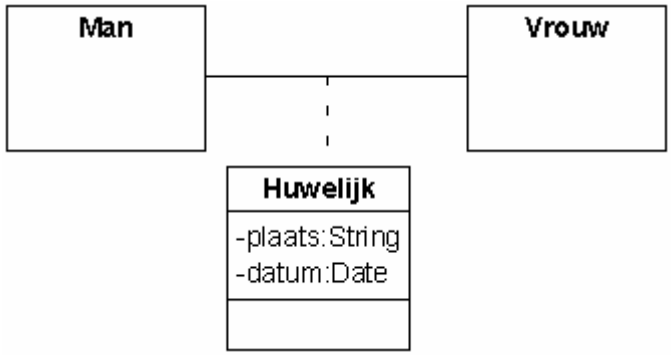
- Als er associatie is tussen twee objecten van de andere classes, is er ook een object van de associatieclass.
- Als de associatie tussen de twee objecten verdwijnt, verdwijnt ook het object van de associatieclass.

Een voorbeeld is de class Huwelijk als associatieclass tussen de classes Man en Vrouw:

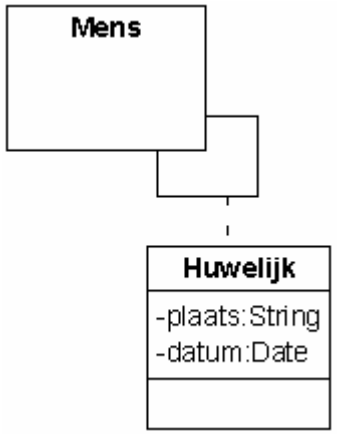
- Als man en vrouw huwen is er een associatie tussen een Man object en een Vrouw object. Aan deze associatie is een Huwelijk object verbonden.
- Als man en vrouw scheiden, verdwijnt de associatie tussen het Man object en het Vrouw object. Dan verdwijnt ook het bijbehorend Huwelijk object.

Meestal gebruik je een associatieclass als je over de associatie zelf attributen of methods wil beschrijven. Bij de Huwelijk class kan je bijvoorbeeld de attributen plaats en datum van het huwelijk bijhouden.

In UML wordt een assocatieclass met zijn associatie verbonden via een gestippelde lijn:



Als je bedenkt dat een man ook met een man kan trouwen, of een vrouw met een vrouw, wijzigt de analyse:



9.8 Aggregatie

Aggregatie is een speciale vorm van associatie.

Aggregatie wijst op samenstelling. Een object van de ene class uit de associatie is samengesteld uit object van de andere class van de associatie.

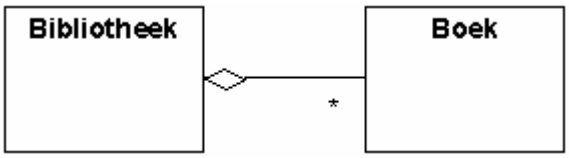
Tussen de classes A en B bestaat aggregatie als je één of meerdere van volgende zinnen kan zeggen:

- Objecten van de class A behoren tot objecten van de class B.
- Een object van de class B bestaat uit objecten van de class A.
- Een object van de class B bevat objecten van de class A
- Objecten van de class A maken deel uit van een object van de class B.

Een voorbeeld is aggregatie tussen de class Boek en de class Bibliotheek:

- Objecten van de class Boek behoren tot de class Bibliotheek.
- Een object van de class Bibliotheek bestaat uit objecten van de class Boek.
- Een object van de class Bibliotheek bevat objecten van de class Boek.
- Objecten van de class Boek maken deel uit van een object van de class Bibliotheek.

In UML zie je dat een associatie een aggregatie is. Dan bevat de associatielijn aan de kant van de class die de samenstelling voorstelt een parallellogram:



9.9 Compositie

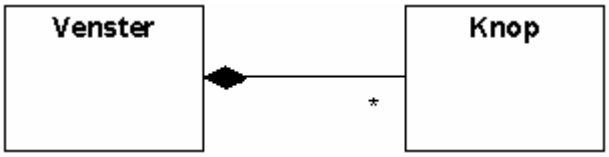
Compositie is een speciale vorm van Aggregatie. De class A heeft een compositie tegenover de class B als:

- Een object van de class B maar tot één object van de class A kan behoren.
- De levensduur van een object van de class B kleiner of gelijk is aan de levensduur van een object van de class A. Als de samenstelling verdwijnt (het object A) verdwijnen ook de onderdelen van de samenstelling (de objecten B).

Een voorbeeld van compositie zijn de knoppen (OK, Cancel, ...) in een dialoogvenster van een computerprogramma:

- Een knop kan maar op één venster staan.
- De levensduur van een knop is kleiner of gelijk is aan de levensduur van zijn venster. Als het venster verdwijnt (het wordt door de gebruiker gesloten), verdwijnen ook de knoppen van dat venster.

In UML zie je dat een associatie een compositie is. Dan bevat de associatielijn aan de kant van de class die de samenstelling voorstelt een zwart gevuld parallellogram:



10 INHERITANCE

10.1 Algemeen

Inheritance (overerving) betekent dat je een class maakt die gebaseerd is op een bestaande class.

De nieuwe class wordt dan een uitbreiding van de bestaande class: de nieuwe class erft (krijgt) de attributen, methoden en associaties van de bestaande class. Daarnaast kan de nieuwe class nog extra attributen, methoden en associaties toevoegen.

De bestaande class wordt superclass of base class genoemd.

De nieuwe class wordt subclass of derived class genoemd.

Een voorbeeld waar je inheritance kan toepassen zijn de classes Zichtrekening en Spaarrekening:

- Beide classes bevatten de attributen saldo en rekeningNummer.
- Beide classes hebben een associatie met de class Klant.
  Deze associatie houdt bij welke klant eigenaar is van de rekening.
- Beide classes bevatten gemeenschappelijke methods: storten, afhalen, overschrijven, geefSaldo.

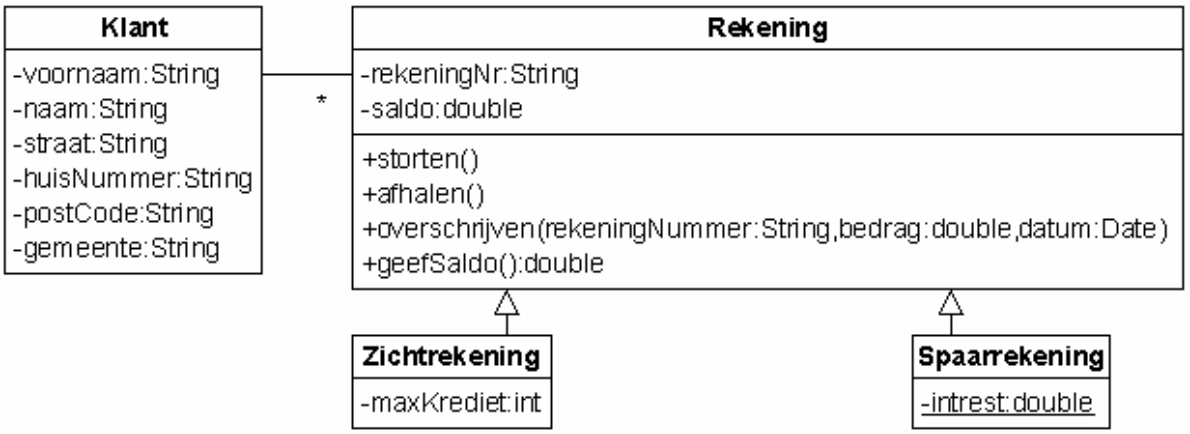
In de plaats van deze gemeenschappelijke attributen, methoden en associaties twee keer uit te werken (in de class Zichtrekening én in de class Spaarrekening), is het interessanter deze gemeenschappelijke attributen, methoden en associaties één keer uit te werken in een base class: Rekening.

Daarna maak je de derived classes Zichtrekening en Spaarrekening die erven van de class Rekening. Deze classes hebben dan automatisch de attributen, methoden en associaties van de base class Rekening ter beschikking.

Attributen die enkel voor een spaarrekening gelden, beschrijf je in de class Spaarrekening. Dit is het attribuut intrest.

Attributen die enkel voor een zichtrekening gelden, beschrijf je in de class Zichtrekening. Dit is het attribuut maxKrediet: hoeveel mag je in het rood gaan op een zichtrekening.

Inheritance wordt in UML voorgesteld als een lijn tussen de derived class en de base class, waarbij de lijn aan de kant van de base class een pijl bevat:



De class Zichtrekening erft van de class Rekening en voegt nog een attribuut toe dat enkel geldt voor een zichtrekening (maar niet voor andere rekeningen): maxKrediet. Dit attribuut bevat per Zichtrekening object het bedrag dat je maximaal mag in het rood gaan.

De class Spaarrekening erft van de class Rekening en voegt nog een attribuut toe dat enkel geldt voor een spaarrekening: intrest. Gezien het intrestpercentage voor alle Spaarrekening objecten gelijk is, is dit een attribuut met class bereik.

Als achteraf de base class (Rekening) uitgebreid wordt met extra attributen, methods of associaties, worden deze automatisch geërfd in de derived classes (Zichtrekening, Spaarrekening).

Voorbeeld: als de base class Rekening uitgebreid wordt met een attribuut creatiedatum (de datum waarop de rekening aangemaakt werd), is dit attribuut ook ter beschikking in de derived classes (Zichtrekening, Spaarrekening).

10.2 Single inheritance – Multiple inheritance

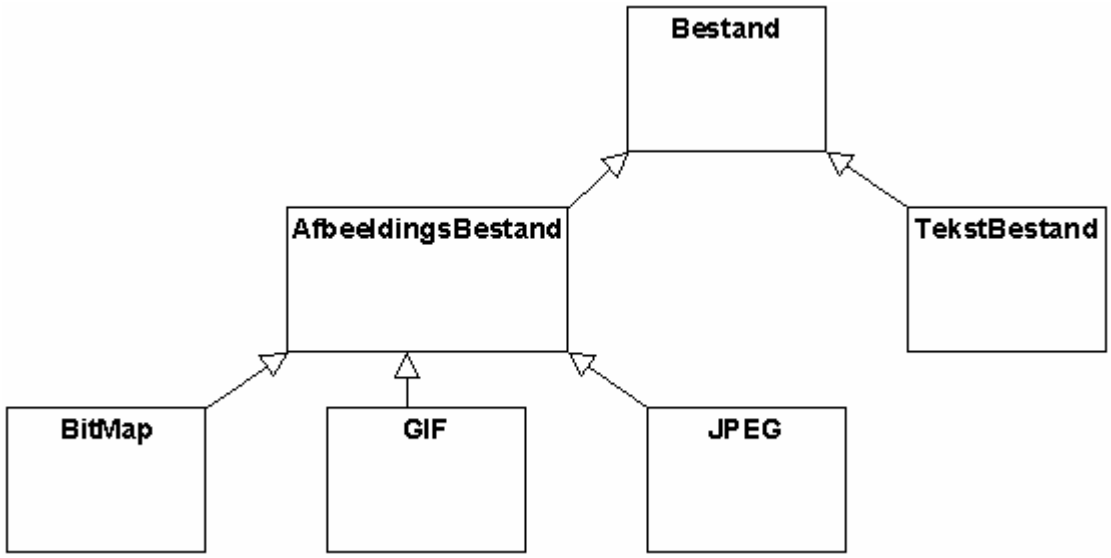
10.2.1 Single inheritance

Bij single inheritance heeft een class maximaal één base class. Omgekeerd kunnen er van één base class meerdere classes afgeleid zijn.

De programmeertalen Java, Visual Basic.net en C# zijn gebaseerd op single inheritance.

De base class van een class kan op zich weer maximaal één base class hebben.

In het volgend voorbeeld vind je single inheritance:



- De class BitMap heeft één base class: AfbeeldingsBestand
- De class AfbeeldingsBestand heeft op zich één base class: Bestand
- De class Bestand heeft geen base class.
- Omgekeerd zijn van de class Bestand wel meerdere classes afgeleid.

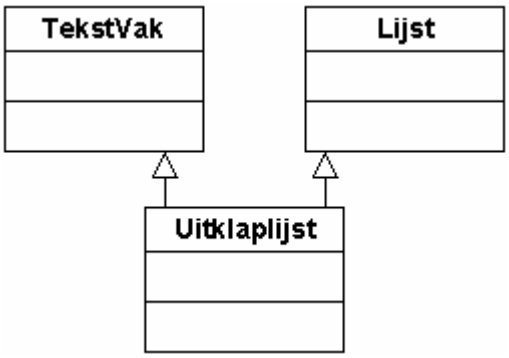
10.2.2 Multiple inheritance

Bij multiple inheritance kan een class meerdere base classes hebben. Omgekeerd kunnen er van één base class ook meerdere classes afgeleid zijn.

De programmeertaal C++ is gebaseerd op multiple inheritance.

Hoewel de programmeertalen Java, Visual Basic.net en C# recenter zijn dan C++, is in deze talen multiple inheritance niet toegepast, omdat de makers van die talen beslisten dat multiple inheritance soms tot moeilijke inheritance structuren tussen classes leiden.

In het volgend voorbeeld vind je multiple inheritance:



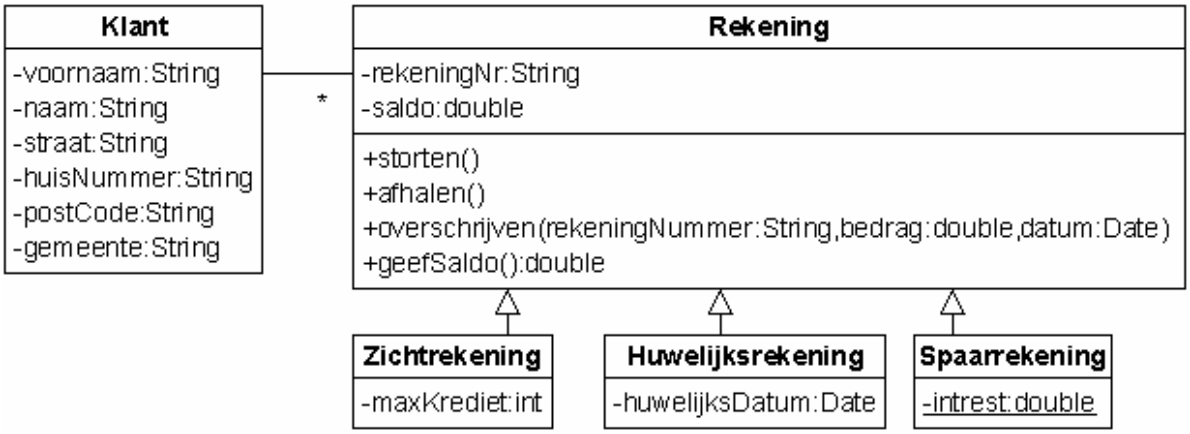
Een uitklaplijst (combobox) van een dialoogvenster heeft een lijstgedeelte (als je het knopje ▾ van de uitklaplijst aanklikt) én een tekstgedeelte. Bij multiple inheritance kan je dat voorstellen als een class uitklaplijst die erft van de class Tekstvak en tegelijk erft van de class Lijst.

Als een taal enkel single inheritance ondersteunt, moet je de uitklaplijst uitwerken met compositie: een uitklaplijst *bevat* een tekstvak én een lijst:

10.3 Method overriding

Method overriding betekent dat een method uit de base class ook voorkomt in de derived class. De bedoeling is dat de method in de derived class andere code bevat dan de method uit de base class. Je doet dit omdat de method in de derived class iets anders moet doen dan dezelfde method in de base class.

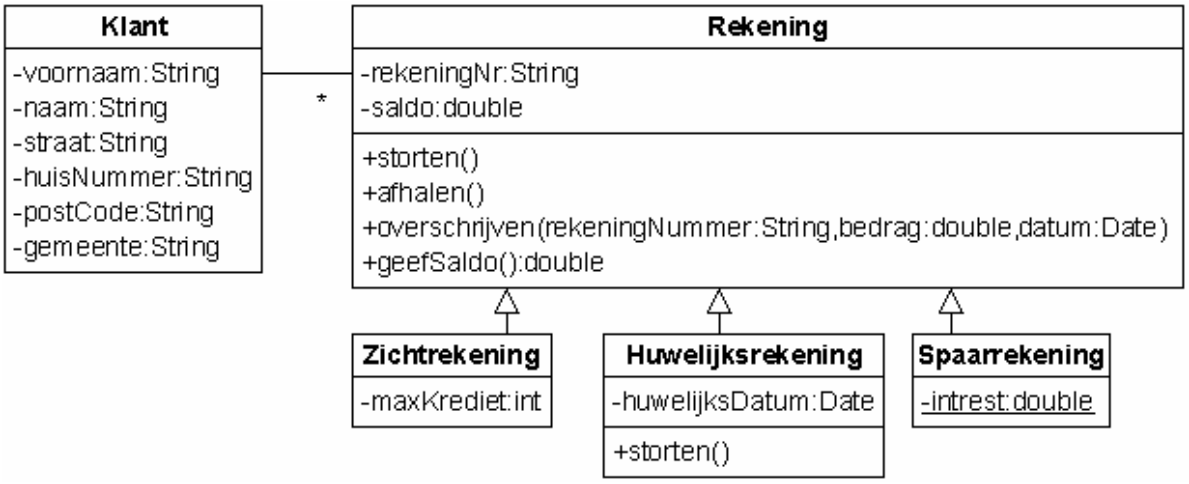
In het voorbeeld van dit hoofdstuk kan je van de class Rekening nog een extra class afleiden: Huwelijksrekening (voorhuwelijkssparen):



Als je op een object van de class Huwelijksrekening de method storten() uitvoert, voer je de method storten() uit zoals beschreven in de base class van Huwelijksrekening: Rekening.

Geld storten op een huwelijksrekening verloopt in de werkelijkheid echter anders dan bij een gewone rekening: je kunt maar een maximaal aantal euro per jaar storten op een huwelijksrekening.

Bij object oriëntatie gebruik je method overriding om dit probleem op te lossen. Je maakt in de derived class Huwelijksrekening ook een method storten(), waarin je code schrijft die controleert dat het gestorte bedrag het maximum stortbaar bedrag niet overschrijdt:



Als je nu op een object van de class Huwelijksrekening de method storten() uitvoert, voer je de method storten() uit zoals beschreven in de class Huwelijksrekening zelf.

10.4 Abstract class

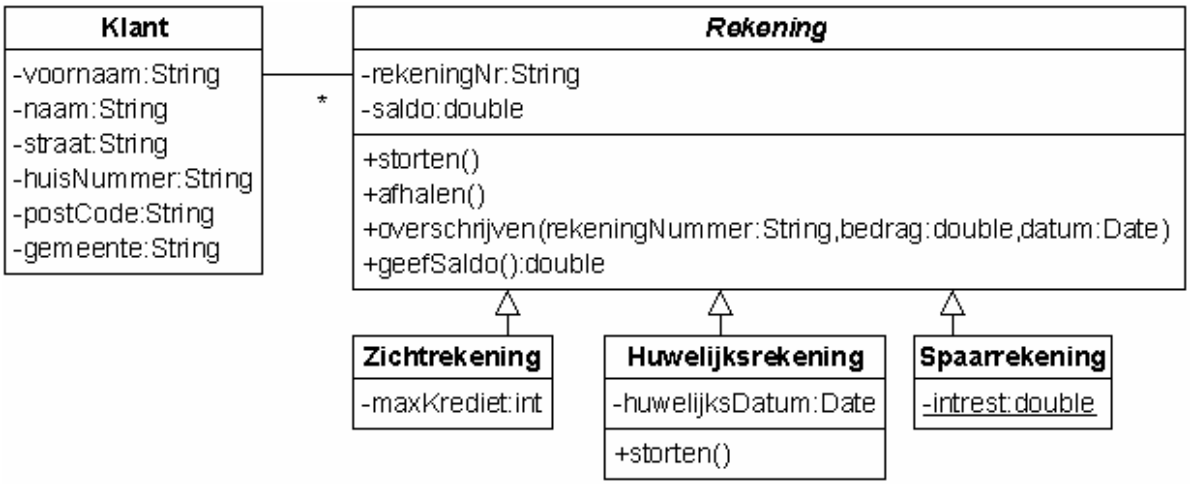
Als een class enkel dient om de gemeenschappelijke attributen, methods en associaties van andere classes te beschrijven, maar er van die class in de echte wereld geen objecten bestaan, wordt deze class een abstract class genoemd.

De class Rekening in ons voorbeeld is een abstract class. Je beschrijft er de gemeenschappelijke attributen, methods en associaties in van de classes Zichtrekening, Spaarrekening en Huwelijksrekening.

In de echte wereld bestaan echter geen rekeningen, enkel zichtrekeningen, spaarrekeningen en huwelijksrekeningen. Het begrip rekening is een abstract begrip om alle mogelijke soorten rekeningen te verzamelen.

Zodra een class als abstract class beschreven is, verhindert de compiler van een object georiënteerde programmeertaal dat je van die class nog objecten maakt.

In UML wordt de naam van een abstract class schuin geschreven:



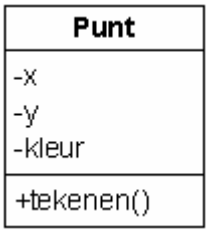
10.5 Abstract method

Een abstract method is een method die geen code heeft in de base class, maar de method krijgt wel code in de derived class(es) via method overriding.

We zullen dit uitleggen aan de hand van een voorbeeld.

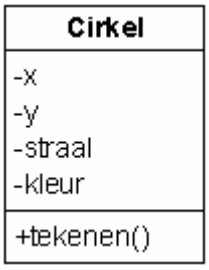
Als je een CADCAM programma op een object georiënteerde manier analyseert, zie je dat in een CADCAM programma verschillende soorten figuren voorkomen: punten, lijnen, rechthoeken, cirkels.

Als eerste stap beschrijf je deze begrippen als classes met attributen:



Een Punt heeft een x coördinaat, een y coördinaat en een kleur. Een Punt bevat code om een punt te tekenen op het CADCAM ontwerp.

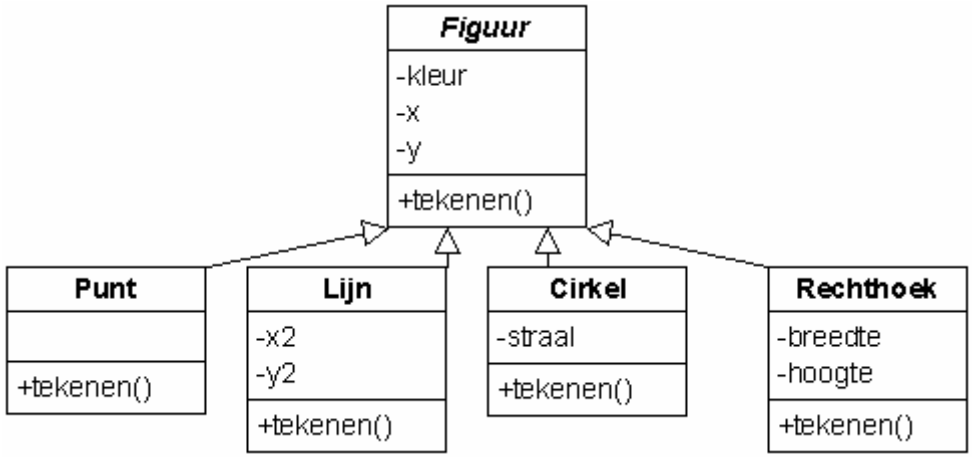
Een Lijn heeft een x coördinaat (x) en een y coördinaat (y) van het linkerbovenpunt waar de lijn begint. En lijn heeft ook een x coördinaat (x2) en een y coördinaat (y2) van het rechteronderpunt waar de lijn eindigt. Een lijn heeft een kleur. Een lijn bevat code om een lijn te tekenen op het CADCAM ontwerp.



Een Cirkel heeft een x coördinaat (x) en een y coördinaat (y) van het middenpunt van de cirkel. En cirkel heeft ook een straal en een kleur. Een cirkel bevat code om een cirkel te tekenen op het CADCAM ontwerp.

Een Rechthoek heeft een x coördinaat (x) en een y coördinaat (y) van het linkerbovenpunt van de rechthoek. Een rechthoek heeft ook een breedte, een hoogte en een kleur. Een rechthoek bevat code om een rechthoek te tekenen op het CADCAM ontwerp.

Als je de classes samen bekijkt, zie je gemeenschappelijke eigenschappen (x, y, kleur) en een gemeenschappelijke method: tekenen(). Je plaatst deze gemeenschappelijke eigenschappen en method in één class: Figuur en je laat de classes Punt, Lijn, Rechthoek en Cirkel erven van de class Figuur. Hierbij besef je ook dat het begrip figuur slechts een abstract begrip is dat dient als verzamelnaam voor de begrippen punt, lijn, rechthoek en cirkel en dus maak je van de class Figuur een abstract class:



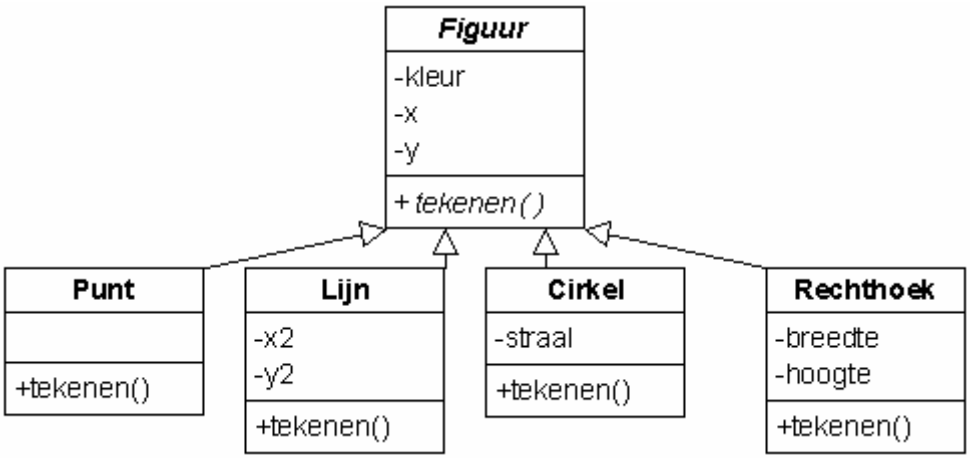
Bemerk dat de method tekenen() toch nog te vinden is in de derived classes Punt, Lijn, Cirkel en Rechthoek. Dit is omdat de code om een punt te tekenen anders is dan de code om een lijn te tekenen. De code om een cirkel te tekenen is ook anders en de code om een rechthoek te tekenen is ook anders.

Uiteindelijk kom je tot het inzicht dat de method iedere keer zodanig verschillend is dat de method in de base class (Figuur) geen code meer bevat.

Dan maak je van de deze method in de base class een abstract method.

Bij een abstract method controleert de compiler of de method in de derived classes wel code krijgt (tenzij de derived class weer abstract zou zijn).

Een abstract method krijgt in UML een schuin lettertype:



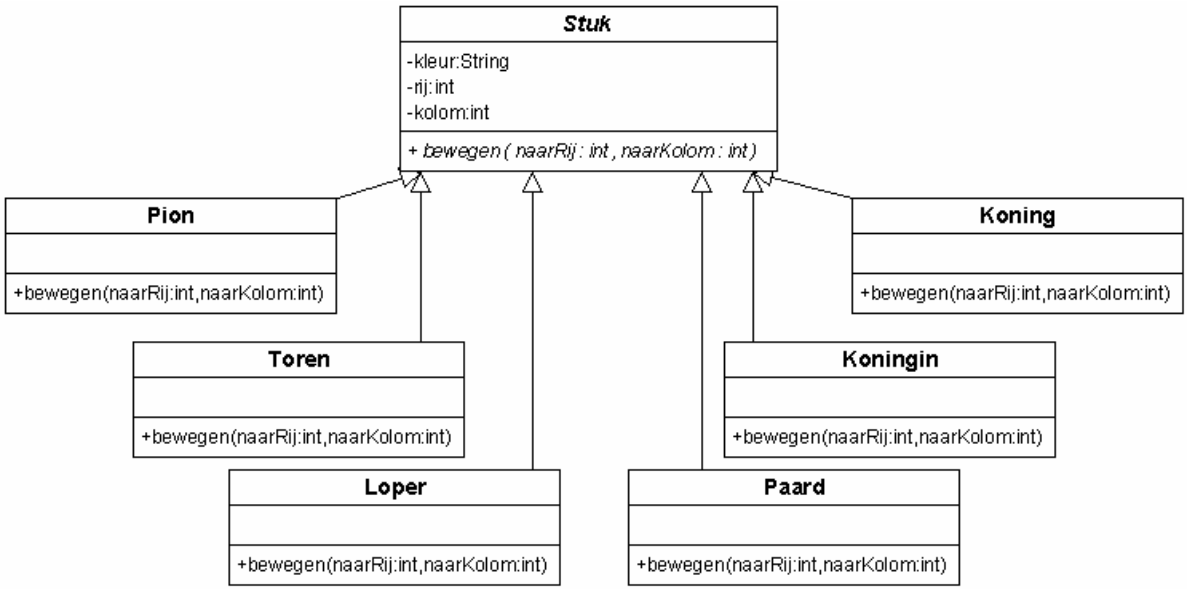
Je kunt je afvragen waarom je de method tekenen nog zou vermelden in de class Figuur. Ten eerste is het zo dat je in de werkelijkheid zegt dat elke figuur kan getekend worden. Dit wordt nu ook duidelijk in je object georiënteerde analyse: in het kader van de class Figuur zie je de method tekenen. Ten tweede is het belangrijk dat de method in de gemeenschappelijke base class van de classes Punt, Lijn, Cirkel en Rechthoek voorkomt, zelfs zonder code, om polymorfisme toe te passen. Polymorfisme komt aan bod in het volgende hoofdstuk.

10.6 Een ander inheritance voorbeeld: het schaakspel.

Bij de automatisering van een schaakspel kan je ook inheritance toepassen.

Er zijn verschillende stukken (pionnen, torens, lopers, paarden, koningen, koninginnen). Alle stukken hebben gemeenschappelijke attributen: kleur, rij op het bord en kolom op het bord. Alle stukken kunnen bewegen, maar elk op hun eigen manier: een loper kan enkel diagonaal bewegen, een toren enkel horizontaal of verticaal.

Uiteindelijk leidt dit tot een abstract base class Stuk, met derived classes Pion, Toren, Loper, Paard, Koning en Koningin:



11 POLYMORFISME

11.1 Algemeen

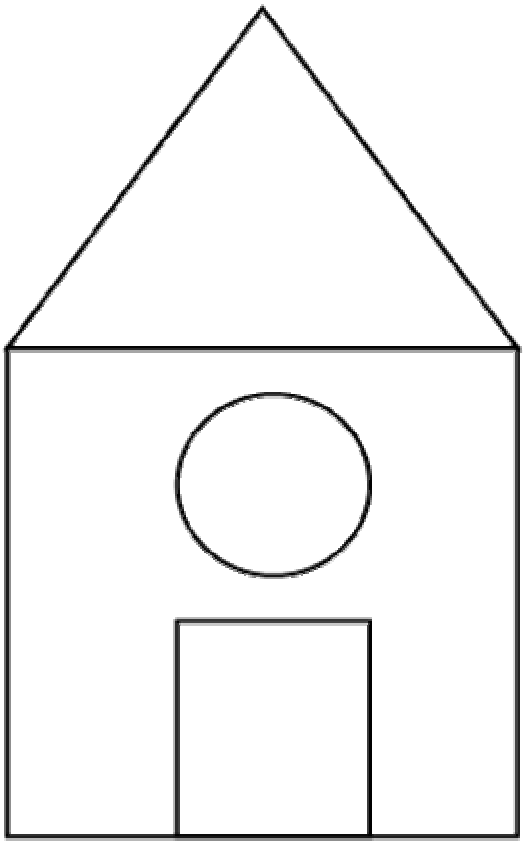
Polymorfisme wil zeggen dat je eenzelfde method vindt in meerdere classes. Bij ieder van deze classes kan de method andere code bevatten en dus een ander resultaat geven bij de uitvoering van de method. Naargelang de class waartoe een object behoort, zal het uitvoeren van de method dus een ander resultaat geven.

In het voorbeeld van het vorige hoofdstuk is de method tekenen een polymorfe method. Je vindt ze terug in meerdere classes (Punt, Lijn, Driehoek, ...). De code van de method tekenen zal in ieder van die class verschillend zijn: de code van de method tekenen in de class Punt toont enkel een punt op het scherm. De code van de method tekenen in de class Driehoek tekent een volledige driehoek op het scherm.

Het resultaat van de method tekenen verschilt dus naargelang je het uitvoert op een Punt object of een Driehoek object.

Polymorfisme is interessant als je de gemeenschappelijke method wil toepassen op een verzameling objecten.

11.2 CADCAM applicatie als voorbeeld van polymorfisme



In het CADCAM voorbeeld bestaat een tekening uit een verzameling figuren.

De CADCAM tekening van het huis hiernaast bestaat uit:

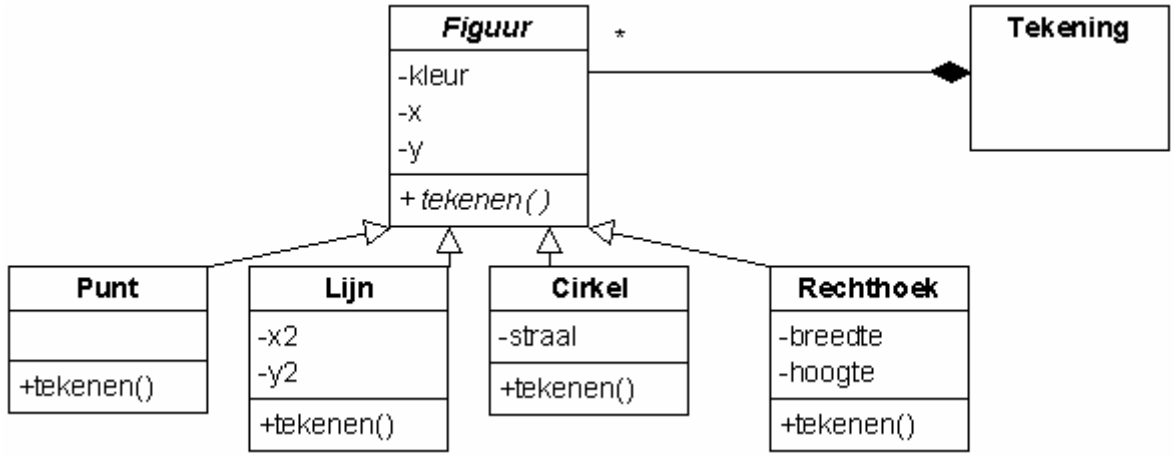
- een driehoek
- een rechthoek
- een cirkel
- een rechthoek.

Om de tekening op het scherm te tonen zal het CADCAM programma de figuren van de tekening één per één doorlopen en iedere figuur vragen de method tekenen uit te voeren.

Het eerste object uit de verzameling is een Driehoek object. Als het CADCAM programma op dit object de method tekenen uitvoert, zie je de driehoek op het scherm.

Het tweede object uit de verzameling is een Rechthoek object. Als het CADCAM programma op dit object de method tekenen uitvoert, zie je de rechthoek op het scherm.

Omdat het concept Tekening in het CADCAM programma een belangrijk concept is, zal het ook een class worden: de class Tekening. Deze class heeft een associatie met de class Figuur: één tekening bevat nul of meerdere figuren. Als de tekening verdwijnt, verdwijnen ook de figuren van de tekening. Tekening is dus een compositie van Figuren:



11.3 Hifi apparatuur als voorbeeld van polymorfisme

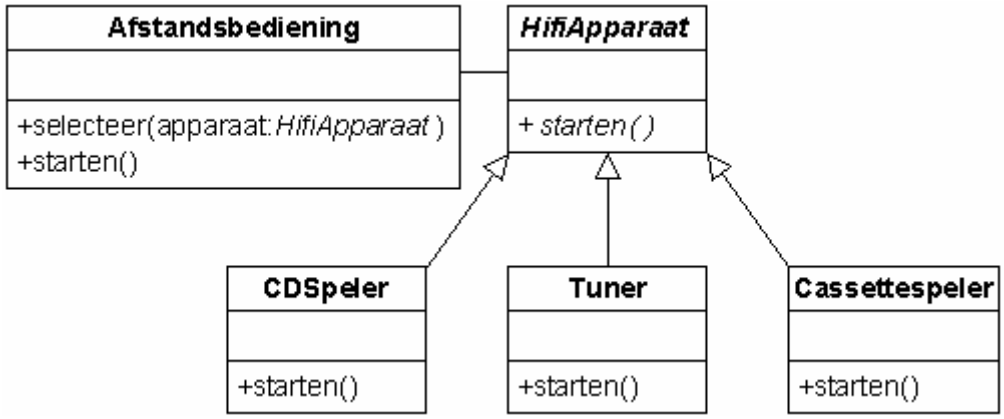
Een tweede voorbeeld van polymorfisme is de method starten van Hifi apparatuur. Er bestaan verschillende soorten Hifi apparatuur: CD speler, Tuner, Cassettespeler, ...

Ieder van die soorten Hifi apparatuur heeft een method starten. Deze method starten is per Hifi apparatuur verschillend uitgewerkt:

- Bij een CD speler speelt de method starten de CD vanaf het eerste liedje.
- Bij Tuner laat de method starten de favoriete radiozender horen.
- Bij Cassettespeler speelt de method starten de cassette verder af waar ze de laatste keer gestopt werd.

Naargelang je met je afstandsbediening de CD speler, de Tuner of de Cassettespeler kiest en daarna de method starten uitvoert, zal je ofwel het eerste liedje van een CD horen, of je favoriete radiozender, of je cassette waar je ze laatst had stopgezet.

In UML kunnen de classes van dit voorbeeld er als volgt uitzien:



De concepten Cd-speler, Tuner en Cassettespeler worden voorgesteld als de classes CDSpeler, Tuner en Cassettespeler. Ieder van deze classes heeft een method starten. De code van deze method is wel in de drie classes verschillend.

De concepten Cd-speler, Tuner en Cassettespeler vallen allen onder de abstracte term Hifi apparaat. In UML laat je deze classes dan erven van een abstract class HifiApparaat. Ieder Hifi apparaat kan je starten, maar je kunt niet concretiseren wat starten van een Hifi apparaat juist betekent (je kan dit pas concretiseren in de afgeleiden: CDSpeler, Tuner, Cassettespeler). Daarom is de method starten in de class HifiApparaat een abstracte method.

Met een afstandsbediening kan je om het even welk Hifi apparaat bedienen. Daarom is er een associatie tussen de class Afstandsbediening en de class HifiApparaat.

Op de afstandsbediening selecteer je welke Hifi apparaat je met de afstandsbediening wil bedienen. In het UML diagram zie je dat daarvoor een method selecteren voorzien is met als parameter het te bedienen Hifi apparaat.

Op de afstandsbediening heb je een knop starten. In het UML diagram vind je een bijbehorende method starten. De code van deze method zal de method starten aanspreken van het Hifi apparaat dat op de afstandsbediening geselecteerd werd.

Als op de afstandsbediening een Cd-speler geselecteerd werd zal je bij de method starten het eerste liedje van een CD horen. Als op de afstandsbediening een radio geselecteerd werd zal je je favoriete radiostation horen, ... Dit is polymorfisme: je voert telkens dezelfde method uit (starten), maar naargelang het object waarop je de method uitvoert, verkrijg je een ander resultaat.

12 INTERFACE

12.1 Algemeen

Een interface is een abstracte class die enkel abstracte methods bevat.

Een interface beschrijft *welke* methods een abstract concept uit de werkelijkheid kan uitvoeren, maar beschrijft niet *hoe* deze methods uitgevoerd worden.

Als een class erft van een interface en de methodes beschreven in de interface code geeft, zegt men dat die class de interface implementeert.

De abstracte class HifiApparaat in het vorige voorbeeld is een interface: ze bevat enkel methods (de method starten) en deze methods zijn abstract.

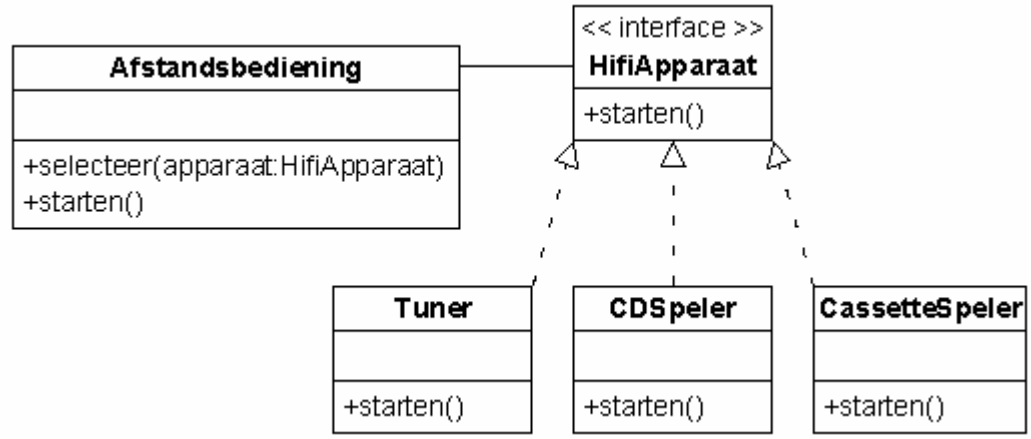
De classes CDSpeler, Tuner en Cassettespeler implementeren de interface HifiApparaat: ze erven van de interface en bevatten de method(s) van de interface: de method starten. Zij geven deze method(s) ook code.

In UML teken je een interface als een rechthoek met twee compartimenten.

- Het bovenste compartiment bevat het woord interface tussen << en >>. Daaronder staat de naam van de interface in het vetjes.
- Het tweede compartiment bevat de methods van de interface.

Er vertrekt een pijl van een class die de interface implementeert naar de interface zelf. Deze pijl is een onderbroken lijn.

Het vorige voorbeeld kan je dus als volgt tekenen:



12.2 Relaties tussen interfaces en classes.

- Een interface kan erven van één of meerdere andere interfaces.
  De afgeleide interface erft dan de methods beschreven in de basisinterface en kan extra methods definiëren.
- Een class kan één of meerdere interfaces implementeren.
  De class moet dan alle methods beschreven in alle geïmplementeerde interfaces bevatten en code geven.

13 SAMENVATTING

13.1 Algemeen

Object georiënteerd programmeren heeft volgens Alan Kay (één van de OOP pioniers) volgende eigenschappen:

13.1.1 Alles is een object.

Je kunt een object aanzien als een variabele die zowel data kan bevatten, maar waar je ook aan kan vragen bepaalde handelingen op zichzelf uit te voeren.

Je kunt ieder concept uit het probleem dat je probeert op te lossen (cd-speler, schaakspel, bankrekening, ...) voorstellen als een object in je programma.

13.1.2 Een programma is een verzameling objecten die elkaar vertellen wat te doen door elkaar berichten te sturen.

Om een object te vragen een bepaalde handeling uit te voeren stuur je een bericht naar dat object. Meer concreet komt een bericht naar een object sturen neer op het uitvoeren van een functieoproep op dat bericht.

13.1.3 Ieder object heeft zijn eigen geheugen dat op zich bestaat uit objecten.

Anders gezegd: je kunt een nieuw soort object maken door bestaande objecten in een nieuw geheel te verpakken. Zo kan je complexe programma's maken en tegelijk de complexiteit verbergen achter de eenvoud van objecten.

13.1.4 Ieder object heeft een type.

Ieder object is een instantie van een class. Hierbij is class een synoniem van type. De belangrijkste eigenschap van een class is: welke boodschappen kan ik er naar sturen ?

13.1.5 Alle objecten van eenzelfde type kunnen dezelfde boodschappen ontvangen.

Omdat een object van de class Cirkel ook een object is van de class Figuur (de class Cirkel erft van de class Figuur), kan een Cirkel object boodschappen aanvaarden die voor Figuur bedoeld zijn. Voorbeelden van zo'n boodschappen (methods): tekenen(), uitvegen(), verplaatsen(), kleurWijzigen(), ...

Dit betekent dat je code kan schrijven die Figuur objecten aanspreekt en dat deze code ook werkt op alle objecten van classes die afgeleid zijn van Figuur (Cirkel, Rechthoek, Driehoek, ...).

Deze vervangbaarheid is een belangrijk concept in OOP.

14 COLOFON

Sectorverantwoordelijke:

Cursusverantwoordelijke:

Medewerkers: Hans Desmet

Versie: 4 april 2006

Nummer dotatielijst: