Object Georiënteerd Programmeren

Object Georiënteerd Programmeren : kernpunten & voorbeeld

In dit document worden zo kort en duidelijk mogelijk de kernpunten van object georiënteerd programmeren uitgelegd. Het betoog wordt geïllustreerd met een voorbeeld dat geënt is op Object Georiënteerd Programmeren in PHP. In principe is de programmeertaal echter onbelangrijk en gaat het om de concepten achter het programmeren. Dit document sluit aan bij hoofdstuk 9 van Informatica-Actief en vervangt deze paragrafen gedeeltelijk.

"Normaal:" Procedureel Programmeren

Object Georiënteerd Programmeren (vanaf nu: OOP) wordt vooral vaak afgezet tegen het traditionelere Procedureel Programmeren. Om OOP beter te begrijpen is het daarom goed even de belangrijkste kenmerken van Procedureel Programmeren langs te lopen.

In de tijd van 'regelnummer' basic was er sprake van **imperatief programmeren**. Dat wil zeggen: de flow van het programma werd volledig bepaald door de programmeur. De rol van de gebruiker beperkt zich dan tot het op de juiste momenten invullen van door het programma benodigde gegevens. Je vult bijvoorbeeld op verzoek van het programma je leeftijd, je inkomen, de gewenste hypotheek en meer van die dingen in en het programma rekent uit hoe hoog je maandlasten zullen zijn.
Procedureel Programmeren kent al veel meer vrijheidsgraden voor de gebruiker. Het te automatiseren probleem, bijvoorbeeld tekstverwerken, wordt onderverdeeld in een aantal 'functies' (denk aan "printen", "tekst bewerken", "zoeken" enzovoorts). De interface van het programma stelt de gebruiker in staat om die functies in de applicatie te gebruiken, soms zelfs op meer dan één manier (muis of toetsenbord bijvoorbeeld). Programmeren komt in dit geval neer op het onderkennen van de juiste functies die het programma aan moet kunnen en het schrijven van de juiste procedures om die functies te kunnen realiseren. Algoritmiseren, dat wil zeggen het bepalen van de juiste stapsgewijze uitvoering van basale bewerkingen in complexe procedures, is daarbij een belangrijke bezigheid.

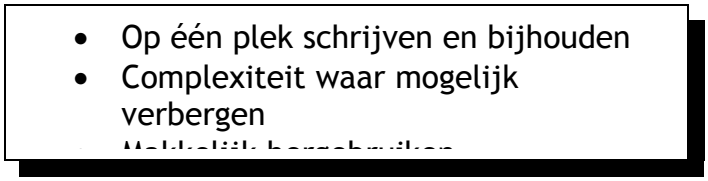
In bijna alle traditionele programmeertalen kun je met Procedureel Programmeren terecht (C, Visual Basic (en de rest), PHP om er maar een paar te noemen).

Voorbeeld in PHP:

Een procedure **db_contact( )** die het contact legt tussen een applicatie en een database.
Een procedure **logincheck( )** die controleert of iemand ingelogd is.
Enzovoorts.

Per functie leg je als programmeur vast:
- welke parameters je aan de functie meegeeft
- welke retourwaarden de functie teruggeeft

Op het moment dat applicaties groter worden en het behouden van overzicht niet meer vanzelf gaat maar afgedwongen moet worden wordt het toepassen van functiebibliotheken die via include's in programma code ingebonden kunnen worden heel belangrijk. Veel gebruikte functies worden dan in zo'n bibliotheek ondergebracht. Hun code komt op maar één plaats voor en als er aanpassingen in nodig zijn hoeft (meestal) alleen die bibliotheek aangepast te worden. Dit levert in rust en overzicht op in de programmacode. Daarnaast kunnen dit soort functiebibliotheken ook 'meegenomen' worden naar andere applicaties (hergebruik van code).



Drie belangrijke eigenschappen van code in complexe applicaties

Met behulp van Procedureel Programmeren kun je op deze manier heel goed professionele applicaties maken.

OOP wordt vaak gezien als het 'professionelere' alternatief voor Procedureel Programmeren. Wat heeft OOP dan wat Procedureel Programmeren niet heeft? Waarin wijkt de aanpak van OOP af van het zojuist geschetste beeld?

Object Georiënteerd Programmeren

Bij OOP staan niet procedures, maar objecten centraal. Een object is 'iets' dat je nodig hebt bij het oplossen van een (automatiserings)probleem. Als je ergens naar toe wilt heb je een auto nodig. Als je iets op wilt schrijven heb je een pen en papier nodig (twee objecten). Als je bijvoorbeeld een inlogsysteem moet maken krijg je met het object **User** te maken, als je een website management systeem moet maken krijg je met de objecten **Pagina**, **Nieuwsitem**, **Forum** en **Database** te maken.

Van objecten kun je meestal twee dingen bespreken: hun eigenschappen (hoe ze er uit zien bijvoorbeeld) en hun methoden (wat ze kunnen of wat er mee gedaan kan worden). Een eigenschap van een User is bijvoorbeeld zijn inlognaam en zijn wachtwoord. Een methode die bij User hoort is het checken of opgegeven inlognaam en wachtwoord kloppen, of het veranderen van persoonlijke gegevens, of het versturen van een emailtje naar een user. Een eigenschap van een Forum is de titel van het forum of de tekst van een mailtje. Methoden van een forum zijn het 'posten' van een forumbericht, het aanmaken van een reactie op een ander forumbericht, het doorzoeken van een reeks forumberichten op een bepaalde zoektekst en het op het scherm laten zien van het forum.

Veel methoden van objecten zijn er op uit om iets te doen met de eigenschappen van de objecten (denk aan het controleren van inloggegevens of het veranderen van een userprofiel).

Methoden worden aan het werk gezet via events, dat wil zeggen acties van de gebruiker of het systeem. Als iemand klikt op de knop **Forum** gaan de methodes aan de slag die het forum op het scherm zetten. Als iemand klikt op **Inloggen** gaat de inlogcheck van een gebruikersaccount aan de slag, enzovoorts.

Op deze manier kun je bijna iedere applicatie beschrijven als een geheel van Objecten, ieder met zijn eigenschappen en methoden, die via events aan elkaar verknoopt zijn en het informatieprobleem oplossen.

De kunst van OOP is dan om, uitgaande van het informatieprobleem, eerst goed na te denken over de noodzakelijke objecten en de interne opbouw van die objecten (eigenschappen en methoden) en daarna over de manier waarop die via events aan elkaar verknoopt worden.

Kenmerkend voor deze benadering is dus dat je éérst, onafhankelijk van de uiteindelijk te bouwen applicatie, nadenkt over de noodzakelijke objecten en hun interne opbouw!

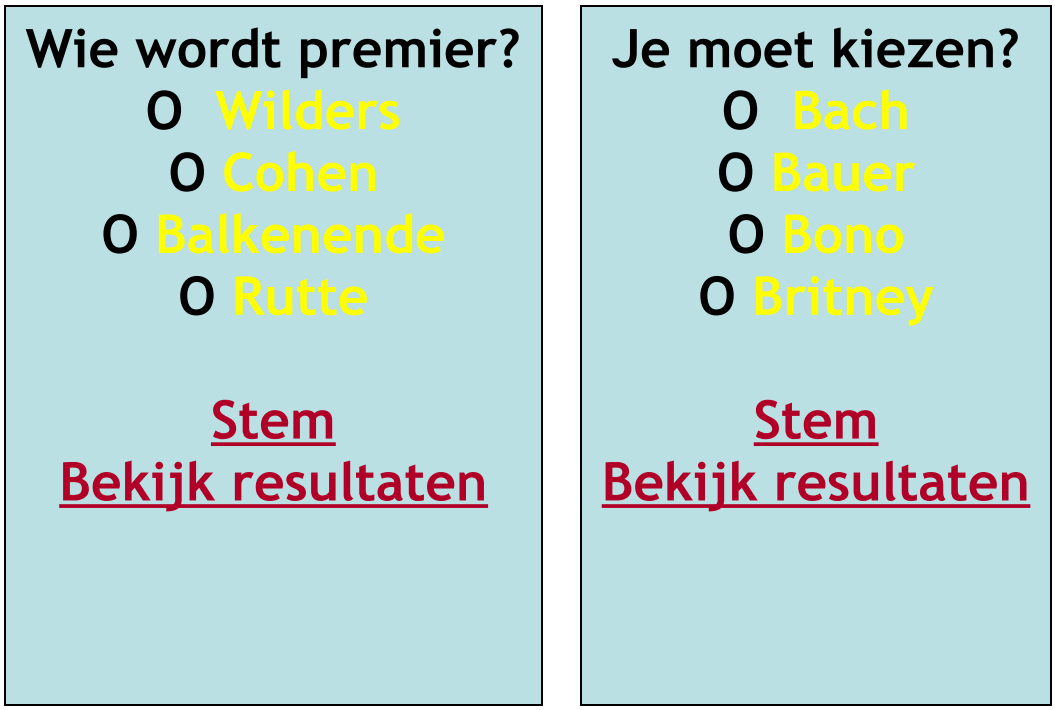
In de wereld van de OOP concepten kom je veel tegen dat gericht is op het zo *makkelijk* mogelijk realiseren van *goede* objectdefinities. Zo zullen we verderop begrippen tegenkomen als 'klasse', 'overerving' en 'interfaces'. Ook zijn er voor OOP complete formele talen verzonnen (UML met name) waarmee de structuur van objecten, los van de programmeertaal, beschreven kan worden. Soms is het dan zelfs mogelijk om na het goed definiëren van een objectstructuur met een druk op de knop alle formele programmeerdefinities aan te maken, denk aan declaraties van eigenschappen en methoden van objecten, zodat je onmiddellijk aan de slag kunt met de 'echte' code.

Wat dan meestal wel weer opvalt is dat binnen de 'echte' code voor het overgrote deel gebruik gemaakt zal gaan worden van dezelfde opdrachten, algoritmen en technieken die je ook bij Procedureel Programmeren hard nodig had. De tegenstelling tussen OOP en PP is zeer beslist niet totaal! Je hebt PP nodig bij OOP.

Het voorbeeld: een eenvoudige meningspeiling (poll) in PHP

We willen graag voor een website een programma schrijven waarmee we snel en gemakkelijk een meningspeiling kunnen opzetten. We willen er een echt systeem van maken, met andere woorden: het systeem moet vaker gebruikt kunnen worden met verschillende meningspeilingen.

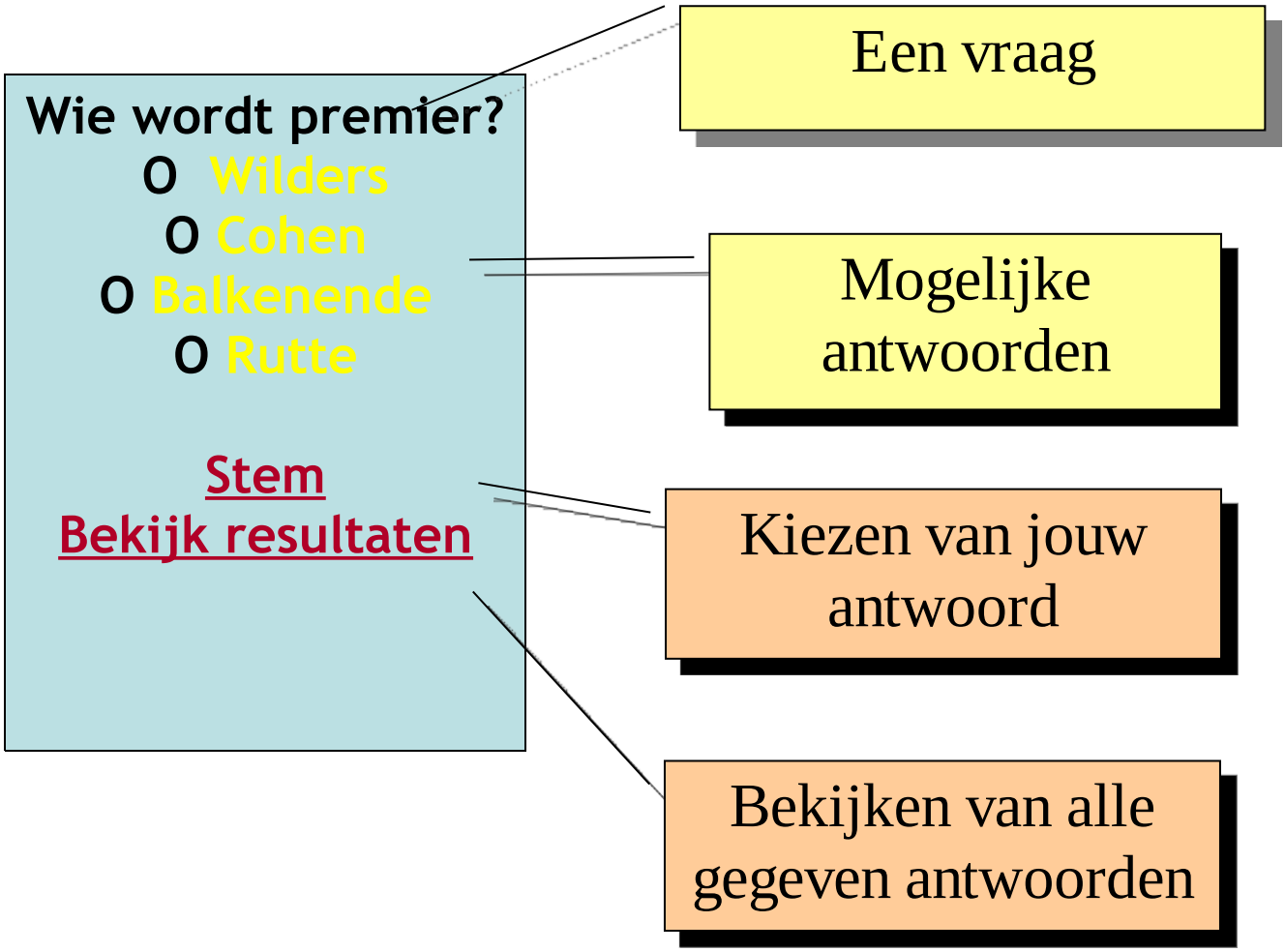
Kenmerkend voor OOP is dat je eerst op zoek gaat naar de objecten die je nodig denkt te hebben en met name naar hun structuur. Dit zijn twee voorbeelden van (eenvoudige) polls die we als uitgangspunt gebruiken.



De volgende stap in het OOP denken is dan het op een abstracte manier modelleren van de eigenschappen en methoden die in deze voorbeelden te onderkennen zijn.

Op de volgende pagina zie je op de onderste helft van de pagina een iets formelere definitie van het object Poll. Hierin is al wat meer nagedacht over de eigenschappen en methoden die nodig zijn en zijn – ter illustratie - bijvoorbeeld ook al de 'onzichtbare' eigenschappen en methoden die met start- en einddatum te maken hebben meegenomen.

De praktijk is meestal dat iemand die op deze manier gaat nadenken over een object steeds meer idee heeft welke eigenschappen en methoden er nodig zijn. Al denkende groeit op die manier het ontwerp.



Poll (object)	
Vraag (tekst) Antwoorden1-4 (tekst) Startdatum (tijd) Einddatum (tijd)	eigenschappen
Aanmaken() Activeren() Tonen() Invullen() ResultatenBekijken() Deactiveren()	methoden

Het blauwdruk van een object: De Klasse

Een objectdefinitie moet vervolgens in de te gebruiken programmeertaal vertaald worden naar een zg. Klasse (Class). In PHP ziet dat er ongeveer zo uit:

```
<?php

class Poll {
        private $vraag;

        function zetVraag($tekst) {
        // hier komt code
        }
}

?>
```

Het is goed gebruik (maar niet verplicht!) om per klasse een php file te schrijven en die ook de naam te geven van de klasse die hier aangemaakt wordt. Dit bestand zou dus Poll.php heten. Gebruikelijk is het om de namen van klassen met een hoofdletter te beginnen (dit is een afspraak, geen verplichting).

De hele klasse definitie staat tussen de { } die volgen op de regel waar **class Poll** genoteerd staat. Ook hier is weer gebruikelijk (en ordelijk) om eerst de noodzakelijke eigenschappen te definiëren (zoals in het voorbeeld $vraag) en daarna de methoden die bij deze klasse horen.

Op zichzelf genomen doet een klasse definitie niets. Je kunt deze klasse pas gebruiken als je in je php applicatie (bijvoorbeeld in **index.php**) op basis van deze klassedefinitie een object aanmaakt (een poll) en die gaat gebruiken. In index.php moeten dan deze regels voorkomen:

```
<?php

include "Poll.php";

$MijnPoll = new Poll();

$MijnPoll->zetVraag("Wat ga je stemmen?");

// enz.

?>
```

Je kunt de blauwdruk van Poll pas gebruiken na de include (logisch). Let op het gebruik van het woord **new** op het moment dat je een nieuw object aanmaakt op basis van de klasse Poll. Een dergelijke handeling wordt instantiëren genoemd. $MijnPoll is dus een **instantie** van de klasse Poll.
Methoden en (soms) eigenschappen van deze instantie kun je gebruiken door na de naam van het object ($MijnPoll) een soort pijl te tekenen: -> gevolgd door de naam

van de eigenschap of de methode die je wilt gebruiken, natuurlijk wel aangevuld met de juiste parameters. Het is niet altijd mogelijk om alle eigenschappen en methoden van een object op deze manier aan te roepen. Hierover later meer.

De Constructor

De constructor is binnen een klasse een speciale methode die *altijd* wordt uitgevoerd op het moment dat er een nieuwe instantie van een klasse gemaakt wordt. Deze methode herken je aan de naam: **__construct()** . De naam begint met twee keer een underscore. Het invullen van deze methode is niet verplicht. Meestal wordt deze methode gebruikt om eigenschappen de juiste startwaarde (de defaullt waarde) mee te geven. Het hangt helemaal van het object en het probleem dat het object op moet lossen af wat er nog verder nodig is in deze methode. Een omgekeerde methode bestaat ook: **__destruct()** Ook deze is niet verplicht. Deze methode wordt uitgevoerd bij het vernietigen van een object.

Toegankelijkheid

Gegeven de wereld waar OOP uit voortgekomen is (het ontwikkelen van grote applicaties) is het begrijpelijk te noemen dat in het denken rond OOP ingebakken zit dat eigenschappen en methoden van een object niet altijd vrij toegankelijk zijn voor andere programmacode. Stel je bijvoorbeeld voor dat je een **User** object maakt en dat de methode die je gebruikt bij het aanmaken van een concreet User object altijd controleert of het wachtwoord wel uit minimaal 8 tekens bestaat, een mix is van hoofd- en kleine letters en minstens één speciaal leesteken bevat. Pas als aan al die voorwaarden voldaan is kan $wachtwoord van een nieuwe useraccount gevold worden.
Vervolgens schrijf een collega van je code die rechtstreeks op het wachtwoord veld aangrijpt (zoiets: $MijnUser->wachtwoord = "12345";). Daarmee passeert hij de vormcheck. Zoiets wil je *per design* voorkomen. Dat doe je door het $wachtwoord veld de extra optie *private* mee te geven bij de definitie:

```
Class User {
        Private $wachtwoord;
        Private $email;
        Public $usernaam;
                                …
}
```

Het gebruik van het toverwoord *private* zorgt ervoor dat uitsluitend de interne methodes van het object dit veld kunnen vullen. Als een eigenschap daarentegen *public* is, kan je er ook van buiten de klassedefinitie op aangrijpen.

Een interne methode kan een eigenschap benaderen of vullen door gebruik te maken van de aanduiding **$this**. $This verwijst altijd naar het huidige object. Dus als **$MijnUser** een object is van het type **User** en de klasse **User** een methode heeft met deze code:

```
Function zetVoornaam($vn) {
        $this->voornaam = $ww;
}
```

zal je met deze opdracht:

$MijnUser->zetVoornaam("Jan");

de voornaam van $MijnUser vullen met "Jan".

Een van de eerste dingen die programmeurs van objecten vaak doen als het gewenst is dat veel eigenschappen *private* blijven is het schrijven van methoden die deze variabelen kunnen *zetten* ('setters') of *ophalen* ('getters'). Het is dan natuurlijk wel zaak om in die methoden controlemechanismen zoals hierboven genoemd in te bouwen. Realiseer je goed dat als je een *private* eigenschap $wachtwoord beschermt met deze 'setter':

```
Function setWachtwoord($ww) {
        $this->wachtwoord = $ww;
}
```

je natuurlijk het kind met het badwater weggooit.

Overerving

Het is in OOP makkelijk mogelijk om een nieuwe objectdefinitie te maken (een nieuwe klasse dus) op basis van een bestaande klasse. Dit doe je door in de regel waarin je de klasse definitie start gebruik te maken van het toverwoord **extends**:

```
Class Poll_database extends Poll ()  {

}
```

Je moet er dan wel voor zorgen dat de klasse Poll bekend is (eerst includen voor je de klasse Poll_database gaat definieren!). De genoemde regel geeft aan dat je een nieuwe klasse wilt gaan definieren **op basis van de klasse Poll**. Dit heeft als praktisch gevolg dat alle eigenschappen en methoden van de klasse Poll ook precies zo bestaan in de klasse Poll_database zonder dat je één regel code hoeft te schrijven.
In de praktijk zal je echter vaak een nieuwe klassedefinitie willen schrijven omdat de oude net niet precies voldoet aan je wensen. Stel je bijvoorbeeld voor dat de klasse Poll een methode **MaakPoll** heeft die de gegevens van een nieuwe poll naar een bestandje poll.txt schrijft. Jij wil echter dat de poll niet in een tekstbestandje maar in een mysql database wordt opgeslagen. Je maakt een nieuwe objectdefinitie voor Poll_database op basis van Poll en zet in die definitie een nieuwe inhoud neer voor **MaakPoll** die gebruik maakt van MySQL. Alle andere methoden en eigenschappen van Poll zijn dan gewoon bruikbaar in Poll_database zonder dat je ze in code hoeft over te kopiëren.

Overerving, Protected en Private Eigenschappen

Eerder hadden we het over private eigenschappen. Dat waren eigenschappen van een object die alleen door methoden van het object zelf benaderd (of gezet) konden worden. Als je een klasse definitie schrijft op basis van een bestaande klassedefinitie die óók dat soort *private* eigenschappen kent, moet je heel erg uitkijken. *Private* eigenschappen zijn voor andere klassen die gebaseerd zijn op de klasse waarin die eigenschappen gedefinieerd worden namelijk óók niet benaderbaar! Soms is dat gewenst, soms is dat ongewenst. Met het veranderen van private in public kun je dat probleem omzeilen, maar dan heb je ook de voordelen van die systematiek niet tot je beschikking. Wat ook kan is eigenschappen niet private of public, maar *protected* declareren. Een protected eigenschap is benaderbaar door methoden uit de eigen klasse of uit klassen die gebaseerd zijn op de eigen klasse.

```
Class Poll {
        Private $vraag ;
        Protected $antwoord;
        …
}

Class PollDatabase extends Poll {
```

```
        Function zetVraag($v) {
                $this->vraag = $v;
        }
        Function zetAntwoord($a) {
                $this->antwoord = $a;
        }
}
```

Dit eenvoudige voorbeeldje heeft in de klasse PollDatabase twee functies die aangrijpen op eigenschappen die al horen bij de 'moederklasse', Poll. De eigenschap $vraag is in Poll private, de eigenschap $antwoord is in Poll protected. Dat heeft tot gevolg dat de methode zetVraag van PollDatabase geen kans heeft en zijn werk niet kan doen, maar de methode zetAntwoord van PollDatabase wel er in zal slagen om $antwoord te veranderen. Code buiten de klassedefinities van Poll of PollDatabase zal hoe dan ook geen van beide variabelen $vraag en $antwoord kunnen benaderen.

Abstracte Klassen en Interfaces

Stel je voor dat je een besturingssysteem schrijft en dat je wilt afdwingen dat printerfabrikanten bij het schrijven van hun printerdrivers een bepaald klassemodel hanteren met precies die eigenschappen en methoden die jij wilt zien. Je kunt dit bereiken door een Abstracte Klasse te schrijven. Hierin noteer je alle eigenschappen en methoden die je (minimaal) toe wilt staan:

```
Abstract class Printer {
        Protected $type;
        Protected $fabrikant;
        Protected $keer;

        Public PrintDocument($file);
        Public FoutMelding($melding);

        Public AantalKerenGebruikt() {
                Print "$keer gebruikt.";
        }
}
```

Het verschil met andere klassen is, behalve de aanduiding Abstract voor Class dat in sommige methoden geen code staat. Op het moment dat een fabrikant op basis van deze klasse een printerdriver schrijft **moet** hij de code voor die functies aanvullen. Een klasse die gebaseerd is op een abstracte klasse moet alle abstracte methoden ingevuld hebben.

Bij een **Interface** ga je nog een klein stapje verder. Een interface bevat helemaal geen code meer, maar echt alleen de namen (en parameters) van methoden die een klasse die gebaseerd is op deze interface (minimaal) moet gaan invullen.

Abstracte Klassen en Interfaces zijn dus bij uitstek hulpmiddelen om bij andere programmeurs af te dwingen dat klassendefinities aan bepaalde eisen voldoen.

Voorbeeldje:
Stel je wilt een klasse Gebruikers gaan bouwen op basis van de abstracte interfaces contacten en users. De interface Contacten beschrijft de algemene architectuur van alle 'contacten' die een bedrijf heeft met anderen, de interface Users beschrijft de algemene architectuur van alles wat netwerkusers van het bedrijf moeten kunnen. De klasse gebruikers is een soort samenstelling van die twee. Je wilt van gebruikers de naam kunnen noteren en het email adres, maar je wilt ook dat ze kunnen inloggen en dat hun inloggegevens kunnen worden gechecked. Door de klasse Gebruikers te baseren op de interfaces Contacten en Users wordt je gedwongen om in de klassedefinitie van gebruikers alle dingen in te vullen die volgens de interfaces horen bij de rollen Contacten en Users.

Zo iets werkt natuurlijk alleen goed als de interfaces zelf zorgvuldig overdacht zijn!

```
Interface contacten        {
                function zetNaam($naam);
                function getNaam();
                function zetEmail($adres);
                funciton getEmail();
        }

Interface users      {
                function checkWachtwoord($ww);
                function checkType($type);
        }

class Gebruikers implements contacten, users {
        // moet dus code bevatten die de volgende functies definiëren:
                function zetNaam($naam) {
                        $this->zetNaam = $naam;
                }
        // enzovoorts

        }
```

Hier ontbreekt de code.

Hier mag dat niet!

Weer wordt duidelijk dat OOP enorm veel waarde hecht aan het overdenken van de architectuur van de objecten die met elkaar een informatieprobleem oplossen. Via een *interface* kun je die architectuur zelfs gaan afdwingen!

Voor- en nadelen van OOP

Hoe nuttig is OOP? Iemand die 'even gauw' een applicatie moet schrijven voor een klein probleem heeft niets aan OOP. OOP heeft alleen zin als je een informatie probleem drastisch en grondig op wilt lossen. Je zult bij OOP in de aanlooptijd naar een werkende applicatie altijd veel tijd moeten stoppen in het nadenken over de architectuur van je objecten.
De zaak ligt echter anders als je bij een applicatie gebruik wilt maken van werk dat anderen al gedaan hebben. Gebruik maken van klassedefinities van anderen is in OOP een fluitje van een cent. En als die klassedefinities een beetje volledig en goed gerijpt zijn neemt dat je een hoop werk uit handen. Het is altijd weer opvallend hoe de programma code die via events objecten met elkaar verknoopt in OOP programma's soms verrassend simpel en overzichtelijk oogt. Dat soort programma's schrijf je snel en je kunt er ook snel de fouten uit halen.

```
<?php

include "Poll_db.php";

$MijnPoll = new Poll_db();

$MijnPoll->zetVraag("Je moet kiezen:");
$nr = $MijnPoll->checkId();
if (empty($nr)) {
        // de poll bestaat nog niet: aanmaken dus!
        $MijnPoll->zetAntwoord(1,"Amsterdam");
        $MijnPoll->zetAntwoord(2,"Zwolle");
        $MijnPoll->zetAntwoord(3,"Hasselt");
        $MijnPoll->zetAntwoord(4,"Heerde");
        $MijnPoll->maakPoll();
} else {
        // de poll bestaat: ophalen dus!
        $MijnPoll->haalOp($nr);
}

$MijnPoll->toon();  # op scherm zetten
$MijnPoll->verwerkStem();  # stem verwerken (incl. resultatenknop!)
$MijnPoll->updatePoll();  # gegevens in database aanpassen

?>
```

Het bovenstaande code voorbeeld is een voorbeeld van de overzichtelijkheid van OOP programma's. Alle complexiteit zit eigenlijk verborgen in de klassedefinitie "Poll_db.php". Bovenstaande code doet niet anders dan een nieuw Poll object definieren, kijken of de vraag die men stellen wil al voorkomt, zo ja onder welk nummer en dan worden alle gegevens opgehaald, zo nee: dan wordt de poll aangemaakt en opgeslagen in de database. En de laatste drie regels doen niets anders dan de poll op scherm zetten, een eventueel uitgebrachte stem te verwerken en die verwerking ook in de database te noteren.

In een taal als Java is *alles* een object, inclusief het aanroepende programma. Dat brengt onder andere met zich mee dat bijna alles in java, en dus ook het aanroepende programma, methoden en eigenschappen heeft die horen bij objecten in Java. De makers van Java hebben dat onder meer gebruikt voor het toevoegen

van allerlei handige debug en rapportage handigheidjes. Denk aan een methode als .toString() die een object "printbaar" maakt. Zo goed als ieder object in java kent die methode. Getalsobjecten (als integer en double) hebben op die manier allerlei ingebouwde functies die de getallen converteren naar andere typen of afronden.
En zo kunnen we nog wel even doorgaan.

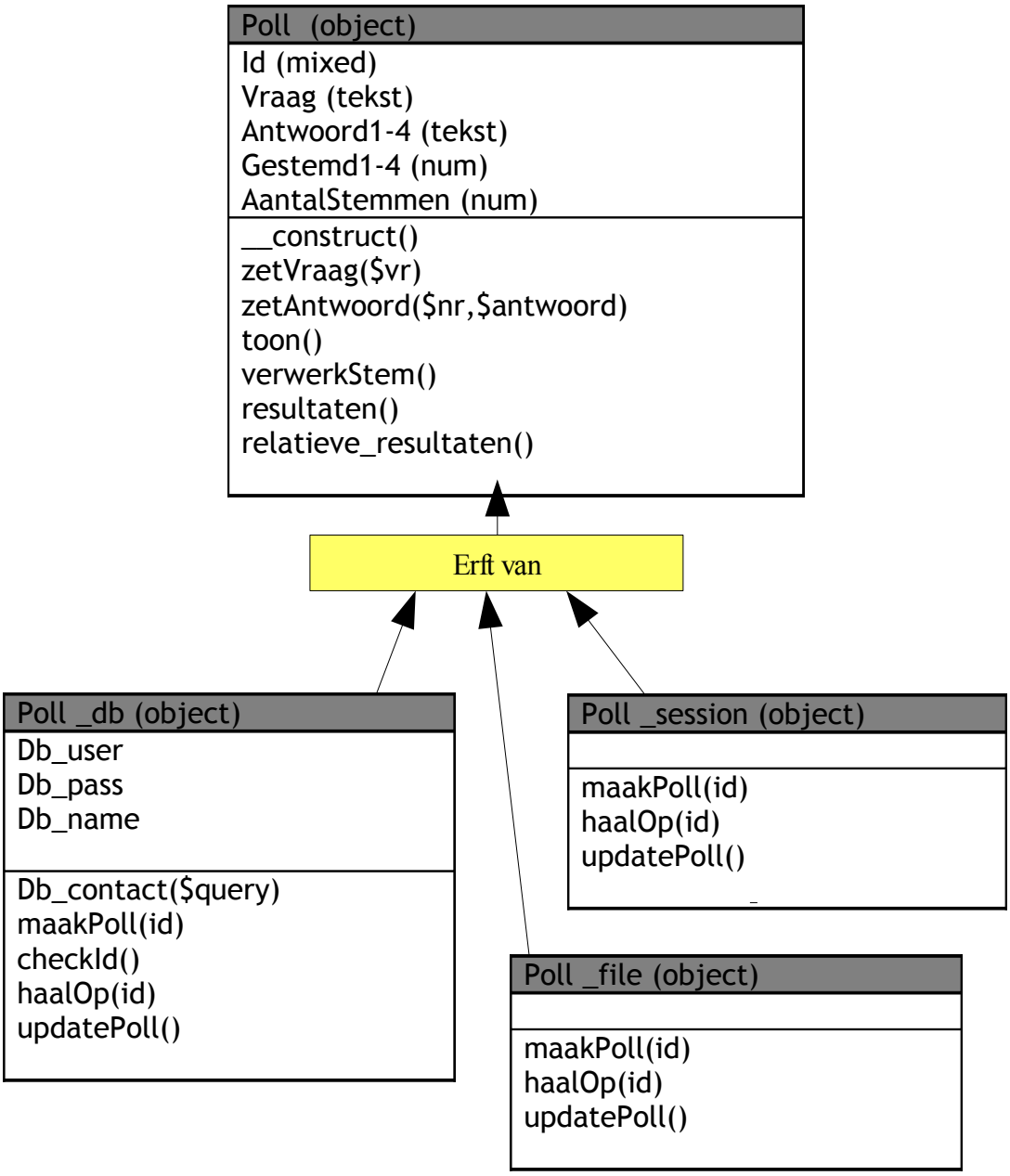
In een OOP wereld is *documentatie* van groot belang. Objecten kun je alleen maar gebruiken als je weet welke eigenschappen en methoden die objecten allemaal hebben. Daarom zie je in talen als Java, Ruby en dergelijke ingebouwde mechanismen die uit een stukje broncode met een paar drukken op de knop heel bruikbare documentatie destilleren (google maar eens op Javadoc).

Onder de streep zou je dus kunnen betogen dat OOP vooral nuttig is omdat het je dwingt de architectuur van je programma en de objecten die het gebruikt goed te overdenken, maar dat de prijs die je daarvoor betaalt is dat deze aanpak veel tijd kost als je je eigen objecten allemaal zelf wilt ontwikkelen. Maak je graag gebruik van de objecten van anderen dan heb je alleen maar voordeel van OOP.

Binnen PHP zijn er complete functiebibliotheken, PEAR bijvoorbeeld, die intern geheel volgens de OOP methode zijn opgezet. Dit soort functiebibliotheken worden bijvoorbeeld gebruikt om:

- inlogsystemen te maken
- databasetoegang te versimpelen
- formuliercontrole makkelijk mogelijk te maken
- webpaginaopmaak te standaardiseren

Als je gebruik maakt van dit soort bibliotheken zul je merken dat je eigen code overzichtelijker en eenvoudiger blijft terwijl de kwaliteit van de informatieverwerking toch fors toeneemt.



Voor code voorbeelden: zie ELO