

# 5 UML

## 5.1 Inleiding UML

Planning	Inleveren
	Opgaven 1 t/m 2

### Websoftwareontwikkeling

De volgende technieken kunnen gebruikt worden bij grote webprojecten:

- iteratieve ontwikkeling
- eisenanalyse
- gegevens modelleren
- coderen en testen

Daarnaast zijn er *good practice*-standaarden die zorgen voor de kwaliteit van het ontwikkelingsproces:

- codeer software die herbruikbaar is;
- codeer software die eenvoudig te onderhouden is;
- gebruik een ontwikkelingsplatform;
- documentatie aanleggen;
- houd logica, content en presentatie apart (JavaScript, PHP, HTML en CSS).

### Websoftwareanalyse en -ontwerp

Softwareanalyse is een systematische benadering voor softwareontwikkeling. Dit is nodig voor complexe maar stabiele websites die goed te onderhouden zijn. Helaas ontbreekt softwareanalyse bij veel webprojecten.

Een reden is dat webontwikkeling vaak documentgeoriënteerd is: documentstructuur, grafisch ontwerp en productie. Deze benadering werkt prima voor kleine en statische sites, maar niet voor complexere sites.

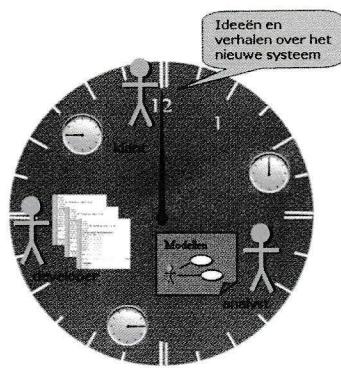
Een tweede reden is dat scripting-talen, zoals HTML, JavaScript en PHP, gemakkelijk en toegankelijk zijn, zodat men meteen gaat coderen zonder veel aandacht voor analyse en beveiliging.

Ten slotte is er bij webprojecten vaak het idee dat er geen tijd is voor enige planning. De resultaten zijn onstabiele applicaties, gemiste deadlines en onleesbare

scripts. Het ontwikkelen van webapplicaties is een nieuwe discipline en een methodische benadering is hard nodig.

### Iteratieve softwareontwikkeling

Iteratieve softwareontwikkeling is een systematische aanpak voor het plannen en bouwen van webprojecten. Iteratieve softwareontwikkeling zou je als volgt kunnen zien:

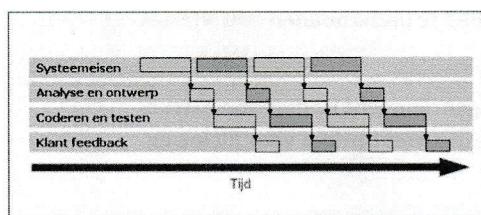


Figuur 5.1 Iteratieve softwareontwikkeling

Een iteratie is een cyclus, zoals de wijzers van de klok die ronddraaien. Eerst wordt de klant geïnterviewd. Vanuit zijn of haar verhalen en ideeën (user-stories) maak je een eisenanalyse. In dit hoofdstuk leren we vier UML-diagrammen kennen om informatiemodellen te maken. Deze modellen geef je aan de developer. De developer vertaalt je modellen naar programmacodes. Aan het einde van de eerste iteratie (cyclus) worden de resultaten gepresenteerd aan de klant. De klant geeft vervolgens zijn of haar mening en feedback. Deze feedback is het begin van de tweede iteratie.

### Planning softwareontwikkeling

De planning van elke iteratie zou je als volgt kunnen zien:



Figuur 5.2 Planning software-ontwikkeling

### Wat is UML?

Unified Modeling Language (UML) is een modelleertaal die gebruikmaakt van grafische notatie om modellen te ontwerpen. UML bevordert de communicatie tussen workflowspecialisten, softwaredesigners en andere professionals, zoals mensen met kennis van een sector (bijvoorbeeld verzekeringen, gezondheidszorg of transport).

UML wordt gebruikt voor het maken van modellen voor alle mogelijke systemen:

- complexe informatiesystemen;
- technische systemen zoals telecommunicatie;
- ingebetde realtime systemen zoals mobiele telefoons en auto's;
- softwaresystemen zoals besturingssystemen en databases;
- bedrijfssystemen hebben beperkingen zoals beschikbaarheid van mensen en middelen en geldende regels.

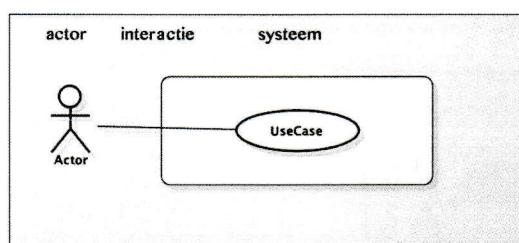
Een model is geen definitieve oplossing. Het is maar een van de vele mogelijke oplossingen. Een bruikbaar model moet aan de volgende eisen voldoen:

- *accuraat*: beschrijft het te bouwen systeem;
- *consistent*: heeft geen conflicterende informatie;
- *begrijpelijk*: is eenvoudig uit te leggen en te wijzigen.

### Eisenanalyse met use-case-diagrammen

Eisenanalyse volgt uit de wensen van de opdrachtgever. De ideeën en verhalen (user-stories) van de opdrachtgever vertaal je in een of meer use-case-diagrammen.

Een use-case-diagram beschrijft wat een systeem moet doen. Het bestaat uit meerdere use-cases die acties en interacties beschrijven tussen de actor (user) en het systeem. Een use-case-diagram gebruikt de volgende symbolen:

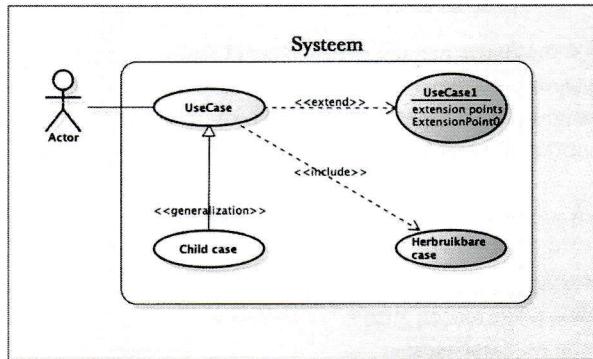


Figuur 5.3 Een actor in interactie met het systeem

Een actor mag een gebruiker of een ander systeem zijn. In de vorige figuur is Use-Case de naam van de use-case.

### Afspraken over notatie

Use-cases gebruiken de volgende notaties om scenario's te modelleren:

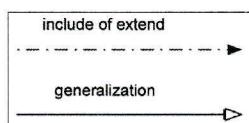


Figuur 5.4 Use-case-notatie

In figuur 5.4 zie je een actor in interactie met een systeem.

Je ziet ook verschillende relaties ontstaan tussen de use-cases.

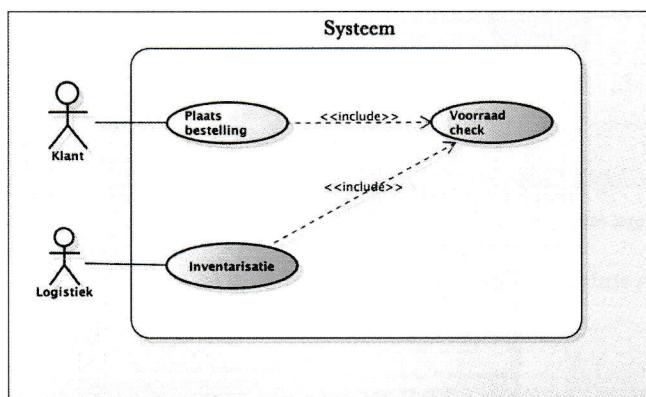
Een <<include>>- en <<extend>>-relatie geef je aan met een gestippelde pijl. Een <<generalization>>-relatie geef je aan met een doorgetrokken pijl:



Figuur 5.5

### De <<include>>-relatie

Gebruik <<include>> bij herbruikbare use-cases. Bijvoorbeeld, zowel de bestelling- als de inventarisatie-use-cases kunnen gebruikmaken van de voorraad-check-use-case. De voorraad-check-use-case mag gebruikt worden door andere use-cases.



Figuur 5.6 Een herbruikbare use-case

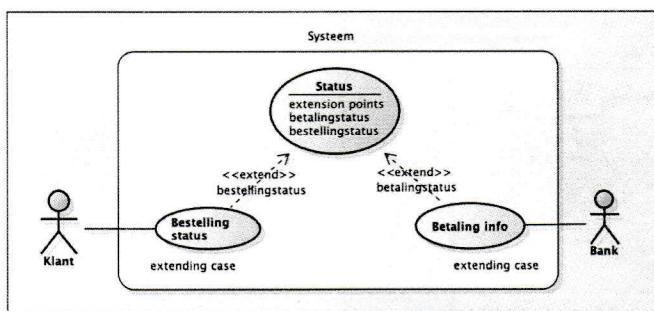
In dit scenario plaatst de klant een bestelling. De bestelling-use-case leidt tot een voorraad-check. De inventarisatie-use-case maakt ook gebruik van de voorraad-check-use-case.

Wanneer je dezelfde functionaliteit identificeert binnen twee of meer use-cases, kun je een single use-case maken. Zo kunnen meerdere use-cases dezelfde functionaliteit hergebruiken.

### De <<extend>>-relatie

Gebruik <<extend>> om extension-points van een use-case toe te voegen aan een extending use-case. Een extension-point is uitgebreide functionaliteit die toegevoegd kan worden aan andere use-cases. Een use-case die gebruik maakt van een extension-point noemen we de extending use-case. Een extension-point is functionaliteit die je soms gebruikt. Dit wordt bepaald door de omstandigheden. Het verschil tussen een extend- en een include- relatie is dat een include-relatie altijd wordt uitgevoerd.

Neem bij voorbeeld de use-case "Ambulance" die de use-case "Sirene" include. Dan zeggen we dat de ambulance een sirene heeft gekregen. Vervolgens hebben we de use-case "Geluidden" met de extension-points "politieGeluid" en "brandweerGeluid". Dan kunnen we desgewenst aan onze "Ambulance" use-case de functionaliteit in de extension-point "politieGeluid" toevoegen.



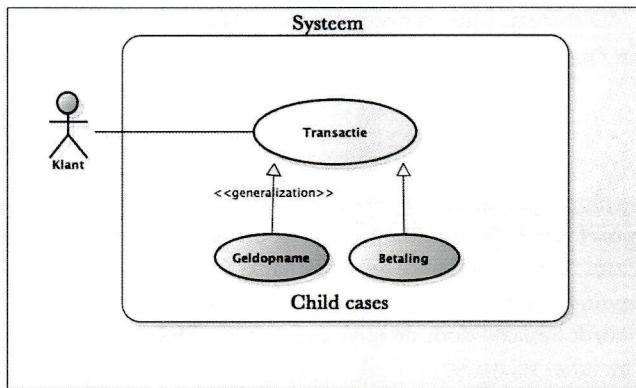
Figuur 5.7 Extending cases

Wanneer een use-case te complex wordt splitsen we de use-case op in extension-points. Een extending case mag een of meer extension-points uit een complexe use-case uitvoeren.

In dit voorbeeld voert de extending case **Bestellingstatus** niet de volledige case **Status** uit, maar vraagt het specifieke extension-point **bestellingstatus** aan de use-case **Status**.

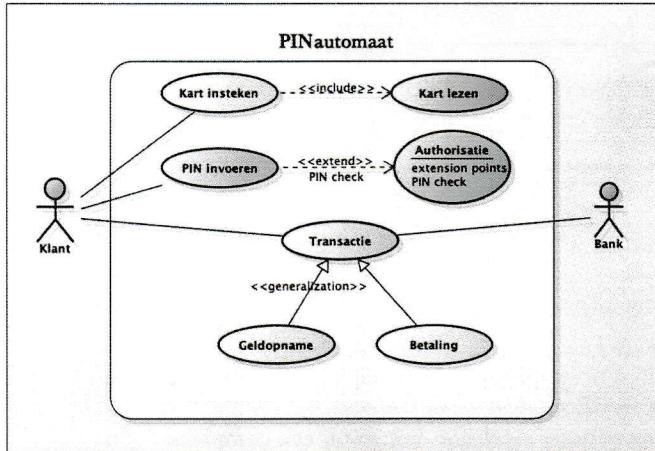
### De <<generalization>>-relatie

Gebruik <<generalization>> bij een use-case die gelijksoortig is aan een andere use-case, maar een gespecialiseerde soort is.



Figuur 5.8 Een generalization use-case

In dit voorbeeld is **Transactie** de parent-use-case en **Geldopname** en **Betaling** zijn de child-cases. De child-case lijkt erg op de parent-case, maar is wel een speciale versie van de parent-case. In de volgende figuur zien we een voorbeeld van de generalization-relatie:



Figuur 5.9 Een geldautomaat use-case-diagram

Dit use-case-diagram beschrijft het volgende scenario:

- De klant steekt de bankpas in de pinautomaat.
- Het systeem leest de bankpas.
- De klant typt zijn pincode in.
- Het systeem voert een pin-check uit.

- De klant of de bank kiest een transactie (geldopname of betaling).
- Het systeem voert de transactie uit.

Er zijn meerdere gratis softwarepakketten voor het tekenen van UML-diagrammen.

Ga bijvoorbeeld naar de volgende website van Astah-community:

<http://astah.net/editions/community>

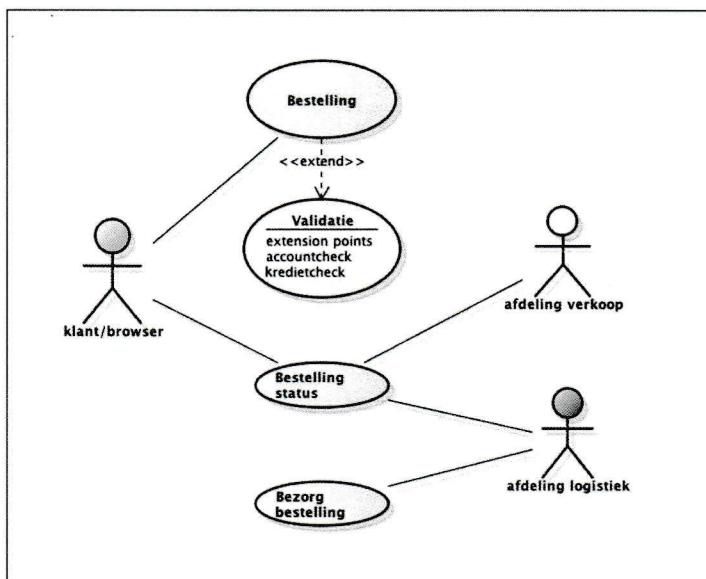
Download en installeer een van de UML-softwarepakketten en maak de volgende opgaven.

- *Opgave 1*

Zoek, download en installeer de software en maak een use-case-diagram volgens het volgende scenario:

- Een klant plaatst een bestelling.
- Het systeem valideert de bestelling als volgt:
  - voert een krediet-check uit.
  - voert een account-check uit.
- De klant, de afdeling verkoop en de afdeling logistiek kunnen:
  - bestellingstatus checken.
- Afdeling logistiek kan:
  - bestelling bezorgen.

Het resultaat moet er als volgt uitzien:



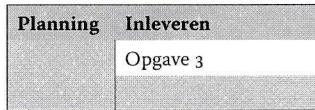
Figuur 5.10

- *Opgave 2*

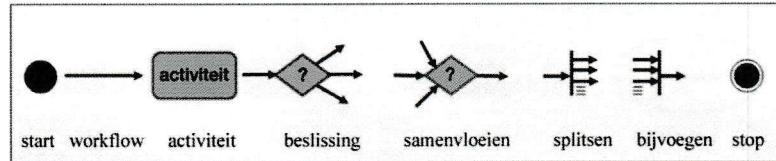
Use-cases vormen een essentiële tool voor de **casenanalyse** en voor de planning en controle van een webproject. In deze opgave maak je bovenstaand use-case-diagram als volgt af:

- Voeg een nieuw extension-point **Inlogcheck** toe bij de use-case **Validatie**.
- Voeg een nieuw extension-point **Voorraadcheck** toe bij de use-case **Validatie**.
- Maak een nieuwe use-case **Inloggen** zodat zowel de klant als de afdelingen verkoop en logistiek kunnen inloggen. Laat deze use-case een **Inlogcheck** uitvoeren.
- Laat de use-case **Bestelling** een **Voorraadcheck** uitvoeren.

## 5.2 Activiteitendiagram (workflow)

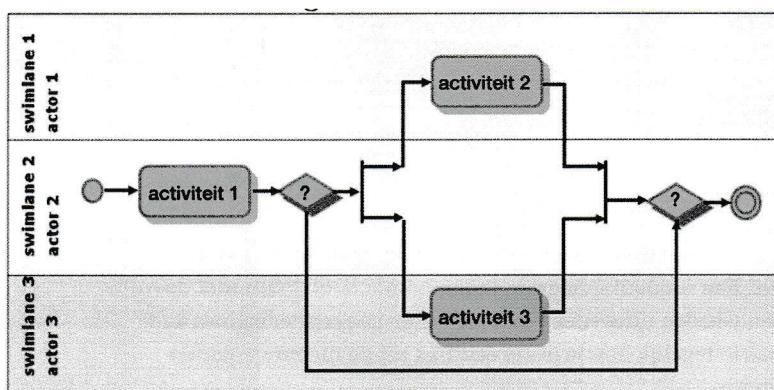


Een activiteitendiagram brengt focus in de stroom van activiteiten van een proces of workflow. Een activiteitendiagram toont hoe activiteiten simultaan lopen of afhankelijk van elkaar zijn. Bij een activiteitendiagram gebruik je de volgende symbolen:



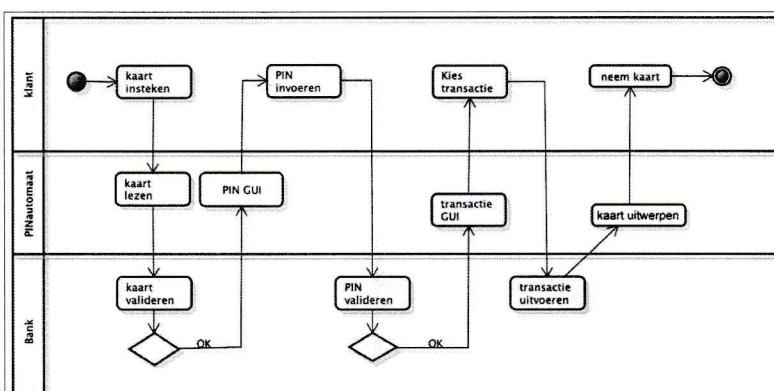
Figuur 5.11 De symbolen van het activiteitendiagram

Hieronder staat een activiteitendiagram met *swimlanes* (zwembanen). Het diagram geeft weer hoe drie actoren in interactie komen tijdens een workflow of proces. Elke actor krijgt zijn eigen zwembaan.



Figuur 5.12 Een activiteitendiagram

Doordat elke actor zijn eigen swimlane krijgt, zijn activiteitendiagrammen ideaal voor het weergeven van processen die parallel plaatsvinden.



Figuur 5.13 Activiteitendiagram van een geldautomaat

- **Opgave 3**

Maak met behulp van Astah bovenstaand activiteitendiagram af door de volgende activiteiten toe te voegen:

- Als de kaart niet geldig is moet de kaart teruggegeven worden.
- Als de pin ongeldig is mag de klant twee keer herkansen.
- Na drie keer fout pinnen kaart uitwerpen.
- Na de eerste transactie mag de klant weer een transactie kiezen.

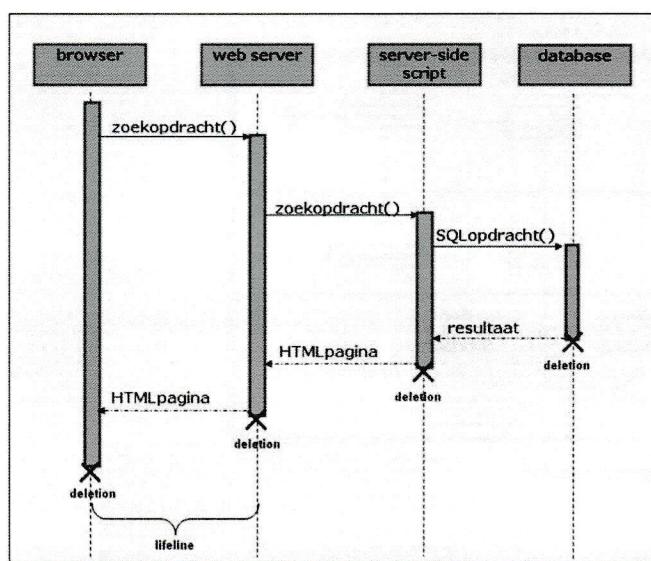
### 5.3 Sequentiediagram

Planning	Inleveren
	Opgave 4

Sequentiediagrammen zijn dynamische diagrammen die beschrijven hoe systemen of actoren samenwerken. Een sequentiediagram is een interactiediagram met details over hoe activiteiten worden uitgevoerd in de tijd. Een sequentiediagram kan maar één use-case-scenario tegelijk beschrijven: één pad uit de meerdere paden van een activiteitendiagram.

Sequentiediagrammen zijn zeer effectief voor het beschrijven van tijdgerelateerde zaken, zoals netwerken en interacties tussen use-cases.

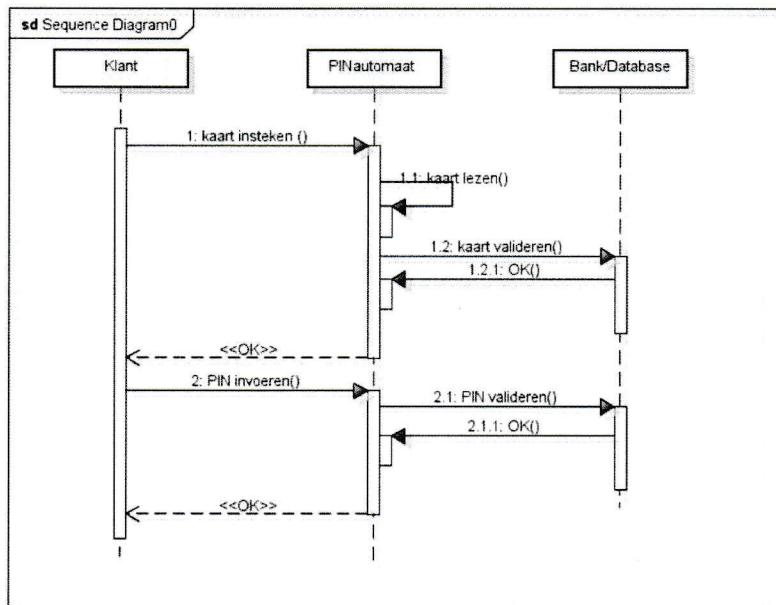
Je zou rechtstreeks vanuit een sequentiediagram kunnen gaan coderen.



Figuur 5.14 Voorbeeld van een volgorde van events

De klant stuurt een http-request naar de webserver. De webserver voert het script uit. In het script wordt een SQL-opdracht naar de database verstuurd. Het resultaat wordt in een HTML-pagina opgesteld en de webserver stuurt de HTML-pagina naar de browser.

In de volgende figuur zien we een begin van een sequentiediagram voor de pin-automaat.



Figuur 5.15 Voorbeeld van een sequentiediagram

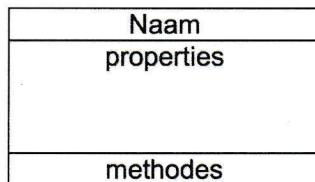
- Opgave 4*

Maak bovenstaand sequentiediagram als volgt af: laat de klant een geldopname-transactie uitvoeren.

## 5.4 Het klassendiagram (class diagram)

Planning	Inleveren
Opgaven 5 t/m 8	

Uit de vorige UML-diagrammen hebben we de actors en hun interactie met het systeem gedistilleerd. Nu gaan we de objecten in het systeem modelleren. Dit doen we met het UML-klassendiagram. Een UML-klassendiagram (class diagram) is een statisch model voor het beschrijven van de objecten in een systeem. Een klasse (class) beschrijft de structuur en de eigenschappen (properties) en tevens de functies (methodes) van het object. Het beschrijft ook de relaties tussen de objecten. We tekenen een klassendiagram als een rechthoek met drie vakken: een vak voor de naam van de klasse, een vak voor de properties en een vak voor de methodes.



Figuur 5.16

### Class met properties

Een class mag properties en methodes hebben. Hieronder zie je een klassendiagram voor een class met de naam Persoon en de properties naam, leeftijd en geslacht:



Figuur 5.17

### Class met methodes

Een class kan methodes hebben. We gebruiken methodes meestal om de properties (eigenschappen) van de class te verwerken. Hieronder zie je een klassendiagram voor de class Persoon met de methode getNaam():



Figuur 5.18

Properties en methodes met twee of meer woorden zoals getNaam() schrijven we in camelCase. Dat wil zeggen: zonder spaties en de eerste letter van het tweede woord met een hoofdletter.

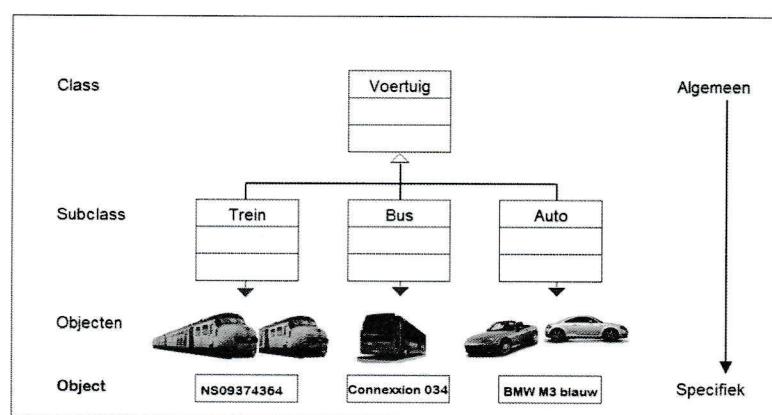
### Subclass

Een class is een algemene beschrijving van een groep objecten. Een subclass is een meer specifieke beschrijving van objecten.

### Object

Een object is een voorbeeld van een class of subclass. Als we eenmaal een subclass gedefinieerd hebben, kunnen we een of meer objecten uit de subclass maken.

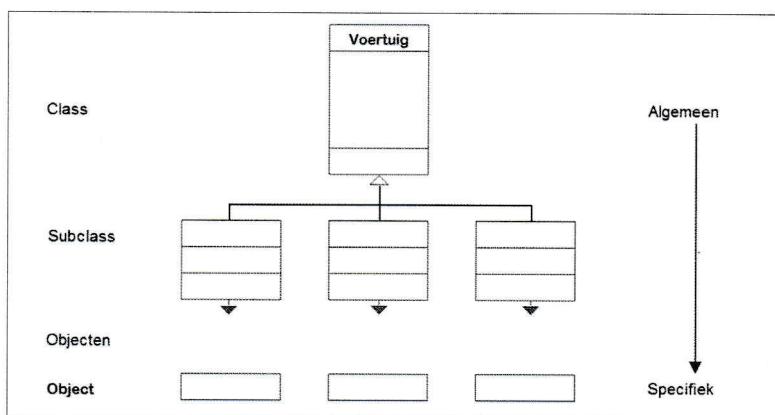
De class Voertuig is een algemene beschrijving van voertuigobjecten. De subclass Trein is een specifieker beschrijving van voertuigobjecten. Het object trein NS09374364 is een voorbeeld van de class Voertuig en de subclass Trein.



Figuur 5.19

- Opgave 5*

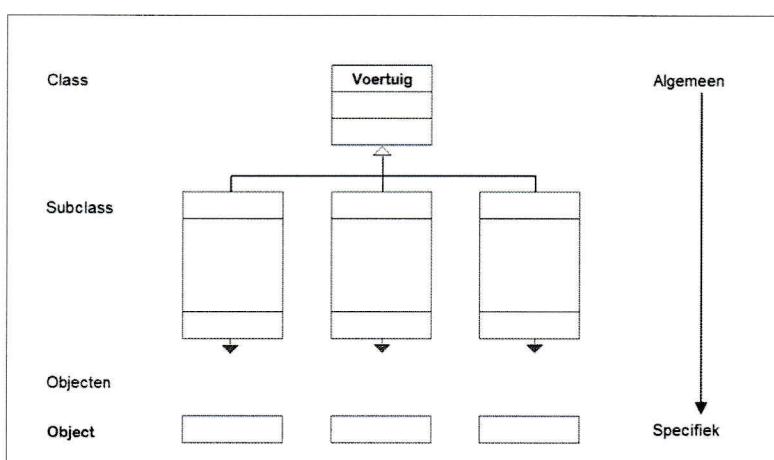
Bedenk drie properties voor de class Voertuig. Schrijf ze op in de volgende figuur.



Figuur 5.20

- Opgave 6*

Voor deze opgave bedenk je drie subclasses voor de class Voertuig. Bedenk drie unieke properties voor iedere subclass. Vervolgens bedenk je drie voorbeelden van objecten van de subclasses.



Figuur 5.21

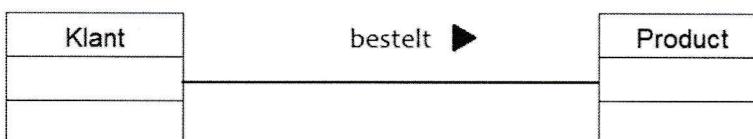
### Relaties

Classes staan niet op zichzelf. Een class heeft relaties met andere classes. De relaties van classes lijken op de relaties tussen de tabellen in een database, maar zijn toch verschillend. We bespreken de volgende class-relaties:

- associaties
- multiplicity
- trio-associatie
- aggregatie
- afhankelijkheid
- generalisatie

### Associaties

Een associatie is een verbinding tussen twee classes en de afgeleide objecten. De associatie zorgt ervoor dat de objecten '*van elkaar bestaan weten*'. Deze relatie wordt getekend als een doorlopende lijn tussen de twee classes zoals in de volgende figuur:



Figuur 5.22

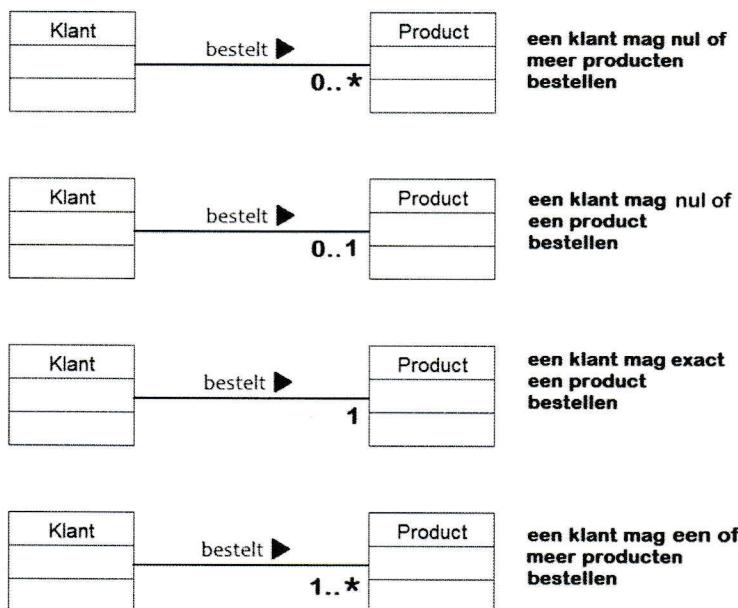
In bovenstaande figuur is de relatie tussen de class Klant en de class Product een associatie. Aan deze associatie hebben we een naam gegeven en deze staat boven de lijn. Deze associatie lezen we als volgt: de Klant bestelt een Product.

### Multiplicity

Een relatie kent multiplicities. Multiplicity betekent ‘veelheid’ en geeft aan hoeveel objecten deelnemen aan de relatie. De volgende tabel geeft de multiplicities en de betekenis aan:

Multiplicity	Betekenis
0	nul
1	een
*	meer
0..*	nul of meer
0..1	nul of een
1..*	een of meer

In de volgende figuur zien we de verschillende mogelijke multiplicities tussen de class Klant en de class Product:

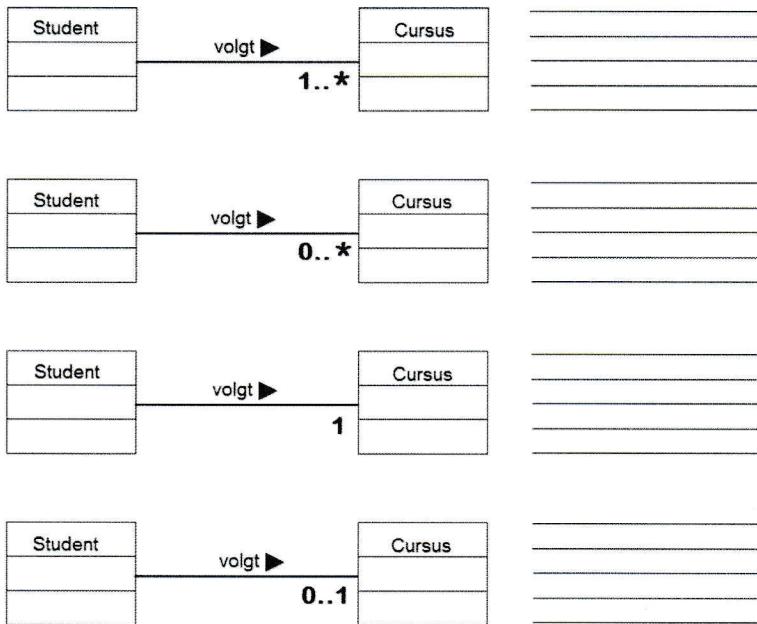


Figuur 5.23 Multiplicity

- **Opgave 7**

In deze opgave schrijf je in de volgende figuur aan de rechterkant de betekenis van de aangegeven relatie en multiplicity.

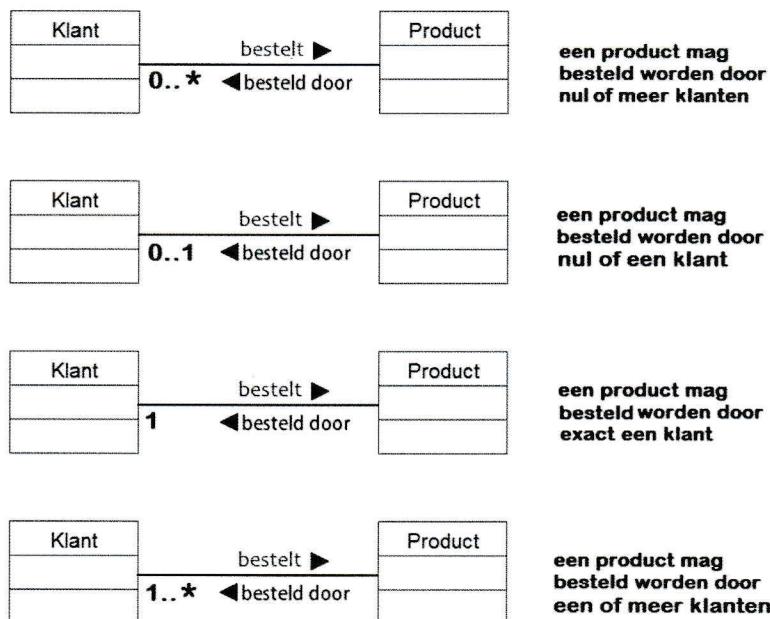
### Multiplicity



Figuur 5.24 Meerdere multiplicities

### Tweerichtingsverkeerrelatie

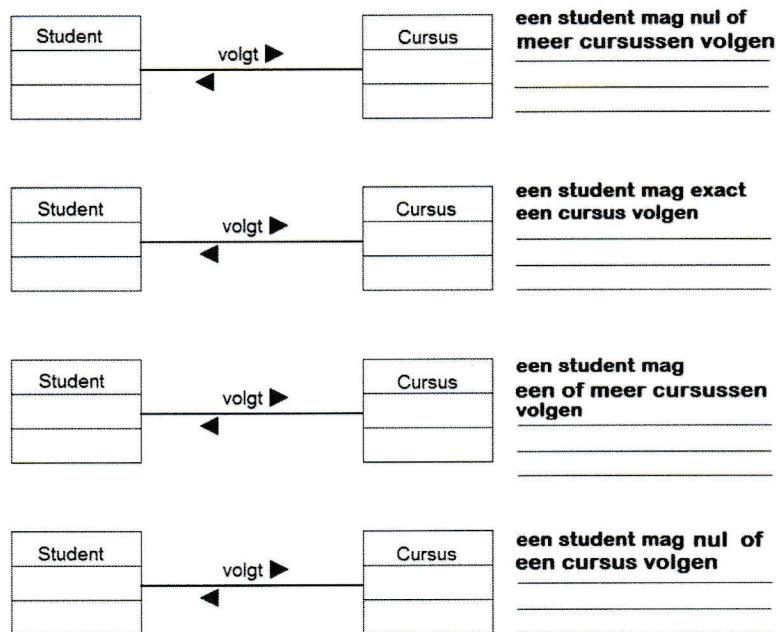
Een relatie kan ook gelezen worden van rechts naar links. In de volgende figuur krijgt ook de eerste class een multiplicity.



Figuur 5.25 Multiplicity: tweerichtingsverkeer

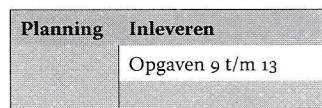
• *Opgave 8*

Geef de tweerichtingsverkeerrelatie aan van de associatie en de multiplicity tussen de klassen Student en Cursus in de volgende figuur.



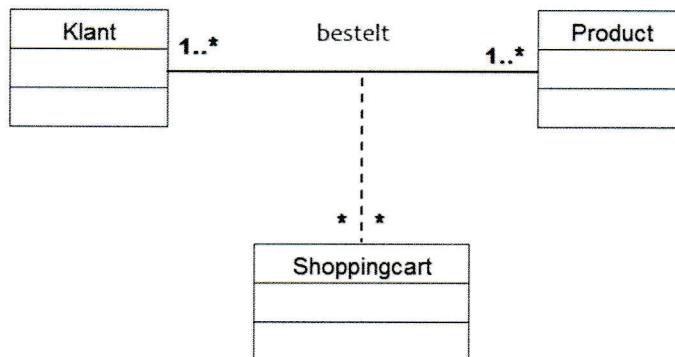
Figuur 5.26

## 5.5 Speciale class-relaties



### Trio-associatie

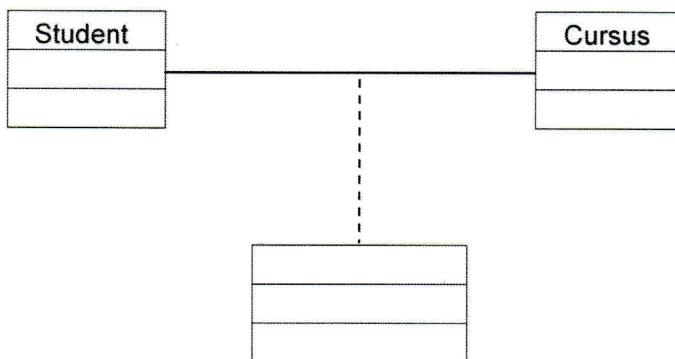
Een trio-associatie geeft de relaties aan tussen drie klassen. In de volgende figuur is te zien hoe een klant een of meer producten mag bestellen en hoe elk besteld product geassocieerd is met een winkelwagen (shoppingcart).



Figuur 5.27 Trio-associatie

- *Opgave 9*

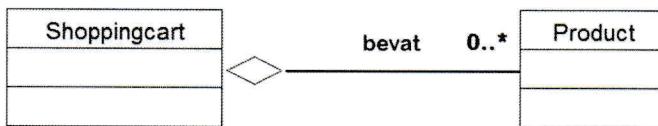
In deze opgave geef je in de volgende figuur aan wat associaties en multiplicities tussen de drie classes zijn.



Figuur 5.28

### Aggregatie

Een aggregatie is een speciale vorm van associatie. Een geaggregeerde class maakt deel uit van een andere class. Een voorbeeld in ons online bestellingsmodel is de class Product.



Figuur 5.29 Aggregatie

Deze relatie wordt getekend met een lijn met een lege ruit aan de kant van de class die de andere class bevat. In dit geval bevat de class ShoppingCart de class Product. De class ShoppingCart blijft bestaan, zelfs als er geen Producten zijn.

• *Opgave 10*

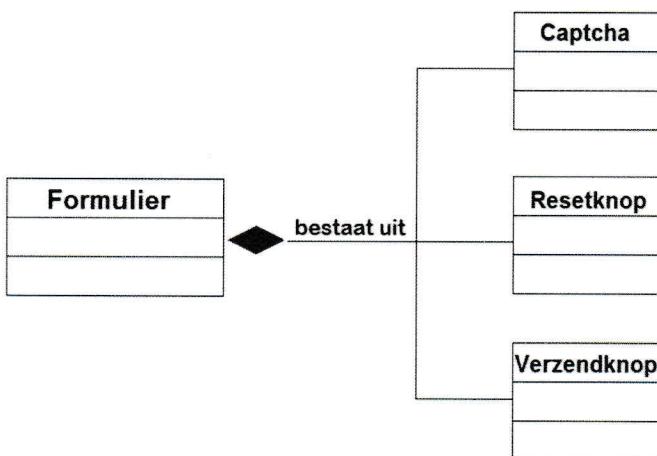
Teken de aggregatierelatie tussen de twee classes in de volgende figuur.



Figuur 5.30 Aggregatierelatie

### Compositie-aggregatie (afhankelijkheid)

Een compositie-aggregatie is eigenaar van al zijn onderdelen. De eigenaar kan niet bestaan zonder zijn onderdelen. Deze relatie tekenen we met een lijn en een gevulde ruit aan de kant van de eigenaar.



Figuur 5.31 Compositie-aggregatie

De relatie compositie-aggregatie krijgt de naam 'bestaat uit'. De class Formulier kan niet bestaan zonder de compositie van de aggregaties (Captcha, Resetknop, Verzendknop). Een compositie-aggregatie wordt gecodeerd als een property in de class.

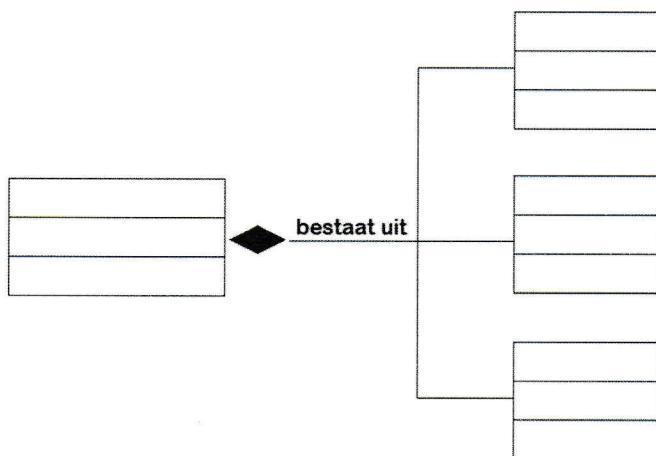


Figuur 5.32



• *Opgave 11*

Teken een compositie-aggregatie-relatie voor een class Bestelformulier.



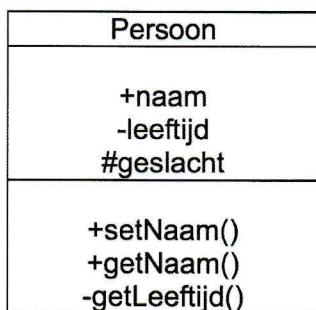
Figuur 5.33 Compositie-aggregatie

### Encapsulation

In het klassendiagram kunnen we de zichtbaarheid van de properties en methodes als public, private en protected bepalen. Dit doen we met de volgende markers.

marker	keyword	betekent
+	public	(publiek) zichtbaar in andere classes
-	private	(privé) zichtbaar alleen in eigen class
#	protected	(beschermd) zichtbaar in eigen class en subclasses

Hieronder geven we de zichtbaarheid aan van de properties in een klassendiagram.

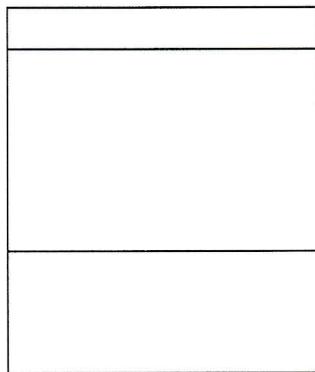


Figuur 5.34

Volgens dit klassendiagram is de property naam public en zichtbaar in alle andere classes. De property geslacht is protected en alleen zichtbaar in deze class en subclasses van deze class. De methode getLeeftijd en de property leeftijd zijn private en alleen zichtbaar in deze class.

- *Opgave 12*

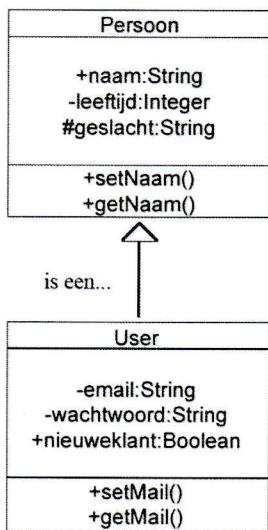
In deze opgave maak je een class-diagram met public, private en protected properties en methodes.



Figuur 5.35

### Generalisatie

De generalisatierelatie is een relatie tussen een algemene class en een specifieke class. De specifieke class bevat aanvullende of specifieke informatie. De specifieke class noemen we een subclass en de algemene class noemen we een superclass. Een generalisatierelatie wordt ook een parent- en child-relatie genoemd. De child-class erft alles van de parent-class, behalve de private methodes en properties.

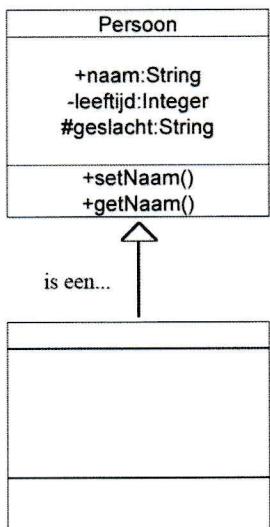


Figuur 5.36

De generalisatierelatie tekenen we met een doorlopende lijn vanuit de specifieke naar de algemene class en met een lege driehoek aan de kant van de algemene class. De naam van de relatie beschrijven we als 'is een...'. In dit geval zeggen we dat de class **User** *is een* **Persoon** en erft alle properties en methodes van de parent-class, behalve de private methodes en properties. De class **User** heeft nog meer specifieke properties en methodes zoals het e-mailadres, het wachtwoordattribuut en de **setMail()**- en **getMail()**-methodes.

- **Opgave 13**

Teken een generalisatierelatie tussen de class **Persoon** en de subclass **Student**. Bedenk drie specifieke properties voor de subclass **Student**.



Figuur 5.37

## 5.6 Abstract classes

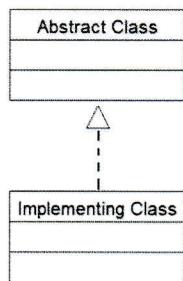
Planning	Inleveren
	Opgave 14

Een abstract class is een class die minstens één abstracte methode heeft. Een abstracte methode heeft alleen de syntaxis van de methode en geen body tussen {}, bijvoorbeeld:

```
public function addToCart(Product $product);
```

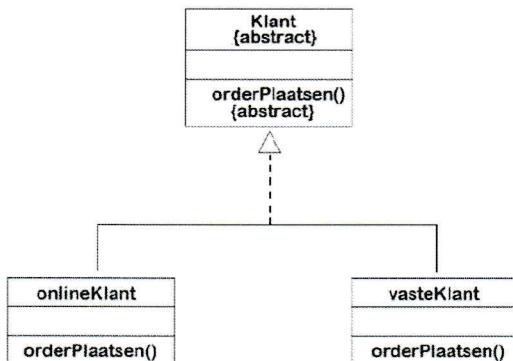
Een abstract class moet geïmplementeerd worden door een andere class. De implementerende class moet dan de abstracte methode implementeren, bijvoorbeeld:

```
public function addToCart(Product $product){
    $this->producten[] = $product;
}
```



Figuur 5.38

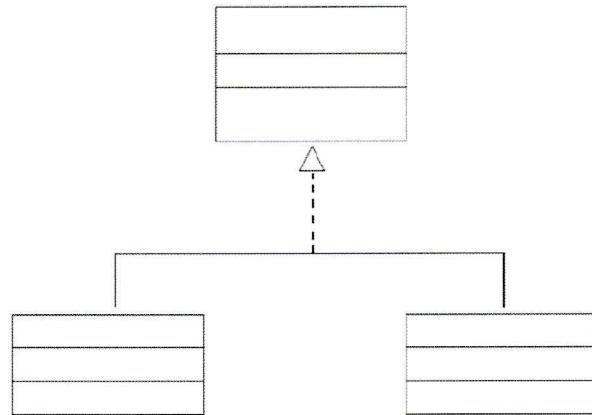
Als een class een of meer abstracte methodes heeft, is de hele class abstract. Een abstracte class krijgt het {abstract}-label naast zijn naam. In de volgende figuur heeft de class Klant de abstracte methode orderPlaatsen(). De implementerende classes moeten zelf de methode orderPlaatsen() implementeren.



Figuur 5.39

- **Opgave 14**

Teken in de volgende figuur de abstracte class Voertuig met twee implementerende classes.

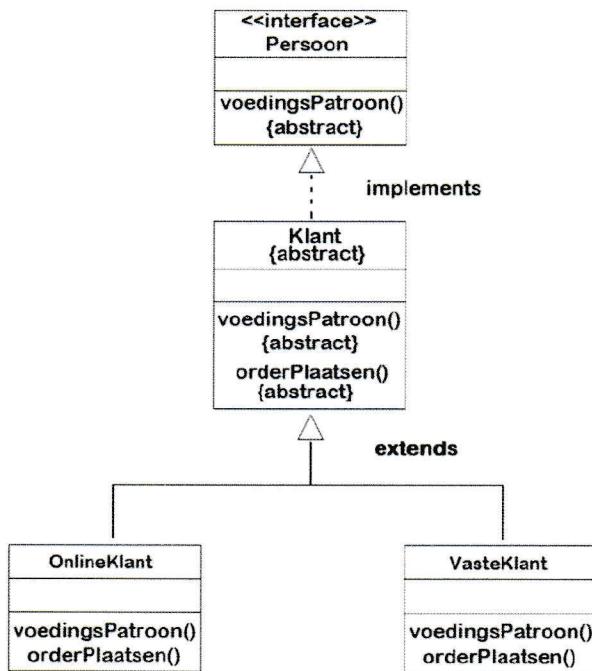


Figuur 5.40

## 5.7 Interfaces en abstracte classes

Planning	Inleveren
	Opgaven 15

Een interface is in essentie een template (sjabloon). Een interface mag de namen van methodes declareren maar mag de methodes zelf niet implementeren. Andere classes mogen de interface implementeren. Een class die een interface implementeert is verantwoordelijk voor het implementeren (coderen) van de methodes in de interface.

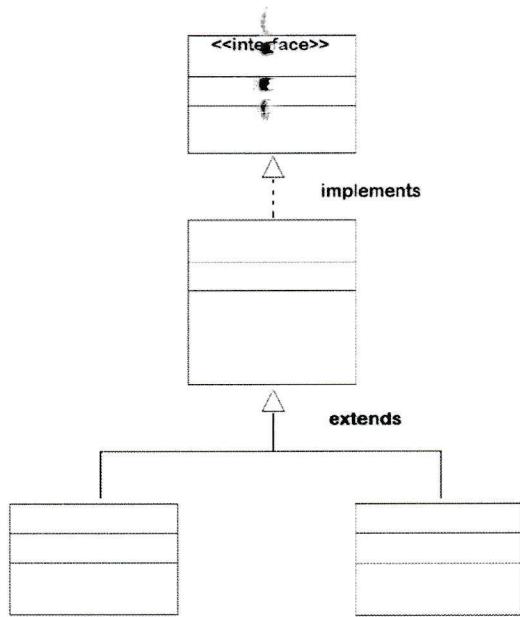


Figuur 5.41 Interfaces

- *Opgave 15*

In deze opgave teken je hieronder:

- De interface Ivoertuig. Bedenk een abstracte methode.
- De abstracte class Voertuig die de interface implementeert. Bedenk een tweede abstracte methode.
- Twee concrete classes die van de abstracte class zijn afgeleid, bijvoorbeeld Auto of Fiets en alle methodes in de interface en in de abstracte class uitvoeren.



Figuur 5.42

## 5.8 Project Openbaar vervoer

Voor deze opdracht maak je de volgende UML-diagrammen voor een openbaarvervoersysteem:

- een use-case-diagram
- een activiteitendiagram
- een sequentiediagram
- een class-diagram

Hier volgt een beschrijving van het ov-systeem:

De reiziger stapt de tram/metro/bus in en scant zijn of haar ov-chipkaart. Het ov-systeem controleert op:

- kaartnummer
- geldigheidsdatum
- voldoende saldo

Als de ov-chipkaart geldig is en voldoende saldo heeft, wordt een reis geregistreerd met de volgende gegevens:

- reisnummer
- kaartnummer
- vervoermiddelnummer
- geldigheidsdatum
- saldo
- beginreisdatum en tijd

- instaphalte
- reisbedrag

Daarna wordt de volgende melding getoond:

[ - Goede reis - ] plus beep.

Anders wordt de volgende melding getoond:

[ - Onvoldoende saldo - ] plus alarm.

Wanneer de bestemming bereikt is scant de reiziger zijn of haar ov-chipkaart. Het ov-systeem registreert:

- uitstaphalte
- eindreisdatum en tijd
- reisbedrag
- nieuw saldo

Het ov-systeem rekent het nieuwe saldo uit. Vervolgens wordt de volgende melding getoond:

[ - Tot ziens - ]

Als de reiziger niet uitcheckt wordt een reis van 10 euro gerekend. Als de reiziger wel uitcheckt wordt een werkelijk reisbedrag uitgerekend gebaseerd op een reis-tabel.

De gegevens van de reis worden in het ov-systeem opgeslagen.

#### **Opdracht 1 - Maak een use-case-diagram**

Maak een use-case-diagram dat gebaseerd is op de beschrijving van het ov-systeem.

#### **Opdracht 2 - Maak een activiteitendiagram**

Maak een activiteitendiagram dat gebaseerd is op het use-case-diagram voor het ov-systeem uit opdracht 1.

#### **Opdracht 3 - Maak een sequentiediagram**

Maak een sequentiediagram dat gebaseerd is op het activiteitendiagram voor het ov-systeem uit opdracht 2.

#### **Opdracht 4 - Maak een classdiagram**

Het vinden van classes binnen een systeem is een kwestie van ervaring en creativiteit. Er bestaan geen methodes of formules om beginnende programmeurs te helpen met het vinden van classes. Het is een designproces waarbij intuïtie maar ook veel oefening een rol spelen.

Hier volgen twee tips bij het zoeken naar classes binnen een systeem:

- Onderstreep alle zelfstandige naamwoorden in de beschrijving van het systeem. Zelfstandige naamwoorden zouden classes kunnen zijn.
- Omcirkel alle werkwoorden in de beschrijving van het systeem. Werkwoorden zouden operaties (methodes) kunnen zijn.

Lees de beschrijving van het ov-systeem weer.

De vraag is: welke van deze zelfstandige naamwoorden zouden classes kunnen zijn en welke niet? Reiziger is zeker een class. Er zijn meerdere reizigers. Maar is instaphalte ook een class? Soms wel en soms niet. In sommige gevallen zou instaphalte een heel uitgebreide class met veel operaties kunnen zijn. In dit geval zeggen we dat instaphalte en uitstaphalte een attribuut van de class reis zijn.

**Opgave:** Maak een class-diagram met alle classes gebaseerd op het ov-systeem.