Het maken en controleren van een UML-analyse

W. van der Ploeg

november 2018
versie 2.0

Inhoudsopgave

Voorwoord

Dit is een handleiding voor het maken, controleren en vergelijken van een beperkte UML-analyse. Er worden een beperkt aantal diagrammen behandeld aan de hand van twee voorbeelden. Die voorbeelden zijn 1: een simpele lift en 2: de registratie van een ontvangst bij een congres. Daarbij wordt vooral in gegaan op de syntactische en inhoudelijke controle van de diagrammen en weinig op de theorie. Ik wil hierbij Melinda de Roo, Wouter Brinksma, Martin Bosgra en Bas van Hensbergen hartelijk bedanken voor hun kritisch meelezen. Het geheel is een 'work in progress' en verre van volmaakt. Daarom wordt iedere student die als eerste een serieuze inhoudelijke fout vindt beloond met 0,5 bonuspunt op het assessment dat bij het vak SON hoort. Immers: degene die snapt waar de fout zit snapt ook het idee achter UML.

Ten opzichte van versie 1.4 zijn zijn er twee hoofdstukken toegevoegd, een over deployment-diagrammen en een over het gebruiken van UML om onderzoek en gemaakte keuzes te kunnen verantwoorden.

Wouter van der Ploeg, november 2018

Hoofdstuk 1

Inleiding

Leeswijzer

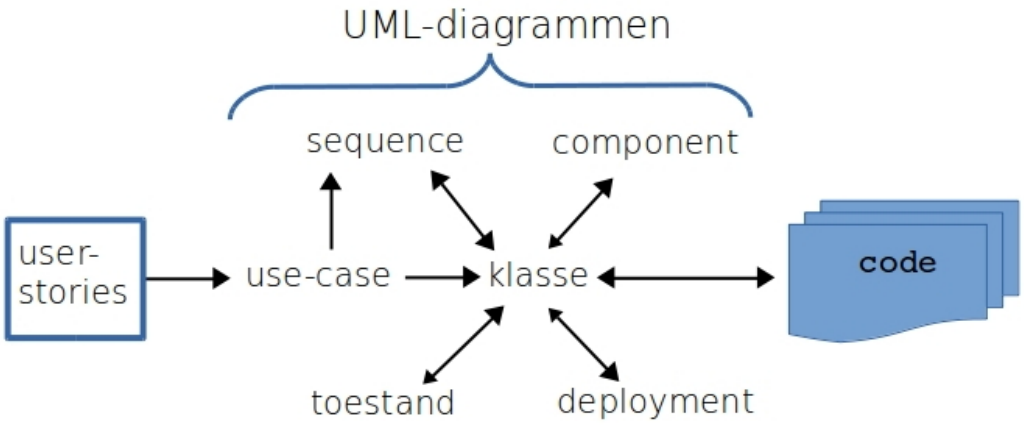
Basis UML-analyse

Deze handleiding geeft handvatten om een UML-analyse te maken en te controleren. UML staat voor Unified Modeling Language, en is dus een taal. Een taal die voor een deel bestaat uit diagrammen. Hoe je die diagrammen moet tekenen vertelt UML niet. Er zijn wel regels voor wat een goed UML-diagram is. In deze handleiding komen 3 zaken aan de orde.

1. hoe je **begint** met een UML-analyse,
2. hoe je diagrammen op **syntax** kunt controleren en
3. hoe je diagrammen **inhoudelijk** kunt controleren.

Een van de mooie dingen van UML is, dat je UML zelf kunt gebruiken om UML uit te leggen. Hiervoor gebruik ik de activiteitendiagrammen van UML. Hoewel user stories niet tot UML gerekend worden sluit de denkwijze erg goed aan bij use-case-diagrammen:hoofdstuk 2 . In de hoofdstukken 3 t/m 7 worden van vijf soorten UML-diagrammen criteria voor de syntaxcontrole en criteria voor de inhoudelijke controle behandeld. Hoofdstuk 3 behandelt use-case-diagrammen, hoofdstuk 4 klassediagrammen, hoofdstuk 5 sequence-diagrammen, hoofdstuk 6 state-transition-diagrammen, hoofdstuk 7 componentdiagrammen en hoofdstuk 8 deployment-diagrammen. In hoofdstuk 9

wordt UML ingezet om onderzoek te verantwoorden. In hoofdstuk 10 vind je activiteitendiagrammen waarin schematisch staat hoe je de voorgaande diagrammen kunt maken.In hoofdstuk 11 tenslotte staat een voorbeeld uitwerking van een administratief systeem. In hoofdstuk 12 staan een aantal tegenvoorbeelden. In hoofdstuk 13 wordt een overzicht gegeven van de andere diagrammen.



Figuur 1.1: Basis UML-analyse, zie ook hoofdstuk 9

Een UML-analyse kun op meerdere mnieren gebruiken: Het maken van een klassediagram waarmee je ten eerste over je code kunt praten met anderen en ten tweede code kunt genereren. Ten derde zijn de UML-diagerammen een uitstekend hulpmiddel bij het doen van onderzoek naar het te ontwerpen systeem. Ten vierde kun je complexe onderdelen van je systeem analyseren met specifieke diagrammen. Omgekeerd kun je met code ook een klassediagram genereren (zie bv figuur 4.8) zodat je over de structuur van je code met collega's kunt communiceren.

Voorbeeldbeschrijving Lift

Een (simpele) lift heeft een deur die open en dicht kan, een bedieningspaneel in de lift met een knop voor elke verdieping, op elke verdieping een paneel met een "haalknop", een motor die de lift op en neer beweegt en op elke verdieping een sensor die aangeeft of de lift daar passeert.

Hoofdstuk 2

User story

Inleiding

User stories worden vooral veel gebruikt in agile ontwikkelmethodes zoals scrum. User stories behoren niet tot de UML-familie, maar kunnen wel als startpunt dienen voor een UML-analyse. Wat is een user-story? Een user-story is een korte simpele beschrijving van een eigenschap [van software] vanuit het perspectief van de persoon die een nieuwe functionaliteit wenst, meestal een gebruiker van het softwaresysteem. (zie ook Mountaingoat 2018 [3]) Ze worden vaak volgens een template of mal geschreven, waarbij de reden niet altijd gegeven hoeft te worden:

> "als *een actor* wil ik *een actie* zodat *een reden*"

Voorbeelden:

- "Als *Flickr-klant* wil ik *verschillende privacy-levels in kunnen stellen bij mijn foto's* zodat *ik kan controleren wie welke foto's van mij ziet.*"
- "Als *gebruiker* wil ik *mijn klanten kunnen zoeken zowel op voornaam als op achternaam.*"

User stories worden vaak op indexkaartjes of 'post-its' geschreven, en verzameld op de muur of op een tafel om een sprint te faciliteren, en om discussie over de gewenste functionalitiet te stimuleren.

Detaillering

Hoe gedetailleerd moet een user-story zijn? Uiteindelijk zo gedetailleerd dat een ontwikkelaar in een paar dagen de gewenste functionaliteit kan realiseren. Een user-story zo goed schrijven lukt niet in één keer natuurlijk. Dus je begint met een globale user-story, en splitst hem net zolang op in kleinere user stories totdat hij in een paar dagen gerealiseerd kan worden.

User stories voor de lift:

1. "Als Passagier wil ik een een knop zodat als ik daar op druk de lift naar mijn verdieping komt en daar automatisch stopt en de deur opendoet."

2. "Als Passagier wil ik in de lift een knop waarmee ik aan kan geven naar welke verdieping ik wil en dat de lift daar automatisch stopt."

	Inhoudelijke criteria user-story
a	Kort (genoeg)?
b	Geformuleerd vanuit een gebruiker?
c	Taalgebruik eenvoudig en concreet?
d	Testbaar en demonstreerbaar?
e	(waarschijnlijk) realiseerbaar in paar dagen?

Tabel 2.1: Inhoudelijke criteria user-story

Hoofdstuk 3

Use-case-diagram

Inleiding

In dit hoofdstuk kijken we naar use-case-diagrammen (UCD). In hoofdstuk 10 (Activiteitendiagram van een use-case-diagram) staat een activiteitendiagram waarin beschreven wordt hoe een use-case-diagram te maken. In dit hoofdstuk bepalen we hoe een use-case-diagram op syntaxfouten te controleren (stap 1 syntaxcontrole) en inhoudelijke fouten (stap 2 vergelijken use-case-digrammen). We nemen als voorbeeld de Lift uit hoofdstuk 2 waarvoor we de verschillende soorten diagrammen gaan maken.

Van user stories naar UCD

In het vorige hoofdstuk hebben we 2 user stories geschreven. Deze user stories nemen we als basis voor het maken van een UCD. Use-cases staan voor de acties die het softwaresysteem moet uitvoeren. Actoren zijn dingen in de buitenwereld die opdrachten geven (de actieve actoren) of opdrachten van de software moeten uitvoeren (de passieve actoren). Een actor is een mens of ding of systeem buiten het systeem dat we analyseren. Als een actor een opdracht geeft aan ons systeem zeggen we dat het systeem en dus de use-case getriggerd wordt door de actor. *Alleen actieve actoren kunnen een use-case triggeren.* Om een use-case te maken nemen we een user-story en plaatsen er de goede actoren bij. Er zijn meerdere manieren waarop we actoren kunnen plaatsen ("er voeren meerdere wegen naar Rome").

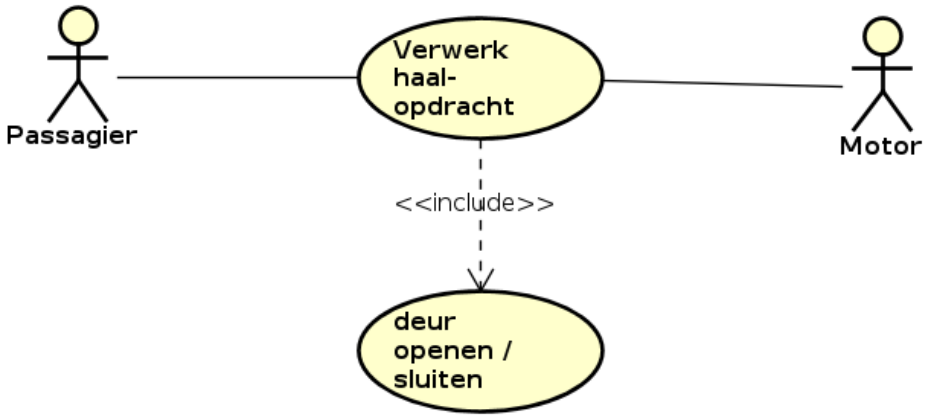
Manier één

User story:
"Als Passagier wil ik een knop zodat als ik daar op druk de lift naar mijn verdieping komt en daar automatisch stopt en de deur opendoet."

Daar moeten we een use-case met actoren van maken. Om van de user-story een stukje van het UCD te maken moeten we 1) bedenken welke actoren een rol spelen 2) de user-story herschrijven vanuit het perspectief van het softwaresysteem.

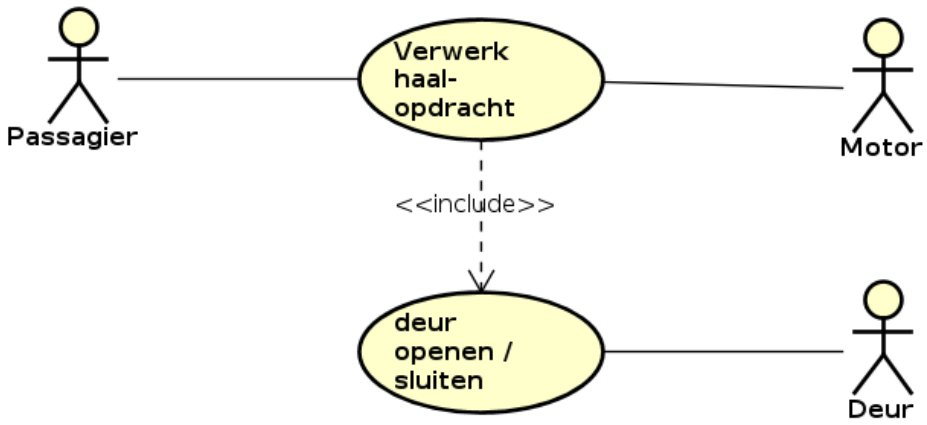
Figuur 3.1: Use-case aan de hand van user-story - poging 1

Merk op dat het *gebruikers*perspectief van de user-story: **"als Passagier wil ik (...)"**, vervangen is door het *systeem*perspectief van de use-case: **"verwerk haalopdracht"**. Het systeem verwerkt een opdracht en kan de motor aan zetten (want er loopt een lijntje naar de actor Motor), maar de deur kan nog niet bestuurd worden. Vandaar de volgende verbetering:



Figuur 3.2: Use-case aan de hand van user-story - poging 2

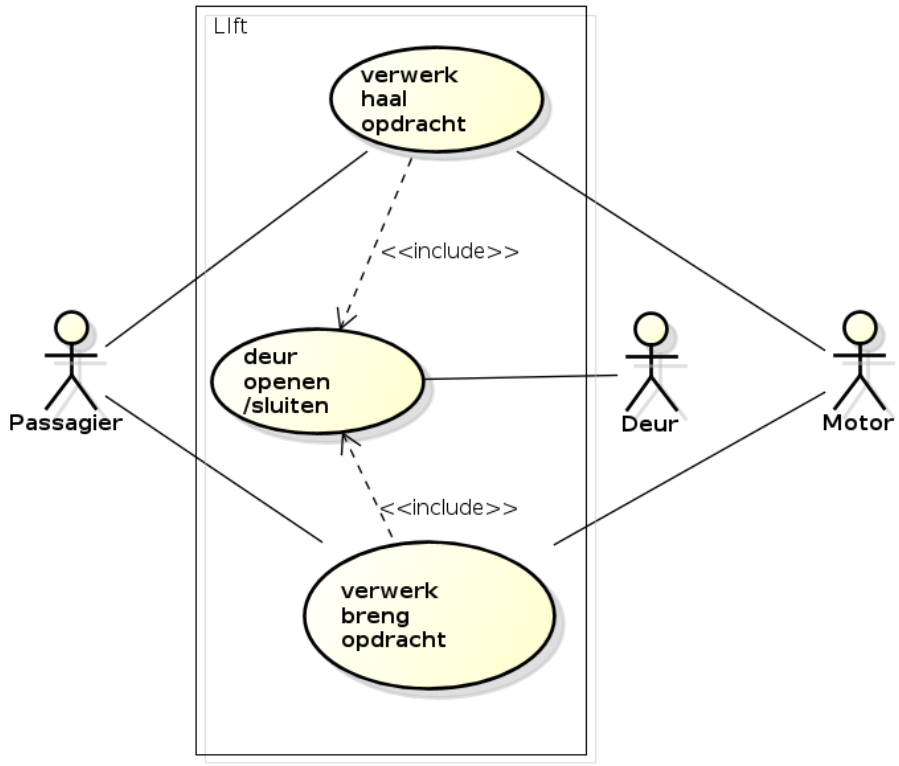
Soms kunnen use-cases acties onderling delen. Hier is bijvoorbeeld de actie (use-case) "deur openen/sluiten" nodig bij het ophalen van de lift en bij het wegbrengen van de lift. Door middel van een ≪include≫ (hetgeen betekent: deze use-case automatisch óók uitvoeren) kun je dat aangeven. Nu hebben we door de ≪include≫ dat de deur als gevolg van de actie van de Passagier (de trigger die de use-case in werking zet) automatisch geopend wordt. Bij figuur 3.2 ontbrak alleen de actor deur nog:



Figuur 3.3: Use-case aan de hand van user-story - poging 3

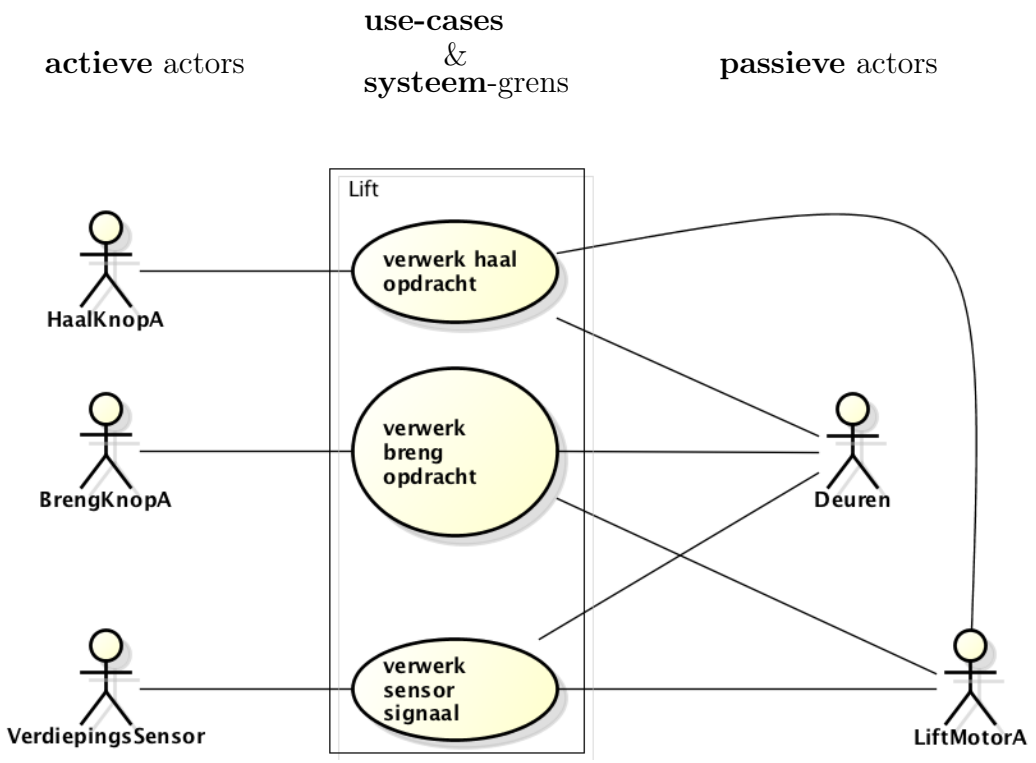
Natuurlijk moeten we use-cases maken voor alle user stories (in ons geval zijn dat er maar 2). En dat leidt tot het UCD in figuur 3.4.

actieve actors | **use-cases** & **systeem**-grens | **passieve** actors



Figuur 3.4: UCD Lift – variant 1

Het is belangrijk om goed aan te geven wat de grenzen zijn tussen het softwaresysteem en de buitenwereld. Hier is de grens aangegeven door middel van een rechthoek en de naam "Lift"



Figuur 3.5: UCD Lift – variant 2

Manier twee

Als je een UCD maakt moet je alles vanuit het perspectief van het systeem formuleren. De vraag hierbij is of het systeem een Passagier "ziet". Het enige wat het (software-) systeem "ziet" is een signaal van de knop waar de passagier op drukt. Dat rechtvaardigt om een totaal ander UCD te maken, waarbij alleen maar fysieke actoren zijn en de passagier geen rol meer speelt. Bijkomend voordeel is dat nu elke use-case z'n eigen trigger/actor heeft. Zie figuur 3.5.

De 'Passagier' uit de user-story is veranderd in de actoren 'Haalknop' en 'Brengknop'. Voor elke user-story moeten we deze omzetting doen. Daarbij hoeft het niet zo te zijn dat elke user-story altijd één use-case oplevert. Het kan nodig zijn om de user-story te splitsen of om twee stories samen te nemen. Bijvoorbeeld is het hier nodig de user stories te splitsen omdat deze

user stories van twee verschillende fysieke actoren signalen krijgen namelijk de haal/brengknop en de verdiepingssensor.

Syntaxcontrole UCD

Hoe weet je welke van de twee varianten beter is? Daar zijn twee soorten criteria voor. Ten eerste moet een diagram aan de syntaxregels voldoen (vergelijk met: 'Les hebben ik ' is syntactisch geen goed Nederlands) Daarnaast is het ook belangrijk om naar de inhoud te kijken, dat gebeurt in de volgende paragraaf. Eerst gaan we kijken of UCD een syntactisch goed is.

Criteria voor de syntaxcontrole van een UCD

(a) Kun je alle opdrachten voor het softwaresysteem terug vinden in de verbindingslijnen tussen de actoren links en één of meer use-cases?

(b) Heeft elke use-case een actieve actor; dat wil zeggen krijgt elke use-case een opdracht van buiten het systeem? Als dat niet zo is moet de use-case op een andere manier geactiveerd worden.

(c) Is elke actie die het softwaresysteem moet uitvoeren terug te vinden in een use-case?

(d) Is elke use-case geformuleerd als actie van het softwaresysteem? (systeemperspectief)?

(e) Zijn alle actoren inderdaad dingen (of mensen of systemen) buiten het softwaresysteem?

(f) Is alle output van het softwaresysteem terug te vinden als lijn tussen een use-case en een actor aan de rechterkant? (soms is een actor èn actief èn passief, dan mag die zowel rechts als links staan)

(g) Is de systeemgrens aangegeven? Heeft het systeem een betekenisvolle naam?

(h) Is de naam van elke actor een zelfstandig naamwoord dat begint met een hoofdletter?

(i) Is er bij elke ≪extend≫ ook een extension point gedefinieerd in de use-case die ge-≪extend≫ wordt?

(j) Wordt de ge-≪include≫ use-case door meer dan één use-cases ge-≪include≫ ? (anders kan hij samengevoegd worden met de aanroepende use-case)

	Syntaxcriteria use-case-diagram
a	alle input terug te vinden?
b	actieve actor bij elke use-case?
c	elke user-story gedekt?
d	systeemperspectief?
e	actoren buiten systeem?
f	is output terug te vinden?
g	systeemgrens?
h	goede namen actoren?
i	extension point?
j	≪include≫ alleen bij meerdere use-cases?

Tabel 3.1: Syntaxcriteria use-case-diagram

Natuurlijk is het goed om van de verschillende varianten van een diagram de goede dingen samen te nemen.

Inhoudelijke controle UCD

Criteria voor de inhoudelijke controle van use-case diagrammen:

(a) Kun je elke use-case uitleggen aan een niet informaticus (bijvoorbeeld de productowner)?

(b) Is het aantal use-cases niet meer dan 7?

(c) Kun je elke use-case koppelen aan een user-story en komen alle user stories aan bod in een use-case?

(d) Is elke ≪include≫ en ≪extend≫ noodzakelijk en/of nuttig?

(e) In hoeverre klopt de opsplitsing van acties in use-cases met de werkelijkheid?

(f) Heeft elke actor interactie met de software?

(g) Is er geen interactie tussen actoren? (behalve overerving tussen actoren mag dat niet want een actor hoort tot de buitenwereld waar het systeem geen zeggenschap heeft)

(h) Is het systeem klein genoeg? Ga je alles binnen de systeemgrens inderdaad programmeren?

	Inhoudelijke criteria use-case-diagram
a	uitleggen?
b	#UC <= 7?
c	user-story koppeling?
d	include / extend nodig?
e	reële opsplitsing?
f	interactie actor-software?
g	geen interactie actoren?
h	is het systeem klein genoeg?

Tabel 3.2: Inhoudelijke criteria use-case-diagram

Hoofdstuk 4

Klassediagram

Inleiding

Hier volgt een basisrecept voor het maken van een klassediagram (KD). Dat betekent niet dat dit recept voor alle situaties goed is, maar wel dat als je niet weet welke klassen een rol gaan spelen in je programma, dit recept je op ideeën kan brengen. Er bestaan veel soorten klassen. Drie van de basissoorten (ook wel stereotypen genoemd)zijn: 1. Boundary-klassen (klassen die zich "aan de grens van het systeem bevinden", anders gezegd klassen die opdracht geven aan of opdracht krijgen van actoren), 2. Controller-klassen (klassen die vooral een controlerende / coördinerende rol hebben) en 3. Data- of Entity-klassen (klassen die vooral data bevatten). Het kan handig zijn om in het begin van je analyse een klassediagram te maken met deze drie soorten om een beter overzicht te krijgen. Alle drie kennen ze naast het algemene symbool voor een klasse ook hun eigen symbool.

Figuur 4.1: Symbolen voor basissoorten

Het basis recept voor het maken van een KD:

1. Maak voor elke actor uit het UCD een (boundary-)klasse. Dat wil zeggen een klasse die kan communiceren met een apparaat, mens of systeem in de buitenwereld.

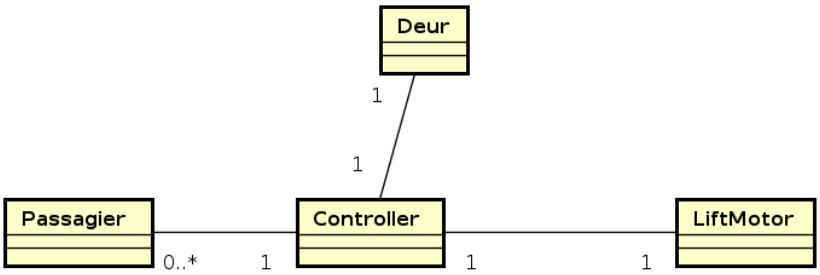
2. Maak een controller-klasse die zelf geen acties uitvoert maar alles coördineert

3. Maak indien nodig één of meerdere data-klassen (bij kleine real-time systemen kun je meestal zonder dataklassen).

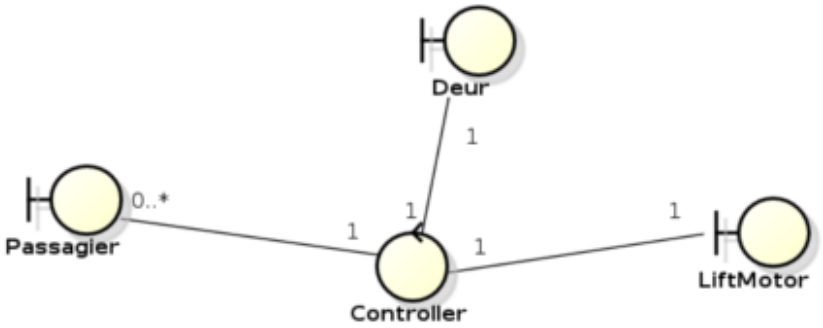
4. Verbind alle klassen met de contoller-klasse dmv een associatie.

5. Corrigeer het KD met behulp van je gezonde verstand.

Op grond van de twee use-case-diagrammen uit hoofdstuk 3, maken we twee klassediagrammen voor de lift.

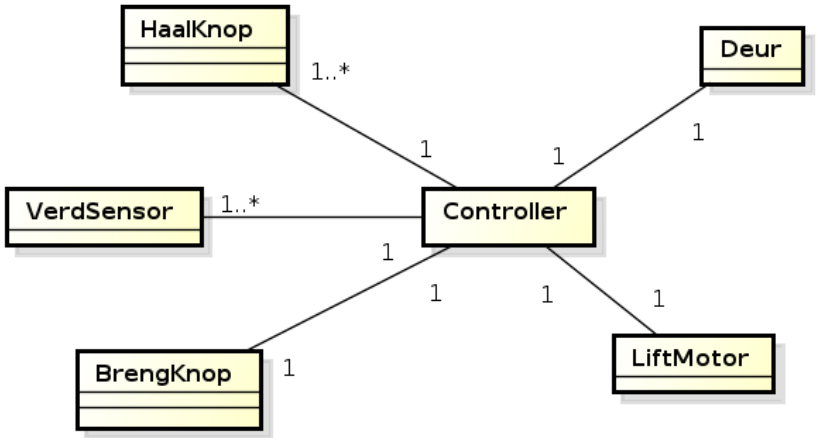


Figuur 4.2: KD Lift variant 1, kaal, 'normaal'

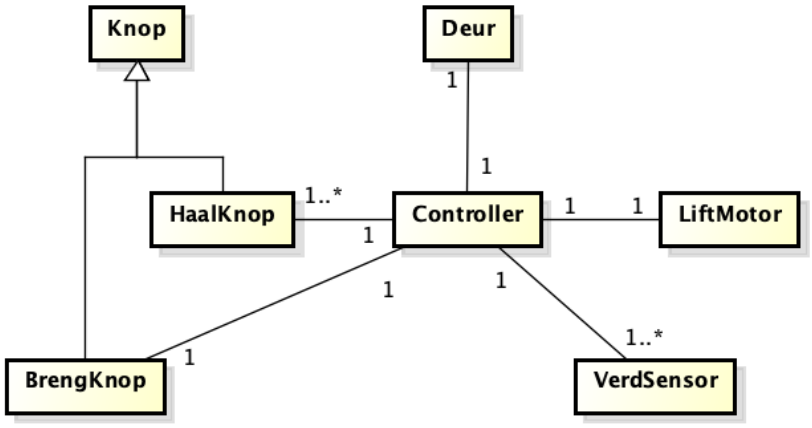


Figuur 4.3: KD Lift variant 1, kaal, met stereotypen

Het KD met stereotypen (figuur 4.3)is hetzelfde KD als het KD met rechthoeken (figuur 4.2), alleen anders getekend, de klassen en hun associaties zijn precies hetzelfde. Het KD tekenen met symbolen lijkt aardig, maar je kunt de methoden en attributen dan niet laten zien. Dus over het algemeen wordt dat alleen voor het maken van grotere overzichten gebruikt. NB: Het cijfertje 1 bij de klasse Deur betekent dat er precies één deur in het hele systeem zit. Als je er vanuit gaat dat er op elke verdieping een deur aanwezig is zou er een * of 1..* bij moeten staan.



Figuur 4.4: KD Lift-kaal variant 2



Figuur 4.5: KD Lift-kaal variant 3

Hoe kunnen we vaststellen welk KD van de drie varianten (figuur 4.2, 4.4 of 4.5) beter is? Net zoals bij use-case-diagrammenhebben we 2 soorten criteria. De eerste om te kijken of de syntax van het KD klopt, de tweede om de klassediagrammen met elkaar te vergelijken. Maar voor we dat kunnen doen moeten we de klassen vullen met methoden en kijken we even naar codegeneratie.

Code genereren

Met een UML-tool kun je nu al code genereren (in Astah is dat bv: Tool → Java → export java). Natuurlijk is er nu nog weinig code, we zijn nog maar nauwelijks begonnen:

```
// gegenereerde code van Controller variant 1
public class Controller {
        private Deur deur;
        private LiftMotor liftMotor;
        private Passagier passagier; }
```

Definiëren methoden

Om syntaxcontrole op een KD te kunnen toepassen moeten we eerst weten welke methoden een klasse heeft. Er drie manieren om te achterhalen welke methoden een klasse zou moeten hebben:

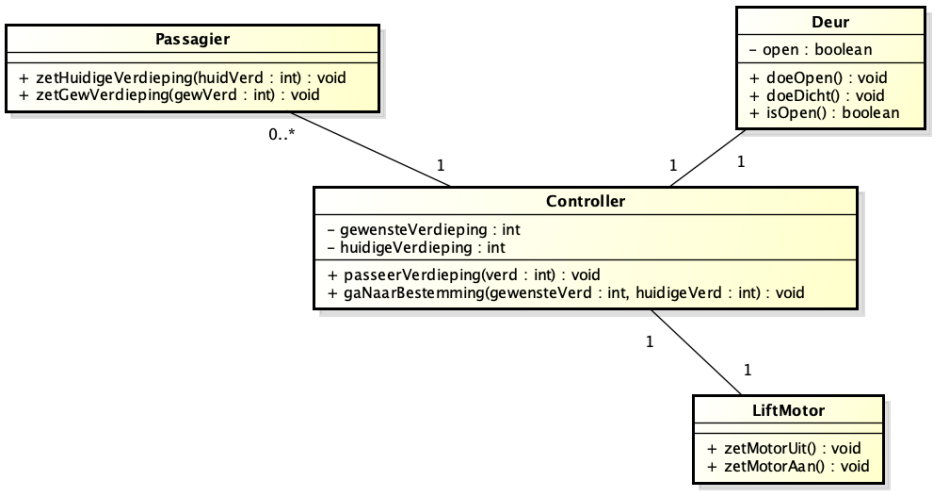
1. met logica
2. met sequence-diagrammen en
3. met state-transition-diagrammen

Het is belangrijk dat je alle drie manieren leert zodat je niet voor één gat te vangen bent. Waarschijnlijk zul je vaak manier 1 toepassen. Waarschijnlijk heb je hem al vaak toegepast. Maar soms, in wat ingewikkelder situaties zijn methode 2 en 3 zeer waardevol. Manier 2, met sequence-diagrammen wordt in hoofdstuk 5 en hoofdstuk 10 (activiteitendiagram) uitgelegd, manier 3, met state-transition-diagrammen, in hoofdstuk 6 en hoofdstuk 10 (activiteitendiagram).

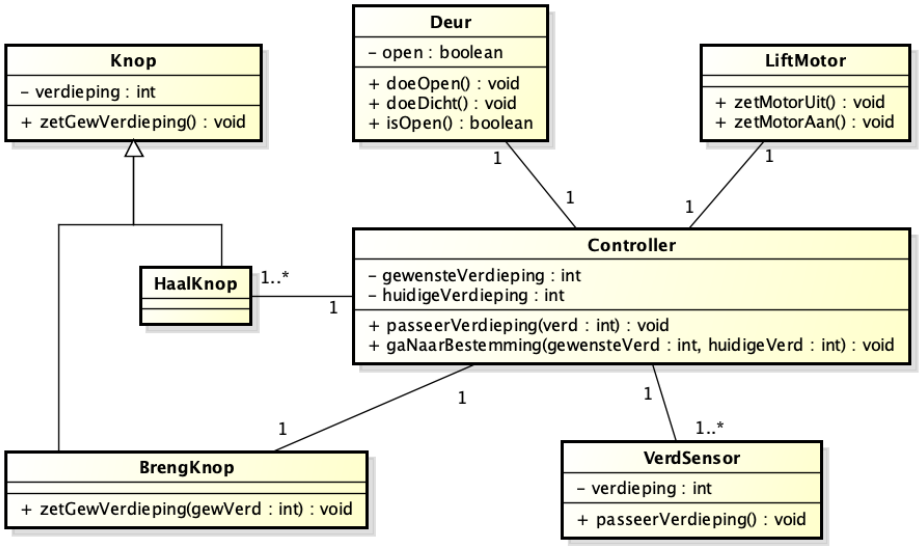
manier 1, met logica

Maar soms is het niet moeilijk om met behulp van logica een methode te bedenken. Bijvoorbeeld bij de lift: De software moet de motor van de actor Lift aan kunnen zetten. Daarom moet er in de klasse LiftMotor een methode `+zetMotorAan():void` aanwezig zijn die een signaal aan de actor Lift geeft. Dat betekent dat de klasse Controller de methode `lMotor.zetMotorAan():void` kan aanroepen, dat wil zeggen het object controller geeft opdracht aan het object `lMotor` om de fysieke motor (= actor) aan te zetten. Het *object* controller is een instantiatie van de *klasse* Controller en het *object* lMotor is een instantiatie van de *klasse* LiftMotor:

```
// met logica achterhaalde, niet-gegenereerde code:
public class Controller {
        private LiftMotor lMotor;
        // lMotor is een attribuut van Controller
        // omdat er in het KD een associatie
        // is tussen LiftMotor en Controller
        public void gaNaarBestemming(
                gewensteVerd:int, huidigeVerd:int) {
                ...
                lMotor.zetMotorAan()
                // public methode van LMotor
                ...
        }
        ...
}
```



Figuur 4.6: KD Lift gevuld variant 1



Figuur 4.7: KD Lift gevuld variant 2

Voor **manieren 2 en 3 om methoden** te definiëren zie hoofdstuk 5 en hoofdstuk 6. Stel dat we alle manieren hebben toegepast en de methoden van de klassen gedefiniëerd hebben. Dan hebben we 2 varianten van het KD. Zie figuur 4.6 en 4.7.

Syntaxcontrole KD

Ook hier gaan we weer syntaxcontrole toepassen om het beste KD te bepalen.

Criteria voor de correctheid van een KD

(a) Zijn alle actoren uit het UCD terug te vinden als (boundary-) klasse? En omgekeerd?

(b) Is elke klasse in tenminste 1 use-case actief?

(c) Zijn er use-cases waarin geen enkele klasse actief is? (dan is je UCD niet goed)

(d) Kun je alles wat in een use-case gebeurt (user-story) terug vinden in de methoden van de klassen?

(e) Worden alle acties die nodig zijn in het systeem gedekt door methoden uit de klassen?

(f) Bestaat er voor elke associatie in het klassediagram ook een aanroep van een methode uit de ene klasse door de andere klasse?

(g) Is er voor elke aanroep van een methode uit de ene klasse door een andere klasse een associatie tussen die 2 klassen?

(h) Kloppen de multipliciteiten met de werkelijkheid?

	Syntaxcriteria klassediagram
a	actor $\rightarrow$ boundary?
b	klasse $\rightarrow$ use-case?
c	alle use-cases gedekt?
d	user-story $\rightarrow$ methode?
e	elke actie door methode gedekt?
f	associatie $\rightarrow$ aanroep?
g	aanroep $\rightarrow$ associatie ?
h	multipliciteiten ?

Tabel 4.1: Syntaxcriteria klassediagram

Inhoudelijke controle KD

Hier volgt een lijst van criteria voor de inhoudelijke controle van een KD

(a) Vormen de klassen logische eenheden? (kun je aan een collega uitleggen waarom de klasse is zoals-ie-is?)

(b) Zijn ze niet te groot en niet te klein? Wanner een klasse te groot of te klein is, is niet in het algemeen te zeggen, het hangt van de situatie af.

(c) Zijn er aparte dataklassen nodig?

(d) Worden die dataklassen gecontroleerd door de controller-klasse of is er een externe database nodig? (en komt die dan voor in het use-case-diagram?)

(e) Als er geen aparte dataklassen nodig zijn, in welke klasse bewaar je dan de benodigde gegevens?

(f) Kun je aan een collega uitleggen wat de activiteiten van een klasse in de use-cases zijn?

(g) Maak je nuttig gebruik van generalisaties?

(h) Weerspiegelt het KD de probleemstelling?

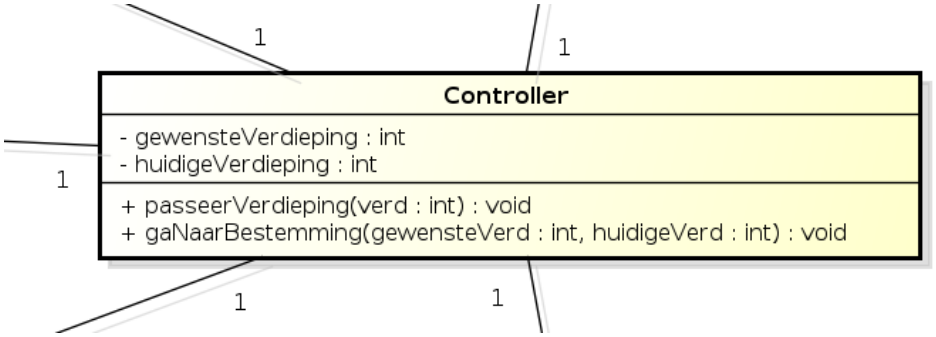
(i) Produceren de klassen goede code?

	Inhoudelijke criteria klassediagram
a	logische eenheden?
b	hebben de klassen de goede grootte?
c	data-klassen nodig?
d	externe database?
e	in welke klasse gegevens?
f	uitleg aan collega?
g	nuttige generalisatie ?
h	goede code ?
i	weerspiegeling probleem ?

Tabel 4.2: Inhoudelijke criteria klassediagram

Code van de Controller

Op een gegeven moment kun je al het werk dat je in de UML-analyse verricht ook gebruiken om code te genereren. Neem bijvoorbeeld de klasse Controller:



Figuur 4.8: De controller-klasse van de lift

De automatisch gegenereerde code hiervan is:

```
public class Controller {
        private int gewensteVerdieping;
        private int huidigeVerdieping;
        private VerdSensor verdSensor;
        private LiftMotor lMotor;
        private Deur deur;
        private HaalKnop haalKnop;
        private BrengKnop brengKnop;
        public void passeerVerdieping(int verd) {
         // body aangevuld mbv SD
         this.huidigeVerdieping = verd
          if (verd == this.gewensteVerdieping):{
           motor.zetMotorUit()
           deur.doeOpen()
           }
        }
        public void gaNaarBestemming(int gewensteVerd,
                                     int huidigeVerd)
                {
                }
}
```

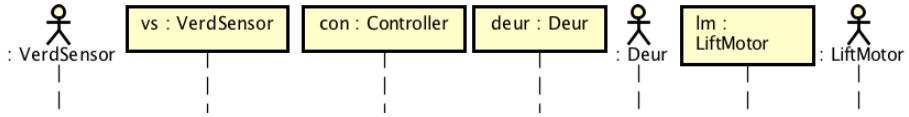
Hoofdstuk 5

Sequence-diagram

Inleiding

Een sequence-diagram (SD) is een uitwerking van één use-case. Een SD toont in een schema hoe de verschillende objecten door middel van het uitwisselen van messages (methode-aanroepen) ervoor zorgen dat de actie in een use-case daadwerkelijk gerealiseerd wordt. In tegenstelling tot de andere diagrammen is het niet nodig om 2 of meer varianten te maken, omdat een SD zelf de uitwerking van een use-case (van een variant van een UCD) is. Dat betekent dat we bij een SD alleen op syntax controleren. Het maken van een sequence-diagram (SD) moet je doen op het moment dat je nog niet weet met welke methoden een klasse zorgt dat een use-case uitgevoerd wordt. Of als controle op de uitwerking van een use-case. Het is daarbij belangrijk dat je UCD en KD al bestaan. Want je moet voor één use-case één SD maken met behulp van de actoren uit het UCD en de klassen uit het KD.

Om een SD van een use-case te maken heb je bestaande actoren en klassen nodig want die sleep je in een UML-tool in het SD. Als je het recept hebt gevolgd uit hoofdstuk 4 dan heb je in het SD de actoren en de boundary-klassen nodig die verbonden zijn met de gekozen use-case en tenminste de controller klasse. Bij een administratief systeem ook nog data (=entity)klassen. In het voorbeeld Lift zijn '*vs*', '*deur*' en '*lm*' boundary-objecten, of, anders gezegd, instantiaties van boundary-klassen. Een eerste versie van het SD "verwerk sensor signaal" is dan figuur 5.1:

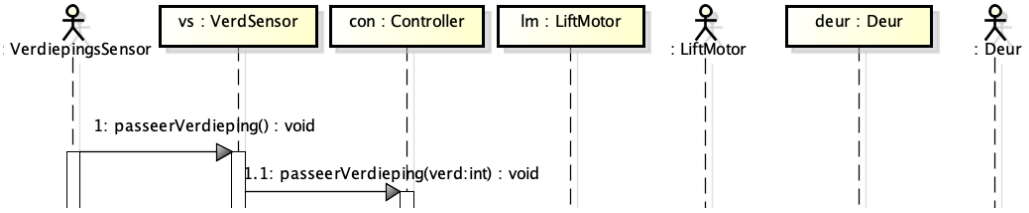


Figuur 5.1: SD verwerk sensor signaal - fragment 1

De actoren en benodigde klassen zijn al aangemaakt in het UML-tool dat je gebruikt. Ieder UML-tool heeft een structuur-window. Daaruit sleep je ze in het SD. Een goede UML-tool maakt van de binnengesleepte klassen automatisch objecten Dat is zichtbaar door de dubbele punt vóór de klassenaam. Als je wilt kun je aan objecten namen geven zoals bijvoorbeeld de naam 'vs' voor het eerste object. Maar dat is niet verplicht.

Figuur 5.2: Als figuur 5.1maar met stereotypen

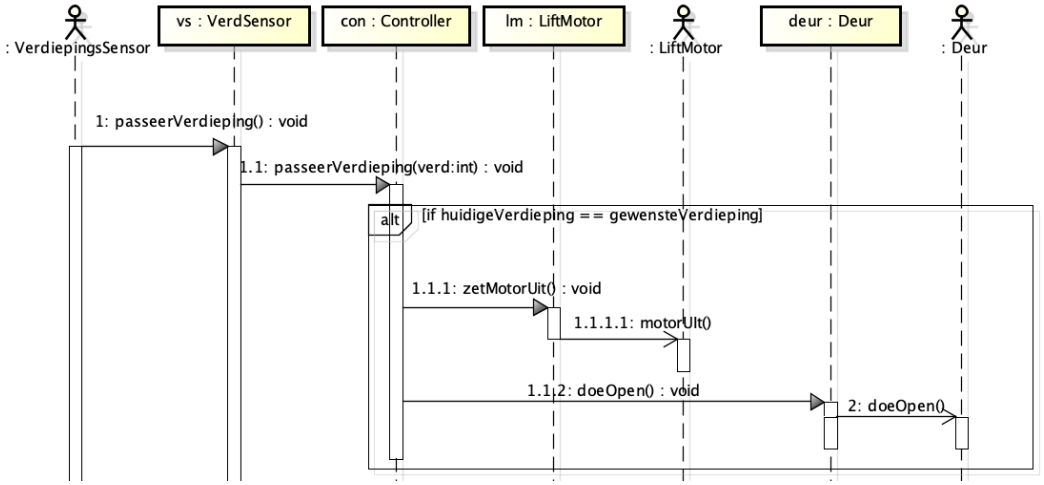
Vervolgens ga je bedenken welke methode het *object* vs (van de klasse VerdSensor) moet hebben die door de *actor* Verdiepingssensor aangeroepen kan worden als de lift een verdieping passeert. Hier lijkt `passeerVerdieping(): void` een goede naam. Vervolgens ga je kijken welke methode in het *object* con (instantiatie van de klasse Controller) aangeroepen moet worden door het *object* vs. Het lijkt logisch om die ook `passeerVerdieping(verd: int): void` te noemen. Maar nu met de parameter `verd:int` erbij omdat de controller moet weten welke verdieping gepasseerd is. Dat betekent dus ook dat de klasse VerdSensor een attribuut `verdieping:int` moet hebben. Dat levert het volgende SD-fragment op:



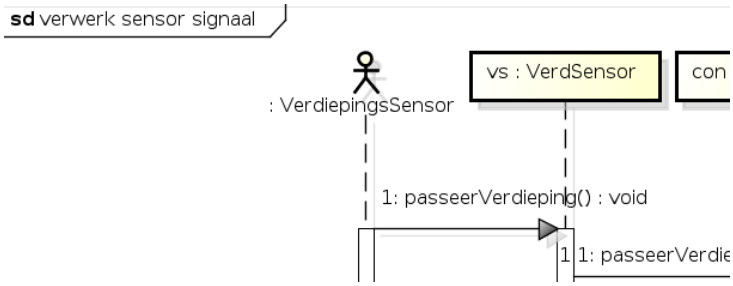
Figuur 5.3: SD verwerk sensor signaal fragment 2

NB: Als je het goed doet worden de nieuw gedefinieerde methoden "automatisch" in de klasse opgenomen. Hoe dat moet hangt af van de UML-

tool die je gebruikt. (Of, je kunt bij het tekenen van een methode in het SD kiezen uit de al gedefinieerde methoden van een klasse). Om nu de controller te laten beslissen of de lift al op de goede verdieping is aangekomen (en de lift te laten stoppen), heeft de controller twee attributen nodig `huidigeVerdieping:int` en `gewensteVerdieping:int`. Als die twee aan elkaar gelijk zijn is de bestemming bereikt, moet de lift stoppen en de deur open. Dit wordt aangegeven in een zogenoemde 'alt'-rechthoek, dat wil zeggen dat alles in de rechthoek wordt alleen uitgevoerd als aan de guard of conditie `huidigeVerdieping == gewensteVerdieping` voldaan is. Het kan en mag ook zonder alt-rechthoek. Zie daarvoor figuur **??** in hoofdstuk 11.



Figuur 5.4: SD verwerk sensor signaal



Figuur 5.5: Detail van figuur 5.4

In de linkerbovenhoek van figuur 5.5 zie je de naam van het SD staan.

Het is een goede gewoonte het SD de naam te geven van de use-case waar het SD de uitwerking van is.

Resultaat van het SD

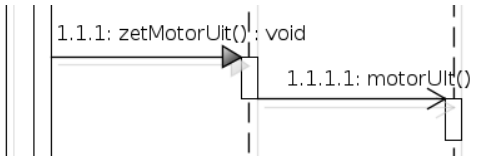
Het resultaat van dit SD is dat we 4 methoden hebben gedefinieerd:

```
VerdSensor.passeerVerdieping():void
Controller.passeerVerdieping(verd:int):void
LiftMotor.zetMotorUit():void
Deur.doeOpen():void
```

Daarnaast zijn er drie attributen gevonden, n.l.

```
Controller.gewensteVerdeping:int
Controller.huidigeVerdieping:int
VerdSensor.verdieping:int
```

En er zijn twee activeringen van actoren gedefiniëerd, nl: motorUit bij de actor MotorA en doeOpen bij de actor DeurA. Let op het verschil in pijlpunten tussen de aanroep van methoden (**synchroon**, de pijlpunt is gevuld en gesloten) en de activeringen van actoren (**asynchroon**, de pijlpunt is open).



Figuur 5.6: Synchrone en asynchrone messages

Daarnaast hebben we de invulling van de methode `Controller.passageVerd(verd:int)`, want we zien in het SD wat deze methode moet doen:

Code

```
public class Controller {
  private LiftMotor lMotor;
  private Deur deur;
  private int gewensteVerdeping;
  private int huidigeVerdieping;
  . . .
  public void passeerVerdieping(int verd) {
    this.huidigeVerdieping = verd
    if  (verd == this.gewensteVerdieping):
      {
        lMotor.zetMotorUit()
        deur.doeOpen()
      }
   }
}
```

Syntaxcontrole SD

Ook voor het sequence-diagram hebben we een syntaxcontrole natuurlijk.

(a) Begint het sequence-diagram met een actor?

(b) Wordt de actie uit de bijbehorende use-case volledig gedekt door de methoden in het sequence-diagram?

(c) Is elk object in het sequence-diagram een instantiatie van een klasse uit het klassediagram?

(d) Kan de aanroep van elke methode uitgevoerd worden met data aanwezig in het object dat de aanroep doet?

(e) Komen alle actoren die verbonden zijn met de use-case van het sequence-diagram in het sequence-diagram voor?

(f) Als er meerdere mogelijkheden (scenario's) voor het verloop van de actie in een use-case zijn, zie je dat dan terug in het sequence-diagram?

(g) Is het mogelijk om het sequence-diagram leesbaar af te drukken op 1 A4-tje? (als dat niet kan is het sequence-diagram te groot en moet het sequence-diagram en de bijbehorende use-case gesplitst worden)

(h) Zijn de de messages naar actoren inderdaad asynchroon, en de andere synchroon?

(i) Zijn de namen van de methoden en attributen betekenisvol en begrijpelijk?

(j) Zijn er parameters gedefinieerd in de methoden? En klopt hun type?

(k) Is het return-type van elke methode goed gekozen?

(l) Heeft het SD de naam van de use-case?

	Criteria sequence-diagram
a	begint SD met actor?
b	use-case gedekt door methoden?
c	komen alle objecten uit het KD?
d	beschikt object over benodigde data?
e	zijn alle actoren aanwezig in SD?
f	komen scenario's terug in SD?
g	leesbaar op A4 ?
h	synchroon - asynchroon?
i	namen goed ?
j	parameters?
k	return type?
l	naam van het SD?

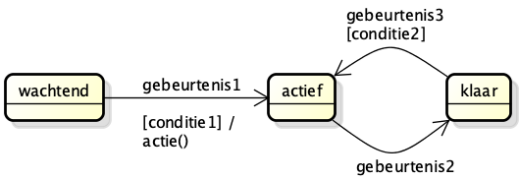
Tabel 5.1: Criteria sequence-diagram

Hoofdstuk 6

Toestandsdiagram

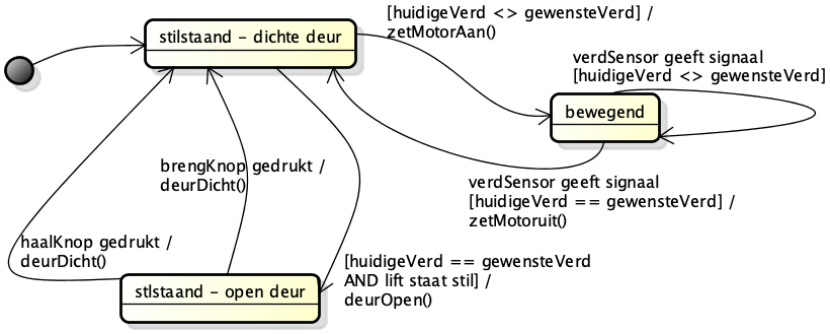
Inleiding

Een STD (State Transition Diagram) beschrijft het gedrag van een object of een systeem door aan te geven welke **gebeurtenis** welke **toestandsverandering** veroorzaakt. Het schema (activiteitendiagram) voor het maken van een state-transition-diagram (STD) staat in hoofdstuk 10. Andere namen voor een STD zijn toestandsdiagram, state-machine-diagram, state-chart en eindige automaat. Het doel van het maken van een STD is het achterhalen of definiëren welke methoden een klasse nodig heeft en wat de flow van code wordt (in welke volgorde de aanroepen plaatsvinden). Kenmerk van een toestand is dat "als er niets gebeurt" de toestand niet verandert. Anders gezegd om van de ene toestand in de andere te komen moet er een gebeurtenis plaatsvinden, nog weer anders gezegd: de toestandsverandering moet getriggerd worden. Er zijn 3 dingen die je bij een toestandsverandering kunt vermelden. Het is verplicht om tenminste een gebeurtenis of een conditie (de voorwaarde waaronder de overgang plaatsvindt) te vermelden. De actie is de methode die het systeem moet uitvoeren.



Figuur 6.1: Toestanden met toestandsovergangen

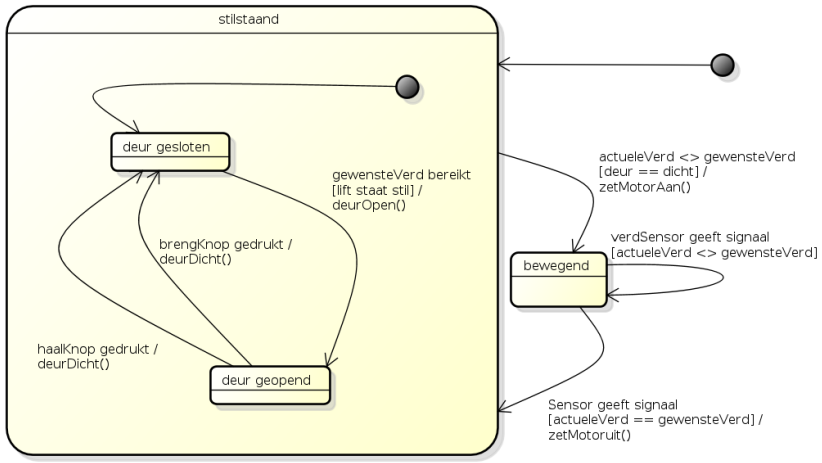
Ook voor het maken van een STD geldt dat er niet (altijd) precies één goede uitwerking is. Je moet dus ook hier zelf beoordelen of jouw uitwerking voldoet aan de eisen die de klant stelt en de eisen die een goede analyse stelt. Daarom geef ik ook hier 2 uitwerkingen van hetzelfde probleem.



Figuur 6.2: STD Lift variant 1

Toelichting figuur 6.2: De lift kan in 3 verschillende toestanden zijn: 1. *bewegend*, 2. *stilstaand – open deur*, 3.*stilstaand – dichte deur*.

De gebeurtenissen zijn hier bv "de gewenste verdiepng is bereikt" of "de brengknop is gedrukt". Er is natuurlijk een sterk verband met user stories! Soms moet er aan een conditie voldaan worden, bijvoorbeeld: huidigeVerd <> gewensteVerd].



Figuur 6.3: STD Lift variant 2

Toelichting figuur 6.3: Bij het vorige STD (figuur 6.2) zagen we twee toestanden stilstaand – open deur, stilstaand – dichte deur, die veel gemeenschappelijk hebben. Daar kun je een supertoestand – stilstaand – van maken die subtoestanden kan bevatten, namelijk deur geopend - deur gesloten.

Figuur 6.4: Uit te voeren methodes *tijdens* een toestand

Syntaxcontrole STD

(a) Is er een begintoestand getekend?

(b) Zijn de namen van de toestanden goed (dus geen acties)?

(c) Heeft elke toestand een uitgang (geen zwarte gaten)?

(d) Kun je alle toestanden vanaf het begin bereiken (geen supernova's)?

(e) Is er bij elke transitie tenminste een gebeurtenis of conditie ingevuld?

(f) Zijn er bij 2 of meer uitgaande transities onderscheidende gebeurtenissen of condities bij de transities vermeld?

(g) Zijn de condities testbaar geformuleerd (bv met == of $\neq$)?

(h) Zijn de triggers werkelijke gebeurtenissen?

(i) Zijn er gebeurtenissen die niet in het STD staan? Klopt dat?

	Syntaxcriteria toestandsdiagram
a	is er een begintoestand?
b	goede namen toestanden?
c	geen zwarte gaten?
d	geen supernova's?
e	gebeurtenis of conditie per transitie?
f	kun je uitgaande transities onderscheiden?
g	testbare condities?
h	zijn de triggers werkelijke gebeurtenissen?
i	dekt het STD alle gebeurtenissen ?

Tabel 6.1: Syntaxcriteria toestandsdiagram

Inhoudelijke controle STD

(a) Zijn alle toestanden werkelijk bestaande toestanden?

(b) Zijn de toestanden (zo veel mogelijk) complementair?

(c) Maak je het voor een real-time systeem, of een object?

(d) Levert het STD (nieuwe) methoden op?

(e) Kun je elke methode toewijzen aan een klasse?

(f) Kun je uit de condities bij de transities attributen van klassen afleiden?

	Inhoudelijke criteria toestandsdiagram
a	werkelijk bestaande toestanden?
b	toestanden complementair?
c	object of real-time STD?
d	nieuwe methoden?
e	elke methode toegewezen aan klasse?
f	leveren condities nieuwe attributen op?

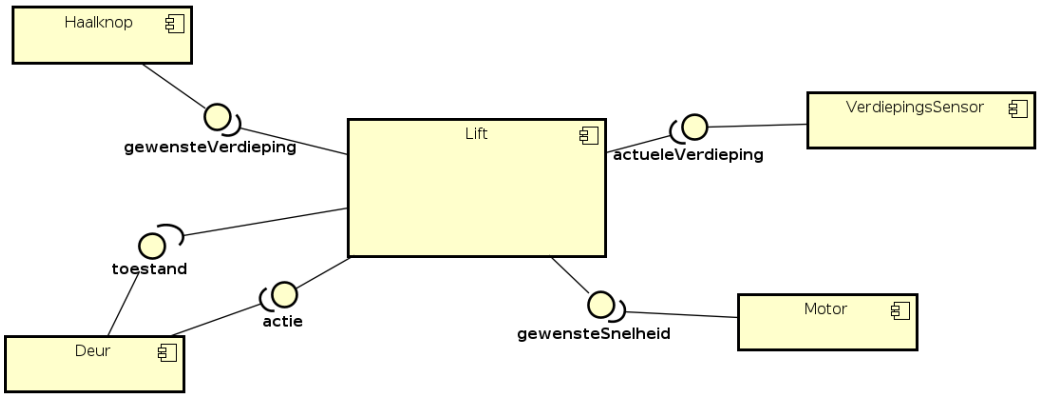
Tabel 6.2: Inhoudelijke criteria toestandsdiagram

Hoofdstuk 7

Componentdiagram

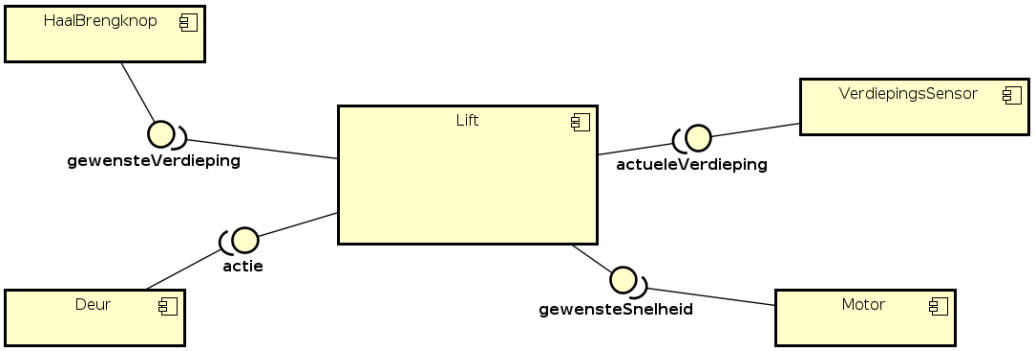
Inleiding

Het schema (activiteitendiagram) voor het maken van een componentdiagram (CD) staat in hoofdstuk 10. Het doel van het maken van een CD is het opdelen van software in logische eenheden en daarbij de verbanden tussen de onderdelen duidelijk maken. Het gaat hierbij om vooral grotere stukken software die ontwikkeld worden door meerdere mensen, of om situaties waarin je stukjes software kunt koppelen aan fysieke onderdelen zoals sensoren. Ook voor het maken van een CD geldt dat er niet precies één goede uitwerking is. Je moet dus ook hier zelf beoordelen of jouw uitwerking voldoet aan de eisen die de klant stelt, de eisen die je collega-software-ontwikkelaar stelt en de eisen die een goede analyse stelt. Daarom ook hier weer 2 uitwerkingen van hetzelfde probleem.



Figuur 7.1: CD Lift variant 1

Als het goed is valt je meteen op dat door de componenten en hun interfaces te benoemen je gedwongen wordt om concreet te definiëren welke parameters door de componenten uitgewisseld worden en welke component de interface levert en welke component ze nodig heeft.



Figuur 7.2: CD Lift variant 2

De verschillen van variant 2 t.o.v. variant 1 zijn dat ten eerste nu ook de Brengknop als extern 'ding' gezien wordt, niet behorend tot de lift zelf en ten tweede de component Deur op dezelfde manier gezien wordt als Motor. Namelijk als iets waar je alleen opdrachten naar toestuurt en niet iets waar je de toestand van op hoeft te vragen omdat je als je zelf de gegeven opdrachten bijhoudt natuurlijk weet in welke toestand (open of dicht) de deur is. Ook bij componentdiagrammen willen we natuurlijk kunnen controleren of ze kloppen:

Syntaxcontrole CD

Criteria voor de correctheid van een CD

(a) Zijn alle componenten met tenminste één andere component verbonden?

(b) Is er voor elke benodigde interface ook een geleverde interface?

(c) Hebben alle componenten een begrijpelijk naam?

(d) Heeft elke poort tenminste een interface?

(e) Zijn de interfaces concrete parameters (attributen) of methodes?

(f) Wordt met dit CD alle software weergegeven?

	Syntaxcriteria componentdiagram
a	geen geïsoleerde componenten?
b	interfaces benodigd en geleverd?
c	begrijpelijke namen?
d	geen nutteloze poorten?
e	zijn de interfaces concreet?
f	alle software meegenomen?

Tabel 7.1: Syntaxcriteria componentdiagram

Inhoudelijke controle CD

We kijken bij elk criterium hoe goed elk CD daarop scoort.
Inhoudelijke critera voor componentdiagrammen:

(a) Zijn er geen cyclische afhankelijkheden?

(b) Gemeenschappelijke sluiting? Maak componenten zo dat veranderingen (zoveel mogelijk) alleen in 1 component doorgevoerd moeten worden

(c) Gemeenschappelijk hergebruik? Als je 1 klasse van een component hergebruikt, hergebruik je elke klasse van de component.

(d) Afhankelijkheidsinversie. Abstractie mag niet van details afhangen. Details moeten uit de abstractie volgen.

(e) Open-dicht. Componenten moeten open zijn voor uitbreiding maar gesloten voor verandering.

(f) Alles of niets. Een component wordt in z'n geheel gereleased of helemaal niet. (en dus niet in gedeelten)

(g) Stabiele abstractie. Een component moet zo stabiel zijn als – ie abstract is. dat wil zeggen: Een uitbreiding moet geen risico voor de stabiliteit zijn.

(h) Stabiel afhankelijk. Als een klasse in een component afhankelijk is van een andere, dan liefst van een stabielere (minder veranderlijke) klasse.

	Inhoudelijke criteria componentdiagram
a	gemeenschappelijke sluiting?
b	gemeenschappelijk hergebruik?
c	volgen details uit de abstractie?
d	open-dicht?
e	alles of niets?
f	stabiele abstractie?
g	stabiel afhankelijk?

Tabel 7.2: Inhoudelijke criteria componentdiagram

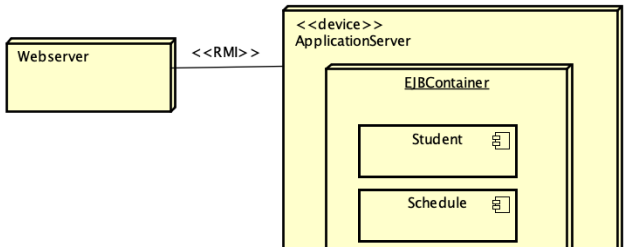
NB: Deze inhoudelijke criteria zijn tamelijk abstract en vereisen inzicht in de architectuur van software. Verwacht dus niet dat je ze 'in een keer' goed krijgt.

Hoofdstuk 8

Deploymentdiagram

Inleiding

Een deploymentdiagram (DD) geeft de inzet van de software op de fysieke onderdelen van een systeem weer. Bijvoorbeeld kan een deploymentdiagram laten zien welke hardware componenten er bestaan (denk aan een webserver of een application server), en welke software op welke node loopt (web applicatie, database etc). De basiselementen van een DD zijn nodes en hun verbindingen (connecties). Nodes kunnen zelf weer nodes bevatten of componenten zoals bedoeld in een componentdiagram (zie hoofdstuk 7).



Figuur 8.1: Deploymentdiagram voorbeeldfragment

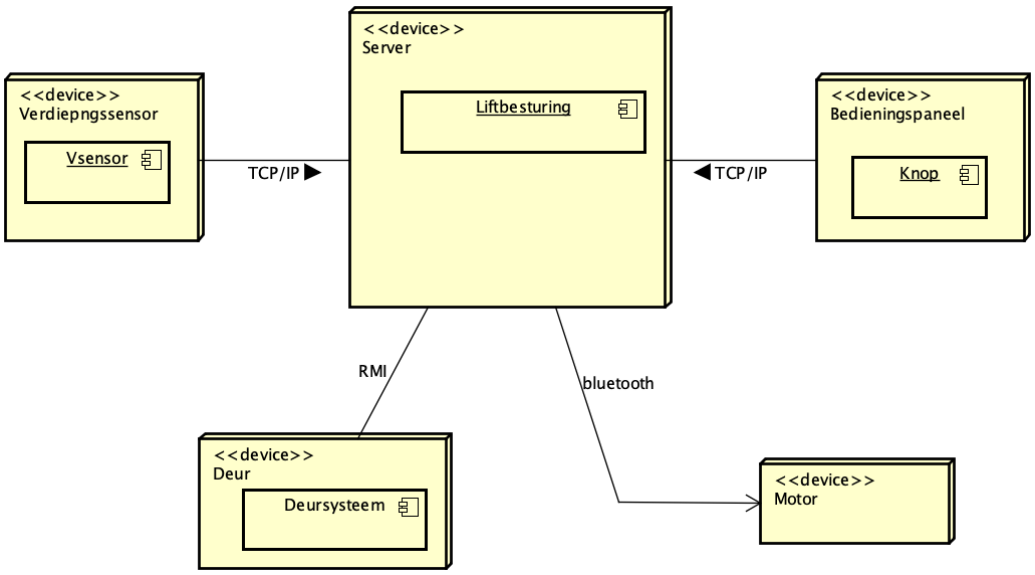
Toelichting bij figuur 8.1: Een node (Webserver) heeft een connectie (≪RMI≫) met een node (ApplicationServer) die een container bevat met twee componenten (Student en Schedule).

In een deploymentdiagram heb je veel vrijheid om elementen te tekenen. Dat maakt het makkelijk omdat je weinig fout kunt doen, maar natuurlijk ook moeilijk omdat je niet weet wanneer je diagram 'goed' is. De criteria

aan het eind van dit hoofdstuk proberen je houvast te geven. Ambler (2018 [1]) geeft een aantal stappen om een DD te maken:

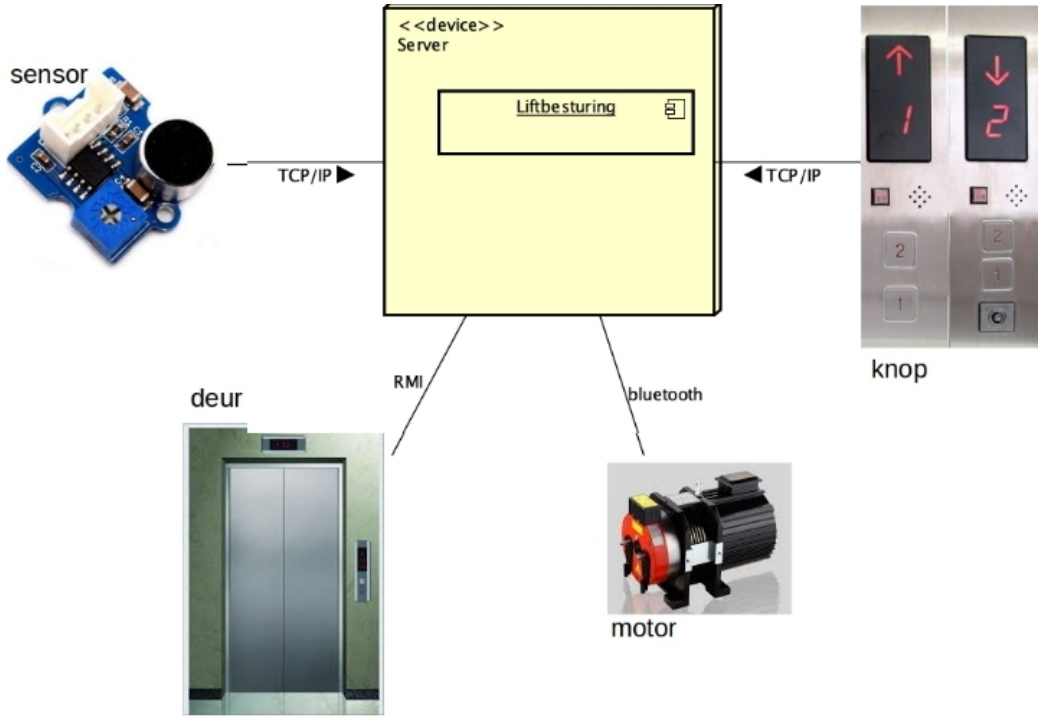
- **Wat is het bereik van je model?** Gaat het om een stand-alone app of wil je alle systemen van een organisatie modelleren?
- **Wat is het technisch fundament van je systeem?** Met welke systemen moet jouw systeem communiceren? Welke security aspecten zijn er (firewall bv)? Hoe robuust moet jouw systeem zijn? Welke gebruikers verwcht je? Op welke hardware draait je systeem? Hoe ga je het systeem beheren?
- **Bepaal de distributie-architectuur** Heb of wil je dikke of dunne clients? In welke technische omgeving gaat je systeem functionenren?
- **Bepaal de nodes en de connecties** Wat zijn de verbindingen en wat zijn de protocollen voor die verbindingen?
- **Verdeel de software over de nodes**

Een deploymentdiagram voor het liftsysteem zou figuur 8.2 kunnen zijn.



Figuur 8.2: Deploymentdiagam lift variant 1

Een betere variant van bovenstaand deployment diagram is figuur 8.3. Want (zie de criteria) het is goed om zoveel mogeljik betekenisvolle afbeeldingen te gebruiken om je deploymentdiagram helder te maken.



Figuur 8.3: Deploymentdiagam lift - variant 2 met afbeeldingen

Inhoudelijke controle DD

Omdat het tekenen van een deploymentdiagram tamelijk veel vrijheid biedt (afgezien van de vanzelfsprekende regels dat nodes verbonden moeten zijn en dat software draait op fysieke nodes) zijn syntaxregels voor een DD niet zo zinvol. Dus volgen hier alleen inhoudelijke criteria:

(a) Zijn er geen cyclische afhankelijkheden?

(b) Is het duidelijk hoe de communicatie tussen de nodes verloopt?

(c) Zijn alle benodigde fysieke elementen in het diagram aanwezig?

(d) Kun je het diagram aan een collega uitleggen?

(e) Komen de (software-)componenten overeen met de componenten in het componentendiagram?

(f) Kan je de hardwareconfiguratie uit het diagram opmaken?

(g) Is de naamgeving goed?

(h) Zijn de stereotypen consistent?

(i) Heb je zo veel mogelijk verhelderende afbeeldingen gebruikt?

	Inhoudelijke criteria deploymentdiagram
a	geen cycles ?
b	communicatie duidelijk?
c	alle fyieke elementen aanwezig?
d	uitleggen?
e	reeele componenten?
f	hardwareconfiguratie duidelijk?
g	goede naamgeving?
h	stereotypen consistent?
i	illustratief?

Tabel 8.1: Inhoudelijke criteria deploymentdiagram

Hoofdstuk 9

UML en onderzoek

Inleiding

Een veel gehanteerde manier om het ontwerp van een systeem te beschrijven is het '4 plus 1' model (Kruchten 1995 [2]). Waarbij er 4 verschillende views plus een verzameling scenario's ingezet worden om (het ontwerp van) een systeem te beschrijven. Dat kan zowel een systeem zijn dat nog gemaakt moet worden, als een systeem dat al bestaat. Dat laatste bijvoorbeeld om de huidige situatie te beschrijven. De vier views met scenario's zijn (zie ook figuur 9.1):

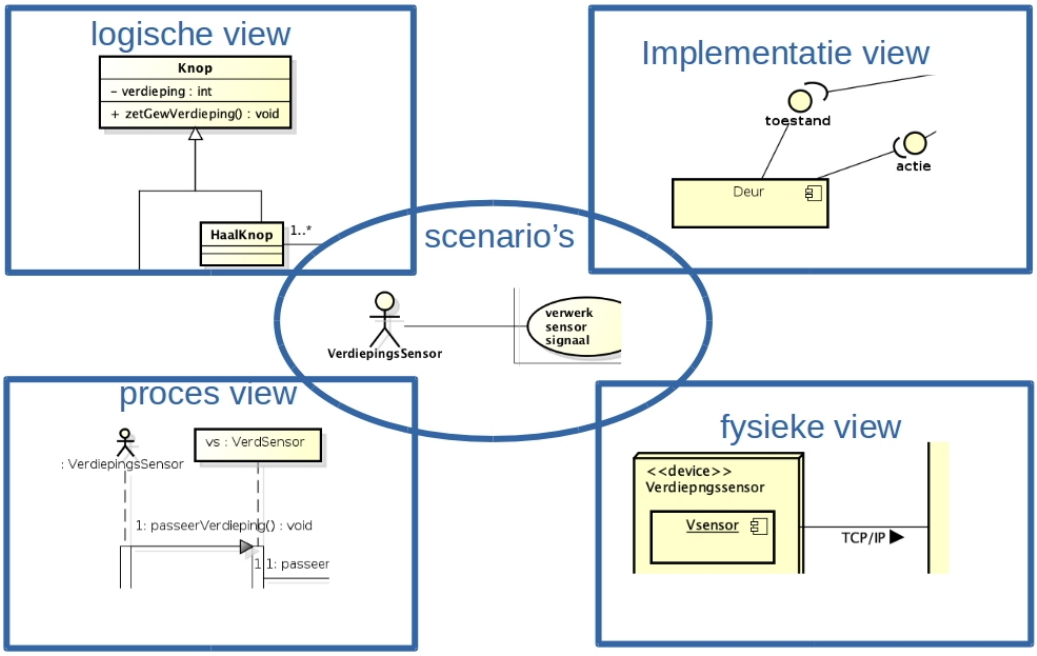
- **Logische of structurele view** In de logische view beschrijf je de functionaliteit van het systeem voor de eindgebruikers. Diagrammen die je daar voor gebruikt zijn bijvoorbeeld klasse- en objectdiagrammen.

- **Proces- of gedragsview** Deze view laat de dynamische aspecten van het systeem zien, dus hoe het systeem zich gedraagt. Hierbij kun je gebruik maken van sequence-, activiteit-, toestands- of communicatiediagrammen.

- **Implementatie view** Deze view gaat over het beheer van software. Diagrammen die je hier kunt gebruiken zijn component- en packagediagrammen eventueel aagevuld met klassediagrammen.

- **Fysieke view** Hierbij gaat het om de uitrol van de software over fysieke systemen. Het deployment-diagram is daarvoor geschikt.

- **Scenario's** De beschrijving van de functionaliteit van een systeem, ook wel requirements-analyse genoemd, met behulp van user-stories, use-case-diagrammen en teksten (scenario's) die de acties binnen een use-case beschrijven.



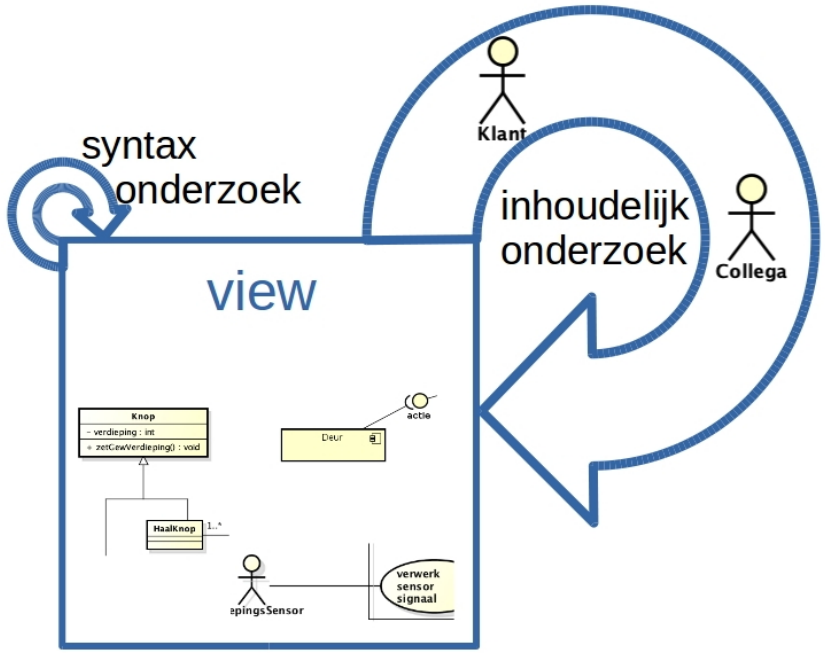
Figuur 9.1: Het 4 plus 1 model

Gebruik '4+1' model voor onderzoek

Je kunt het '4+1' model gebruiken om te laten zien dat je niet 'zomaar' aan het programmeren bent maar dat je *onderzoekt* wat er nodig is voor het systeem. Dat kan op twee verschillende manieren. Met kleine of grote 'lusjes'.

- **controleer elk diagram op de syntaxregels**. Hier mee onderzoek je of je diagrammen voldoen aan de regels die UML stelt. Dit voer je zelf uit op je eigen diagrammen of die van een collega. Een klein lusje.

- **controleer elk diagram op de inhoudelijke regels**. Hiermee maak je een grotere 'lus'. Je moet dan namelijk met je opdrachtgever, klant of collega's onderzoeken of de diagrammen voldoen aan de eisen die aan het systeem gesteld worden. Deze controle kan plaats vinden tijdens het ontwikkelproces en als (stukken van) de software opgeverd worden



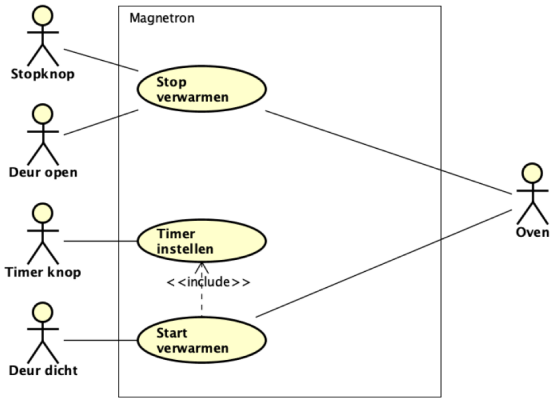
Figuur 9.2: Het onderzoeken van een view, kleine en grote lus

Door deze twee vormen van onderzoek te documenteren laat je zien dat je over een onderzoekende houding beschikt.

de kleine lus

Hoe werkt dat in de praktijk? Stel je hebt een use-case-diagram met mogelijke fouten (figuur 9.3)

Daar zet je de syntaxregels voor een UCD naast met een check kolom. Met bijvoorbeeld 'ν' = goed, '?' = onduidelijk, '$\times$' = niet goed, '-' = nvt.



Figuur 9.3: Diagram voor onderzoek

Syntaxcriteria	
alle input aanwezig	ν
actieve actor bij elke UC	ν
elke user-story gedekt	?
systeemperspectief	$\times$
actoren buiten systeem	ν
output terug te vinden	?
systeemgrens	ν
goede namen actoren	$\times$
extension point	-
include als $\geq$ 2 UC	$\times$

Tabel 9.1: Syntaxcriteria

De volgende stap is het verbeteren van het diagram. Hier is een use-case-diagram gebruikt als voorbeeld van een kleine lus, maar dit kan natuurlijk bij elk diagram.

de grote lus

Het onderzoek in de grote lus is meer werk. Het gaat erom dat je een gegeven diagram voorlegt aan een klant, een opdrachtgever, een collega, een projectleider, een teamgenoot. Aan de hand van de inhoudelijke criteria onderzoek je of het diagram voldoet, dat wil zeggen je verzamelt commentaar van de belanghebbenden en verbetert daarmee het ontwerp. Dat kan op verschillende momenten in het ontwikkelproces. Bij een agile methode kan dat bij elke sprint die van toepassing is gebeuren. Het kan ook bij de oplevering van een product, Het nut van het onderzoek beperkt zich dan tot het achteraf verantwoorden of advies voor een volgende versie van he product. Doorgaans zal de *methode* van onderzoek bij de grote lus de *interviewmethode* zijn. Het is daarbij belangrijk om om te schrijven *welke veranderingen je waarom* doorgevoerd hebt. Want daarmee onderbouw je de door jou gemaakte keuzes
In de praktijk is het niet werkbaar om van elke view en een kleine en een grote lus te documenteren. Dat zou neerkomen op tenminste 4 diagrammen die je met behulp van 2 soorten criteria wellicht herhaald onderzoekt. Het is dus aan te raden om er een of twee views uit te kiezen waarbij je je onderzoek expliciet documenteert. Bij de andere views zul je evengoed de controle uitvoeren, maar dan impliciet (ongedocumenteerd).

Hoofdstuk 10

Activiteitendiagrammen – The making of

In dit hoofdstuk staan schema's over hoe hoe je de verschillende diagrammen moet maken: UCD, KD, SD, STD en CD. De schema's zijn activiteitendiagrammen: AD, ook een soort UML-diagram.

AD basis UML-analyse

Als eerste een schema over hoe je met user stories en 5 UML-diagrammen een basis UML-analyse maakt. Zie figuur 10.1

AD use-case-diagram

Het activiteitendiagram om een use-case-diagram te maken bestaat uit 4 zogenoemde lanen of stappen: **1)** informatie verzamelen: Het kan zijn dat je zelf informatie (user stories) moet verzamelen bij de klant, het kan ook zijn dat dat al voor je gedaan is **2)** maak use-case: per user-story (of combinaties daarvan) moet je een use-case maken met actoren. Net zo lang tot alle user stories behandeld zijn **3)** bepaal systeemgrenzen: in deze stap ga je na wat je wel en wat je niet gaat programmeren. Alles wat buiten de systeemgrenzen valt ga je niet bouwen. **4)** terugkoppeling: om ervoor te zorgen dat je systeem aansluit bij wat de klant wil, ga je met het gemaakte diagram naar de klant toe en legt hem of haar aan de hand van het use-case-diagram uit wat je gaat bouwen. Als het niet goed is begin je opnieuw. Zie figuur 10.2

AD Klassediagram

Het activiteitendiagram voor het maken van een klassediagram bestaat (ook) uit 4 zogenoemde lanen of stappen. **1)** voorbereiden klassediagram: Je moet het use-case-diagram op orde hebben om een klassediagram te maken **2)** klassen tekenen: Per actor en gegevens-eenheid moet je een klasse tekenen **3)** associaties leggen: Alle klassen kunnen met elkaar worden verbonden. Ga na welke verbanden nodig zijn. **4)** bepaal methoden: Bijna elke klasse bevat methoden waarmee "het werk gedaan wordt". Zie figuur **??**

AD sequence-diagram

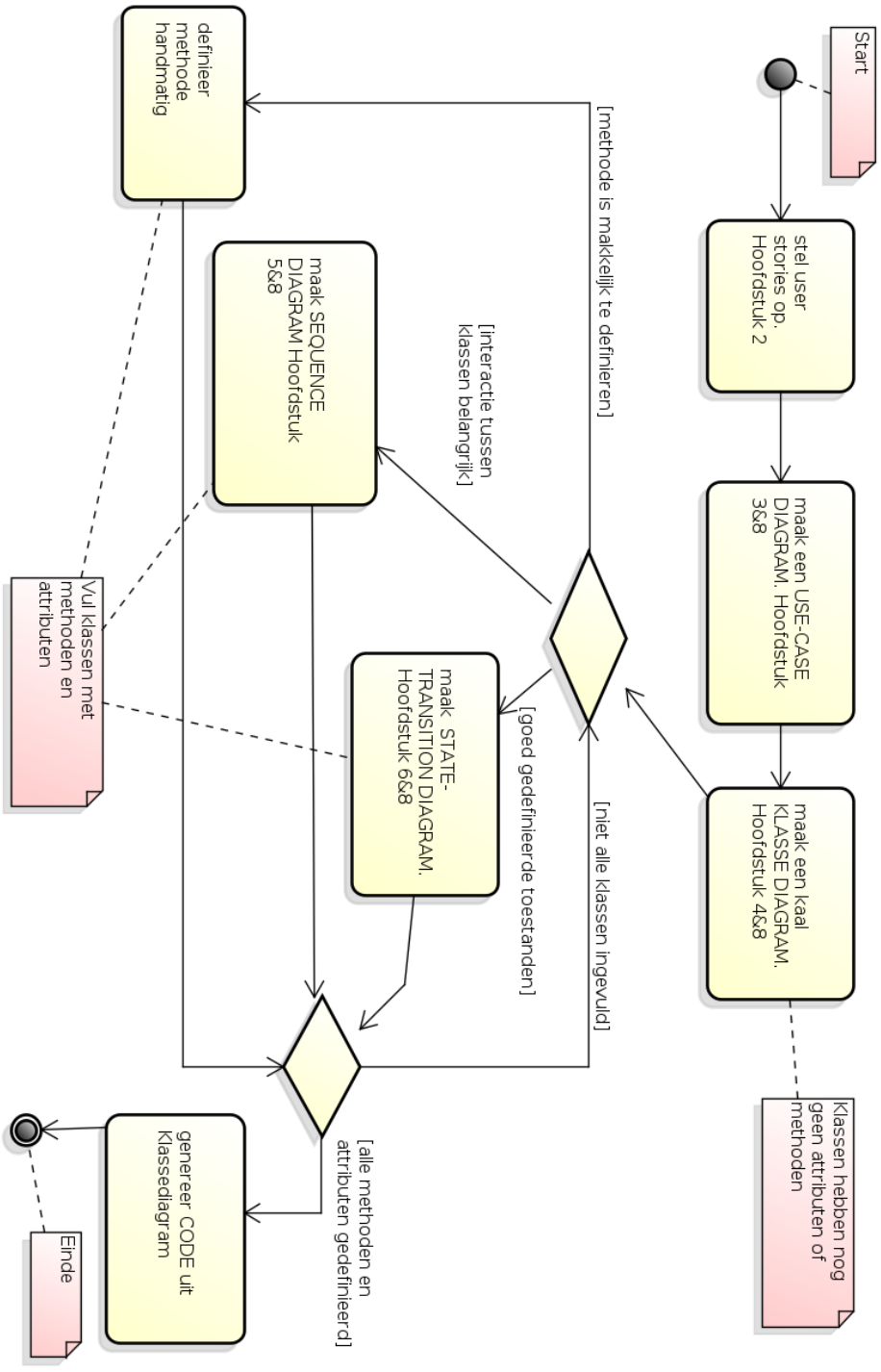
Het activiteitendiagram voor het maken van een sequence-diagram bestaat uit 5 lanen. **1)** voorbereiden sequence-diagram: Je moet een use-case-diagram en een klassediagram hebben **2)** actieve actoren: Per actieve actor een start van het sequence-diagram **3)** controller en passieve actoren: Het toevoegen daarvan aan het sequence-diagram **4)** dataobjecten toevoegen: Als er speciale data-klassen zijn moet die aangesproken worden. **5)** Methoden toevoegen. Zie figuur 10.4

AD State transition diagram

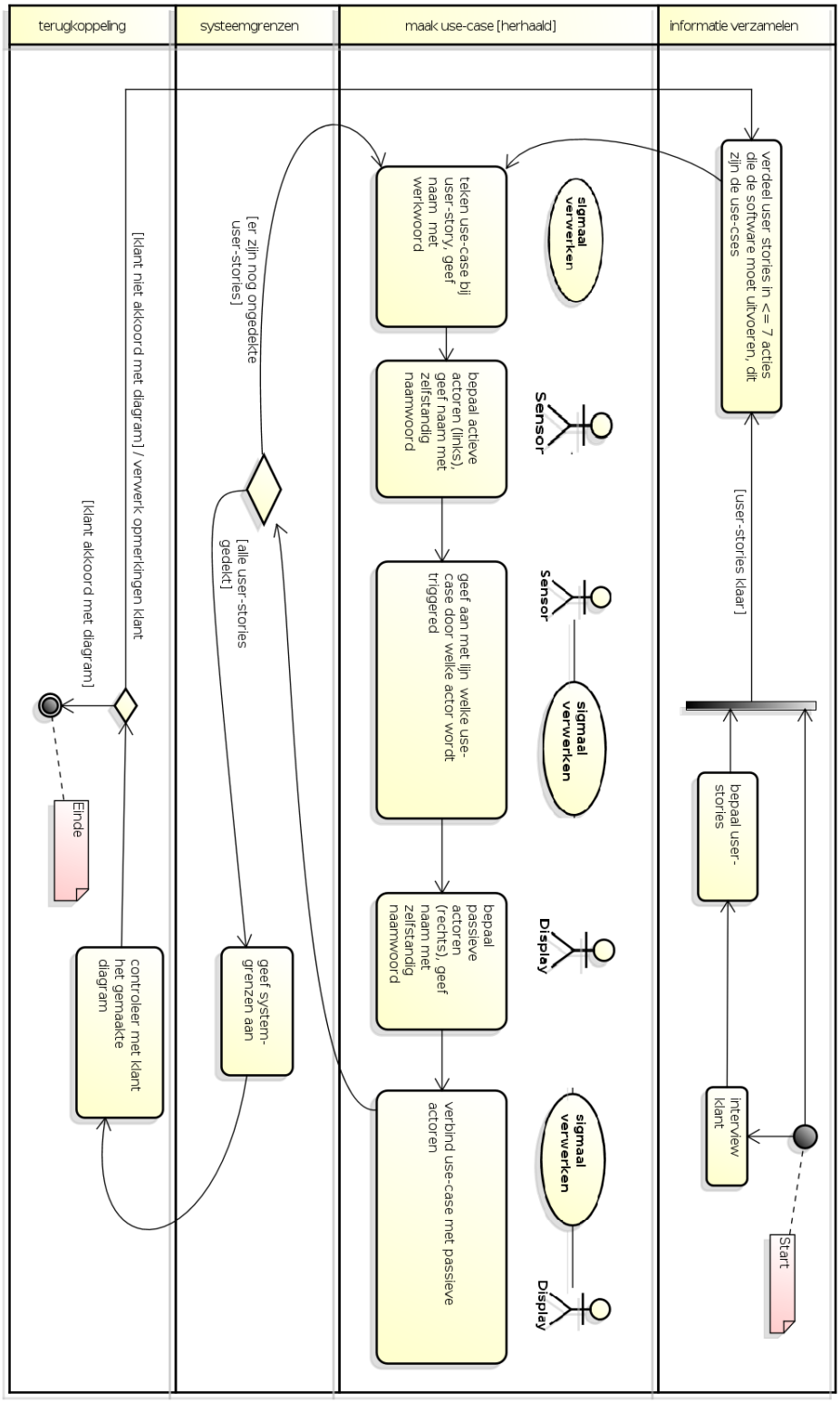
Het activiteitendiagram voor een STD bestaat uit 3 lanen. **1)** voorbereiding: Om te beginnen moeten de toestanden goed gedefinieerd worden, zo dat ze zo weinig mogelijk overlappen. **2)** toestanden en transities: Daarna moeten ze getekend worden en de overgangen (transities) bepaald en ingevuld worden. **3)** controle: Het wordt afgesloten met een controle op zwarte gaten en supernova's. Zie figuur 10.5

AD Componentdiagram

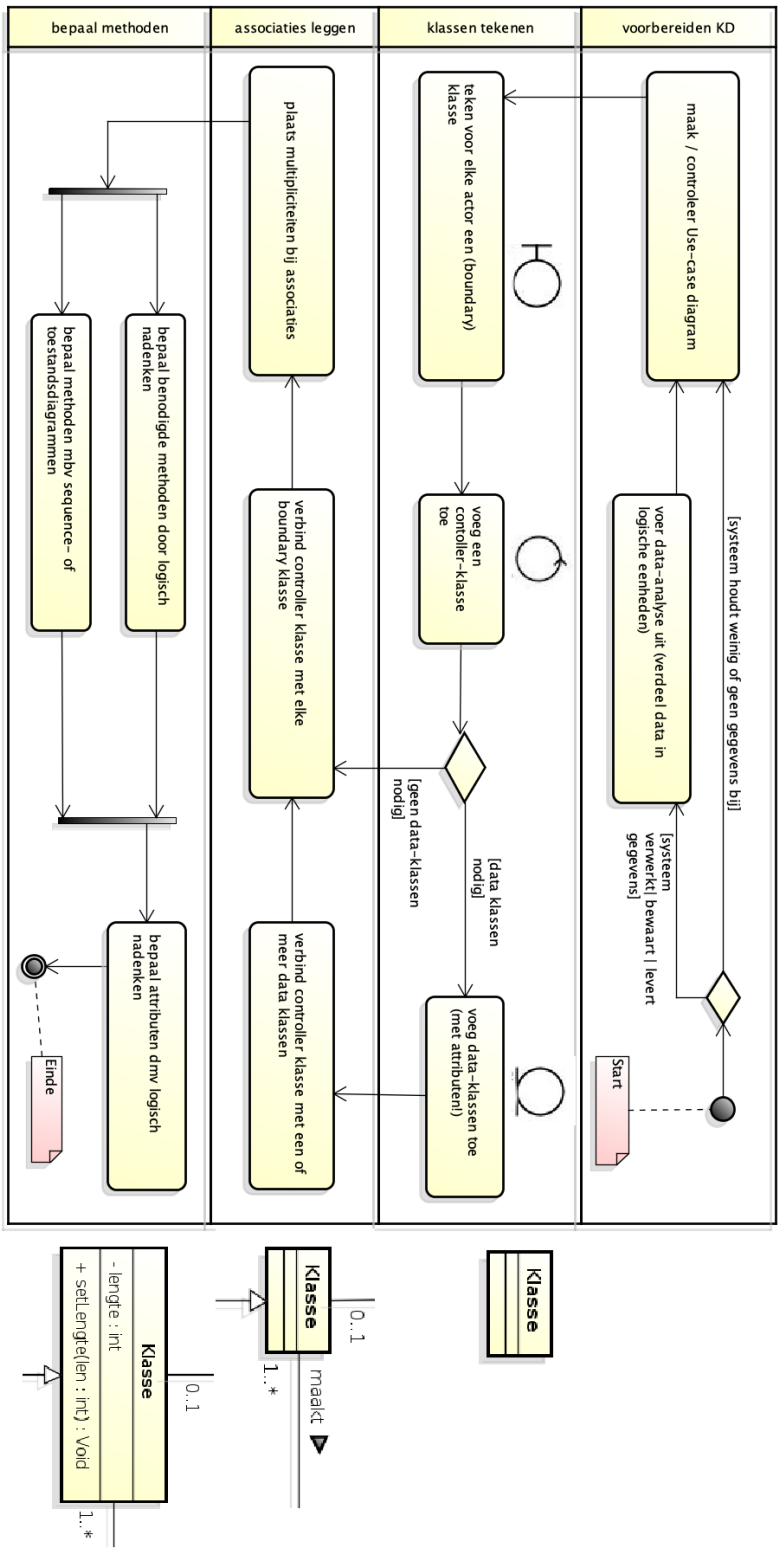
Het componentdiagram is het meest abstracte van de diagrammen die hier behandeld worden. Het tekenen van het diagram is niet zo lastig, wèl het op de juiste manier verdelen van software in componenten. Een goede taktiek hierbij is het minimaliseren van de interface (attributen en methoden die uitgewisseld worden tussen componenten). Zie figuur 10.6



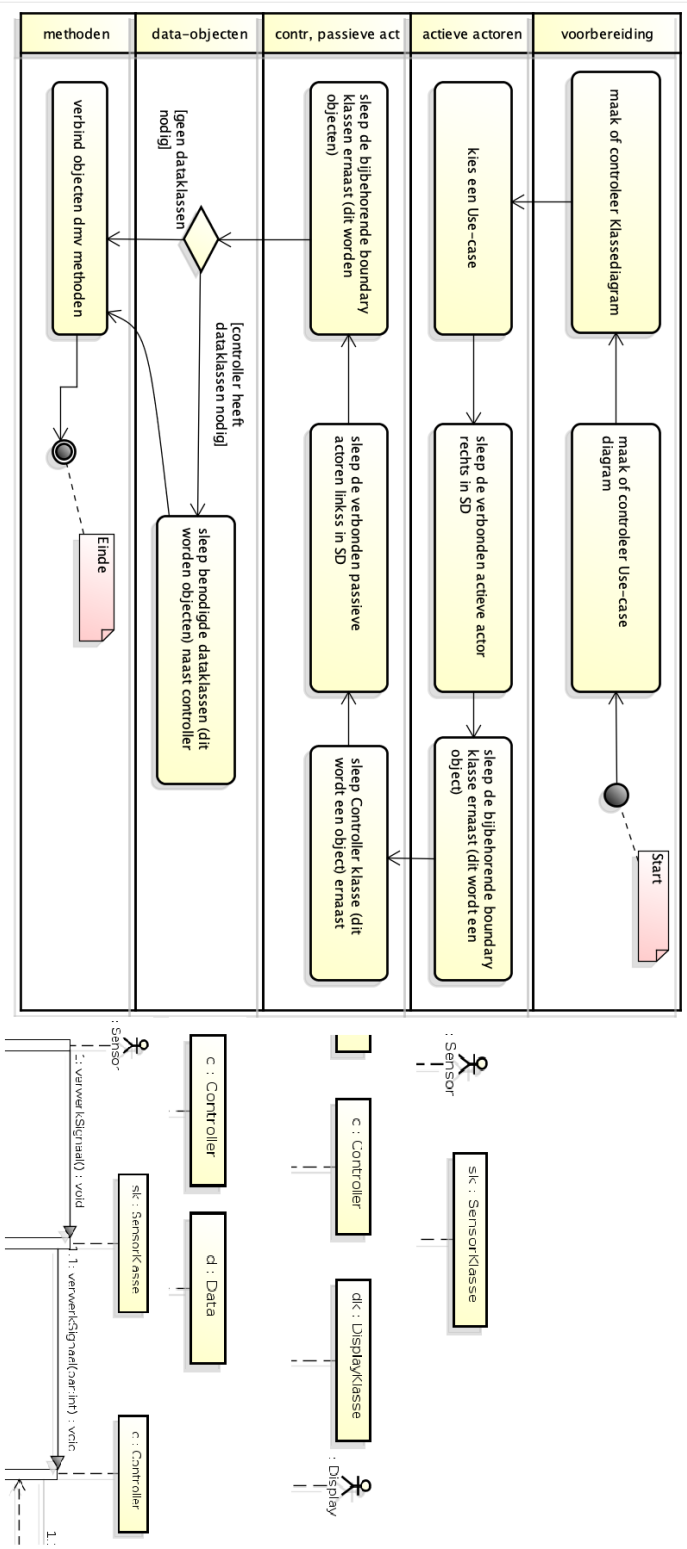
Figuur 10.1: AD basis UML-analyse



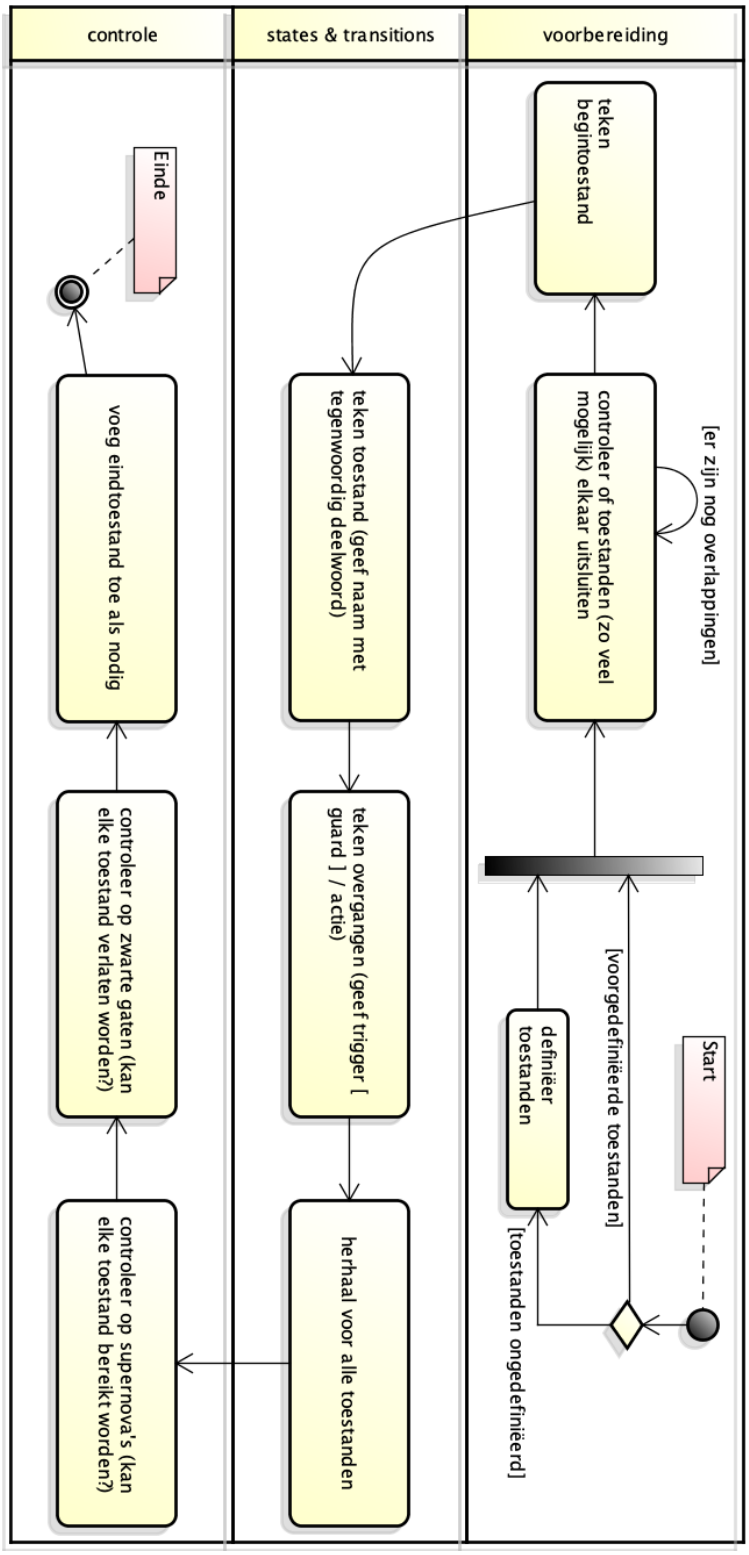
Figuur 10.2: AD voor het maken van een simpel use-case-diagram



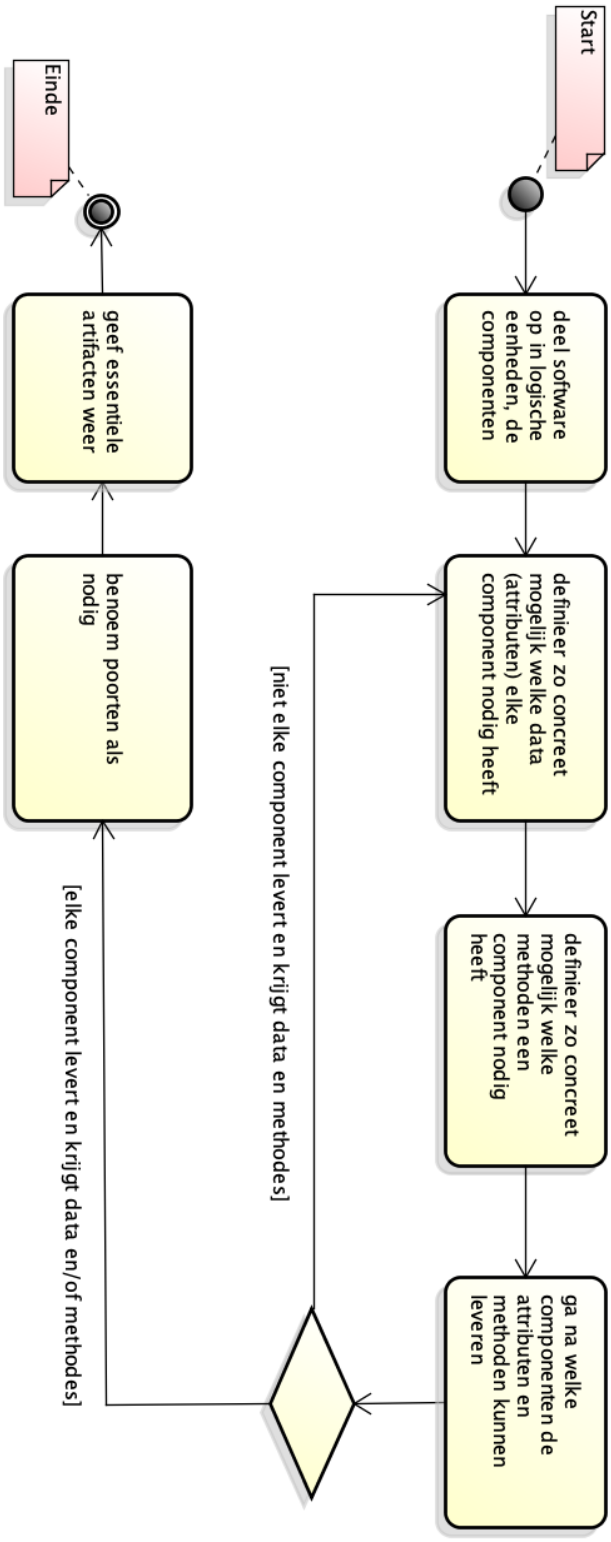
Figuur 10.3: AD voor het maken van een klassediagram



Figuur 10.4: AD voor het maken van een sequence-diagram



Figuur 10.5: AD voor het maken van een toestandsdiagram



Figuur 10.6: AD voor het maken van een componentdiagram

Hoofdstuk 11

Analyse administratief systeem

Voorbeeldbeschrijving congres ontvangst

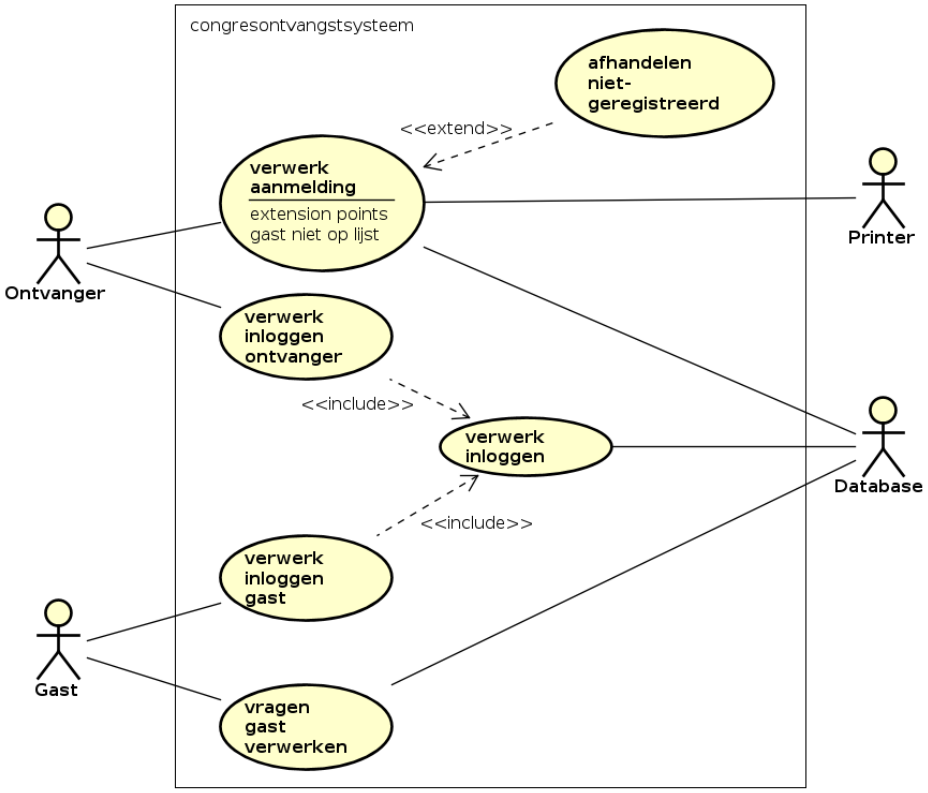
Registratie van deelnemers bij het begin van een congres.

Contextbeschrijving: De deelnemers hebben zich vooraf geregistreerd. Wat nu moet gebeuren is dat als een deelnemer aankomt bij het congres er gecontroleerd wordt aan de hand van een gastenlijst of hij of zij zich heeft ingeschreven. Vervolgens wordt aangevinkt dat de gast aanwezig is, een badge uitgedraaid, en de gast krijgt toegang tot het congressysteem middels een code. Als de gast die code activeert kan hij of zij met z'n mobiele apparaat de gastenlijst raadplegen, het programma lezen, de presentaties van de sprekers bekijken etc. Bovendien wordt er een tasje met de nodige prullaria overhandigd. De mensen die de deelnemers ontvangen, de zogenaamde ontvangers moet in het systeem inloggen om bij de gastenlijst te kunnen. Er is een aparte database met een gastenlijst, welke van die gasten sprekers zijn en de presentaties die die sprekers gaan houden.

User Stories

- "Als ontvanger wil ik kunnen inloggen"
- "Als ontvanger wil gasten op de lijst kunnen afvinken"
- "Als ontvanger wil een badge kunnen uitdraaien zodat ik die aan de gast kan geven"

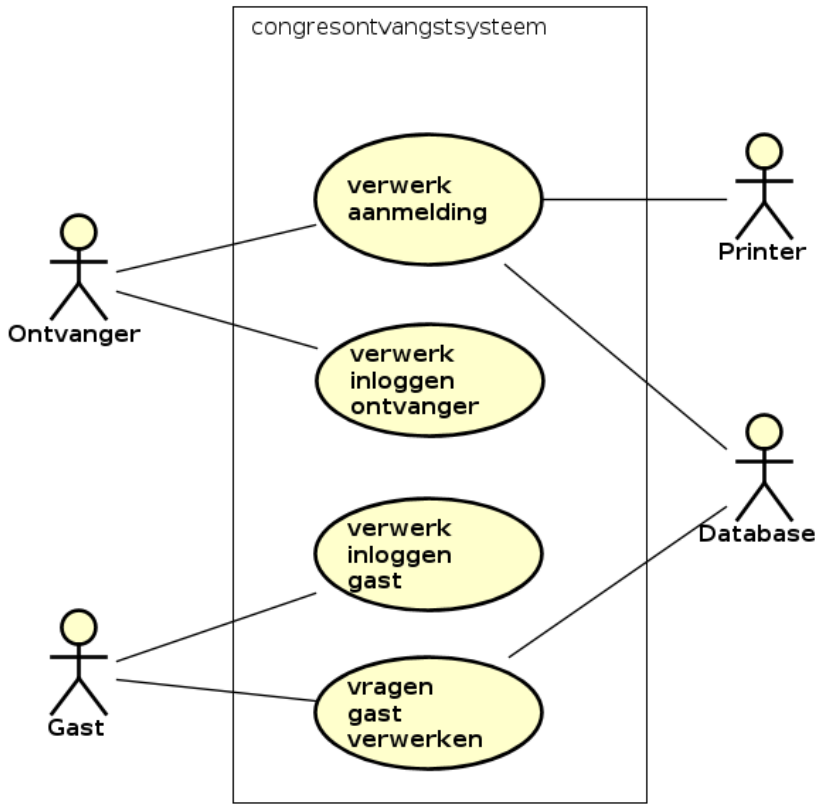
- "Als gast wil ik kunnen inloggen"
- "Als gast wil de gastenlijst raadplegen"
- "Als gast wil ik het programma van het congres zien"



Figuur 11.1: UCD van ontvangst van een congres - vaiant 1

NB: Er zijn flink wat user stories samengevoegd om het UCD overzichtelijk te houden! Waarom zit het overhandigen van het tasje niet in het UCD?

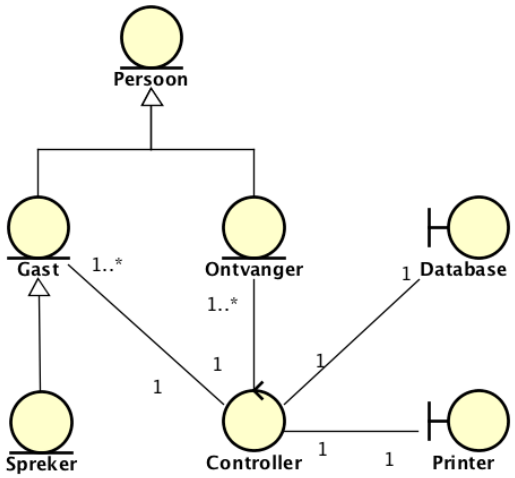
Deze variant (figuur 11.2) van het UCD is veel overzichtelijker, maar je kunt de uitzonderingen (gast is niet geregistreerd) natuurlijk niet goed zien, evenmin als de include. M.a.w. dit UCD is wat abstracter. Soms, zeker bij grotere systemen kan dat erg handig zijn, juist om overzicht te houden.



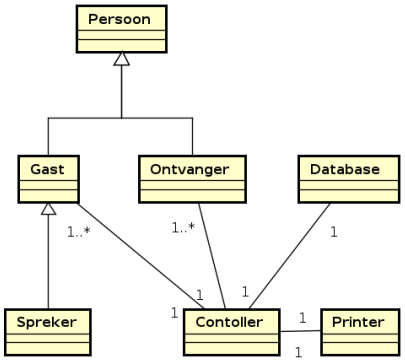
Figuur 11.2: UCD van ontvangst van een congres - vaiant 2

Klassediagrammen (kaal)

Merk op dat er 'ineens' 2 klassen bij zijn gekomen die niet in de beschrijving stonden. Dit gebeurt vaak bij niet-real-time-systemen omdat je dataklassen kunt hebben die niet expliciet in de probleemomschrijving (specs / user stories) staan. Hier is bedacht dat de gast en de ontvanger allebei personen zijn waarvan je gegevens wil opslaan. Het is handig om een generalisatie persoon te maken omdat je dan maar bij één klasse een inlogmethode hoeft te schrijven. De gast en de ontvanger erven dan 'automatisch' deze methode. Daarnaast is een spreker natuurlijk een gast, maar wel een speciale gast. Immers van haar of hem staan de presentaties in de database. Verder is het een kwestie van smaak welke versie (met of zonder symbolen) je handiger vindt.

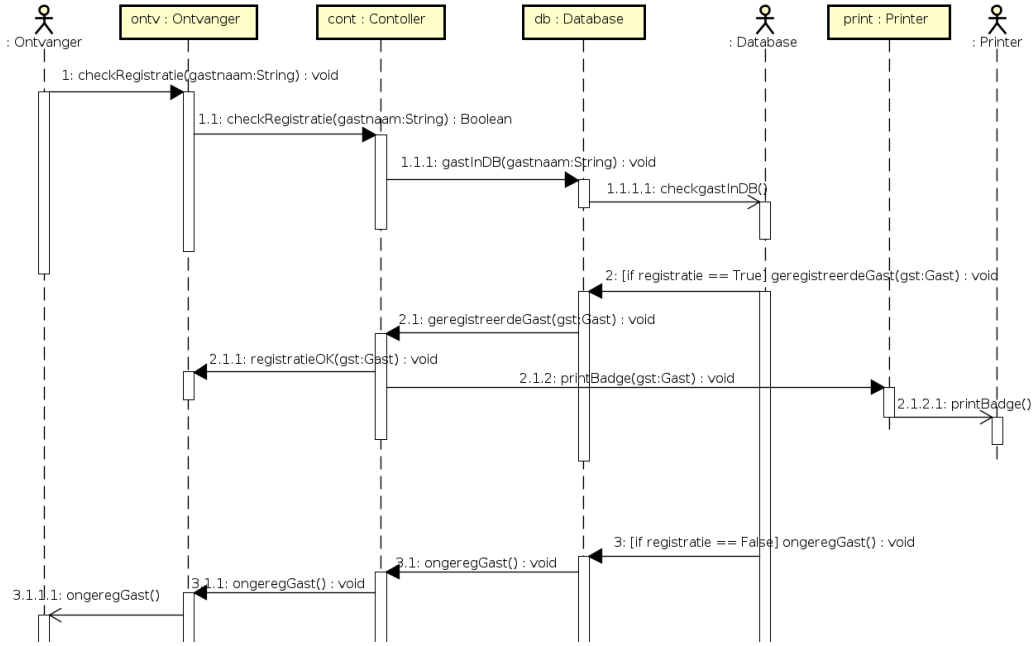


Figuur 11.3: KD van ontvangst van een congres - met symbolen

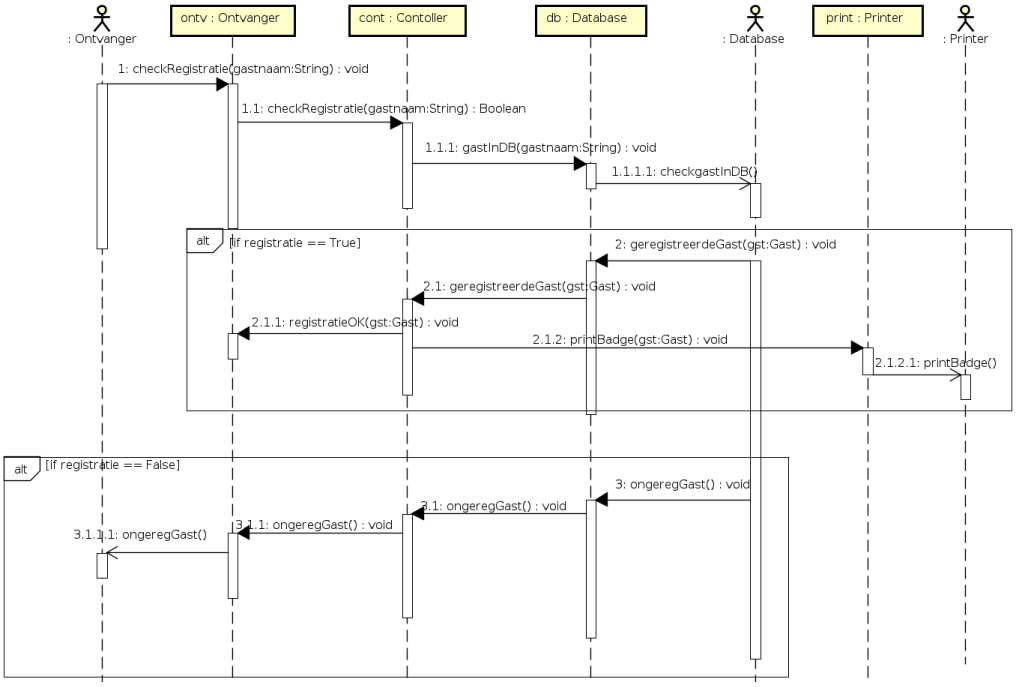


Figuur 11.4: KD van ontvangst van een congres - 'normaal'

Sequence-diagrammen



Figuur 11.5: SD congresontvangst verwerk aanmelding variant 1



Figuur 11.6: SD congresontvangst verwerk aanmelding variant 2

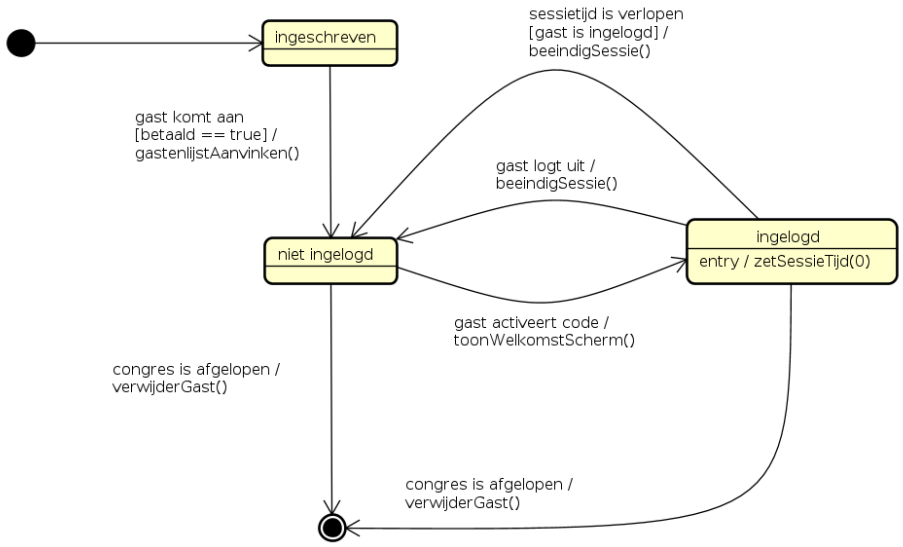
Bij deze variant (figuur 11.6) zijn [alt-] rechthoekjes om de condities gezet zodat je in een oogopslag kunt zien welke methode wanneer worden uitgevoerd, èn welke objecten daarbij een rol spelen

Toestandsdiagrammen

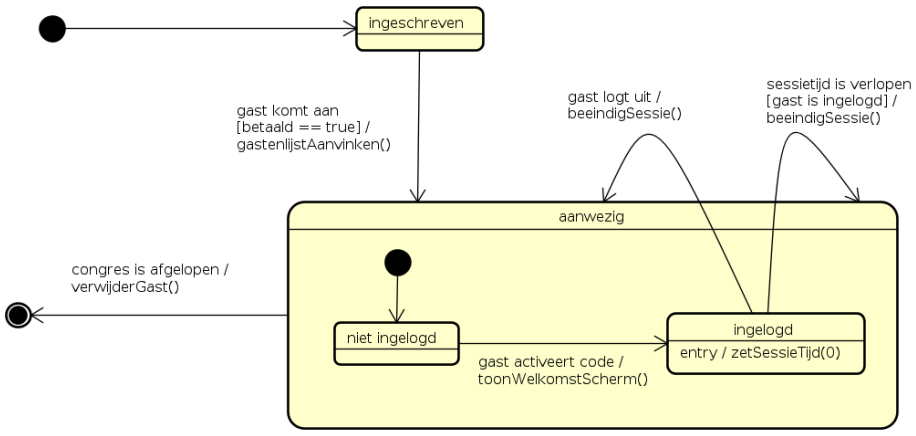
Bij het maken van een toestandsdiagram van een administratief systeem is het logisch om een toestandsdiagram van één object te maken, immers de toestanden van het hele systeem zijn niet zo interessant (aan/uit?, wachtend/actief?).

Merk op dat we hier (figuur 11.7) een aanvulling op de methoden voor de controller gevonden hebben namelijk `gastenLijstAanvinken()` die we in het SD vergeten waren.

Bij de volgende variant van het STD (figuur 11.8) is een nieuwe toestand toegevoegd, nl dat een gast simpelweg aanwezig kan zijn. De toestanden ingelogd of niet ingelogd zijn dan subtoestanden van aanwezig zijn. Merk op dat op elk niveau van een STD opnieuw een start toestand verplicht is. Hier in de toestand aanwezig, omdat je anders niet weet in welke subtoestand van aanwezig de gast 'begint'.



Figuur 11.7: STD congresontvangst toestanden van het object gast variant 1

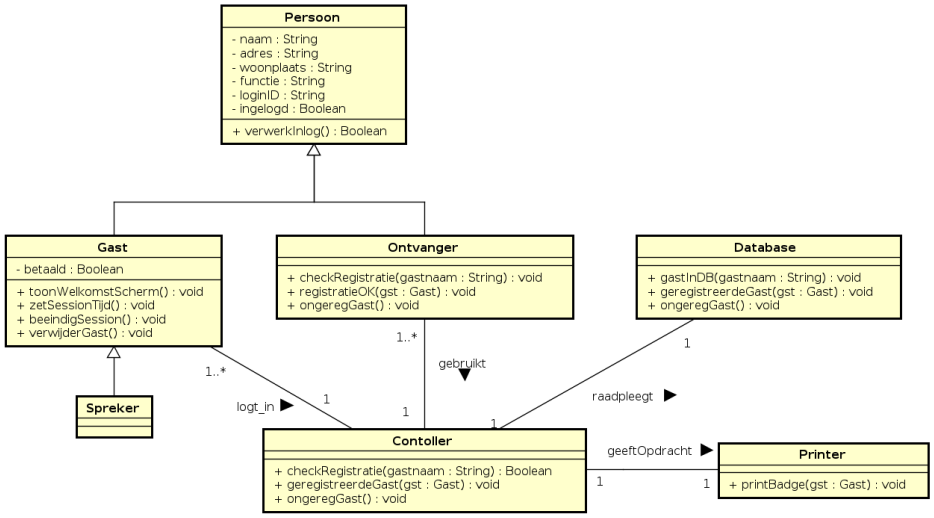


Figuur 11.8: STD van het object gast - met subtoestanden

Klassediagram

Op grond van het SD en het STD zijn methoden toegevoegd aan het KD (figuur 11.9). Merk op dat door logisch nadenken ook attributen zijn gevonden voor Persoon en Gast. *Gegenereerde code Controller en Ontvanger:*

```
import java.util.Collection;
public class Contoller{
        private Gast[] gast;
        private Ontvanger[] ontvanger;
        private Database database;
        private Printer printer;
        public Boolean checkRegistratie(String gastnaam){
                return null;}
        public void geregistreerdeGast(Gast gst) {}
        public void ongeregGast() {}
}
public class Ontvanger extends Persoon{
        private Contoller contoller;
        public void checkRegistratie(String gastnaam) {}
        public void registratieOK(Gast gst) {}
        public void ongeregGast() {}
}
```

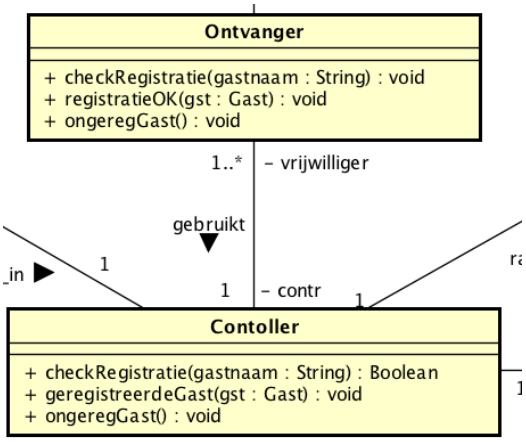


Figuur 11.9: KD congresontvangst

associatie-einden

Merk op dat als je in je klassediagram de *associatie-einden* namen geeft (`vrijwilliger` en `contr` in figuur 11.10) je automatisch in de code *objecten* namen geeft:

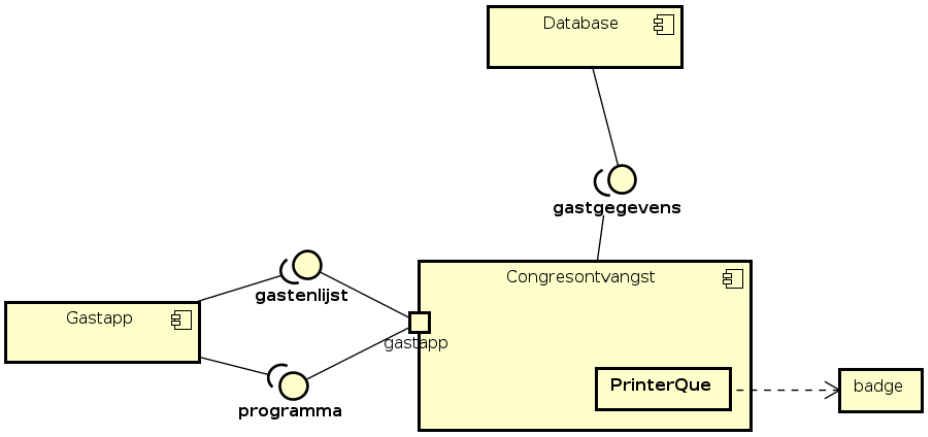
```
//  fragment
public class Contoller {
  private Gast[] gast;
  private Ontvanger[] vrijwilliger;
  private Database database;
public class Ontvanger extends Persoon {
  private Contoller contr;
```



Figuur 11.10: Associatie-einden expliciet benoemd

Componentdiagram

Omdat het registreren van de aanwezigheid van gasten geen groot systeem is, zou het in de praktijk wat minder zinvol zijn om hier een componentdiagram van te maken. Maar als oefening doen we het toch. Zinvol is b.v. om van de database een aparte component te maken omdat die extern is. Veronderstel daarbij een aparte app voor de gasten. NB: Let op de poort en het *artifact* 'badge'.



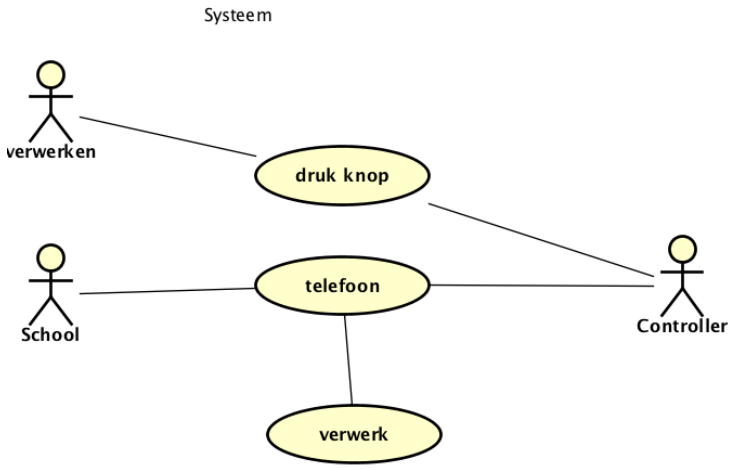
Figuur 11.11: CD congresontvangst

Hoofdstuk 12

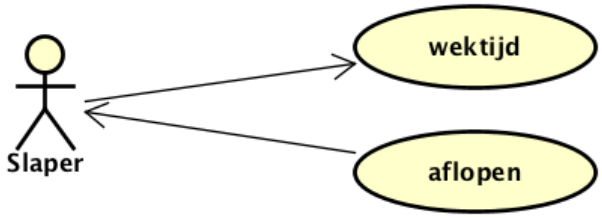
Tegenvoorbeelden

In dit Hoofdstuk laten we voorbeelden zien die ***niet goed*** zijn. Soms kun je namelijk veel leren van van een voorbeeld dat niet goed is ("zo moet het niet"). Ga bij elk van de volgende diagrammen met de bijbehorende criteria na welke fouten er gemaakt zijn. Voor als je jezelf wil controleren staan aan het eind van dit hoofdstuk de fouten opgesomd.

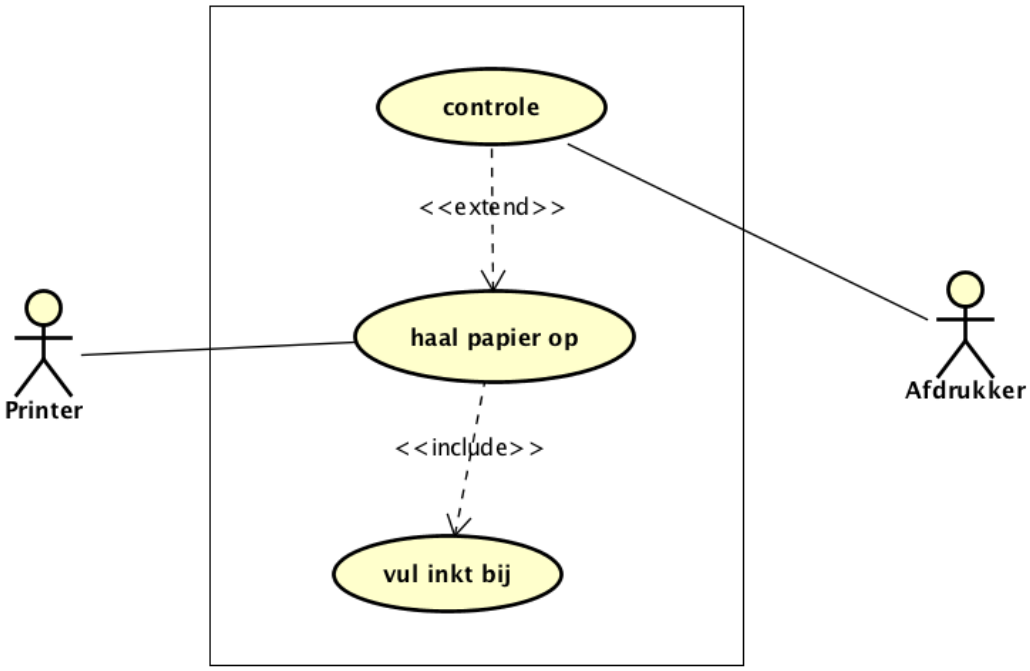
UCD tegenvoorbeelden



Figuur 12.1: UCD tegenvoorbeeld 1. Een use-case-diagram voor een schoolsysteem.

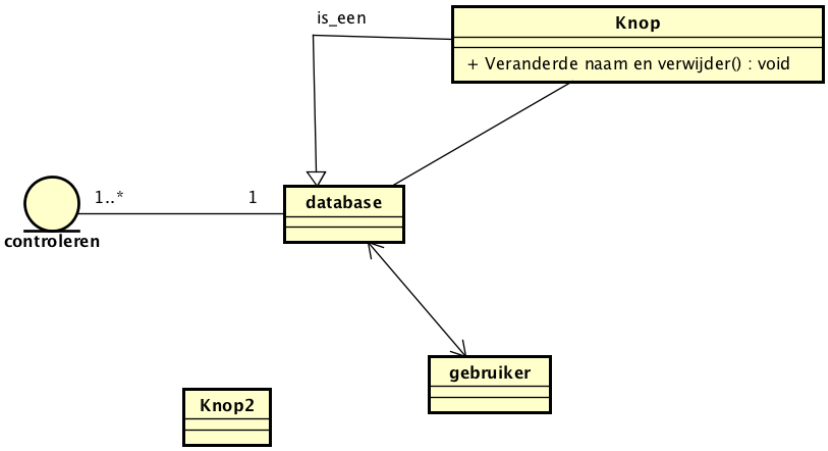


Figuur 12.2: UCD tegenvoorbeeld 2. Een use-case-diagram voor een wekker(-app).



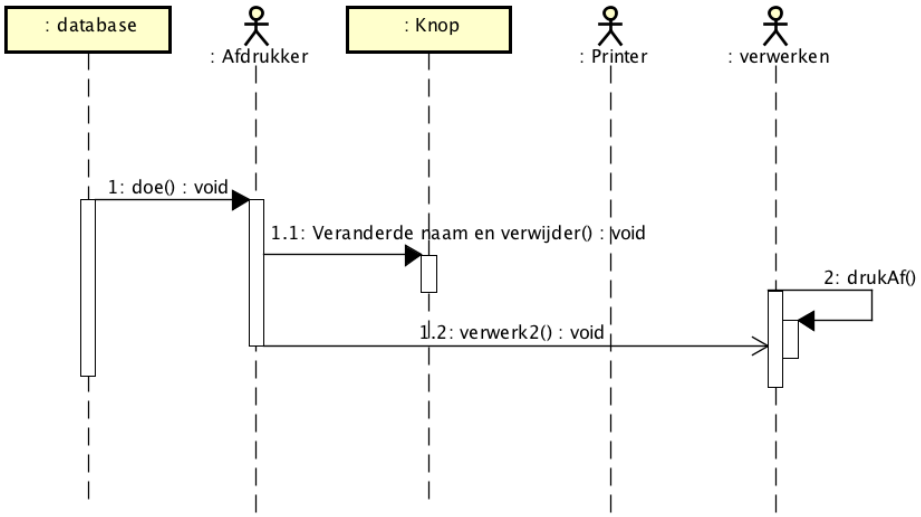
Figuur 12.3: UCD tegenvoorbeeld 3. Een use-case-diagram voor een printer.

KD tegenvoorbeeld

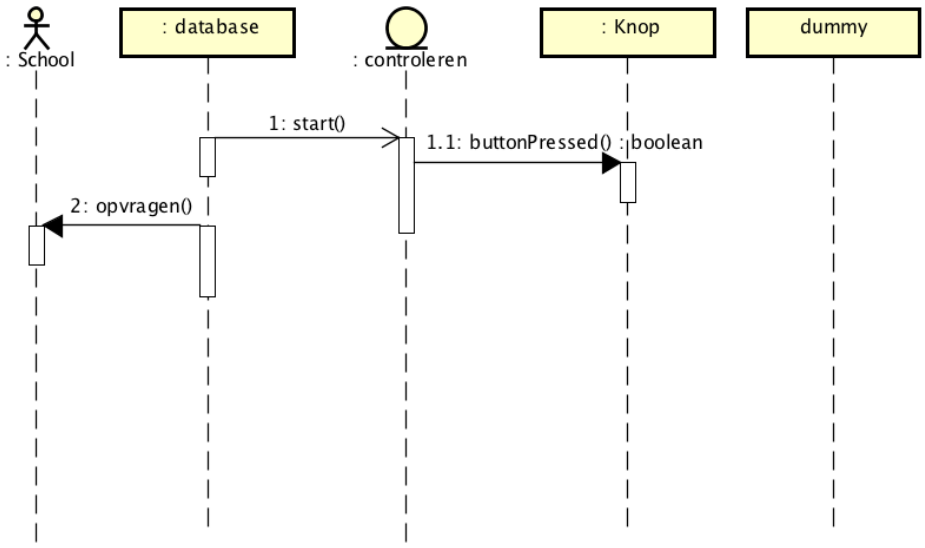


Figuur 12.4: KD tegenvoorbeeld 1.

SD tegenvoorbeelden

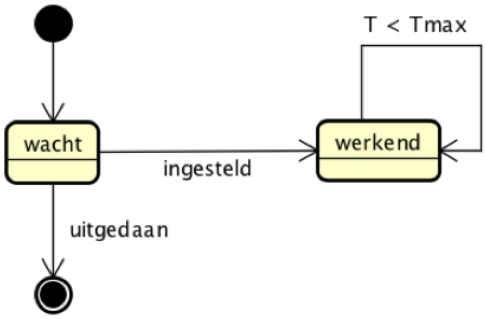


Figuur 12.5: SD tegenvoorbeeld 1.

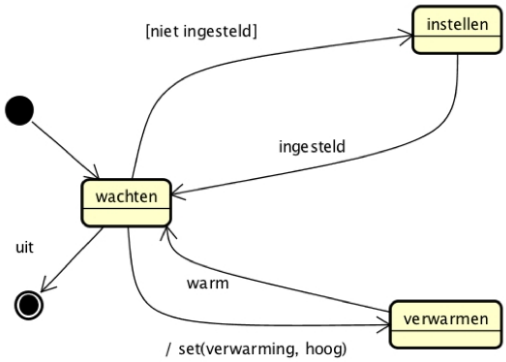


Figuur 12.6: SD tegenvoorbeeld 2.

STD tegenvoorbeelden

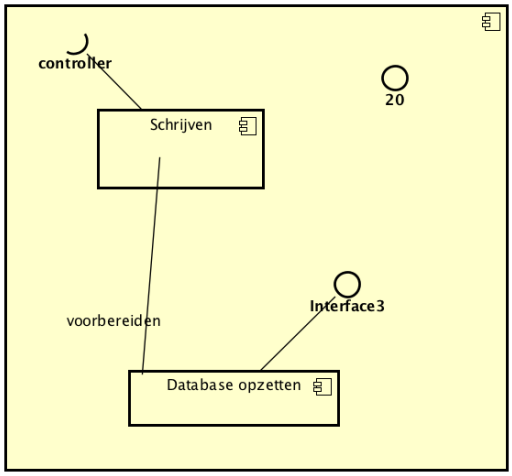


Figuur 12.7: STD tegenvoorbeeld 1.



Figuur 12.8: STD tegenvoorbeeld 2.

CD tegenvoorbeeld



Figuur 12.9: CD tegenvoorbeeld 1.

Fouten van de tegenvoorbeelden:

UCD tegenvoorbeeld 1:
(g) geen systeemgrens,
(h) 'verwerken' is een actie – geen naam van een actor,
(d) telefoon is een ding – geen naam voor een actie,
(i/j) verbinding tussen 2 use cases verboden behalve bij extend of include,
(h) 'Controller' lijkt op een naam van software – verwarrend,
alleen als deze software inderdaad iets naar een Controler stuurt is is dit een goede actor
(d) de naam 'druk knop' is niet goed omdat de software zw geen knopje indrukt,
(f) 'verwerk' heeft noch input noch output,
en wat wordt verwerkt?
(g) de naam van het systeem "Systeem" is te algemeen,
(d) niet elke use-case is geformuleerd als actie van de software,

UCD tegenvoorbeeld 2:
pijltjes zijn niet nodig
(g) geen systeem grens
(g) geen systeemnaam
(b) de use-case wordt niet getriggerd door een actor, maar komt spontaan (mag niet) tot actie
(d) 'aflopen' is een onduidelijke naam

UCD tegenvoorbeeld 3
(g) geen systeemnaam
(i) geen ex.point
(j) 'vul inkt bij' wordt maar 1 keer gebruikt geinclude
(d) 'haal papier op' lijkt niet op actie van de software
(d) 'controle' is een te algemene naam

KD tegenvoorbeeld 1
De associatie naam 'is_een' is onnodig
de klassenamen beginnen niet allemaal met een hoofdletter
Knop2 staat niet in verband met de andere klassen
Knop2 lijkt een erg slecht naam (ook omdat er al een klasse Knop is)
(h) veel associaties zonder naam en zonder multipliciteiten

pijltjes tussen database en gebruiker lijken overbodig
iconen (symbolische weergave van stereotypen) niet consequent
overerving naar beneden
controleren is een werkwoord (en geen zelfstandig naamwoord)
controleren heeft het symbool van een entity- of dataklasse
het is onlogisch dat de database alle andere klassen "kent"
overerving hoeft geen naam (is altijd "is_een")
dubbele pijl tussen database en gebruiker: betekent niets

SD tegenvoorbeeld 1
SD begint niet met (actie van) actor (deze software begint "spontaan")
naam methode bevat spaties!
de message 'drukAf' begint vóór dat de methode verwerk binnen komt en is synchroon
de actor Printer doet niks en krijgt niks
de methode 'doe' heeft een slechte naam en zou asynchroon moeten zijn
de message 'verwerk' loopt van een actor naar een actor: mag niet
er is geen coordinator of controller

SD tegenvoorbeeld 2
De eerste actie 'start()' vindt 'spontaan' plaats vanuit een object: dat kan niet
Bovendien zou deze actie synchroon moeten zijn (het is een aanroep van een methode van controleren)
De actie 'opvragen()' zou *a*synchroon moeten zijn.
Het object 'dummy' is zw geen object (er staat geen dubbele punt voor)
Het object dummy heeft geen verbinding met de rest
Er gebeurt helemaal niets met de aanroep 'buttonPressed()',
en het lijkt er onlogisch om het object 'controleren' deze methode bij het object 'knop' te laten aanroepen. Omgekeerd lijkt logischer.

STD tegenvoorbeeld 1
'werkend' is een zwart gat
'ingesteld' en 'uitgedaan' geven alleen gebeurtenissen weer (geen acties en of condities)
de namen 'ingesteld' en 'uitgedaan' lijken meer op toestanden dan op gebeurtenissen

STD tegenvoorbeeld 2
verwarmen en instellen zijn acties, geen toestanden
ingesteld en warm zijn geen gebeurtenissen maar toestanden
'set(verwarming(hoog)' is een goede actie maar er staat geen gebeurtenis bij
'uit' en 'warm' zijn slechte omschrijvingen van een gebeurtenissen

CD tegenvoorbeeld 1
'20' en 'Schrijven' zijn slechte namen
de interface 'controller' wordt niet geprovide
de interface 'interface 3' (slechte naam) wordt niet gebruikt door een andere componenten
de omvattende component heeft geen naam
voorbereiden is een link die niks doet

Hoofdstuk 13

Andere UML2 diagrammen

Inleiding

UML2 is versie twee van UML. Binnen UML2 zijn er 14 diagrammen. Die 14 diagrammen kun je indelen in 2 soorten namelijk structuurdiagrammen en gedragsdiagrammen, waarbij je met de eerste soort de statische structuur van je systeem kunt weergeven, en met de tweede soort het gedrag van je systeem. Van de eerste soort zijn hier het klasse-, component- en deployment-diagram behandeld. Van de gedragsdiagrammen zijn de use-case-, sequence- en toestandsdiagrammen behandeld en van activiteitsdiagrammen zijn veel voorbeelden gegeven in Hoofdstuk 10. Maar er zijn dus nog 7 diagrammen over die niet behandeld worden. Om jullie toch een indruk te geven wat voor diagrammen dat zijn wordt er van elke soort één diagram met enige toelichting getoond.

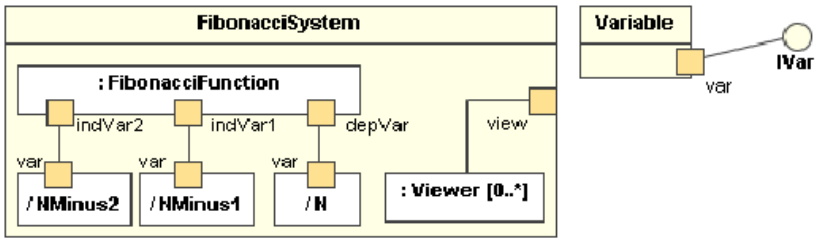
Andere *structuur*diagrammen

Dat zijn het composite-structure-diagram, het objectdiagram, het package-diagram en het profilediagram.

Het composite-structure-diagram

Het composite-structure-diagram, soms ook wel composite-internal-structure diagram is een diagram dat de interne structuur va een klasse laat zien en

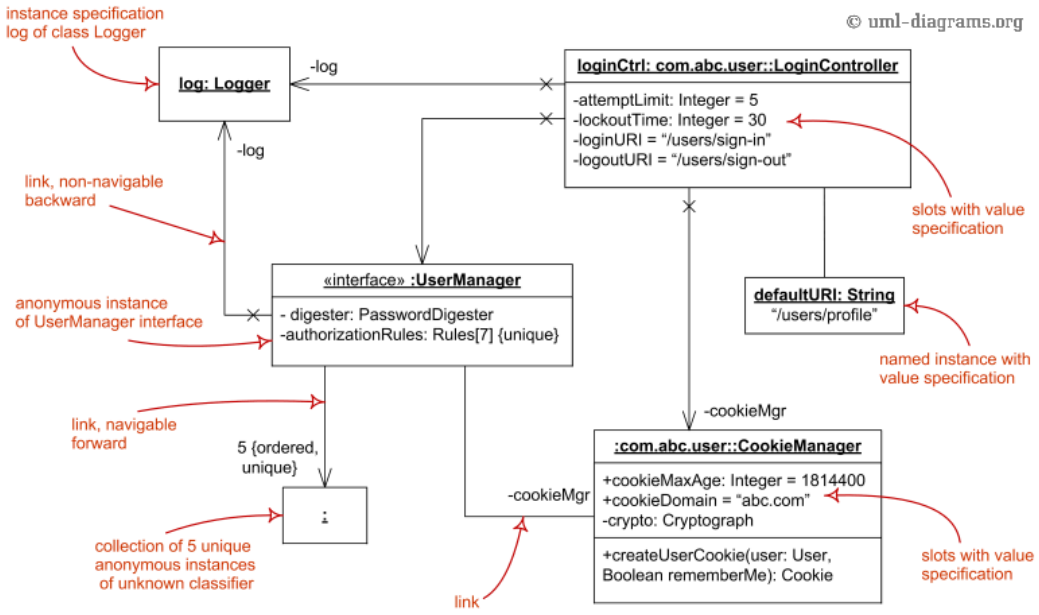
samenwerkingen (collaborations) die de structuur mogelijk maakt. Het lijkt wel wat op een component-diagram, maar dan voor de interne structuur van een klasse.



Figuur 13.1: Composite-internal-structure diagram

Dit diagram specificeert dat de instanties van de "Fibonacci-systeem"-klasse bestaan uit een aantal delen. De bovenste is de classifier 'Fibonacci-Functie'. Drie van die delen worden bepaald door de rol die ze spelen in de instanties van "Fibonacci-systeem": de NMinus2-rol, de Nminus1-rol en de N-rol. Het 5e onderdeel, 'Viewer', geeft de multipliciteit aan. Op runtime kunnen er 0, 1 of meer instanties van Viewer bestaan. Op runtime moeten de instanties de interface |Var kunnen leveren dmv hun var-poorten. Een voorbeeld daarvan staat rechts van het "Fibonacci-systeem". De poort "view" is een non-public poort om toegang te krijgen tot de optionele instanties van Viewer. Figuur overgenomen van wikipedia.org 2018 [6]

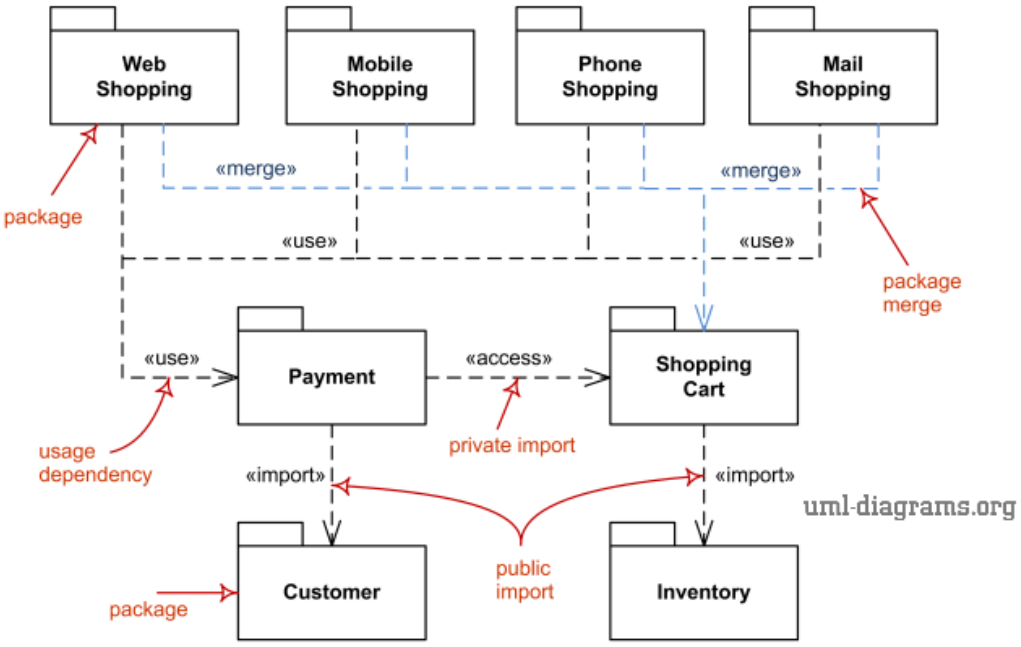
Het objectdiagram



Figuur 13.2: Objectdiagram

Een objectdiagram laat zien welke instanties er zijn en welke verbanden er tussen instanties bestaan. Figuur overgenomen van uml-diagrams.org 2018 [5]

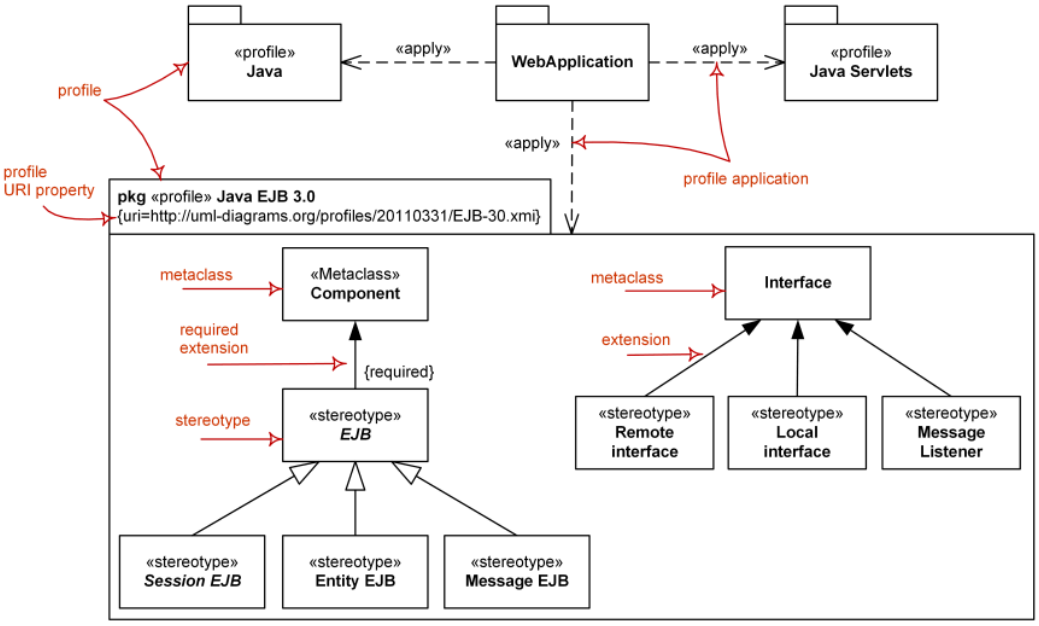
Het packagediagram



Figuur 13.3: Packagediagram

Een packagediagram laat de afhankelijkheden tussen packages (verzamelingen klassen) zien. Figuur overgenomen van uml-diagrams.org 2018 [5]

Het profilediagram



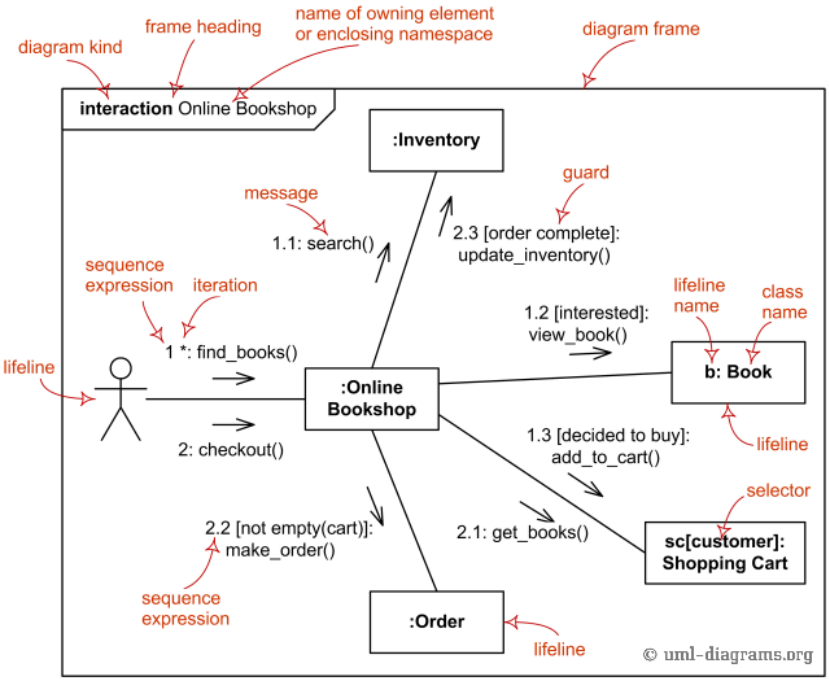
Figuur 13.4: Profilediagram

Een profilediagram is een metamodeldiagram dat verbanden tussen klassen en packages laat zien. Figuur overgenomen van uml-diagrams.org 2018 [5]

Andere *gedrags*diagrammen

De niet behandelde gedragsdiagrammen zijn communicatiediagrammen, interactie-overzicht-diagrammen en timingdiagrammen.

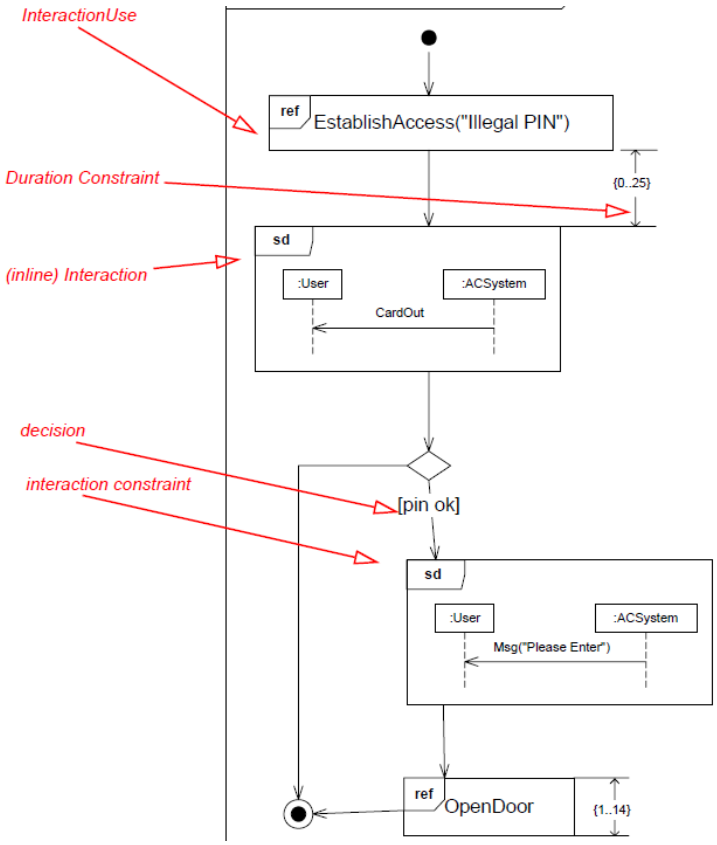
Het communicatiediagram



Figuur 13.5: Communicatiediagram

Een communicatiediagram laat de interactie tussen verschillende objecten zien en is goed vergelijkbaar met een sequence-diagram. Alleen kunnen de objecten op een 'vrije' manier getekend worden en niet 'op een rijtje' zoals in een sequence-diagram. In UML1 werd een communicatiediagram een collaboratiediagram genoemd. Figuur overgenomen van uml-diagrams.org 2018 [5]

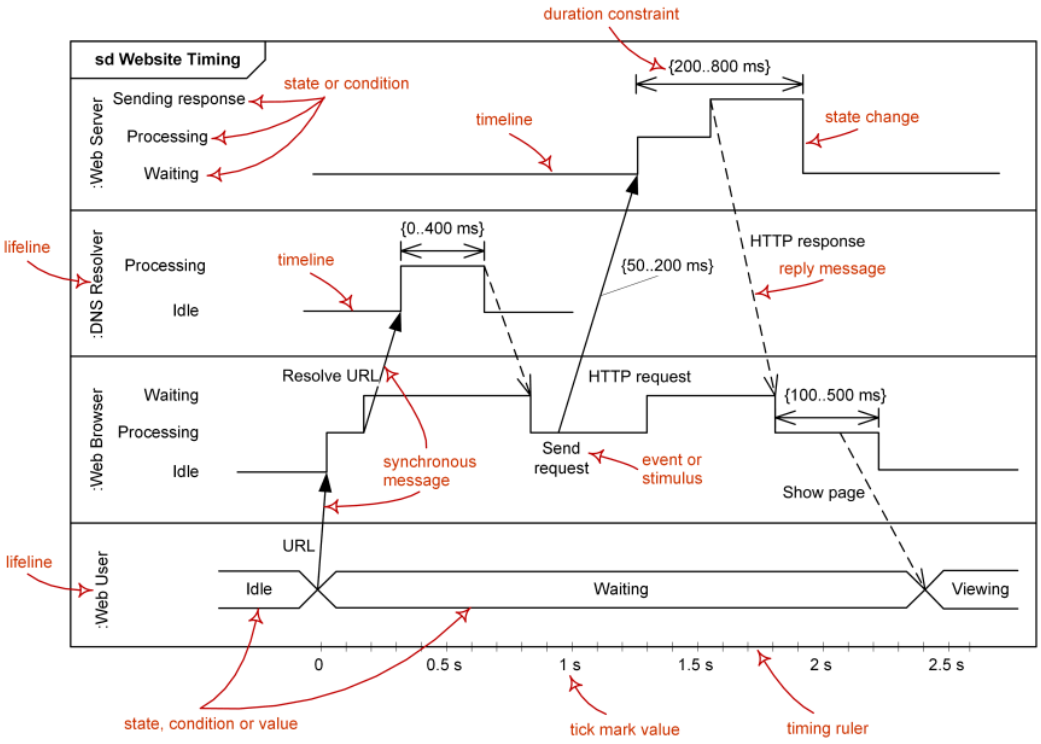
Het interactie-overzicht-diagram



Figuur 13.6: Interactie-overzicht-diagram

Een interactie-overzicht-diagram lijkt op een activiteitendiagram. Beide geven activiteiten in een beaalde volgorde weer. Het verschil is dat een interactie-overzicht-diagram nesting mogelijk maakt. Dat wil zeggen dat een activiteit in een raam geplaatst wordt wat zelf weer een reeks activiteiten kan bevatten. Zodoende kun je een interactie-overzicht-diagram gebruiken om een complex scenario te visualiseren. Figuur overgenomen van NobleProg 2017 [4]

Het timingdiagram



Figuur 13.7: Timingdiagram

Een timingdiagram is een speciale vorm van een sequence-diagram. De tijd loopt hier niet van boven naar beneden, maar van links naar rechts. De levenslijnen van de verschillende objecten worden in aparte lanen getoond. Figuur overgenomen van uml-diagrams.org 2018 [5]

lijst van figuren

Bibliografie

[1] Ambler, Scott W.
http://www.agilemodeling.com/artifacts/deploymentDiagram.htm

[2] Kruchten, Philippe *Architectural Blueprints — The "4+1" View Model of Software Architecture.*, IEEE Software 12 (6), pp. 42-50, 1995.

[3] Mountaingoat Software
http://www.mountaingoatsoftware.com/agile/user-stories

[4] NobleProg training materials
https://training-course-material.com/training/UML_interaction_Overview_Diargam

[5] uml-diagrams.org
https://www.uml-diagrams.org/timing-diagrams.html

[6] wikepedia.org
https://en.wikipedia.org/wiki/Composite_structure_diagram

lijst van tabellen