KU LEUVEN

Ontwikkeling van bedrijfstoepassingen (B-KUL-D0I60A)

Professoren
Monique Snoeck
Guido Dedene

Auteurs
Jonas Devlieghere

Bijdragers
n.v.t.

Inhoudsopgave

Voorbeschouwing

De bedoeling van dit document is het gestructureerd samenvatten van de inhoud omvat in de cursus *Ontwikkeling van berijfstoepassingen* van professor Monique Snoeck. Dit omvat hoofdstukken 3 tot en met 5 van de cursus *ontwikkeling van bedrijfstoepassingen*, zoals deze gegeven werd in het academiejaar 2011-2012.

Deze tekst is gebaseerd op de cursustekst van professor Snoeck. Ik heb van haar de toestemming deze samenvatting te delen met de studenten die dit vak opnemen.
Dit document werd met de grootst mogelijke zorg samengesteld. Desondanks garandeert de auteur de correctheid van dit document niet. Een laatste versie kan teruggevonden worden op `https://github.com/KULeuven-CS/OBT`.

Hoofdstuk 1

Algemeen kader

De hoofdstukken behandeld door professor Snoeck beslagen de analysefase binnen het ontwerp van bedrijfstoepassingen. Deze fase bevindt zich onder volgende fase die overlopen worden bij dergelijke ontwikkeling.

1.1 Ontwikkelingsfasen

1. **Definitiefase**
   - Project wordt afgebakend
   - Er wordt een **business-case** opgesteld (kosten/baten analyse)
   - Verschillende alternatieven worden bekeken
2. **Analysefase**
   - Vereisten worden in kaart gebracht
   - Onderscheid tussen **high-level analyse** en **gedetailleerde analyse**
   - Onderscheid **requirements gathering** en **requirements modelling**
3. **Ontwerpfase**
   - Er wordt ontwerp gemaakt van het systeem
   - Gebaseerd op de gedetailleerde analyse
4. **Bouwfase**
   - Programmeren en testen van het systeem
5. **Testfase**
   - Formele testfase
   - Acceptatietest uitgevoerd door gebruikers
6. **Gebruiksfase**
   - Wordt ook **productiefase** genoemd

- Zodra het systeem formeel aanvaard is door opdrachtgever

7. **Onderhoudsfase**
   - Zodra het systeem in gebruik genomen is
   - Verder op punt stellen op basis van problemen/vragen van gebruikers

De manier waarop men bovenstaande fasen doorloopt is afhankelijk van de gekozen **systeemontwikkelingsmethode**. Enkele populaire methodes zijn **Waterfall**, **eXtreme programming**, **Agile**, **RUP**.

1.2 Analysefase

De focus van dit deel van de cursus ligt op de **Analysefase**. Binnen deze fase zijn twee onderdelen te onderscheiden: **verzamelen van requirements** en **modelleren van requirements**.

Requirements gathering De businessanalist probeert zich een inzicht te vormen van de dominante activiteiten. Pas wanneer deze hierover beschikt kan hij de vereisten formeel modelleren.

Requirements modelling De analist kan nu de functionele vereisten op een gedetailleerde manier vastleggen. Hierdoor kunnen inconsistenties en onvolledigheden worden opgemerkt.

Hoofdstukken 2 en 3 in deze samenvatting (respectievelijk 3 en 4 in de cursus) gaan over het modelleren. In hoofdstuk 4 (5 in de cursus) wordt er vanuit een breder perspectief naar de belangrijke aspecten van het analyseproces in zijn geheel gekeken.

Hoofdstuk 2

Static Business Modelling

Statisch modelleren geeft een tijdsonafhankelijk beeld van het systeem. Dit wil zeggen dat hier structuren worden besproken die aanwezig zijn in het probleemdomein.

2.1 Wat is modelleren

Er zijn twee manieren om software te definieëren.

- **Engineering persective:** Software is an artefact that we will design, build, test and deploy, We rely on testing and inspection for validation.
- **Mathematical perspective:** Software is a mathematical abstraction, a theory which can be analyzed for consistency and can be refined to a more specialized theory.

2.2 Engineering Perspective

2.2.1 Het probleem

De rede dat we dergelijk **artefact** bouwen is dat het een oplossing is voor een probleem. Het is de taak van de analist om te onderzoeken welk probleem moet opgelost worden. Hiervoor worden **sleutelgebruikers** geïnterviewd.

> Gezien we geen gebruikers ter onzer beschikking hebben in deze cursus worden problemen aangereikt onder de vorm van een **use case**.

De informatie waarop de analist zich baseert heeft een bepaald karakter. De **use case** heeft het eerstgenoemde karakter.

Exemplaristisch karakter De gebruiker geeft een voorbeeld van hoe het systeem zal moeten werken.

Prototypisch karakter De gebruiker geeft een algemeen geldende beschrijving van de gewenste functionaliteit.

> **VOORBEELD**: *exemplaristische use case*
>
> Ik wens een systeem dat mij toelaat om $1 + 1$ te berekenen.
> Ik wens een systeem dat mij toelaat om $2 + 2$ te berekenen.
> Ik wens een systeem dat mij toelaat om $5 + 5$ te berekenen.

Een goed systeem is in staat **alle** exemplaristische use cases te ondersteunen. Een ideaal systeem is bovendien in staat om alle toekomstige uses cases ook te ondersteunen.

2.2.2 De oplossing

Nu er kennis is van het probleem, zullen we denken aan de oplossing. Hiervoor maken we een schets of plan van hoe die oplossing er uit zal zien, wat we het **model** noemen. Dit model heeft twee functies.

- **Probleemdomein beschrijven:** Enerzijds moet het model de noden van de gebruiker kunnen voorstellen. Dit is immers een onderdeel van zowel het probleem als de oplossing.
- **Oplossing beschrijven:** Anderzijds willen we graag dat het model een blauwdruk vormt voor de oplossing.

2.2.3 Het lastenboek

In deze cursus ligt de nadruk op het correct beschrijven van het probleemdomein. Hiervoor nemen we het perspectief in van de opdrachtgever of eindgebruiker. Dit komt overeen met het **lastenboek**.

Het lastenboek Het lastenboek beschrijft alle vereisten die de opdrachtgever heeft ten aanzien van het te bouwen systeem.

De vereisten vallen uiteen in twee categorieën:

Functionele vereisten Deze beschrijven **wat** het systeem moet kunnen vanuit het perspectief van de eindgebruiker.

Niet-functionele vereisten Deze beschrijven **de manier** waarop iets moet worden gerealiseerd. Typisch in termen van kwaliteit of performantie.

> **VOORBEELD**: *functionele vereisten*
>
> - De mogelijkheid om gegevens te kunnen opvragen
> - Mogelijkheid tot het afdrukken van rapporten

> **VOORBEELD**: *niet-functionele vereisten*
>
> - Responstijd moet niet meer dan 5 ms bedragen
> - GUI moet voldoen aan *any surfer labe*

De focus ligt in deze cursus op het in kaart brengen van de **functionele vereisten**. Hoewel dezelfde notaties als voor het **systeemontwerp** (dus het model van de oplossing) gebruiken, wordt gebruikt, gaat de aandacht in de eerste plaats naar het modelleren van het probleem. Dit omdat de kwaliteit van het lastenboek van **primoridaal belang** is.

Een goed begrip van het probleem domein is op twee manieren belangrijk:

- Het draagt bij tot de kwaliteit van het lastenboek
- Het verwerven van inzicht in het probleemdomein biedt de mogelijkheid tot het voorstellen van mogelijke verbeteringen.

2.2.4 Model van het probleemdomein

Het model van het probleemdomein beschrijf de **universe of discourse** op een prototypische manier. Het is een abstracte beschrijving die geldt voor de feiten in het domein. Er zijn twee niveau's van denken:

- **Level 1: Het modelniveau**
  - Een persoon
  - Een product
  - Een opleidingsonderdeel
- **Level 0: Het voorbeeldniveau**
  - Jan, Piet, Els
  - Intel Core i5
  - Ontwikkeling van bedrijfstoepassingen

Modelleren is abstraheren: het is de overgang maken **level 0 → level 1**

2.3 Mathematical Perspective

2.3.1 Abstractie

Het onderscheid tussen de levels kan in de wiskunde gemaakt worden aan de hand van verzamelingen:

- **Level 0:** de elementen van de verzameling
- **Level 1:** de definitie van de verzameling

> **VOORBEELD**: *levels als verzamelingen*
>
> PERSOON = $\{p \mid p \text{ is een persoon}\}$
> PERSOON = {Jan, Piet, Els}

Het modelleren gebeurt door verzamelingen te definiëren en de eigenschappen van de elementen in die verzameling te beschrijven.

2.3.2 Validatie

We kunnen het model op verschillende manieren valideren:

- **Instantiëren:** kunnen we elementen vinden die tot de verzameling behoren? Kloppen de verbanden die we hebben gedefinieerd met de werkelijkheid? Maken de overgang level 1 $\rightarrow$ level 0

  = externe validatie: controle of het model een correcte weergave is van de werkelijkheid.

- **Redeneren:** axioma's en stelling toepassen uit de verzamelingsleer.

  = interne validatie: controle of het model op zichzelf consistent is.

2.4 Modelleringstaal

Het neerschrijven van het model of het lastenboek kan op verschillende manieren.

- **Natuurlijke taal:** Vaak enkel voor de eerste inventaris van de wensen van de gebruiker. Eenvoudig maar niet efficiënt.
- **UML:** Industriestandaard voor het beschrijven van een informatiesysteem. In eerste plaats bedoeld voor het beschrijven van de oplossing, maar kunnen ook het probleemdomein beschrijven.

Om de gebruiker ook in staat te stellen het model te begrijpen, beperken we ons tot een subset van de beschikbare technieken. We laten wat bedoeld is voor de technische implementatie.

We zullen de vertaling maken naar **algebra**. We gebruiken het wiskundig perspectief om

- **Betekenis** van het schema te definiëren
- **Abstract denken** te stimuleren
- Modellen te **valideren**

2.4.1 Terminologie

Modelleringstaal Geheel van technieken bedoeld voor het neerschrijven van (software-) specificaties.
(Bijvoorbeeld **UML**)

Modelleringstechniek Samenhangend geheel van symbolen en regels die toelaten een bepaald aspect van de software te modelleren.
(Bijvoorbeeld **UML Class Diagram**, **ER Modelling**, **Data Flow Diagram**)

Diagramma, schema of model Model of tekening opgesteld volgens een bepaalde voorafbepaalde techniek.
(Bijvoorbeeld **UML klassendiagram**)

2.4.2 Klassen en objecten

- **Wiskundige voorstelling:** We noteren de verzameling als een **ovaal**. De elementen in de verzameling vormen **punten** in deze ovaal.
- **UML:** Verzamelingen worden voorgesteld als **rechthoeken** met daarin de naam van de verzameling. We noemen de verzameling een **klasse** en de elementen **objecten**.

2.4.3 Attributen

Bij het definiëren van een verzameling, beschrijven we de elementen van die verzameling aan de hand van hun eigenschappen. Deze worden de **attributen** van het object genoemd.

- Eigenschappen die object altijd bezit
- Attribuutnaam ligt vast
- Waarde kan eventueel veranderen

2.5 Associaties

2.5.1 Principe

Elementen van een verzameling kunnen een **verband** hebben met andere elementen uit een andere verzameling.

Wiskundige benadering

We kunnen de verbanden definiëren als afbeeldingen tussen verzamelingen.

Afbeelding Een afbeelding $f : A \to B$ is een functie die een element uit de verzameling van A afbeeldt op elementen van verzameling B. A is het **domein** van de afbeelding f en B is het **beeld** van de afbeelding f.

$$f : A \to B : a \mapsto f(a) \text{met } a \in A \text{ en } f(a) \subseteq B$$

Het verband kan ook in de omgekeerde richting worden beschouwd. Vaak heeft het verband in elke richting een eigen naam.

UML benadering

We noemen het verband tussen twee of meer klassen een **associatie**. Deze wordt weergegeven als een **lijn** met de naam van het verband naast de lijn geschreven. Wanneer het verband in beide richting een naam krijgt, noemt men dit de **rollen** van de associatie.

> **VOORBEELD**: *rollen*
>
> Veronderstel twee klassen: werknemer en departement.
> $\to$: Werknemer *werkt in* departement
> $\leftarrow$: Departement *heeft* werknemer

2.5.2 Karakter van een verband

Verplicht/optioneel karakter (Wiskunde)

Een verband kan ofwel verplicht of optioneel zijn.

- **Verplicht:** Een afbeelding is verplicht indien elk element van A verplicht een beeld in B heeft.

$$\boxed{f : A \to B \text{ is verplicht} \Leftrightarrow \forall a \in A : f(a) \neq \emptyset}$$

- **Optioneel:** Een afbeelding is optioneel indien niet elk element van A verplicht een element in B heeft.

Het **verplicht** of **optioneel** karakter wordt gedefinieerd voor elk object op elk moment. Deze verandert **niet** doorheen de tijd. De interpretatie die hier gegeven wordt aan een verplicht verband wordt in de temporele logica gemodelleerd met de $\forall$-kwantor.

UML maakt geen gebruik van temporele kwantoren. Een verplicht verband moet **altijd** geldig zijn. Uitspraken met de $\exists$-kwantor zijn niet mogelijk.

kardinaliteit van een verband (UML)

kardinaliteit De kardinaliteit van een verband f zegt wat het maximum aantal elementen is waarop een element uit het domein wordt afgebeeld door f.

- **kardinaliteit maximum 1**
  Een verband $f : A \to B$ heeft een **carinaliteit van maximum 1** indien elke $a \in A$ op maximum één $b \in B$ wordt afbeeld.

$$\boxed{f : A \to B \text{ heeft een kardinaliteit van maximum 1} \Leftrightarrow \forall a \in A : |f(a)| \leq 1}$$

- **kardinaliteit many**
  Een verband $f : A \to B$ heeft een **carinaliteit van many** indien elke $a \in A$ mogelijks op meerdere elementen $b_i \in B$ wordt afbeeld.

$$\boxed{f : A \to B \text{ heeft een kardinaliteit van many} \Leftrightarrow \forall a \in A : |f(a)| \in \mathbb{N}}$$

Het verplicht of optioneel karakter wordt dan als volgt weergegeven:

$f : A \to B$		Schrijf langs f, aan de kant van B
f is optioneel	f heeft carinaliteit 1 $\|f(a)\| \leq 1$	[0..1]
f is optioneel	f heeft carinaliteit many	[0..*] (afgekort *)
f is verplicht $\|f(a) > 0\|$	f heeft carinaliteit 1 $\|f(a)\| \leq 1$	[1..1] (afgekort 1) $\|f(a)\| = 1$
f is verplicht	f heeft carinaliteit many	[1..*]

2.5.3 Navigeren over meerdere associaties

Waneer het beeld van het ene verband f gelijk is aan het domein van een ander verband g kunnen we verbanden **combineren** tot één verband $g \bullet f$

$$f : A \to B \text{ en } g : B \to C \text{ dan geldt } g \bullet f : A \to C : a \mapsto g(f(a))$$

Het optioneel of verplicht karakter en de kardinaliteit van het **samengesteld verband** kunnen afgeleid worden uit de eigenschappen van de deelverbanden.

> **VOORBEELD**: *samengesteld verband*
>
> Veronderstel dat f kardinaliteit 1 heeft en g kardinaliteit many.
> Dan geldt $\forall a \in A$ dat er maximum één element zit in $f(a)$, waarbij $f(a) \subseteq B$. Voor elk element $b \in B$ en dus ook $b \in f(a)$ geldt dat er mogelijks meerdere elementen zijn in $g(b)$. We kunnen besluiten dat er in $g(f(a)) = (g \bullet f)(a)$ ook mogelijks meerdere elementen zitten.

2.6 Speciale vormen van associaties

In plaats van relaties te beschouwen als afbeeldingen, kunnen deze ook worden gemodelleerd als een **productverzameling** van twee of meerdere verzamelingen.

Een relatie R tussen een klasse A en een klasse B wordt gedefinieerd als:

$$\boxed{R \subseteq A \times B}$$

De overeenkomstige afbeelding is dan:

$$\boxed{r : A \to B : a \mapsto f(a) \text{ met } b \in f(a) \Leftrightarrow (a, b) \in R}$$

Het karakter:

- **Verplicht:** r is verplicht $\Leftrightarrow \forall a \in A : \exists b \in B : (a, b) \in R$
- **Kardinaliteit één:** r heeft kardinaliteit $1 \Leftrightarrow |\{b \in B | (a, b) \in R\}| \leq 1$

Deze manier van definiëren maakt duidelijk dat een element slechts éénmaal kan voorkomen in een relatie. Algemeen kunnen we zeggen dat elk element (a, b) in een productverzameling $A \times B$ uniek geïdentificeerd wordt door middel van a en b.

Het is wel toegelaten twee verbanden (met een verschillende naam) te leggen tussen twee dezelfde klassen A en B.

2.6.1 N-aire associaties

Met bovenstaande definitie is het eenvoudig verbanden te leggen tussen meer dan twee klassen. Het is soms zo dat het niet mogelijk is een verband tussen meerdere klassen weer te geven aan de hand van verschillende **binaire relaties**. We kunnen dan gebruik maken van een **N-aire relatie**.

Gegeven een verband $R \subseteq (A_1 \times A_2 \times ... \times A_n)$. We definiëren voor elke A_i een verband r_i als volgt:

$$\boxed{r_i : A_i \to A_1 \times ... \times A_{i-1} \times A_{i+1} \times ... \times A_n : a_i \mapsto (a_1, ..., a_{i-1}, a_{i+1}, ..., a_n)}$$

Het karakter:

- **Verplicht:**
  r is verplicht $\Leftrightarrow \forall a_i \in A_i : \exists (a_1, ..., a_{i-1}, a_{i+1}, ..., a_n) \in (A_1 \times ... \times A_{i-1} \times A_{i+1} \times ... \times A_n)$
- **Kardinaliteit één:**
  r heeft kardinaliteit 1 $\Leftrightarrow |\{(A_1 \times ... \times A_{i-1} \times A_{i+1} \times ... \times A_n) | (a_1, ..., a_{i-1}, a_{i+1}, ..., a_n) \in R\}| \leq 1$

VOORBEELD: *ternaire relatie*

We beschouwen het verband tussen *product*, *levernacier* en *project*: een leverancier levert een bepaald product voor een bepaald project. Veronderstel dat we dit modelleren aan de hadn van drie binaire relaties:

- Leverancier-*levert*-product
- Leverancier-*levert_voor*-project
- Project-*gebruikt*-product

Deze verbanden zijn onvoldoende omdat:

- Met de combinatie *project-leverancier* komen mogelijks meerdere producten overeen
- Ook met de combinatie *project-product* kunnen meerdere leveranciers overeenkomen
- Tenslotte kan ook met combinatie *leverancier-product* meerdere projecten overeenkomen

Het is niet mogelijk het probleemdomein correct weer te geven op deze manier. Met een ternaire relatie kunnen we dit wel. Het verband:

$$\boxed{\text{gebruikt} : \text{PROJECT} \to \text{LEVERANCIER} \times \text{PRODUCT}}$$

modelleert welk product van een bepaalde leverancier werd gebruikt in welk project.

2.6.2 Associaties als klasse

Het is mogelijk dat een verband tussen twee klassen zelf als een klasse wordt gemodelleerd. Zie hiervoor 4.4.7 in het handboek.
In volgende situaties kiezen we om de associatie als klasse voor te stellen:

- Soms geeft de identiteit van de deelnemende objecten geen unieke identificatie van de relatie, en zijn bijkomende attributen nodig. Wanneer een verband eigen karakteristieken nodig heeft voor het uniek identificeren van de elementen in de associatie, modelleert men deze als een klasse.

- Soms zijn de deelnemende objecten wel voldoende om het relatie-object uniek te bepalen, maar zijn er desondanks nuttige attributen voor de associatie. Ook dan modelleren we deze als een klasse.

Bovendien wordt er standaard gekozen om een associatie voor te stellen als klasse in volgende gevallen:

- **Many-to-many verbanden**
  - Afkomstig van normalisatie bij relationele gegevensbanken.
  - Oorspronkelijke deelnemende klassen zijn niet altijd voldoende als identificatie
  - Altijd nagaan of er attributen moeten worden toegevoegd voor deze uniciteit
- **N-aire verbanden met** $N > 2$
  - Eveneens afkomstig uit databankontwerp
  - Enkel binaire verbanden in relationeel model

Het invoeren van een **association-as-class** maakt de modelleringstaal rijker naar introduceert ook een vorm van complexiteit. Omdat de toegevoegde waarde vanuit ons standpunt klein is, worden dergelijke klassen opgenomen als *volwaardige klassen* in het schema.

2.6.3 Aggregatie en compositie

Aggregatie Aggregaties zijn een speciaal type associaties waarin de twee deelnemende klassen geen gelijkwaardige status hebben, maar een *geheel-deel* relatie vormen. Een aggregatie beschrijft hoe de klasse die de rol van geheel heeft, samengesteld wordt uit de andere klassen, die de rol van deel hebben. Voor aggregaties heeft de klasse die optreedt als geheel, altijd de multipliciteit ëën.
(Worden voorgesteld in UML door transparante ruit.)

Compositie Composities zijn associaties die zeer sterke aggregaties vertegenwoordigen. Dit betekent dat composities evengoed geheel-deel relaties vormen, maar de relatie is zo sterk dat de delen niet op zichzelf kunnen bestaan. Zij bestaan slechts binnen het geheel, en als het geheel vernietigd wordt, gaan de delen er ook aan.
(Worden voorgesteld in UML door ingekleurde ruit.)

Het gebruik van dergelijke structuren wordt afgeraden.

2.7 Overerving

Zie hiervoor ook 4.2.8 in het handboek.

2.7.1 Definitie

Wanneer verschillende klassen sterke gelijkenissen vertonen kunnen we proberen die te verzamelen in een overkoepelende klasse. De oorspronkelijke klassen erven dan de gemeenschappelijke eigenschappen over en kunnen de definitie ervan eventueel verfijnen.

- Vooral belangrijk voor het vermeiden van duplicatie van code bij OOP.
- Nauw verwant met concept van taxonomie

Wanneer wel gebruiken

Omdat het model ook het oplossingsdomein moet omschrijven en we dus later willen omzetten naar code, moeten we de principes omtrent overerving op het niveau van programmacode respecteren. We gebruiken dit principe daarom enkel als:

- Er geen overlappende subklassen zijn
- Elementen gedurende volledige levensduur tot dezelfde subklasse behoren.

Wanneer niet gebruiken

Indien de elementen wel van subklasse veranderen, kunnen deze best worden gemodelleerd aan de hand van rollen. Gedragsaspecten worden immers beter gemodelleerd aan de hand van operaties en toestanden.

Algemeen zijn volgende redenen **niet goed** om subklassen te definieren:

- Deelnemen aan verband met een andere klasse
- Opdelen van klassen in functie van de waarde van een attribuut
- Opdelen van klassen in functie van de toestand van het object

In al deze situaties zouden elementen van subklassen kunnen veranderen.

2.7.2 Overerving associaties

De definitie van een **generalisatie-specialisatieverband** tussen twee klassen impliceert dat de gespecialiseerde klasse alle eigenschappen van de generalisatieklasse overerft.

De specialisatie erft dus zowel de attributen als de associatie van de generalisatieklasse.

2.8 OCL Constraints

Schema's kunnen beperkingen opleggen waar de gegevens aan moeten voldoen. Dergelijke beperkingen of **constaints** vormen een krachtig instrument om de correctheid van het schema te bewaken.

> **VOORBEELD**: *schemabeperkingen*
>
> We legden al beperkingen op met het verplicht maken van associaties. Een gift kon niet geregistreerd worden als die niet verbonden was met een persoon die deze gift deed.

UML laat toe beperkingen op te leggen die niet omvat kunnen worden in het schema. Het formuleren van dergelijke constraints gebeurt aan de hand van **OCL**, de **Object Constraint Language**.

Zie hiervoor ook hoofdstuk 7 in het handboek.

2.8.1 Tips bij het maken van oefeningen

1. Geef de context, een klassenaam die in het schema voorkomt
2. Geef de operatie waarop een **preconditie** van toepassing is
3. Controleer of het resultaat een Booleaanse waarde geeft
4. Controleer de kardinaliteiten van de associaties waarlangs wordt genavigeerd
5. Zoek een goed startpunt voor navigatie
6. Controleer de opgave voor woorden zoals **een** welke verschillen van **alle**

Hoofdstuk 3

Dynamic Business Modelling

Statisch modelleren geeft in tegenstelling tot static modeling wel een tijdsafhankelijk beeld van het systeem. Het beschrijft het gedrag van het systeem doorheen de tijd.

3.1 Dynamische diagrammen UML

Binnen UML worden volgende diagrammen gebruikt bij het **dynamisch modelleren**:

- **Use Case Diagrammen**
  Deze beschrijven de manier waarop een gebruiker het systeem wenst te gebruiken. Het beschrijft dus problemen waar de gebruiker een oplossing voor wenst.
  - Beschrijft het probleem: functionele vereisten
  - Kan niet gebruikt worden voor model van oplossing
    (omvat immers geen toekomstige use cases)
- **Sequentie Diagram**
  Beschrijft hoe objecten in het systeem samenwerken.
  - Beschrijf de oplossing
- **Toestandsdiagrammen**
  Deze worden gebruikt om de **toestanden** waarin een object zich kan bevinden te beschrijven. Het legt ook vast wanneer welke overgangen mogelijk zijn.
  - Beschrijven probleemdomein
  - Beschrijven oplossing
- **Activiteitsdiagrammen**
  Procesgerichte beschrijvingen om het gedrag van een systeem, bestaande uit meerdere objecten, te modelleren.
  - Workflow modelleren
  - Concretiseren use case

- **Interactiediagrammen**
  Deze worden gebruikt voor het modelleren van de interacties tussen de objecten.
  - Gebonden aan OOP
  - Beschrijven oplossingsdomein
  - Om ze toch te gebruiken in probleemdomein: **Object-Event tabel**

3.1.1 Positioneren technieken

	Problem Domain		Solution Domain
	Domain Knowledge (*)	IS FUnctionality (**)	(***)
Class Diagram	Y	Y	Y
Use Case Diagram		Y	Y
Sequence Diagram			Y
Finite State Diagram	Y		Y
Activity Diagram (BPMN)	Y		Y

*: Rij 2 van raamwerk Zachman

**: Rij 3 van raamwerk Zachman

***: Rij 4,5,6 van raamwerk Zachman

3.1.2 Diagramtechnieken in deze cursus

- **Domeinmodel**
  - Klassendiagrammen
  - Toestandsdiagrammen
- **Functionaliteit**
  - Use cases
- **Ontwerp**
  - Sequentie- en interactiediagrammen

3.2 Use Case diagrammen

Zie ook hoofdstuk 8 in het handboek

Use cases bieden een **gebruikersperspectief** op het systeem. Ze worden gebruikt voor **requirements elicitation**: de techniek tracht de functionele vereisten van het systeem te achterhalen.

- Snijdt doorheen alle architectuurlagen heen
- Geen goede basis voor systeemontwerp

3.2.1 Controleren

Bij het opstellen van een use case kunnen we volgende criteria gebruiken om de volledigheid van de analyse te controleren:

1. Elke actor heeft minstens één use case om met het systeem te interageren.
2. Elke bedrijfsklasse heeft een use case waarmee men de lijst van objecten van die klasse kan opvragen
3. Elke bedrijfsklasse heeft een use case waarmee men de details van een gegeven object uit de klasse kan opvragen
4. Elke bedrijfsgebeurtenis heeft een use case waarmee men de uitvoering kan triggeren

3.3 Toestandsdiagrammen

Toestandsdiagrammen vinden hun oorsprong in de theorie van **eindige toestandsautomaten**.

3.3.1 Concepten

Toestand Een toestand definieert een stabiele toestand waarin een object zich kan bevinden. Het wordt weergegeven door een rechthoek met afgeronde randen.

- **Begintoestand**: De toestand die vooraf gaat aan de creatie van een object en wordt weergegeven als een zwarte cirkel
- **Eindtoestand**: Toestand waarin elk object belandt bij het beeindigen van zijn levenscyclus. Dit wordt weergegeven zwarte gevulde cirkels met rand.

Transitie Een transitie is een toestandsovergang, tussen twee verschillende toestanden of tussen een toestand en zichzelf.

Activiteit Een activiteit is een operatie die uitgevoerd wordt binnen de tijd dat een object zich in één toestand bevindt. Ze stopt vanzelf of wanneer er een overgang naar de volgende toestand is.

- Binnen de toestand achter het label "do/ "

Guard Een voorwaarde die aangeeft of een transitie plaats kan vinden. Het resultaat is een booleaanse waarde.

- Expressie tussen vierkante haken

Actie of Event Atomaire operatie (kost logisch gezien geen tijd). Ofwel wordt de volledige operatie uitgevoerd, ofwel niet.

- *event* /^ *actie* bij boven de overgangspijl

3.3.2 Event

Een transitie wordt veroorzaak door een event. Dit zullen in het geval van business objecten steeds reële-wereldgebeurtenissen zijn. Zoals reeds gezegd kost een event geen tijd.

> **VOORBEELD**: *event*
>
> Het afhalen van geld bij een bank neemt enige tijd in beslag. De overgang zou in dit geval kunnen zijn dat de klant in *het rood* komt te staan. We moeten hier abstractie maken van de tijdsduur.

Wanneer het niet mogelijk is om het tijdsbeslag te abstraheren kunnen we het begin en einde van het event apart modelleren.

> **VOORBEELD**: *opdelen event*
>
> Een speeltijd op een school kunnen we modelleren als event *StartSpeeltijd* en *StopSpeeltijd*.

Een speciaal geval is het **lege event**, deze wordt in automatentheorie aangegeven met het "&-symbool en veroorzaak automatisch een overgang.

Met toestandsautomaten kunnen we de klassieke voorstellen aan de hand van:

- Sequentie
- Keuze
- Iteratie

3.3.3 Kwaliteitscontrole

Het concept werd zoals eerder gezegd al uitgebreid bestudeerd. We zullen deze theorie gebruiken om de kwaliteit van onze toestandsautomaten te onderzoeken. Dit wordt standaard niet omvat in UML.

We wensen dat onze automaten:

- Vrij zijn van onbereikbare toestanden (voor- en achterwaarts)
- Deterministisch zijn

3.3.4 Toestandsautomaten

We definieëren een **eindige toestandsautomaat** of **finite state machine** (FSM) als een tupel:

$$\boxed{\mathrm{M} = (\Sigma, Q, \Delta, q_0, F)} \tag{3.1}$$

- Σ de verzameling events
- Q de verzameling toestanden
- Δ De transitiefunctie $\Delta : Q \times \Sigma \to Q : (q, a) \mapsto q'$
- q_0 De initiële toestand
- $F \subseteq Q$ De verzameling eindtoestanden

3.3.5 Onbereikbare toestanden

Voorwaartse onbereikbaarheid

Een toestandsautomaat $M = (\Sigma, Q, \Delta, q_0, F)$ heeft geen **voorwaarts onbereikbare toestanden** indien elke toestand vanuit de begintoestand bereikbaar is.

Wiskundig betekent dit:

$$\forall q \in Q, \exists a_1, a_2, ..., a_n \in \Sigma, \exists q_1, q_2, ..., q_{n-1} \in Q$$

zodat

$$\Delta(q, a_1) = q_1, \Delta(q_1, a_2) = q_2, ..., \Delta(q_{n-1}, a_n) = q$$

Achterwaartse onbereikbaarheid

Een toestandsautomaat $M = (\Sigma, Q, \Delta, q_0, F)$ heeft geen **achterwaarts onbereikbare toestanden** indien vanuit elke toestand een eindtoestand kan bereikt worden.

Wiskundig betekent dit:

$$\forall q \in Q, \exists a_1, a_2, ..., a_n \in \Sigma, \exists q_1, q_2, ..., q_{n-1} \in Q, \exists q_f \in F$$

zodat

$$\Delta(q_0, a_1) = q_1, \Delta(q_1, a_2) = q_2, ..., \Delta(q_{n-1}, a_n) = q_f$$

3.3.6 Determinisme

Een toestandsautomaat $M = (\Sigma, Q, \Delta, q_0, F)$ is deterministisch indien er gegeven een bepaalde toestand voor elk event maximum één volgende toestand mogelijk is.

Wiskundig betekent dit:

$$\forall q \in Q, \forall a \in \Sigma : \neg(\exists q_1, q_2 \in Q : \Delta(q, a) = q_1 en \Delta(q, a) = q_2)$$

Een toestandsautomaat die niet aan deze voorwaarde voldoet wordt een **niet-deterministische automaat** genoemd. Wiskundig valt aan te tonen dat elke niet-deterministische automaat een deterministische equivalent heeft. Dit is eventueel mogelijk door het gebruik van **guards**.

3.3.7 Minimale automaten

Automaten die geen lege transacties hebben worden **empty-free** genoemd. Opnieuw is aantoonbaar dat elke automaat met lege transacties omgevormd kan worden tot een equivalent zonder.

Minimale automaat Een automaat die zowel empty-free is als deterministisch.

3.3.8 Gestructureerde automaten

Grote automaten worden nogal eenvoudig chaotisch. Door gebruik te makken van horizontale of verticale assen als indicatoren van de voortgang kunnen we dergelijke automaten terug overzichtelijk maken.

- As stelt voortgang van begin naar eindtoestand voor
- Begin- en eindtoestand bevinden zich op uiterste lagen
- Lussen vallen steeds binnen één laag

Dit proces wordt **stratificatie** genoemd. Het is mogelijk elke FSM te stratificeren. Hier wordt aangeraden de voorbeelden in de cursus te bekijken.

3.3.9 Parallelle automaten

Wanneer verschillende groepen van events (binnen een klasse) volledig onafhankelijk van elkaar zijn, kunnen we gebruik maken van **parallellisme**. Dit verhindert de explosie van toestanden.

> **VOORBEELD**: *parallellisme*
>
> Beschouw een filmster en de events die diens professioneel en persoonlijk leven bepalen. Veronderstel dat kan worden aangeworven (en afgedankt) en kan trouwen (en scheiden). Samen voorgesteld leidt dit tot 2×2 toestanden:
>
> - single, zonder contract
> - single, met contract
> - gehuwd, zonder contract
> - gehuwd, met contract
>
> Het vereist geen betoog dat twee automaten dit veel duidelijker weergeven dan één alomvattende.

In UML zouden we gebruik kunnen maken van de **fork** en **join** constructies. Voor meer informatie zie de website van de OMG.

3.4 Interactie tussen bedrijfsobjecten

Bij gebeurtenissen zijn doorgaans verschillende objecten betrokken. De objecten werken dan samen om een event af te werken.

> **VOORBEELD**: *interactie tussen bedrijfsobjecten*
>
> In de SAI case is bij een inschrijving zowel een persoon bertrokken als object van de klasse inschrijving. Het toewijzen van de activiteit aan de klasse *Inschrijving* zou niet tonen dat er ook op andere niveaus dingen moeten gebeuren:
>
> - Nagaan of er kan worden ingeschreven voor een activiteit
> - Nagaan of de persoon lidgeld heeft betaald

In UML kunnen we deze samenwerking modelleren aan de hand van **collaboration diagrams**. Hierin wordt gemodelleerd welke boodschappen worden verstuurd en in welke volgorde.

3.4.1 Sequence Diagram

Het opstellen van een sequence diagram heeft als voordeel dat er vrij eenvoudig code uit kan worden gegenereerd. Het heeft anderzijds ook nadelen:

- **Sterk object georiënteerd**: Beide technieken zijn implementatiegericht, meerbepaald op dat van objectgeoriënteerd programmeren.
- **Gedetailleerde beschrijving**: Dit is niet altijd voordelig
- **Unidirectioneel**: De diagrammen tonen een visuele voorstelling van **message passing**. Op softwareniveau is dit steeds binair en unidirectioneel, iets dat niet altijd overeenstemt met de equivalent in de reële wereld.
- **Vanuit user infterface**: Meestal vertrekken de interacties vanuit de gebruikersinterface, iets wat niet is opgenomen in het bedrijfsmodel.

De concrete volgorde is niet van primair belang bij het opstellen van een bedrijfsmodel. De nadruk ligt op welke gebeurtenissen deelnemen en waarom. De focus ligt dan ook op het identificeren van de bedrijfsregels.

> **VOORBEELD**: *volgorde*
>
> Bij SAI is het niet belangrijk of er eerst wordt gecontroleerd of een persoon lid is en dan pas of er kan ingeschreven worden. Centraal staat wel dat beide dingen moeten gebeuren, maar niet in welke volgorde.

3.5 Object-Event tabel

Om niet te vroeg te focussen op de juiste interactiescenario's modelleren we de deelnemende objecten per geïdentificeerde bedrijfsgebeurtenis. Hiervoor gebruiken we een tabel waarbij elke cel zegt aan welke voorwaarden voldaan moet zijn.

		Klasse	*Klasse*	*Klasse*
Gebeurtenis	Voorwaarde			
	Actie			
Gebeurtenis	Voorwaarde			
	Actie			

3.5.1 De object-event tabel

We onderzoeken welke bedrijfsobjecten betrokken zijn en wat het effect is van de gebeurtenis. De tabel bestaat uit:

- Rij per gebeurtenistype
- Kolom per bedrijfsobjecttype

Een cel heeft volgende markeringen:

- **C**: Creator
- **M**: Modifier
- **E**: Ending-event

Wanneer een cel gemarkeerd is, wil dit zeggen dat het object voorzien wordt van een procedure. Deze procedure bepaalt wat het effect is van een gebeurtenis op de objecten van dit type. Wanneer de gebeurtenis optreedt, worden **alle** verbonden procedures uitgevoerd.

De tabel inventariseert alle mogelijke plaatse waar bedrijfsregels kunnen worden gespecificeerd. Wanneer blijkt dat bepaalde gemarkeerde cellen hier toch geen aanleiding tot geven, kunnen de nodige optimalisaties worden doorgevoerd bij de de implementatie.

Het belangrijkste voordeel van deze tabel is dat alle gebeurtenissen als zelfstandig kunnen worden beschouwd. Dit wil zeggen los van de bedrijfsobjecten.

3.5.2 Validatie

Tabellen of matrices zoals de OET zijn voornamelijk bekend als **CRUD**-matrices. Er werden een aantal regels ontwikkeling om een OET te valideren op interne consistentie. Dit gebeurt aan de hand van FSMs en het klassendiagram.

Bepaalde regels zijn intuitief:

- Per objecttype is er minstens één gebeurtenis die deze creëert
- Per objecttype is er minstens één die deze beëindigt

In volgende paragraaf bekijken we de regels voor het nagaan van consistentie. De overige regels vallen buiten de scope van deze cursus.

3.5.3 Inventariseren bedrijfsgebeurtenissen

Men kan op twee manieren bedrijfsgebeurtenissen inventariseren:

1. Kijken naar de gebeurtenissen die horen bij creatie, wijziging en beëindiging van objecten
2. Kijken naar de bedrijfsprocessen

> **VOORBEELD**: *bedrijfsprocessen als basis inventaris*
>
> Bij het verkoopsproces vinden we een hele lijst aan gebeurtenissen: creëren van een klant, van een order en van een orderregel. Ondertekenen van een order, het leveren en het factureren. Voor elke gebeurtenis kan men nagaan welke objecten erbij betrokken zijn.

Wanneer men gebeurtenissen identificeert, moeten de bedrijfsregels getoetst worden aan volgende criteria:

- **Atomiciteit**
  Ofwel vind de gebeurtenis volledig plaats, ofwel helemaal niet. Een gebeurtenis die kan onderbroken worden is geen goede bedrijfsgebeurtenis.

  > **VOORBEELD**: *niet atomaire gebeurtenis*
  >
  > Het registreren van een order is niet atomair. Het kan worden opgesplitst in het aanmaken van een nieuw order, het toevoegen van het product aan het order etc.

- **Relevantie**
  Gebeurtenissen die relevant zijn op software-niveau maar niet op modelleringsniveau komen niet in aanmerking.

  > **VOORBEELD**: *niet relevante gebeurtenis*
  >
  > Het afhalen van geld is een relevante gebeurtenis. Het invoer van de kaart, intikken van de code zijn dat niet.

Het voordeel van te werken met **bedrijfsgebeurtenissen** is dat het geïmplementeerd kan worden als een zelfstandige component. Die vormt dan een uniek aanspreekpunt om de bedrijfsobjecten te manipuleren. In plaats van berichten naar de objecten te sturen, kan een service nu de uitvoering van een gebeurtenis activeren en en zorgt het overeenkomstige gebeurtenisobject voor de afhandeling. Dit is vooral handig wanneer dezelfde functionaliteit onder verschillende vormen wordt aangeboden.

> **VOORBEELD**: *zelfde functionaliteit onder verschillende vormen*
>
> Er zijn verschillende manieren om geld af te halen. Dit kan via het loket, een ATM of via de proton kaart. In een event-loos systeem moet voor elke dienst message-passing worden ontwikkeld.

Hoofdstuk 4

Systeemanalyse

Systeemanalyse is het interdisciplinair proces tussen informatica en bedrijfskunde dat gericht is op het analyseren van systemen. In de inleiding hebben we in grote lijnen al aangehaald welke activiteiten dit inhoud.

4.1 Requirements versus specifications

Requirements Engineering is het proces waarbinnen all eisen voor het te ontwikkelen systeem worden ontdekt en gedocumenteerd. Het spreekt voor zich dat dit een essentieel onderdeel is van de systeemanalyse. We moeten onze gebruikers immers goed begrijpen.

We vormen binnen **RE** een **ontologie**. Dit is het resultaat van een poging tot het maken van een uitputtend, strikt schema over een bepaald domein. In dit geval beslaat dat domein de vereisten van de eindgebruiker. Een ontologie verschilt van andere structuren door de aanwezigheid van regels en (al dan niet hiërarchische) ordening.

4.1.1 Formele respresentatie

De betekenis van de gebruikte termen moet liggen in de reële wereld. De juistheid van formele asserties zijn hier immers van afhankelijk.

> **VOORBEELD**: *terminologie*
>
> $\forall s \forall c(enrolled(s, c) \Rightarrow student(s) \wedge course(s))$
> Kunnen enkel studenten ingeschreven zijn voor een cursus? Wat is een student? Wat is een cursus?

Designation Het is nodig om de primitieven een betekenis toe te kennen door ze te definiëren met een duidelijke verklaring. Deze verklaring noemen we een **designation**. Ze zijn essentieel en onderdeel van **elke** klasse.

Designations duiden op het verschil tussen definities en asserties.

- Een **assertie** kan waar of vals zijn

- Een **definitie** is (zoals de uitdrukking zegt) per definitie waar

> **VOORBEELD**: *definitie vs assertie*
>
> - $\forall s(student(s) \Leftrightarrow \exists c enrolled(s, c))$
> Veronderstel het bestaan van een *designation* voor student en *enrolled*
> - $student(s) \stackrel{\text{def}}{=} \exists c enrolled(s, c)$
> Vereist geen designation van *student*, maar wel van *enrolled*
>
> De designation voor *student* zou hier kunnen zijn: *een student is een persoon die ingeschreven is aan een universiteit en nog niet is afgestudeerd of teruggetrokken.*

Requirements hebben doorgaans het doel om een bepaalde doelstelling te bereiken. Aan de hand van designations beperken we het domein waarbinnen alternatieven kunnen worden gezocht. Bovendien zijn ze essentieel om de **identiteit** te bepalen van een object.

> **VOORBEELD**: *designations en identieit*
>
> Als ik een SMS-bericht krijg van mijn jongere zus, en ik stuur die door naar mijn oudere zus, hoeveel SMS-berichten zijn er dan ?

Het bepalen van designations is één van de taken van **bedrijfsmodelleren**.

4.1.2 Impelementation Bias

Requirements moeten beschrijven **wat** de gewenste machine doet, maar niet **hoe** die dat doet.

Willen alleen statements over de omgeving

Statements over machine vorm een bias

Twee beschrijvingen volstaan:

- **Indicative Mood**: beschrijving van hoe de omgeving is zonder de machine
- **Optative Mood**: beschrijving van hoe we willen dat de omgeving is met het systeem

Het is bovendien belangrijk dat de specificaties beschrijven wat kan worden geobserveerd aan de hand van de interface tussen omgeving en machine. Ze moeten dus **action-based** zijn en niet state-based. We kunnen acties verder opdelen:

- **Controlling Actions**
  - Environmental controlled
  - Machine controlled
- **Sharing Actions**
  - Shared: gedeeld tussen machine en omgeving
  - Unshared: enkel tot omgeving behoren

Het behoeft geen betoog dat de combinatie *unshared* en *machine controlled* niet mogelijk is.

VOORBEELD: *acties*

- Environment controlled shared action
  - Een student schrijft zich in voor een cursus aan de universiteit
  - Een boek in de bibliotheek wordt uitgeleend ($\neq$ meegenomen!)
- Environment controlled unshared action:
  - Een student volgt een hoorcollege van een cursus
  - Een boek wordt verplaatst in het boekenrek
  - Een boek wordt meegenomen zonder registratie van de uitlening
- Machine controlled shared action:
  - Een vraag voor het invullen van ISP wordt doorgestuurd.
  - Een reminder voor het binnenleveren van een boek wordt verzonden.

4.1.3 Domeinkennis

Domeinkennis slaat de brug tussen requirements enerzijds en specificaties anderzijds.

- **Requirements:** Optatieve eigenschappen die de wensen van de stakeholders vervullen wanneer aan de requirements is voldaan.
- **Specificaties:** Optative eigenschappen die implementeerbaar zijn

Het is mogelijk dat requirements geen specificaties zijn. Ze worden dan via de domeinkennis omgezet naar specificaties:

$$\boxed{S, K \vdash R}$$

VOORBEELD: *requirements omzetten naar specificaties*

- Studenten moeten hun ISP doorsturen voor 15/10 $\rightarrow$ **requirement**
- Het automatisch doorsturen van default ISP voor de nog niet doorgestuurde doorgestuurde ISPs ISPs op 15/10 $\rightarrow$ **specificatie**

4.1.4 Conclusies

- Alle terminologie in RE zou een weerspiegeling moeten zijn van de omgeving waarbinnen de te bouwen machine zich bevindt.
- Het is niet nuttig of wenselijk de te bouwen machine te beschrijven. We beschijven echter we de onmgeving:
  - Zoals die is **zonder** de machine

  - Zoals die zou moeten zijn **met** de machine

- Er moet onderscheid gemaakt worden tussen acties die
  - **Gecontroleerd** worden door de omgeving
  - **Gedeeld** worden door de machine
- De primaire rol van domeinkennis in RE is het ondersteunen van de **verfijning** van vereisten tot specificaties

4.1.5 Tekortkomingen

Bovenstaande ontologie, ontwikkeld door Zave en Jackson, hebben enkele tekortkomingen:

- Geen ruimte voor **gedeeltelijke** invulling van requirements
- Geen specificaties definieerbaar
- Optimaliteit en non-functionele requirements komen niet aan bod

Daarom werd de **CORE ontologie** opgesteld. Deze vertrekt vanuit de gedachte dat elke requirement in essentie een **speech act** (taalhandeling) is van de stakeholder aan de ingenieur.

Deze speech acts kunnen van het volgende type zijn:

- **Belief (B)**: Leiden tot domeinaannames
- **Desire (D)**: Leiden tot requirements
- **Intention (I)**: Leiden tot specifications
- **Attitude (A)**: Komt niet aan bod (!)

Speech acts kunnen volgende soorten **illocutary forces** hebben:

- **Assertives (B)**: Assert a proposition that the speaker believes true
- **Directives (D)**: Convey a proposition that the speaker wants to see become true
- **Commissives (I)**: States what the speaker intends to make a proposition
- **Expressives (A)**: Convey an emotion/attitude about speaker or hearer
- **Declarations (B)**: By the very act of being stated make a proposition true
- **Representative declarations (B)**: Recognize the truth of a proposition that has been made true by a declaration true

De combinatie van het type speech act met de illocutionary force ervan geeft ons een basis om de content van een speech act te indentificeren voor RE.

- **Beliefs**

  - Assertives

    > **VOORBEELD**: *assertives*
    >
    > Studenten moeten hun studentenkaart tonen vooraleer ze hun examenattest krijgen.

  - Declarations

    > **VOORBEELD**: *declarations*
    >
    > Wie 10 of meer behaalt is geslaagd.

  - Representative declarations

    > **VOORBEELD**: *representative declarations*
    >
    > De examencommissie heeft bepaald dat student X geslaagd is met onderscheiding

  → Catalogiseren we als **assumpties**.

- **Desires**
  - Indien uitdrukking als **directive**
  - Maar **geen** uitspraken over kwaliteiten

    > **VOORBEELD**: *desires*
    >
    > Nadat de bevestiging van de betaling is binnengekomen, moet de boeking bevestigd worden.

- **Desires**
  - Indien uitdrukking als **directive**
  - Maar **wel** uitspraken over kwaliteiten

    > **VOORBEELD**: *desires*
    >
    > Het moet mogelijk zijn een vlucht te boeken via het doorlopen van minder dan 5 schermen.

  → Catalogiseren we als **quality constraints**.

- **Desires**
  - **Directive** waarbij kwaliteiten en constraints op kwaliteiten
  - Maar definitie is subjectief of niet afgelijnd.

    > **VOORBEELD**: *desires*
    >
    > Het moet mogelijk zijn een vlucht te boeken via een gebruiksvriendelijk systeem.

→ Catalogiseren we als **soft goals**.

- **Intents**
  - Uitgedrukt als commisive
  - Beschrijft de manier waarop (= specificatie)

  → Catalogiseren we als **plan**.

- **Attitudes**
  - Uitgedrukt als expressives
  - Waarbij een voorkeur wordt uitgedrukt mbt de voorgaande elementen

  → Catalogiseren we als **Optionaliteiten en Preferenties**.

4.1.6 Herdefinitie van het RE probleem

De oorspronkelijke definitie was

$$\boxed{S, K \vdash R}$$

De requirements moesten verfijnd worden tot implementeerbare specificaties, zodat de combinatie van assumpties (K) en specificaties (S) voldoen aan de requirements (R).

De definitie wordt nu:

$$\boxed{K^*, P^* \vdash G^*, Q^*, A^>}$$

We kiezen tussen de alternatieve assumptie (K) en de alternatieve specificatie (P) een combinatie K, P zodat:

- Preferenties $A^>$ maximaliseren
- Dat voldaan is aan de verplichte goals (G) en quality constraints (Q), en aan zoveel mogelijk optionele Goals.

4.2 Plaats van bedrijfsmodelleren in systeemanalyse

Het vinden van de juiste klassen en verbanden is een van de moeilijkste elementen van modelleren. Verschillende technieken maken het ons gemakkelijker.

4.2.1 Lagen

Klassen worden georganiseerd in **lagen**. Binnen dezelfde laag kan er gebruik gemaakt worden van diensten van onderliggende lagen. Wijzigingen propageren zich dus enkel opwaarts. De onderste lagen zijn dan ook de meest stabiele.

Typische lagen:

- **User Interface**

  > **VOORBEELD**: *gebruikersinterface*
  >
  > Knoppen, velden

- **Applicatielogica**

  > **VOORBEELD**: *applicatielogica*
  >
  > Berekenen van de datum

- **Bedrijfsobjecten**
  - Logica

    > **VOORBEELD**: *logica*
    >
    > Regels in verband met het verlengen van de uitleentermijn

  - Data

    > **VOORBEELD**: *data*
    >
    > Titel van een boek

We zien dat de lagen de volgorde van variabiliteit volgen.

4.2.2 Soorten specificaties: Zachman

Voor een informatiesysteem kan men verschillende plannen maken. Het **Zachman framework** is een classificatieschema voor complexe informatiesystemen.

- Laat toe onderdelen te bestuderen zonder context uit het oog te verliezen
- Biedt overzicht van complex systeem
- Combineert twee perspectieven:
  - Organisatorisch perspectief
  - Informatiesysteem perspectief

  Laat toe de bedrijfs- en informatiestrategie op elkaar af te stemmen

Voor de juiste indeling, zie schema in de cursus op pagina 51.

Rij 2: Het enterprise-model perfectief

Dit perspectief stemt overeen met dat van de **eigenaar** of **opdrachtgever**. Dit model geeft een overzicht van:

- Werking van de organisatie
- Businessobjecten
- Geldende regelgeving

Omdat dit deel zich bezig houdt met het beschrijven van de *business*, wordt dit ook vaak **business modelling** genoemd.

> VOORBEELD: *kul*
>
> Het onderwijs- en examenreglement vormt een onvolledige en ongestructureerde beschrijving van de *business* van de KUL.

Rij 3: Informatiesysteem-model perspectief

Dit perspectief stemt overeen met dat van de **ontwerper**. Deze heeft kennis van zowel de **bedrijfsaspecten** als de **technische aspecten**.

Hier wordt de gewenste functionaliteit bepaalt, rekening houdend met de beperkingen van de omgeving waarin het systeem operationeel is.

4.2.3 Doel van bedrijfsmodellering

Tot dusver bekeken we het perspectief van de opdrachtgever: het lastenboek. Dit omvat de bovenste drie lagen van het framework.

- Bepalen scope
- Vastleggen bedrijfsregels
- Bepalen van de informatiesysteemondersteuning

De oefeningen besloegen slechts het enterprise model (rij 2).

4.2.4 Het bedrijfsmodel

Het bedrijfsmodel...

- **Als kernmodel**
  Het bedrijfsmodel definieert de bedrijfsobjecten en legt tevens de bedrijfslogica vast. Dit vormt de kern van het informatiesysteem.
- **Als middel tot alignering onder de gebruikers**
  Het is een gestructureerd en meer formele voorstelling van alle bedrijfsconcepten, -regels en -processen. Het vormt een gemeenschappelijke taal onder de betrokkenen.

- **Als middel tot een robuust systeem**
  Het bedrijfsmodel beschrijft naast het probleemdomein ook de kern van de oplossing. Een beter begrip leidt tot betere software. (zie fabel bootverhuur)

4.2.5 Het enterprise model

Zodra duidelijk is welke specificaties betrekking hebben met het bedrijfsmodel, kunnen we de klassendefinities opstellen.

Dit proces is normaal al meermaals aan bod is gekomen. Wie het toch nog wil herhalen kan het nalezen op pagina 56 van de cursus.

Idealiter beschrijft het bedrijfsmodel ook de dynamische aspecten. Hiervoor zijn twee invalshoeken:

- Gedragsaspecten per business beschrijven door middel van een **FSM**.
- Opstellen van bedrijfsmodellen: dit is een keten van activiteiten die wordt ontplooid met het oog op het realiseren van de bedrijfsdoelstellingen.

Beide processen zijn complementair aan elkaar.

4.3 Patterns

Een oplossing die we mits aanpassing kunnen toepassing in verschillende context wordt een **pattern** genoemd.

Het omzetten van een verband naar een tussenliggende klasse is een bewerking die herhaaldelijk wordt toegepast. Formeel zijn er verschillende definities, waarvan de meer concrete luidt:

> A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts.

Patterns zijn interessant omdat ze domeinkennis bevatten en de grondgedachte documenteren:

- Hergebruiken wijsheid en ervaring
- Bieden naast oplossing ook context
- Ontleden de oplossing

4.3.1 Soorten patterns

Patterns kunnen worden opgedeeld volgens de fasen van de systeemontwikkelign:

- **Analyse:** Analyse-patterns
  Pattern waarvan de vorm beschreven wordt met behulp van voorwaarden en concepten uit het applicatiedomein.
- **Design:** Design-patterns
  Pattern waarvan de vorm beschreven wordt software design constructies zoals objecten, klassen, overerving, ...

- **Implementatie:** Code-patterns
  Patters waarvan de vorm beschreven wordt met programmeertaalinstructies.

Patterns kunnen toegepast worden op verschillende domeinen.

4.3.2 Elementen van een pattern

Een pattern kan op verschillende manieren worden beschreven. Twee veelvoorkomende vormen zijn:

- **Alexandrian Form**: doorlopende tekst
- **Complien Form**: gestructureerd

De **Complien Form** bestaat uit volgende elementen:

- **Naam** referentie naar het pattern voor het gemakkelijk terugvinden.
- **Probleem** Uiteenzetting van het probleem
- **Krachten** Welke krachten en beperkingen spelen
- **Oplossing** Statische relaties en dynamische regels beschrijven hoe de gewenste Uitkomst kan worden gerealiseerd
- **Voorbeelden** Ter illustratie
- **Resulteren contex** Configuratie van het systeem nadat het pattern is toegepast
- **Rationale** Verklaring van de stappen of regels en verklaring waarom het werkt.
- **Gerelateerde patterns** Andere patterns delen vaak dezelfde krachten.
- **Gekende gebruiken** Voorkomen van het pattern binnen bestaande systemen

4.3.3 Anti-patterns

Een pattern beschrijft een *best practice*, maar met een **anti-pattern** kan het tegengestelde bieden. Er zijn twee soorten:

- Slechte oplossing beschrijven en de problemen die daarmee verbonden zijn
- Slechte situatie voorkomen en hoe vandaar gewerkt kan worden naar een goede oplossing

Wanneer gerelateerde patterns aan elkaar worden gewoven vormt men een **pattern language**. Deze vormen samen de oplossing voor een complex probleem.

4.3.4 Pattern mining en PLoP

Het proces van zoeken naar patterns om ze te documenteren wordt **pattern-mining** genoemd.

PLoP of **Pattern Languages of Programming**-conferenties zijn een mogelijkheid voor de auteur om hun patterns te laten beoordelen door collega's.

4.4 Kwaliteitscontrole

Door het gebruik van meerdere technieken voor het beschrijven van één probleem, komen sommige elementen meermaals voor.

> **VOORBEELD**:
>
> De klassen die aan bod komen in de FSM zijn ook gedefinieerd in het klassendiagram. De events die overgangen bepalen zijn ook onderdeel van de OET.

Het spreekt voor zich dat we moeten controleren of deze wel consistent zijn. Dit gebeurt aan de hand van kwaliteitscontrole.

4.4.1 Randvoorwaarden

De taal het domein en het publiek vormen de **randvoorwaarden** die de kwaliteit van het model beïnvloeden.

- **Relatie taal-domein**: Is de taal geschikt voor het neerschrijven van het model?
- **Relatie taal-publiek**: Is de taal geschikt om te communiceren met het publiek?
- **Relatie publiek-domein**: Het juiste publiek moet gekozen worden om daarop een model te baseren.

4.4.2 Types van kwaliteitscontrole

Gegeven een domein, taal en publiek. We kunnen dan volgende types van kwaliteitscontrole onderscheiden:

- **Syntax:** voldoet het madel aan de vormeisten opgelegd door de taal?
- **Semantiek:** klopt de betekenis van het model?
  - **Validiteit:** Klopt het?
  - **Volledigheid:** Is alles opgenomen?
- **Pragmatiek:** kan het model begrepen worden door het publiek?

4.4.3 Middelen tot kwaliteitscontrole

De meeste modelleringstools bieden een aantal hulpmiddelen voor de controle op kwaliteit:

Syntax

Meestal laat het gebruikte programma niet toe syntaxfouten te maken.

Semantiek

Voor semantische controle is het belangrijk dat de betekenis van de concepten in de modelleringstaal eenduidig is gemodelleerd. Het is van groot belang dat de uitspraken elkaar niet tegenspreken.

- **Externe validiteit:** Controle op consistentie met het domein
- **Interne validiteit:** Controle op consistentie met de reeds expliciete of afgeleide uitspraken

Er bestaan drie manieren waarop aan consistentiecontrole wordt gedaan:

- Consistentie via **analyse**: Controleren aan de hand van een algoritme.
- Consistentie via **monitoring**: Bij het toevoegen direct controleren of er geen tegenspraak is ontstaan.
- Consistentie via **constructie** Bij het toeveogen direct afgeleide uitspraken vormen en toevoegen.

Dergelijke controle is enkel mogelijk als er voldoende duidelijke regels bestaan.

Pragmatiek

Voor de controle op pragmatiek bieden modelleringsprogramma's eveneens verschillende hulpmiddelen.

Index