design your future

Systeemanalyse

studiegebied
handelswetenschappen en bedrijfskunde

bachelor

toegepaste informatica

Inhoud

Hoofdstuk 1: Inleiding....3
1. Inhoud van de cursus....3
2. Wat is systeemanalyse?....5
3. Belang van systeemanalyse....13
4. De fasen in softwareontwikkeling....17
5. Het watervalmodel....18
6. Een iteratief model: Scrum....21
7. Objectgeoriënteerd werken: werken met objecten en klassen....21
8. OOA en OOP....30
9. UML (Unified Modeling Language)....34
10. Opdrachten en oefeningen....36
Hoofdstuk 2: Casussen....38
1. De likkebaard....38
2. De eenvoudige bibliotheek....42
3. Beheer van afwezigheden....43
4. De computeroefenzaal....45
5. Galerie de Aardbei....45
6. De eenvoudige bank....46
7. Donation bank....47
8. Medicatiebeheer....47
Hoofdstuk 3: Vooronderzoek....56
1. Doel van het vooronderzoek....56
2. Analyse van de huidige situatie....57
3. Beschrijving van de behoeften (gewenste situatie)....73
4. Korte beschrijving van de oplossingsalternatieven....75
5. Selectie van de beste oplossing....75
6. Evaluatie van de oplossing....78
7. Verificatie van het vooronderzoek....83
8. Technieken voor het vooronderzoek....83
9. Het gebruik van activeitendiagrammen....84
10. Samenvatting van het vooronderzoek....93
Hoofdstuk 4: Analyse- of specificatiefase....96
1. Inleiding....96
2. Conceptuele lagen....96
3. Statische systeemstructuur (domeinmodel)....104
4. Vaststellen van de vereisten....131
5. Modelleren met use-cases....141
6. Dynamische systeemstructuur (business events)....162

7. Verificatie van de analysefase....179
8. Rapid prototyping....181
9. Samenvatting van de analysefase....185
Hoofdstuk 5: Ontwerpfase....189
1. Inleiding....189
2. Ontwerpen van de globale architectuur....190
3. Ontwerpen van de opslagstructuur....207
4. Ontwerpen van de mens-machine-interactie....231
5. Detailontwerp....233
6. Maatregelen voor security en privacy....242
7. Ontwerpen van manuele procedures....245
8. Handleiding, on-line help en opleiding....246
9. Technische infrastructuur....247
10. Opstellingsdiagrammen (deployment diagrams)....248
11. Opstellen programmeer en testplan....251
12. Testen van het ontwerp....251
13. Samenvatting ontwerpfase....252
Hoofdstuk 6: Realisatiefase....256
1. Inleiding....256
2. De iteratieve aanpak....256
Hoofdstuk 7: Integratiefase....265
1. Inleiding....265
2. Wat is testen?....265
3. Testniveaus volgens het V-model....267
4. Aanpak van de testactiviteit....273
5. Vuistregels voor testen....276
6. Invoering....276
7. Oefeningen en opdrachten....277
Hoofdstuk 8: Onderhoudsfase....279
1. Types onderhoud....279
2. Het onderhoudsproces....280
3. Problemen bij onderhoud....281
4. Reverse engineering....282

Hoofdstuk 1: Inleiding

In dit hoofdstuk worden de basisbegrippen die nodig zijn om de rest van de cursus goed te begrijpen uitgelegd. Er wordt uitgelegd wat systeemanalyse en software engineering precies is en wat het belang ervan is. De 'Software Development Life-Cycle' wordt besproken, en een aantal visies daarop. Ten slotte worden ook de basisconcepten van objectoriëntatie verduidelijkt en het begrip UML wordt verklaard. Kort samengevat moet elke student op het einde van het hoofdstuk in staat zijn op de volgende vragen te antwoorden:

- *Wat zijn analyse en ontwerp?*
- *Wat is het watervalmodel?*
- *Wat is iteratief ontwikkelen?*
- *Wat is de rol van een systeemanalist en wat zijn de vaardigheden waarover hij dient te beschikken?*
- *Welk voordeel geeft een modelgerichte aanpak tegenover een puur tekstuele aanpak?*
- *Waarom is systeemanalyse belangrijk?*
- *Wat bedoelen we met objectgeoriënteerde systeemanalyse en wat zijn nu objecten?*
- *Wat is UML precies en welke rol speelt UML in deze cursus?*

1. Inhoud van de cursus

Zoals verder in dit hoofdstuk wordt verklaard, kunnen systemen en software niet ontwikkeld worden zonder een goede analyse en een doordacht ontwerp.

Als je een huis wil bouwen, dan ga je niet onmiddellijk met bakstenen en mortel aan de slag. De kans dat je bouwavontuur slecht afloopt is dan veel te groot. Eerst ga je nadenken wat je verwacht van het huis: hoeveel slaapkamers, hoeveel garages, … . Er wordt een onderzoek gedaan van de bodem en de bouwreglementen. Je zoekt uit hoeveel je kunt lenen, en wat het huis mag kosten. Je gaat na of het niet gemakkelijker zou zijn om een bestaand huis te kopen. Als je beslist om zelf te bouwen, dan zoek je een architect. Die architect zal een plan tekenen, en dat samen met jou verfijnen. Ook de aannemer begint niet zonder overleg te bouwen. Hij heeft een goede kennis van de technieken en de materialen die nodig zijn om een huis te bouwen en hij gaat planmatig te werk.

Hetzelfde geldt voor het ontwikkelen van software: in de *analysefase* worden de behoeften van de opdrachtgever geanalyseerd en wordt de huidige situatie precies en duidelijk in kaart gebracht. Wanneer dit gebeurd is, zal een goed gefundeerd *ontwerp* uiteindelijk de basis vormen voor de uiteindelijke realisatie. Software van hoge kwaliteit

kan niet ontwikkeld worden zonder analyse en ontwerp.

Het is dan ook niet meer dan logisch dat er in het curriculum van een opleiding toegepaste informatica een cursus analyse en ontwerp voorkomt. In deze cursus worden een aantal technieken voor analyse en ontwerp aangeleerd. Tegelijk krijgt de student een inzicht in het software-ontwikkelingsproces en in de problemen die daarin kunnen optreden. De cursus is dus ruimer van opzet dan enkel technieken en methoden van systeemanalyse. De belangrijkste aspecten van het gebied van de *Software Engineering* komen in deze cursus ook aan bod.

Deze cursus is een inleiding in het domein. Na deze cursus kan niemand beweren dat hij een uitstekend systeemanalist is. Een goed systeemanalist word je door ervaring op te bouwen. De voornaamste bedoeling is voldoende achtergrondkennis mee te geven en een 'feeling' te kweken zodat je in staat bent met een beter begrip het werk van analisten te beoordelen en te gebruiken. Ook eventueel om, na het verwerven van voldoende praktijkervaring, dan toch de boeiende en creatieve functie van systeemanalist te kunnen uitoefenen. De cursus heeft ook als doel een hulpmiddel te zijn voor het analysewerk dat je tijdens projectwerk en stage moet uitvoeren. Daar krijg je de kans om alles in de praktijk te brengen.

Je zult misschien het gevoel krijgen dat wat in deze cursus aan bod komt overdreven is, dat je sneller resultaten zou boeken als je onmiddellijk begint te programmeren. Het is natuurlijk zo dat je niet altijd alle technieken uit deze cursus nodig hebt. Als je een kippenhok wilt bouwen, dan ga je niet met een architect en een aannemer aan de slag. Dan maak je een schets, ga je materiaal inkopen, en begin je te bouwen. De kans is groot dat het resultaat een goed kippenhok is. Bij het maken van sofware is het ook zo: kleine systemen kun je maken met een beperkte analyse, maar hoe groter het systeem dat je wilt bouwen, hoe belangrijker het is om wel een goede analyse te maken.

In deze cursus wordt vooral gekozen voor objectgeoriënteerde analysetechnieken. De voornaamste inspiratie is de Merode-methode die aan de faculteit economische wetenschappen van de Universiteit Leuven ontwikkeld werd[1]. Wij proberen de technieken en methoden ook altijd te situeren binnen het geheel van het ontwikkelings-proces. De cursus is opgebouwd als het watervalmodel, dat is een onwikkelproces waarbij de fasen elkaar strikt opvolgen. Dit is een overzichtelijke structuur om de verschillende begrippen te introduceren. Verder in dit hoofdstuk wordt uitgelegd dat dit watervalmodel in de praktijk weinig gebruikt wordt. Ontwikkelen gebeurt meestal met een iteratief proces. Maar ook in een iteratief proces heb je analyse en ontwerptechnieken nodig.

---

[1] Snoek Monique e.a.. *Object-Oriented Enterprise Modelling with Merode*. Leuven University Press, Leuven, 1999. 227 blz.
http://merode.econ.kuleuven.ac.be/

De cursus bevat veel theorie, maar probeert die op een toepasbare manier aan te bieden. De beste manier om het vak onder de knie te krijgen is veel oefenen.

2. Wat is systeemanalyse?

2.1. Informatiesysteem

In Wikipedia[2] vinden we de volgende definitie van een informatiesysteem:

"*Een* ***informatiesysteem*** *is een systeem waarmee informatie over objecten of personen beheerd – verzameld, bewerkt, geanalyseerd, geïntegreerd en gepresenteerd – kan worden. Tot een informatiesysteem in ruime zin worden naast de data en de technieken en faciliteiten om data te ordenen en te interpreteren vaak ook de ermee verbonden organisatie, personen en procedures gerekend.*
Veel informatiesystemen zijn geautomatiseerd en omvatten dus hardware en software."

Informatici staan in voor de ontwikkeling, de implementatie en het onderhoud van geautomatiseerde informatiesystemen.

2.2. Systeemanalyse

Om het geautomatiseerde informatiesysteem met succes te ontwikkelen moeten een hele reeks belangrijke beslissingen genomen worden. Er moet daarvoor een antwoord gevonden worden op een hele reeks vragen, zoals:

> *Wat zijn precies de doelstellingen van het informatiesysteem? Welke informatie moet het systeem kunnen verschaffen? Hoe moet de gebruiker met het systeem omgaan? Hoeveel mag het systeem kosten? Hoe zit het met de beveiliging?*

In functie daarvan moet de geschikte hardware gekozen worden en moet kwalitatief hoogstaande software ontwikkeld worden. Bij al die beslissingen staat één zaak centraal: de informatiebehoefte van de gebruiker. Het doel van een informatiesysteem is aan de behoefte van de gebruiker te voldoen.

Systeemanalyse is een verzamelnaam voor een reeks technieken en methodes om

- de informatiebehoefte van de gebruiker ondubbelzinnig in kaart te brengen (analyse) en
- om het informatiesysteem op hoog (logisch) niveau te gaan ontwerpen (*ontwerp*).

Zoals je ziet maken we een onderscheid tussen *analyse* en *ontwerp*.

De **analyse** zelf bestaat uit twee aspecten:

[2] http://nl.wikipedia.org/wiki/Informatiesysteem, geraadpleegd 2019-07-18

- De analyse van het probleem (*domeinanalyse*, in het Engels domain analysis).
- De analyse van de vereisten (*vereistenanalyse*, in het Engels requirements analysis).

Domeinanalyse is het uiteenrafelen en structureren van het probleem, zodat je het goed genoeg begrijpt om er een informatiesysteem voor te ontwikkelen.

Vooraleer we een informatiesysteem kunnen ontwerpen, moeten we begrijpen waar het systeem voor dient. Je kunt geen systeem voor loonberekening ontwikkelen als je niets kent van loonadministratie. Je kunt geen systeem voor voorraadbeheer ontwikkelen als je niet weet wat er in voorraad gehouden wordt. We moeten dus weten in welke context het informatiesysteem moet functioneren, met wat voor dingen de gebruiker werkt, wat de kenmerken zijn van die dingen, hoe ze zich tot elkaar verhouden en welke wijzigingen ze kunnen ondergaan.

We noemen dit het **probleemdomein**, of korter het **domein**. Het domein is het 'onderwerp' van het systeem dat we willen bouwen. Om een goed systeem te kunnen bouwen, hebben we kennis over het domein nodig. Dat is het doel van domeinanalyse: kennis over het domein opdoen en vastleggen.

Naast de structuur van het probleemdomein, moeten we ook beschrijven welke verwachtingen de opdrachtgevers en eindgebruikers van het systeem hebben. Dit is de **vereistenanalyse**. In de vereistenanalyse beschrijven we welke functies er in de software voorzien moeten worden en welke niet. We willen ook weten aan welke andere eisen de software moet voldoen, bijvoorbeeld op welke platformen het programma moet kunnen draaien. De opsomming en beschrijving van de gevraagde functies en de andere eisen noemen we de vereistenanalyse.

De beschrijving van het probleemdomein (domeinanalyse) en de vereistenanalyse vormen samen de analyse. We kunnen stellen dat de analyse beschrijft **WAT** ontwikkeld moet worden.

Met **ontwerp** bedoelen we een conceptuele voorstelling van de oplossing. Zonder al alle details vast te leggen wordt hier beschreven hoe de oplossing logisch gestructureerd is. Ontwerp kan onder andere zijn: het ontwerp van het globale applicatiearchitectuur, het databaseontwerp, het ontwerp van softwareklassen en -objecten, het ontwerp van de veiligheidsmaatregelen, enzovoort. We zeggen dat het ontwerp beschrijft **HOE** de oplossing er zal uitzien. Het ontwerp moet uiteindelijk in een concrete implementatie omgezet kunnen worden. Het ontwerp is natuurlijk gebaseerd op de resultaten van de analyse.

Op basis van de analyse en het ontwerp kunnen ontwikkelaars de modules gaan programmeren, kunnen netwerkspecialisten, systeembeheerders, database administrators

overgaan tot de aankoop van de geschikte hardware en standaardsoftware, kan een testplan opgemaakt worden, kunnen afdelingsmanagers de nodige organisatorische wijzigingen plannen en kunnen projectmanagers beginnen met de organisatie van de gebruikersopleiding.

2.3. Analist als brugfunctie

Bij de ontwikkeling van een informatiesysteem zijn er altijd minstens twee partijen betrokken: de o*pdrachtgever/ klant/ eindgebruikers/eindgebruiker* aan de ene kant en het *informaticateam* aan de andere kant.

Tussen die twee partijen zijn er communicatiestoornissen mogelijk. Ze kunnen vooroordelen hebben over elkaar. We horen dan uitspraken van eindgebruikers over informatici als:

> *Die informatici luisteren niet. Je vraagt iets en ze komen uiteindelijk met iets anders aandraven. Wat doen zij eigenlijk, wat op de computer tokkelen en altijd de laatste nieuwe technische snufjes aankopen? Altijd zijn ze te laat met hun software en het kost veel te veel. Het doet dan nog nauwelijks wat we willen en is veel te ingewikkeld.*

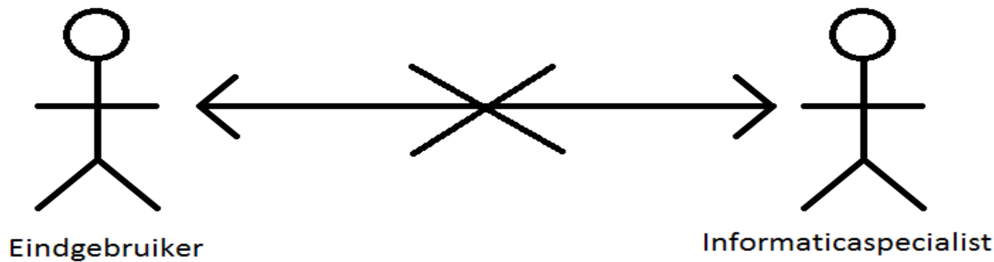
Informatici ergeren zich dan weer aan eindgebruikers:

> *Gebruikers weten niet wat ze willen. Ze veranderen voortdurend van mening. Ze stellen ook onmogelijke eisen, ze verwachten dat wij complexe vragen in enkele dagen oplossen. Zij begrijpen helemaal niet waar informatica over gaat, gebruiken de software verkeerd en leggen niet duidelijk uit wat ze willen.*

Wat is nu de oorzaak van die communicatiekloof?

De gebruiker (klant/ opdrachtgever) heeft een probleem dat door de informatica opgelost kan worden. Hij kent zijn bedrijf door en door en weet ongeveer wat hij wil, maar zelf heeft hij onvoldoende technische kennis om het informatiesysteem te bouwen.

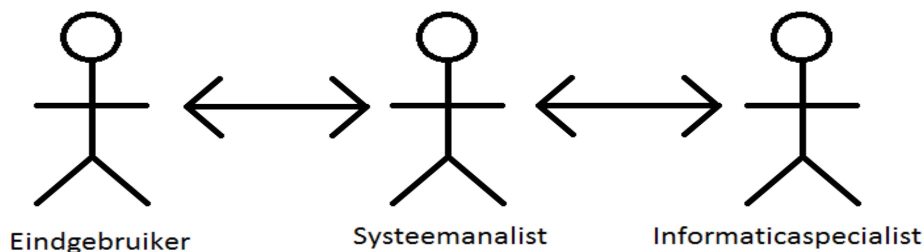
Informatici daarentegen hebben de technische middelen en knowhow om uitstekende informatiesystemen te bouwen. Alleen, zij weten niet wat de opdrachtgever precies wil en hoe zijn bedrijf precies in elkaar zit. Zij hebben met andere woorden geen duidelijk inzicht in de behoeften van de gebruiker. De opdrachtgever kent dus het probleem maar kan het niet oplossen. De programmeur zou het probleem kunnen oplossen maar hij begrijpt het probleem niet. Dat zorgt voor een communicatiekloof.



Deze communicatiekloof is niet zo gemakkelijk om te overbruggen. De informaticus heeft allerlei ideeën over hoe informatica er kan en moet uitzien en hoeveel ontwikkelingstijd dat kost, maar hij slaagt er niet in om dit aan leken uit te leggen. Hij is ook zo gefocust op informatica-oplossingen dat hij zich moeilijk kan inleven in de problematiek van de gebruiker en de neiging heeft te vergeten dat informatica geen doel op zich is, maar alleen als doel heeft de bedrijfsdoelstellingen te helpen realiseren.

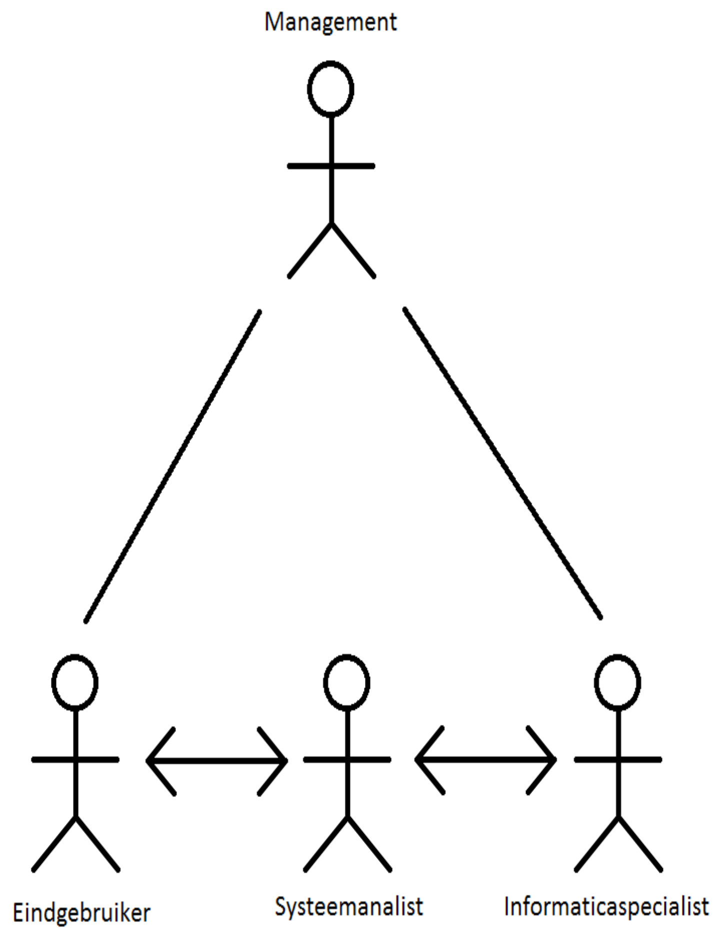
De opdrachtgever/gebruiker daarentegen heeft slechts een vaag idee van wat technisch mogelijk is en wat een informatiesysteem kan kosten. Hij kent zijn bedrijf door en door, maar vergeet dat anderen niet vertrouwd zijn met de bedrijfsterminologie en de bedrijfsprocedures. Soms weet hij niet helemaal precies wat nu juist zijn informatiebehoefte is. Hij is ook ongeduldig en kan maar niet begrijpen waarom de informaticus niet lijkt te snappen wat hij precies moet ontwikkelen en waarom dat zo lang moet duren.

Hier ligt een belangrijke taak voor de systeemanalist. Hij moet de behoeften van de opdrachtgever en de eindgebruikers proberen te achterhalen en die vertalen naar exacte technische specificaties waarmee de programmeur aan de slag kan. Hij waakt er ook over dat de opdrachtgever en de eindgebruikers in het hele ontwikkelingsproces centraal blijven staan.



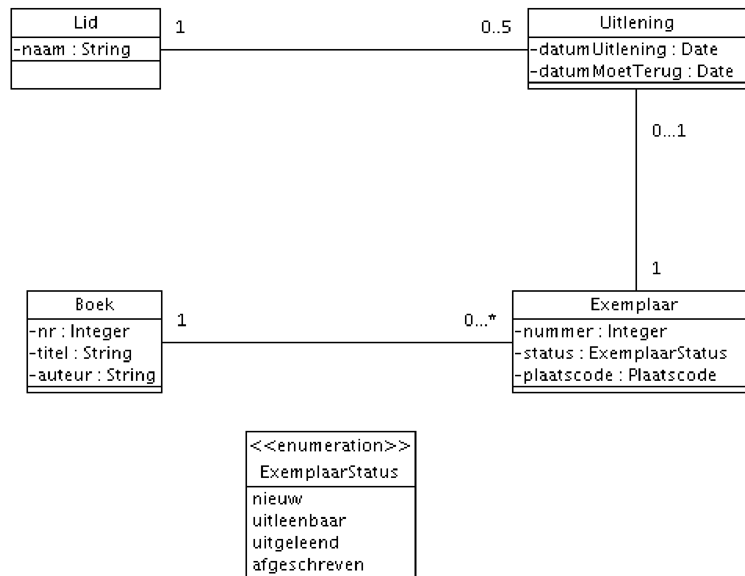
Een derde partij is het management of de financier van een project. Het management moet het ontwikkelingsproces in de hand houden. Programmeren is duur. Het management ziet het systeem als een investering en kijkt erop toe dat de te investeren middelen op een verantwoorde wijze worden uitgegeven. De systeemanalist moet ervoor zorgen dat hij het management voorziet van een middel om deze controle

mogelijk te maken.



2.4. Analysemodellen

Wanneer een analist een probleem aangeleverd krijgt, doet hij via interviews, lectuur van bedrijfsdocumenten, observaties, ... grondig onderzoek naar de behoeften van de klant en de structuur van het probleemdomein. Het resultaat van dit onderzoekswerk is een verzameling teksten: interviewrapporten, samenvattingen, ...



Voorbeeld van een klassendiagram

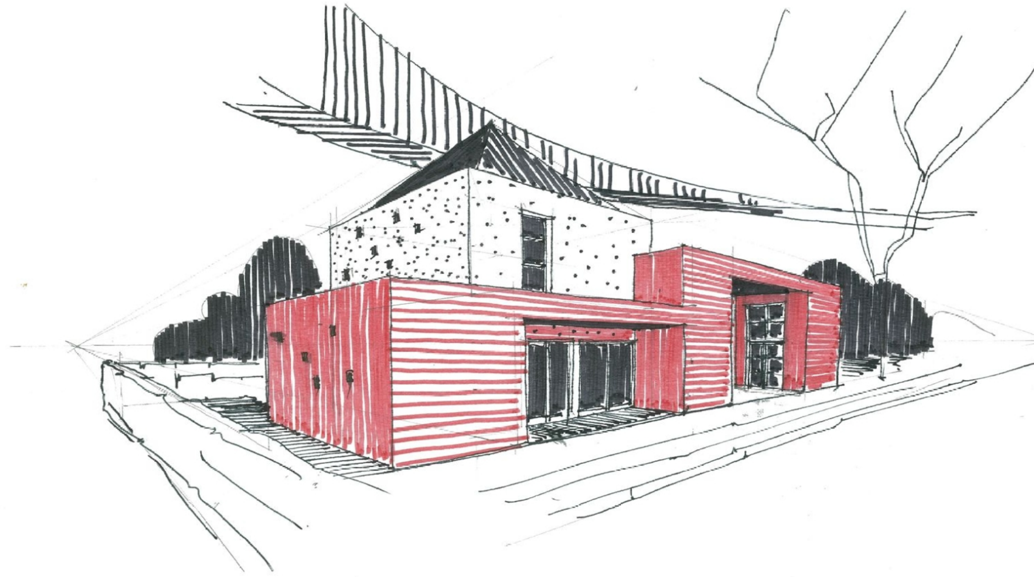
Met teksten alleen kan de analist niet verder. Teksten hebben beperkingen.

1. *Teksten zijn niet overzichtelijk*. Teksten bevatten een enorme hoeveelheid informatie die niet in één oogopslag te overzien is. Het voortdurend raadplegen van de teksten om de analyse te controleren is tijdrovend en moeilijk.
2. *Teksten zijn vaak dubbelzinnig*. De precieze betekenis van termen is soms slecht afgebakend. Bijvoorbeeld: wat wordt bedoeld met 'klant'? Is dat iemand die al een bestelling heeft geplaatst, of kan dat ook een potentiële klant zijn? Vaak worden ook synoniemen gebruikt: cliënt, klant, customer worden vaak in dezelfde tekst gebruikt. Heeft de tekst het dan telkens over hetzelfde?
3. *Teksten zijn vaak onvolledig*. Door de onoverzichtelijkheid van teksten is het ook moeilijk om te achterhalen of de tekst wel alle noodzakelijke informatie bevat. Is de analyse wel volledig?

Het is de taak van een analist om de informatie een duidelijke structuur te geven. Goede analyses zijn *model-driven.* Dit betekent dat de analist waar mogelijk zijn ideeën en bevindingen weergeeft in visuele modellen. Modellen geven een visueel beeld van een probleemgebied of een oplossingsvoorstel. Een model is een vereenvoudiging van de realiteit van uit een bepaald perspectief (functionaliteit, datastructuur, fysieke architectuur, ...).

Nemen we nog eens het voorbeeld van het bouwen van een huis. Je kunt een huis beschrijven met tekst (Een vrijstaande woning voor één gezin met twee kinderen, met twee bouwlagen, gebouwd op 2 meter van de rooilijn, met een breedte van …). Een

huis in tekst beschrijven zodat iemand zich kan inbeelden hoe dat huis er uit zal zien is heel moeilijk. Maar visuele modellen van het huis, dus schetsen en plannen, zijn veel overzichtelijker en veel duidelijker.



(bron figuur: Axi Architecten)

Modellen hebben aantal duidelijke voordelen in vergelijking met een beschrijving die enkel uit tekst bestaat:

- Modellen zijn gebaseerd op ondubbelzinnige definities van de symbolen en hun betekenis. Teksten bevatten heel wat vaagheden en dubbelzinnigheden.
- Modellen zijn visueel en dus overzichtelijk. Complexe problemen kunnen duidelijker voorgesteld worden en zijn beter te overzien. Een visueel model kan je in één oogopslag beoordelen, wat bij een tekst onmogelijk is.
- Een model verbergt bepaalde implementatiedetails die op dat ogenblik of vanuit dat perspectief niet relevant zijn. Elk model toont één aspect van de realiteit. Een klassendiagram toont bijvoorbeeld de objecten, hun kenmerken en relaties, maar laat het dynamische aspect buiten beschouwing. Toestandsdiagrammen tonen juist de toestandsveranderingen in een object, concentreren zich op het dynamische aspect, maar tonen de relaties met andere objecten niet. Door hun beperkte invalshoek zijn modellen duidelijker dan tekst.
- Door hun overzichtelijkheid en ondubbelzinnigheid zijn visuele modellen een uitstekend communicatiemiddel tussen de verschillende leden van een

projectteam. De modellen vormen een duidelijke basis waarop alle teamleden hun werk kunnen baseren. Misverstanden worden hierdoor vermeden.

- De strikte definities in een model en de visuele overzichtelijkheid hebben als gevolg dat ontwerpfouten sneller ontdekt worden: gebrek aan structuur, te hechte koppeling tussen modules, enzovoort, worden sneller ontdekt en kunnen tijdig gecorrigeerd worden.

Maar modellen alleen zijn niet voldoende. Modellen moeten aangevuld worden met teksten om het systeem volledig te beschrijven.

2.5. Vaardigheden van een analist

Uit het bovenstaande is gebleken dat een goede analist over heel wat vaardigheden moet beschikken. We geven enkele voorbeelden:

- Analytisch vermogen om elk aspect van een probleem uiteen te rafelen.
- Abstract denkvermogen om concrete problemen in algemene modellen om te zetten.
- Synthetisch vermogen om de verschillende analyseresultaten als een ovezichtelijk geheel te zien.
- Communicatieve vaardigheden om met alle betrokkenen te kunnen praten en om conflictsituaties op te lossen en om te gaan met tegenstrijdige wensen.
- Interviews afnemen om elke betrokkene te kunnen interviewen.
- Schriftelijk rapporteren.
- Mondelinge presentaties geven om het management, de klant en het ontwikkelingsteam op de hoogte te houden van de analyseresultaten en van de projectvoortgang..
- Vergaderen om de teamvergaderingen te volgen en/of te leiden.
- Kennis van en interesse voor de meest recente informatietechnologieën en trends.
- ...

De meeste van deze vaardigheden zijn voor elke informaticus van belang, maar voor een analist zijn zij essentieel. Het zou dan ook onverstandig zijn die zaken in de loop van de opleiding te verwaarlozen.

3. Belang van systeemanalyse

3.1. Kenmerken van een goed informatiesysteem

Een goed informatiesysteem is (in volgorde van belang):

- **Bruikbaar**. Het spreekt vanzelf dat een informatiesysteem moet voldoen aan de functionele en niet-functionele eisen die door de gebruiker vooropgesteld worden. Het systeem moet aan de behoeften van een klant voldoen.
- **Betrouwbaar**. De software bevat niet te veel fouten. Gebruikers kunnen erop rekenen dat de software correcte resultaten aflevert.
- **Gebruiksvriendelijk**. De software is gemakkelijk te bedienen zonder dat de interne implementatie voor de gebruiker bekend is.
- **Onderhoudbaar**. Kleine fouten moeten gecorrigeerd kunnen worden en kleine aanpassingen moeten mogelijk zijn, zonder dat de structuur degenereert en zonder dat de betrouwbaarheid in het gedrang komt. Onderhoudsactiviteit moet snel en goedkoop uitgevoerd kunnen worden.
- **Flexibel** (aanpasbaar). De software moet op vrij eenvoudige wijze aangepast kunnen worden aan nieuwe behoeften van de gebruikers en eenvoudig overdraagbaar zijn op andere hardwareplatformen of besturingssystemen. Flexibiliteit betekent ook dat de software met relatief weinig inspanning geïntegreerd kan worden in andere systemen en andere organisaties. In tijden van snelle overnames en fusies is dit een erg belangrijke eigenschap.
  In de informaticasector gaat er veel meer inspanning naar het aanpassen van bestaande systemen, dan naar het ontwikkelen van nieuwe systemen. Daarom zijn onderhoudbaarheid en flexibiliteit zeer belangrijk.
- **Efficiënt**. De kost voor ontwikkeling en onderhoud moet in verhouding zijn tot de kwaliteit van de software en het concurrentiële voordeel dat met de software behaald kan worden.

3.2. Ideaal versus realiteit

In de bovenstaande paragraaf hebben we de kenmerken van een goed informatiesysteem opgesomd. De informatiesystemen die werkelijk al deze kenmerken hebben, zijn eerder zeldzaam. Het ontwikkelen, d.w.z. het analyseren, het ontwerpen, bouwen en implementeren van een goed informatiesysteem blijkt geen gemakkelijke opgave te zijn. Dit blijkt uit de vele verhalen over geheel of gedeeltelijk mislukte informaticaprojecten.

Enkele vaststellingen:

- Schattingen variëren, maar specialisten[3] beweren dat tussen 50% en 80% van de kost van een software-applicatie (lifecycle cost) gaat naar het onderhouden en aanpassen van deze applicatie. Software ontwikkelen is duur, maar de software gebruiken is nog duurder. **Het onderhouden van software is de grootste kost, en vaak is die kost onnodig hoog omdat het systeem slecht onderhoudbaar is.**
- Heel wat software blijkt bij de aflevering niet of nauwelijks aan de verwachtingen van de gebruiker te voldoen. Maandenlang wordt door de IT-afdeling in stilte aan een nieuw informatiesysteem gewerkt. Daarvoor worden grote bedragen uitgegeven en bij de invoering blijkt een systeem te verschijnen dat de toekomstige gebruikers meer frustratie dan vreugde bezorgt en dat nauwelijks de concurrentiële kracht van het bedrijf versterkt (gebrek aan bruikbaarheid en gebruiksvriendelijkheid). **Men heeft een goed, maar nutteloos systeem gebouwd.**
- IT-projecten zijn net zoals veel andere projecten: bij het begin van het project is men te optimistisch. De projectduur en de kosten worden te laag ingeschat. In 2004 bleek de gemiddelde budgetoverschrijding 43% te bedragen[4]. 15% van de IT-projecten bleek een complete mislukking te zijn. IT-projecten zijn investeringen voor bedrijven: ze moeten opbrengen. **Veel IT-projecten zijn te duur en te laat klaar.**
- Nog erger is het wanneer informatiesystemen onbetrouwbaar blijken te zijn. In september 2004 verloren luchtverkeersleiders door een softwarefout het contact met 400 vliegtuigen die op dat ogenblik aan hun vlucht bezig waren.[5]

---

[3] Canfaro, Gerardo en Cimitile, Aniello. Software Maintenance. November 2000 op ftp://cs.pitt.edu/chang/handbook/02.pdf (geraadpleegd 2011-08-06)
Stafford e.a. Sofware Maintenance As Part of the Software Life Cycle. December 2033 op http://hepguru.com/maintenance/Final_121603_v6.pdf (geraadpleegd 2011-06-08)

[4] http://www.softwaremag.com/focus-areas/application-development/product-coverage/standish-project-success-rates-improved-over-10-years/. January 2011 (geraadpleegd 2011-08-06)

[5] Bron: http://www.cse.lehigh.edu/~gtan/bug/softwarebug.html#aircontrol

(gebrek aan betrouwbaarheid). Vliegtuigen zijn neergestort door fouten in de besturingssoftware. Patiënten zijn gestorven door slecht geprogrammeerde bestralingssystemen. Zelfrijdende auto's kunnen dodelijke ongevallen veroorzaken. **We leggen onze levens meer en meer in handen van software, maar die is dikwijls onvoldoende betrouwbaar.**

3.3. Belang van analyse neemt toe?

In het pionierstijdperk van de informatica werd er aan analyse en ontwerp weinig tijd en energie gespendeerd. Vaak werd software ontworpen volgens een trial-and-error-methode. Er werd iets gebouwd. Wanneer achteraf bleek dat het niet aan de gebruikers-behoefte voldeed of te onbetrouwbaar was, werd de software beetje bij beetje aangepast.

Die trial-and-errormethode bleek niet effectief te zijn. Men begon in te zien dat een goede analyse en ontwerp de kans op succes van een softwareproject sterk verhogen. Waarom is dat zo?

Er zijn globaal genomen drie redenen:

- Softwareontwikkeling wordt gestuurd door de klant en niet door de technologie.
- De complexiteit van software is toegenomen.
- De kosten van software zijn groter geworden.

Een **eerste reden** is dat de optiek waaruit men systemen ontwikkelde, in de loop van de tijd veranderd is. Vroeger werden veel systemen ontwikkeld vanuit een visie die vooral uitging van wat technisch mogelijk was. Informatica was nieuw en revolutionair. Elke computerrealisatie was een nieuwe revolutie en software was vaak een showcase van wat technisch mogelijk was.

Langzamerhand is men gaan inzien dat het succes van een informatiesysteem niet in eerste instantie wordt bepaald door de technische geavanceerdheid, maar door de mate waarin zo'n systeem daadwerkelijk tegemoet komt aan de behoeften van de gebruikers (de effectiviteit van een informatiesysteem). Software heeft een cruciale positie in onze maatschappij ingenomen. Bedrijven en non-profitorganisaties zijn voor hun overleven vaak op software aangewezen. Zonder goede software kunnen zij niet effectief of efficiënt werken of kunnen zij geen concurrentieel voordeel behalen. Zij zijn dan ook niet geïnteresseerd in de ICT-technologie op zich, maar willen via ICT een aantoonbaar voordeel voor hun organisatie realiseren.

Ontwikkelaars zijn gaan beseffen dat geen enkel systeem succesvol kan zijn, zolang de specificaties, de *functionele en niet-functionele eisen* die aan het systeem gesteld worden, niet precies, eenduidig en in samenwerking met de klant/gebruiker, vastgelegd zijn. Die specificaties vastleggen is een deel van de analyse.

Tweede reden waarom een goede analyse en een degelijk ontwerp belangrijker zijn

geworden is de toegenomen complexiteit van software. In het begintijdperk was de situatie overzichtelijk en beheersbaar. De software draaide op een centrale mainframe en gebruikers werkten via terminals op het systeem. Het aantal programma's was beperkt en zij werden op de centrale mainframe beheerd.

Op dit ogenblik heeft bijna iedereen minstens één computer (een smartphone, een PC, een tablet, ...). Al die systemen zijn via het internet met elkaar verbonden. Gegevens kunnen lokaal opgeslagen worden of in de cloud. Applicaties draaien deels lokaal, deels op een server.

De complexiteit van de programma's zelf en de eisen op het vlak van veiligheid en gebruiksvriendelijkheid zijn sterk toegenomen. Een ontwikkelingsteam bestaat dan ook uit verschillende mensen met verschillende competententies (analist, user-interface-ontwerper, databaseontwerper, programmeur, tester, documentatieschrijver, netwerk-specialist, ...).

In een dergelijke complexe omgeving is de oorspronkelijke trial-and-error-methode niet meer toereikend. Een *gestructureerde aanpak* waarbij gewerkt wordt in een reeks beheersbare en controleerbare stappen is noodzakelijk.

Een **derde reden** heeft te maken met de kosten van een informatiesysteem. Apparatuur wordt voortdurend goedkoper. Maar hardwarekosten zijn echter een klein deel van de kosten van een informatiesysteem.

Software schrijven on onderhouden gebeurt door mensen, en die moeten betaald worden. Die *loonkost* is meestal veel groter dan de hardwarekost van een systeem. Analyse en ontwerp kunnen die loonkost verminderen, door te zorgen dat het ontwikkelen van het systeem doelmatiger en dus sneller gaat, en door te zorgen dat een systeem gemakkelijk onderhoudbaar en aanpasbaar is.

Een andere kost is de *kost van slecht werkende systemen*. Software moet mensen helpen om hun werk te doen. Slechte software maakt mensen minder efficiënt. Slechte software kan ook schade veroorzaken.

Op het web kan je talloze voorbeelden vinden van softwareprojecten die fortuinen gekost hebben, maar nauwelijks enige meerwaare hebben gehad, of van gigantische kosten door onbetrouwbare software.

Enkele voorbeelden:

> Voedingsdistributeur Sainsbury moest in 2005 526 miljoen dollar afschrijven die geïnvesteerd was in een geautomatiseerd systeem voor supply-chain management. De artikels geraakten nooit in de winkels, maar bleven in de magazijnen door een fout in deze software. 3000 extra personeelsleden moesten uiteindelijk het magazijnbeheer

manueel doen om de situatie te herstellen[6].

Het Nederlands ministerie voor Volkshuisvesting, Ruimtelijke Ordening en Milieubeheheer (VROM) zette in 2007 een automatiseringsproject stop dat in vier jaar 16,6 miljoen euro gekost heeft. De reden van de stopzetting was dat het systeem nooit volledig bruikbaar zou worden omdat het niet compatibel bleek te zijn met alle systemen in het ministerie. Dit wijst duidelijk op een gebrekkige analyse[7].

Door bij het ontwikkelen van software een methodische aanpak te volgen en uit te gaan van een goede analyse en een doordacht ontwerp, kan men duurzame informatie-systemen bouwen :

- die aan de behoeften van de betrokken organisaties voldoen en voor hen een betekenisvolle meerwaarde betekenen;
- die goed gestructureerd zijn, overzichtelijk zijn en beantwoorden aan de eisen van betrouwbaarheid, veiligheid en overdraagbaarheid;
- die flexibel zijn en aangepast kunnen worden een nieuwe eisen of nieuwe technologieën;
- waarvan de kosten lager liggen dan de baten.

De afgelopen jaren is er echter ook een soort tegenbeweging. Iteratief ontwikkelen en 'lean' ('mager') ontwikkelen hebben opgang gemaakt. Die methodes leggen de nadruk op code schrijven, met kleine teams, en met veel, elkaar snel opvolgende versies van de software. Een uitgebreide analyse en ontwerp vooraf worden als overtollig vet beschouwd.

Dat wil echter niet zeggen dat analyse en ontwerp niet nodig zijn. Ook als je lean en iteratief werkt zijn analyse en ontwerp nuttig om sneller resultaat te boeken. Analyse en ontwerp gebeuren nog steeds, maar niet alles in het begin van het project.

4. De fasen in softwareontwikkeling

Software ontwikkelen is een gestructureerd proces, het gebeurt niet in het wilde weg. Er bestaan verscheidene **procesmodellen** die beschrijven in welke fasen men moet te werk gaan bij het ontwikkelen van software en welke taken in elke fase vervuld moeten worden. Deze fasering moet zo opgezet zijn dat het management beschikt over

[6] Charette Robert. Why Software Fails. September 2055 op http://spectrum.ieee.org/computing/software/why-software-fails (geraadpleegd 201108-06)

[7] Gijzenmijter, Martin. Peperduur ICT-project VROM mislukt. Webwereld, december 2007 op http://webwereld.nl/nieuws/49134/peperduur-ict-project-vrom-mislukt.html (laatst geraadpleegd op 2011-08-06)

controlemiddelen waarmee het kan nagaan of het project succesvol verloopt.

Na elke fase wordt er een **mijlpaaldocument** (Engels: milestone report) afgeleverd, een rapport met daarin de resultaten van de afgelopen fase, en een vooruitblik naar de volgende fase. Het management kan zo een mijlpaaldocument gebruiken om de voortgang van het project op te volgen. Na beoordeling van het mijlpaaldocument wordt beslist of het project kan doorgaan naar de volgende fase, of er bijgestuurd moet worden, of het project misschien helemaal geannuleerd moet worden.

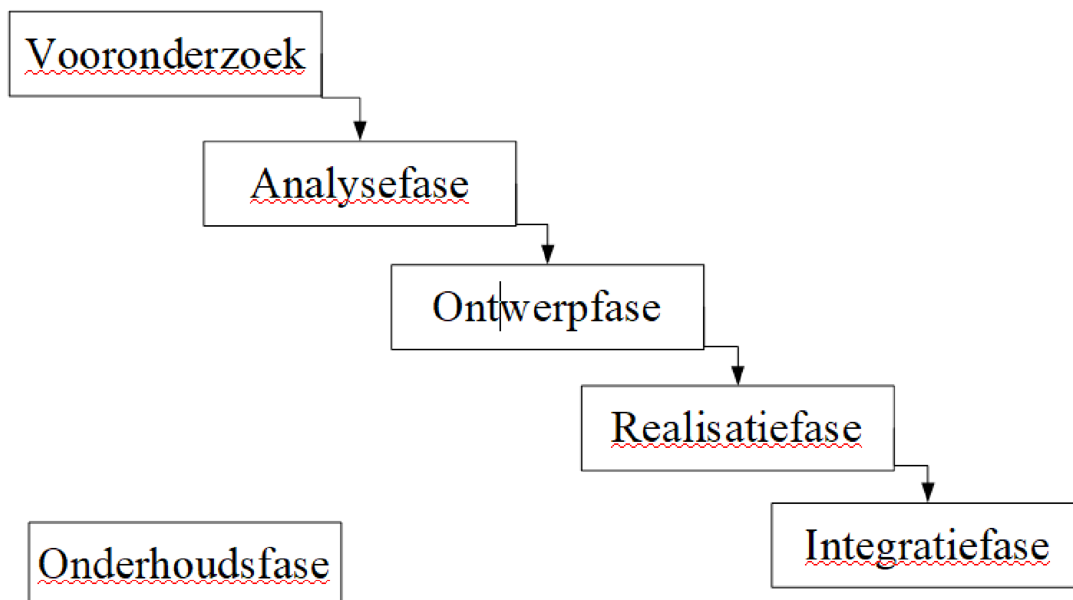
Wanneer we het in het Engels hebben over het ontwikkelingsproces gebruiken we de term '*the Software Development Life-Cycle*'.

Het **watervalmodel** is zo een procesmodel. In het watervalmodel worden de verschillende soorten activiteiten (analyse, ontwerp, realisatie, ...) in het ontwikkel-proces strikt na elkaar uitgevoerd. Het is een heel eenvoudig en overzichtelijk model. Dit watervalmodel heeft echter een aantal belangrijke nadelen, en wordt daarom in de praktijk niet veel gebruikt.

De meest gebruikte procesmodellen zijn **iteratieve modellen**. Ook hier wordt het ontwikkelproces onderverdeeld in fasen. In elke fase gebeuren alle soorten ontwikkel-activiteit. Een voorbeeld van een iteratief model is de scrum-methode. Bij scrum is het de bedoeling op het einde van elke fase een product te hebben dat je aan de gebruikers kunt tonen, zodat die feedback kunnen geven.

5. Het watervalmodel

Het watervalmodel wordt niet veel toegepast, maar is wel nuttig om te kennen. De figuur toont het schema.



Voor elk soort ontwikkelactiviteit is er een fase. De fasen volgen elkaar strikt op.

Een fase die afgelopen is, wordt later niet meer hernomen of aangepast. Het ontwikkelwerk stroomt van de ene fase in de andere, zoals water van een helling afstroomt. Vandaar de naam watervalmodel.

De inhoud van elke fase wordt hier kort besproken. Elke fase is ook een hoofdstuk in deze cursus.

5.1. Vooronderzoek

Software ontwikkelen is duur. Om niet het risico te moeten lopen dat een project halverwege strandt zullen analisten eerst een **haalbaarheid**sstudie uitvoeren. In het vooronderzoek zullen zij de volgende zaken onderzoeken:

- Men probeert een duidelijk inzicht te krijgen in de omgevingsvariabelen, m.a.w. men onderzoekt wat de bedrijfsstructuur en de huidige manier van werken is.
- De onderzoekers gaan ook de randvoorwaarden na: uiterste leverdatum, maximale kosten, personeelsbezetting, enzovoort.
- Verder probeert men een globaal inzicht te krijgen in de wensen van de klant.
- De analisten zullen een aantal alternatieve oplossingen evalueren en proberen in te schatten welke oplossing het best geschikt is voor dit probleem.
- Ten slotte zal men op basis van de voorgaande gegevens proberen de kosten en de baten van het project in te schatten en de haalbaarheid in te schatten.
- Wanneer het project kan doorgaan, wordt een globale planning opgesteld.

5.2. Analyse- of specificatiefase

In deze fase is het essentieel om een duidelijk inzicht te krijgen in het probleemgebied. De belangrijkste taken in de analysefase zijn:

- Gedetailleerd onderzoek van de vereisten.
- Modelleren van de statische structuur van het probleemgebied door het opstellen van een klassendiagram.
- Identificatie van de elementaire bedrijfsgebeurtenissen (business events).
- Identificatie van de essentiële bedrijfsregels (business rules) door het opstellen van toestandsdiagrammen en door het onderzoeken van de klasseninvarianten bij elke klasse en de precondities voor elk event.

In het algemeen kunnen we zeggen dat we in deze fase in detail gestructureerd beschrijven **WAT** er ontwikkeld moet worden.

5.3. Ontwerpfase

In de ontwerpfase beschrijven we **HOE** de applicatie ontwikkeld moet worden. Dit omvat onder andere:

- Ontwerpen van de basisarchitectuur
- Databaseontwerp.
- Ontwerp van de gebruikersinterface (UI-ontwerp).
- Gedetailleerd moduleontwerp en opstellen programmaspecificaties.
- Ontwerpen fysieke architectuur
- Ontwerpen veiligheidsplan.
- Ontwerpen procedures.
- Plannen en opstellen handleidingen, on-line help en opleidingen.
- Ontwerpen gedetailleerd programmeer- en intergratieplan.

5.4. Realisatiefase

In deze fase wordt er volop geprogrammeerd. De ontwerpmodellen worden in code omgezet, de nodige hardware, hulpmiddelen worden aangekocht en de handleidingen worden uitgewerkt en de modules worden grondig getest.

5.5. Integratiefase

In deze fase worden de afgewerkte componenten gecombineerd tot een bedrijfszeker product. Dan wordt er een systeemtest uitgevoerd om na te gaan of het product correct werkt en er worden acceptatietesten georganiseerd om na te gaan of het product aan de wensen van de klant voldoet. Ten slotte wordt de nodige hardware en software bij de klant geïnstalleerd, en neemt de klant het systeem in gebruik.
Integratie wil hier zeggen: het systeem integreren in de bedrijfsomgeving van de klant.

5.6. Onderhoudsfase

In deze fase worden aanpassingen gedaan om het systeem bruikbaar te houden voor de gebruikers.

5.7. Geen testfase?

Je vraagt je misschien af of er niet een testfase ontbreekt. Testen is nodig in elke fase. Het is naïef en zelfs gevaarlijk om het testen uit te stellen tot de software volledig gecodeerd is. Het spreekt natuurlijk vanzelf dat de testactiviteit in de realisatie- en intergratiefase relatief belangrijker wordt dan in de andere fasen. Niettemin zal er in de andere fasen ook zoveel mogelijk getest worden. Daarom brengen wij testen niet onder

in een aparte fase, maar zullen wij voor elke fase bespreken hoe er getest kan worden.

6. Een iteratief model: Scrum

Zoals hierboven al vermeld: er bestaan veel verschillende iteratieve procesmodellen. Ze hebben allemaal met elkaar gemeen dat het ontwikkelwerk verdeeld wordt in fasen die *iteraties* genoemd worden. Een iteratie heeft een min of meer vaste duur, bijvoorbeeld drie weken. Op het einde van elke iteratie wordt er gekeken of het project nog op schema zit, en of er eventueel bijgestuurd moet worden.

Scrum is een zo een iteratief ontwikkelmodel. Bij Scrum is het de bedoeling om op het einde van elke iteratie iets te hebben wat aan de gebruikers getoond kan worden. Tijdens elke iteratie worden er functies aan het product toegevoegd, tot het project af is.

Het werk dat gedaan moet worden wordt onderverdeeld in kleine onderdelen: de user stories. Van elke user story wordt geschat hoeveel tijd het duurt om die te realiseren. Bij het begin van een iteratie worden een aantal user stories uitgekozen om tijdens die iteratie te realiseren. Bij elke user story hoort wat analysewerk, ontwerp, realistatie en integratie, maar die opsplitsing wordt niet gemaakt. De user story is de eenheid van werk.

Bij scrum hoort een strikte taakverdeling. Er is een klein team van *ontwikkelaars*. De ontwikkelaars worden ondersteund door een *product owner* en een *scrum master*. De product owner is verantwoordelijk voor het contact met de klant. Hij/zij moet aan de ontwikkelaars duidelijk maken wat de wensen van de klant zijn, dus wat de user stories zijn, en welke user stories de belangrijkste zijn.. De scrum master moet er voor zorgen dat de ontwikkelaars de scrum-methodiek correct toepassen.

Een iteratie in scrum heet een sprint. Een sprint begint met een planningsvergadering, waar afgesproken wordt welke user stories er in de komende sprint gerealiseerd zullen worden. Elke dag tijdens de sprint is er een kort overlegmoment waar de voortgang besproken wordt. En op het einde van de sprint is er een demo voor de klant, en een reflectievergadering waar de afgelopen sprint besproken wordt.

Dit is nog geen volledige beschrijving van de scrum-methode. Je leert er meer over in de praktijkprojecten.

7. Objectgeoriënteerd werken: werken met objecten en klassen

In deze cursus is er gekozen voor objectgeoriënteerde analyse- en ontwerptechnieken. Wat bedoelen we daar nu eigenlijk mee?

In de cursus objectgeoriënteerd programmeren heb je uitgebreid kennis gemaakt met wat objecten zijn. De belangrijkste concepten worden hier herhaald.

7.1. Objecten

Een **object** (in analyse) is een voorstelling van een **ding** of een concept uit de realiteit. Bijvoorbeeld: een student, een klaslokaal, een stoel, een les, een vak, …

Objecten hebben een identiteit, een toestand en een gedrag.

Een object heeft een **identiteit**. Door die identiteit kun je een object onderscheiden van een ander object. Elke student is bijvoorbeeld verschillend. Zelfs studenten met precies dezelfde naam zijn verschillende personen. Alle stoelen in een klaslokaal zien er misschien hetzelfde uit, maar zijn elk aparte stoelen met een eigen indentiteit. Je kunt een stoel aanwijzen en zeggen: 'die stoel daar', en dan is voor iedereen duidelijk dat je het hebt over die ene stoel, en niet over de andere in het lokaal.

Die identiteit kun je vastleggen, door elk ding een naam of een nummer te geven. Bijvoorbeeld: elke student van VIVES heeft zijn eigen studentennummer, elke Belg heeft zijn eigen rijksregisternummer. Maar ook zonder zo een naam of nummer is er identiteit: de zakdoekjes in een pakje wegwerpzakdoekjes hebben geen naam of nummer, maar zijn toch elk aparte zakdoekjes met een eigen identiteit.

Dingen zoals de lucht die we ademen hebben geen identiteit, en zijn dus geen objecten.

Objecten hebben ook een **toestand**. De toestand is hoe een object is op een bepaald moment. De toestand van een stoel is onder andere: welke kleur de stoel heeft, waar de stoel staat, hoe de stoel staat (rechtop of ondersteboven of nog anders), of de stoel stoffig is, …

Om de toestand van een object te beschrijven gebruiken we **attributen**. Attributen zijn kenmerken van een object. Een attribuut heeft een waarde. De waarden van alle attributen van een object zijn een beschrijving van de toestand van dat object. In het voorbeeld van de stoel: de attributen zijn kleur, plaats, oriëntatie, netheid. De waarden van de attributen kleur, plaats, oriëntatie en netheid zijn delen van de toestand van de stoel.

De waarde van een attribuut kan veranderen. Bijvoorbeeld: als de stoel verplaatst wordt, dan verandert de waarde van het attribuut plaats van de stoel. Als de waarde van een meer attributen van een object verandert, dan verandert ook de toestand van dat object. Als de toestand van een object verandert, dan betekent dat de waarde van een of meer attributen verandert.

Van sommige attributen kun je de waarde nauwkeurig vastleggen: bijvoorbeeld het gewicht van iets. Voor andere attributen is dat moeilijker. Het is bijvoorbeeld lastiger om een kleur exact te beschrijven, of de netheid van iets.

Het is dikwijls ook lastig om de volledige toestand van een object vast te leggen. Om de toestand van een student volledig te beschrijven moet je de houding van zijn of haar lichaam gaan beschrijven, de gemoedstoestand, wat er allemaal in het geheugen van die student zit, en nog veel meer.

Objecten hebben ook een **gedrag**. Met gedrag wordt bedoeld: welke dingen doet een object, wat kan er gebeuren met dat object?

Een stoel kan op een andere plaats gezet worden, kan scheef gezet worden, kan afgestoft worden, … Het zijn allemaal stukken gedrag van de stoel.

Een student kan gaan zitten, rechtstaan, omvallen, les volgen, rondkijken, schrijven, … Dat zijn stukken gedrag van de student.

Gedrag zorgt er voor dat de toestand van een object verandert.

7.2. Klassen

Een **klasse** is een verzameling objecten die gelijkaardige toestanden kunnen hebben, en gelijkaardig gedrag hebben. De klasse van een ding is de soort waar het ding toe behoort.

De naam van een klasse schrijven we met een hoofdletter, om aan te geven dat het een klasse is, en niet een object.

Alle studenten behoren tot de klasse Student. Alle studenten hebben gelijkaardige toestanden, en hebben gelijkaardig gedrag. Alle stoelen behoren tot de klasse Stoel.

Een object dat tot een bepaalde klasse behoort noemen we ook een **instantie** van die klasse. Een bepaalde stoel is een instantie van de klasse Stoel. We kunnen ook zeggen dat de klasse Stoel het **type** is van het stoel-object.

Een object kan tot meer dan één klasse behoren. Een student die in zijn vrije tijd voetbalt is lid van de klasse Student en van de klasse Voetbalspeler.

7.3. Soorten objecten en klassen

In de informatica worden de begrippen object en klasse in verschillende contexten gebruikt, met licht verschillende betekenissen. Dat kan soms verwarrend zijn, maar daar is niet veel aan te doen. Er zijn in de informatica nog termen die naargelang de context een verschillende betekenis kunnen hebben. Het is nu eenmaal zo, en we moeten er mee leven.

In deze cursus worden de termen object en klasse in de volgende contexten gebruikt:

- Heel algemeen. Een object is een ding, een klasse is een soort dingen.

- Objecten en klassen in het domein van een softwaretoepassing.
- Softwareobjecten en softwareklassen.

De eerste context is die van de uitleg hierboven. De tweede en de derde context worden hieronder verder toegelicht.

7.4. Domeinobjecten en domeinklassen

Software (toch zeker toepassingssoftware) gaat over bepaalde dingen in de realiteit. Het stuk realiteit waar de software over gaat noemen we het **domein** van die software.

Bijvoorbeeld: de software voor studentenadministratie in VIVES heeft als domein de studenten van VIVES en hun activiteiten als student: de inschrijvingen, de examenresultaten, de diploma's die aan afgestudeerden uitgereikt werden en nog veel meer. Software voor het verkopen van vliegtickets heeft als domein vluchten, passagiers, tickets, luchthavens, …

Een van de activiteiten in analyse is om te proberen het domein te beschrijven: we maken een **domeinmodel**. Een model: dat wil zeggen dat we het domein niet tot in de details realistisch proberen te beschrijven, want dat is ondoenlijk. Een goed domein-model is beperkt tot de aspecten van het domein die voor de toepassing van belang zijn. Studenten zijn een deel van het domein van een studentenadministratiesysteem. De leeftijd van een student, en welke richting de student in het middelbaar onderwijs gevolgd heeft, zijn zaken die voor studentenadministratie van belang zijn, en die zouden dus in het domeinmodel moeten zitten. De kledij die een student draagt, en wat een student eet, dat zijn ook aspecten van een student. Maar die aspecten hebben voor studentenadministratie geen belang, en horen dus niet thuis in een domeinmodel voor studentenadministratie.

Een goed domeinmodel moet voldoende uitgebreid zijn, zodat het alle aspecten van het domein bevat die voor het te ontwikkelen systeem van belang zijn. Het mag echter niet te uitgebreid zijn, want dan maken we het de ontwikkelaars te moeilijk.

Een van de mogelijke manieren om een domeinmodel te maken is door te werken met **domeinobjecten**. Dan wordt het domein gemodelleerd als een verzameling objecten. De toestanden en het gedrag van die domeinobjecten zijn de toestanden en het gedrag die voor de toepassing van belang zijn.

Een domeinobject is dus een vereenvoudiging van een heel algemeen object.

Een domeinklasse is een verzameling domeinobjecten met gelijkaardige toestanden en gelijkaardig gedrag.

Een heel algemeen object kan tot meerdere klassen behoren. Domeinobjecten niet, die kunnen maar van één klasse lid zijn.

7.5. Softwareobjecten en softwareklassen

Veel programmeertalen zijn objectgeoriënteerd. Denk maar aan java, javascript, C#, Python, … Dat wil zeggen dat software die in zo een taal geschreven is een verzameling **softwareobjecten** is, die met elkaar samenwerken.

Omdat elk softwareobject apart schrijven niet efficiënt is, schrijft een programmeur de **softwareklassen**. Dat wil zeggen: hij of zij maakt een beschrijving van de mogelijke toestanden en gedrag van een hele verzameling gelijkaardige objecten. Bij de uitvoering van het programma worden van die klassen instanties gemaakt: de softwareobjecten.

Een softwareklasse heeft attributen (andere mogelijke termen zijn velden of instantievariabelen) om de toestand van de objecten van die klasse te beschrijven. Die attributen zijn data, gegevens.

Om het gedrag te beschrijven heeft een softwareklasse methodes. Die methodes zijn functies: stukken programmacode die iets doen.

Het idee dat achter objectoriëntatie steekt is dat data en functies meestal bij elkaar horen. Bijvoorbeeld: op de gegevens van een student doe je bepaalde bewerkingen, en die bewerkingen zijn alleen zinvol op de gegevens van een student. Daarom worden de gegevens van een student en de bewerkingen op die gegevens samengebracht in een klasse Student.

Net als domeinobjecten horen softwareobjecten bij één klasse.

Softwareobjecten moeten met elkaar samenwerken, om zo de functionaliteit van de software te realiseren. Softwareobjecten kunnen op twee manieren met elkaar samenwerken: een object kan de methodes van een ander object gebruiken, of een object kan de waarde van de attributen van een ander object veranderen.
Bijvoorbeeld: studenten moeten ingedeeld worden in labogroepen. In de software die daarvoor gebruikt wordt is er een klasse Labogroep en een klasse Student. Een van de attributen van Labogroep is de verzameling van alle studenten in die labogroep. Wanneer we een student willen toevoegen aan een labogroep (dus wanneer we een object van klasse Labogroep willen laten samenwerken met een object van klasse Student) kan dat op twee manieren. Ofwel voegen we de student aan de verzameling, ofwel gebruiken we een methode van Labogroep om dat te doen.
In pseudocode:

```
klasse Labogroep {
        Lijst studenten;
        voegStudentToe(Student student);
}
```

In de pseudocode is studenten een attribuut, en voegStudentToe een methode. Het verschil zie je aan de naamgeving (een zelfstandig naamwoord bij het attribuut, een

soort werkwoordsvorm bij de methode), en aan het feit dat er haakjes staan achter de naam van de methode.
De ene manier van samenwerken, het veranderen van een attribuut, zou er zo kunnen uitzien:

```
Labogroep groep1;
Student studentX;
groep1.studenten.voegToe(studentX);
```

De tweede manier zou er zo kunnen uitzien:

```
Labogroep groep1;
Student studentX;
groep1.voegStudentToe(studentX);
```

In de tweede manier is het dan de taak van de methode `VoegStudentToe()` om de student toe te voegen aan de lijst `studenten`.

De ervaring heeft geleerd dat de tweede manier de betere is. Objecten moeten met elkaar samenwerken door elkaars methodes te gebruiken, niet door elkaars attributen te wijzigen. Op die manier is het veel gemakkelijker om er voor te zorgen dat er geen dingen fout gaan. Stel bijvoorbeeld dat er in een labogroep hoogstens 20 studenten mogen zitten. Bij de tweede manier kan de klasse Labogroep in de methode VoegStudentToe controleren dat dat aantal niet overschreden wordt. Bij de eerste manier kan dat niet.

Daarom moeten objecten hun attributen afschermen, zodat andere objecten die niet rechtstreeks kunnen manipuleren. Dit heet **encapsulering**. In veel programmeertalen encapsuleer je iets door het **private** te maken (de term private is Engels, vandaar de schrijfwijze). Private attributen en private methodes zijn attributen en methodes die een klasse verbergt voor andere objecten.

De methodes van een klasse die andere objecten wel mogen gebruiken, die zijn **public**. De verzameling van alle public methodes van een klasse noemen we ook wel de **interface** van die klasse.

Het idee van encapsulering zie je ook heel veel in het dagelijkse leven. Een auto, bijvoorbeeld, heeft een 'interface' voor de bestuurder: het stuur, de pedalen, het dashboard, de versnellingspook, … Wat onder de motorkap zit is geëncapsuleerd. De bestuurder kan zich niet rechtstreeks moeien met de hoeveelheid brandstof die in een cylinder ingespoten wordt, de klepopeningstijden, en een heleboel andere details. Dit heeft voordelen: eens je de bestuurdersinterface kent, kun je met heel veel verschillende modellen auto's rijden. Je moet als chauffeur niet per se begrijpen hoe al die systemen onder de motorkap werken. Doordat de systemen onder de motorkap verborgen zijn, kun je als chauffeur een heleboel dingen niet doen die de auto zouden kunnen beschadigen.

Bij methodes horen nog een paar termen. Als een object een methode van een ander object gebruikt, dan zeggen we dat het die methode **aanroept**.
Een **parameter** van een methode is een object dat aan de methode gegeven wordt om zijn werk te kunnen doen. Bij methode VoegStudentToe is wat tussen de haken staat een parameter: het is het student-object dat aan de labogroep toegevoegd moet worden. Niet alle methodes hebben parameters, en methodes kunnen ook meer dan één parameter hebben.
Een methode kan ook een bepaald resultaat hebben die aan het aanroepende object bezorgd moet worden. Dat is de **returnwaarde** van de methode.
Al deze termen zijn van toepassing op domeinklassen en op softwareklassen.

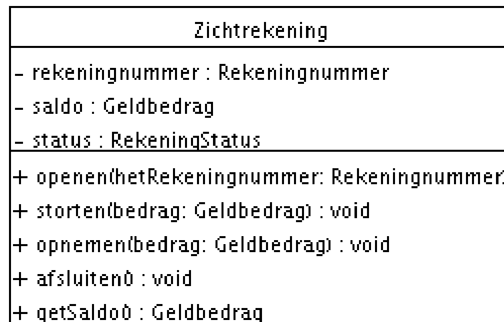
Voorbeeld:

Voor een bank moet een informatiesysteem gemaakt worden . Een objecttype binnen dit systeem is ‘zichtrekening’. Iedere zichtrekening heeft een rekeningnummer. Van een rekening willen we het saldo bijhouden, en of de rekening al dan niet afgesloten is. De diensten die objecten van het type ‘Zichtrekening’ aan andere objecten moeten aanbieden, zijn:

- openen van de rekening;
- storten van een bedrag op de rekening;
- afhalen van een bedrag van de rekening;
- afsluiten van de rekening;
- melden van het saldo van de rekening.

De klasse Zichtrekening heeft drie attributen: rekeningnummer, saldo, en 'afgesloten' (een attribuut met de mogelijke waarden ja of nee). Die attributen zijn private.

Verder heeft de klasse vijf public methodes: openen, storten, opnemen, afsluiten en getSaldo. De figuur toont een schema dat de klasse voorstelt. Dat soort schema wordt in hoofdstuk 4 besproken.

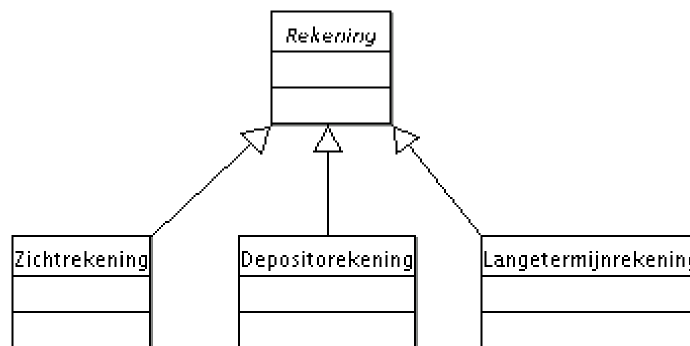


7.6. Specialisatie, generalisatie en overerving

In het vorige voorbeeld zagen we een klasse Zichtrekening. Er zijn nog andere financiële producten die op een zichtrekening lijken, zoals een depositorekening, en een langetermijnrekening. Depositorekeningen en langetermijnrekeningen hebben attributen en methodes die erg lijken op die van een zichtrekening: een nummer, een saldo, storten, geld opnemen, … Zichtrekening, depositorekening en langetermijnrekening zijn alle drie specifieke vormen van een rekening. Het concept 'rekening' is een veralgemening van de concepten 'zichtrekening', 'depositorekening' en 'langetermijn-rekening'.

In termen van klassen zeggen we dat de klasse Rekening een **generalisatieklasse** is van de klassen Zichtrekening, Depositorekening en Langetermijnrekening. Andersom is de klasse Zichtrekening een **specialisatieklasse** van klasse Rekening. Klasse Zicht-rekening is **afgeleid** van klasse Rekening.

De figuur toont hoe je dat schematisch voorstelt. (Dit soort schema's wordt behandeld in hoofdstuk 4.)



Een specialisatieklasse *erft* alle attributen en alle methodes die de generalisatieklasse heeft. Een specialisatieklasse kan daarbij nog eigen attributen en eigen methodes hebben. Een specialisatieklasse kan ook methodes van de generalisatieklasse *speciali-seren*. Dat wil zeggen dat de specialisatieklasse een of meer methodes heeft met dezelfde naam als methodes van de generalisatieklasse, maar dat die methodes iets

anders doen dan in de generalisatieklasse. (Bij programmeren in java wordt specialisatie aangegeven met @override.)

Generalisatie en specialisatie gebruik je wanneer er een 'is een'-relatie bestaat tussen klassen (in het Engels: 'is a'-relationship). Een zichtrekening *is een* rekening.

Sommige generalisatieklassen zijn zo algemeen dat er van de concepten die ze voorstellen geen instanties bestaan. Een financiële rekening is een concept dat in de praktijk niet voorkomt. Elke rekening die in het echt bestaat is van een bepaald specialisatietype. Zulke klassen die niet instantieerbaar zijn noemen we **abstracte** klassen.

Heel algemene klassen kunnen meer dan één generalisatieklasse hebben. Een laptop is een computer, en is ook een apparaat dat op batterijen kan werken, en een apparaat met een beeldscherm.

Een softwareklasse kan hoogstens één generalisatieklasse hebben. (Dat is zo omdat, toen objectgeoriënteerde programmeertalen bedacht werden, het technisch niet mogelijk was om meervoudige overerving te realiseren.)

Dat een softwareklasse maar één generalisatieklasse kan hebben is een vrij zware beperking. Daarom is er een workaround bedacht: een softwareklasse kan ook **interfaces implementeren**. Met 'interface' wordt hier bedoeld: een klasse zonder attributen, en met methodes die niks doen. Een interface is een groepje methodenamen, meer niet. Een softwareklasse kan dus maar één generalisatieklasse hebben (in java wordt daarvoor het sleutelwoord *extends* gebruikt), maar kan veel interfaces implementeren (in java wordt daarvoor het sleutelwoord *implements* gebruikt).

Domeinklassen kunnen net als heel algemene klassen meerdere generalisatieklassen hebben. In de praktijk wordt dat weinig gedaan. Het is gebruikelijk om ook bij domeinklassen de regel te hanteren dat een domeinklasse hoogstens één generalisatieklasse heeft.

Dit zijn de voordelen van overerving:

- **Hergebruik van code.** Attributen en operaties die voor alle rekeningtypen gelden, moeten niet naar de verschillende klassen gekopieerd worden.
- **Aanpasbaarheid van software.** We kunnen gemakkelijk de attributen en/of het gedrag (de operaties) van alle specifieke rekeningklassen aanpassen, beïnvloeden of uitbreiden enkel en alleen via de algemene klasse Rekening.

- **Verhinderen van redundantie.** Specificaties die voor meerdere klassen gelden, hoeven slechts op één plaats gespecificeerd te worden.

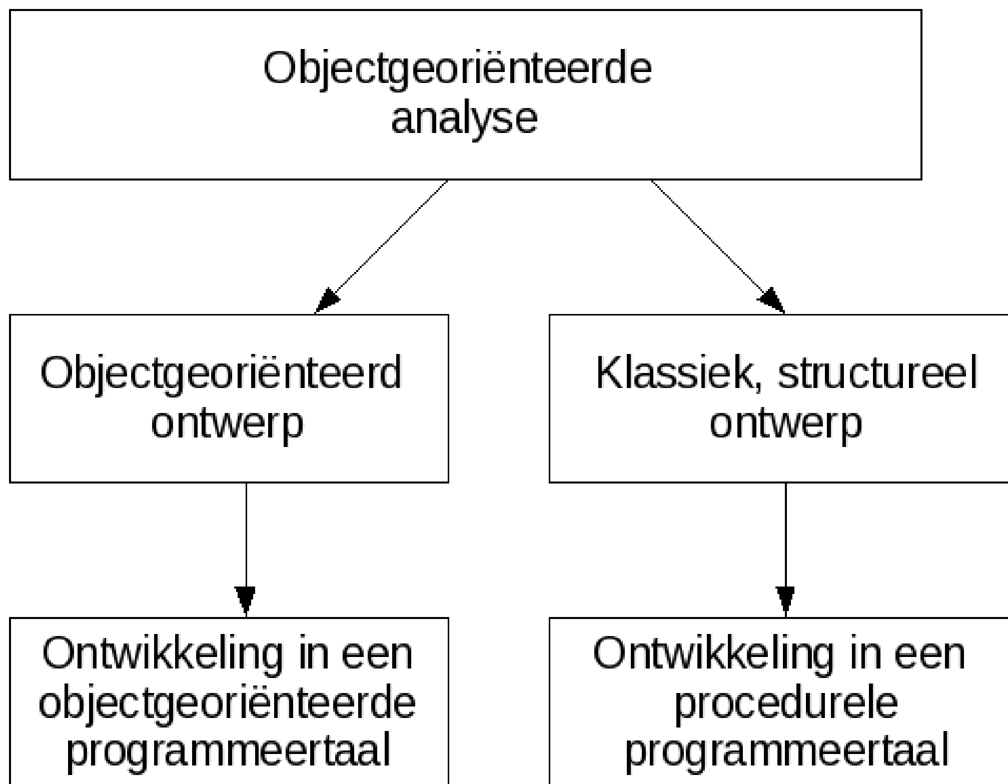
Overerving heeft ook een beperking. Als we nadenken over dingen, dan zullen we heel dikwijls beginnnen met het specifieke te zien, en dan gaan we dat in gedachten veralgemenen. Onze gedachten gaan dus van specifiek naar algemeen.
Maar bij programmeren is de werkwijze andersom het gemakkelijkst: van algemeen naar specifiek. Het is gemakkelijk om een specialisatieklasse te maken van een bestaande klasse, maar het is lastig om een generalisatie te maken van een bestaande klasse. Voor je begint te programmeren moet je je dus eerst afvragen waar er mogelijk generalisatieconcepten nodig zijn, en dat is niet altijd gemakkelijk.

8. OOA en OOP

We kunnen objectgeoriënteerd analyseren (OOA), en ook objectgeoriënteerd programmeren (OOP). In het eerste geval proberen we een model van een stuk van de wereld te maken met objecten en klassen. In het tweede geval schrijven we een programma dat is opgebouwd uit objecten en klassen.

Objectgeoriënteerde programmeertalen bestaan al langer dan objectgeoriënteerde analyse- en ontwerpmethoden. De meeste objectgeoriënteerde programmeertalen ontstonden in de jaren zeventig en tachtig van de twintigste eeuw. Op het einde van de jaren tachtig zijn die talen echt doorgebroken, en ook nu nog is objectgeoriënteerd programmeren de meest gebruikte vorm van programmeren. Objectgeoriënteerde analyse- en ontwerpmethoden kwamen pas later, halverwege de jaren negentig.

Jarenlang heeft men objectgeoriënteerd geprogrammeerd zonder een object-georiënteerde ontwerpmethode te gebruiken. Dat bleek niet zo efficiënt te zijn. Objectgeoriënteerd programmeren combineren met objectgeoriënteerde analyse en ontwerp gaf betere resultaten.
Meer nog: het bleek dat objectgeoriënteerde analyse en ontwerp ook combineerbaar zijn met klassiek procedureel programmeren (bijvoorbeeld in COBOL of RPG).

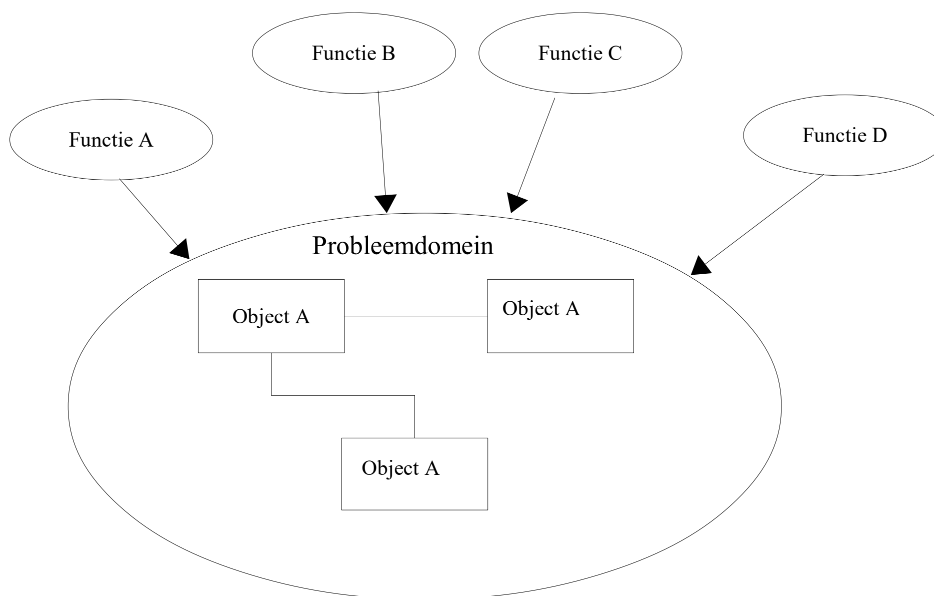


Voor men begon met objectgeoriënteerde analyse en ontwerp werden er vooral functiegerichte analysemethoden gebruikt. Bij functiegerichte analyse begin je met te zoeken naar de functionaliteiten die een systeem moet hebben. Je probeert de basis-functionaliteiten te vinden, en die dan op te splitsen in deelfuncties. Bij die functies ga je dan na welke datastromen er zijn.

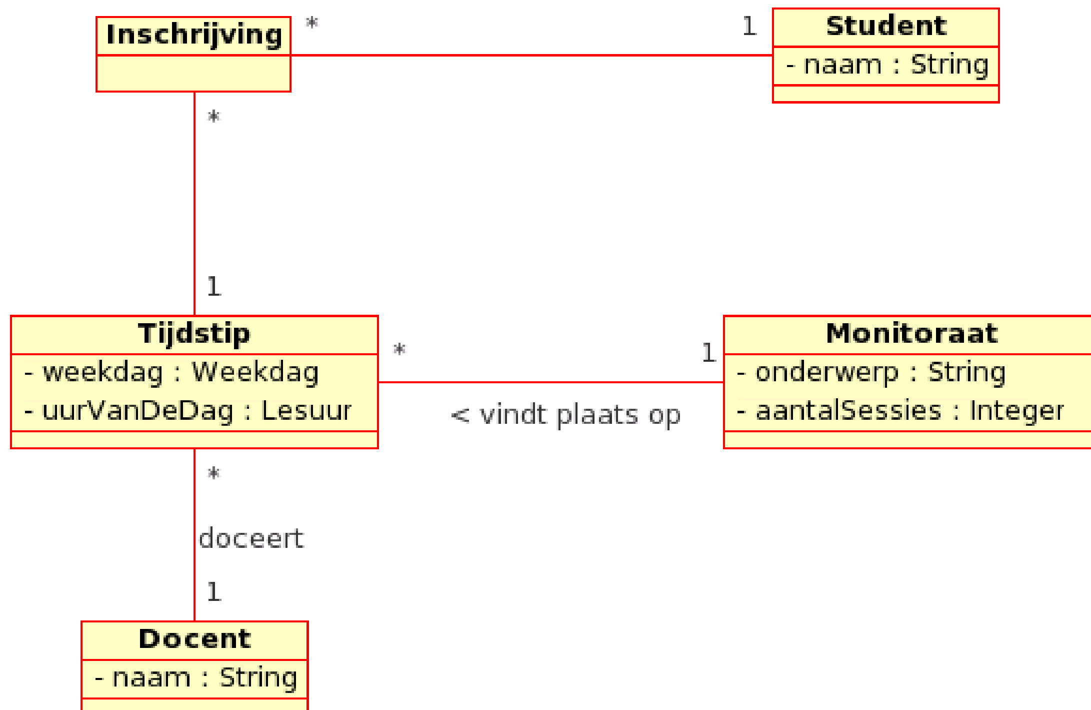
Het nadeel van die functiegerichte methoden was dat de aanpasbaarheid van systemen vaak niet optimaal was en dat er weinig mogelijkheden tot hergebruik waren. De oorzaak daarvan is dat bij functiegerichte methoden de structuur van het informatie-systeem geen weerspiegeling is van de structuur van het probleemdomein. Wijzigingen in de functionele eisen konden leiden tot verregaande wijzigingen in de software.

We moeten ons niet in de eerste plaats concentreren op de gevraagde functionaliteiten, maar op de structuur van het probleemdomein. Wanneer die structuur goed is beschreven, kunnen we nagaan welke functionaliteiten er op die structuur geënt moeten worden. Een kleine wijziging aan het probleem zal dan ook maar een kleine wijziging aan het informatiesysteem met zich mee brengen.

Bij een objectgeoriënteerde methode gaat men uit van de objecten die de basis vormen voor het probleem, de domeinobjecten. Men streeft er dus naar de structuur van het informatiesysteem te enten op de structuur van het probleemgebied. Men identificeert de domeinobjecten en hun eigenschappen en onderlinge relaties. Dit zal de basis vormen voor het systeem. Pas achteraf zal men de functies toevoegen aan de objecten. Functies wijzigen immers geregeld, de structuur van het domein wijzigt veel minder dikwijls.



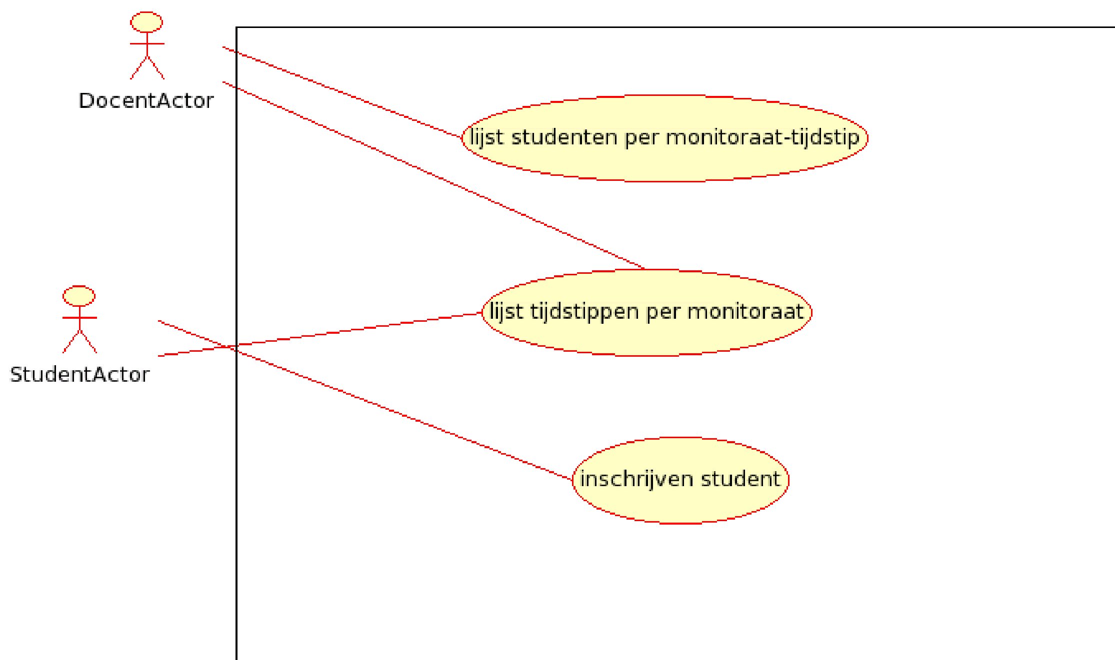
Nemen we het voorbeeld van een systeem om studenten in te schrijven voor een monitoraat. De kernobjecten in dit systeem zijn: student, monitoraat, tijdstip(pen) waarop het monitoraat plaatsvindt, docent (die het monitoraat geeft). Die objecten hebben bepaalde kenmerken en bepaalde relaties tot elkaar. Een monitoraat wordt gekenmerkt door een onderwerp en door het aantal sessies dat georganiseerd wordt. Elk monitoraat heeft een relatie met de tijdstippen waarop het monitoraat plaatsvindt. Eén monitoraat kan immers op verschillende tijdstippen georganiseerd worden. Een monitoraat op een bepaald uur wordt door een bepaalde docent gegeven. Studenten kunnen zich inschrijven voor een monitoraat op een bepaald tijdstip. Deze structuur wordt weergegeven in een klassendiagram.



Een object krijgt ook een gedrag. Elk object kan elementaire wijzigingen ondergaan. Een monitoraat wordt toegevoegd. Het aantal sessies wordt gewijzigd. Er wordt een nieuw tijdstip toegevoegd aan een monitoraat. Voor een student wordt een inschrijving voor een monitoraat op een bepaald tijdstip gecreëerd, enzovoort. Deze elementaire wijzigingen noemen we bedrijfsgebeurtenissen.

De bedrijfsgebeurtenissen en de objecten vormen de kern van het systeem. De kern van het informatiesysteem heeft een gelijkaardige objectstructuur als het probleem in de reële wereld. Een kleine wijziging in het probleemdomein veroorzaakt slechts een kleine wijziging in het informatiesysteem.

Naast deze kern hebben we de functies die door de opdrachtgever gevraagd worden. Het kan zijn dat er een inschrijvingsmodule moet gebouwd worden en dat de docenten een lijst van studenten willen die deelnemen aan hun monitoraat. De lijst van functies geven we weer in een use-casesdiagram.



Wanneer er functies bijkomen of bestaande functies gewijzigd worden, blijft het probleemdomein ongewijzigd. Het komt er dan enkel op aan nieuwe functies toe te voegen. Die nieuwe functies maken gewoon weer gebruik van de diensten van de objecten in het domeinmodel. Een objectgeoriënteerde aanpak is meer bestand tegen veranderingen in de wensen van de gebruiker.

Sommige objecten zijn vrij algemeen van concept en kunnen in andere applicaties opnieuw gebruikt worden. Een objectgeoriënteerde ontwerp besteedt meer aandacht aan hergebruik.

9. UML (Unified Modeling Language)

In de jaren negentig waren er tientallen objectgeoriënteerde analysemethoden. Elke methode had zijn eigen werkwijze en zijn eigen symbolen. Er was geen sprake van enige standaard. Toch hadden alle methoden heel wat gemeenschappelijk.

Drie toonaangevende OOA-goeroes, Grady Booch, Jim Rumbaugh en Ivar Jacobson, met elk hun eigen OOA-methode, besloten de handen in elkaar te slaan met de bedoeling hun drie methodes samen te brengen in één supermethode. Vanaf dan werden zij *the three amigos* genoemd. De supermethode hebben ze niet kunnen maken. Want zijn ondervonden dat verschillende organisaties en verschillende probleemgebieden andere ontwerpmethoden vereisen.

De drie slaagden er wel in één modelleertaal te ontwikkelen, the **Unified Modeling Language, UML**. UML is een grafische specificatietaal voor objectgeoriënteerde systemen. Het is geen methode. Dit maakt UML uniek en gemakkelijk inzetbaar.

Met andere woorden: wat het gevolgde specificatieproces ook is, je kunt UML gebruiken om de resultaten weer te geven. De meeste objectgeoriënteerde methoden hanteren immers dezelfde concepten. UML probeert één standaardnotatie te definiëren om analyse- en ontwerpmodellen op te stellen. De manier waarop men tot die modellen komt, verschilt echter nog steeds van methode tot methode.

Standaardisering

Op het einde van de vorige eeuw hebben de bedenkers van UML aan OMG (Object Management Group) voorgesteld om UML te gebruiken als standaardnotatie voor objectgeoriënteerde systeemmodellering. OMG is het belangrijkste internationale standaardiseringinstituut voor objectgeoriënteerde aangelegenheden. Heel veel grotere en kleinere softwarebedrijven zijn er lid van. Sindsdien is de OMG verantwoordelijk voor de verdere ontwikkeling van UML. In 2005 werd UML ook een ISO-standaard.

De recentst vrijgegeven versie is de versie 2.5 (juli 2015). De officiële website van UML is http://www.uml.org

BPMN

UML is niet de enige modelleertaal. Er is bijvoorbeeld ook BPMN (Business Process Model and Notation). Ook BPMN wordt beheerd door de OMG.

BPMN heeft een ander doel dan UML: het dient om bedrijfsprocessen te modelleren.

Er is overlap tussen wat je met UML en wat je met BPMN kunt modelleren. UML-activiteitendiagrammen lijken heel erg op BPMN-flowcharts. De twee zijn echter niet hetzelfde.

In deze cursus wordt UML gebruikt als modelleertaal. Een aantal diagramtypes worden voorgesteld op verschillende plaatsen in de cursus. UML is echter veel uitgebreider dan wat er in deze cursus aan bod komt. Een volledig overzicht van UML zou een cursus op zich zijn.

> In deze cursus wordt uitsluitend UML gebruikt als modelleertaal. Er wordt op het examen dan ook verwacht dat je altijd UML-diagrammen gebruikt, en geen BPMN-diagrammen.

10. Opdrachten en oefeningen

10.1. Slecht functionerende software

Zoek een drietal voorbeelden van zogenaamde *softwarerampen:* software die door het slecht functioneren voor grote kosten of rampen gezorgd heeft.

10.2. Mislukte softwareprojecten

Zoek een voorbeeld van een mislukt softwareontwikkelingsproject. Een project kan mislukt zijn omdat:

- het resultaat nooit werd bereikt, de software nooit volledig ontwikkeld is geraakt;
- het resultaat helemaal niet voldoet aan de eisen van de klant;
- de ontwikkelingstijd en -kosten vele malen groter waren dan begroot.

Probeer ook na te gaan in welke fase volgens jou de oorzaak van de fout lag (voorzonderzoek, analyse, ontwerp, realisatie, integratie).