Reader System Engineering

[v0.4, 2023_02_01, door Marius Versteegen]

Inhoudsopgave

Voorwoord

Deze reader is grotendeels een beknopte, nederlandse samenvatting van de belangrijkste delen uit de volgende documenten:

- **Architectural Reasoning explained** (Muller)
- Hoofdstuk 2 van het **Incose System Engineering Handbook**
- Hoofdstuk 2 van het **NASA System Engineering Handbook**
- De paper "**Casting Software Design in the Function-Behavior-Structure Framework**" (Kruchten)

De betreffende documenten fungeren als **referentie-materiaal** voor het geval je nog wat meer over het onderwerp wilt lezen.

Verder een groot deel van:

- **SysML Distilled** (Delligatti)

Het is te veel werk om de belangrijke hoofdstukken (de eerste helft) van het laatstgenoemde boek volledig in deze reader te coveren. Het is voor deze cursus belangrijk dat je die hoofdstukken goed bestudeert. Wat in deze reader staat, geeft je een primer om die belangrijke hoofdstukken wat makkelijker te kunnen lezen. Het vervangt die hoofdstukken dus **niet**.

Een ander document dat je voor deze cursus goed moet bestuderen, is:

- **ISO 25010:2011** (voorhanden op vele websites)

Inleiding – System Engineering in vogelvlucht

Stel, je hebt een goed idee. Hoe kom je van dat **idee naar werkelijkheid**? Zo'n casus kom je typisch tegen bij het afstuderen, bij een minor of als je een startup wilt oprichten.

Dit vak geeft je een gestructureerde manier om te komen van je idee naar werkelijkheid. Het vak heet system engineering, maar het behandelt voornamelijk het "ontwerp deel" ervan: **system architecture**.

Bij dit vak ga je samen met je team een systeemarchitectuur opzetten. Het (systemengineerings-) deel dat samenhangt met het productieproces komt zelf niet aan bod. Wel leer je de systeemarchitectuur zo op te zetten, dat **system engineers** er mee aan de slag kunnen.

Vergelijking met eerdere modelling-vakken

Bij **eerdere modelling-vakken** passeerden al een aantal **UML** diagrammen de revue.

Bij **dit vak** wordt er bij het ontwerp met veel meer factoren rekening gehouden. Om dat te ondersteunen leer je nog een aantal **SysML** diagrammen bij en ook een hele waslijst andere diagrammen en overzichten.

De systeemarchitectuur die je gaat maken houdt rekening met klantenwensen, eisen, constraints, traceability (natrekbaarheid) van de impact van veranderingen, failure mode effect analysis en meer.

Ook een typerend verschil van dit vak met de eerdere modelling-vakken is de potentiele **scope** waarop het toepasbaar is. Een systeem-architectuur maak je na het volgen van dit vak ook probleemloos voor een **groot systeem**, zoals een raket naar de maan. Je concentreert je dan alleen op de grote lijnen.

Wat is een systeem?

Okee, het vak heet system engineering, maar wat is dan eigenlijk een systeem?

Een officiele definitie:

- Een "systeem" is een constructie of een **verzameling** van verschillende **elementen**, die samen een resultaat levert dat niet door de elementen alleen kan worden geleverd.

  Deze elementen of onderdelen kunnen **mensen**, **hardware**, **software**, **faciliteiten**, **beleid** en **documenten** omvatten; alle dingen die nodig zijn om de resultaten op systeemniveau te kunnen produceren.
  (NASA (2007) *NASA Systems Engineering Handbook*, SP-2007-6105 - Rev1. h2. p 21)

Vrij vertaald: een verzameling van elementen die met een bepaald doel zijn samengevoegd.

Wat is engineering?

Een officiele definitie:

- Het **toepassen** van **wetenschappelijke principes** bij het **ontwerpen** of **ontwikkelen** van **constructies**, **machines**, **apparatuur** of **productieprocessen**, of **werkzaamheden** die hiervan gebruikmaken in meerdere of mindere mate.
  (Definitie van Amerikaanse Engineers' Council for Professional Development uit 1947:)

Vrij vertaald: Als een beta/analyticus systematisch werken bij ontwerp en constructie.

Wat is system engineering?

Een officiele definitie:

- **System engineering** is an interdisciplinary field of engineering and engineering management that focuses on how to design and manage complex systems over their life cycles.
  (Wikipedia)

Vrij vertaald: Interdisciplinair analytisch systematisch **ontwerp en life cycle management** van complexe systemen.

Wat is system architecture?

Een officiele definitie:

- A **system architecture** is the conceptual model that defines the structure, behavior, and more views of a system.
  (Wikipedia)

Vrij vertaald – het is een **ontwerp van een systeem** dat heel veel aspecten van een systeem weergeeft.

System Engineering vs System Architecture

In relatie tot elkaar kun je zeggen:

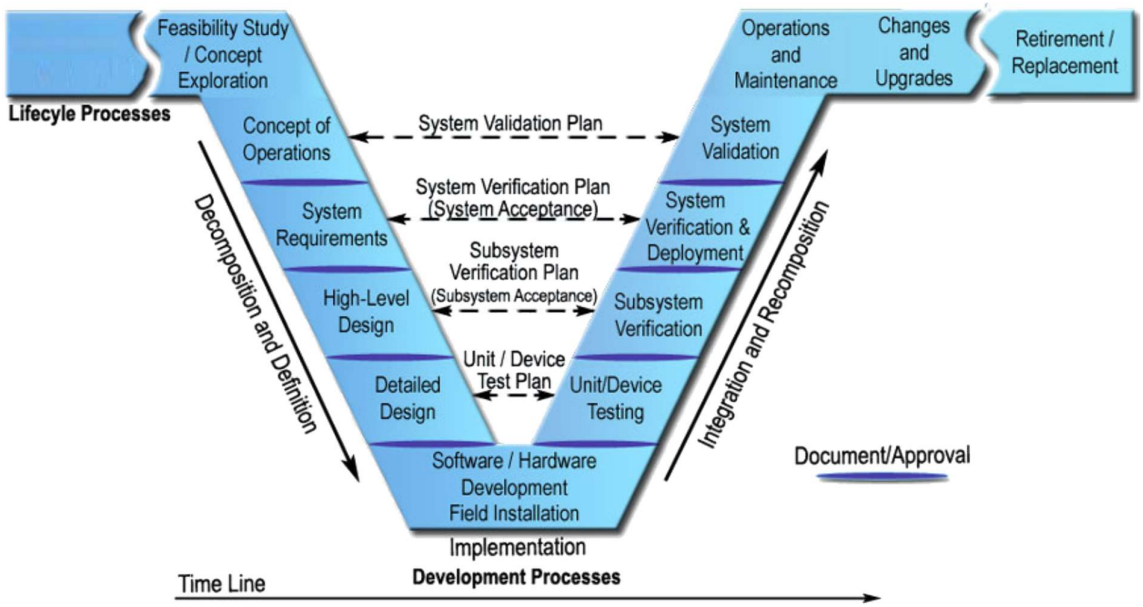
- **System Architecture** is een (**hoog niveau**) ontwerp.
  Dat is typisch ontwerp de **natte vinger**.
- **System Engineering** omvat daarop voortbouwend **ook** alle processen en cyclie om een product **uit te ontwikkelen en te bouwen**.
  Alles wordt daarbij zeg maar tot het laatste boutje doorberekend en bewezen.

System engineering methoden

System engineering kan worden bedreven met de volgende methoden:

- Het **V-model**
- De **System Engineering Engine** (NASA)

Het V-model



(Source: Kevin Forsberg en Harold Mooz (1991), *The Relationship of System Engineering to the Project Cycle.* Chattanooga, Tennessee: Proceedings of the National Council for Systems Engineering (NCOSE) Conference, pp. 57–65.)

Het V-model is een manier van werken waarbij je een groot probleem opsplitst in kleinere problemen (**decompositie**), die kleinere problemen los je op, en voeg je weer samen (**recompositie**). Zo krijg je de oplossing van het grote probleem.

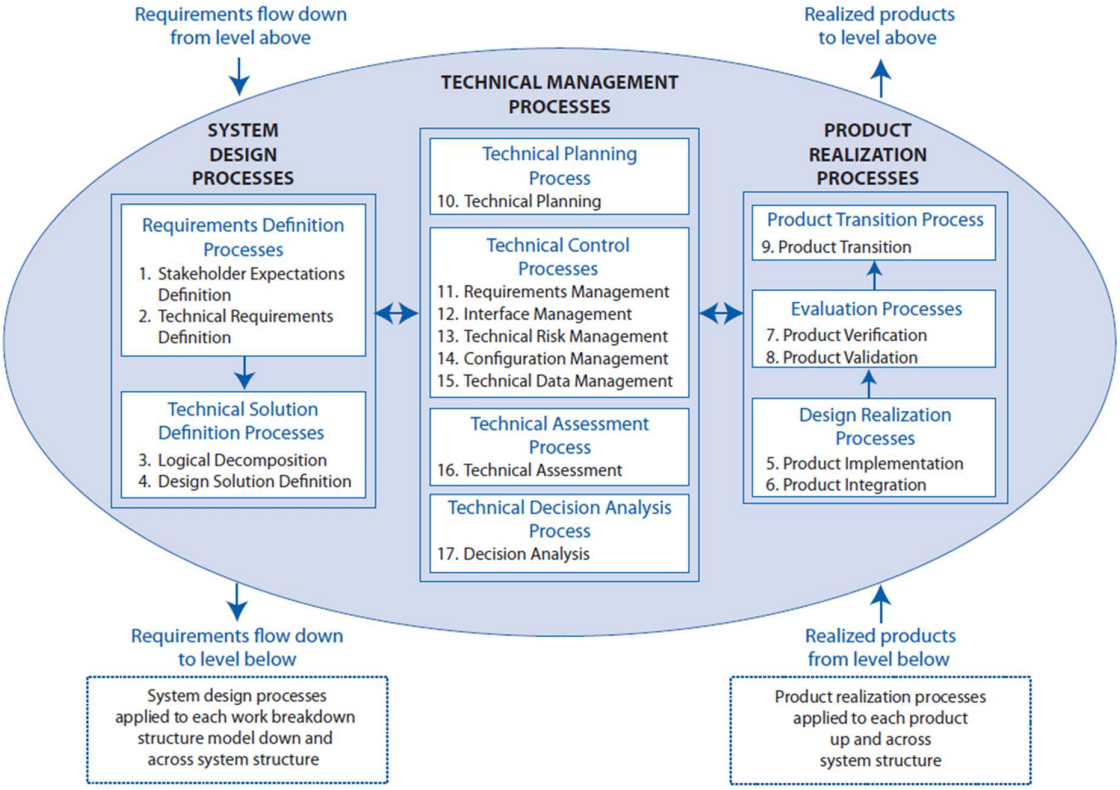
Bijvoorbeeld: Stel, je wilt een maanraket ontwerpen. Die splits je op in de 1e trap, 2e trap en 3e trap.
De 1e trap splits je op in astronautencapsule en omhulling. De astronautencapsule splits je op in chassis en apparaten. Een van die apparaten is de radio. Die splits je op in zijn onderdelen..

Die kleinste onderdelen maak je en test je afzonderlijk. Vervolgens voeg je ze samen tot de radio, en test je weer. De radio voeg je samen met de overige apparaten van de capsule. Vervolgens test je weer. Vervolgens voeg je de apparaten samen met het chassis van de capsule – en test je weer. Vervolgens bouw je de capsule in de omhulling van de 1e trap, en test je weer. Vervolgens voeg je de 3 rakettrappen samen.. en test je (voor zover mogelijk..) weer.

De **linker kant** van de V is de **ontwerp** kant. Daar ontwerp je en definieer je **requirements** voor steeds kleinere sub-systemen. De **rechter kant** van de V is de **bouw, test en integratie** kant.

Nasa System Engineering Engine

Het voorbeeld van de maanraket is niet toevallig. De Nasa werkt ook met het V-model. Dat is terug te zien in onderstaande "Nasa System Engineering Engine".



Source: NASA (2007) NASA Systems Engineering Handbook, SP-2007-6105 - Rev1. h2. p 21

In dit plaatje worden de onderdelen van het V-model gegroepeerd naar de processen/werkzaamheden die nodig zijn om ze voor elkaar te krijgen.

Als je je ogen half dichtknijpt, zie je er het V-model in terug: aan de linker kant zie je de systeem-definieering. Aan de rechter kant het integreren en testen.

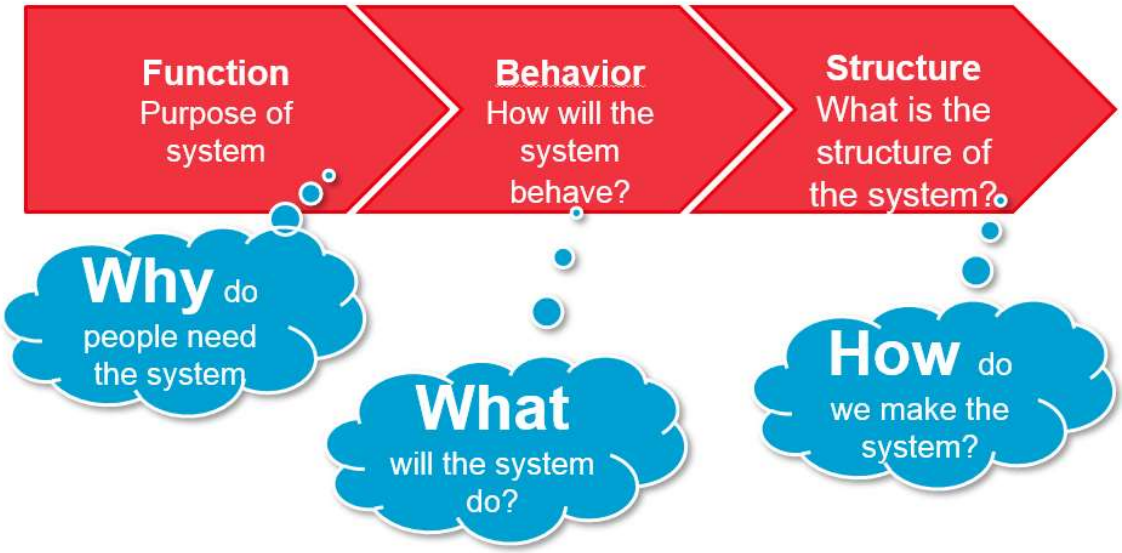
System architecting methoden

Bij system architecting wordt er geen aandacht besteed aan de creatie processen.
Voor **system architecting** zijn onder andere de volgende **methoden** te gebruiken:

- **FBS**
- **CAFCR**

Function-Behaviour-Structure Ontology (FBS)

(**Ontology** = "verzameling van **concepten** en categorieen en **relaties** daartussen")



Gero J.S. (1990) Design prototypes: a knowledge representation schema for design. *AI Magazine*, 11(4), pp. 26-36.

FBS is een system architecting methode waarbij achtereenvolgens "views" (representaties) worden gemaakt van:

1. Het **doel** van het systeem
2. Het **gedrag** dat het systeem moet vertonen om dat doel te realiseren
3. Hoe je het systeem gaat opbouwen om dat gedrag te **realiseren**.

Later volgt meer detail over deze methode.

CAFCR

Een andere system architecture methode (die bij Philips Medical vandaan komt) is **CAFCR**.

Muller G.M. (2004) *CAFCR: A Multi-view Method for Embedded Systems Architecting; Balancing Genericity and Specificity (Doctoral Dissertation)*. Available from NACSIS database (ISBN 90-5639-120-2)
Gero J.S. (1990) Design prototypes: a knowledge representation schema for design. *AI Magazine*, 11(4), pp. 26-36.

CAFCR lijkt op FBS, maar heeft op het hoogste niveau een **opsplitsing in 5 delen**:

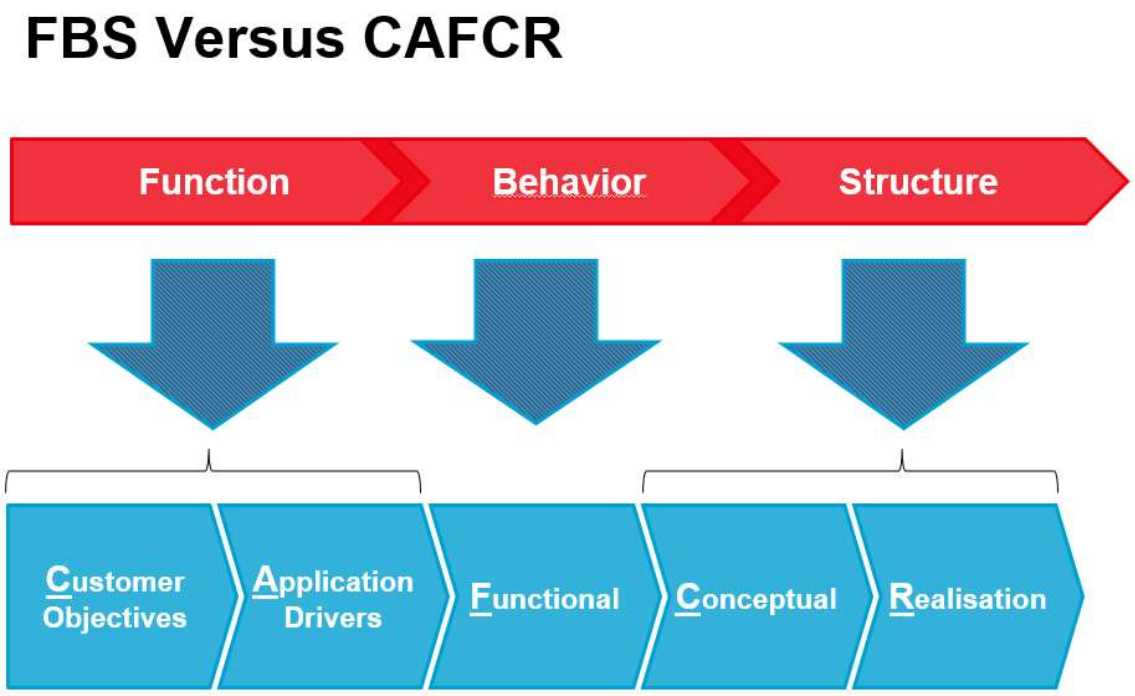
1. **Customer Objectives**: Doelen van de klant
2. **Application Drivers**: Doelen van de systeem onderdelen om de doelen van de klant te realiseren
3. **Functional**: Gedrag van het systeem dat nodig is om de doelen van het systeem te realiseren.

4. **Conceptual**: Logisch ontwerp om het gedrag te realiseren.
5. **Realisation**: De daadwerkelijke fysieke hardware die nodig is om het logisch ontwerp te realiseren.

FBS op CAFCR mapping

Elke fase van FBS en CAFCR hierboven is niet meer dan een verzameling van views.

Het eerste onderdeel van FBS komt overeen met de eerste twee fasen van CAFCR.
Het laatste onderdeel van FBS komt overeen met de laatste twee fasen van CAFCR.

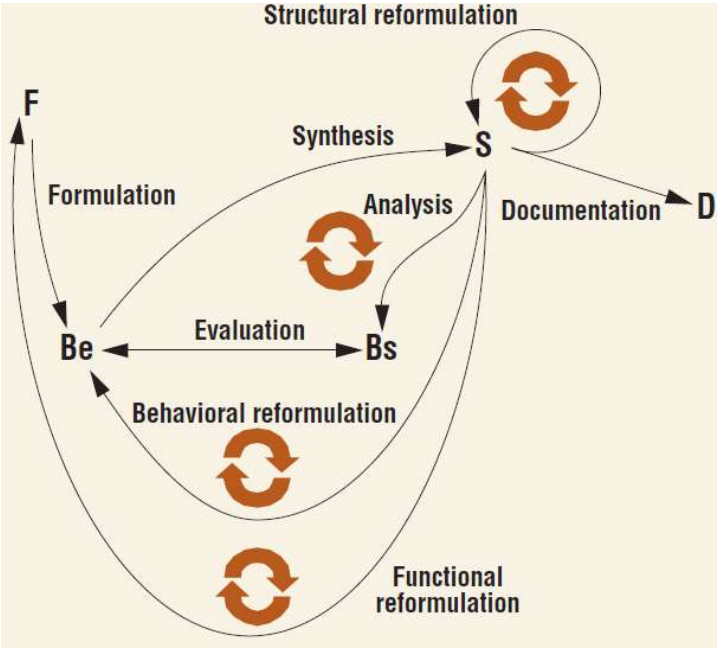


Muller G.M. (2004) *CAFCR: A Multi-view Method for Embedded Systems Architecting; Balancing Genericity and Specificity (Doctoral Dissertation)*. Available from NACSIS database (ISBN 90-5639-120-2)
Gero J.S. (1990) Design prototypes: a knowledge representation schema for design. *AI Magazine*, 11(4), pp. 26-36.

Dit is handig om te weten. We kunnen dus views van **CAFCR** gebruiken (en daar hebben we "toevallig" **gratis** documentatie van: het CAFCR lesmateriaal van Gerrit Muller) om een architectuur-document te maken met een FBS partitionering (dat is ook leuk, want **FBS** is meer een **internationaal** gangbare partitionering).

Meer FBS

Nog eens FBS, in wat meer detail. Er zit een volgordelijke methodiek achter die wordt beschreven in onderstaande diagrammen. F staat voor Function – de doelen. Be voor "Behaviour", ofwel ontworpen gedrag/functionaliteit. Bs staat ook voor "Behaviour", maar dan het gedrag dat is gemeten bij de laatst gemaakte versie. S staat voor "Structure": een versie van het product. In het plaatje lijkt het net of alleen de structuur wordt gedocumenteerd. Dat is wat geflatteerd. Uiteraard worden alle stappen gedocumenteerd.

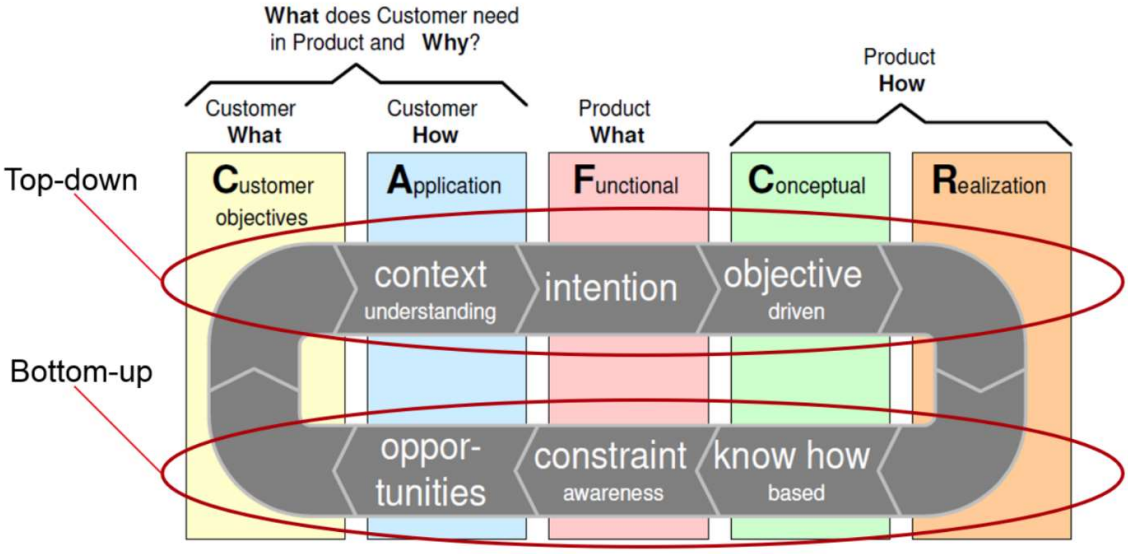


Stappen	Relaties	
Formuleren	F→Be	Requirements definiëren
Synthetiseren	Be→S	Analyse & ontwerp
Analyseren	S→Bs	Testen – reviewen
Evalueren	Be→Bs	Evaluatie tussen gewenste gedrag vs actuele gedrag (wat is gebouwd)
Documenteren	S→ D	Architectuur documentatie – Implementatie (design, realisatie) documentatie
Structuur reformuleren	S→S	Refinement of design, code, refactoring Fixing defects in design and code
Gedrag reformuleren	S→Be	Requirements wijzigen
Funtioneel reformuleren	S→ F	Systeem wijziging

Meer CAFCR

Ook bij de fasen van CAFCR hoort een soort van proces gedachte. In onderstaande diagrammen wordt die geduid:

Engineering is een cyclisch proces



Gerrit Muller (2013) *Architectural Reasoning Explained, preliminary draft*. version: 3.4. Buskerud University College, maart, 2013. h2 p.9

Elk blok dat rechtvaardigt of vereist datgene wat het blok er rechts van doet. Andersom, elk blok bevredigt of ondersteunt het blok dat er links van staat.

Net als bij FBS is er sprake van **design-cycli**. Begrip van de context leidt tot doelen. De eisen aan het gedrag worden daarvan afgeleid. Die eisen leiden tot een conceptueel design en een realisatie. Verworven kennis door metingen / analyses aan het gerealiseerde leidt tot het bewust worden van beperkingen die eerder over het hoofd werden gezien. Ook kan het leiden tot nieuwe ideeen tot verbetering of uitbreiding van het product, wat leidt tot het bijstellen van de doelen. Daar begint de cyclus weer opnieuw.

Systeem Context

Om **snel** wat **overzicht** te krijgen van wat er allemaal samenhangt met een systeem, kun je een **Systeem Context Diagram** tekenen.

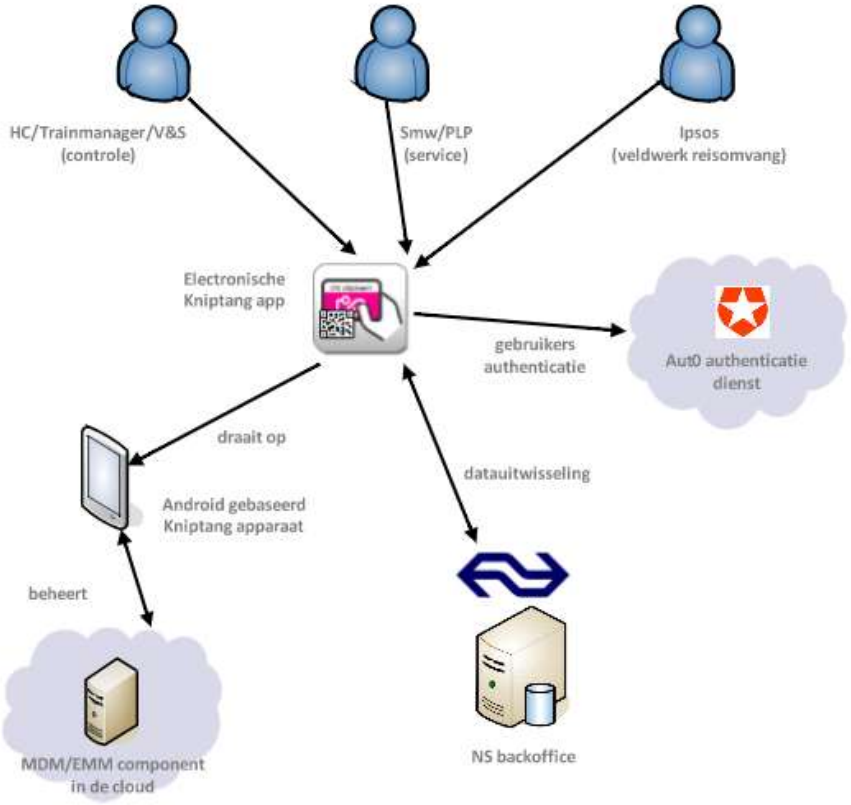
Definitie

Een definitie van zo'n diagram is:

"A **S**ystem **C**ontext **D**iagram (SCD) in engineering is a diagram that **defines** the **boundary** between the **system**, or part of a system, **and** its **environment**, showing the entities that interact with it."

"IT is **The High Level View** of a System"
(www.en.wikipedia.org/wiki/System_context_diagram)

Een voorbeeld van zo'n diagram is onderstaand context diagram voor een electronische kniptang app:



In het midden staat het zogenaamde "**System of Interest**", of afgekort "S.O.I.". Dat is het systeem dat het middelpunt is van onze analyse. De **overige** elementen bevinden zich **buiten** het systeem. Middels pijlen worden de relaties tussen de elementen en het S.O.I. verduidelijkt.

Een goed context diagram is zo helder mogelijk. De volgende eigenschappen kunnen helpen om dat te bereiken:

- **Verduidelijkende iconen** gebruiken – dat reduceert leeswerk.
- **Beperk** de hoeveelheid externe elementen tot de zeg maximaal 10 belangrijkste.
- Relaties verduidelijken met een **beschrijving** erbij.
- Een werkwoord in de **3e persoon**, zoals hierboven "draait op" als beschrijving verduidelijkt wat de actie is die het ene element op het andere uitvoert.
  (NB: Het bovenstaande voorbeeld is wat dit punt betreft dus niet optimaal)
- Als er **wederzijdse** relaties zijn, **dubbelzijdige pijlen** gebruiken ipv twee pijlen.
- Bij wederzijdse relaties die **niet symmetrisch** zijn, kun je de **verschillende beschrijvingen** langs beide uiteinden zetten.

NB: in het bovenstaande diagram mis ik nog wat duidelijkheid in de relaties. Beheren die Android phone en die MDM component elkaar nu? Wat zijn nu precies de relaties van de 3 personen met het S.O.I.?

Stakeholders

We zijn het in kaart brengen van het systeem begonnen met het maken van een context diagram. Daar heb je al enige stakeholders in gezet.
Een natuurlijke vervolgstap is om echt zoveel mogelijk stakeholders van het product in kaart te brengen, en te identificeren welke de belangrijkste zijn. Daardoor kunnen we van begin af aan rekening houden met hun eisen en wensen.

Wat zijn stakeholders?

Stakeholders zijn alle **personen of entiteiten** die een **belang** hebben bij je product/project, "belanghebbenden".

Voorbeelden van stakeholders

1. **De opdrachtgever**
   Degene die jou opdracht geeft/de eigenaar van het product/project
2. **De klant**
   Degene die het systeem gaat kopen.
   Vaak is dat de opdrachtgever, maar dat hoeft niet.
3. **De gebruiker**
   Degene die het systeem (voornamelijk) gaat gebruiken
4. **Overige stakeholders:**
   Verkoopkanalen, onderhoudspersoneel, vakbonden, overheden, omwonenden, etc.

Vaak is de opdrachtgever ook de klant. Dat hoeft echter niet.

Voorbeeld:

- Een speelgoedmerk is opdrachtgever: het geeft de opdracht tot het ontwerpen van een leuk stuk speelgoed. Laten we zeggen, een robot-huisdier.

- Ouders zullen het speelgoed gaan kopen. Dat zijn de klanten.
- Kinderen zijn de gebruikers.

Hoe brengen we de stakeholders en hun belangen in kaart?

We brengen de stakeholders in kaart in 3 stappen:

1. Stakeholders **identificeren**:
   We proberen zoveel mogelijke stakeholders te ontdekken.
2. **Doelbepalingen**:
   Per stakeholder inventariseren we welke belangen ze hebben met het product.
3. Stakeholders **classificeren**:
   We ordenen de gevonden stakeholders op invloed en betrokkenheid.
   Op basis daarvan kunnen we bepalen wat de belangrijkste stakeholders zijn.
   Dat is belangrijk om te weten, omdat we dan tijdens het design rekening kunnen houden met hun belangen.

Stakeholders Identificeren

Voor het identificeren van stakeholders kun je – naast vrij brainstormen - gebruik maken van de volgende methoden:

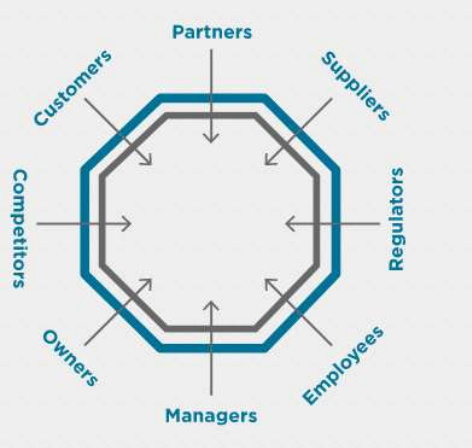
1. Stakeholder **wheel**
2. Stakeholder **nominatie**
3. Stakeholder **achtergrondonderzoek**

Stakeholder wheel

Het “stakeholder wheel” is niets meer dan een lijstje van veel voorkomende stakeholders dat je even kan turven als onderdeel van je effort om zoveel mogelijk stakeholders in kaart te brengen:

- Leveranciers
- Leidinggevenden
- Medewerkers
- Managers
- Eigenaren
- Concurrenten
- Klanten

Het lijstje is bedacht door de Britisch Computer Society (BCS). Het heet het stakeholder “wheel”, omdat zij de leden van dit lijstje in een circeltje hebben weergegeven. Door ze in een circeltje weer te geven, wordt duidelijk dat het lijstje niet bij voorbaat al gesorteerd is op belangrijkheid. Vergelijkbaar met waarom koning arthur een ronde tafel gebruikte voor zijn “ridders van de ronde tafel”.



Stakeholder nominatie

De stakeholders die je al gevonden hebt, kun je vragen of zij nog andere mensen, groepen, organisaties of afdelingen kennen die te maken krijgen met het systeem.

Stakeholder achtergrondonderzoek

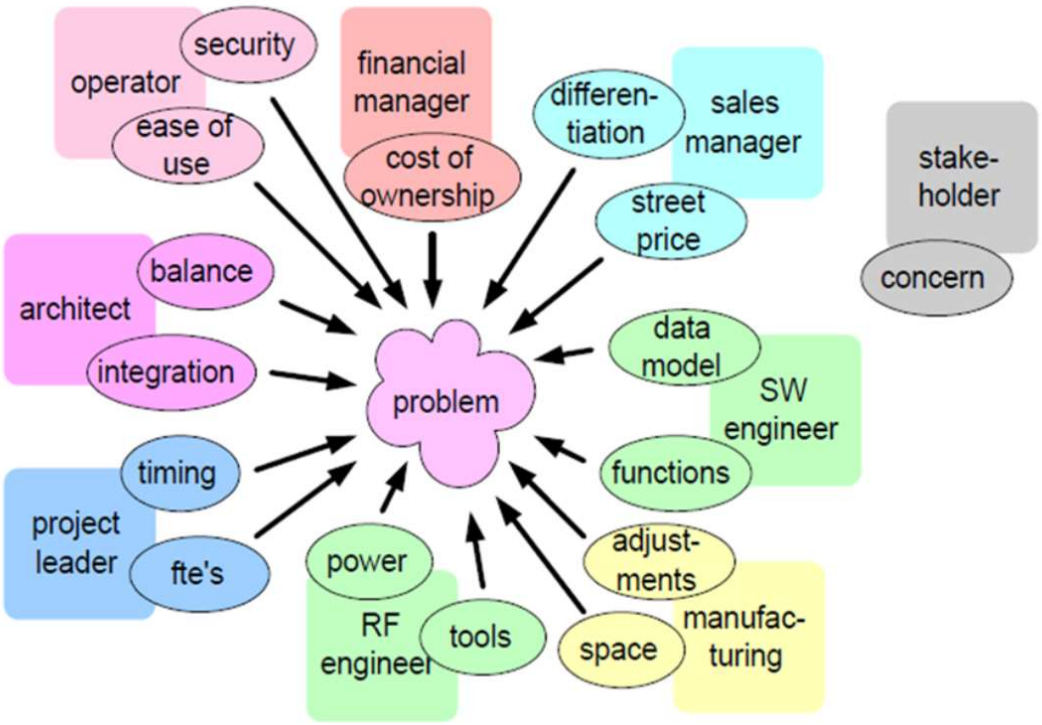
Op stakeholders die al gevonden hebt, kun je achtergrondonderzoek verrichten. Daardoor kom je meer te weten over partijen waar de betreffende stakeholder mee te maken heeft. Misschien dat daar ook stakeholders voor jouw product bij zitten.

Bestudeer daartoe bestaande literatuur, websites en dergelijke. Onderzoek:

- Wie er in de markt actief zijn
- Wie er betrokken zijn bij het product/systeem
- Welke belangen er zijn
- Of er groepen of organisaties zijn die belangen behartigen

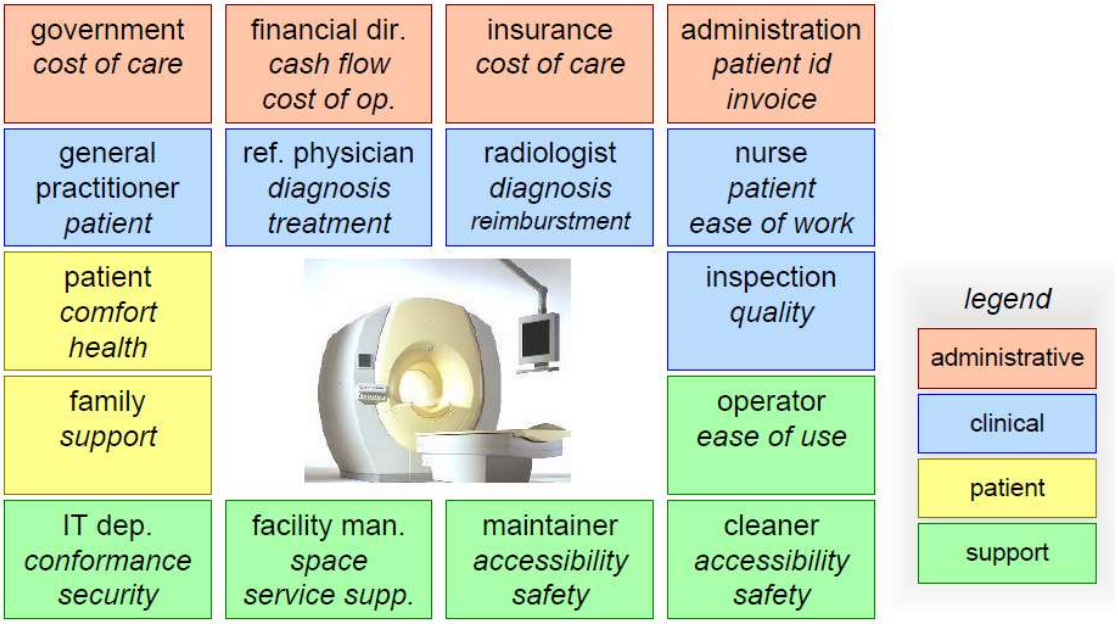
Doelbepalingen

Je hebt een groep stakeholders verzameld. Nu willen we voor elk van hen de belangrijkste belangen in je product vinden. Je kunt dat doen door elk van en te interviewen. Je kunt vaak ook een eind komen met zogeneaamde “view point hopping”.
Je zet jezelf beurtelings de pet op van een van een stakeholder, en je probeert vanuit zijn gezichtspunt te kijken wat je belangrijk vindt voor (een aspect van-) het product.



Bijvoorbeeld, met de pet van de sales manager op, bedenk je je dat de verkoopprijs een belangrijk punt is. En productdifferentiatie, zodat het product verkocht kan worden aan meerdere doelgroepen.

In onderstaande figuur is als voorbeeld een MRI-scanner genomen, en zijn er belangen van 4 stakeholders in kaart gebracht (administratie, medisch personeel, de patient en ondersteunend personeel).



Stakeholders Classificeren

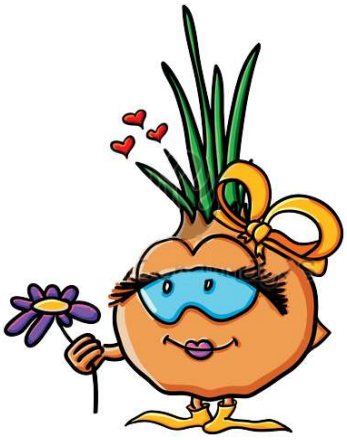
Alle stakeholders hebben belangen bij jouw systeem. Maar ze zijn niet allemaal even belangrijk.

Het classificeren van de stakeholders helpt om te bepalen welke stakeholders je wel en niet betrekt tijdens het realiseren van het product/systeem.

Bij dit vak classificeren we de stakeholders met achtereenvolgens de volgende methoden:

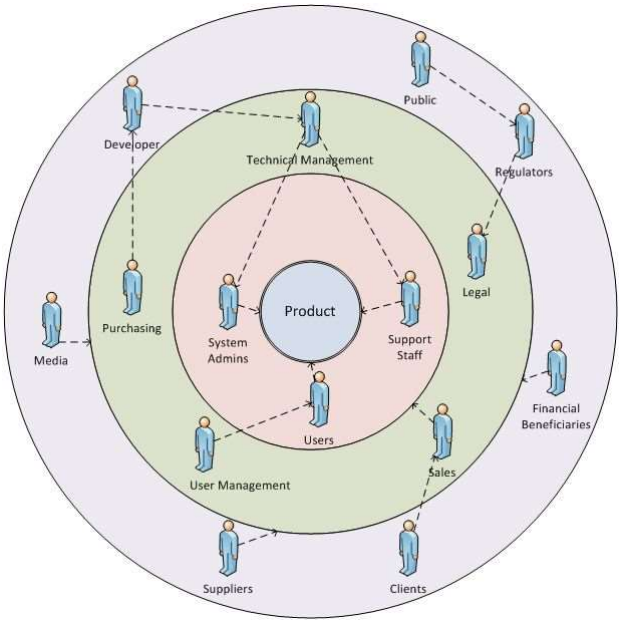
1. Een **Onion model**
2. Een **Invloed versus Betrokkenheid diagram**

Onion model



Met het Onion model kunnen we de stakeholders en hun positie ten op zichte van het beoogde systeem visualiseren.

In onderstaande figuur is een voorbeeld van een Onion model weergegeven.



Ringen

Het Onion model bestaat 4 ringen, van binnen naar buiten:

1. **Systeem/product**
2. **Actoren van het systeem**
   Stakeholders die werken met of directe interactie of belangen hebben met het systeem
3. **Business stakeholders**
   Stakeholders binnen de organisatie die niet direct interactie hebben met het systeem maar wel baat bij hebben.
4. **Omgeving**
   Stakeholders die zich buiten het bedrijf bevinden maar nog steeds belangrijk zijn en effect hebben op het systeem

Relaties

Daarnaast kun je, zoals je in het bovenstaande Onion model ziet, via gestippelde pijlen relaties tussen de diverse stakeholders weergeven.

Stakeholder invloed versus betrokkenheid

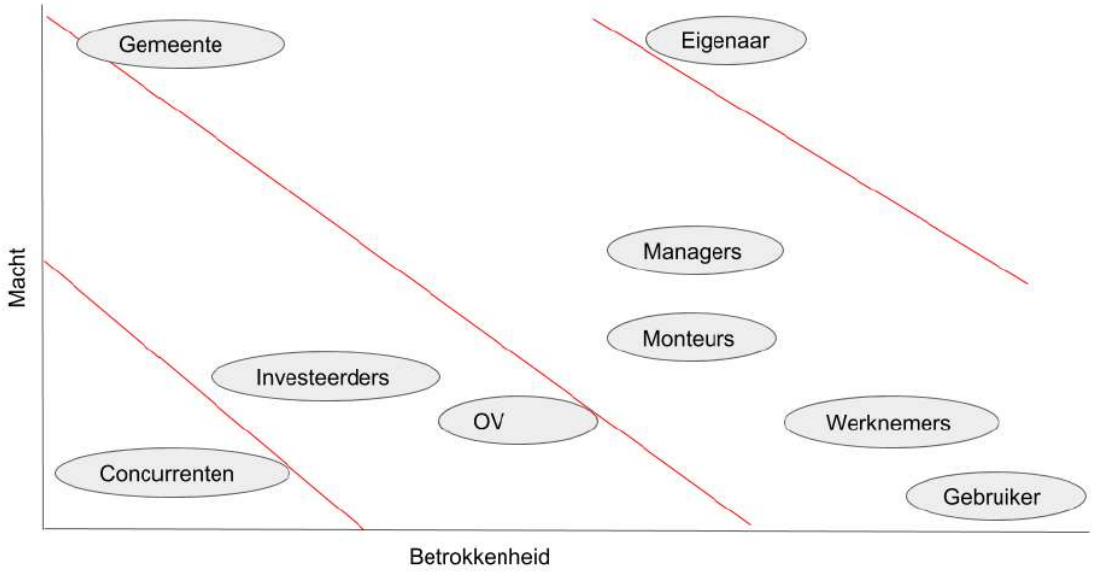
Een andere nuttige classificatie krijg je door de stakeholders te sorteren in de mate van invloed en betrokkenheid.

- Invloed
  De mate van invloed van de stakeholder op de besluitvorming. Voorbeeld: Door wettelijke beperkingen te stellen kan de wetgever een hoge invloed hebben.
- Betrokkenheid
  Hoeveel belang de stakeholder heeft bij de uitkomsten. Voorbeeld: een mogelijke uitkomst voor een pacemaker is dat de batterij lang mee gaat. Dat is belangrijk voor

de patient die het krijgt. Die hoeft dan minder vaak onder het mes om de batterij te vervangen.

Invloed versus betrokkenheid diagram

De categorisatie van invloed en betrokkenheid kun je handig visualiseren in een invloed versus betrokkenheid diagram. Een team uit een eerder jaar heeft voor een bepaald product het onderstaande invloed versus betrokkenheid diagram gemaakt. De stakeholders zijn weergegeven in ellipsen, op plaatsen die overeenkomen met hun invloed (zij noemden dat macht) en betrokkenheid. In hun geval was de betrokkenheid van de gebruiker hoog (zoals meestal), maar de macht laag. De gebruiker zou dus bijvoorbeeld een bejaarde covid-patient kunnen zijn, en het product een covid-vaccin. Het is van levensbelang dat het vaccin goed werkt, vandaar de hoge betrokkenheid. De gebruiker heeft verder weinig macht. Hij mag niet kiezen uit de vaccins, en moet maar genoegen nemen met wat er op dat moment op voorraad is.



Wat kun je met het invloed versus betrokkenheid diagram?

Het diagram is nuttig voor twee doeleinden:

1. Bepalen wie de belangrijkste stakeholders zijn
2. De strategie bepalen van hoe om te gaan met welke stakeholders

Bepalen wie de belangrijkste stakeholders zijn

In het bovenstaande invloed versus betrokkenheid diagram zie je enkele schuine hulplijnen. Als de horizontale en verticale as even lang zouden zijn, zou je ze onder een hoek van 45 graden tekenen. In dit geval is de horizontale as wat langer, dus lopen de lijnen wat vlakker.

Voor het bepalen van de belangrijkste stakeholders kun je een lineaal pakken, en die parallel aan de rechtsbovenste hulplijn leggen. Vervolgens schuif je die lineaal zonder de hoek te veranderen, richting de oorsprong van de grafiek. Zodra er een stuk of 4 stakeholders zich aan de rechter/boven kant van de lineaal bevinden, stop je. Dat zijn de belangrijkste

stakeholders. De rest van het ontwerp baseer je op de wensen en belangen van die stakeholders.

De strategie bepalen van hoe om te gaan met welke stakeholders

De stakeholders in de voorgaande diagram zou je ook kunnen opdelen in 4 kwadranten (middels 2 extra hulplijnen: een horizontale en een verticale scheidslijn. De onderstaande tabel geeft richtlijnen van hoe je het beste om kunt gaan met de betreffende stakeholders.

Voorbeeld: we hebben eerder gezien dat de ontvanger van het covid-vaccin zich in het rechtsonderste kwadrant bevindt. Het beste wat we dus tijdens de ontwikkeling van het product kunnen doen, is hem op de hoogte houden. Bijvoorbeeld: tijdig in het nieuws brengen van de voorspoedige testresultaten. Tijdig laten weten wanneer het product op de markt te verwachten is, etc.

Invloed \ Belang	Laag	Matig	Hoog	Zeer Hoog
Zeer hoog	Beïnvloeder (tevreden houden)		Belangrijke speler (vertroetelen)	
Hoog				
Matig	Toeschouwer (weinig aandacht)		Belanghebbende (op de hoogte houden)	
Laag				

Key drivers en application drivers

Eerder hebben we al de stakeholders geidentificeerd, hun doelen bepaald, en ze geclassificeerd. Daaruit volgden de belangrijkste stakeholders. Voor die stakeholders gaan we door beantwoording van de volgende vragen de zaak nader verfijnen:

- Wat wil de klant?
  -> deze doelen noemen we “key drivers”
- Wat willen de andere stakeholders?
  -> in CAFCR worden die doelen “concerns” genoemd.
- Welke (sub- / meer specifieke) doelen volgen uit de key drivers?
  -> in CAFCR worden dat “application drivers” genoemd.

Voor het gemak zien wij de klant gewoon als een van de stakeholders, en noemen we de **doelen van zowel de stakeholders als de klant** "**key drivers**".

Achterhalen wat de klant wil

We hebben ongetwijfeld in het voorgaande geconstateerd dat onze klant / opdrachtgever een van de belangrijkste stakeholders is. Het is dus belangrijk om goed uit te zoeken wat allemaal zijn belangen zijn. In veel gevallen is een klant-interview of een enquete een goed middel daartoe.

Klant-interview

In het geval van een of slechts enkele klanten is het een goed idee om die uit te vragen middels interview. Bij grote groepen potentiele klanten zou je kunnen kiezen om een steekproef van die klanten te interviewen.
(Tips voor het afnemen van een goed interview zijn te vinden in het document met die topic dat op canvas te vinden is.)

Enquete

Een andere optie is om het meer kwantitatief aan te pakken, middels het houden van een enquete.

Key drivers opstellen

We hebben nu goed de belangen van onze belangrijkste stakeholders in kaart gebracht.
De volgende stappen zijn:

1. **Sorteer** de key drivers op belangrijkheid, en neem de **eerste 4 a 8** daarvan als uitgangspunt van je verdere ontwerp. Let daarbij op dat je niet de meest vanzelfsprekende keydrivers weglaat, zoals wat het hoofddoel/functie is van het product.
2. Gebruik voor je key drivers **passende namen**

Passende namen voor key drivers kiezen

Zorg dat de namen van je key drivers voldoen aan het volgende:

1. Gebruik **korte**, voor de klant **herkenbare** namen.
2. Gebruik zoveel mogelijk markt- en klant **specifieke** namen.
   Voorkom dus vaagheid, en wees duidelijk.
   Voorbeeld: niet "hoog rendement", maar "lage integrale kosten per patient"

Application drivers

Zoals we zagen, zijn key drivers de "hoofd" doelen. Ze komen overeen met het antwoord dat een stakeholder geeft als je hem vraagt wat zijn belangrijkste wensen of zorgen zijn ten aanzien van het product. Vaak impliceert zo'n key-driver een aantal "**sub-doelen**", genaamd "application drivers". Dat zijn **diensten** die het **systeem** moet leveren / middelen om de

hoofddoelen te bereiken. Dankzij die opsplitsing kun je gerichter een oplossing gaan ontwerpen.

Voorbeelden:

Key driver	**Bijbehorende Application Drivers**
Rapport kwaliteit	• Selecteren van relevant materiaal. • Gebruik maken van standaarden.
Lage down-time	• Procedures volgen. • Alleen onderhoud plegen als er aanleiding toe is. • Swappen met een mirror-systeem.

Requirements

Uit de key drivers en eventueel bijbehorende application drivers kun je vervolgens de **requirements** van je systeem gaan afleiden.

Wat zijn requirements?

Requirements zijn een **beschrijving** van wat de **stakeholders** van het systeem **verwachten**, en dus wat het **ontwikkelteam moet opleveren**. Het is dus een soort van contract tussen de stakeholders en het ontwikkelteam. Pas als beide partijen het over de requirements eens zijn, kan er aan de ontwikkeling worden begonnen. Die beschrijving moet **eenduidig** en **specifiek** zijn, zodat

Waarom goed geschreven requirements belangrijk zijn

- De requirements vormen **de basis** voor overeenstemming tussen de **stakeholders** en het **ontwikkelteam** over wat het product moet doen.
- De ontwikkeling gaat **voorspoediger** doordat er minder herstel nodig is vanwege ontbrekende of onbegrepen eisen.
- Het biedt een **basis** voor het schatten van **kosten** en **planningen**.

Drie soorten requirements

Er zijn drie soorten requirements: Functional Requirements, Non-Functional Requirements en Constraints

1. **Functional Requirements**
   Beschrijven **wat** het systeem moet **doen**.
   Voorbeeld: *"de robothond moet kunnen springen"*

2. **Non-Functional Requirements**
   Beschrijven **overige kenmerken** van het systeem (dwz: kenmerken die niet onder Functional requirements of Constraints vallen)
   - **Kwaliteitsaspecten** (performance, onderhoudbaarheid, etc) – bijvoorbeeld **de mate waarin / hoe** functionele requirements vervuld moeten worden.
     Voorbeeld: "de hoogte die de robothond kan springen bedraagt ten minste 1 meter"
   - Beschrijving van **interfaces**:
     Voorbeeld: "de robothond beschikt over een bluetooth interface, tbv draadloze aansturing."
   - Beschrijving van **algemene fysieke kenmerken**:
     Voorbeeld: "De robothond weegt niet meer dan 5 kilo"
3. **Constraints**
   Beschrijven beperkingen van je systeem.
   Beperkingen van buitenaf – situaties van overmacht waar je niets aan kunt doen. (door wetgeving, door budget, door natuurwetten e.d.)
   Voorbeeld: "de robothond mag geactiveerd zijn op prive-terrein, omdat de wet dat voorschrijft"

Requirements opstellen

Een requirement kun je overzichtelijk vastleggen met tabellen in het onderstaande format:

F01 - Houvast	
Omschrijving	Gebruikers moeten zich ergens aan vast kunnen houden.
Rationale	De gebruikers moeten zich vast kunnen houden, zodat zij niet vallen als ze hun evenwicht verliezen.
Business Priority	Must have

- **ID en Naam**
  Het eerste veld zie je een **unieke ID** van de requirement. De ID's bestaan uit een letter en een getal.
  - De ID's voor Functional requirements beginnen met een **F**, die van NonFunctional met **NF** en constraints met **C**.
  - Door het te laten volgen met een **uniek nummer**, kun je de bijbehorende tabel makkelijk opzoeken. De volgorde van de nummers is niet van belang. Ook is het niet erg als er een nummer wordt overgeslagen. Als bijvoorbeeld een Functional requirement komt te vervallen, hoef je niet nummertjes van de rest door te schuiven o.i.d.

  De **naam** moet bondig de essentie weergeven. Het mag ook een kort zinnetje zijn.
- **Omschrijving**
  De omschrijving heeft dusdanig detail, dat het geen misverstanden kan opleveren.

Dus als het ontwikkelteam de omschrijving realiseert, dan krijgt de opdrachtgever zonder twijfel wat hij wil (meer hierover volgt in de komende paragrafen).

- **Rationale**
  De rationale geeft de achterliggende gedachte, de reden van de requirement.
- **Business Priority**
  De business priority is een van de woorden uit **MoSCoW** (Must have, Should have, Could have, Won't have)

Het is niet altijd nodig om alle velden toe te voegen. Bijvoorbeeld bij constraints kan de business priority worden weggelaten. Heel soms is de naam al zo duidelijk dat een beschrijving niets meer toevoegt. Soms is de rationale zo'n inkoppertje dat je die kunt weglaten. Maar in het algemeen geldt: "better be safe than sorry". Er is (vast) nog nooit een project mislukt door extra duidelijk te zijn. Het omgekeerde is zeker waar.

Zorg dat zowel je functional en non-functional requirements voldoen aan het onderstaande.

Gemeenschappelijke criteria voor zowel functional en non-functional requirements:

- **Specifiek**
  Zijn ze eenduidig, zonder misverstanden te interpreteren?
- **Meetbaar**
  (hoe) kan achteraf gemeten worden dat de requirement bereikt is?
- **Acceptabel**
  Is de requirement acceptabel voor de opdrachtgever/gebruiker?
- **Nuttig**
  Draagt de requirement bij aan een doel van een stakeholder?
- **Realiseerbaar**
  Is de requirement haalbaar?
- **Vermijd onnodige implementatie-keuzes**
  Fout voorbeeld: "de robot kan objecten optillen met een hydraulische grijparm"
  Tijdens de implementatie kan blijken dat een servo gestuurde grijparm een betere keuze is.
  Het is daarom beter om de requirement te focussen op wat het syteem moet doen: "De robot kan objecten vastgrijpen en optillen"
  De implementatiekeuzes komen dus zo veel mogelijk verder op de rit aan bod.
- **Vermijd keuzes die alleen een sub-systeem aangaan**
  Fout voorbeeld: "de grijparm van de robot gaat dicht na het aansturen van een "ga dicht" code via een I2C bus". Beter: belicht wat het systeem doet om een doel van de gebruiker te vervullen: "De robot grijpt een object vast zodra de gebruiker hem via een voice command daartoe de opdracht geeft".

Specifiek voor functional requirements moeten onderstaande criteria ook voldaan zijn.

Additionele criteria voor Functional requirements:

- **Maak actieve zinnen**
  Het beschrijft immers wat het systeem moet **doen**.

Ook specifiek voor Non Functional requirements zijn er additionele criteria.

Additionele criteria voor Non Functional requirements:

- **Kwantificeerbaar**
  Druk non-functional requirements **zoveel mogelijk** uit in **getallen**. Dat voorkomt misverstanden en conflicten achteraf.
  Fout voorbeeld: "de robot moet zich snel kunnen verplaatsen". De klant heeft misschien een andere mening van wat "snel" is dan het ontwikkelteam. Beter is dus: "de robot moet zich kunnen verplaatsen met een snelheid van ten minste 5 kilometer per uur."

Criteria voor Constraints:

- **Specifiek**
  Zijn ze eenduidig, zonder misverstanden te interpreteren?
- **Kwantificeerbaar**
  Gebruik waar mogelijk, getallen of formules.
  Voorbeeld: "De snelheid waarmee de sonar objecten kan detecteren is gelimiteerd door de geluidssnelheid".
  Beter is: "De snelheid waarmee de sonar objecten kan detecteren is gelimiteerd door de geluidssnelheid: minimaleDetectieTijd = objectAfstand/snelheidVanGeluidInWater"

Non Functional requirements

Zoals we eerder zagen, kunnen Non Functional requirements gaan over interfaces en over algemene fysieke kenmerken, maar dat komt niet vaak voor, omdat dat vaak zou conflicteren met de criteria dat onnodige implementatie-keuzes en keuzes die alleen een subsysteem aangaan, zoveel mogelijk moeten worden vermeden. Meestal gaan Non Functional requirements over **kwaliteitskenmerken**.

Veelgebruikte **standaardlijsten** met kwaliteitskenmerken zijn **ISO25010-2011** en **FURPS**. Het is een goed idee elk van die kwaliteitskenmerken langs te lopen, en te checken of ze van belang zouden kunnen zijn voor jouw product.

ISO25010-2011

Een beschrijving van onderstaande kwaliteitskenmerken is te vinden in het ISO25010-2011 document (download dat ergens van het internet). Een deel van de kwaliteitskenmerken gaat over de productkwaliteit. Een ander deel over de "gebruikskwaliteit".

Productkwaliteit

Functionele geschiktheid	Prestatie-efficiëntie	Uitwisselbaarheid	Bruikbaarheid
Functionele compleetheid Functionele correctheid Functionele toepasbaarheid	Snelheid Middelenbeslag Capaciteit	Beïnvloedbaarheid Koppelbaarheid	Herkenbaarheid van geschiktheid Leerbaarheid Bedienbaarheid Voorkomen gebruikersfouten Volmaaktheid gebruikersinteractie Toegankelijkheid

Betrouwbaarheid	Beveiligbaarheid	Onderhoudbaarheid	Overdraagbaarheid
Volwassenheid Beschikbaarheid Foutbestendigheid Herstelbaarheid	Vertrouwelijkheid Integriteit Onweerlegbaarheid Verantwoording Authenticiteit	Modulariteit Herbruikbaarheid Analyseerbaarheid Wijzigbaarheid Testbaarheid	Aanpasbaarheid Installeerbaarheid Vervangbaarheid

Gebruikskwaliteit

Effectiviteit	Efficiëntie	Voldoening	Vrijheid van risico	Context dekking
Effectiviteit	Efficiëntie	Bruikbaarheid Vertrouwen Tevredenheid Welzijn	Economisch risico beperking Gezond- en veiligheidsrisico beperking Omgevingsrisico beperking	Context compleetheid Flexibiliteit

FURPS

Daarnaast is er specifiek voor software een lijstje met kwaliteitskenmerken dat "FURPS" wordt genoemd:

- **F**unctionality - Features, mogelijkheden, beveiliging
- **U**sability - Menselijke factoren, esthetische eigenschappen, consistentie, documentatie
- **R**eliability - Faalfrequentie, betrouwbaarheid, faalimpact, herstelbaarheid, voorspelbaarheid, nauwkeurigheid, mean time to failure
- **P**erformance - Snelheid, efficiëntie, resourceverbruik, throughput, responsetijd
- **S**upportability - Testbaarheid, uitbreidbaarheid, aanpasbaarheid, onderhoudbaarheid, compatibiliteit, configureerbaarheid, serviceability, installeerbaarheid, localizability, overdraagbaarheid

Key Driver Graph

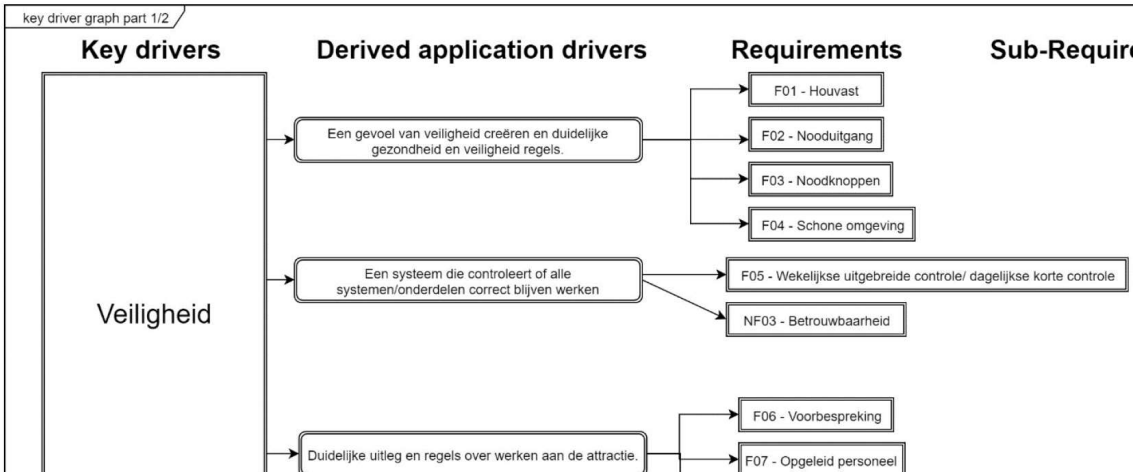
De **koppeling** tussen de **key drivers** en de **requirements** die ervan afgeleid zijn, kunnen we in een zogenaamde "Key Driver Graph" visualiseren.

Dankzij die grafiek kunnen we:

- De **klant** beter **begrijpen**
- Beter het **belang van elke requirement**t beoordelen
- Iteratief **verbeteren**
  (Bijvoorbeeld door in te zien dat bepaalde application drivers eigenlijk requirements zijn, en vica versa)

Daardoor kunnen we beter **met de klant afstemmen** wat voor product hij nodig heeft en het project **gerichter leiden**.

Een duidelijk vormgegeven Key Driver Graph (studenten van een eerder jaar) is:



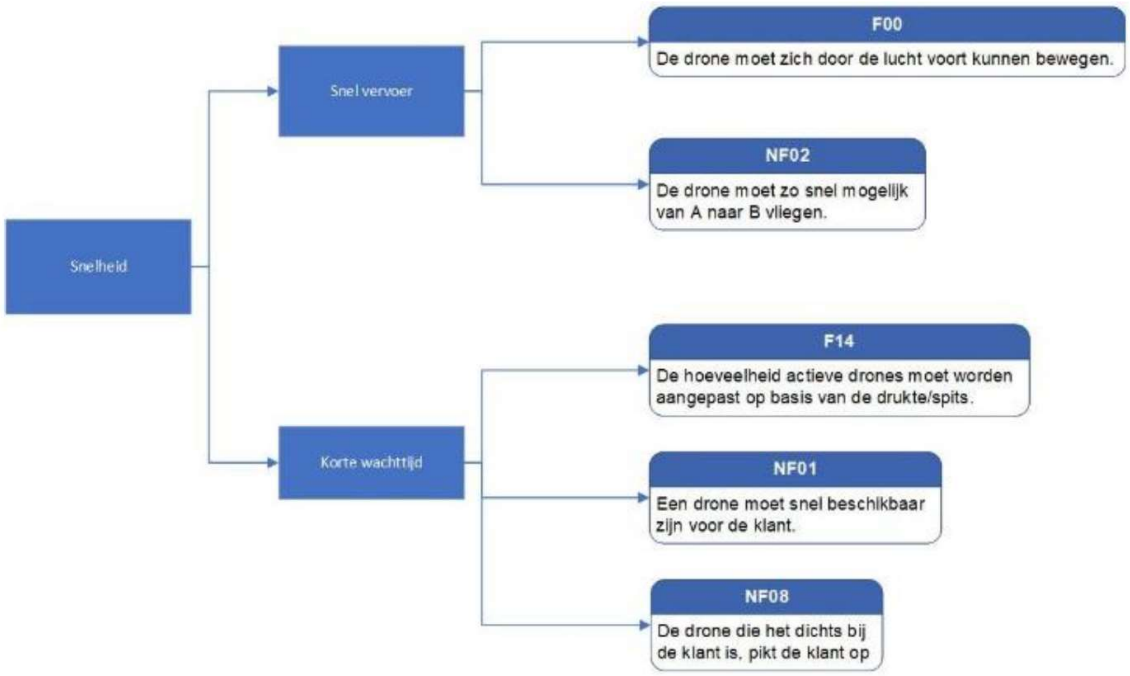
Discussie over de inhoud:

- In het bovenstaande voorbeeld komt het niet voor, maar het is ook mogelijk dat een enkele requirement van meerdere key drivers is afgeleid.
- Schone omgeving is meer een fysiek kenmerk, en dus een Non-Functional requirement.
  "Schoonhouden van de omgeving" zou wel een Functional requirement zijn (dat zou iets zijn dat het systeem doet).
- "Houvast" zit op de grens. "Er moet houvast zijn" is een fysiek kenmerk, dus non-functional.
  "Er moet houvast worden geboden" is een meer dienst die het systeem moet verlenen, en meer een functional requirement.
- "Een systeem dat controleert of alle onderdelen correct blijven werken" is geen geweldige formulering van een **application driver.** Het **doel** is geen systeem, maar

de **dienst** die het systeem levert. Een betere formulering zou dus zijn: "controleren of alle onderdelen blijven werken".

- "NF03-betrouwbaarheid" is wat niet specifiek. Wat moet er precies betrouwbaar zijn? Het staat vast in de beschrijving van NF3. Een duidelijkere naam zou bijvoorbeeld zijn "NF03-betrouwbare controles". (betrouwbare onderdelen zijn misschien ook belangrijk, maar dat is geen logische requirement om uit die application driver voort te vloeien.
- .. etc.. :-)

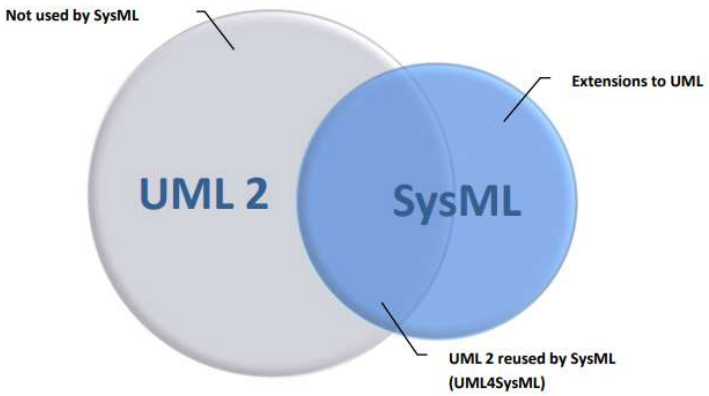
Een andere, inhoudelijk sterke key-driver graph (ook van studenten van een voorgaand jaar), is onderstaand weergegeven:



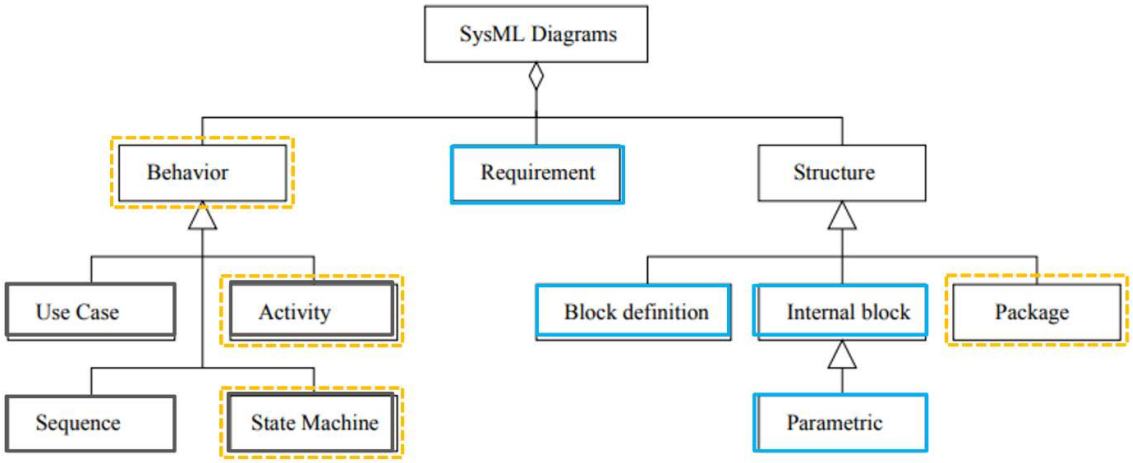
Nog wat discussie:

- De **titels** zijn bondig, maar niet te bondig – en daardoor **duidelijk**.
- Non functional requirements moeten waar mogelijk zo kwantificeerbaar mogelijk zijn. NF02: "zo snel mogelijk" alleen, is niet erg kwantificeerbaar. "De drone moet zo snel mogelijk, met een minimum gemiddelde snelheid van 30km per uur van A naar B vliegen".
  Dat is een betere, volledig geformuleerde non-functional requirement. Alleen is het minder bondig als titel, zodat de key-driver chart minder leesbaar zou worden.
  Een bondige titel met kwantificatie is natuurlijk king. Maar als dat niet mogelijk is zonder dat het onduidelijk wordt, is het ook prima om een **bondige titel zonder kwantificatie**, te gebruiken, zolang bij de detail-beschrijving van NF02 **elders** wel de goed **gekwantificeerde** beschrijving is te vinden.

UML en SysML



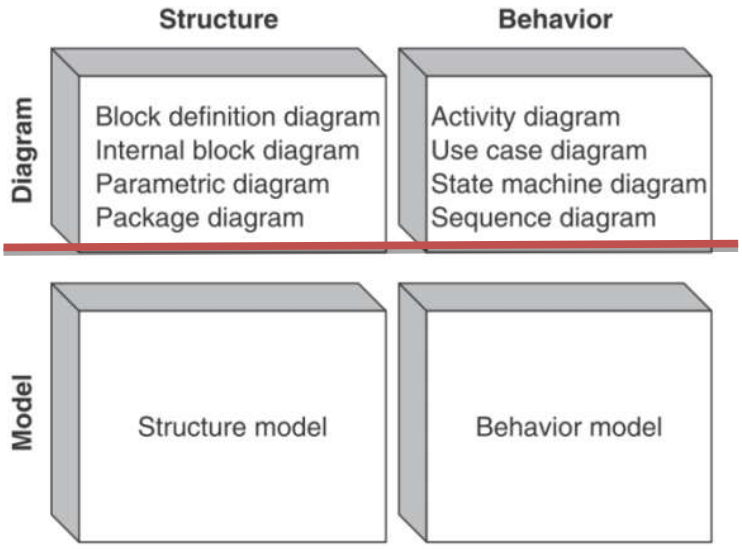
Net als UML is SysML een modelleer-taal.
Een gedeelte van de UML-diagrammen (oranje gestippeld in de onderstaande boom) wordt ook gebruikt door SysML, maar dan met uitgebreide mogelijkheden. Daardoor zijn de SysML diagrammen expressiever, en kunnen ze bijvoorbeeld ook voor hardware gebruikt worden.



Naast de uit UML geadopteerde diagrammen heeft SysML ook nog een paar addiditionele diagrammen. In bovenstaande figuur zijn die met blauw aangeduid.

SysML Diagrammen vs Models

Bij System engineering zijn de **diagrammen** "**views**" van **onderliggende** "**modellen**". System engineers gebruiken geavanceerde software. Als je het onderliggende model aanpast, worden alle er aan verwante views automatisch geupdate. Of er wordt in de betreffende views aangegeven op welke punten ze moeten worden aangepast om in sync te komen met het model.



Voor system engineers is het werken met dergelijke (dure) software onontbeerlijk. Voor system architects gelukkig een stuk minder. Tijdens dit vak gaan we het dus niet hebben over eventuele onderliggende modellen, maar beperken we ons tot de SysML diagrammen. Daarbij moeten we er dus handmatig voor zorgen dat alle diagrammen op elkaar aansluiten.

Structure vs Behavior

In bovenstaande figuur zie je ook dat de SysML diagrammen grofweg zijn op te delen in diagrammen die de structuur beschrijven (hoe het is opgebouwd) en diagrammen die het gedrag (hoe het werkt) beschrijven.

Use Case

Om tot een goed ontwerp te komen is het belangrijk om in kaart te brengen welke berichten van buitenaf bij een systeem binnenkomen en hoe het systeem daar op reageert. Precies dat wordt beschreven in zg "use cases".

Een use case is:

1. *Een **samenhangende reeks** van acties*

2. *die door het systeem worden **uitgevoerd** om*
3. *aan een **doel** van een **actor** te voldoen.*

Actor

Een actor is **iets** of **iemand** die **buiten het systeem** bestaat, en die deelneemt in de opeenvolgende **activiteiten** in een **dialoog met het systeem** om een bepaald **doel** te bereiken.

Voorbeelden van actoren zijn:

- Eindgebruikers
- Andere systemen
- Meetbare grootheden: de omgevingstemperatuur, de hoeveelheid zonneschijn.

Actoren kun je weergeven als poppetjes of rechthoeken. Als het om personen gaat, is het het duidelijkst als je poppetjes gebruikt. Als het om externe systemen gaat, is het het duidelijkst als je rechthoeken gebruikt.

Use Case Diagram

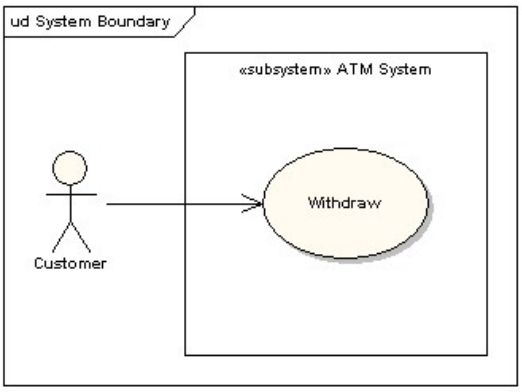
Het verband tussen actoren en use cases en use cases onderling kan worden gevisualiseerd met een Use Case Diagram.

Mogelijke relaties tussen de use cases onderling zijn:

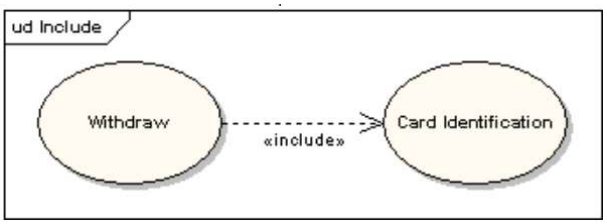
- **Include relaties**
  Betekenis: als de use case aan het **begin** van de pijl wordt uitgevoerd, wordt de use case aan het **einde** van de pijl **ook** uitgevoerd.
- **Extend relaties**
  Betekenis: als de use case aan het **eind** van de pijl wordt uitgevoerd, **kan** de use case aan het **begin** van de pijl ook worden uitgevoerd.
  (NB: de pijlrichting kan hierbij dus wat contra-intuitief overkomen)
- **Specialisatie relaties**
  Voor de volledigheid: use cases kunnen ook van andere use cases overerven. Dit zie je echter zelden (ik ben het zelf nog niet tegengekomen in een systeem architectuur).

Voorbeelden

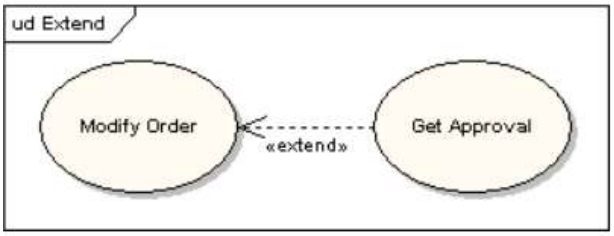
In onderstaande use case diagram zie je use case “Withdraw” binnen het ATM systeem en de actor Customer buiten het systeem.



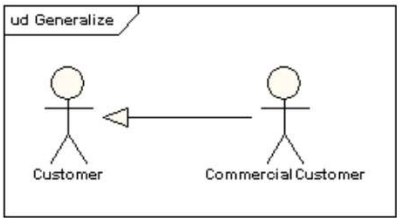
In het onderstaande use case diagram wordt de use case “Card Identification” uitgevoerd als onderdeel van een use case “Withdraw”.



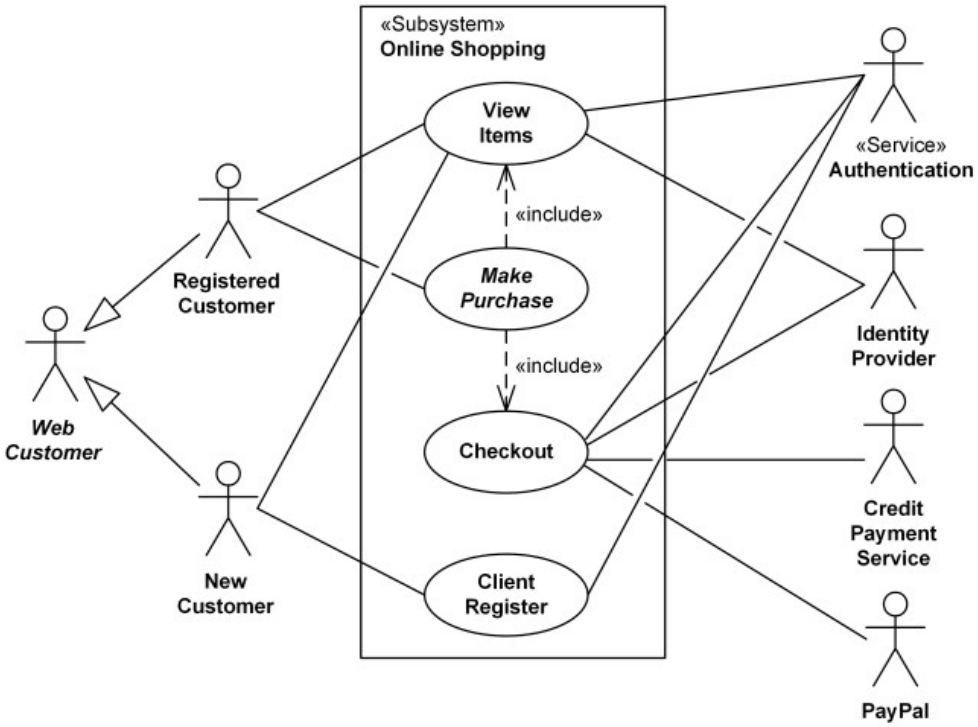
In onderstaande use case diagram kan de use case “Modify Order” optioneel de use case “Get Approval” uitvoeren.



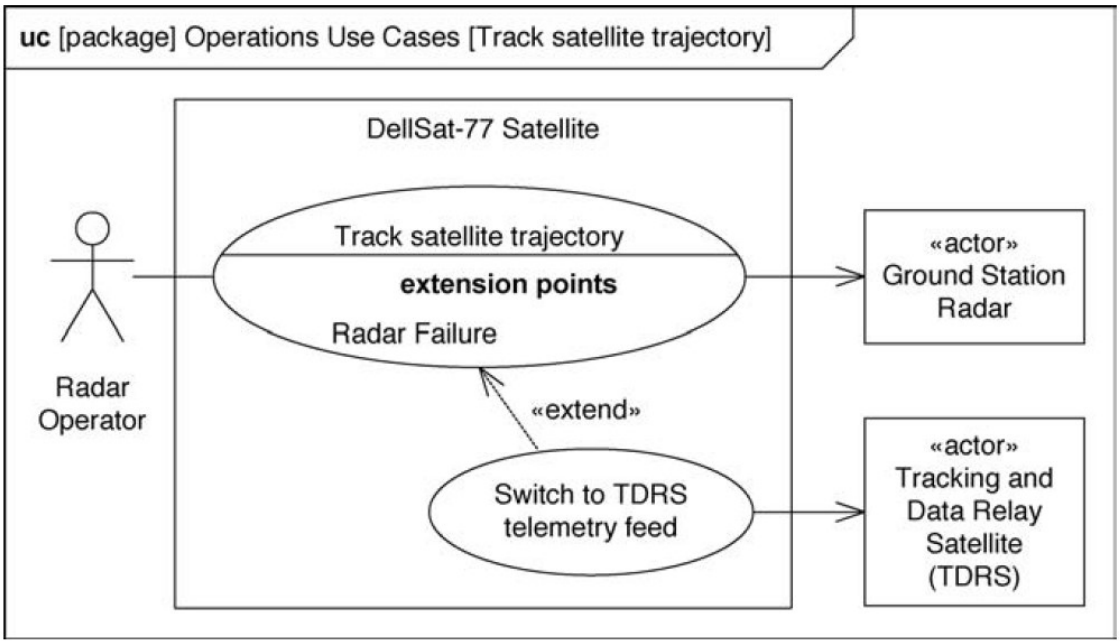
Actoren kunnen erven van andere actoren. In het onderstaande is de actor CommercialCustomer een specialisatie van de meer algemene actor Customer.



Onderstaand is een voorbeeld van een use case diagram waar enige bovenstaande relaties gecombineerd voorkomen.



Het onderstaande voorbeeld (uit Delligatti) is ook een use case diagram, maar dan in SysML stijl. Dat is de manier die we in het architectuurdocument graag terugzien.



SysML diagram – format

Discussie van bovenstaande SysML use case diagram:

- Alle SysML diagrammen zijn **omkaderd** met een rechthoek.

- Alle SysML diagrammen hebben in de linkerbovenhoek een rechthoek met een **afgeknot** puntje, de "**header**" genaamd.
- In die afgeknotte rechthoek vindt je achtereenvolgens:
    - Een **2-letterige code** die aangeeft wat de inhoud van het diagram voorstelt. In dit geval "**uc**", hetgeen staat voor "use case diagram".
    - Tussen **[]** haken zie je het **type** van iets dat het **frame** van dit usecase diagram representeert, zeg maar zijn "**parent**". In dit geval, is dat een "**package**".
    - Vervolgens komt de **naam** van die **parent** package: "Operations Use Cases".
    - Vervolgens komt tussen **[]** haken de **naam** van de **inhoud** van dit use case diagram :"Track Sattelite Trajectory". Die naam zegt meestal iets over het doel van de diagram.

      Samengevat: de header geeft aan dat dit een use case diagram is ("uc") met de naam "Track Sattelite Trajectory", welke in een ander diagram voorkomt als een package genaamd "Operations Use Cases".

      Door de parent te vermelden kun je makkelijker de verbanden zien tussen de diagrammen. Met de juiste software kun je er makkelijk doorheen **navigeren**. Bovendien verschaft het unieke **namespacing**.

      De volledige naam van de Radar Operator in het use case diagram zou bijvoorbeeld kunnen zijn: "Operations Use Cases::Radar Operator"

      Wat betekent dit concreet voor je Systeemarchitectuurdocument?
        - Als je diagram geen duidelijke parent heeft in je architectuurdocument (meestal), kun je het vermelden van parent en parent type weglaten. In dat geval vermeld je in de header alleen de type identifier en de naam van het diagram, zonder []: "**uc Traject Sattelite Trajectory**"
        - Gebruik de mogelijkheid van het vermelden van de **parent** in de header als je grote diagrammen **opsplitst** in kleinere diagrammen, of bij nesting.
          Voorbeeld: een State Transition Diagram bevat een symbool van een compound state. Die **compound state** is uitgewerkt in een ander State Transition Diagram. **Ander voorbeeld**: als je een **ibd** (een "Internal Block Diagram"-komt later) maakt, is het essentieel om de parent te vermelden (anders weet je niet van welk block het de internals beschrijft).

- Zoals het hoort, is de **grens van het systeem** dat de **use cases** afhandelt in een use case diagram aangegeven met een rechthoek. Bovenin de rechthoek is de naam van het systeem te zien: "Delsat-77 Sattelite"
- In dit use case diagram is gekozen voor de mooie methode om menselijke actoren als poppetjes weer te geven en systeem-actoren als rechthoeken met daarin het stereotype <<actor>>.

- Je ziet een **<<extend>> relatie** die zegt dat de use case "Switch to TDRS telemetry feed" **kan** optreden als de use case "Track sattelite trajectory" wordt uitgevoerd.
- Het use case diagram bevat nog een tweede extend relatie, maar die is op een andere manier genoteerd. De use case "Track sattelite trajectory" bevat daarvoor een "**extension ponts**" **compartment**, met daarin de use case "Radar Failure". Dat betekent dat (ook) "Radar Failure" kan optreden als de usecase "Track sattelite trajectory" optreed.
  Een compartment wordt in SysML altijd afgescheiden middels een **horizontale lijn**.

Lees voor **meer over use case diagrammen** het bijbehorende hoofdstuk van **Delligatti** en het bijbehorende deel van Appendix A van dat boek.

Usecase Beschrijving

Elke usecase wordt apart verduidelijkt met een usecase beschrijving. Onderstaand vind je een aardig voorbeeld uit een verslag van een voorgaand jaar:

UC03 – Melden afwezigheid	
Actor	Wolfpack
Samenvatting	Als een Wolf Unit een gebied verlaat, verlaat de Wolf Unit ook de Wolfpack. Dit wordt gemeld aan de Wolfpack en de Wolfpack kan hier de benodigde acties op ondernemen.
Preconditie	De Wolf Unit staat op het punt zijn patrouillegebied en daarmee de Wolfpack te verlaten.
Scenario	1. De Wolf Unit verstuurt een signaal dat het afwezig gaat zijn. 2. Het signaal wordt opgevangen door een of meerdere andere Wolf Units, die dit signaal verder doorsturen. 3. De Wolf Unit kan opmaken uit het herhaalde signaal of het een andere Wolf Unit heeft bereikt en indien nodig het signaal nogmaals versturen. 4. De Wolf Unit verlaat het patrouillegebied zodra hij een doorgestuurd signaal van een andere Wolf Unit binnen krijgt, of het na meerdere pogingen niet gelukt is om een andere Wolf Unit te bereiken.
Postconditie	De Wolf Unit kan het gebied verlaten met minimale impact op de werking van de Wolfpack. Het Wolfpack is in staat acties te ondernemen om de werking van de ontbrekende Wolf Unit op te vangen.
Uitzonderingen	Als de Wolf Unit niet in staat is om een andere Wolf Unit te bereiken zal de melding niet lukken.

Dus:

- Een **identificatiecode** (hier UC03) gevolgd door de usecase naam.
- **Actor** (en). De betrokken actoren
- De overige velden zijn **optioneel**, en voeg je alleen toe als het van toepassing is.
- **Scenario**. Normaal gesproken is een scenario van toepassing. I.t.t. bij CSM hoeft het bij System Engineering er **niet** een soort van uitputtende pseudocode te zijn waarin wordt beschreven hoe het systeem in elke mogelijkheid reageert. Een scenario is hier meer een omschrijving van een **veel/meest voorkomende, verwachte interactiereeks**. Als er meerdere behoorlijk verschillende normale interactiereeksen zijn kun je eventueel meerdere scenario blokken toevoegen (Scenario 1, Scenario 2)
- **Preconditie** is alleen nodig als er niet-triviale randvoorwaarden nodig zijn om de usecase te kunnen starten. Voor "Het apparaat staat aan", "de batterij is niet leeg", "het apparaat is niet kapot" en dat soort trivialiteiten hoeft geen preconditie sectie te worden toegevoegd.
- **Postconditie** is alleen nodig als er een niet-triviaal resultaat is met niet-triviale betekenis of gevolgen. Bij de usecase "instellen" hoeft dus geen postconditie met "het apparaat is ingesteld").
- **Uitzondering** (en) is alleen nodig als er iets is wat het normale, gewenste gedrag afbreekt.
- **Invariant** (en) een regel die tijdens de usecase bij voortduring geldig is.
  In het bovenstaande voorbeeld zou dat bijvoorbeeld zoiets kunnen zijn:
  "Na elke keer dat de wolfunit een bericht verstuurt, laat hij dat merken door gedurende 0.3 seconde een groen lichtsignaal af te geven."
  Gebruik invarianten **alleen** om de scenario beschrijving leesbaarder te houden. Niet om een compleet andere usecase in te zetten.

SysML Activity diagram

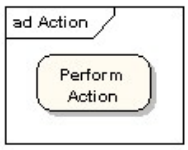
Het SysML activity diagram bevat alle mogelijkheden van het UML activity diagram, plus nog wat extras.

Activity

Een activity diagram beschrijft een "activity": een gedragsreeks. Een activity kan bestaan uit acties, control flows en meer.

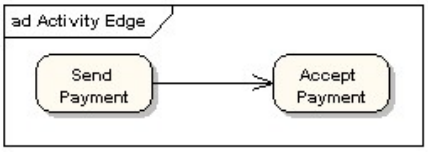
Action Node

De action node representeert een stap binnen een activiteit. Acties worden aangegeven met afreronde rechthoeken.



Control flow

De control flow geeft de volgorde aan van de ene actie naar de volgende. Ze wordt weergegeven met een pijl.

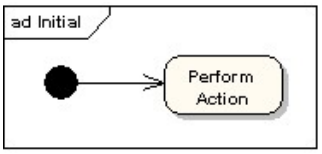


Control tokens

Bij control flows spreek je over het stromen van "**control tokens**".
Als de uitvoering van "send payment" overgaat naar "accept payment", wordt ook wel gezegd dat het "control token" vanuit "send payment" bij "accept payment" aankomt.
Vergelijk het met een stokje dat bij een **estafette** wordt doorgegeven.

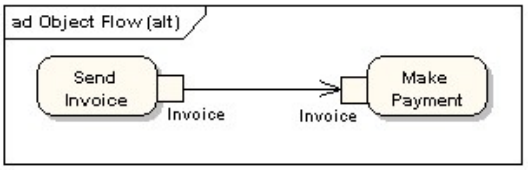
Initial node

Met een initial node wordt het startpunt van een activiteit weergegeven.



Objects en object flows

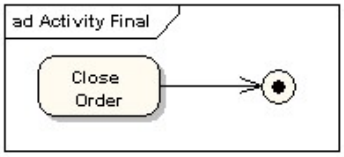
Een object flow is een pad waarlangs objecten of data kunnen passeren. Het wordt weergegeven met een pijl met vierkantjes.
Bij een object flow moet op tenminste een van zijn uiteinden een objectType zijn weergegeven.



Final nodes

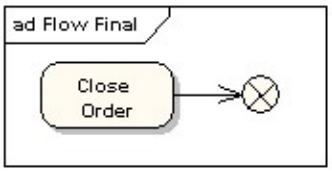
Er zijn twee soorten final nodes.

Acivity final



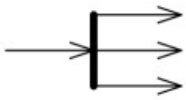
Het bereiken van deze node stopt ook alle overige flows / threads binnen de activity.

Flow final



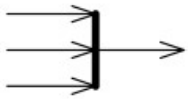
Het bereiken van deze node stopt alleen de betreffende flow binnen de activity. Andere flows die binnen de activity actief zijn, gaan wel gewoon door.

Fork



Een fork is een control node met een ingaande, en meerder uitgaande flows (threads).

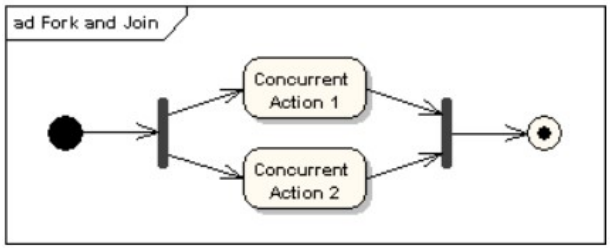
Join



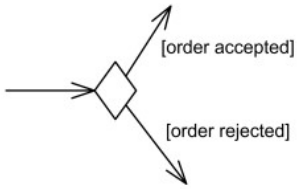
Bij een join node komen meerdere flows samen. Pas als van alle binnenkomende flows de controltokens zijn binnengekomen, vertrekt er een controltoken in de uitgaande flow (dwz: hij wacht tot alle binnenkomende threads afgerond zijn voor hij verder gaat).

Typische combinatie van Fork en Join nodes

Typisch worden Fork en Join nodes gecombineerd. Zodra er even iets parallel moet gebeuren, wordt er geforkt. Vervolgens wordt er weer gejoind.

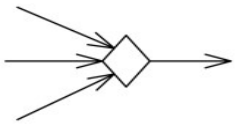


Decision Nodes



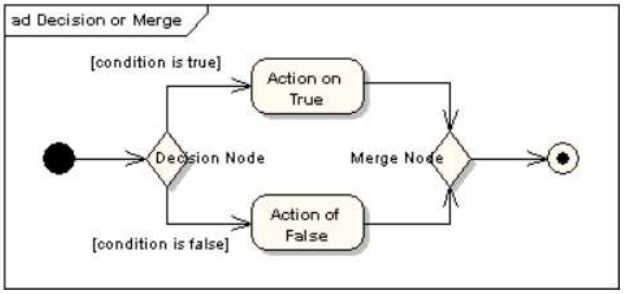
Een decision node is als een weg met meerdere afslagen. Het control token kiest de afslag waarvoor geldt dat de guard (de boolean expressie tussen []) waar is.

Merge node



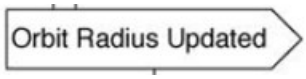
Een merge node is als een punt waar meerdere wegen bij mekaar komen. Zodra een token van een van de binnenkomende paden aankomt, reist het verder via de uitgaande flow.

Net als bij Fork en Join nodes komen Decision en Merge nodes vaak voor in paartjes.



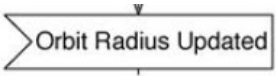
In het bovenstaande voorbeeld moet er even iets anders gedaan worden, afhankelijk van welke guard waar is. Vervolgens wordt er weer op dezelfde manier verder gegaan.

Bericht versturen



Via een rechthoek met een driehoekje eraan wordt weergeven dat een **bericht** naar een andere thread wordt **verstuurd**.

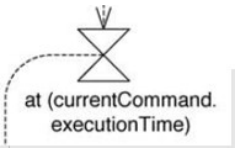
Bericht ontvangen



Via een rechthoek met een driehoek-hapje er uit wordt weergeven dat een **bericht** vanuit een andere thread wordt **ontvangen**.

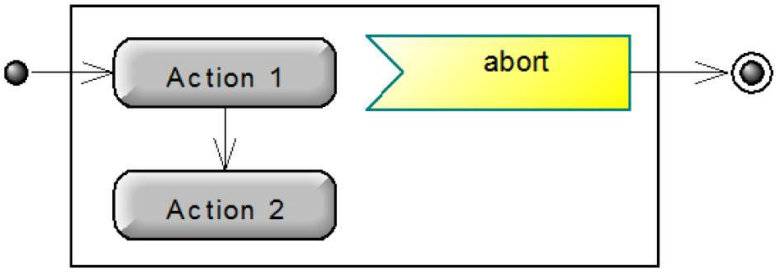
Timer

Een timer-actie kan met een zandloper-symbool worden weergegeven.



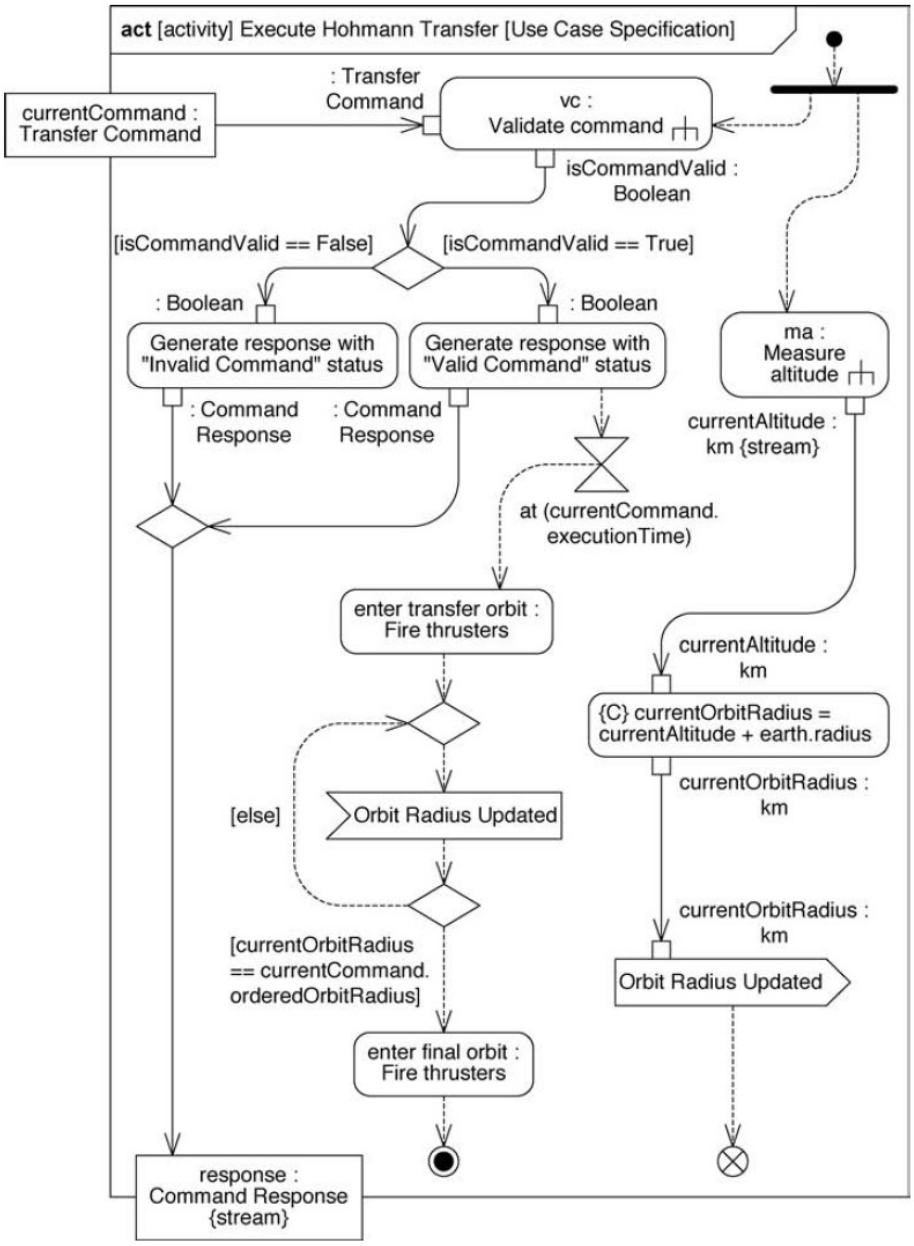
Interruptable Activity Region

Middels een rechthoek kun je een zogenaamde "Interruptable Activity Region" (IAR) afbakenen. Onderstaand is een voorbeeld te zien. Na het uitvoerenvan Action 1 wordt Action 2 uitgevoerd. Beide "actions" kunnen langer duren. Echter, op het moment dat er een abort bericht binnenkomt, worden alle lopende activiteiten binnen de IAR direct afgebroken. Een IAR is ideaal om bijvoorbeeld uitzonderingen op normaal gedrag op te vangen. Bijvoorbeeld wat er gebeurt als er aan een noodrem wordt getrokken.



Voorbeeld van een SysML activity diagram

Onderstaand is een uitgebreid voorbeeld van een SysML activity diagram weergegeven.



Discussie van het bovenstaande SysML activity diagram:

- Aan de randen van het activity diagram zie je **overlappende rechthoeken** getekend. Dat zijn "pinnen", **aansluitpunten** met de **buitenwereld**. Meestal zijn ze voorzien van een **token-specificatie**. Linksboven zien we bijvoorbeeld een pin waar een token "currentCommand" met als type "Transfer Command" binnenkomt. Dat token reist door naar de action node "vc".
- In sommige afgeronde rechthoeken zie je het symbool van een **omgekeerde drietand**: ⏄.
  Dat betekent dat het niet om een eenvoudige action gaat, maar om een complete **activity**.
  In dit voorbeeld is er bijvoorbeeld een activity "vc" van het **type** "Validate Command".

Dat betekent dat er **elders** een **activitydiagram** moet zijn dat "Validate Command" heet.

- Bij **object Flows** hoeft alleen maar aan een zijde het **type** object gespecificeerd te zijn (dat is het geval bij ":Command Response")
  Maar je mag het ook aan beide kanten doen, en je mag het **lokale object-token** ook een **naam** geven (zoals in dit geval "currentOrbitRadius" en "currentAltitude", twee object tokens van het type "km").
  Het is ook nog mogelijk om achter het object type de **multiplicity** tussen [] weer te geven. In het bovenstaande diagram is daar geen gebruik van gemaakt. De multiplicity is dan impliciet [1]. Uit een "Measure postition" activity zou bijvoorbeeld een object flow kunnen vertrekken met vecPosition:km[3]. Dat geeft dan aan dat vecPosition een flow object is dat uit 3 delen van type km bestaat.
- Een **actie** kan ook uit een stukje **programmeercode** bestaan. In dat geval gaat de code vooraf door de **naam van de programmeertaal** tussen **{}**. In dit voorbeeld is er code uit de taal C gebruikt voor het bepalen van de "currentOrbitRadius".
- Bij de output van de activity ma:MeasureAltitude staat **{stream}**.
  Dat betekent dat de betreffende activity een "**streaming activity**" is. In plaats van dat het na het binnenkomen van een token eenmalig een nieuw token verder stuurt, **blijft** het periodiek **nieuwe tokens creeren**. In dit geval blijft het periodiek de hoogte meten en doorsturen.
  NB: dit is dus een alternatieve manier om aan te geven dat er meerdere dingen gelijktijdig kunnen plaatsvinden.
- Bij de response:CommandResponse **output pin** staat ook een **{stream}** specificatie. Daar kun je aan zien dat het verschijnen van een response token aan de output niet het einde betekent van de activity. (zodra er nieuwe input via de input pin "currentCommand:TransferCommand" binnenkomt, wordt deze opnieuw verwerkt en resulteert deze weer in een nieuwe response.

SysML Requirements

In eerdere hoofdstukken zijn we al de drie typen requirements tegengekomen: Functional, Non-Functional en Constraints.

Relaties

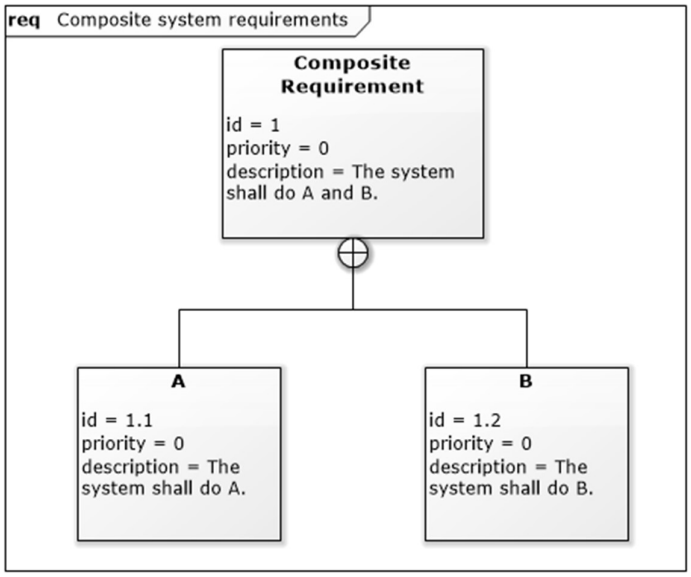
Binnen SysML kunnen die requirements aan elkaar en met andere onderdelen van het ontwerp gekoppeld worden via de volgende zes relaties:

1. Containment
2. Derive
3. Refine
4. Satisfy
5. Verify
6. Trace

Containment-relatie

De containment (of **composite**) relatie (met **plus-teken**) wordt gebruikt om aan te geven dat een **samengestelde** requirement bestaat uit een of meerdere onderliggende requirements.

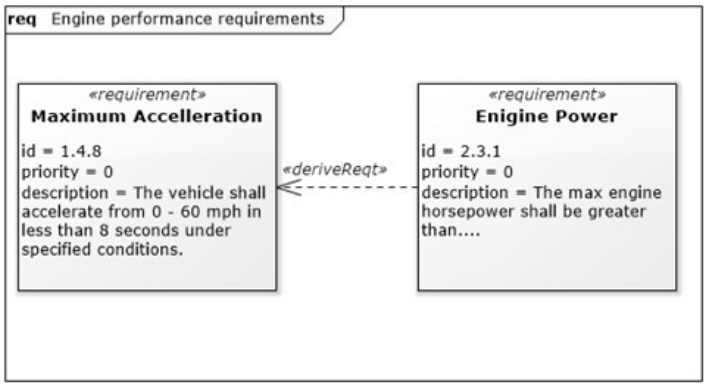
Met de containment relatie kan een **hiërarchie** van requirements worden weergegeven. In onderstaande diagram is een voorbeeld gegeven.



Derive-relatie

Afgeleide requirements **geven invulling** aan requirements met een hoger niveau van **abstractie**.

Onderstaand is een requirements diagram als voorbeeld gegeven.



In dit geval geeft de requirement “Engine Power” **een concretere invulling** aan de requirement “Maximum Acceleration”.

NB: net als bij de <<extend>> relatie in use case diagrammen kan de **pijlrichting** misschien wat contra-intuitief overkomen. Eerst wordt immers de requirement “Maximum Accelleration” bedacht. Later wordt de requirement “Engine Power” daarvan afgeleid om er concretere invulling aan te geven.

Refine-relatie

De **refine** relatie is vergelijkbaar met de derive relatie. Het verschil is echter dat een **ander type model element conretere invulling** (verfijning) geeft aan de requirement.

Bijvoorbeeld in het onderstaande voorbeeld geeft de use case “Drive Vehicle” een concretere invulling aan de requirement “Maximum Accelleration”.

TODO: add example

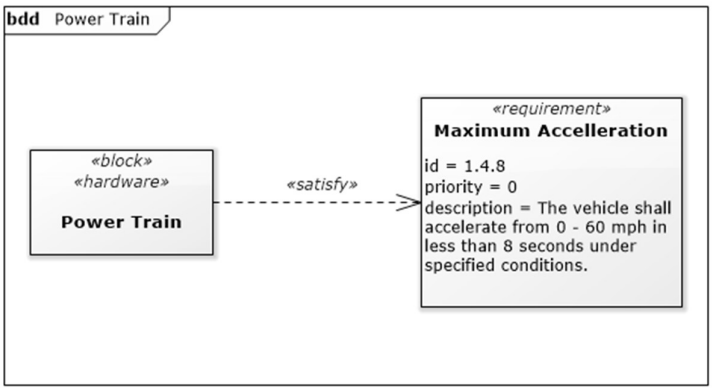
NB: de pijlrichting van de <<refine>> relatie is wel weer logisch.

Satisfy-relatie

De **satisfy** relatie beschrijft hoe een **ontwerp** of **systeem-concept** voldoet aan een of meerdere requirements.

Een system engineer kan weergegeven dat de ontwerp-elementen **bedoeld** zijn om aan de requirement te voldoen.

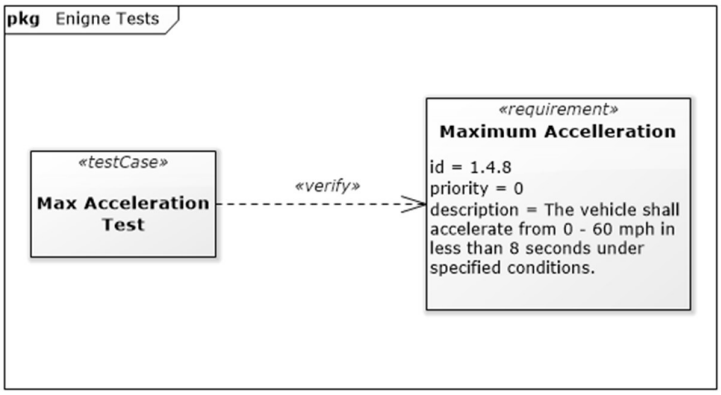
In het onderstaande voorbeeld zorgt het hardware block “Power Train” ervoor dat aan de “Maximum Accelleration” requirement wordt voldaan.



Verify-relatie

De **verify**-relatie bepaalt hoe een **testcase** of ander model element **controleert** of aan een **requirement** is voldaan.

In het onderstaande voorbeeld wordt met de testCase “Max Acceleration Test” geverifieerd of aan de requirement “Maximum Accelleration” is voldaan.



Trace relatie

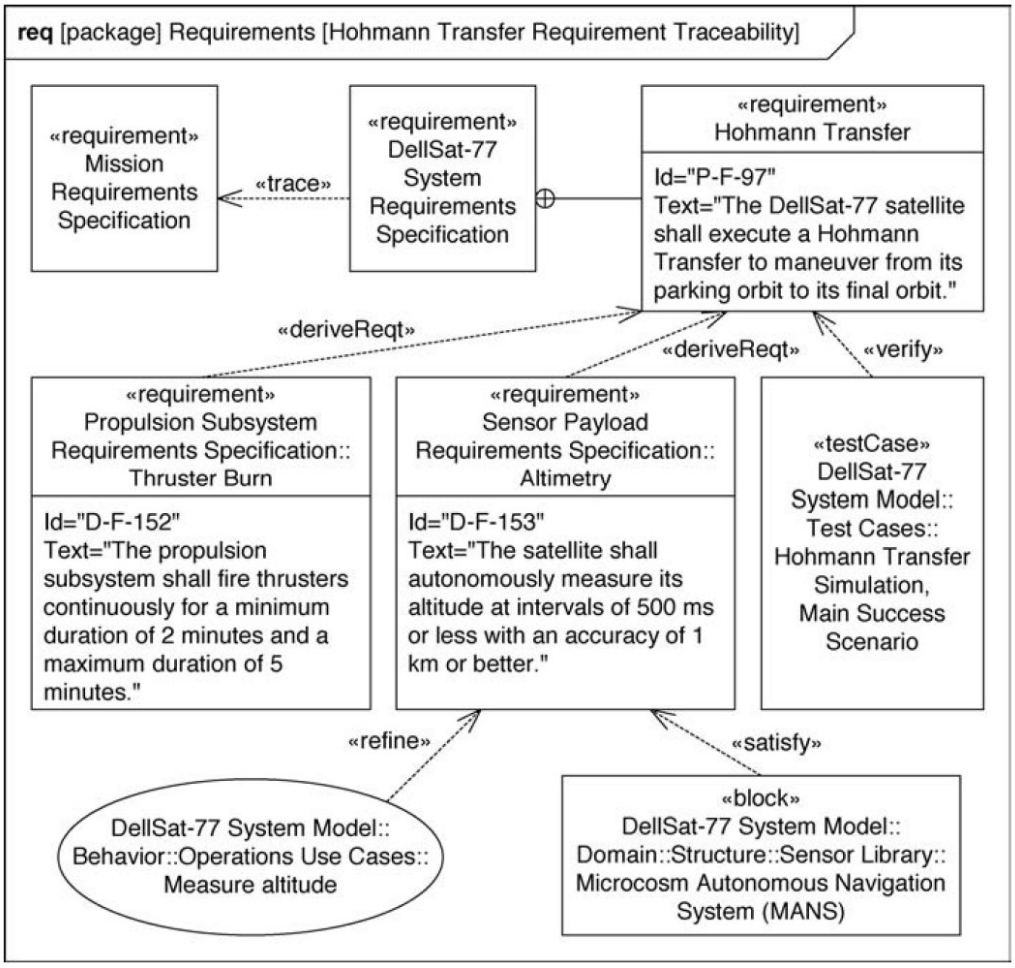
Trace-relaties zijn "algemene" relaties. Je kunt ze bijvoorbeeld gebruiken als een van de andere typen relaties niet geschikt lijkt.

Het onderstaande voorbeeld laat zien dat er een verband is tussen de requirement "Maximum Accelleration" en "Market Analysis". Vermoedelijk wordt bedoeld dat de gekozen waarden in de "Maximum Accelleration" requirement zijn gebaseerd op een markt-analyse.

SysML Requirements Diagram

Het SysML Requirements diagram is bedoeld om een overzicht te geen van requirements, en te laten zien op welke manier ze van elkaar afhangen.

Onderstaand is een voorbeeld van zo'n diagram weergegeven.



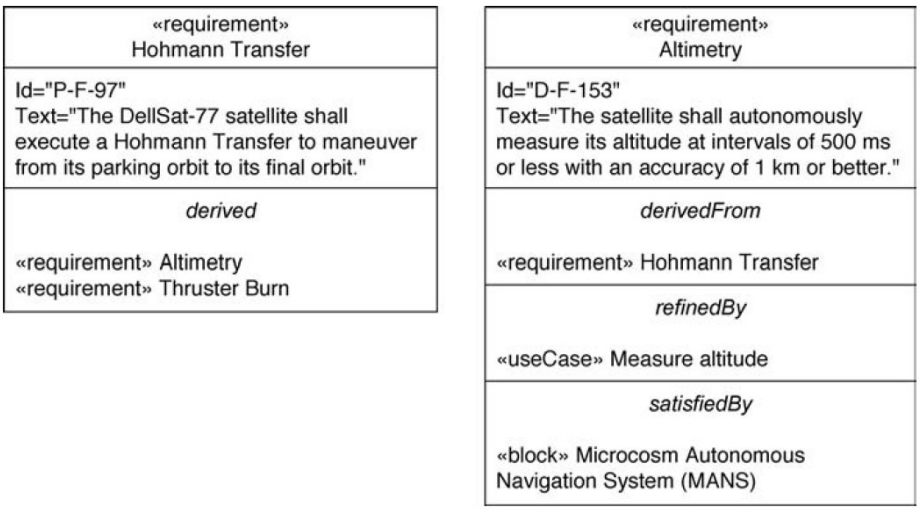
In de SysML header wordt middels de diagram type specifier "**req**" aangegeven dat het om een requirements diagram gaat.

Kijk voor jezelf het diagram eens door, en test of je nog scherp hebt wat de betekenis van elk van de relaties is.

Compartment notation

Ook bij requirement diagrammen kunnen we er voor kiezen om **relaties** met behulp van **compartimenten** in plaats van met behulp van pijlen weer te geven.

Het onderstaande voorbeeld illustreert dat.



Wat in het **derived** compartiment van "Hohmann Transfers" staat, zijn requirements die daarvan zijn afgeleid. Als dat met <<derive>> pijlen zou zijn weergegeven, zouden ze bij de "niet-pijpunt-kant" staan (zie het er aan voorafgaande diagram).

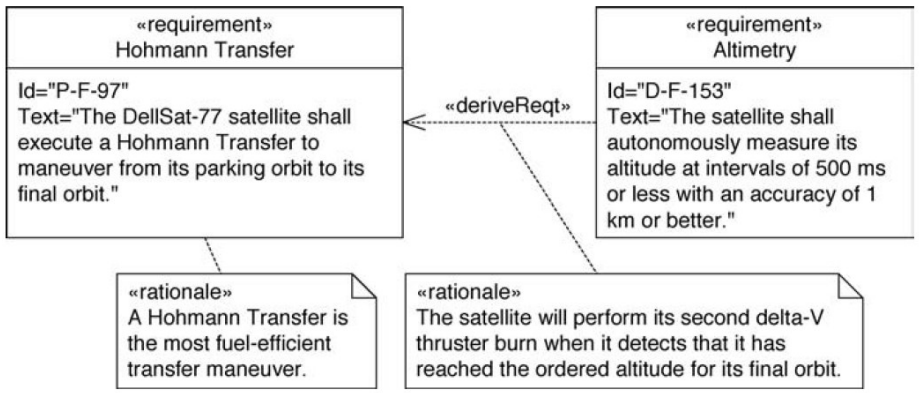
Wat er in het **derived from** compartiment van "Altimetry" staat, zou juist bij de "pijl-punt-kant" van de bijbehorende <<derive>> pijlen staan.

Kortom, als een requirement ergens in een derived compartiment staat, **moet** die requirement zelf een derivedFrom compartiment hebben die terugverwijst (een relatie-pijl heeft immers zowel een begin als een eind).

Rationales

Je kunt aspecten van SysML diagrammen verduidelijk met "**rationales**". Dat zijn comment-blocks die uitleg geven / een **reden** geven, gekoppeld via een **stippelijntje**.

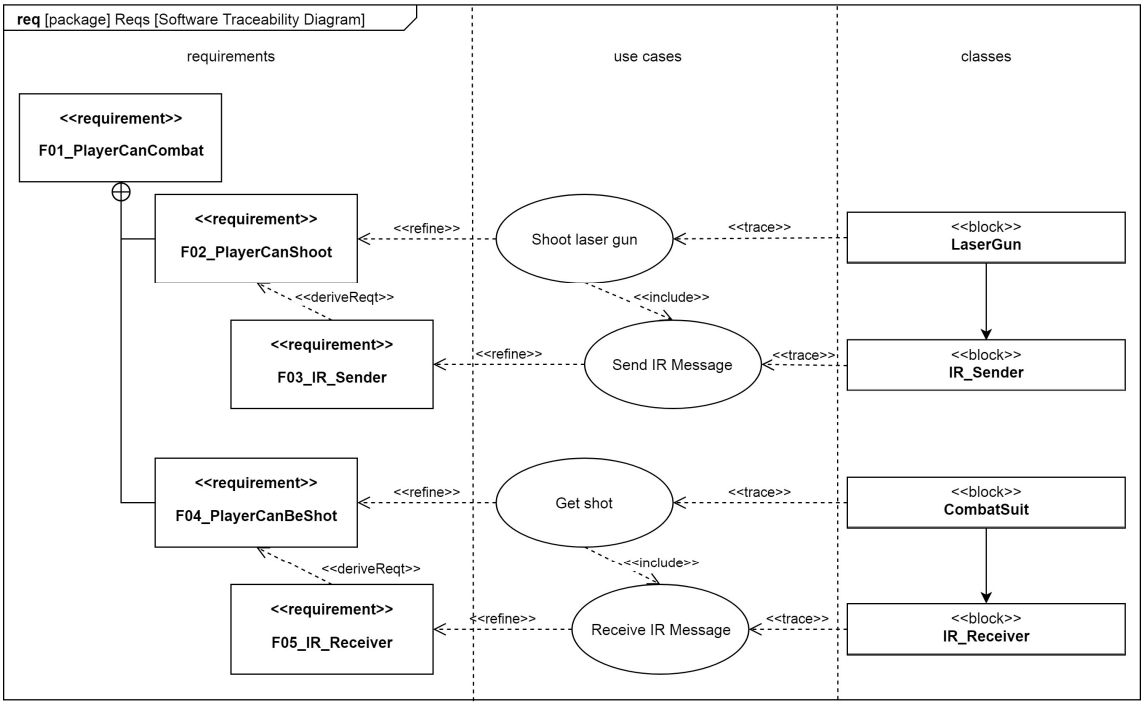
In onderstaande diagram wordt bijvoorbeeld toegelicht waarom is gekozen voor de Hohman Transfer, en waarom de Altimetry is afgeleid van de Hohmann Transfer.



Traceability diagram

Een speciaal geval van een requirements diagram is een traceability diagram. Daarin kun je de ontwerpflow van requirements naar implementatie volgen (tracen).

Onderstaand is een voorbeeld ervan weergegeven.



Discussie aan de hand van het bovenstaande diagram:

- In dit Traceability diagram worden de **verbanden** tussen **requirements**, **usecases** en de bijbehorende **software blocks** weergegeven.
  (software blocks zijn een uitgebreide SysML variant van wat je al kent als “classes”)
- Voor de duidelijkheid zijn de domeinen van elkaar gescheiden met **verticale stippelijnen**.

Wat is het nut van een Traceability diagram?

Het helpt je om **overzicht** te krijgen, en om makkelijker over je ontwerp te kunnen brainstormen:

- Zijn alle requirements wel **geimplementeerd**?
  (als ze niet met een use case of class verbonden zijn, is dat een teken dat dat niet het geval is)
- Zijn alle use cases wel **geimplementeerd**?
  (als ze niet met een class verbonden zijn, is dat een teken dat dat niet het geval is)

- Zijn alle use cases wel **nodig**?
  Als er geen requirement met een use case wordt gerealiseerd, kun je je afvragen of de usecase niet overbodig is.
- Zijn alle classes wel **nodig**?
  Als er geen use cases of requirements met een block worden gerealiseerd, kun je je afvragen of het niet overbodig is.

Kortom: in een goed Traceability diagram is **alles van links naar rechts met elkaar verbonden**. Als er een verbinding ontbreekt, is dat meestal een indicatie dat er iets nog niet klopt.

Het bovenstaande Traceability diagram heeft in de rechter kolom software blocks (classes).

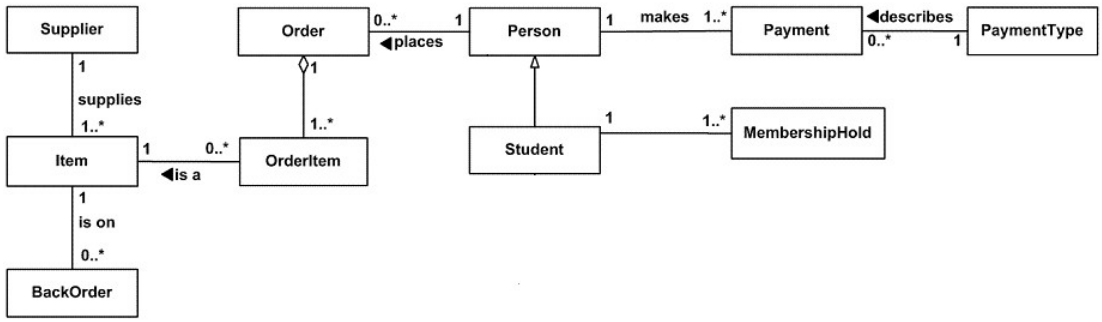
Het is ook mogelijk een Traceability diagram te maken met in de rechter kolom **hardware blocks**, of blocks uit de logical view (komt later).

Informatiemodel

Iets wat ook nuttig kan zijn bij de beschrijving van systemen is een zogenaamd "Informatiemodel".

Een informatiemodel is niets anders dan een klassestructuur die laat zien hoe bepaalde informatie binnen het systeem met elkaar samenhangt.

Onderstaand is er een voorbeeld van weergegeven.



Een publiek bekend informatiemodel kan het koppelen van (deel-) systemen met elkaar vergemakkelijken. Het voorkomt immers dat data opnieuw georganiseerd en geformatteerd moet worden voordat je het overzendt.

Structuur ontwerpen – de Conceptuele fase

Waar komen we vandaan

In eerdere hoofdstukken hebben we stakeholders gevonden, hun doelen bepaald, er requirements uit afgeleid. De volgende stap is om een zodanige structuur van hardware en software te ontwerpen, dat aan die requirements wordt voldaan.

De vraag is nu hoe we kunnen komen tot een model van een product dat aan de specaties voldoet.

Volgende doel: een "Logical View"

Een eerste stap daarin is het ontwikkelen van een "High level Objectmodel". In dit vak noemen we dat ook wel een "Logical View". Dat lijkt op het objectmodel waarmee je al bekend bent uit de "Design Like a Robot" methode van het vak concurrent system modelling. Een **object** in de logical view hoeft echter niet per se een **software object** te zijn. Het **mag ook** een **hardware** object zijn, **of** een object van **mixed hardware en software**.

Dankzij die vrijheid kun je makkelijker creatief brainstormen over de structuur van wat je wilt ontwikkelen.

Concepten ontwikkelen

Bon, je hebt de requirements, usecase diagrammen en eventueel daarvan afgeleide activity diagrammen klaar liggen. Hoe kom je op ideeen om er een structuur voor te maken?

Voor het ontwikkelen van goede concepten kun je (eventueel achtereenvolgens) gebruik maken van de volgende methoden:

1. Mindmappen
2. Morfologische analyse
3. Design Like a Robot Object Model ontwerp (voor SysML)
4. Decompositie

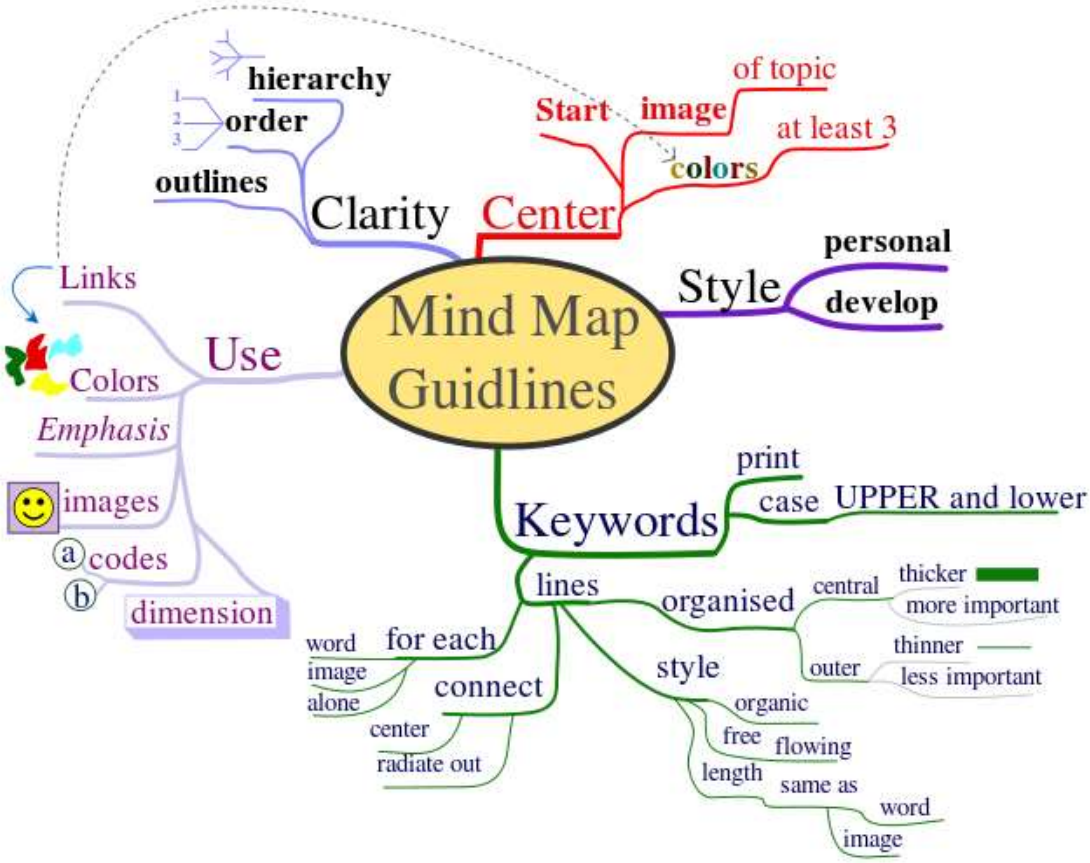
Mindmappen

Mindmappen is een snelle en overzichtelijke manier om structuur te geven aan je gedachten. Het stelt je in staat om niet alleen lineair, maar ook associatief te denken.

Mindmappen stap voor stap:

1. Schrijf of nog beter teken de **probleemstelling** in het **midden** van een A4-tje.
2. Schrijf alle **associaties** op, gebruik hierbij verschillende **kleuren**.
3. **Associeer** vervolgens weer **verder** op deze **sleutelwoorden**.
4. Geeft mindmaps een **boomstructuur** door stevige **takken** en dunne **twijgjes**.

5. **Vervang** zoveel mogelijk **woorden** door **tekeningen**.
6. Kijk of je de mindmap kan verbeteren door een **structuur** aan te brengen, breng gerust **dwarsverbanden** aan.

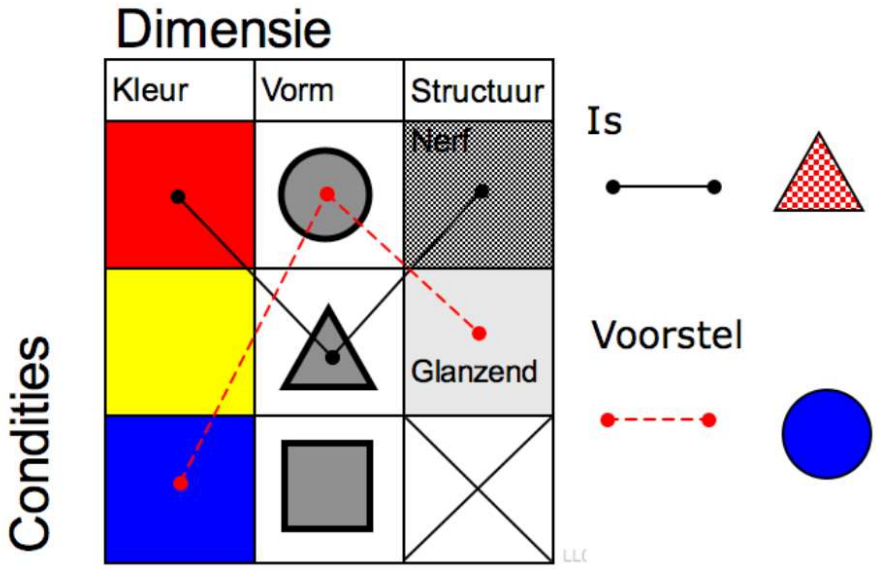


Morfologische analyse

We kunnen met een morfologische analyse verschillende ontwerp keuzes tegen mekaar afwegen. Willen we over Mars voortbewegen door met wielen te rijden? Of door sprongen uit te voeren? In combinatie met een kussen onder het voertuig? Of met een valbeugel?

In een **Morfologische Matrix** kunnen we een overzicht verkrijgen van de verschillende keuzemogelijkheden:

Morfologische Matrix



Door er paden door te trekken kunnen we nadenken over wat verschillende combinaties voor verschillende producten kan opleveren. Misschien zijn er wel meerdere combinaties gunstig, afhankelijk van welke doelgroepen je wilt bereiken.

DLAR_Logical

(Een DLAR variant voor het maken van de Logical View)

Een variant van de "Design Like a Robot" methode van het vak Concurrent Systems Modelling is een handige manier om te komen tot (een deel van-) de Logical View:

- Maak een **object** aan voor **elke use case**
- Verwerk de **use case beschrijvingen** zin voor zin:
  - Lees een **zin**.
  - Identificeer de "**dingen**" in die zin.
  - Maak voor elk van die "**dingen**" een **object** aan.
    Maak je nog geen zorgen over of het object **hardware**, **software** of een combinatie van beide representeert.
  - Voeg het object toe aan het **object model**.
  - Ga na wat de **relatie** is tussen de dingen.
  - Geef die relatie weer **langs** een **pijl**.

Een relatie hoeft niet per om het versturen van berichten te gaan. Het kan ook iets **fysieks** zijn.
I.t.t. bij DLAR, noteer je bij DLAR_Logical niet elk bericht, maar noteer je hooguit het soort berichten dat wordt verstuurd. De logical view is immers een high-level ruw concept ontwerp.

Voorbeeld: Een cameramodule is via een pijl verbonden met een imageprocessormodule. Naast de pijl staat iets als "stuurt beelden". (en niet: ReportNewFrame(frameHeader,frameBody,timeStamp, etc.))

Goed alternatief: als er een **activity diagram** van de use case bestaat, kun je de bovenstaande stappen uitvoeren door een voor een langs de elementen van het activity diagram te lopen.

Decompositie

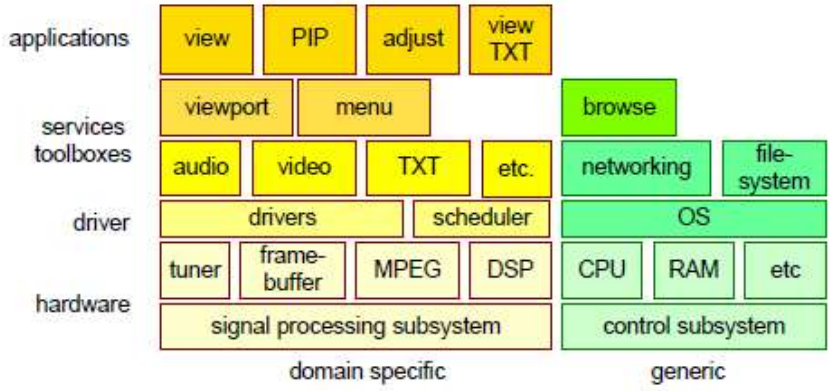
Vaak zijn de elementen in de structuur die je vindt nog te complex, en is nog niet duidelijk hoe die gerealiseerd kunnen worden. Je kunt het ontwerpen van de betreffende elementen wederom met een ronde mindmappen, morfologische analyse en DLAR-ontwerp te lijf gaan.

Je kunt ook decompositie toepassen. Decompositie is het **opbreken** van een groot probleem **in** kleine **subproblemen**, en dat net zo lang blijven doen tot de subproblemen zo klein zijn dat de oplossing niet moelijk meer is.

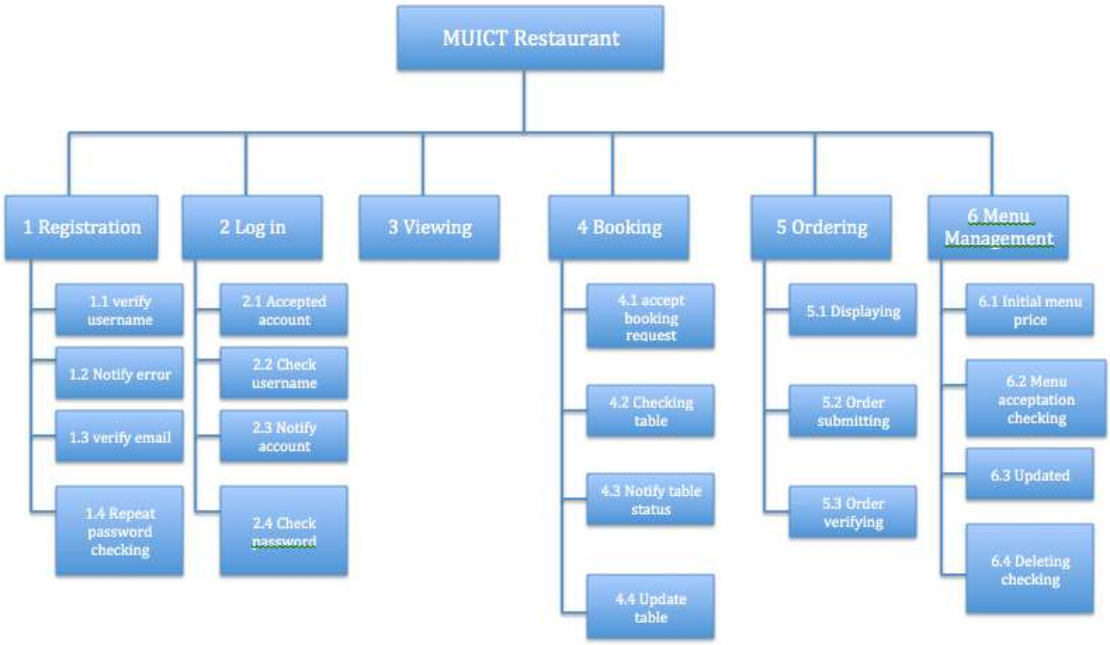
Er zijn twee veel toegepaste decomposities:

1. **Constructie decompositie**
   Het opbreken van het systeem (ook software) in **losse onderdelen**.
2. **Functionele decompositie**
   Het opbreken van de functies van het systeem in **deelfuncties**.
   Dat geeft typisch een **hierarchische** structuur.

Voorbeeld van constructie decompositie van een TV:



Voorbeeld van een functionele decompositie van een restaurant UI:



Naamgeving

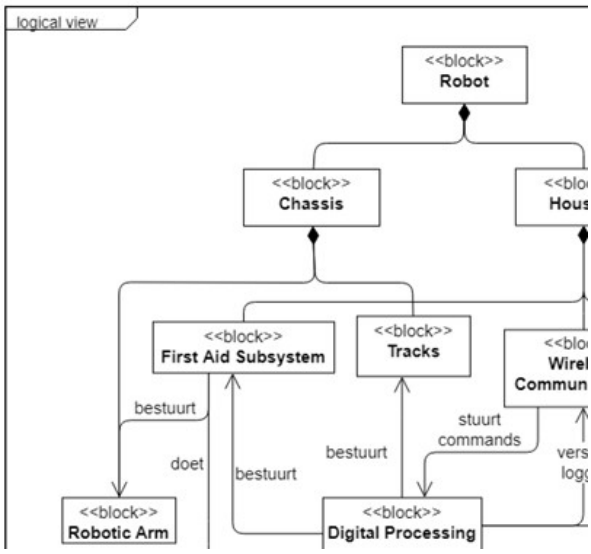
- Laat de functienamen indien mogelijk bestaan uit werkwoord + lijdend voorwerp, zoals "Verifieer username". Het bovenstaande voorbeeld had op dat punt wat duidelijker gekund. Bijvoorbeeld door in plaats van "Registration" te kiezen voor "Klant registeren". In plaats van "Ordering" "Maaltijd bestellen", etc.

De Logical View

Met behulp van de vier voorgaande methoden kunnen we een ruw concept-ontwerp van hoe het systeem er grofweg uit moet komen te zien. Dat noemen we de "Logical View".

De Logical View is een BDD (Block Definition Diagram) met alleen **block namen** en **relaties** ertussen. Het laat de **decompositie** zien van het systeem in hardware/software/functies/modules.

Onderstaand is een voorbeeld weergegeven van een stukje logical view van studenten van een eerder jaar.



Discussie van bovenstaande Logical View

- In de logical view is mooi een hierarchische decompositie weergegeven met **compositie-relaties**.
- De objecten kunnen **software, hardware of een mix** zijn.
  Bijvoorbeeld de Robot Arm – daar zitten misschien servo-motoren in, vastgeschroefd aan metalen beugels, verbonden met een microcontroller waar software op draait die binnenkomende instructies voor de positionering van de arm verwerkt.
- Als het om een aansturende relatie gaat, kun je "**bestuurt**" noteren.
  NB: op het hoge niveau van een logical view is het nog **niet** belangrijk om **detail in berichten** aan te brengen. De "Digital Processing" Module "bestuurt" de Tracks van onze rescue robot.
  Welke berichten er allemaal met die besturing samenhangen, daar gaan we in de logical view nog niet op in.
- In dit voorbeeld zie je geen stroom van fysieke objecten. Als er kisten bananen van het ene object naar het andere gaan, kun je "**stuurt kisten bananen**" noteren.
- Persoonlijk zou ik in dit diagram de eindgebruiker (brandweerman) en de persoon die gered wordt ook hebben weergegeven, als **actor**-poppetjes.

Process View

Je kunt ook door de "proces" bril naar een project kijken. Wat je dan ziet, noemen we de "Process View". De Process View van een project kun je weergeven met de volgende diagrammen:

1. Activity Diagrammen
2. Een Subsystem Process Table
3. State Transition Diagrammen
4. Sequence diagrammen

Voor het architectuurdoment in dit vak zijn alleen activity diagrammen en subsystem proces tabel vereist. Met het toevoegen van de andere twee typen diagrammen kunnen bonus-punten worden gescored.

SysML State Transition Diagrammen en SysML Sequence diagrammen komen in deze reader niet aan bod. Als je die diagrammen gaat maken, lees dan de bijbehorende hoofdstukken van Delligatti ter voorbereiding goed door.

Het SysML Activity diagram is eerder in deze reader aan bod gekomen. In de volgende paragraaf komt de Subsystem Process Table aan bod.

Subsytem Process Table

Een Subsystem Process Table beschrijft ruwweg wat **parallel** werkende **SubSystemen** doen. Dat kan in termen van de belangrijkste "**states**" of "**activities**" (kies er een). Daarnaast worden ook de belangrijkste **events** weergegeven waar de processen naar luisteren.

Onderstaand is een voorbeeld van studenten van een voorgaand jaar weergegeven:

7.1 Subsystem proces tabel

Subsystem	*Subsystem proces beschrijving*
Robotic Arm	Aansturen van de robotische arm. Er is obstakel dete voor de armstukken zodat er geen extra letsel bij het ontstaan. *States: Idle, Bereken Doel Assen, Bereken Pad, Stuur Motor Comr* *Events: Input Detected, Obstacle Detected, Doel Assen Behaald*
Track Locomotion	Aansturen van de tracks. De track locomotion subsys zorgen dat de motoren via seriële commando's vanu kan worden aangestuurd. *States: Idle, Rijden, Draaien In Richting* *Events: Input Detected*
Power Control	Regelen van de elektriciteitsvoeding van alle andere Wanneer de robot word aagezet zal elk subsystem vo *States: Leveren Van Elektriciteit* *Events: Turn Off Signal*
Digital Processing	Verantwoordelijk voor digitale communicatie, sensor aansturen van andere subsystems. *Threads: Real-Time Communcations Rx, Real-Time Communicatic* *Sensor Reader, Robotic Arm Controller, Track Controller* *States (Real-Time Communications Rx): Idle, Verstuur Rx Data* *Events (Real-Time Communications Rx): Rx Data Ontvangen* *States (Real-Time Communcations Tx): Idle, Verwerk Tx Data* *Events (Real-Time Communcations Tx): Tx Data Ontvangen* *States (Sensor Reader): Idle, Meet m.b.v. Environmental Sensors,* *Events (Sensor Reader): timer(200ms)* *States (Robotic Arm Controller): Idle, Verstuur Command Naar Ro* *Events (Robotic Arm Controller): Command Ontvangen*

Discussie van het bovenstaande voorbeeld:

- In dit voorbeeld is gekozen om alles in termen van states (toestanden) te beschrijven. Als je een en ander al in activity diagrammen hebt uitgewerkt, is het misschien handiger om alles in termen van activities te beschrijven.

SysML Block Definition Diagram (BDD)

Block Definition Diagrams worden gebruikt om de **structuur** van het systeem mee weer te geven.

Je kunt het gebruiken voor zowel software als hardware. Als je het gebruikt voor software, dan is het een SysML variant van een UML klasse diagram. Dus een klasse diagram met veel extra mogelijkheden tot expressie.

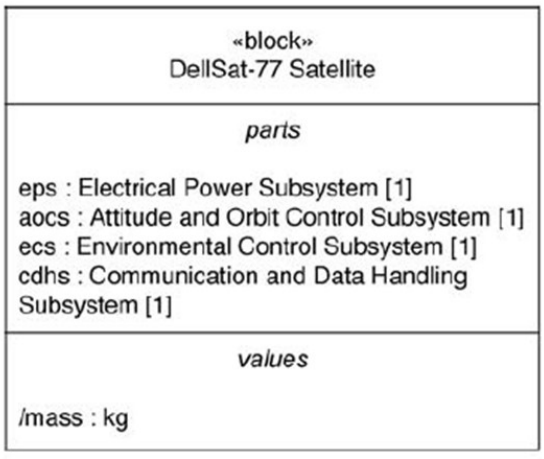
Block

Een BDD bestaat uit "blocks" die via relaties met elkaar verbonden zijn.

Voorbeelden van wat zo'n block zou kunnen representeren, zijn:

- Hardware
- Software
- Data
- Procedure
- Faciliteit
- Persoon

Onderstaand is een voorbeeld van een block weergegeven.



Een block kan opgebouwd zijn uit meerdere compartimenten.

Naam compartiment

Het **bovenste** compartiment is het "Naam Compartiment". Dat bevat het stereotype **<<block>>**, met daaronder de **naam** van het block.

Overige compartimenten

Daarnaast kunnen er optioneel nog andere vakken (compartimenten) zijn, die kenmerken van het block beschrijven. Elk compartiment begint met een **horizontale scheidslijn**, gevolgd door de **naam** van het compartiment. In het bovenstaande voorbeeld heeft het block ook een parts en een values compartiment. In een latere paragraaf komen onder andere die typen compartimenten aan bod.

Model elementen van een BDD

Naast Blocks zijn er nog andere "Model elementen" die in een BDD kunnen voorkomen.

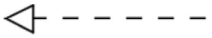
Model elementen van een BDD:

- Blocks
- Value Types
- Constraint Blocks
- Flow specifications
- Interfaces

Deze "Model Elementen" dienen als typen in de overige soorten diagrammen.

Dependency relaties

Daarnaast bevatten BDDs de afhankelijkheidsrelaties tussen deze model elementen. Die zijn dezelfde als bij het klassediagram in het vak Concurrent System Modelling:

- **Compositie**: onderdeel van
  - Met: 
  - **Of**: met een **parts compartment**
- **Reference**: lets "kennen"
  - Met: 
  - **Of**: met een **references compartment**
- **Specialisatie**: Overerven
  - **Van een interface:** 
  - **Overig:** 
  - **Of: via ports**

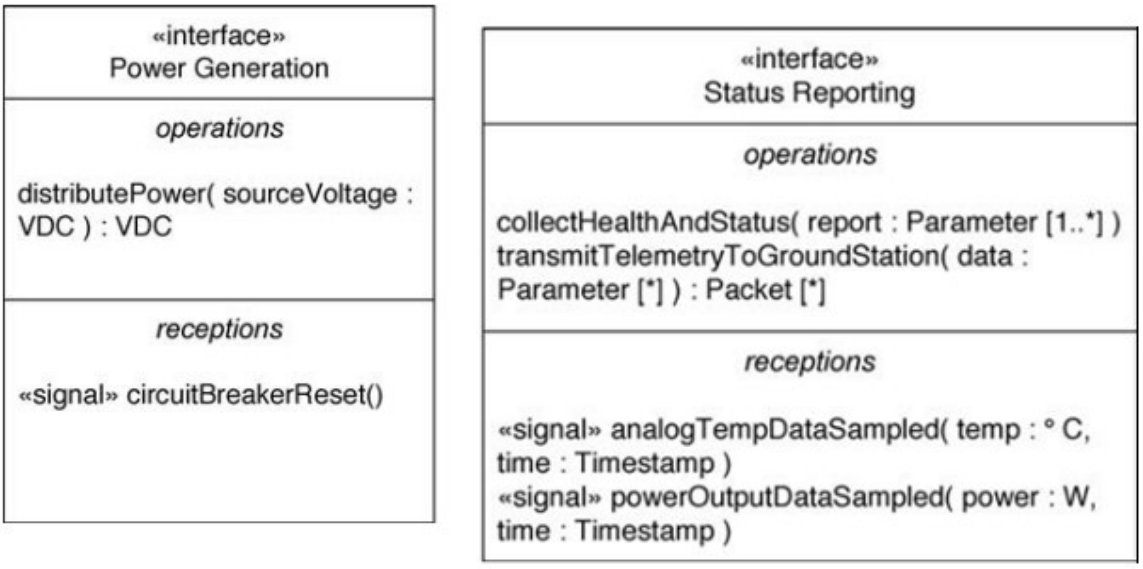
Ports komen verderop aan bod.

BDD – Interface

Een interface is een **gedragscontract**. Het bevat een reeks **operaties** en **attributen** waar clients en providers beide aan moeten voldoen om met elkaar informatie uit te wisselen.

Een BDD-interface is vergelijkbaar met interface klassen in klasse diagrammen, met extra uitdrukkingsmogelijkeden.

Onderstaand zijn twee voorbeelden van BDD-interfaces weergegeven.



Operations Compartment

Het operations compartment geeft weer welke **synchrone operaties** er worden aangeboden. Na aanroep van zo'n synchrone operaties wordt deze direct afgehandeld, en wordt een eventueel resultaat aansluitend geretourneerd. Meestal gaat het om kortdurende, "klaar terwijl u wacht" operaties.

Receptions Compartment

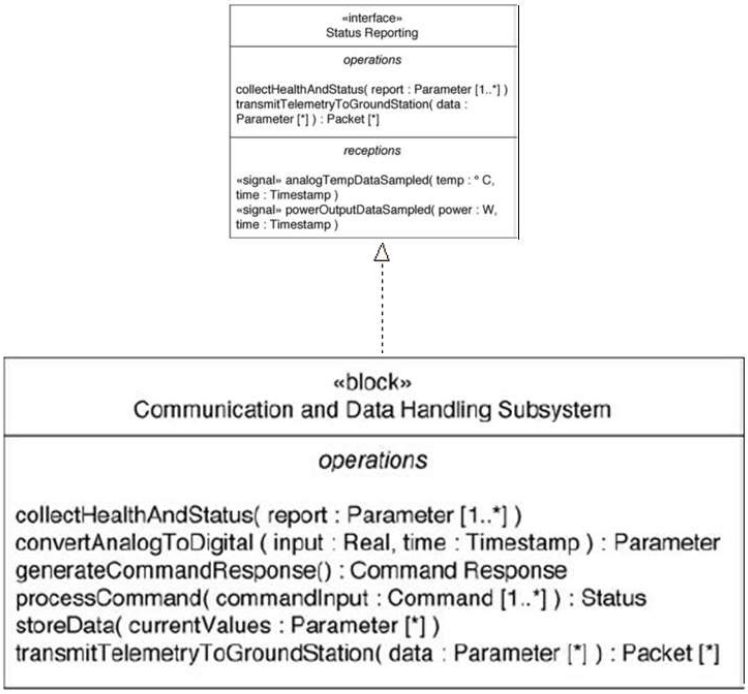
De afhandeling van asynchrone aanroepen (events) wordt geadresseerd in het receptions compartiment. Een reception begint met de aanduiding <<signal>>. De aanroeper gaat verder met andere zaken terwijl het subsytem het "signal event" afhandelt.

Zowel bij operations als bij receptions kunnen er parameters meegegeven worden.

Overerving van een interface

Er kan op verschillende manieren in een BDD worden aangegeven dat een block een interface aanbiedt. Een daarvan is door een **"overerving van een interface" relatie** weer te geven. Zo'n relatie ziet er uit als een leeg driehoekje met een stippelijn. De interface staat altijd aan de kant van het driehoekje.

Het onderstaande voorbeeld geeft weer dat het block "Communication and Data Handling Subsystem" de interface "Status Reporting" aanbiedt.



Ports

Een andere manier om aan te geven dat een block een interface aanbiedt, is door gebruik te maken van **ports**. Een groot voordeel daarvan is, dat je dan ook duidelijk kunt zien welke blocks welke interfaces van welke andere blocks aanroepen.

Ports representeren **interactiepunten**.

Voorbeelden van ports:

- Voorbeelden ports op een **hardwareblock**:
  een HDMI aansluiting, een brandstofverstuiver
- Voorbeelden van ports op een **softwareblock**:
  een message queue, een GUI
- Voorbeelden van ports op een **onderdeel** van een **bedrijfsorganisatie**:
  een website, een mailbox

Soorten ports

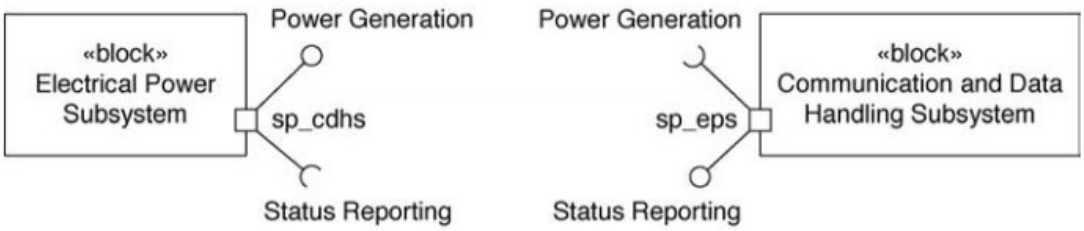
Binnen SysML kun je kiezen uit twee smaken ports:

- Standard Ports
- Flow Ports

Elk van deze port types heeft zijn eigen voordelen. We zullen nu elk van hen behandelen.

Standard Ports

Een standard port is een interactiepunt die de interactie via een of meer interfaces weergeeft.



In het bovenstaande voorbeeld zijn het block "Electrical Power Subsystem" en het block "Communication and Data Handling Subsytem" aan elkaar gekoppeld via standard ports.

Een standard port wordt weergegeven met een vierkantje op de rand van een block.
Aan dat vierkantje kunnen een of meerdere stokjes met bolletjes of "holletjes" zitten.

Een **bolletje** representeert een **interface** die het block **aanbiedt** (dus waarvan het geerfd heeft).
In het bovenstaande voorbeeld biedt het block "Electrical Power Subsystem" de interface "Power Generation" aan (die interface is op een eerdere pagina in een voorbeeld afgebeeld). Dat betekent dat dat block de operatie "distribute power" kan uitvoeren.

Een **holletje** representeert dat het block **gebruik maakt** van een ander block dat een bepaalde interface aanbiedt. In het bovenstaande voorbeeld heeft de standard port van het block "Communication and Data Handling Subsystem" een holletje met de naam "Power Generation". Dat betekent dat "Communication and Data Handling Subsystem praat met de interface "Power Generation" van de standard port van een ander block. In dit geval de standard port van het "Electrical Power Subsystem".

Slimme naamgeving van standard ports

In het bovenstaande voorbeeld is de naam van de standard port die aan het "Electrical Power Subsystem" zit "sp_cdhs". Door die naamgeving zie je dat het een standard port is die is verbonden met een andere **s**tandard **p**ort op het block "**C**ommunication and **D**ata **H**andling **S**ubsystem".

Een standard port is dus als een soort "stekker contact" voor een groep draden.

De standard port samengevat

Kortom, de standard port is een fantastisch instrument om mee aan te geven wie welke interface aanbiedt, en wie er gebruik van maakt.

Flow Ports

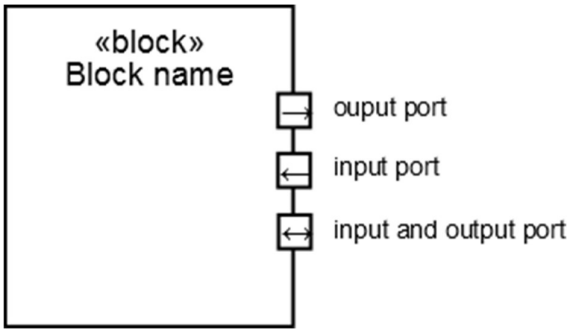
Interfaces kunnen zowel getters als setters aanbieden. Bij standard ports is het niet makkelijk te zien in welke richting de data of de objecten stromen.

Flow ports benadrukken daarentegen wat er stroomt, en in welke richting.

Er zijn twee typen flow ports:

- **Atomic** flow ports:
  Via deze flow ports stroomt maar 1 type data, materie of energie.
- **Non-atomic** flow ports:
  In deze flow ports worden meerdere typen stromen gecombineerd.
  (analoog aan een stekker-contact van een kabel bestaande uit meerdere draden)

De richting van de flows wordt aangegeven met een pijl of dubbele pijl in een vierkantje.



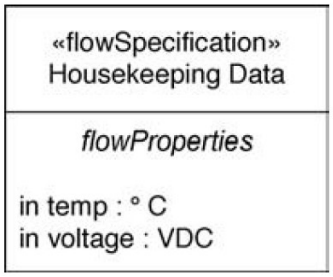
In de bovenstaande afbeelding is een block met drie ports weergegeven:

- Een output port
  Daar stroomt alleen maar uit.
- Een input port
  Daar stroomt alleen maar in.
- Een input-output port
  Daar **kan** zowel in als uit stromen.

De voorgaande afbeelding is niet compleet. Naast de port hoort namelijk de naam te staan van wat er flowt.

In het onderstaande voorbeeld worden het block “Modulator” en “Transmitter” met elkaar verbonden via een output flow port op het ene block en een input flow port op het andere blok. Wat er flowt, is data van het type “Radio Frequency Cycle”.

In het onderstaande voorbeeld is gebruik gemaakt van een input-output port.

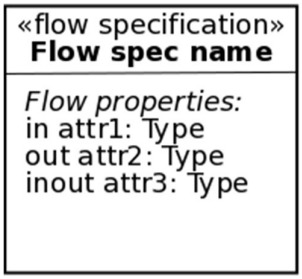


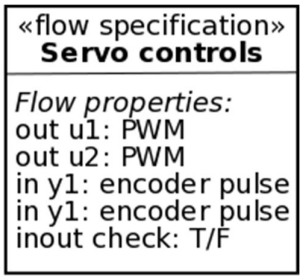
Bij gebruik van een input-output port moet worden aangeduid wat er stroomt met een flow-specification. Dat is een record van data en voor elk type data de stroomrichting.

Bij het linker object zie je de voorafgaand aan de flow specification een tilde (**~**).
De betekenis daarvan is dat op die poort de "geconjugeerde" (alles omgekeerde) flow specification van toepassing is. Ofwel, temp data stroomt **uit** de linker poort, evenals voltage data.

In het bovenstaande voorbeeld bevat de flow specification alleen maar data van het type "in". In plaats van een flow-port van het type input-output had hetzelfde dus ook (duidelijker) weergegeven kunnen worden met bij het ene block een output-port en bij het andere block een input port.

Onderstaand zijn nog twee voorbeelden van flow-specifications weergegeven:





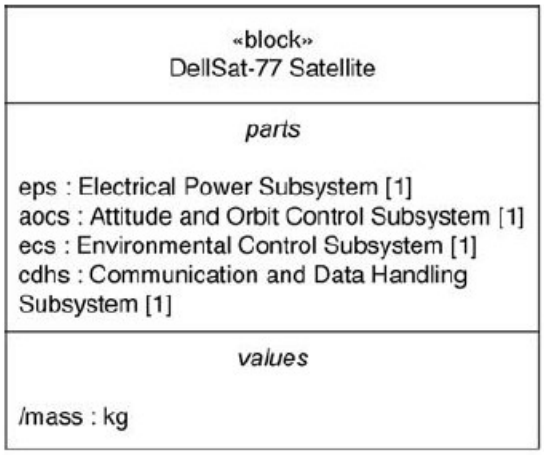
(NB: mooier/consistenter was geweest: flow properties (zonder :) bovenin de partitie)

Verbindingslijntjes

Zoals bovenstaande voorbeelden laten zien, kunnen verbindingen tussen ports weergegeven worden zonder verbindingslijntjes tussen de ports te tekenen. Dat mag echter wel. Het kan duidelijker zijn als je de verbindingslijntjes tussen de ports tekent. Als de ports ver uit elkaar liggen, of de lijntjes kruisen met andere lijntjes, dan is dat in het algemeen niet duidelijker.

Part compartiment

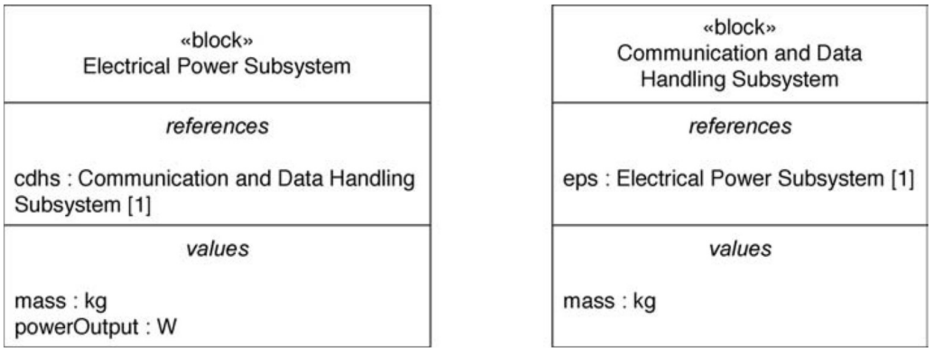
Zoals we weten kunnen we compositie-relaties weergeven met lijntjes met een zwarte ruitje aan het begin. SysML biedt daaraast nog een alternatieve manier om een compositie-relatie aan te geven. Er geldt immers een compositie-relatie met elk object dat in het parts compartiment van een block wordt opgesomd.



Het bovenstaande voorbeeld laat zien dat een DellSat-77 Satteliet is opgebouwd uit een Electrical Power Subsystem, een Attitude and Orbit Control Subsystem, een Environmental Control Subsystem en een Communication and Data Handling Subsystem.

References Compartiment

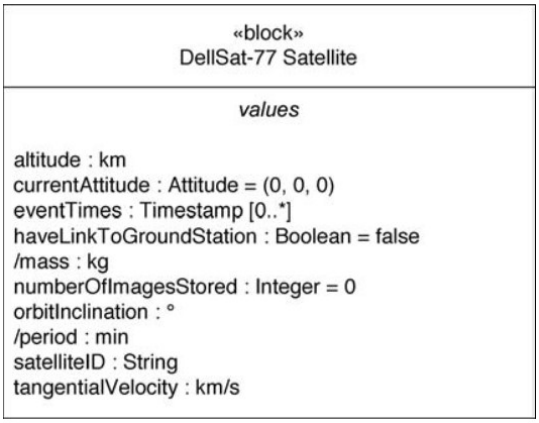
Op soortgelijke manier kan in plaats van de associatiepijl een reference (het kennen van iets) worden weergegeven door datgene dat je kent op te nemen in een references compartiment.



In het bovenstaande voorbeeld is middels references compartimenten weergegeven dat het "Electrical Power Subsytem" 1 "Communication and Data Handling Subsystem" kent, en vica versa.
De multipliciteit wordt tussen vierkante haken weergegeven. [1] betekent dat het er maar 1 is. [1..*] zou betekenen: 1 of meer.

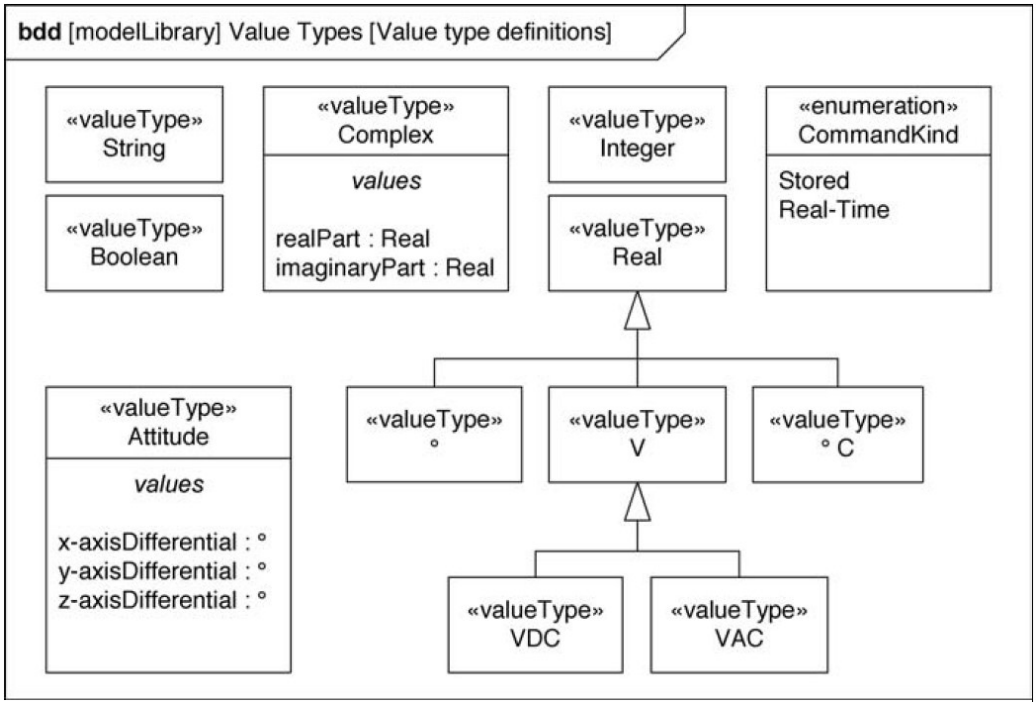
Values Compartiment

Value members van een block representeren een hoeveelheid/grootheid, een boolean of een string. Ze worden weergegeven in het values compartiment van een block.



In het bovenstaande voorbeeld zie je dat eventTimes van het block Sattelite bestaat uit nul of meer Timestamps. (De leading "/' characters voor mass en period zijn typos, net als Attitude ipv Altitude – volgens mij had Delligatti soms een beetje haast :-))

De types van de values, zoals Altitude zijn elders gespecifeerd als "value types":



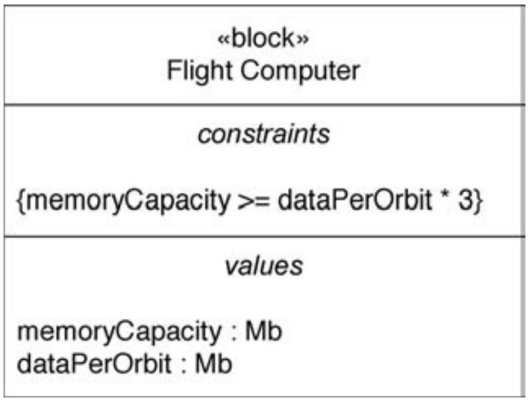
In bovenstaande voorbeeld zie je ook dat bijvoorbeeld het value type V (voor voltage) is afgeleid van het value type Real. Door bij voltages met een specifiek value type te werken (ipv direct met het type Real), kun je op een plek in je architectuur het onderliggende type aanpassen, mocht dat ooit gewenst zijn. Stel bijvoorbeeld dat je besluit dat je voltages wilt gaan weergeven met 8 bit bytes, dan kun je dat doen door V af te leiden van een byte ipv een Real, en hoef je niet in alle operaties waar voltages gebruikt worden real door byte te vervangen.

Constraint compartimenten

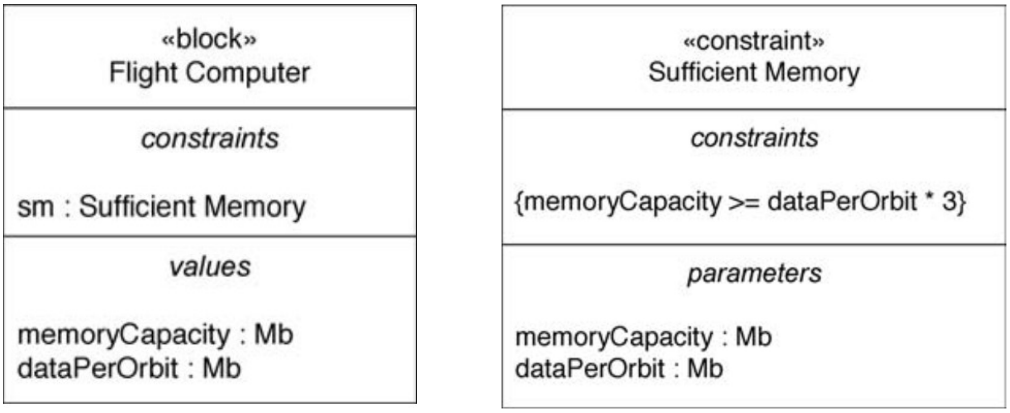
In constraint compartimenten kunnen constraints worden weergegeven. Dat zijn mathematische relaties tussen een aantal eigenschappen.

Elke constraint is geformuleerd tussen kringelhaken: {}
In het onderstaande voorbeeld is de constraint geformuleerd dat de geheugencapaciteit altijd minimaal zo groot moet zijn als 3 maal de hoeveelheid "dataPerOrbit".

NB: Uiteraard moeten de values die in de constraint gebruikt worden, ook in het values compartiment van het block terug te vinden zijn!



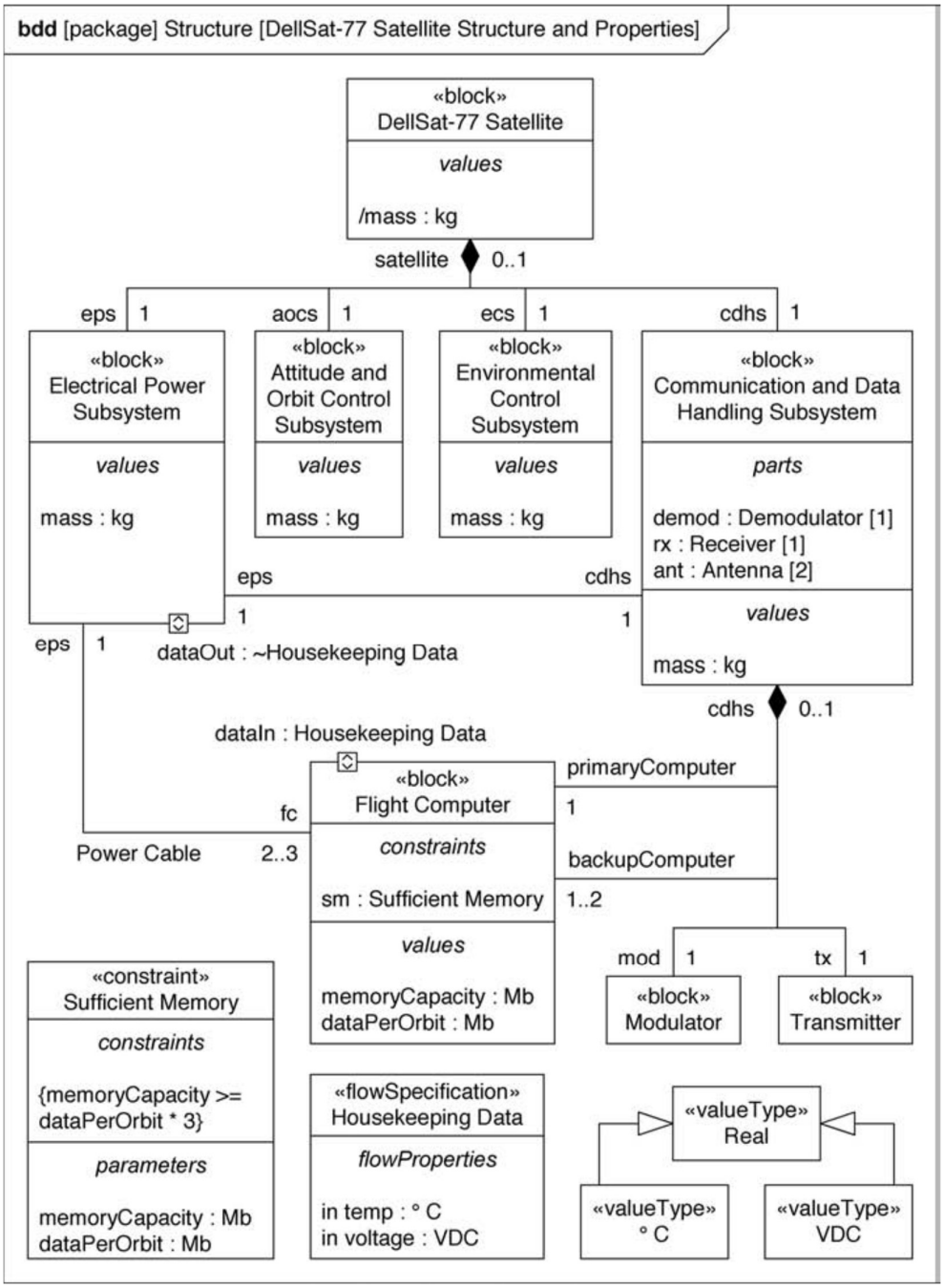
Als een constraint op meerdere plaatsten gebruikt wordt, is het het mooist om gebruik te maken van een constraint specificatie elders (om duplicatie van formules te voorkomen):



Zowel het block als de constraints specificatie bevatten in dat geval een constraints compartiment. De betrokken parameters zijn in de constraints specificatie overzichtelijk opgesomd in een parameters compartiment.

Een voorbeeld van een BDD

In onderstaande diagram is een voorbeeld van een BDD waar veel van de mogelijkheden gedemonstreerd worden.



Discussie van het bovenstaande voorbeeld:

- Aan de header van de BDD is te zien aan de afkorting bdd dat dit diagram een block definition diagram laat zien, met de naam “DellSat-77 Sattelite Structure and Properties”, en dat het in het model terug te vinden in een package genaamd “Structure”.
- Aan het uiteinde van de compositiepijl die het block “DellSat77-Sattelite” verbindt met het block “Electrical Power Subsystem” zien we multipliciteit 1 en de naam “eps”. Dat betekent dat het laatstgenoemde block onderdeel is van het eerstgenoemde block onder de (variabele-) naam “eps”. Als alternatief zou hetzelfde weergegeven

kunnen worden door het block “DellSat-77 Sattelite” een parts compartiment te geven met daarin een member genaamd “eps” met als type “Electrical Power Subsystem”.

- Bij het vak Concurrent Systems Modelling werd gevraagd om in de klassediagrammen voor compositie en reference members zowel die members in de klasse zelf weer te geven als de bijbehorende relatie-pijlen in het diagram. Dat was even ter bewustwording. Bij dit vak doen we dat niet meer. Maak een keuze: gebruik voor een bepaald composite member een composite pijl of zet hem in een parts compartiment, maar niet beide. Hetzelfde geldt voor de references.
- In dit diagram worden geen standard ports gebruikt, al is dat ook valide een moglijkheid binnen een BDD.

Hoe ontwerp je een BDD?

Een BDD is en blijft een soort klassediagram met extra expressiemogelijkheden. Bij het ontwerpen van een BDD kun je de oorspronkelijke Design Like a Robot methode van het vak Concurrent System Modelling hanteren. Vanuit het mixed-object model uit de conceptuele fase, use casebeschrijvingen en activity diagrammen ontwerp je een software-object model, met de berichten als commando’s langs de pijlen. Van daaruit ontwerp je het block diagram (voorheen klasse diagram) op de bekende manier. Vervolgens ga je dat diagram zo duidelijk mogelijk weergeven door goed te overwegen op welke manier je composities, references, interface koppelingen e.d. wilt weergeven.

Meer over BDD’s is te vinden in het bijbehorende hoofdstuk in Delligatti.

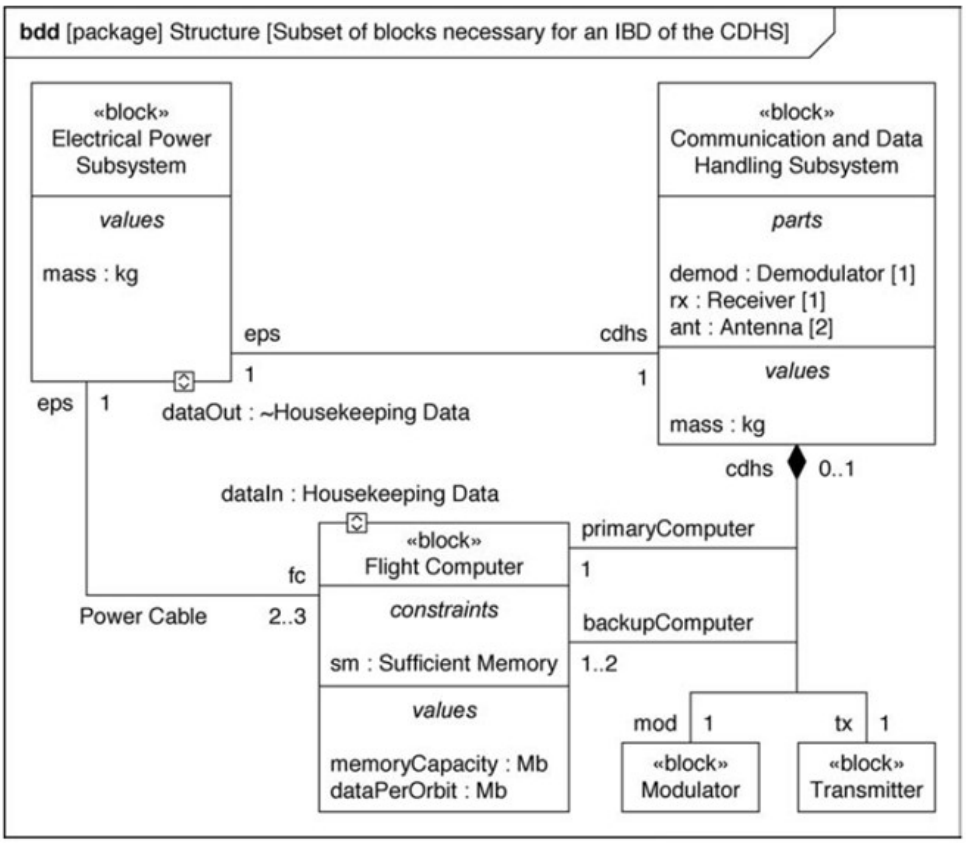
Internal Block Diagram (IBD)

In BDD’s komen we blocks tegen, laten we zeggen, een soort van klassen met parts en references partities.

Een Internal Block Diagram laat **een valide configuratie** van **instanties** binnen **een block** zien, met relaties en/of flows tussen die instanties.

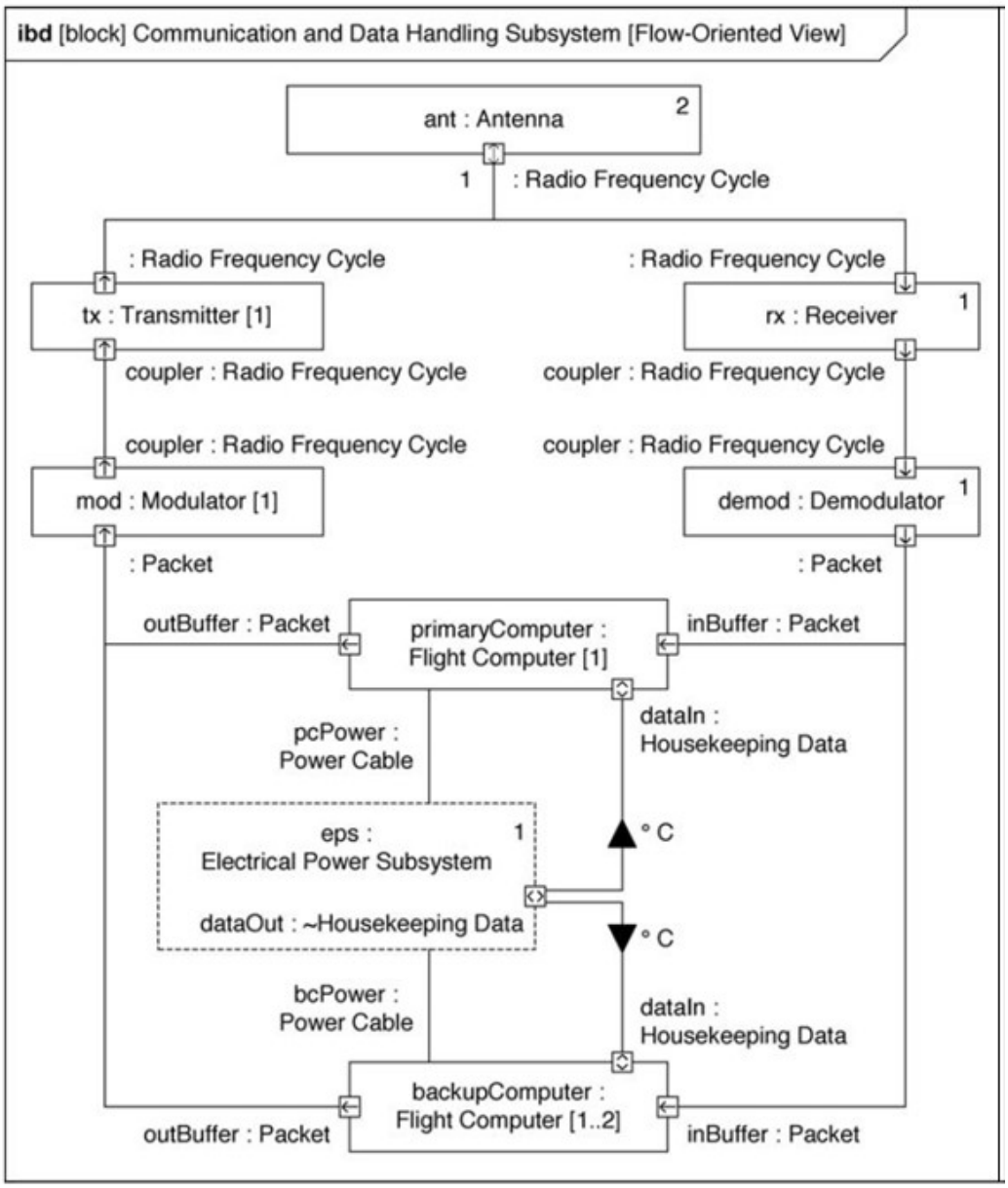
Het geeft dus een indruk van hoe instanties van de parts en references die horen bij een block gebruikt zouden kunnen worden en met elkaar samenhangen.

Stel je voor dat we een IBD willen maken van het block “Communication and Data Handling Subsystem”, dat rechtsboven staat in onderstaande fysieke BDD:



In het bovenstaande voorbeeld kun je zien dat het fysieke block “Communication and Data Handling Subsystem” de volgende parts en references (in totaal 8) heeft:
eps, tx, mod, primaryComputer, backupComputer, demod, rx en ant (2 Antennas).

Het onderstaande voorbeeld geeft een IBD van het fysieke block “Communication and Data Handling Subsystem”. De IBD laat een valide voorbeeld zien van hoe binnen dat block elk van die 8 instanties met elkaar samenwerken.



Discussie van het bovenstaande voorbeeld:

- In de SysML diagram header van dit diagram zie je aan de afkorting “ibd” dat het om een Internal Block Diagram gaat, dat “Flow-Oriented View” heet, en welke zelf voorkomt in het model als onderdeel van het block dat “Communication and Data Handling Subsystem” heet.
- Elk van de 7 composite members van “Communication and Data Handling Subsystem” zijn in zijn IBD terug te vinden als “gesloten rechthoeken”. Zijn reference members (in dit geval maar eentje: eps) zijn in de IBD terug te vinden als “gestippelde rechthoeken”.
- In dit geval is er voor een IBD gekozen dat de flow van informatie tussen de instanties goed in kaart brengt, door de port-connecties als flow-ports vorm te geven.

- In plaats daarvan (of in additie) zou je nog een andere IBD kunnen maken, waarbij de instanties met elkaar verbonden zijn via standard ports. Met die IBD zou dan duidelijk gemaakt kunnen worden via welke interfaces de instanties op elkaar aangesloten zijn. NB: het is doorgaans het duidelijkst als je een keuze maakt: ofwel standard ports, ofwel flowports gebruikten, afhankelijk van wat je met je view (de IBD is maar een view) wilt verduidelijken.
- Multipliciteit:
  Multipliciteit van de instanties kan op twee plekken worden weergegeven (kies er eentje van, voor je diagrammen): een in de rechter bovenhoek van de instantie, of tussen vierkante haken [] achter de type-naam van de instantie.
- Bij alle flow ports wordt het type van de data (zoals RadioFrequencyCycle) of de flow specification (voor de inout ports) (zoals HouseKeeping Data) erbij vermeld (dat is verplicht). Bij sommige ports is de naam van de connectie voor de dubbele punt weergegeven (zoals "coupler"), of alleen de naam van de bijbehorende buffer (zoals "inBuffer").
- Je ziet ook ergens op een verbinding een pijl afgebeeld met een eenheid erbij: °C, om te verduidelijken dat dat informatie is die in de aangegeven richting stroomt. Dat kan verduidelijkend werken als je via input-output-flowports verbinding maakt.
- Bovenstaande voorbeeld laat een fysieke IBD zien: een view op hoe de verwante delen van een fysieke Block via informatiestromen gekoppeld zijn / samenwerken.
- NB dat er geen Power-cable poort op de rand van het schema is getekend. Dat komt doordat het externe, gerefereerde Electrical power subsystem al binnen de IBD getoond wordt.
- Maar wanneer voeg je dat wel een poort toe op de rand van een IBD?
  Dat kan bijvoorbeeld een output-port zijn voor een output waarvoor geen ontvanger bekend is (bijvoorbeeld de uitgangsspanning van een voedingsmodule), of een input-port voor een input waarvan je Block niet weet waarvan het vandaan komt (omdat hij er geen reference naar heeft), zoals misschien water dat binnenstroomt vanaf een waterpomp.
- **Bij een software IBD** speelt is er nog een additionele mogelijkheid: Het kan namelijk zijn dat sub-blokken niet direct gekoppeld zijn, maar dat **het block (=de klasse) zelf nog bepaalde operaties** met de data uitvoert. In dat geval kun je het bijbhorende block in het IBD dezelfde naam als het Block zelf waar de IBD een view op geeft geven, met als **postfix "_internal"**.
  Voorbeeld: stel je maakt een IBD voor het softwareblock "Persoondetectie". Videobeelden kunnen dan evt stromen van een Camera-block naar een blok genaamd "Persoonsdetectie_internal". Die module stuurt de beelden bijvoorbeeld door naar een Hitteanalyse-block, een DierenDetectie-block, een HoofdDetectie-blok, etc, en combineert de resultaten daarvan alvorens die weer door te sturen naar een module die op de conclusie wacht (staat er daadwerkelijk een persoon, en met welke kenmerken?).

Meer over IBD's is te vinden in het bijbehorende hoofdstuk van Delligatti.

CAFCR Realisation View

In de conceptual view hebben we een implementatie van de specificaties ontworpen. De volgende stap is de realization view. Daarbij maken we keuzes t.a.v. welk concrete onderdelen er in een systeem moeten komen te zitten.

Waarom eerst een conceptual view?

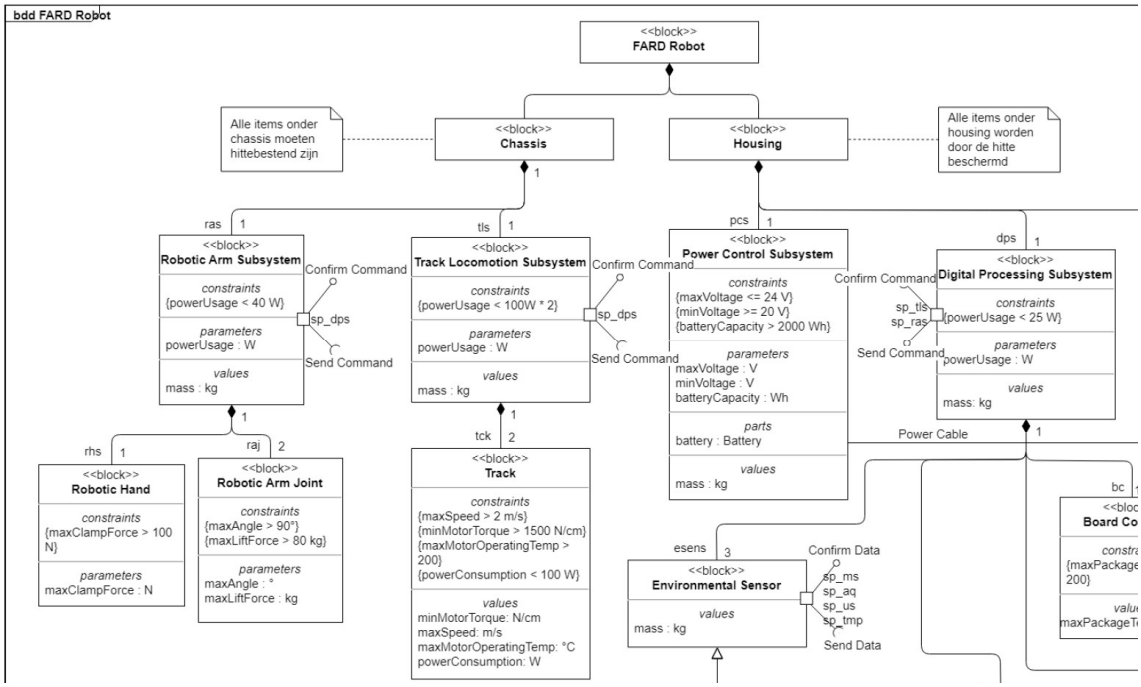
Omdat een conceptueel ontwerp een langere levensduur heeft en herbruikbaarder is dan die van een specifiek ontwerp. Voorbeeld: Een block diagram van de software kan vele generaties van een product worden hergebruikt, ook als bij die generaties de microcontroller en/of programmeertaal wordt aangepast. Bijvoorbeeld naar een snellere arduino, of een switch van een atmel naar een arm processor, etc.

In de realisation view moeten dus een beeld zien te maken van hoe de huidige versie van ons product er in concreto uit komt te zien. Daarvoor moeten er allerlei knopen worden doorgehakt. In de volgende paragrafen wordt daarom aandacht besteed aan de volgende onderwerpen:

- Physical View (ontwerp van de hardware)
- Besluitvorming (algemene gedachtenstappen)
- Besluitvormingstechnieken (4 handige trucs om een goede keus te maken)
- FMEA (in kaart brengen van foutafhankelijkheden)
- Budgettering (hoeveel mag alles kosten)

Physical View

De physical view bestaat uit een of meer BDD's en evt IBD's van de hardware. Onderstaand is een voorbeeld van een physical view van een groep studenten van een voorgaand jaar weergegeven.



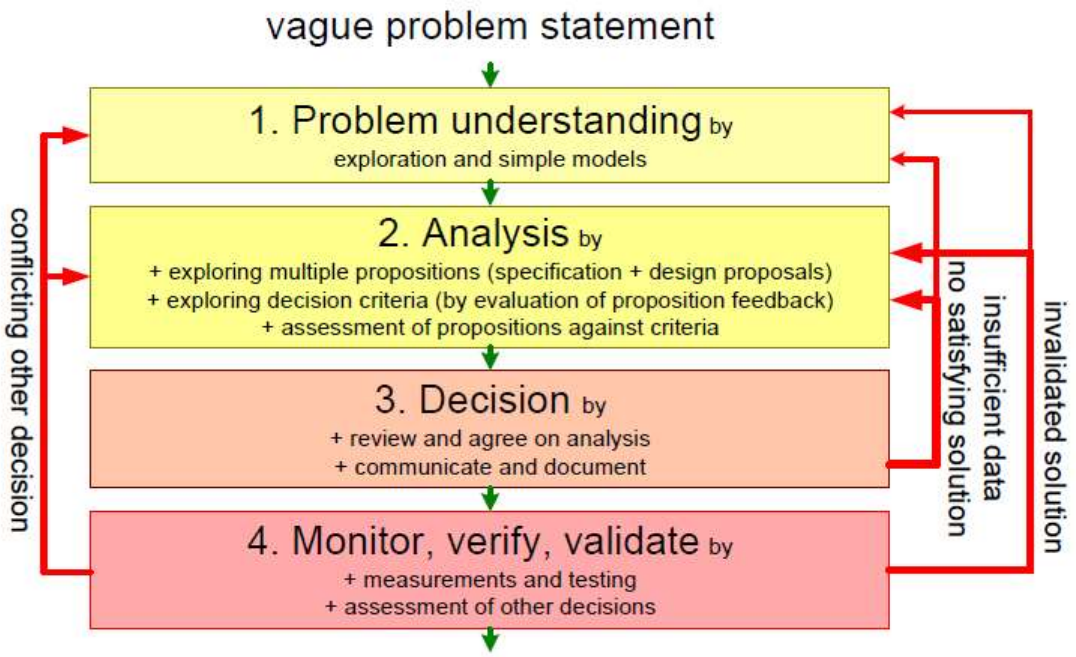
Discussie van het bovenstaande diagram:

- Zorg er voor dat je in de begeleidende tekst bij zo'n diagram alle niet-triviale keuzes beargumenteert. Bijvoorbeeld door te verwijzen naar beslissingsmatrices of berekeningen waar de gemaakte keuzes meer worden verklaard.
- Dus: Waarom moet de maxClampForce minimaal 100N zijn? Waarom moet maxLiftForce minimaal 80kg zijn? Hoeft de robot geen slachtoffers van 90kg te redden? Waarom moet de totale powerconsumption onder de .. W blijven? (Laat met een beslissingsmatrix zien waarom je voor welke batterij hebt gekozen, en een berekening van het aantal watts dat je mag gebruiken om te zorgen dat de gebruiksduur minimaal x uur is. En een beargumentatie waarom x voldoende is.
- .. etc..

Besluitvorming

Om tot een concreet ontwerp te komen, moeten er veel knopen worden doorgehakt. Welke microcontroller en welke programmeertaal gaan we gebruiken? Welke sensoren en actuatoren gaan we inkopen? Misschien gaan we kant en klare subsystemen inkopen? Welke kunnen we dan het beste kiezen? Hoe gaan we ons apparaat vormgeven? Met welke materialen? Etc..

Als je een gefundeeerd besluit wil nemen, kun je globaal genomen het onderstaande pad bewandelen (dat geldt overigens niet uitsluitend voor de realisatiefase).



Als eerste stap probeer je een basis-begrip te krijgen voor het probleem. Bijvoorbeeld door er wat over te lezen of door te experimenteren met eenvoudige modellen.

Als tweede stap ga je in kaart brengen waar je allemaal precies uit kunt kiezen, en ga je voor jezelf op een rijtje zetten welke criteria je voor je keuzes belangrijk vindt.

Als derde stap neem je op basis van de verzamelde informatie van stap twee een beslissing.

Als vierde stap houd je via testen en monitoring in de gaten of je beslissing aan de verwachtingen voldoet.

Elk van deze stappen kan “mislukken”. In dat geval moeten er meestal een of meerdere stappen terug worden gegaan. Als je er bij stap drie bijvoorbeeld achter komt dat je toch nog niet genoeg begrip hebt van het onderwerp om te kunnen beslissen, ligt het voor de hand om terug te gaan naar stap een.

Besluitvormingstechnieken

Er zijn enkele technieken die je kunnen helpen om makkelijker tot een goede beslissing te komen. Daarvan komen in dit vak de volgende aan bod:

- Voordelen en nadelen schema
- Long list en Short list
- SWOT analyse
- Beslissingsmatrix

Voordelen en Nadelen schema

Bij deze methode maak je eenvoudigweg een tabel waarbij je de voor- en nadelen voor elk keuze-alternatief naast elkaar zet.

Alternatief	Voordelen	Nadelen
Alternatief 1	..	…
Alternatief 2	..	..
Alternatief 3	..	..
Alternatief 4	..	..

Long list en Short list

Als je veel alternatieven hebt en veel criteria, dan kost het maken van een keuze veel tijd. Je zou dan het volgende kunnen doen:

- Maak eerst op basis van een beperkt aantal criteria een voorselectie van de alternatieven (dat wordt een long list)
- Maak vervolgens op basis van de volledige criteria een definitieve keuze uit de overblijvende alternatieven (short list)

Voorbeeld: Stel, je wil zowiezo niet meer dan 5 euro voor een sensor uitgeven. Maak dan eerst een lijst met sensoren die minder dan 5 euro kosten (dat is de long list). Pas op die gereduceerde lijst je overige criteria toe (detectiebereik, stroomverbruik, voedingsspanning, etc).

SWOT analyse

Een SWOT (**S**trengths,**W**eaknesses,**O**pportunities,**T**hreats) analyse helpt je passende besluitvorming te vinden voor een bepaalde doelstelling (die doelstelling kan bijvoorbeeld het op de markt brengen van je product zijn).

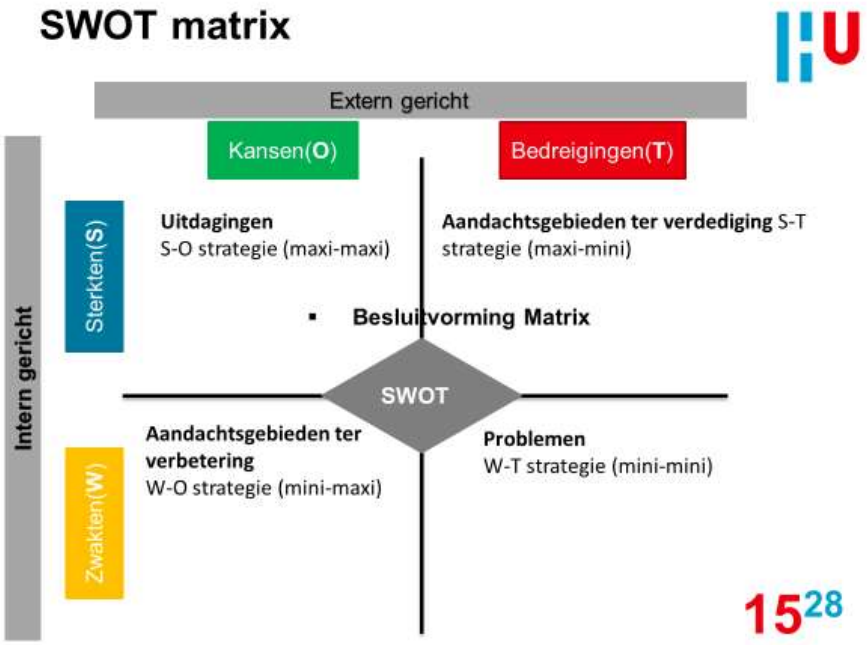
In het algemeen zijn er (potentiele) **eigenschappen** van je doelstelling / product, die je kunt **sorteren**:

- **Strengths** (sterktes) zijn positieve eigenschappen die bijdragen aan het behalen van je doelstelling.
- **Weaknesses** (zwaktes) zijn kritische eigenschappen die het behalen van je doelstelling in de weg staan.

Daarnaast zijn er (externe) **omstandigheden** die je kunt **sorteren** op of zij positief of negatief bijdragen aan het behalen van de doelstelling:

- **Opportunities** (kansen) zijn omstandigheden die het behalen van de doelstelling helpen.
- **Threats** (bedreigingen) zijn omstandigheden die een negatieve bijdrage leveren aan de doelstelling.

Het is goede traditie om de tuples van eigenschap en omstandigheid kun je in een SWOT matrix weergeven, zoals in het onderstaande voorbeeld.



Op deze manier heb je een meer grafisch overzicht van de situatie. Afhankelijk van het kwadrant waar je tuple terecht komt, kan het geclassificeerd worden als **Probleem**, **Uitdaging**, **Aandachtsgebied** ter **verdediging** en **Aandachtsgebied** ter **verbetering**.

Het inkoppertje is (zet je teiltje vast klaar..) dat je probeert bedreigingen en zwaktes te verminderen **mini**maliseren) en kansen en sterktes uit te buiten (**maxi**maliseren).

Voorbeeld: We maken ventilators voor laptops. Een eigenschap daarvan is een lage geluidsproductie. Die is voor onze ventilator behoorlijk goed. De geluidsproductie is nu dus een **strength**. Nu weten we dat een concurrerende fabrikant bezig is met de ontwikkeling van een stiller type ventilator. Dat is een negatieve omstandigheid, en dus een **threat**. De geluidsproductie is dus een **aandachtspunt ter verdediging**. Nu we dat hebben vastgesteld, kunnen we daar onze strategie op afstemmen. Bijvoorbeeld snel onze voorraden voor een lagere prijs op de markt dumpen en vervolgens stoppen met ventilatorproductie voordat die concurrent op de markt komt. Of zelf ook innoveren en proberen met de geluidsproductie omlaag te brengen. Of t.z.t. ventilators bij de concurrent inkopen, onze eigen sticker er op plakken en doorverkopen.

Beslissingsmatrices

Last but not least: een fantastisch instrument om een transparante en afgewogen beslissing te maken is een beslissingsmatrix. Hoe zo'n matrix werkt, wordt uitgelegd aan de hand van het onderstaande voorbeeld, een beslissingsmatrix die helpt bij de keuze van het vervoermiddel voor een uitstapje naar Parijs:

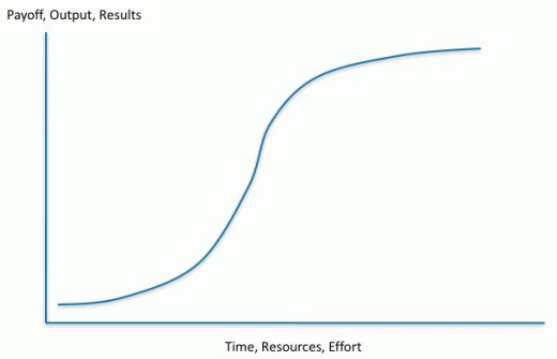
Kwaliteitsattribuut	Weegfactor	Auto		Bus		HST		Vliegtuig	
		Waarde	Score	Waarde	Score	Waarde	Score	Waarde	Score
Comfort	0,2	+	40	+	50	++	100	-	30
Kosten	0,3	€ 125	40	€ 50	100	€ 130	40	€ 160	30
Milieu	0,2	1000MJ	30	750MJ	40	300MJ	100	3200MJ	10
Privacy	0,1	++	100	-	20	+	50	-	25
Reistijd	0,2	5.8u	55	7.0u	45	4.4u	70	3.2u	100
Gewogen gemiddelde			47		59		71		40
Minimum score			30		20		40		10

- **Kwaliteitsattribuut**
  In de linker kolom zetten we kwaliteitsattributen die we belangrijk vinden voor zo'n reisje. We hechten aan hoog comfort, gunstige kosten, gunstig voor het milieu, privacy en gunstige reistijd.
- **Weegfactor**
  In de kolom ernaast geven we weer **hoe zwaar elk van die kwaliteitsattributen weegt**, met een getal dat we **weegfactor** noemen. In dit voorbeeld is de weegfactor voor kosten 0.3, terwijl die voor privacy 0.1 gekozen is. Daarmee wordt aangegeven dat kosten zwaarder meewegen in de besluitvorming dan de mate van privacy. Als de persoon die deze beslissingsmatrix had ingevuld een multi-miljonair zou zijn, had hij vermoedelijk andere weegfactoren gekozen.
  NB: om het overzicht van het belang van elke weegfactor te houden, is het belangrijk dat de **som van de weegfactoren** gelijk is aan **1.0**.
- **Keuzealternatieven**
  In de overige kolommen zien we de keuzealternatieven waaruit we de meest geschikte willen kiezen. In dit geval zijn dat Auto, Bus, HST en Vliegtuig.
- **Waarde**
  Voor elk keuzealternatief vullen we een waardering in voor elk kwaliteitsattribuut. Als het even kan, vullen we een kwantitatieve grootheid in. Bijvoorbeeld voor kosten van de auto naar Parijs kunnen we berekenen dat er 125 euro aan brandstof, 1000 MegaJoule aan energie en 5.8 uur reistijd nodig is.
  In sommige andere gevallen, zoals bij comfort en privacy is kwantificeren lastig. Daar geven we een meer gevoelsmatig oordeel aan middels --,-,+,++
- **Score**
  De waarden bij de verschillende kwaliteitsattributen hebben verschillende eenheden en verschillende bereiken. Hoe kunnen we nu 125 euro vergelijken met 5.8 uur reistijd?
  Om dat mogelijk te maken **zetten** we alle **waarden om in** (eenheidsloze) **scores**. Dat doen we **per rij** (kwaliteitsattribuut) in de tabel. Voor de kwalitatieve beoordelingen --,-,+ en ++ vullen we in principe scores in van respectievelijk 0,25,50 en 100. Eventueel kunnen we vervolgens nog wat nuances aanbrengen. Als we het

comfort van de auto wat lager beoordelen dan die van de bus, kunnen we de score wat verlagen, van 50 naar 40. Voor de **kwantitatieve** scores vullen we voor de **meest gunstige** waarde in de rij de score **100** in, en voor de overige waarden een score die **relatief** daar aan is. Voorbeeld: de reistijd per vliegtuig, 3.2 uur, is het best. Die krijgt dus een score van 100. De reistijd van de HST is 4.4 uur. De verhouding 3.2/4.4 is ongeveer 0.7, ofwel 70%. Daarom geven we de reistijd van de HST een score van 70 (Om precies te zijn 71.2, maar voor deze beslissingstabel is een score-nauwkeurigheid van 5 meer dan voldoende, vandaar dat we het afronden naar 70 – daar wordt de tabel leesbaarder door).

- **Gewogen gemiddelde**
  Vervolgens bepaal je voor elk keuzealternatief het gewogen gemiddelde van zijn scores. Dat doe je door elk van zijn scores te vermenigvuldigen met de bijbehorende weegfactor en vervolgens bij mekaar op te tellen. Het gewogen gemiddelde reflecteert dus zowel hoe belangrijk je elk attribuut vindt, als de mate waarin het keuzealternatief scoort op dat attribuut. Voor de auto wordt het gewogen gemiddelde dus: 0.2*40 (comfort) + 0.3*40 (kosten) + 0.2*30 (milieu) + 0.1*100 (privacy) + 0.2*55 (reistijd) = 47
- **Minimum score**
  We kunnen ook per scorekolom bepalen wat de kleinst voorkomende score is.
- **Besluitvorming**
  Je kunt met deze resultaten op verschillende manieren tot een besluit komen. Een veelgebruikte manier is dat je eerst de alternatieven afstreept met een minimum score onder een bepaalde grens, laten we zeggen van 15 of lager. Daarmee voorkom je dat er een alternatief bij zit dat extreem slecht scoort op een bepaald punt. In ons voorbeeld zouden we dus het vliegtuig (minimum 10) al bij voorbaat kunnen wegstrepen.
  Vervolgens kiezen we van de overgebleven alternatieven die met het hoogste gewogen gemiddelde. De HST heeft een gewogen gemiddelde van 71, en scoort daarmee beter dan de auto en de bus. In dit geval kiezen we dus voor de HST voor ons uitstapje naar parijs. Stel nu dat de spoorwegen staken, dan blijven de auto en de bus over. Van die twee heeft de bus het hoogste gewogen gemiddelde, dus in dat geval kiezen we voor de bus.

S-curve



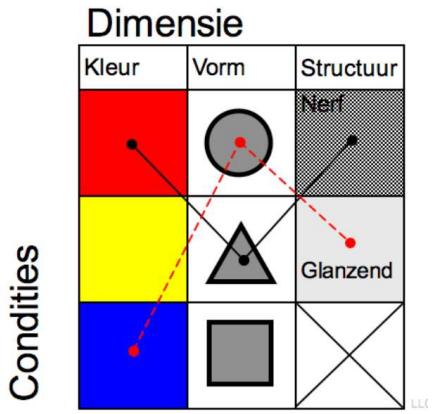
In het bovenstaande voorbeeld hebben we de kwantitatieve waarden linear/evenredig omgezet naar score waardes. In de praktijk is er vaak ook nog wat voor te zeggen on die scores nog eens door een s-curve te halen, bijvoorbeeld: newscore = $\frac{100}{1+e^{-\frac{(oldscore-5)}{100}}}$.

De achterliggende gedachte is: goed is goed genoeg. De allerhoogste score is leuk, maar er is vaak onevenredig veel voor moet opofferen is om dat te bereiken. Voorbeeld: als je een grafische kaart koopt die een jaar geleden uitkwam, is hij twee keer zo goedkoop als de nieuwste grafische kaart, maar slechts 20% minder snel. De sweetspot / bang-for-je-buck-spot op de s-curve is vaak een punt waar je resultaat hoog is, maar de s-curve al aardig aan het afvlakken is.

Aan de andere kant: als de prijs van de grafische kaart bekend is, en al expliciet verdisconteerd in de beslissingsmatrix voorkomt, dan is gebruik van zo'n s-curve wellicht niet nodig.

De Morfologische Matrix

Bij de realisatiefase is ook de al in een eerder hoofdstuk behandelde morfologische matrix ook een geschikt instrument. Kies je ervoor de software te laten draaien op een Raspberri PI, een ESP32 of een BluePill? Kies je ervoor de user interface aan te bieden via een WebInterface, een TouchScreen of een Pc-applicatie? Etc.. Elke combinatie heeft zijn voor en nadelen. Door zo'n matrix met keuzes op te stellen maak je het palet aan mogelijke combinaties van keuzes inzichtelijk.

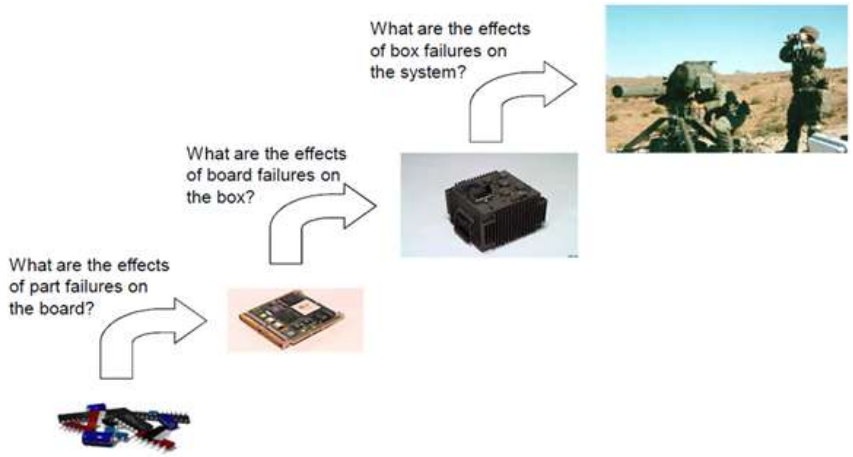


FMEA

Een belangrijke analysetool is de **F**ailure **M**ode and **E**ffect **A**nalysis (FMEA).

- Je kunt daarmee **in kaart** brengen hoe een product zou kunnen **falen**.
- Het legt de **zwakke punten** bloot voordat ze geld kosten.
- Daarmee kun je ontevreden klanten, uitval en **reparaties beperken**.
- Zo biedt het een basis voor **onderhoudbaarheid**, **veiligheid** en **testbaarheid**.

Bottom-up failure modes



FMEA spreadsheet

Je kunt dit systematisch beschrijven door voor elke failure mode een FMEA spreadsheet te maken, zoals in het onderstaande voorbeeld:

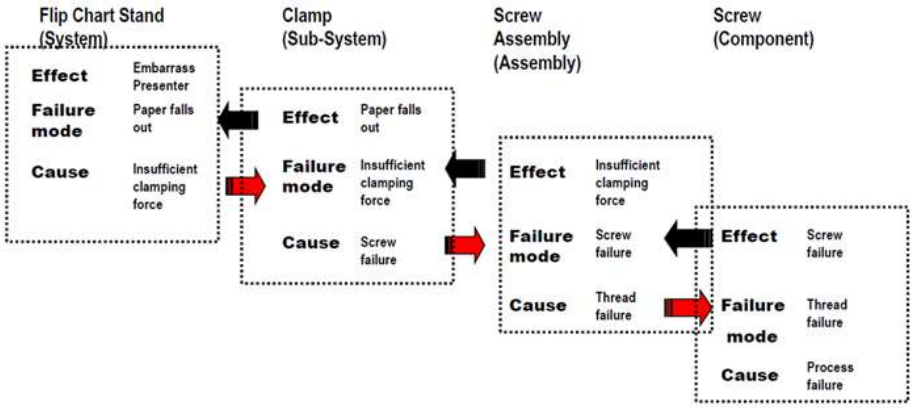
Functie	Failure mode	Oorzaak (Cause)	Effect
Afdruk op papier maken	Inkt komt niet op papier	Inktreservoir is leeg	Geïrriteerde klant Onbruikbare afdruk
		Inktuitgang is geblokkeerd	
	Papier scheurt	Papier te vochtig	Geïrriteerde klant
		Papier is gekreukeld	

Het in kaart brengen van kettingreacties

Stel je voor dat het regulator-chipje stuk gaat, wat voor invloed heeft dat op het gedrag van het de microelectronica module waar het deel uit maakt? Wat voor invloed heeft dat op het subsysteem waar het deel uit maakt? Wat voor invloed heeft dat op het functioneren van het product?

FMEA waterval

In dat kader wordt ook wel gesproken van de "FMEA waterval".
Daarbij wordt een hele keten van FailureMode-Cause-Effect blokken in kaart gebracht.



Discussie van het bovenstaande diagram:

- De **Failure Mode** van de **fase** (traptrede) "Screw Assembly" is "Screw Failure".
  Dat is per definitie **gelijk aan** de oorzaak (**cause**) van de Failure mode in de trap **links** ervan,
  en aan het **effect** van de Failure mode in de fase (trede) **rechts** ervan.
- Hetzelfde geldt voor de overige failure modes

Zo zie je twee stromen van causale verbanden. De effecten van wat er gebeurt, stromen van het laagste niveau naar het hoogste niveau. De oorzaken van wat er gebeurt, vind je door steeds naar een lager gelegen niveaus te kijken.

Faalkennis kwantificeren

Het is een goed idee om voor het bovenste niveau (je eindproduct) een kwantificatie voor het "risico" van je effect te bepalen.

Het Risico

Risico = **kans** op falen * het **effect** van falen
(meestal uitgedrukt in €)

Dus het "**gemiddeld genomen bedrag**" dat je per product er aan kwijt bent.
Voorbeeld: de kans dat de productiefout van een schroefje optreedt is 0.01%.
Het effect ervan is dat de auto moet teruggehaald worden naar de garage voor vervanging van het schroefje: 250 euro. Het risico (gemiddelde faal-kosten) dat hoort bij dat schroefje is dan 0.01% maal 250 euro = 2.5 eurocent. Stel dat er in een auto 5000 schroefjes zitten. Dan is het risico van de schroefjes voor de auto 5000 maal 2.5 eurocent = 125 euro. Dus dat is het bedrag dat we in de kostprijsberekening van de auto moeten toevoegen om dat risico af te dekken.

NB: vaak zijn er meerdere effecten. Bijvoorbeeld dat je **merknaam** wordt beschadigd. Dat is vaak moeilijker te kwantificeren.

Budgettering

Resources

Resources zijn over het algemeen **schaars**. Om **nare verrassingen** achteraf te **voorkomen** is het verstandig om ze vooraf te budgetteren.

Voorbeelden van resources die kunt budgetteren zijn:

- Geheugen
- Reactietijd
- Energieverbruik
- Geld
- Gewicht

Voorbeeld van een geheugen-budget

In het onderstaande voorbeeld is een geheugen-budgettering gemaakt:

Onderdeel	Code [MB]	Object data [MB]	Bulk data [MB]	Totaal [MB]
shared code	11.0			11.0
user interface process	0.3	3.0	12.0	15.3
database server	0.3	3.2	3.0	6.5
print server	0.3	1.2	9.0	10.5
optical storage server	0.3	2.0	1.0	3.3
communication server	0.3	2.0	4.0	6.3
UNIX commands	0.3	0.2	0	0.5
compute server	0.3	0.5	6.0	6.8
system monitor	0.3	0.5	0	0.8
application SW total	**13.4**	**12.6**	**35.0**	**61.0**
UNIX Solaris 2.x				**10.0**
file cache				**3.0**
total				**74.0**

Per subsysteem is het geplande geheugengebruik in kaart gebracht – bestaande uit een aantal megabytes voor code, een aantal voor de object data (bijv. persoonsgegevens) en een aantal voor de bulk data (bijv. plaatjes en videos).

Zo krijg je van tevoren een idee hoeveel geheugen er voor het systeem in totaal nodig is. Dat heeft twee belangrijke voordelen:

- Je krijgt een idee van wat iets in totaal **gaat kosten**.
- Het biedt een **leidraad** voor **ontwikkelteams**.
  (Bij software development is er vaak een uitwisseling mogelijk tussen geheugengebruik en performance of kwaliteit. Budgettering van het geheugen helpt om de juiste trade-off te maken).