Hoofdstuk 1
Introductie

1.1 Over softwareontwikkeling

In de meeste gevallen zijn er veel mensen betrokken bij de ontwikkeling van software: niet alleen de klant die de opdrachtgever is en de programmeurs die het eigenlijke werk doen, maar ook mensen die deskundig zijn op het gebied waar de applicatie over gaat, analisten, systeembeheerders, mensen met financiële verantwoordelijkheid et cetera.

Een ontwikkeltraject kan maanden of zelfs jaren duren. Omdat er vaak veel mensen bij betrokken zijn, is er in feite sprake van een *project*. In zo'n geval moet je een projectmatige aanpak hebben om alles in goede banen te leiden. Meestal is er dan ook een projectmanager die de gang van zaken bewaakt. Over projectmanagement gaat dit boek niet.

Behalve een algemene projectmatige aanpak heb je ook een meer inhoudelijke *methode* nodig om binnen redelijke tijd tot het gewenste product te komen. In het verleden zijn er veel methoden bedacht en beschreven om op een systematische manier te komen tot softwareontwikkeling. Bekende methoden zijn *watervalmethode*, *Rational Unified Process* (RUP) en *eXtreme Programming* (XP). Vaak geven deze methoden aanwijzingen hoe je het ontwikkelproject moet inrichten.

1.1.1 De watervalmethode

Bij de watervalmethode verloopt het ontwikkelproces als een stroom van fasen die je achtereenvolgens doorloopt om het eindproduct te krijgen. De fasen waaruit deze methode bestaat zijn de volgende:

- vastleggen van de eisen waaraan de applicatie moet voldoen;
- maken van een ontwerp;
- coderen van diverse gedeelten van de applicatie (implementatie);
- samenvoegen van de verschillende onderdelen (integratie);
- testen van de applicatie en het aanbrengen van verbeteringen;
- afleveren en installeren bij de klant;
- onderhoud na aflevering.

Deze fasen vormen een vrij natuurlijke route waarlangs je een willekeurig product (niet alleen een applicatie) tot stand kunt brengen. Immers, voordat je gaat ontwerpen moet je weten wat de eisen zijn die de klant aan het product stelt. Het ontwerp is natuurlijk nodig om te kunnen coderen, zeker als je met een team van programmeurs werkt. De geschreven code moet je uiteraard testen voor je het aflevert en installeert bij de klant.

Er is vrij veel kritiek op de watervalmethode, waarschijnlijk vooral omdat deze in het verleden nog wel eens rigoureus is toegepast: elke fase moest volledig zijn afgerond alvorens je aan een volgende fase kon beginnen. Dat betekent bijvoorbeeld dat de klant in het begin van het proces, aan het einde van de eerste fase, zijn handtekening zet onder de eisen waaraan de applicatie moet voldoen. Wijzigingen in de eisen zijn niet mogelijk, of alleen ten koste van veel mankracht en geld.

Maar in de praktijk komt het vaak voor dat de klant weliswaar van tevoren een aantal eisen heeft, maar dat deze eisen gaandeweg de ontwikkeling van de applicatie kunnen veranderen, dat niet alle eisen even zwaar blijken te wegen, dat de klant sommige oplossingen waar de programmeurs mee komen niet heeft kunnen voorzien en om die reden sommige eisen kan laten vallen, of juist met nieuwe eisen komt.

Voordeel van de watervalmethode is dat de fasen helder zijn afgebakend, waardoor iedereen weet waar hij aan toe is; nadeel is dat het model geen ruimte laat om later terug te komen op eerder genomen beslissingen.

Er zijn andere methoden, waarin de flexibiliteit als het ware is ingebouwd.

1.1.2 Rational Unified Process

Het Rational Unified Process (RUP) is een methode ontworpen door Rational Software (nu IBM). De methode onderscheidt in principe vier fasen:

- *inception*, de startfase van het project waarin gekeken wordt naar de haalbaarheid, de financiële aspecten en de risico's van het project;
- *elaboration*, de uitwerkingsfase waarin een analyse wordt gedaan van het probleem en een zogeheten ontwerpmodel wordt gemaakt op grond van de resultaten van de analyse;
- *construction*, de fase van het maken van de software;
- *transition*, de fase waarin de software wordt overgedragen aan de klant.

RUP heeft expliciet een aantal momenten ingebouwd waarop het proces kan worden afgeblazen, bijvoorbeeld wanneer aan het eind van de startfase blijkt dat de financiële consequenties te groot zijn.

RUP is een *iteratief* proces, voornamelijk in de constructiefase. Iteratief wil zeggen dat de software niet in één keer tot stand komt, maar in betrekkelijk kleine, min of meer afgeronde componenten. Bij de oplevering van een component wordt deze geëvalueerd, de eisen eventueel bijgesteld, de analyse en het model eventueel veranderd en de opgedane

ervaring meegenomen bij de ontwikkeling van de volgende component. Bij dit alles maakt RUP nadrukkelijk gebruik van grafische weergaven van aspecten van de software met behulp van UML omdat RUP veel waarde hecht aan goede en uitvoerige documentatie, in zowel geschreven als grafische vorm.

Over RUP valt veel meer te zeggen dan ik hier heb gedaan. Op internet zijn veel bronnen te vinden die uitvoerig ingaan op deze methode.

1.1.3 eXtreme Programming

Een betrekkelijk nieuwe methode van softwareontwikkeling is eXtreme Programming (XP) met een paar opvallende kenmerken. XP houdt er niet van om de eisen voor een applicatie eerst uitvoerig op papier te formuleren, omdat dit veel tijd kost en de eisen later toch vaak blijken te veranderen. In plaats daarvan wordt de klant nauw bij het XP-ontwikkelproces betrokken.

Als uitgangspunt voor het proces gelden zogeheten *user stories*, een tamelijk globale omschrijving door de gebruiker van een deel van de functionaliteit die het systeem moet krijgen. Het mag niet langer dan een tot drie weken kosten om een user story te implementeren. Tijdens de analyse en het ontwerp van een user story en tijdens de implementatie is de klant continu aanwezig om vragen over details te beantwoorden en om feedback te geven op het geleverde werk.

XP hecht veel waarde aan het in een vroeg stadium testen van kleine stukjes code (*unit testen*) en het regelmatig verbeteren van de structuur van de code (*refactoring*). Het programmeerwerk wordt gedaan door programmeurs die in tweetallen aan een pc zitten (*pair programming*). Het voordeel hiervan is, eenvoudig gezegd, dat twee meer weten dan een en dat programmeurs elkaar kunnen stimuleren en aanvullen. Terwijl de een code tikt, kan de ander nadenken. Regelmatig wisselen de programmeurs van rol. Als een bepaalde user story geïmplementeerd is, voert de klant een test uit om te zien of hij het gemaakte deel accepteert.

XP maakt op een betrekkelijk informele manier gebruik van UML en alleen als het nodig is.

1.1.4 Gemeenschappelijke aspecten van de methoden

Er bestaan nog veel meer methoden voor softwareontwikkeling, en vaak vind je daarin aspecten van de drie hier genoemde methoden terug. De verschillen zitten vooral in de nadruk die op sommige aspecten wordt gelegd of juist niet, op de volgorde van de werkzaamheden en op de organisatie van het hele project.
Ondanks deze verschillen is een groot aantal aspecten in alle methoden terug te vinden:

- onderzoek naar de eisen waaraan het systeem moet voldoen;
- analyse van de eisen en van het probleemgebied;
- het maken van een model;
- het schrijven van code met het model als uitgangspunt;

- het testen van het systeem;
- de oplevering van het systeem.

1.1.5 Welke methode gebruikt dit boek?

Dit boek gebruikt geen vastomlijnde methode, maar richt zich op de algemene aspecten van softwareontwikkeling die in de vorige paragraaf staan genoemd en die, onafhankelijk van de methode die je kiest, belangrijk zijn voor elke programmeur.

1.2 Samenvatting

- Bij softwareontwikkeling is vaak sprake van teamwerk gedurende langere tijd.
- Er zijn veel verschillende methoden voor de ontwikkeling van applicaties die alle tot doel hebben het ontwikkelproces in goede banen te leiden.
- De watervalmethode, RUP en XP zijn bekende ontwikkelingsmethoden.
- Naast grote verschillen zijn er ook veel overeenkomsten tussen de verschillende methoden.

1.3 Vragen

1. Geef een korte omschrijving van de watervalmethode.
2. Geef een korte omschrijving van RUP.
3. Geef een korte omschrijving van XP.

1.4 Oefeningen

Oefening 1.1

Zoek meer informatie over XP en geef hiervan een beschrijving in je eigen woorden. Welke voor- en nadelen zie je in deze methode?

Oefening 1.2

Zoek meer informatie over RUP en geef hiervan een beschrijving in je eigen woorden. Welke voor- en nadelen zie je in deze methode?

Hoofdstuk 2

Use cases

2.1 Inleiding

Heb je wel eens aan iemand, die waarschijnlijk wat ouder is, moeten uitleggen wat MSN is? Misschien heb je zoiets als dit geantwoord: MSN is chatten met je vrienden, en chatten is eigenlijk met elkaar praten door om de beurt iets in te tikken.

Zonder dat je het besefte, heb je uitgelegd wat MSN is aan de hand van een *use case*. Een use case is (een beschrijving van) een taak van een softwaresysteem vanuit het gezichtspunt van de gebruiker. Iets anders gezegd: een use case bestaat uit alle activiteiten die leiden tot een doel dat een gebruiker met een systeem wil bereiken.

Als je een nieuw programma schrijft, in opdracht van iemand anders, gebruik je use cases vooral om het met de opdrachtgever eens te worden over datgene wat hij met het systeem wil bereiken. Om echt nuttig te zijn voor de ontwikkeling van software moet zo'n beschrijving wat uitgebreider zijn dan in het voorbeeld hiervoor.

In dit hoofdstuk leer je hoe je use cases kunt maken, hoe je een use-casediagram tekent, hoe je een use-casebeschrijving maakt, hoe je in een use case gebruik kunt maken van andere use cases en hoe je een use cases kunt vertalen naar een *activiteitendiagram*.

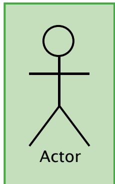
2.2 Een eenvoudige use case

Laten we eens kijken naar een eenvoudig systeem dat weliswaar niets met software te maken heeft, maar waarvan toch iets te leren valt: een *fiets*. Het is niet moeilijk hiervoor een use case te vinden, want het belangrijkste doel van het gebruiken van een fiets is: *een stuk fietsen*. Een use case geef je een korte naam die duidelijk aangeeft wat het doel is van de use case, in dit geval: *Een stuk fietsen*, of, nog korter: *Fietsen*.

Om gemakkelijk en zinvol over de use case te kunnen communiceren maak je een *use-casediagram* en een *use-casebeschrijving*. Ik zal met diagrammen beginnen.

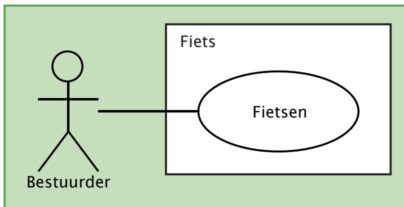
2.2.1 Een use-casediagram

In een use-casediagram teken je in elk geval degene die de use case uitvoert: deze heet de *actor*. De actor is degene die de handeling, de actie, uitvoert. Een actor stel je voor door middel van een draadfiguurtje als in figuur 2-1.



Figuur 2-1

Verder bestaat het use-casediagram uit een rechthoek. Die rechthoek met het gedeelte daarbinnen stelt het systeem voor, in dit geval een fiets. Binnen deze rechthoek zet je de naam van het systeem (Fiets) en de eigenlijke use case: een ovaal met daarin de naam van de use case, zie figuur 2-2.



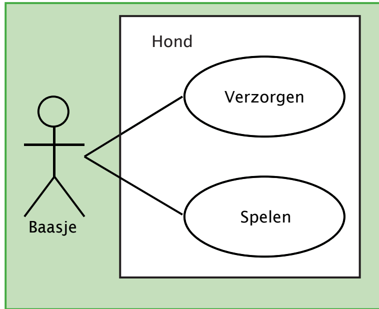
Figuur 2-2

Zoals je ziet heeft de actor een naam gekregen: Bestuurder. Als je fietst ben je de bestuurder van de fiets, dat is de rol die je op dat moment speelt. Geef een actor altijd een naam die past bij de rol die hij of zij in het systeem speelt.
Van de bestuurder loopt een lijn naar de use case, om aan te geven dat deze actor deze use case uitvoert. Zo'n lijn heet in UML een *associatie* (Engels: association).

De rechthoek die het systeem begrenst heet de *systeemgrens* (system boundary). De systeemgrens wordt vaak weggelaten omdat hij niet echt veel informatie toevoegt aan het plaatje en de meeste mensen die met het diagram werken toch wel weten om welk systeem het gaat.

2.3 Meer dan één use case

Bij een wat ingewikkelder systeem zul je vaak meer dan één use case kunnen vinden. Neem bijvoorbeeld een *hond*. De meeste hondenliefhebbers doen ruwweg twee dingen met hun hond: hem *verzorgen* en met hem *spelen*. Dat zijn twee verschillende use cases en het use-casediagram komt er dan uit te zien als in figuur 2-3.



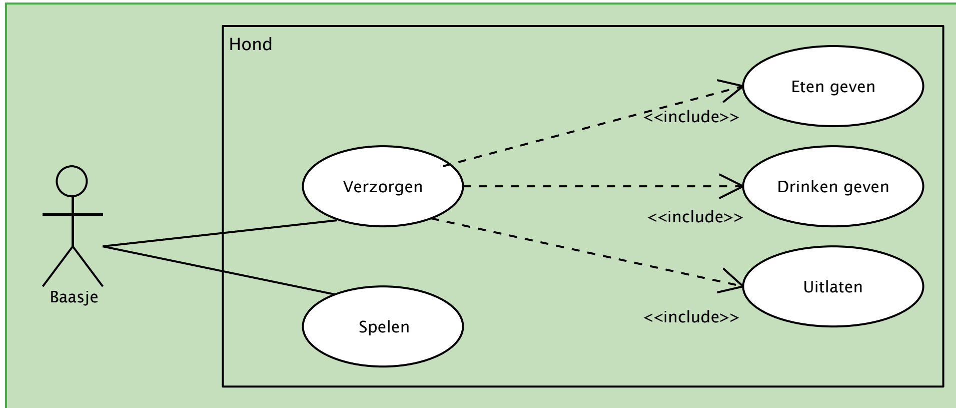
Figuur 2-3

Het kan heel goed zijn dat meer dan één persoon de hond verzorgt en met hem speelt. Degene die dat doet, heeft op dat moment de rol van zijn baasje overgenomen. Daarom is de naam van de actor in dit geval: Baasje.

2.3.1 Include

Een use case als Verzorgen is heel globaal en daardoor nogal vaag. Als je nadenkt over wat het verzorgen van een hond inhoudt, kom je al gauw tot de volgende drie dingen: eten geven, drinken geven en uitlaten. Dit zijn op zichzelf drie verschillende use cases. Je kunt zeggen dat de use case Verzorgen is opgebouwd uit de use cases Eten geven, Drinken geven en Uitlaten, zie figuur 2-4.

Om aan te geven dat de ene use case gebruikmaakt van de andere teken je een pijl met een streepjeslijn van de ene naar de andere use case. Bij de pijl zet je het woord *include* tussen twee speciale tekens: «`include`».
Deze tekens heten *guillemets*, en ze worden in Franse teksten (en in UML) als aanhalingsteken gebruikt. In UML heet een woord tussen deze aanhalingstekens een *stereotype*. De gestippelde pijl heet in UML een *afhankelijkheid* (Engels: dependency). Het welslagen van de use case Verzorgen is *afhankelijk* van het welslagen van de drie use cases Eten geven, Drinken geven en Uitlaten.



Figuur 2-4

De drie pijlen met «`include`» geven aan dat je de use case Verzorgen pas succesvol kunt afronden als je de drie use cases waar de pijlen naar wijzen eerst hebt afgerond.

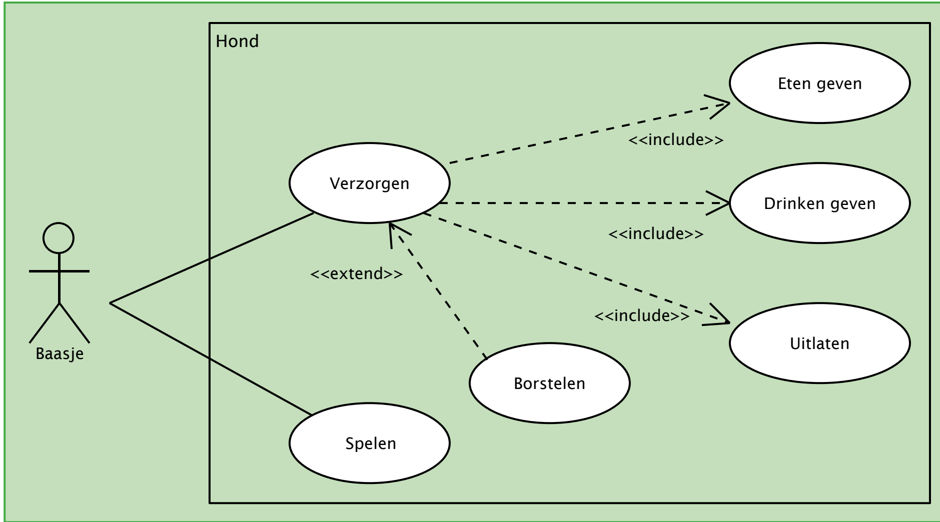
2.3.2 Extend

Het verzorgen van een hond bestaat niet alleen uit eten geven, drinken geven en uitlaten, maar eventueel ook uit borstelen. Het borstelen hoeft niet elk dag te gebeuren, maar alleen als dat nodig is, of als het baasje het leuk vindt om te doen. Borstelen is een uitbreiding van de use case Verzorgen, maar het is niet noodzakelijk hem uit te voeren, zoals bij include. In UML geef je zo'n uitbreiding aan met het stereotype «`extend`», zie figuur 2-5.

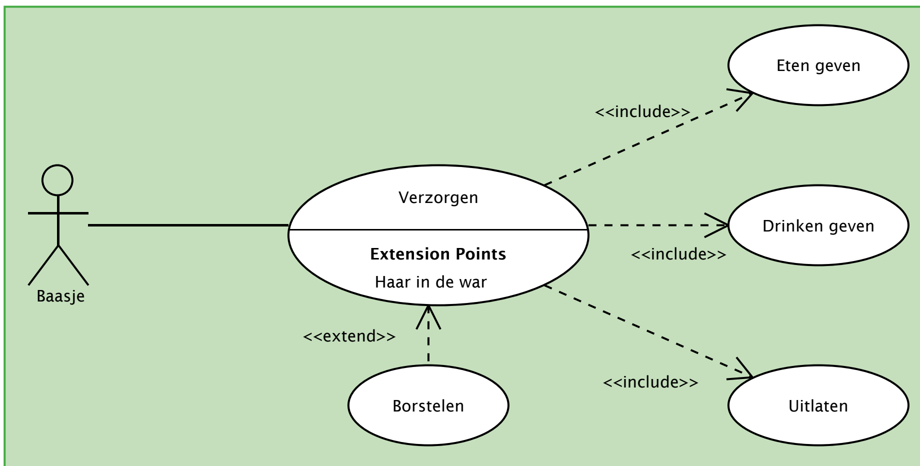
Merk op dat bij extend de pijl naar de use case Verzorgen wijst, en bij include juist in de omgekeerde richting loopt. De reden is dat de gestippelde pijl een afhankelijkheid weergeeft. De uitvoering van Borstelen is afhankelijk van de keuze die je in de use case Verzorgen maakt.

2.3.3 Extension point

Als je wilt kun je de achtergrond van de keuze in de use case Verzorgen duidelijk maken in de vorm van een zogeheten *extension point*. In figuur 2-6 zie je hoe dat gaat: je verdeelt het ovaal van de use case met een lijntje in twee delen, zet in het onderste deel de woorden *Extension Points* (er kan er meer dan één zijn) en daaronder geef je aan wat de reden is voor de uitbreiding. In dit geval: *Haar in de war.*



Figuur 2-5



Figuur 2-6

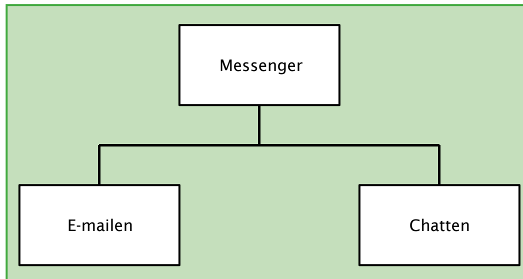
Als er meer dan één uitbreiding is, kun je onder Extension Points een opsomming geven van de verschillende redenen voor de extensies.

2.4 Use cases van Messenger

Het is tijd om naar een echt softwaresysteem te kijken, Windows Live Messenger (MSN Messenger) bijvoorbeeld. Dit is een applicatie die bestaat uit verschillende onderdelen met heel veel functies. Hoe kun je van zo'n systeem de use cases vinden?

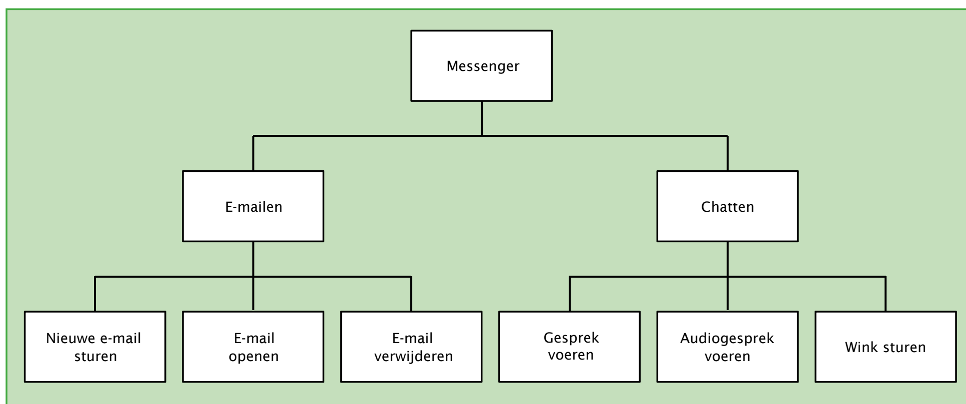
2.4.1 Het vinden van use cases

Als een systeem wat groter en ingewikkelder is, zoals Messenger, dan is het handig eerst een overzicht te maken met de functies van het systeem, van heel globaal naar steeds concreter. In een boomdiagram kun je alles op een overzichtelijke manier voorstellen. Messenger valt in elk geval uiteen in twee grote toepassingsgebieden, mailen en chatten, zie figuur 2-7.



Figuur 2-7

E-mailen is eigenlijk een verzamelnaam voor een reeks doelen die je als gebruiker wilt bereiken, zoals een nieuw bericht sturen, een bericht openen of een bericht verwijderen. In figuur 2-8 zie je een, niet volledig, beeld van de mogelijkheden van Messenger.



Figuur 2-8

Het diagram begint bovenin met de naam van de applicatie *Messenger*, daaronder een grove omschrijving van de onderdelen waaruit het programma bestaat (*E-mailen*, *Chatten*) en verder naar onder staat wat je als gebruiker concreet met het systeem kunt doen (bijvoorbeeld *E-mail openen*, *Wink sturen*). Dus aan de onderkant in het boomdiagram staan de use cases.