Systeemontwikkeling met UML

De visuele modelleertaal Unified Modeling Language (UML) is een gezamenlijk product van een groot aantal bedrijven. Het is een standaard die naar aanleiding van een 'request for proposals' van de Object Management Group (OMG) ontwikkeld is door vooraanstaande personen binnen het objectgeoriënteerde vakgebied.

2.1 Leerdoelen

In dit hoofdstuk leren we uit welke onderdelen UML bestaat en hoe die in de systeemontwikkeling gebruikt kunnen worden. We leren ook dat UML op veel manieren gebruikt kan worden, van heel globaal tot heel gedetailleerd, afhankelijk van de doelstelling. Ook zullen we de samenhang tussen de diagrammen zien, zodat we de diagrammen die in de volgende hoofdstukken behandeld worden goed kunnen plaatsen. Dit hoofdstuk is van belang voor iedereen die bij een project waarin UML wordt gebruikt betrokken is. Met het in dit hoofdstuk beschreven overzicht ben je in staat om de rol van jouw werk en dat van anderen goed binnen de totale context te plaatsen.

2.2 Het ontstaan van UML

Begin jaren negentig kwamen de eerste objectgeoriënteerde systeemontwikkelingsmethoden in de publiciteit. Daaronder waren de door James Rumbaugh en anderen ontwikkelde methode Object Modeling Technique (OMT) [Rumbaugh91] en de methode van Grady Booch [Booch94]. Deze beide methoden waren zeer populair. In 1995 is James Rumbaugh verbonden geraakt aan Rational, het bedrijf waarbinnen Grady Booch zijn Booch-methode op de markt heeft gebracht. Rumbaugh en Booch zijn samen gaan werken om hun beider methoden te integreren tot één methode, de Unified Method genaamd [Booch95]. In 1996 is ook het bedrijf waarin Ivar Jacobson zijn OOSE-methode [Jacobson92] – ook wel Objectory genoemd – heeft ontwikkeld door Rational overgenomen. Ivar Jacobson is bekend vanwege de use- cases die hij in zijn methode gebruikt. Daarmee was duidelijk dat ook aspecten van de OOSE-methode in deze gezamenlijke methode geïntegreerd zouden gaan worden.

Ongeveer gelijktijdig met dit integratieproces kwam de OMG met een verzoek om voorstellen voor een standaard op het gebied van objectgeoriënteerde modellering. De OMG wilde echter geen gestandaardiseerde methode, dat wil zeggen een werkwijze gecombineerd met eisen aan de resultaten van die werkwijze, ze wilde 'slechts' de resultaten van de werkwijze standaardiseren. Op deze wijze kunnen alle ontwikkelaars hun eigen werkwijze gebruiken, maar zijn hun resultaten uitwisselbaar. Rational had intussen contact gezocht met een aantal andere bedrijven. Gezamenlijk werd de Unified Method omgevormd tot een Unified Modeling Language en als voorstel voor een mogelijke standaard bij de OMG ingediend.

In totaal werden zes voorstellen bij de OMG ingediend. Daarvan was het voorstel van Rational het meest uitgewerkt. De andere voorstellen bevatten echter ook waardevolle zaken. Besloten werd dan ook om de voorstellen tot één Unified Modeling Language te integreren. Een van de zaken die aan UML werden toegevoegd is de Object Constraint Language (OCL) [Warmer03], welke deel uitmaakte van het IBM/ObjecTime voorstel. In 1997 is UML de standaard geworden.

In 2006 is de gereviseerde versie 2.1 van deze standaard [OMG06] uitgebracht. Daarna zijn nog enkele versies uitgebracht waarin kleine wijzigingen zijn verwerkt. In dit boek wordt versie 2.3 beschreven.

2.3 Een taal, geen methode

Zoals uit de vorige paragraaf duidelijk is geworden, is UML geen methode, maar een taal. UML standaardiseert geen werkwijze, wel begrippen en diagrammen waarin die begrippen gebruikt kunnen worden. De werkwijze die in dit boek beschreven wordt is een werkwijze die in de praktijk van de auteurs zijn nut heeft bewezen. Omdat de werkwijze geen onderdeel is van de standaard, zal in dit boek de werkwijze strikt gescheiden gehouden worden van aspecten die wel behoren tot de UML-standaard.

2.4 Modeling Maturity Levels

Waarom modelleren we eigenlijk? In de praktijk blijkt dat modellen op meerdere manieren voor verschillende doeleinden gebruikt worden. Om hier orde in te scheppen definiëren we verschillende niveaus van modelleren, Modeling Maturity Levels (MMLs) genoemd. Deze niveaus zijn nuttig om vast te stellen waar iemand zich bevindt en naar welk niveau iemand streeft. We onderscheiden de volgende zes niveaus.

Modeling Maturity Level 0: Geen specificatie

Op niveau nul bevindt de specificatie van de te bouwen software zich geheel in het hoofd van de ontwikkelaar(s). Omdat er niets wordt vastgelegd ontstaan er snel conflicten tussen de gebruiker(s) en ontwikkelaar(s) over wat nu precies de bedoeling is. De programmeur neemt alle beslissingen. Als de oorspronkelijke programmeur vertrekt, gaat alle kennis over het deel van het systeem dat door hem is gebouwd verloren. Feitelijk is op dit niveau geen professionele softwareontwikkeling mogelijk.

Modeling Maturity Level 1: Tekstuele specificaties

De specificatie wordt vastgelegd in één of meer documenten, volledig in natuurlijke taal. De specificatie zorgt ervoor dat de ontwikkelaar en de gebruiker de afspraken expliciet maken. Omdat natuurlijke taal per definitie ambigu is, zal bij een dergelijke specificatie toch veel onduidelijk blijven. De ontwikkelaar zal nog steeds veel beslissingen moeten nemen. Een tweede probleem is het feit dat de specificatie na het wijzigen van de software moeilijk up-to-date te houden is.

Modeling Maturity Level 2: Tekst met diagrammen

De specificatie op dit niveau wordt vastgelegd in één of meer tekstuele documenten, zoals op Modeling Maturity Level 1. Dit wordt echter aangevuld met enkele diagrammen op hoog niveau, vaak om de algemene architectuur van het systeem te verduidelijken. Het voordeel van het gebruik van diagrammen op dit niveau is dat ze helpen om de tekst beter te structureren en beter te kunnen begrijpen. De diagrammen worden hier voornamelijk gebruikt als communicatiemiddel. Alle nadelen van het eerste niveau gelden hier nog steeds.

Modeling Maturity Level 3: Modellen met tekst

Op niveau drie wordt de specificatie vastgelegd in een aantal modellen. Met model wordt hier bedoeld: een diagram of tekst geschreven in een niet-natuurlijke taal waarvan de betekenis helder en eenduidig is. UML is een voorbeeld van zo'n taal. Andere voorbeelden zijn: Z [Wordsworth92], SDL [Ellsberger97], maar zelfs programmeertalen kunnen binnen deze definitie als specificatietaal worden beschouwd. De broncode is een specifieke, niet ambigue specificatie van het systeem, zij het dat broncode meestal minder geschikt is om mensen begrip te geven van de werking van het systeem.

Bijbehorende tekstdocumenten beschrijven de details en de motivatie achter de modellen. Op dit niveau zijn de modellen een afspiegeling van het systeem. De modellen worden, meestal met de hand, omgeschreven naar code. Omdat de modellen een preciezere specificatie vormen dan de tekst op niveau twee zal de programmeur veel minder onduidelijkheden tegenkomen en hoeft hij derhalve minder eigen beslissingen te nemen.

Modeling Maturity Level 4: Exacte modellen

Op niveau vier vormen de modellen het belangrijkste onderdeel van de specificatie van een systeem. Op dit niveau hebben de modellen een dergelijke samenhang dat men ook zou kunnen spreken van één enkel model dat bestaat uit een aantal verschillende onderdelen. Zo bestaat een UML-specificatie uit verschillende diagrammen. Elk diagram kan beschouwd worden als een apart model, maar als het onderlinge verband tussen de diagrammen groot genoeg is vormt het geheel één model.

De modellen op dit niveau zijn precies genoeg om een directe link te hebben met de code. Tekstdocumenten worden nog steeds gebruikt, maar vormen nu uitleg bij de modellen en beschrijven de motivatie achter gemaakte keuzes. Op dit niveau hoeven programmeurs maar weinig eigen beslissingen te nemen. Omdat automatische generatie van belangrijke delen van de code mogelijk is, kunnen de modellen en de code up-to-date gehouden worden. Om dezelfde reden is het op dit niveau mogelijk om sterk iteratief te werken.

Modeling Maturity Level 5: Alleen modellen

Op niveau vijf zijn de modellen volledig en precies genoeg om alle details te beschrijven. De code kan geheel uit de modellen gegenereerd worden. Net zoals Assembler vandaag gebruikt wordt, kan de code in principe geheel onzichtbaar blijven voor de ontwikkelaar. Op dit niveau is de modelleertaal feitelijk een hoger niveau programmeertaal geworden. Vandaag de dag is dit niveau helaas nog onbereikbaar, met uitzondering van bijzondere, beperkte domeinspecifieke omgevingen.

Het ambitieniveau van dit boek is om UML te kunnen gebruiken op niveaus drie en vier. Op de niveaus nul en één worden geen modellen gebruikt. De diagrammen op niveau twee hebben een hoog 'plaatjes'-gehalte. Het zijn slechts schetsen zonder een duidelijke en precieze betekenis. Niveau vijf is op dit moment nog niet haalbaar. Er

zijn (nog) geen algemeen bruikbare specificatietalen waarin alle delen van een softwaresysteem in voldoende detail beschreven kunnen worden.

2.5 De diagrammen en hun samenhang

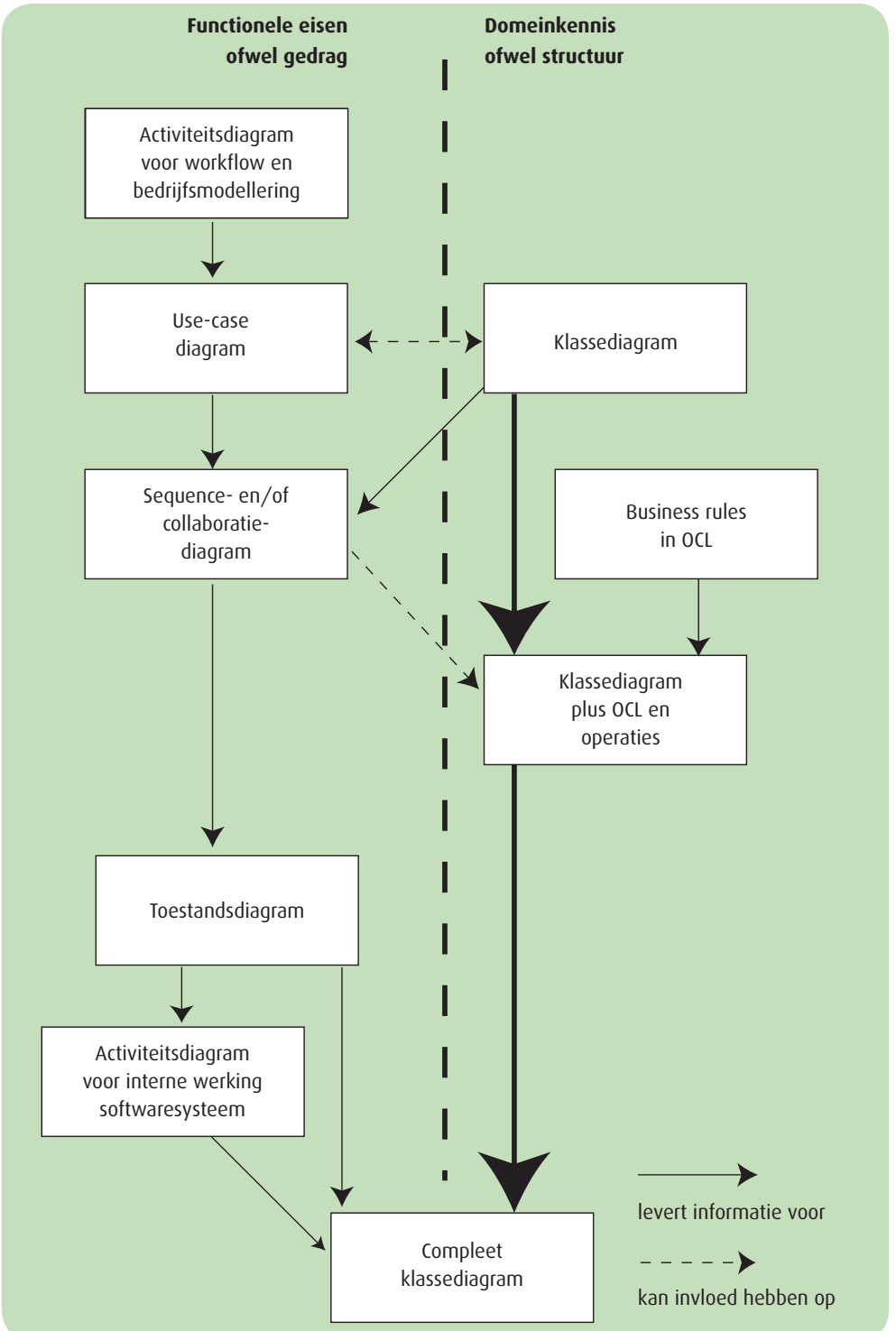
UML biedt een aantal diagrammen die gezamenlijk de specificatie van het softwaresysteem vormen.

- Het use-case-diagram toont hoe het systeem kan worden gebruikt door externe entiteiten zoals menselijke gebruikers.
- Het klassediagram (Eng. class diagram) toont de statische structuur van het softwaresysteem weergegeven als klassen en hun relaties.
- Het objectdiagram toont de statische structuur van het softwaresysteem weergegeven als objecten en hun relaties.
- Het sequence-diagram toont de volgorde in tijd van de boodschappen die in het systeem verstuurd en ontvangen worden.
- Het communicatiediagram toont hoe de objecten samenwerken om een doel te bereiken.
- Het toestandsdiagram toont de toestanden waarin een object zich kan bevinden gedurende zijn levensloop.
- Het activiteitsdiagram toont de activiteiten die door een deel van het systeem worden uitgevoerd, inclusief eventueel parallellisme.
- Activiteitsdiagrammen kunnen ook gebruikt worden voor het beschrijven van bedrijfsprocessen en de workflow.
- Het componentdiagram toont de verdeling van het gehele systeem in componenten en de relaties tussen die componenten.
- Het deployment-diagram toont hoe de softwarecomponenten in een bepaalde systeemconfiguratie worden gebruikt.

Het use-case-diagram wordt ook wel een requirements-diagram genoemd. Het klassediagram en het objectdiagram vormen samen de statische diagrammen. Het sequence-, communicatie-, toestands- en activiteitsdiagram worden ook wel de dynamische diagrammen genoemd. Het component- en deployment-diagram zijn de implementatiediagrammen.

Naast de diagrammen biedt de UML-standaard een taal waarin beperkingen, condities en regels die gelden in de diagrammen weergegeven kunnen worden. Deze taal heet de Object Constraint Language (OCL). Hoofdstuk 6 gaat in op het gebruik van deze taal.

Figuur 2-1 geeft een overzicht van de samenhang van de verschillende diagrammen. Een pijl van het ene diagram naar het andere geeft aan dat het tweede diagram informatie uit het eerste diagram nodig heeft. Een gestippelde pijl betekent dat het eerste diagram invloed heeft op het tweede, zonder dat delen van het tweede diagram direct uit het eerste voortkomen. In het midden van de figuur zie je een dikke pijl. Het klassediagram (in verschillende stadia) dat in het traject langs die pijl wordt ontwikkeld, is het centrale diagram in de methode.



Figuur 2-1 Samenhang van de diagrammen

In figuur 2-1 wordt een onderscheid gemaakt tussen structurele en gedragsaspecten van het softwaresysteem. De structurele aspecten worden weergegeven in het klassediagram, de gedragsaspecten in de overige diagrammen, met uitzondering van de component- en deployment-diagrammen. Uiteindelijk worden beide zijden geïntegreerd in het klassediagram.

2.6 Diagrammen in verschillende fases

Alle diagrammen kunnen op elk moment tijdens het systeemontwikkelingstraject gemaakt worden. In de praktijk zullen veel van de diagrammen 'meegroeien' tijdens de ontwikkeling. Soms kan het echter nodig zijn om een onderscheid te maken tussen de verschillende stadia van ontwikkeling waarin de diagrammen zich kunnen bevinden. Hiertoe worden de voorvoegsels domein, applicatie en implementatie gebruikt.

- Een *domeinmodel of -diagram* (Eng. domain of conceptual model) is een model of diagram waarin alleen aspecten naar voren komen uit de werkelijkheid waarin het beoogde systeem moet gaan werken (het domein), en geen automatiseringsaspecten.
- Een *applicatiemodel of -diagram* (Eng. application model) is een model of diagram waarin naast de domeinaspecten ook keuzen naar voren komen die betrekking hebben op hoe de gebruiker met dit specifieke systeem kan (gaan) werken. Een applicatiemodel kan gezien worden als een uitbreiding op het domeinmodel, maar in principe zijn beide verschillende ontwikkelstadia van een en hetzelfde model.
- Een *implementatiemodel of -diagram* (Eng. implementation model) is een model dat exact weergeeft hoe de implementatie van het systeem eruitziet of zal gaan zien. Een implementatiemodel bevat alle aspecten die naar voren zijn gekomen in de vorige stadia: in het domeinmodel en in het applicatiemodel. Daarnaast bevat een implementatiemodel aspecten die alleen betrekking hebben op de wijze van implementeren die gekozen is, zoals gebruik van opslagstructuren. In het implementatiemodel worden al deze aspecten zo uitgewerkt dat deze direct kunnen worden omgezet naar programmacode.

In bijvoorbeeld het *domeinklassediagram* voor een luchthavenbeheersysteem zullen vliegtuigen, landingsbanen, vluchtschema's en dergelijke voorkomen, maar nog geen informatie over de nodige views op de vluchtschema's. Zodra die informatie aan het objectmodel wordt toegevoegd noemen we het een *applicatieklassediagram*.

Het is lang niet altijd zinvol om van elk diagram deze drie verschillende versies te maken. Tabel 2-1 laat zien welke diagrammen in welke stadia zinvol zijn.

	domein	applicatie	implementatie
use-case-diagram	nee	ja	nee
klassediagram	ja	ja	ja
objectdiagram	ja	ja	ja
sequence-diagram	beperkt	ja	ja
communicatiediagram	beperkt	ja	ja
toestandsdiagram	ja	ja	ja
activiteitsdiagram	ja, workflow	ja	ja
deployment-diagram	nee	nee	ja
componentdiagram	nee	nee	ja

Tabel 2-1 Overzicht van diagrammen in verschillende stadia

2.7 Projectfasering

Binnen projecten wordt vaak een fasering gebruikt. Hierin komen vaak de volgende fases voor: conceptualisatie- of requirementsfase, analyse, ontwerp, implementatie, en daarna testen. In de 'traditionele' watervalmethode worden deze fases sequentieel, dus na elkaar, uitgevoerd. In iteratieve werkwijzen worden in verschillende iteraties kleine watervalletjes uitgevoerd. Ook hierin worden de verschillende fases vaak onderscheiden. (Zie § 2.8.) Het is van belang om bij ieder diagram duidelijk te weten voor welk doel het gebruikt wordt.

2.7.1 Conceptualisatie

Het use-case-diagram speelt in de conceptualisatiefase een belangrijke rol. Veelvuldige communicatie met de eindgebruikers en andere betrokkenen is noodzakelijk om een goed beeld te krijgen van de taken die de eindgebruikers met het systeem moeten gaan uitvoeren. De use-cases vormen een gestructureerde beschrijving van deze taken in natuurlijke taal. Use-cases worden beschreven in hoofdstuk 8.

2.7.2 Analyse

Tijdens de analysefase zijn het klassediagram en het sequence-diagram (of het equivalente communicatiediagram) het belangrijkst. Alle termen die in de diagrammen gebruikt worden komen uit het *domein*, dat wil zeggen dat stukje werkelijkheid waarbinnen het softwaresysteem een rol speelt. Hiertoe zien we ieder object als afspiegeling van hetzij een fysiek ding uit de werkelijkheid, hetzij een concept dat gangbaar is in die werkelijkheid. Fysieke objecten zijn over het algemeen duidelijk te onderscheiden: tafel, stoel, auto, bal, enzovoorts. Omdat de objecten in het model een directe afspiegeling van dingen en concepten uit de werkelijkheid zijn, is een dergelijk model te begrijpen door niet-automatiseerders. De diagrammen moeten dan ook worden besproken met de domeinexperts. Domeinexperts en automatiseerders

dienen het er samen over eens te worden dat de diagrammen een goede weergave van het relevante deel van de werkelijkheid zijn. Wanneer het softwaresysteem omvangrijk is, kan het al in deze fase worden opgedeeld in componenten. Het component- en het deployment-diagram kunnen gebruikt worden om deze verdeling weer te geven. Het klassediagram wordt beschreven in hoofdstuk 4. Het sequencediagram wordt beschreven in hoofdstuk 10. Het componentdiagram en het deployment-diagram worden beschreven in hoofdstuk 14.

2.7.3 Ontwerp

Pas in de ontwerpfase is de aandacht gericht op wat veel mensen beschouwen als het 'echte' automatiseringswerk. Alle niet-functionele eisen gaan een rol spelen. Naast aanpassingen aan het klassediagram en het sequence-diagram zullen nu de toestands- en activiteitsdiagrammen gemaakt worden. Ook zullen nu eventuele rolmodellen met behulp van communicatiediagrammen gemaakt worden. Een punt van aandacht tijdens deze fase is de consistentie tussen de diagrammen.

Essentiële principes uit de software-engineering zijn het uitgangspunt bij het ontwerp. Een object verbergt zijn binnenkant (hoe worden operaties uitgevoerd, wat wordt er precies opgeslagen en hoe, enzovoorts) voor alle andere objecten in het systeem. Dit principe van *inkapseling* (Eng. encapsulation) is een welhaast perfecte vorm van *information hiding*. Het belang van information hiding staat, los van de gebruikte technieken, buiten kijf. Ieder modern boek over software-engineering zal dat direct beamen.

Traditioneel blijken aanpassingen in software zeer moeilijk. Zelfs kleine wijzigingen van functionaliteit veroorzaken vaak een domino-effect: één kleine wijziging op één plaats in de software heeft vele, vaak ongewenste en onverwachte neveneffecten op grote delen van het systeem. Door middel van inkapseling is het mogelijk om aanpassingen in de software in belangrijke mate te lokaliseren. Aanpassen betekent dus minder werk, maar vooral ook minder kans op het introduceren van fouten door ongewenste neveneffecten. Hierdoor blijft de kwaliteit van de software, ook na herhaaldelijk aanpassen, op een hoog peil. Een systeem dat eenvoudig en met minder moeite aan te passen is, zorgt voor flexibiliteit. Het kan snel genoeg meeveranderen met de behoeften van de organisatie en gebruikers.

Uiteindelijk wordt alle informatie teruggebracht in het klassediagram. Het klassediagram is een complete specificatie van (de interface van) alle objecten. Het communicatiediagram wordt beschreven in hoofdstuk 10. Het toestandsdiagram wordt beschreven in hoofdstuk 12; het activiteitsdiagram in hoofdstuk 15.

2.7.4 Implementatie

In deze fase worden alle objecten die in het ontwerp gespecificeerd zijn geprogrammeerd. Uitgangspunt hierbij is het klassediagram. Als we een objectgeoriënteerde taal ter beschikking hebben is het werk in deze fase eenvoudigweg de informatie in

dit diagram rechttoe rechtaan uitschrijven, hetgeen resulteert in code. Het opdelen in componenten van een omvangrijk softwaresysteem kan ook in deze fase plaatsvinden.

2.8 Projectmanagement

Er zijn verschillende manieren waarop een project georganiseerd kan worden. Traditioneel wordt vaak een watervalmodel gebruikt, terwijl tegenwoordig iteratieve of agile methoden steeds meer gebruikt worden.

2.8.1 Watervalontwikkeling

Traditioneel worden projecten vaak uitgevoerd volgens het zogenaamde *watervalmodel*. Een project wordt, zoals hiervoor beschreven, in fases opgedeeld. De fases worden sequentieel na elkaar uitgevoerd. Bekende nadelen van deze aanpak zijn de vaak lastige overdracht tussen de verschillende fases en de lange tijd die het duurt voordat er een systeem is gebouwd dat de gebruiker kan testen. Systeemontwikkeling met UML biedt echter ook andere mogelijkheden.

2.8.2 Incrementele en iteratieve ontwikkeling

Het is aan te raden om bij een objectgeoriënteerde werkwijze ook gebruik te maken van een *incrementeel* en *iteratief* ontwikkeltraject. Incrementeel wil zeggen dat telkens een klein deel van het gehele systeem gebouwd wordt. Ieder increment bouwt voort op datgene wat al bestaat. Iteratief wil zeggen dat de 'traditionele' fasen herhaaldelijk doorlopen worden. Na implementatie en testen worden opnieuw de systeemeisen bijgesteld, een nieuwe analyse gemaakt, enzovoorts.

Een incrementeel en iteratief ontwikkeltraject kan worden opgedeeld in kleinere tijdspannen, die in het Engels wel *time-boxes* worden genoemd. Deze manier deelt het gehele traject dus op in een aantal time-boxes. De kosten en de einddatum van de time-box liggen altijd vast; wat kan variëren is hoeveel van het systeem binnen die tijd geïmplementeerd wordt. Elke time-box kan wel worden opgedeeld in een conceptualisatiefase, analysefase, ontwerpfase enzovoorts. Let er dan wel op dat de time-box (en dus het increment) klein genoeg is hiervoor.

Vanaf het moment dat de eerste time-box is afgelopen is er dus een werkend systeem. Dit systeem kan heel erg klein zijn, met zeer beperkte functionaliteit. Van belang is echter dat het werkt en dat het compleet is, met alle documentatie, en dat de gebruiker het al kan uitproberen. Elke volgende time-box werkt nu verder met de basis die in de eerste time-box gelegd is. Aan het eind van een time-box wordt vaak een beslismoment ingelast. Op dit moment wordt het traject geëvalueerd en worden beslissingen genomen over de vervolgstappen. Ook is het mogelijk om niet verder te gaan met het project. Een bruikbaar, werkend systeem is het resultaat, onafhankelijk van deze beslissing.

Een eerste increment kost vanwege een initiële aanlooptijd gemiddeld drie maanden, volgende incrementen duren ongeveer anderhalf tot twee maanden.