Unified Modeling Language

Een overzicht

Danny Greefhorst
Matthijs Maat

19 december 1997

Software Engineering Research Centre

Stichting SERC
Postbus 424, 3500 AK Utrecht
Telefoon: (030) 2545412 Fax: (030) 2545948
E-mail: info@serc.nl WWW: http://www.serc.nl

Inhoudsopgave

1 Inleiding

Geen serieus bouwwerk zonder bouwtekening, geen werkende chip zonder schema; het belang van het maken van ontwerpmodellen bij de ontwikkeling van complexe constructies is in de meeste vakgebieden evident. Software-ontwikkeling vormt hierop geen uitzondering, het hebben van een goede 'bouwtekening' is essentieel voor het welslagen van de constructie van een goed software-systeem. Modellen van software-systemen maken het mogelijk bepaalde onderdelen of aspecten van complexe systemen afzonderlijk te beschouwen en te begrijpen, modellen zijn essentieel bij de communicatie tussen software-ontwikkelaars en domeinexperts of opdrachtgevers, modellen zijn onmisbaar voor communicatie tussen software-ontwikkelaars onderling, kortom: geen goede software zonder goed model!

Dat er gemodelleerd moet worden is intussen algemeen aanvaard. Hoe er gemodelleerd moet worden is een heel andere vraag. Met name met de opkomst van object-oriëntatie zijn er tal van ontwikkelmethoden ontstaan, met als belangrijkste exponenten Rumbaugh's *Object Modeling Technique* [Rumbaugh91] en *Booch* [Booch94]. Hoewel de verschillende methoden elkaar weinig ontlopen wat expressiviteit en gemodelleerde concepten betreft, heeft ieder zijn eigen notatie, een eigen procesbeschrijving en eigen ondersteunende tools. De keuze voor een bepaalde methode is dus een belangrijke: de geboorte van een heuse 'method war' is daar. Worden het de wolkjes van Booch of de rechthoeken van OMT?

Een software-model wordt uitgedrukt in een bepaalde *modelleringstaal*. Het veelvoud aan beschikbare talen bemoeilijkt echter het aanleren van modellering en is bron van irritatie voor meer ervaren ontwikkelaars; communicatie tussen ontwikkelaars wordt onnodig ingewikkeld door verschillen in notatie. In 1994 nemen James Rumbaugh en Grady Booch het initiatief tot samenvoegen van hun ontwikkelmethoden — te beginnen met de bijbehorende talen. Nadat Ivar Jacobson (de man achter de *OOSE/Objectory* ontwikkelmethoden [Jacobson94]) zich bij hen heeft aangesloten ontstaat het skelet van een algemene modelleringstaal, begin 1997 uiteindelijk resulterend in de *Unified Modeling Language version 1.0* (UML). Hoewel sterk gebaseerd op OMT, Booch en OOSE, is met UML geprobeerd de 'best practice' uit de gehele modelleringswereld te combineren.

UML, inmiddels gevorderd tot versie 1.1, wordt door velen gezien als de modelleringstaal voor de toekomst. Brede ondersteuning uit de industrie en de acceptatie van UML als standaard door de *Object Management Group* (OMG) zorgen ervoor dat UML inderdaad een goede kans maakt een einde te maken aan de Babylonische spraakverwarring binnen de software-modellering. Dit artikel geeft een overzicht van zowel de taal UML zelf, als het gebruik van UML binnen het software-ontwikkelingsproces. Het artikel is niet bedoeld als volledige handleiding, maar biedt een algemene indruk van de notatie en de mogelijkheden tot gebruik van UML.

Ter introductie wordt in het artikel eerst een korte beschrijving gegeven van de context en de ontstaansgeschiedenis van UML. UML is een volgende stap in de evolutie van object-georiënteerde analyse- en ontwerpmethoden en kan dus het best bezien worden in relatie tot deze methoden. Vervolgens wordt UML zelf nader belicht; de elementen van de taal en de mogelijkheden om UML uit te breiden voor specifieke domeinen komen in afzonderlijke hoofdstukken aan bod. UML is een taal, géén methode; het gebruik van UML binnen ontwikkelmethoden en, meer specifiek, het gebruik van UML bij het ontwerpen van databases wordt beschreven in de twee volgende hoofdstukken. Het artikel wordt afgesloten met een samenvatting van de belangrijkste punten en een korte blik in de toekomst van UML.

UML is een object-georiënteerde modelleringstaal; de lezer wordt bekend verondersteld met de belangrijkste concepten uit het object-georiënteerde paradigma. In bijvoorbeeld [Troyer93] en [Florijn95] kan meer informatie en uitleg gevonden worden over object-oriëntatie.

2 Analyse, Ontwerp en Methoden

2.1 Object-georiënteerde Analyse en Ontwerp

In software-ontwikkeling, object-georiënteerd of niet, is een zestal verschillende fasen te onderkennen. Dit zijn achtereenvolgens de requirements-analysefase, de analysefase, de ontwerpfase, de implementatiefase, de testfase en exploitatiefase.

Het ontwikkelen van een software-systeem vindt vrijwel altijd plaats om een bepaald probleem op te lossen. In de eerste fase van het ontwikkelingsproces, de *requirements-analysefase*, wordt dit probleem geïnventariseerd vanuit het gezichtspunt van de gebruiker. Hierbij kunnen gebruikersscenario's een belangrijke rol spelen. Er wordt in principe nog niet gekeken naar hoe een concrete invulling van de functionaliteit gegeven zou kunnen worden. In de volgende fase binnen het ontwikkelingsproces, de *analysefase*, wordt het probleem geabstraheerd naar modellen, resulterend in een precieze en eenduidige probleemspecificatie. In deze modellen worden onder meer de relevante klassen, objecten en hun onderlinge relaties geïnventariseerd, eventueel gebruik makend van kennis van soortgelijke problemen. Een weergave van het probleem in de vorm van modellen vindt plaats aan de hand van een bepaalde notatietechniek.

In de *ontwerpfase* wordt de initiële probleemdefinitie verder gedetailleerd en verschuift de nadruk meer naar het definiëren van een oplossing voor het probleem. Hierbij zullen oplossingsspecifieke constructies in het model worden geïntroduceerd. Voor de beschrijving van het ontwerp kunnen over het algemeen dezelfde modellen worden gebruikt als in de analysefase, waardoor een vertaling van het analysemodel naar het ontwerpmodel voorkomen kan worden. Het resultaat van de ontwerpfase is een gedetailleerde beschrijving van het systeem die gebruikt kan worden in volgende fasen van het ontwikkelingsproces. In de *implementatiefase* worden de modellen uit de ontwerpfase omgezet in daadwerkelijke code. Indien het ontwerp gedetailleerd is uitgewerkt kan deze fase relatief eenvoudig zijn.

Na het implementeren van het systeem dient er een *testfase* plaats te vinden. Idealiter zal een groot deel van de activiteiten uit deze fase al gedurende de analyse-, ontwerp- en implementatiefasen kunnen plaatsvinden. In de laatste fase uit het ontwikkelingsproces, de *exploitatiefase*, wordt het systeem uiteindelijk daadwerkelijk in gebruik genomen en dient het systeem zowel technisch als functioneel te worden beheerd.

Bovenstaande zes fasen spelen ook een belangrijke rol bij object-georiënteerde software-ontwikkeling. Bij object-georiënteerde ontwikkelmethoden ligt de nadruk vaak vooral op de

eerste twee fasen van het ontwikkelingsproces waarvoor ook wel de term *object-georiënteerde analyse en ontwerp* (meestal aangeduid als OOA&D) wordt gebruikt.

2.2 Analyse- en Ontwerpmethoden

De resultaten uit de verschillende fasen van het ontwikkelingsproces worden vastgelegd in *modellen*. Een model wordt uitgedrukt in uit een bepaalde notatie die het mogelijk maakt de begrippen en concepten die tijdens het ontwikkelproces gebruikt worden visueel weer te geven. De verschillende onderdelen van een notatie hebben daarbij ieder een eigen semantiek die aangeeft wat de notatie precies betekent. In het verleden waren modellen vaak sterk gekoppeld aan specifieke object-georiënteerde *methoden*; een methode beschrijft hierbij de precieze invulling van de zes fasen uit het software-ontwikkelingsproces en bestaat uit modellen en een proces. Het proces beschrijft de te volgen stappen en de onderlinge volgorde van deze stappen.

Drie belangrijke en veel gebruikte object-georiënteerde analyse- en ontwerpmethoden zijn:

- **OMT** — De *Object Modeling Technique* (OMT) is een mede door James Rumbaugh [Rumbaugh91] ontwikkelde analyse- en ontwerpmethode. De nadruk ligt bij deze methode op het beschrijven van de statische structuur van een software-systeem, uitgedrukt in een zogenaamd Object Model. In dit Object Model zijn klassen (en objecten) en hun onderlinge relaties weergegeven. De dynamische aspecten van een systeem kunnen in OMT worden beschreven in het Dynamic Model met behulp van statediagrammen. In deze diagrammen worden de verschillende mogelijke toestanden van een object en de overgangen tussen deze toestanden beschreven. De functionele eigenschappen tenslotte worden uitgedrukt in het Functional Model met behulp van data-flowdiagrammen. Hierbij staan de gegevens die worden uitgewisseld tussen objecten centraal.
- **Booch** — Deze door Grady Booch [Booch94] beschreven methode is beschikbaar in een aantal verschillende versies. Booch beschrijft een systeem als een aantal views, waarbij elke view wordt beschreven door een aantal modellen. De Booch methode is erg uitgebreid, maar de gebruikte notatie (de beroemde wolkjes) wordt door een groot aantal gebruikers als weinig praktisch ervaren. De methode bevat een incrementeel en iteratief proces waarmee een systeem zowel op een hoog abstractieniveau als op een zeer gedetailleerd niveau kan worden geanalyseerd
- **OOSE** — De in 1994 door Ivar Jacobson [Jacobson94] beschreven OOSE-methode, waarbij *OOSE* staat voor *Object-Oriented Software Engineering*, stelt use-cases centraal. Use-cases beschrijven de initiële requirements van een systeem vanuit het gezichtspunt van een externe gebruiker. De use-cases worden in alle fasen van het ontwikkelingsproces gebruikt, van de implementatiefase tot en met de testfase.

Ondanks sterke overeenkomsten zijn de verschillende methoden niet direct onderling uitwisselbaar aangezien de resulterende modellen van elkaar verschillen. Dit kan bij organisaties leiden tot frustraties: ontwikkelaars moeten beschikken over kennis van verschillende methoden, hetgeen zeker een remmende factor is bij toepassing van object-georiënteerde software-ontwikkeling en modellering. Ook de ontwikkeling van CASE-tools wordt hier negatief door beïnvloed doordat dergelijke tools meerdere, op zich sterk op elkaar gelijkende, modellen en bijbehorende notatiewijzen moeten ondersteunen. Er is dus duidelijk behoefte aan één standaard modelleringstaal.

2.3 Unified Modeling Language

UML is bedoeld om een oplossing te bieden voor de diversiteit aan beschikbare modellen en bijbehorende notatiewijzen; de taal biedt een standaard notatie met bijbehorende semantiek voor allerlei aspecten die een rol spelen bij het modelleren van object-georiënteerde systemen. De notatie is ontstaan door het beste van de huidige notatiewijzen met elkaar te combineren. Hierbij is getracht zo volledig mogelijk te zijn; indien iets niet direct uitgedrukt kan worden in UML dan kan er gebruik gemaakt worden van de speciaal ingebouwde uitbreidingsmogelijkheden. UML bestaat in principe alleen uit een notatie; het ontwikkelingsproces zelf is geen onderdeel van de specificatie. Dit betekent dat UML in principe gebruikt kan worden als notatie voor allerlei bestaande methoden. Meer over het gebruik van UML binnen ontwikkelingsmethoden is te vinden in hoofdstuk 5.

UML is ontstaan in 1994 toen James Rumbaugh en Grady Booch van de Rational Software Corporation besloten tot integratie van hun methoden, OMT en Booch, tot de Unified Method. Een eerste versie van deze methode, versie 0.8, kwam in oktober 1995 beschikbaar. Eind 1995 sloot ook Ivar Jacobson van Objectory zich aan bij de Unified Method, wat resulteerde in een integratie van zijn OOSE methode. Resultaat van deze inspanningen was versie 0.9 van de Unified Method welke in Juni 1996 beschikbaar kwam.

In de loop van 1996 werd duidelijk dat een aantal bedrijven de Unified Method van strategisch belang achtten voor hun organisatie. De Object Management Group (OMG) maakte daarop kenbaar voorstellen te willen voor een standaard object-georiënteerde notatie, voor Rational de directe aanleiding om de Unified Method als notatiestandaard in te dienen. Het procesaspect van de Unified Method werd hierbij niet in het standaardisatieproces betrokken, maar zou bij Rational wel verder ontwikkeld worden tot de Objectory methode. In samenwerking met een groot aantal andere bedrijven waaronder DEC, HP, IntelliCorp, IBM, Microsoft, TI, Oracle en Unisys vormde Rational het UML Partners consortium. Dit leidde in Januari 1997 tot UML 1.0.

In Januari 1997 bleken ook anderen een voorstel voor een notatiestandaard bij de OMG te hebben ingediend, waaronder IBM & ObjecTime, Platinum, Ptech, Taskon & Reich en Softeam. In samenwerking met deze bedrijven is uiteindelijk de UML 1.1 standaard

opgesteld welke sinds september 1997 publiekelijk beschikbaar is en sinds kort door de OMG ook officieel als standaard wordt erkend.

De huidige UML-standaard bevat een groot aantal zaken die direct ontleend zijn aan technieken ontwikkeld bij de verschillende UML-partners. Zo heeft IBM bijvoorbeeld de *Object Constraint Language* (OCL) ingebracht, onder andere gebruikt voor het beschrijven van UML zelf en geschikt voor het opleggen van beperkingen (*constraints*) aan modellen, heeft i-Logix bijgedragen met de *Statecharts* van David Harel en heeft HP de kennis van de *Fusion-methode* toegevoegd.

3 Overzicht van de UML Notatie

3.1 Inleiding

De resultaten uit de verschillende fasen van het ontwikkelingsproces, beschreven in het vorige hoofdstuk, worden vastgelegd in modellen. Een software-model wordt uitgedrukt in een *modelleringstaal*. Een dergelijke taal moet een aantal zaken omvatten [Rational97b]:

- **Model-elementen** — Definitie van fundamentele concepten die in het model weergegeven kunnen worden.
- **Notatie** — Grafische weergave van de model-elementen.
- **Vuistregels** — Beschrijving van het gebruik van de taal binnen het ontwikkelingsproces.

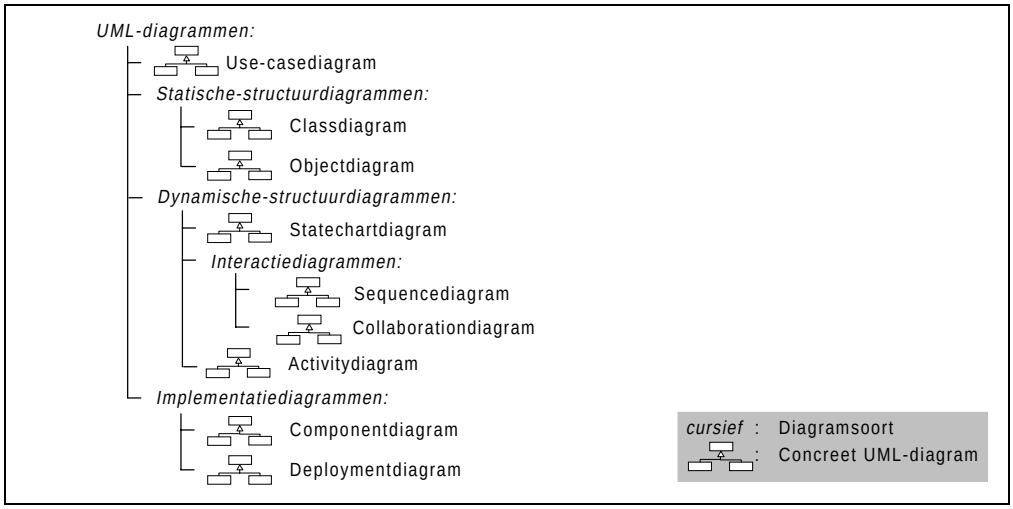
UML biedt deze zaken — en méér. De UML-specificatie beschrijft namelijk niet alleen een notatie, maar ook een *metamodel*: een formeel model van UML, opgesteld in UML. Het UML-metamodel, hoewel interessant, wordt in dit artikel verder buiten beschouwing gelaten.

3.1.1 Views, Diagrammen en Elementen

De taal UML bestaat uit vier verschillende onderdelen, te weten *views*, *diagrammen*, *model-elementen* en *algemene mechanismen* [Eriksson98]. Een *view* laat één bepaald aspect van het te modelleren systeem zien. Een view is zelf geen grafische component, maar een abstractie bestaande uit een aantal diagrammen. Door het combineren van de verschillende views, ieder dus gericht op een bepaald aspect, kan een complete beschrijving van een systeem worden verkregen. In [Eriksson98] worden de volgende UML views gedefinieerd:

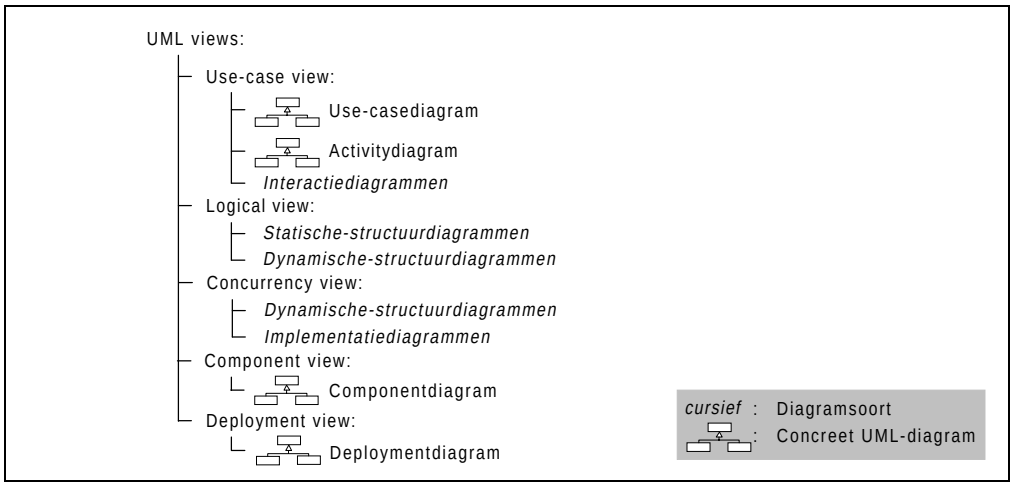
- **Use-case view** — De functionaliteit van het systeem zoals gezien door de ogen van externe actoren.
- **Logical view** — De representatie van de functionaliteit binnen het systeem, in termen van de statische structuur en het dynamische gedrag van het systeem.
- **Component view** — De organisatie van de verschillende codecomponenten.
- **Concurrency view** — Weergave van parallellisme in het systeem, toegespitst op aanwezige problemen met communicatie en synchronisatie.
- **Deployment view** — Toepassing van het systeem op de fysieke architectuur

Diagrammen zijn de grafische weergave van de inhoud van een bepaalde view. Er zijn in UML negen soorten diagrammen die in verschillende combinaties gebruikt kunnen worden om de verschillende views te beschrijven. Figuur 1 geeft de verschillende UML-diagrammen weer. De grafische elementen in de diagrammen, de *model-elementen*, zijn uiteindelijk de visualisatie van de te modelleren object-georiënteerde concepten en de relaties daartussen.



Figuur 1 UML-diagrammen

De verschillende UML-diagrammen zijn ieder meer of minder geschikt voor het modelleren van een bepaald aspect van een systeem. Welke diagrammen in principe geschikt zijn om welke views te beschrijven is te zien in figuur 2.



Figuur 2 Een UML-model: views en diagrammen

Naast views, diagrammen en elementen zijn er in UML ook enkele *algemene mechanismen* gedefinieerd. Deze mechanismen voegen extra commentaar, informatie of semantiek toe aan model-elementen.

Dit hoofdstuk biedt een overzicht van de UML-notatie. In de volgende paragraaf worden daartoe eerst nog een aantal meer algemene mechanismen geïntroduceerd. Vervolgens wordt elk van de negen soorten diagrammen besproken, zowel wat betreft notatie als het typisch gebruik. Model-elementen worden niet apart behandeld, maar geïntroduceerd daar waar nodig. Voor een meer complete beschrijving van UML wordt verwezen naar [Rational97b], [Rumbaugh98] en [Eriksson98].

3.1.2 Algemene Mechanismen in UML

UML bevat een aantal algemene mechanismen die in alle diagrammen gebruikt kunnen worden. Deze mechanismen voegen extra informatie of semantiek toe aan model-elementen:

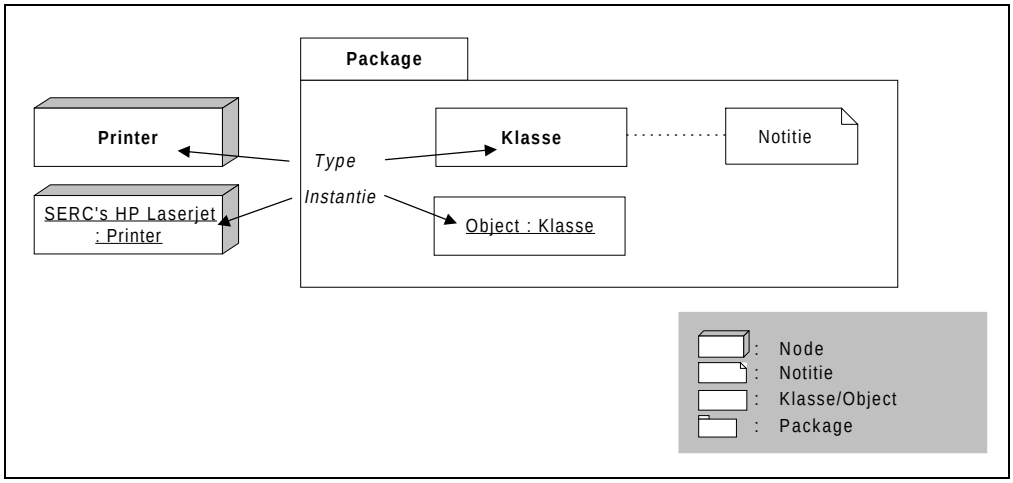
- **Decoraties** — *Decoraties* worden gebruikt om extra semantiek toe te voegen aan model-elementen. Een voorbeeld van het gebruik van zo'n decoratie is de manier om in UML onderscheid te kunnen maken tussen types van model-elementen en instanties van deze types. Zo worden in principe zowel *klassen* als *objecten* in UML gerepresenteerd met eenzelfde rechthoek; door de naam van het element dik gedrukt weer te geven kan worden aangegeven dat het om een klasse gaat, een onderstreepte naam geeft aan dat het gaat om een instantie (een object dus). Hetzelfde mechanisme, dik gedrukt of onderstreept, kan gebruikt worden om onderscheid te maken tussen een *node-type* (een type van een fysiek apparaat) en een concrete instantie van een dergelijk type. Het gebruik van decoraties is weergegeven in figuur 3.
- **Notities** — Hoe uitgebreid de UML-definitie ook is, er is altijd wel informatie die niet direct middels een standaard element duidelijk weer te geven is. Voor dit soort extra informatie zijn er *notities*: elementen die overal in elk diagram geplaatst mogen worden en die willekeurige tekst mogen bevatten.
- **Specificaties** — Er is een grens aan de hoeveelheid informatie die in een diagram bij model-elementen weergegeven kan worden. Meer uitgebreide specificaties van elementen kunnen worden gedocumenteerd als *properties* (eigenschappen van elementen), die een bepaalde waarde kunnen hebben. In UML zijn een aantal van deze eigenschappen voorgedefinieerd; voorbeeld hiervan is de eigenschap `persistence` die gebruikt kan worden om aan te geven dat een element persistent is.

Niet zozeer onderdeel van het model maar meer een manier om modellen te organiseren zijn:

- **Packages** — *Packages* bieden een manier om een complex model goed georganiseerd te houden. Een package is een groepering van andere UML-elementen (zoals model-elementen, diagrammen of weer packages) en heeft als zodanig alleen betekenis voor

modellering; een package stelt in principe geen eigenschap of onderdeel van het uiteindelijke systeem voor. Een package kan associaties hebben met andere packages. Merk op dat UML-packages dus iets anders zijn dan de packages in bijvoorbeeld Java, die wel degelijk betekenis hebben in de implementatie van het systeem.

De grafische representatie van decoraties, notities en packages is te zien in figuur 3.



Figuur 3 Algemene mechanismen in UML

3.1.3 De UML-diagrammen

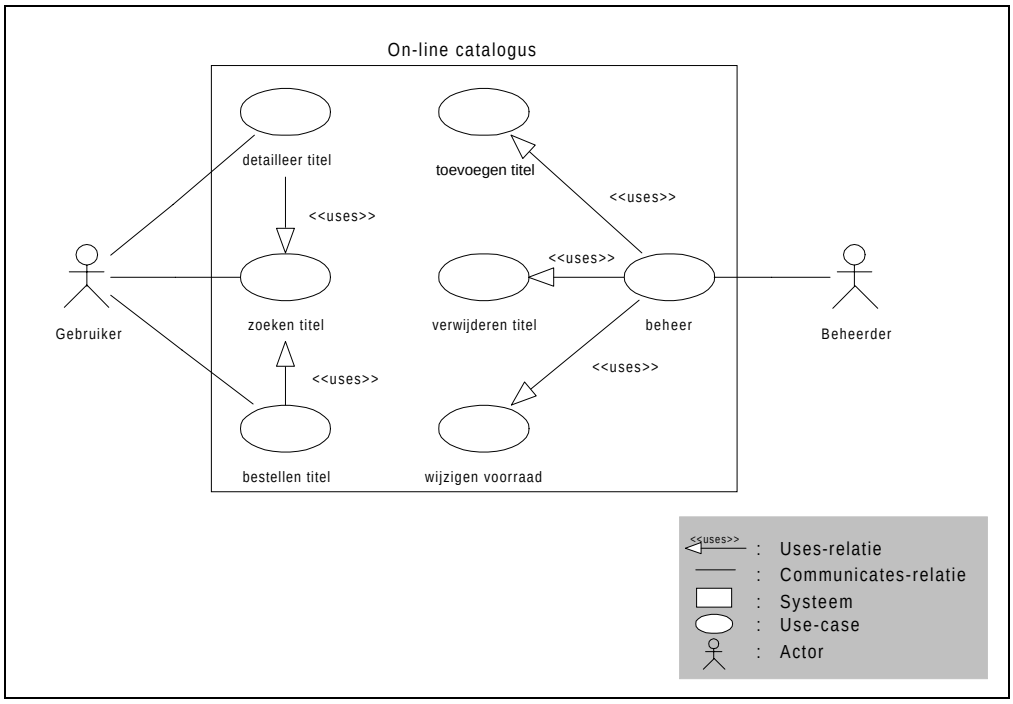
Diagrammen zijn in UML de eigenlijke grafische weergave van een model. Een diagram bestaat uit een aantal model-elementen die tezamen een bepaald systeemaspect modelleren. Een diagram maakt onderdeel uit van een bepaalde view, waarbij sommige typen diagrammen in principe onderdeel kunnen zijn van meerdere views. Alle diagrammen samen vormen de concrete weergave van een model van een systeem.

De verschillende typen UML-diagrammen worden in dit hoofdstuk geïntroduceerd aan de hand van een doorlopend voorbeeld: een on-line catalogussysteem voor een platenzaak. Gebruikers kunnen vanaf een andere locatie in het systeem kijken of een bepaalde titel in voorraad is en eventueel een bepaalde titel bestellen. De gegeven voorbeelden zijn puur bedoeld als introductie van de concepten uit de verschillende soorten UML-diagrammen, er is geen volledigheid van het model nagestreefd.

3.2 Het Use-casediagram

UML-diagrammen:
- **Use-case**
- *Statisch:*
  - Class
  - Object
- *Dynamisch:*
  - Statechart
  - *Interactie:*
    - Sequence
    - Collaboration
  - Activity
- *Implementatie:*
  - Component
  - Deployment

Een belangrijk aspect bij het modelleren van een systeem is de functionaliteit die het systeem biedt zoals gezien door de ogen van gebruikers. In UML kan dit aspect worden gemodelleerd in de use-case view. Belangrijkste onderdeel van deze use-case view zijn de use-casediagrammen, waarin de externe gebruikers van het systeem en hun relatie tot de use-cases die het systeem aanbiedt weergegeven kunnen worden. In figuur 4 is een use-casediagram te zien waarin de functionaliteit van het on-line catalogussysteem is gemodelleerd.



Figuur 4 Voorbeeld van een use-casediagram

Een *use-case*, in een use-casediagram weergeven als ellips, is een typische interactie tussen een gebruiker en een systeem; een use-case beschrijft een compleet stuk functionaliteit dat een systeem aanbiedt aan een gebruiker en dat een voor de gebruiker observeerbaar resultaat oplevert. Use-cases worden meestal nader toegelicht middels een tekstuele beschrijving.

Een *actor*, weergegeven met een poppetje, is uitvoerder van use-cases. Een actor is dus diegene (of datgene) die het systeem gebruikt. Een actor communiceert met een systeem door het sturen of ontvangen van berichten of informatie en kan dus zowel een mens als een ander systeem representeren. Het feit dat een bepaalde actor deelneemt in een bepaalde use-case wordt weergegeven met een *communicates*-relatie, een lijn tussen actor en use-case.

Het gemodelleerde systeem zelf wordt in een use-casediagram weergegeven met een rechthoek. Een use-casediagram geeft op die manier de grenzen van het systeem aan; er wordt vastgelegd wat de verantwoordelijkheden van het systeem zijn. Het systeem in een use-casediagram is een *black-box*, er wordt alleen beschreven welke functionaliteit het systeem ondersteunt, niet hoe het systeem dat doet.

Behalve relaties tussen actoren en use-cases kunnen in een use-casediagram ook relaties tussen use-cases onderling worden aangegeven. Er zijn twee standaard relaties: de *uses*-relatie (een pijl met label «`uses`») en de *extends*-relatie (een pijl met label «`extends`»). Een extends-relatie kan gebruikt worden om aan te geven dat een bepaalde use-case een uitbreiding of variatie is op een andere use-case. Een uses-relatie kan worden gebruikt om gedrag te modelleren dat door meerdere use-cases gebruikt wordt.

De use-case view is de centrale view bij modelleren van systemen. Aangezien het uiteindelijke doel van het systeem implementatie van de in de use-case view omschreven functionaliteit is, beïnvloedt deze view alle anderen. Use-cases kunnen tijdens het gehele ontwikkeltraject worden gebruikt, van het vastleggen van functionele eisen tijdens analyse tot aan het testen van het systeem toe.

3.3 Statische-structuurdiagrammen

Er zijn in UML twee soorten diagrammen waarin de statische structuur van een systeem kan worden beschreven. Classdiagrammen geven de statische structuur van het model weer, meer in het bijzonder de bestaande klassen, hun interne structuur en de onderlinge statische relaties. Classdiagrammen geven in principe geen dynamische informatie weer, maar kunnen eventueel wel de gegevens die gebruikt worden om deze informatie vast te leggen modelleren. Een objectdiagram geeft een instantie weer van een classdiagram op een bepaald moment in de tijd.

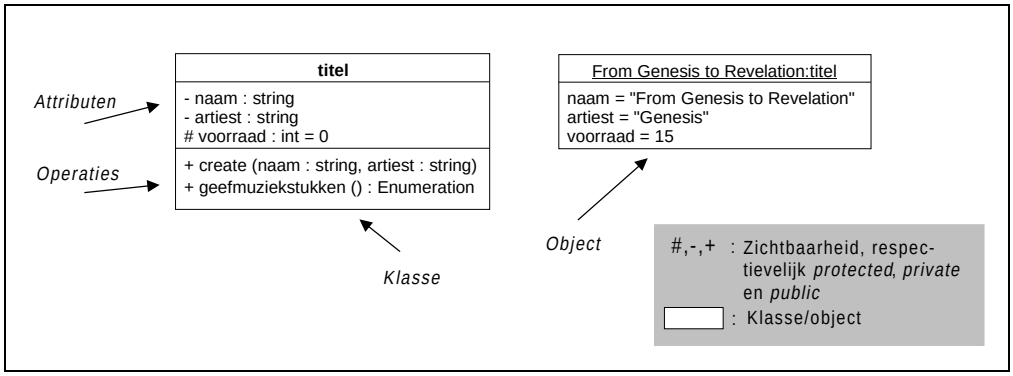
In deze paragraaf zullen de verschillende constructies worden besproken die betrekking hebben op deze statische-structuurdiagrammen. Meer het bijzonder wordt de notatie voor klassen, objecten en hun mogelijke onderlinge relaties besproken. De notatie die in deze paragraaf wordt gepresenteerd vormt een basis voor de andere UML-diagrammen. In het algemeen geldt dat in de modellen veel notatie-elementen niet volledig weergegeven of zelfs weggelaten kunnen worden zonder de betekenis van het model te wijzigen; de mate van detail kan naar behoefte worden aangepast.

3.3.1 Klassen

UML-diagrammen:
- Use-case
- *Statisch:*
  - **Class**
  - Object
- *Dynamisch:*
  - Statechart
  - *Interactie:*
    - Sequence
    - Collaboration
  - Activity
- *Implementatie:*
  - Component
  - Deployment

Klassen, de representatie van verschillende domein- of implementatieconcepten in het te modelleren systeem, worden in UML weergegeven als een rechthoek bestaande uit drie compartimenten die respectievelijk de naam, de attributen en de operaties van de klasse weergeven. Eventueel kunnen er implementatiespecifieke compartimenten worden toegevoegd.

In het eerste compartiment van de klasse-representatie wordt de naam van de klasse dik gedrukt en gecentreerd weergegeven. *Abstracte klassen* kunnen worden aangeduid door de naam van de klasse cursief weer te geven. Het middelste compartiment bestaat uit een lijst van *attributen* van de klasse. Voor elk attribuut kunnen de zichtbaarheid, het type, de initiële waarde en specifieke eigenschappen worden aangegeven. Het type van een attribuut is direct gerelateerd aan de gebruikte programmeertaal en wordt niet gespecificeerd in UML. Het onderste compartiment bevat tenslotte een lijst van *operaties* van de klasse. Bij een operatie kunnen, naast de naam, eventueel ook de zichtbaarheid van de operatie, een lijst met parameters, het resultaattype en meer specifieke eigenschappen worden weergegeven. Figuur 5 geeft een voorbeeld van de weergave van een klasse uit het catalogussysteem in UML.



Figuur 5 Klassen en objecten

3.3.1 Objecten

UML-diagrammen:
- Use-case
- *Statisch:*
  - Class
  - **Object**
- *Dynamisch:*
  - Statechart
  - *Interactie:*
    - Sequence
    - Collaboration
  - Activity
- *Implementatie:*
  - Component
  - Deployment

Objecten zijn instanties van klassen en kunnen worden beschreven in een objectdiagram. Een dergelijk objectdiagram geeft dus een invulling van een classdiagram op een bepaald moment in tijd. Eventueel kunnen objecten ook direct in classdiagrammen worden opgenomen, bijvoorbeeld ter verduidelijking van het gebruik van een klasse. Het gebruik van objectdiagrammen is beperkt en zal voornamelijk dienen ter illustratie van concrete objecten en relaties die op een bepaald tijdstip in een systeem aanwezig kunnen zijn. De anders nogal abstracte classdiagrammen kunnen op die manier concreter gemaakt worden.

De notatie voor een object is afgeleid van die van een klasse. Hierbij worden de naam van het object en de bijbehorende klasse onderstreept en gescheiden door een dubbele punt, overeenkomstig de al eerder besproken standaard notatie voor het maken van onderscheid tussen typen en instanties. Indien de naam van het object wordt weggelaten is er sprake van een ‘anoniem’ object. Een voorbeeld van het gebruik van de notatie voor objecten is te zien in figuur 5. Merk op dat in een objectdiagram bij objecten concrete attribuutwaarden kunnen worden weergegeven. Operaties zijn niet specifiek voor objecten, maar voor klassen van objecten en zijn daarom weggelaten in objectdiagrammen.

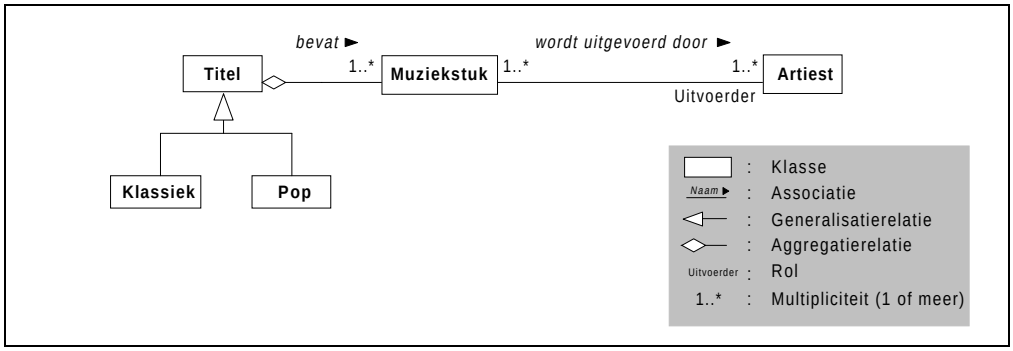
Objecten kunnen, als gevolg van *multiple inheritance*, een instantie zijn van verschillende klassen; in een objectdiagram kunnen al deze klassen bij een object worden weergegeven. Eventueel kunnen bij objecten ook de mogelijke *toestanden* worden aangegeven, die mogelijk zijn gedefinieerd middels de nog te bespreken statechartdiagrammen. Het is in UML mogelijk *samengestelde objecten* weer te geven door de objecten die logisch gezien deel uit maken van een ander object binnen het samengestelde object te tekenen.

3.3.3 Relaties

Klassen, en ook objecten, zijn vrijwel altijd onderling gerelateerd. Er wordt in UML onderscheid gemaakt tussen een aantal verschillende soorten relaties. Een *associatie* is een relatie tussen twee klassen die met elkaar in verband staan. Op object niveau wordt een dergelijke relatie een *link* genoemd. Een voorbeeld van een associatie is te zien in het classdiagram in figuur 6. Associaties kunnen eventueel een bepaalde richting hebben, aangegeven door een pijltje bij de naam van de associatie. Bij een associatie kunnen ook rolnamen worden weergegeven die de *rollen* aangeven die de bijbehorende klassen in een associatie vervullen.

Een *aggregatierelatie* is een speciaal soort associatie die een samengesteld object relateert aan zijn deelobjecten. Indien deze objecten onlosmakelijk zijn verbonden met het samengestelde object wordt over een *compositierelatie* gesproken. Beide relaties worden weergegeven door een kleine diamantvorm aan de kant van het samengestelde object, zoals te zien in figuur 6. In het geval van compositie is deze diamant gevuld.

Een *generalisatierelatie* relateert een bepaald model-element aan een meer specifiek element. Het specifieke element is volledig consistent met het eerste element (ook wel het *generieke* element genoemd), maar voegt extra informatie toe: het is een specialisatie van het eerste element. Generalisatierelaties komen typisch voor tussen klassen, zoals te zien in figuur 6, maar kunnen ook gebruikt worden om packages, use-cases of andere model-elementen te relateren.



Figuur 6 Relaties tussen klassen

Naast associaties, aggregaties en generalisaties kunnen ook *afhankelijkheidsrelaties* bestaan tussen elementen uit het model. Een afhankelijkheidsrelatie geeft aan dat een wijziging in een element invloed heeft op een ander element en wordt weergegeven door een onderbroken pijl vanaf het afhankelijke object.

Bij de verschillende soorten relaties kan eventueel nog extra informatie worden opgenomen. Een *kwalificatie* is een eigenschap van een relatie die aangeeft welke onderscheidende attributen een object heeft in relatie tot een ander object. De *multipliciteit* van een relatie geeft aan hoeveel objecten aan de ene kant van de relatie gerelateerd kunnen zijn aan objecten aan de andere kant van de relatie. Multipliciteit wordt in een model weergegeven door aan één kant of beide kanten van een relatie een onder- en bovengrens of een constant aantal aan te geven.

3.3.4 Geavanceerde Notatie

In statische-structuurdiagrammen kunnen naast klassen, objecten en relaties ook een aantal meer geavanceerde concepten worden weergegeven. Een relatie die belangrijke eigenschappen bevat of die zelf ook gerelateerd is aan andere elementen kan worden weergegeven als een *associatieklasse*. Deze wordt genoteerd als een klasse met een onderbroken lijn naar de relatie.

Een *interface* is een specificatie van de extern zichtbare operaties van een klasse, een component of ander model-element, zonder de daarbij horende implementatie. Voor klassen komen interfaces dus overeen met *abstracte klassen* zonder attributen of methoden, maar met alleen abstracte operaties. Interfaces kunnen op twee verschillende manieren worden weergegeven in UML. Eén manier is om een interface weer te geven als een klasse met het stereotype «`interface`». (Stereotypes worden besproken in paragraaf 4.3). Een andere manier om interfaces te modelleren is om bij klassen die een bepaalde interface implementeren deze interface als open bolletje met een lijn, verbonden aan de klasse weer te geven. De naam van de interface staat bij het bolletje. Een voorbeeld van het gebruik van een interface voor een component is te zien in figuur 11 in paragraaf 3.5.1.

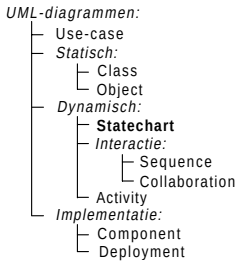
Templates zijn klassen die worden geparametriseerd met typen (klassen of primitieve typen) of constanten. Een template wordt weergegeven als een klasse met in de rechterbovenhoek een onderbroken rechthoek met daarin de parameters.

3.4 Dynamische-structuurdiagrammen

Een dynamisch model richt zich op het vastleggen van het gedrag van een bepaald systeem. Niet de objecten die binnen het systeem aanwezig zijn staan centraal, maar de manier waarop deze objecten samenwerken om een bepaalde taak te volbrengen. UML kent vier diagramtypen voor het modelleren van dynamisch gedrag:

- **Statechartdiagram** — Beschrijft het gedrag van een object; de verschillende toestanden waarin het object zich tijdens zijn levensduur kan bevinden, het gedrag in die verschillende toestanden, en de gebeurtenissen die overgangen tussen toestanden kunnen bewerkstelligen.
- **Sequencediagram** — Beschrijft de interactie tussen bepaalde objecten, waarbij de nadruk vooral ligt op de berichten die de objecten elkaar sturen en de volgorde van deze berichten.
- **Collaborationdiagram** — Beschrijft ook de interactie tussen objecten, maar benadrukt daarbij vooral de (statische) relaties die bestaan tussen de communicerende objecten onderling.
- **Activitydiagram** — Beschrijft niet zozeer de samenwerking tussen objecten bij het uitvoeren van een bepaalde taak, als wel de taak zelf; het diagram geeft de activiteiten (en de volgorde daartussen) weer die nodig zijn voor het uitvoeren van een taak.

3.4.1 Het Statechartdiagram

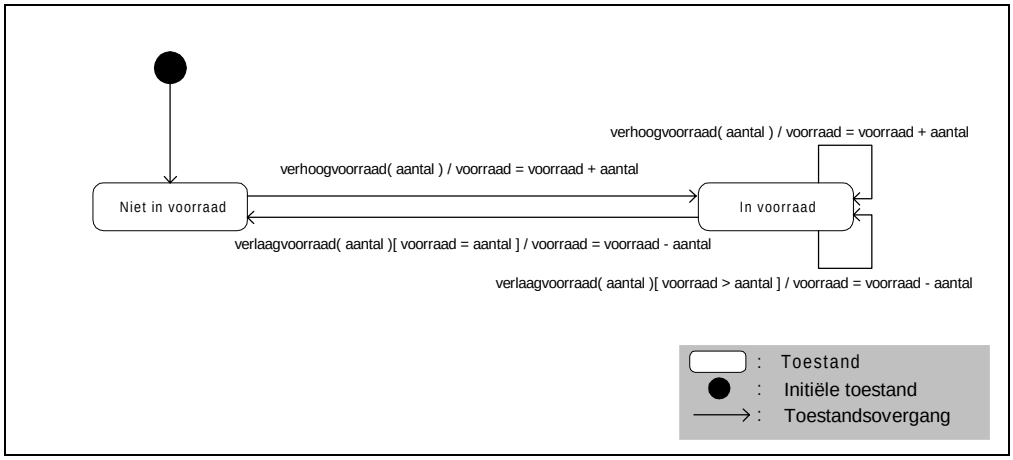


Een statechartdiagram, ook wel aangeduid als ‘toestandsdiagram’, beschrijft de volgorde van toestanden die een object doormaakt in zijn levenscyclus in relatie tot gebeurtenissen in de omgeving van het object, samen met zijn reacties en acties. Het diagram is een graaf bestaande uit toestanden en toestandsovergangen behorende bij objecten van een bepaalde klasse in het algemeen, of bij de gedetailleerde specificatie van een methode uit een bepaalde klasse in termen van subtoestanden, acties en gebeurtenissen in het bijzonder.

Een *toestand* is een bepaald moment in de levenscyclus van een object waarin een actie wordt uitgevoerd, wordt gewacht op het optreden van een bepaalde gebeurtenis of waarin de attributen een bepaalde waarde hebben. Een toestand wordt in UML weergegeven door een rechthoek met afgeronde hoeken, met daarin een compartiment voor de naam van de toestand en een compartiment met een verdere specificatie. De verdere specificatie kan bestaan uit gebeurtenissen en bijbehorende acties; deze actie-expressie verwijst typisch naar een methode in het object. De specificatie van de toestand kan eventueel zelf weer bestaan

uit een toestandsdiagram. Dit diagram kan losstaand of in het compartiment zelf worden weergegeven. Naast diagrammen die sequentieel te doorlopen subtoestanden weergeven, is het ook mogelijk parallelle subtoestanden weer te geven; parallelle subtoestanden kunnen worden gebruikt om weer te geven dat een object zich in meerdere subtoestanden tegelijkertijd kan bevinden. Subtoestanden in een toestandsdiagram kunnen worden verborgen door het gebruik van *stubs*, een grafisch mechanisme om toestanden te kunnen samenvatten in één abstracte toestand. Eventuele overgangen naar subtoestanden worden vervangen door overgangen naar de stubs.

Naast normale toestanden zijn er in UML twee speciale toestanden gedefinieerd: de initiële en de finale toestand. Deze pseudo-toestanden worden gebruikt om de overgang naar de begintoestand en van de eindtoestand weer te geven. De notatie bestaat uit een gevuld bolletje voor de initiële toestand, en een gevuld bolletje met een cirkel voor de finale toestand. Figuur 7 geeft een voorbeeld van een statechartdiagram voor de klasse `titel` uit het model voor het catalogussysteem, bestaande uit een initiële toestand en de toestanden `Niet in voorraad` en `In voorraad`.



Figuur 7 Statechartdiagram voor klasse 'titel'

Indien er in de omgeving van een object een gebeurtenis optreedt dan kan er een *overgang* van de ene naar de andere toestand plaatsvinden. Zo'n gebeurtenis kan zijn een conditie die waar wordt, het ontvangen van een signaal, de aanroep van een methode of het verstrijken van een bepaalde tijd. Een toestandsovergang wordt weergegeven door een pijl van de ene naar de andere toestand. Bij deze pijl kan de naam van de gebeurtenis staan, met eventuele parameters, condities, en uit te voeren acties of zendacties. Met behulp van een zendactie is het mogelijk bij het optreden van een gebeurtenis een bericht naar een ander object te sturen. De syntax van een zendactie bestaat uit een '^', gevolgd door de ontvanger van het bericht, het bericht zelf, en eventuele parameters. Het alternatief voor deze syntax is het gebruik van een onderbroken pijl vanaf de overgang naar een doelobject of overgang. Voorbeelden van toestandsovergangen met parameters, condities en acties zijn te zien in figuur 7.

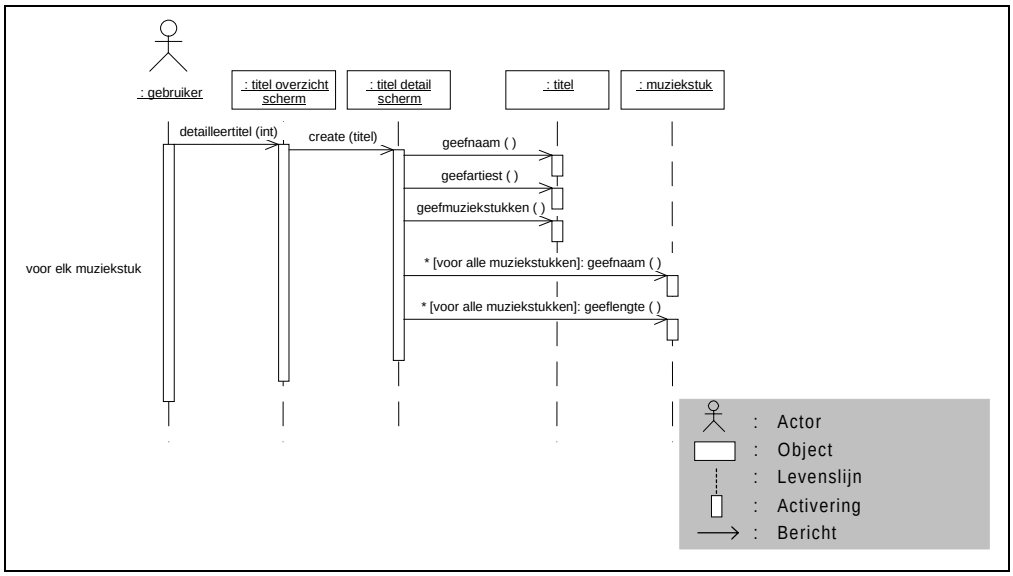
Complexe overgangen maken het mogelijk om een toestandsovergang op te splitsen in meerdere overgangen of om overgangen samen te voegen. In het eerste geval worden er bij het optreden van een gebeurtenis meerdere nieuwe gebeurtenissen gecreëerd. In het tweede geval treedt een gebeurtenis pas op als alle andere gebeurtenissen zijn opgetreden. De notatie bestaat uit een verticale balk waaraan zowel de bron- als de doelovergangen zijn verbonden. *Interne toestandsovergangen* zijn overgangen die de toestand van het object ongewijzigd laten.

3.4.2 Het Sequencediagram

UML-diagrammen:
- Use-case
- *Statisch:*
  - Class
  - Object
- *Dynamisch:*
  - Statechart
  - *Interactie:*
    - **Sequence**
    - Collaboration
  - Activity
- *Implementatie:*
  - Component
  - Deployment

Een sequencediagram wordt gebruikt om een interactie tussen objecten weer te geven, gerelateerd aan de tijd. Er zijn twee varianten van het sequencediagram. In de generieke variant worden *alle* mogelijke interacties tussen bepaalde soorten objecten weergegeven, inclusief eventuele lussen en sprongen. Op instantieniveau, de tweede variant, wordt *één* specifieke interactie weergegeven.

Het sequencediagram wordt weergegeven aan de hand van objecten die horizontaal gerangschikt zijn, hun tijdslijn die verticaal is aangegeven en de interacties tussen de objecten. Op de verticale as wordt het verloop in tijd weergegeven, waarbij de tijd verloopt van boven naar beneden. Hierbij kunnen specifieke beperkingen, met betrekking tot bijvoorbeeld tijd, worden weergegeven in de linker kantlijn. In figuur 8 is met behulp van een sequencediagram een uitwerking gegeven van de use-case `detailleer titel` uit het model voor het on-line catalogussysteem.



Figuur 8 Sequencediagram voor use-case ‘detailleer titel’

Een object in een sequencediagram kan een object zijn dat gedurende de gehele interactie bestaat, of dat tijdens de interactie wordt gecreëerd en/of verwijderd. Een *passief object* is een object dat zelf niet in staat is actie te ondernemen tenzij het expliciet de controle krijgt. De tijd dat een object bestaat wordt aangegeven door een onderbroken verticale lijn, de *levenslijn*. De momenten waarop het object actief is worden aangegeven met een dunne rechthoek langs de levenslijn van het object. Deze rechthoek begint op het moment dat het object actief of gecreëerd wordt, en eindigt als het object niet meer actief is of wordt verwijderd. In dat laatste geval wordt er onderaan de rechthoek een grote 'X' weergegeven. Indien een object actief is maar de controle tijdelijk heeft gegeven aan een ander object, kan dat gedeelte van de rechthoek worden gearceerd.

Een *actief object* is zelf in staat actie te ondernemen zonder dat het expliciet de controle krijgt. Dit betekent dat zo'n object continu actief is en dat er een rechthoek over de volle lengte van het diagram voor dit object is getekend. Bovendien wordt het object zelf weergegeven met een dikkere rand. De volgorde van objecten in een sequencediagram heeft geen formele betekenis maar verloopt vaak in de volgorde van activering.

Wanneer een object communiceert met een ander object wordt een *bericht* uitgewisseld. Een bericht wordt weergegeven door een solide pijl van de levenslijn van een object naar de levenslijn van een ander object. Bij de pijl kunnen de naam van het bericht, eventuele parameters en een conditie worden aangegeven. Ook is het mogelijk berichten een volgnummer te geven, hoewel de volgorde van berichten in een sequencediagram in principe impliciet is. De vorm van de pijl kan worden aangepast om weer te geven dat het om een *asynchroon* bericht (halve simpele pijlpunt) of een *procedureaanroep* (volledige solide pijlpunt) gaat. In het laatste geval is het mogelijk met een onderbroken pijl aan te geven wanneer de aanroep terugkeert. In een systeem met parallel gedrag geeft een volledige pijlpunt aan dat de controle over wordt gegeven aan een andere thread (synchroon) en een halve pijl dat een bericht wordt verzonden zonder wisseling van controle (asynchroon).

Indien het verzenden van een bericht logisch gezien niet als atomaire actie kan worden beschouwd kan een pijl schuin naar beneden worden weergegeven. Een conditionele vertakking wordt weergegeven door meerdere pijlen die vanuit één punt vertrekken, met de condities geplaatst boven de pijlen. In de linker marge kunnen de tijden waarop berichten worden verzonden of ontvangen aangegeven worden met een naam. Deze namen kunnen worden gebruikt in expressies om tijdsbeperkingen aan te geven.

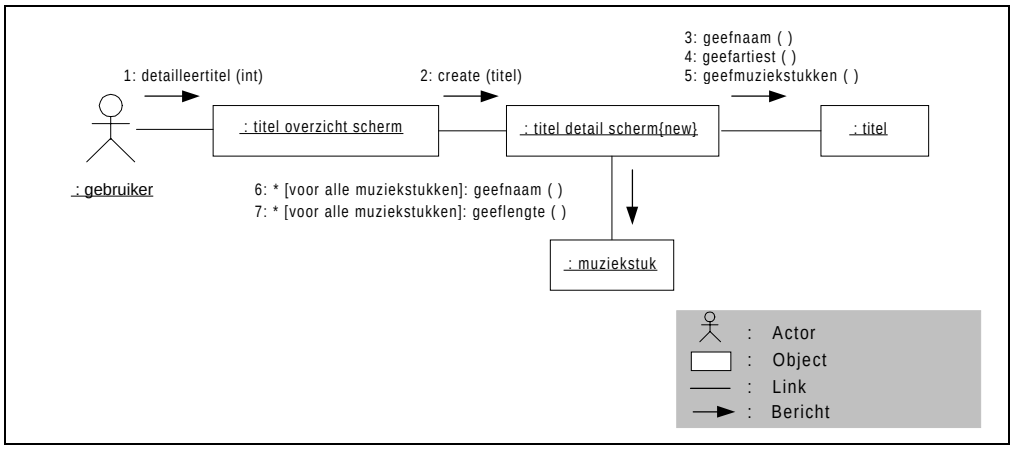
3.4.3 Het Collaborationdiagram

UML-diagrammen:
- Use-case
- *Statisch:*
  - Class
  - Object
- *Dynamisch:*
  - Statechart
  - *Interactie:*
    - Sequence
    - **Collaboration**
  - Activity
- *Implementatie:*
  - Component
  - Deployment

Collaborationdiagrammen kunnen worden gebruikt om de interactie tussen samenwerkende objecten weer te geven, een vergelijkbare functie dus als die van sequencediagrammen. Daar waar in een sequencediagram echter de nadruk ligt op de volgorde en het tijdsverloop van de tussen objecten uitgewisselde berichten, benadrukt een collaborationdiagram vooral de relaties tussen de samenwerkende objecten.

Zoals al eerder beschreven kunnen objecten en relaties tussen objecten worden weergegeven in een objectdiagram. De basis van een collaborationdiagram wordt dan ook gevormd door een objectdiagram; de notatie voor objecten en relaties is in beide diagrammen gelijk. Naast objecten en relaties geeft een collaborationdiagram echter ook nog aan welke berichten tussen gerelateerde objecten verstuurd worden.

Berichten worden weergegeven met labels bij de verbindingen tussen objecten, zoals te zien in het voorbeeld in figuur 9. De figuur geeft een uitwerking van de use-case `detailleer titel` waarin de gebruiker informatie over een bepaalde titel op kan vragen. In de labels kunnen verschillende soorten informatie worden gecodeerd, waaronder de (eventueel geneste) volgorde van berichten, eventuele condities op berichten of iteraties van berichten, en resultaatwaarden. De vorm van de pijl bij de berichten kan gebruikt worden om verschillende soorten berichten te onderscheiden, analoog aan het onderscheid in sequencediagrammen. De in de figuur gebruikte dichte pijl geeft aan het bericht de vorm van een procedure aanroep of andere geneste control flow heeft. Met labels bij de objecten, bijvoorbeeld het label `{new}` bij het `titel detail scherm` in de figuur, kan worden aangegeven dat objecten tijdens de interactie gecreëerd of verwijderd (of allebei) worden.



Figuur 9 Collaborationdiagram voor use-case ‘detailleer titel’

Net als sequencediagrammen zijn collaborationdiagrammen geschikt voor het weergeven van het gedrag van samenwerkende objecten, bijvoorbeeld bij executie van een use-case of een operatie. De twee soorten interactiediagrammen zijn daarbij vooral gericht op het modelleren van de samenwerking tussen objecten; voor een wat meer expliciete definitie

van het gedrag zelf kan beter een activitydiagram worden gebruikt. De keuze tussen het gebruik van een sequence- of een collaborationdiagram hangt af van de gewenste nadruk van het model: een collaboration diagram geeft naast de berichtenstroom ook de relaties tussen de communicerende objecten weer en is daarom vaak meer geschikt voor het begrijpen van een interactie, een sequencediagram geeft een betere visualisatie van de tijdsvolgorde van de verstuurde berichten.

Een toepassing van collaborationdiagrammen is het modelleren van standaard ontwerpconstructies of *design patterns*, collaborationdiagrammen zijn geschikt voor het weergeven van de benodigde context en interactie van de patronen. Meer over patronen en het gebruik van UML bij de modellering daarvan is te vinden in [Elswijk98] en [Eriksson98].

3.4.4 Het Activitydiagram

UML-diagrammen:
- Use-case
- *Statisch:*
  - Class
  - Object
- *Dynamisch:*
  - Statechart
  - *Interactie:*
    - Sequence
    - Collaboration
  - **Activity**
- *Implementatie:*
  - Component
  - Deployment

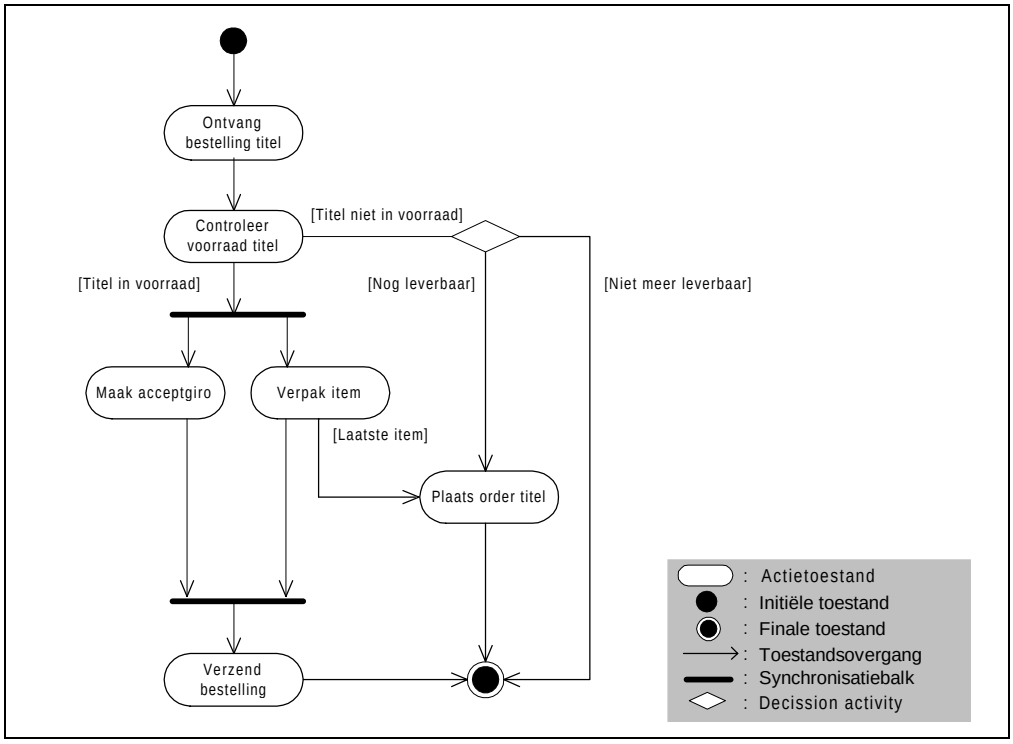
Statechartdiagrammen zijn geschikt om het gedrag van een object te beschrijven als reactie op gebeurtenissen in de omgeving van het object, interactiediagrammen kunnen worden gebruikt om samenwerking tussen objecten bij het uitvoeren van een bepaalde taak weer te geven. Voor de uitwerking van het eigenlijke werk dat gedaan moet worden, bijvoorbeeld in een use-case of binnen een methode in een klasse, zijn deze diagrammen minder geschikt. Speciaal voor het modelleren van concreet uit te voeren activiteiten en de volgorde tussen die activiteiten (samen te vatten in de term *workflow*) biedt UML het activitydiagram.

Activitydiagrammen zijn een variatie op statechartdiagrammen. Een toestand in een activitydiagram stelt echter een bepaalde actie of hoeveelheid werk voor, en niet de toestand van een specifiek object. Overgangen tussen deze *actietoestanden* worden in principe dan ook niet te weeg gebracht door externe gebeurtenissen, maar door de voltooiing van de acties in een actietoestand.

Een voorbeeld van een activitydiagram is te zien in figuur 10, waar een uitwerking is gegeven van een gedeelte van de use-case `wijzigen voorraad`. De use-case beschrijft het binnenkomen en afhandelen van een bestelling van een titel door een klant. Actietoestanden worden in UML weergegeven met een rechthoek met ronde zijkanten, overgangen met pijlen tussen actietoestanden. Overgangen in activitydiagrammen worden in principe niet gelabeld met gebeurtenissen; overgangen tussen actietoestanden zijn impliciet gevolg van voltooiing van de activiteiten in een toestand. Overgangen kunnen eventueel wel voorzien worden van conditionele expressies.

Een belangrijke eigenschap van activitydiagrammen is de mogelijkheid om parallel gedrag weer te geven. Parallel gedrag wordt gerepresenteerd middels een *synchronisatiebalk*, waar gelijktijdige activiteiten kunnen ontspringen of samenkomen. De synchronisatiebalk geeft aan dat de volgorde waarin acties worden uitgevoerd er niet toe doet; de enige voorwaarde is dat alle parallelle acties moeten zijn uitgevoerd voordat opvolgende actietoestanden

bereikt kunnen worden. *Samengestelde beslissingen* kunnen in een activitydiagram worden weergegeven met een *decision activity*, het diamantsymbool in de figuur.



Figuur 10 Activitydiagram voor gedeelte use-case 'Wijzigen voorraad'

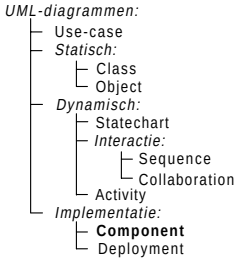
Een activitydiagram geeft in principe alleen aan hoe een bepaalde taak of operatie moet worden uitgevoerd, niet wie of wat verantwoordelijk is voor de uitvoer van de activiteiten. Middels *swimlanes*, verticale banen in het activitydiagram die ieder een bepaalde klasse voorstellen, kunnen activiteiten worden gegroepeerd bij de verantwoordelijke uitvoerder. Objecten kunnen ook op een andere manier worden gebruikt in activitydiagrammen; activiteiten kunnen objecten als invoer of uitvoer hebben.

Activitydiagrammen kunnen op verschillende conceptuele niveaus worden gebruikt; de notatie is geschikt voor het vastleggen van workflow tijdens business modellering, voor het uitwerken van use-cases tijdens systeemanalyse of voor uitwerken van specifieke operaties in klassen bij ontwerp of implementatie.

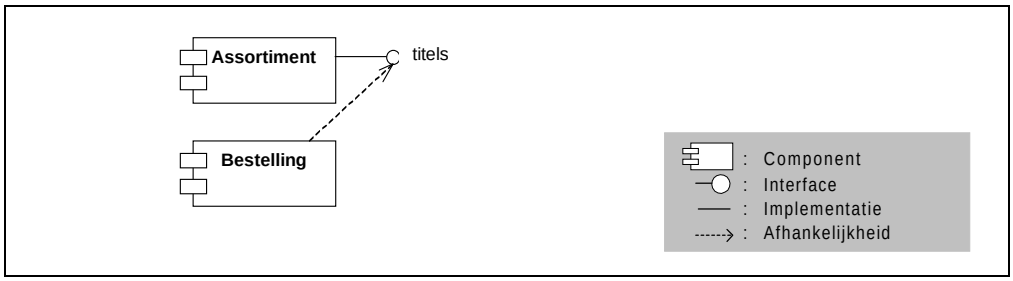
3.5 Implementatiediagrammen

In UML bestaat de mogelijkheid om niet alleen aspecten vast te leggen over de statische of dynamische structuur van het model, maar ook aspecten die betrekking hebben op de uiteindelijke implementatie van het model. *Implementatiediagrammen* geven deze implementatieaspecten van een systeem weer, inclusief de broncode- en runtimestructuur. Er zijn twee soorten implementatiediagrammen: het componentdiagram geeft alle compile-, link- en runtimecomponenten weer die in the systeem aanwezig zijn, het deploymentdiagram geeft aan hoe deze componenten verdeeld worden over *runtime nodes*.

3.5.1 Het Componentdiagram



Een *component* is een tastbaar stuk van een implementatie van een systeem. Dit kunnen software-componenten zijn die tijdens compile- (broncode), link- (objectcode) of runtime (machinecode) van belang zijn, maar het kunnen ook andersoortige documenten zijn die bij een systeem bestaan. Deze componenten kunnen afhankelijk zijn van elkaar, net als klassen of objecten afhankelijk kunnen zijn van elkaar. Figuur 11 geeft een voorbeeld van een componentdiagram; het model bevat twee componenten waarbij de component `Assortiment` een implementatie biedt voor de interface `titels`.

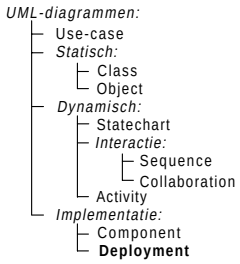


Figuur 11 Componentdiagram

In een componentdiagram worden de componenten en hun afhankelijkheden inzichtelijk gemaakt. Een component wordt hierbij weergegeven als een rechthoek met aan de linkerkant twee uitstekende rechthoekjes; in of onder het component wordt het type weergegeven. In het componentdiagram komen alleen componenttypen voor, de uiteindelijke componentinstanties zien we terug in het deploymentdiagram. Dit onderscheid tussen type en instantie volgt de standaard type/instantie-notatie, zoals beschreven in paragraaf 3.1.2.

De afhankelijkheden tussen componenten worden weergegeven met onderbroken pijlen. Deze pijlen kunnen ook naar eventuele interfaces van een component wijzen. Indien een component deel uit maakt van een andere component dan wordt deze in de bevattende component weergegeven.

3.5.2 Het Deploymentdiagram

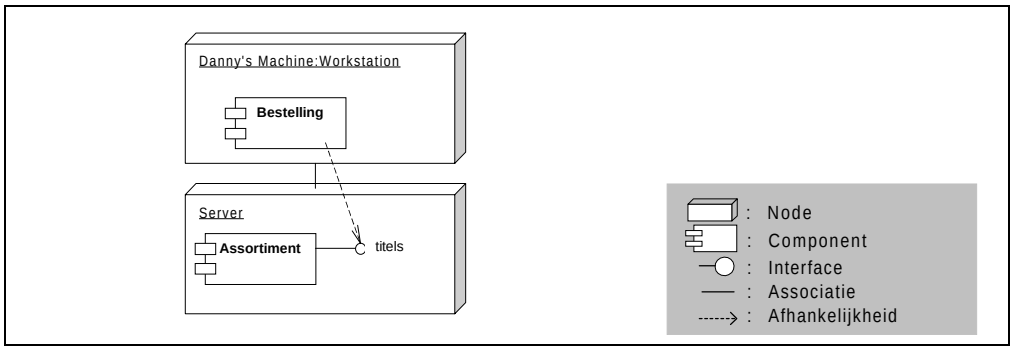


In een deploymentdiagram wordt de runtime structuur van een software-systeem weergegeven. Hierbij bestaat het systeem uit *nodes*, fysieke objecten die in staat zijn bewerkingen uit te voeren, componenten en objecten. Alleen de componenten die tijdens runtime van belang zijn worden hierbij weergegeven. Een node beschikt meestal over geheugen en rekenkracht en kan zowel een computer, een menselijke uitvoerder, als een andere mechanische bron voorstellen. Een node kan zowel in type- als instantievorm voorkomen en wordt weergegeven als een driedimensionale rechthoek. In of onder de rechthoek wordt het type, en optioneel de naam van de instantie weergegeven.

In een node kunnen componenten en objecten zijn weergegeven; dit betekent dat deze entiteiten zich tijdens runtime op de betreffende node kunnen bevinden. Indien een entiteit kan worden verplaatst naar een andere node wordt dat aangegeven met een onderbroken pijl met stereotype «`becomes`» tussen instanties van de entiteit in de eerste en de tweede node.

Ook een component kan objecten bevatten; deze objecten maken deel uit van de component en worden in de component weergegeven. Deze compositierelatie kan eventueel ook met een compositieassociatie worden weergegeven. Tevens is het mogelijk hiertoe een *location attribuut* in het object op te nemen dat naar de omvattende component verwijst. Deze notatievormen voor compositierelaties zijn ook geldig voor een node.

Een *afhankelijkheidsrelatie* van het stereotype «`supports`» geeft aan welke componenten zich op welke nodes kunnen bevinden. *Communicatie* tussen nodes wordt weergegeven door een lijn tussen nodes. Hierbij kan een naam bij de lijn aangeven wat voor soort communicatie er plaats vindt. Een voorbeeld van een deploymentdiagram is te zien in figuur 12.



Figuur 12 Deploymentdiagram

4 UML uitbreiden

In het vorige hoofdstuk is de notatie van de negen soorten UML-diagrammen besproken. Om te voorkomen dat UML te complex zou worden hebben de ontwerpers een aantal definities en mechanismen uit de notatie weggelaten die in andere modelleringstalen wel aanwezig zijn. Om de uitdrukkingskracht van UML niet te beperken is er echter een aantal uitbreidingsmechanismen ontworpen. Met deze mechanismen is het mogelijk beperkingen (constraints), eigenschappen (properties) en semantiek (stereotypes) aan UML-elementen toe te voegen.

4.1 Constraints

Constraints zijn beperkingen die betrekking hebben op één type element. Zo'n beperking kan een van de veertien voorgedefinieerde beperkingen zijn of een door de gebruiker gedefinieerde beperking. Voorgedefinieerde beperkingen kunnen betrekking hebben op onder andere generalisatierelaties, associaties, associatierollen, berichten of objecten. Voor een generalisatierelatie kan bijvoorbeeld worden aangegeven dat deze volledig, onvolledig, disjunct of overlappend is: zo kan een overlappende generalisatierelatie worden aangeduid door het label `{overlapping}`. Voor associaties zijn beperkingen gedefinieerd voor impliciete en disjuncte associaties. Associatierollen hebben slechts één voorgedefinieerde beperking die aangeeft dat een associatie geordend is. Voor berichten, rollen en objecten kan worden aangegeven of deze bijvoorbeeld globaal of lokaal beschikbaar zijn, of dat ze worden gecreëerd tijdens een interactie.

Het is mogelijk om zelf beperkingen aan elementen te koppelen. Hiervoor kan een natuurlijke of formele taal worden gebruikt. Aanbevolen wordt om gebruik te maken van OCL (Object Constraint Language) welke ook gebruikt is bij het definiëren van het UML metamodel. OCL is een eenvoudige expressietaal waarin beperkingen kunnen worden beschreven. Een OCL expressie heeft geen bijwerkingen en levert bij elke evaluatie dus hetzelfde resultaat op. OCL expressies kunnen onder andere bestaan uit pad-expressies waarmee genavigeerd kan worden naar bijvoorbeeld attributen en associaties. Verder is een aantal standaard operaties gedefinieerd in OCL die in een expressie kunnen worden gebruikt. Een eenvoudig voorbeeld van een OCL expressie is `{titel.prijs >= 0}`.

Een beperking wordt weergegeven tussen accolades bij het element waarop de beperking betrekking heeft. Indien een beperking betrekking heeft op meerdere elementen dan kan deze worden weergegeven naast een onderbroken lijn door alle betrokken elementen.

4.2 Properties

Properties zijn eigenschappen van een bepaald element, voorgedefinieerd in UML of gedefinieerd door de gebruiker zelf. Eigenschappen worden tussen accolades weergegeven in of bij het element waarop ze betrekking hebben. De definitie van een eigenschap bestaat vaak uit een *tagged value*, een naam-waarde tupel; een logische waarde kan worden weergegeven met alleen de naam van een tagged value. Eigenschappen kunnen worden gebruikt om extra informatie aan de gebruiker kenbaar te maken of om automatische mechanismen te kunnen ondersteunen, zoals bijvoorbeeld het genereren van programmacode.

Voor elementen is er een `documentation` eigenschap die kan worden gebruikt om een element te documenteren. Voor typen, instanties, operaties en attributen zijn eigenschappen gedefinieerd voor onder andere invarianten (`invariant`), precondities (`precondition`), postcondities (`postcondition`) en abstracte elementen (`abstract`). De `location` eigenschap geeft aan in welk component of node een element is geplaatst.

Voor zelfgedefinieerde eigenschappen is het belangrijk goed te documenteren waar de eigenschap toe dient, op welke elementen de eigenschap betrekking op heeft en hoe de eigenschap gebruikt dient te worden. Een voorbeeld van een zelf gedefinieerde eigenschap is `Authors`, welke de auteurs van een element aangeeft. De weergave van deze eigenschap in een model kan zijn `{Authors = “Danny Greefhorst, Matthijs Maat”}`.

4.3 Stereotypes

Stereotypes geven de mogelijkheid bestaande elementen in UML semantisch uit te breiden of te specialiseren waardoor nieuwe soorten elementen kunnen worden gedefinieerd. In plaats van het toevoegen van eigenschappen of beperkingen aan bestaande model-elementen, kan nieuwe semantiek worden toegevoegd aan bestaande elementen. Ontbrekende concepten kunnen op deze manier toegevoegd worden aan UML door gebruik van *specialisatie*; nieuwe concepten kunnen worden geïntroduceerd door eerst een gelijkend bestaand UML-element te vinden en vervolgens de semantiek van dit element uit te breiden of aan te passen. Stereotypes kunnen alleen gebruikt worden om de semantiek van bestaande elementen uit te breiden; de structuur van de elementen blijft echter ongewijzigd.

Er is een groot aantal stereotypes voorgedefinieerd in UML, maar het is dus ook mogelijk zelf nieuwe stereotypes te definiëren. De weergave van een stereotype is in beide gevallen gelijk en bestaat uit de naam van het stereotype tussen guillemets (bijvoorbeeld «`interface`»), welke voor of boven het element wordt weergegeven waarop het betrekking heeft.

Het is ook mogelijk een stereotype aan te duiden met een grafisch icoon. Dit icoon kan zelfstandig of in combinatie met het element of de tekstuele weergave van het stereotype worden gebruikt. Zo is het stereotype «`actor`», een uitbreiding van het model-element `type` welke onder andere in use-cases kan worden gebruikt om externe actoren aan te duiden, geassocieerd met een poppetje. Ook voor zelf gedefinieerde stereotypes kunnen representatieve iconen worden gebruikt.

Indien stereotypes worden gedefinieerd, dan is het belangrijk om te documenteren op welk element het stereotype is gebaseerd, welke semantiek het heeft en hoe het stereotype dient te worden geïmplementeerd. Op deze manier is het mogelijk stereotypes te definiëren voor elk soort element dat men nodig heeft, bijvoorbeeld voor een specifieke ontwikkelmethode.

5 Analyse en Ontwerp met UML

5.1 Notatie versus Methodologie

In de voorgaande twee hoofdstukken is een overzicht gegeven van de belangrijkste eigenschappen van UML. Merk op dat deze hoofdstukken alleen notatie beschrijven; er is beschreven wat een UML-model is, niet hoe je aan zo'n model zou moeten komen. Dit heeft alles te maken met het feit dat UML alleen een modelleringstaal is en géén modelleringsmethodologie. Hoe kunnen zinvolle UML-diagrammen worden afgeleid? Heeft een taal zonder methode eigenlijk wel zin?

Om met de laatste vraag te beginnen, een modelleringstaal is veruit het meest belangrijke onderdeel van een ontwikkelmethode. Bij de communicatie over een systeemontwerp is immers het wederzijds begrip van de notatie cruciaal, niet de manier waarop het ontwerp tot stand is gekomen. Een standaard notatie, het uiteindelijke doel van UML, heeft dus wel degelijk nut.

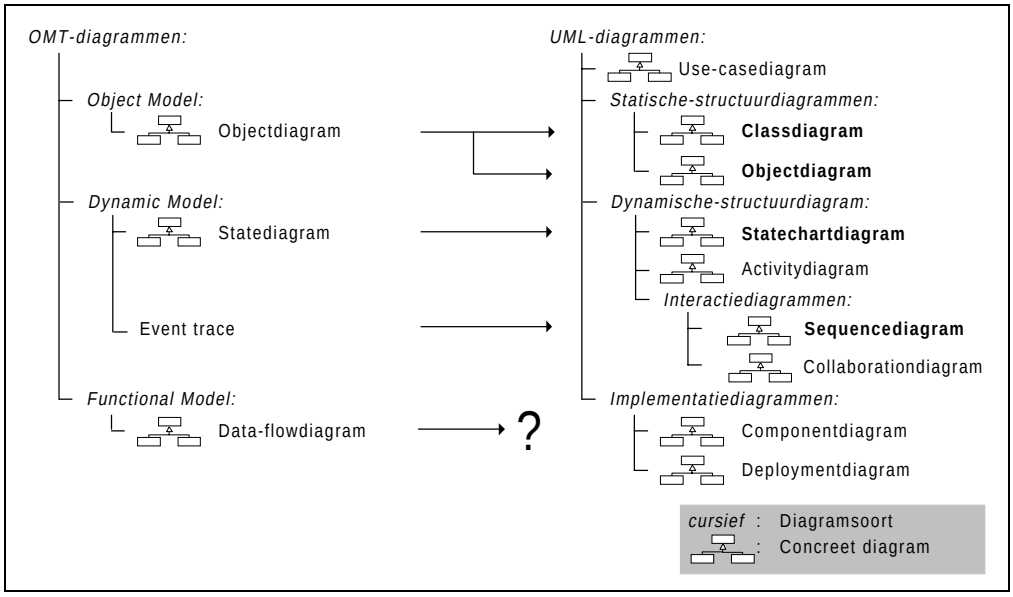
De vraag hoe het ontwikkelproces om te komen tot UML-modellen er uit zou kunnen zien is wat lastiger. UML is in principe methode onafhankelijk; UML kan dus worden gebruikt voor het weergeven van modellen ontwikkeld volgens een willekeurige methode. Eén manier om UML te gebruiken bij software-ontwikkeling is dan ook om UML in te passen in een bestaande ontwikkelmethode zoals bijvoorbeeld Rumbaugh's *Object Modeling Technique* [Rumbaugh91]. Een andere manier is het gebruik van *Objectory*. Booch, Jacobson en Rumbaugh hebben niet alleen een poging ondernomen tot unificatie van hun notatie, maar ook van hun processen. Het resultaat van deze onderneming, Objectory, is een nieuw object-georiënteerd ontwikkelproces, gebaseerd op UML. Dit hoofdstuk beschrijft beide mogelijkheden tot het gebruik van UML.

5.2 UML en OMT

OMT [Rumbaugh91] is een veel gebruikte object-georiënteerde ontwikkelmethode. De methode laat zich gebruiken als een soort recept voor systeemontwikkeling; OMT beschrijft alle benodigde stappen om te komen tot een consistent model van het te ontwikkelen systeem. Het resulterende OMT-analysemodel bestaat uit drie onderdelen:

- **Object Model** — Een beschrijving van de statische structuur van het systeem, uitgedrukt in een *objectdiagram* bestaande uit klassen, objecten, relaties, attributen en operaties.
- **Dynamic Model** — Een beschrijving van het dynamisch gedrag van het systeem, uitgedrukt in *statediagrammen* voor elke klasse met interessant gedrag. Tijdens de analyse van het dynamisch gedrag wordt ook gebruik gemaakt van *event traces*.
- **Functional Model** — Een specificatie van berekeningen en daarvoor benodigde data, weergegeven in *data-flowdiagrammen.*

De meeste concepten uit de OMT-modellen zijn terug te vinden in UML, zei het vaak met een wat andere (en uitgebreidere) notatie. In figuur 13 is weergegeven welke OMT-diagrammen equivalenten hebben in UML.



Figuur 13 Afbeelding van OMT-modellen naar UML

Zoals uit de figuur blijkt, zijn er ten opzichte van de OMT-modellen in UML twee wijzigingen. De eerste wijziging is dat in UML event traces opgewaardeerd zijn tot volwaardig onderdeel van het model in de vorm van sequencediagrammen. Een meer ingrijpende wijziging is dat in UML geen equivalent meer te vinden is van het OMT functionele model. Activitydiagrammen kunnen weliswaar gebruikt worden om beperkte object-flow te modelleren, maar de nadruk ligt daar toch meer op procedurele control-flow dan op data-flow.

Hoewel UML dus in principe methode onafhankelijk is, kan de notatie niet zomaar naadloos in bestaande ontwikkelmethoden worden ingepast. Enerzijds komt dit doordat in UML wel veel, maar niet alle concepten uit eerdere notaties zijn opgenomen. Een voorbeeld hiervan is dat het functionele model, één van de resultaten uit het OMT-analysemodel, niet

direct omgezet kan worden naar een UML equivalent. Anderzijds is dit te wijten aan het feit dat UML juist een veel grotere uitdrukkingskracht biedt dan notaties uit eerdere ontwikkelmethoden; een methode als OMT biedt bijvoorbeeld geen handvatten om activitydiagrammen te ontwikkelen. Op het iets beperktere gebied van statische modellering middels class- en objectdiagrammen, het in de praktijk meest gebruikte onderdeel van object-georiënteerde modellering, is de aansluiting tussen UML en OMT beter: UML biedt in principe alle mogelijkheden van OMT, en is zelfs nog iets uitgebreider.

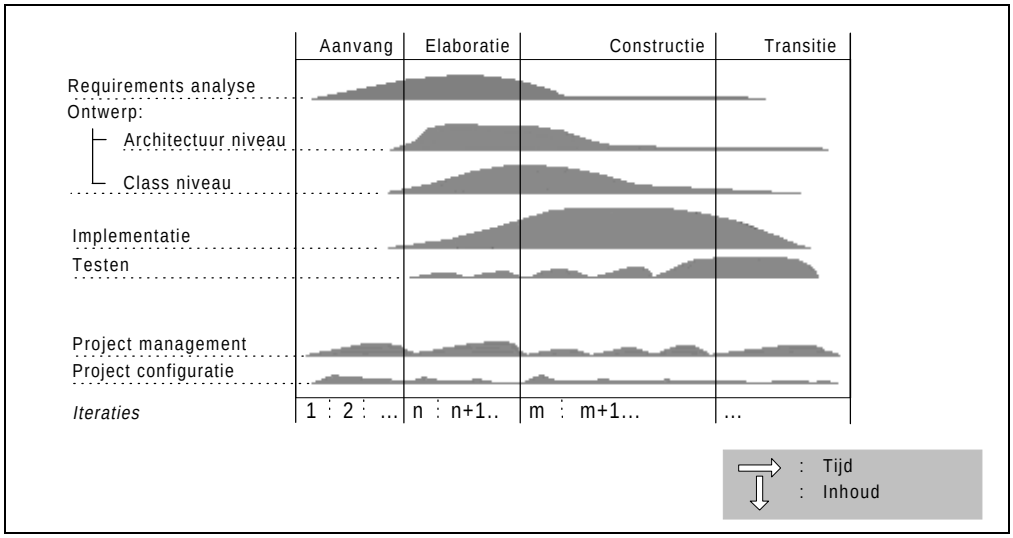
5.3 UML en Objectory

Booch, Jacobson en Rumbaugh hebben niet alleen een poging ondernomen tot unificatie van hun notatie, maar ook van hun object-georiënteerde ontwikkelprocessen. Het resultaat van deze onderneming, *Objectory*, is een beschrijving van een nieuw object-georiënteerd ontwikkelproces, gebaseerd op UML.

Objectory is georganiseerd langs twee dimensies:

- **Tijd —** Het ontwikkelproces bestaat uit vier elkaar opvolgende fases (*aanvang*, *elaboratie*, *constructie* en *transitie*), ieder weer bestaand uit een aantal *iteraties*. De fases beschrijven de levenscyclus van het te ontwikkelen systeem, van het oorspronkelijke idee, de aanvang, tot aan de uiteindelijke overdracht van het systeem aan de gebruikers, de transitie.
- **Inhoud —** Binnen het ontwikkelproces zijn verschillende soorten activiteiten te onderscheiden, namelijk *requirements-analyse*, *ontwerp* (zowel op architectuurniveau als op detail-ontwerpniveau), *implementatie* en *testen*.

In elke fase van het proces worden in principe alle soorten activiteiten uitgevoerd; de nadruk op een bepaalde soort activiteiten verschilt echter wel per fase. De structuur van het Objectory proces is grafisch weergegeven in figuur 14.



Figuur 14 Structuur van het Objectory ontwikkelproces

Naast de structuur van het ontwikkelproces beschrijft Objectory ook welke resultaten wanneer en door wie opgeleverd moeten worden, onder andere in termen van de UML-diagrammen.

In vergelijking met een ontwikkelmethode als OMT geeft Objectory een beschrijving van het ontwikkelproces op een hoger abstractieniveau; daar waar OMT ook concrete stappen beschrijft voor het opstellen van de modellen, blijft Objectory op het procesniveau steken. Uitgebreide richtlijnen voor het gebruik van de verschillende soorten diagrammen, zoals bijvoorbeeld te vinden in [Eriksson98], kunnen dit gebrek aan detail gedeeltelijk opvangen, maar een vergelijkbare koppeling tussen proces en notatie als in OMT is met Objectory en UML vooralsnog niet mogelijk. Objectory is overigens momenteel nog in ontwikkeling, de huidige status en het uiteindelijke resultaat zijn daarom nog niet geheel helder. Meer duidelijkheid kan worden verwacht met het verschijnen van [Jacobson98].

6 UML en database-ontwerp

Het ontwikkelen van database-systemen kan niet zonder meer gelijkgesteld worden aan het ontwikkelen van software in het algemeen, het ontwerpen van een database vormt een specifiek onderdeel van het totale software-ontwikkelingsproces. Hoewel afhankelijk van de hoeveelheid vereiste logica, ligt de nadruk bij het ontwerp van een database-systeem meestal vooral op de statische structuur van het systeem. De dynamische aspecten van een systeem zullen voornamelijk vertaald worden naar programmacode die in principe los staat van de eigenlijke database. Tijdens het ontwerpen van een database-systeem dienen een aantal database-specifieke beslissingen te worden genomen, zoals onder andere de keuze voor het type database (object-georiënteerd of relationeel) en de vertaling van het analyse- en ontwerpmodel naar een concrete database-representatie. In dit hoofdstuk zullen de database-ontwerpspecifieke beslissingen en de rol van UML in het database-ontwerpproces verder worden toegelicht.

De rol van object-georiënteerde ontwikkelmethoden bij het ontwerpen van database-systemen wordt onder meer behandeld in [Blaha93] en [Blaha98]. Database-ontwerp wordt beschreven als onderdeel van het OMT-ontwikkelproces, waarbij database-specifieke zaken geïntegreerd zijn in de ontwerp- en implementatiefasen. De analysefase, de fase in OMT voorafgaand aan het ontwerp en de uiteindelijke implementatie, verschilt voor database-systemen niet wezenlijk van de analysefase voor andere systemen. Dit komt vooral omdat tijdens de analyse de nadruk hoofdzakelijk ligt op het identificeren van domeinconcepten, niet op de implementatie of opslag hiervan. Het analyseproces voor database-systemen wordt hier verder dan ook buiten beschouwing gelaten. Een ontwerp bestaat in OMT typisch uit twee delen: een systeemontwerp en een detail- of objectontwerp. De database-specifieke aandachtspunten bij het opstellen van OMT-ontwerpmodellen worden hieronder besproken.

6.1 Systeemontwerp

In de eerste plaats dient tijdens het systeemontwerp een architectuur voor het te ontwikkelen database-systeem te worden opgesteld. In [Blaha93] worden een aantal aandachtspunten voor het ontwerpen van een systeem beschreven. Zo is het bijvoorbeeld van belang of het te ontwikkelen systeem bedrijfskritisch of meer ondersteunend van aard is, dat de applicatielogica wordt gescheiden van de gebruikersinterface en dat er een afweging gemaakt wordt tussen het gebruik van database queries en programmacode. Vervolgens dient een keuze gemaakt te worden voor een specifiek gegevensopslag-paradigma, waarbij kan gekozen worden voor een *relationeel* of *object-georiënteerd*

database-systeem of eventueel zelfs voor een *plat bestandsformaat*. Een derde belangrijke afweging is het bepalen van de wijze waarop de gegevens uiteindelijk zullen worden benaderd. Dit kan bijvoorbeeld door dynamisch gegenereerde SQL-statements naar de database te sturen of door de statements in te passen in de programmacode. Een verdergaande integratie tussen database en programmacode is mogelijk door gebruik te maken van een object-georiënteerde database of een object-georiënteerd/relationeel adapterpakket.

Ook op een iets lager abstractieniveau moeten een aantal keuzen gemaakt worden. Zo moet bepaald worden op welke wijze de *objectidentiteit* wordt vertaald naar de database. In object-georiënteerde databases wordt hiervoor voornamelijk gebruik gemaakt van zogenaamde *object identifiers* die door het systeem worden gegenereerd. Voor relationele databases kan naast gegenereerde identifiers eventueel ook gebruik worden gemaakt van combinaties van attributen, de zogenaamde *primary keys* of *primaire sleutelwaarden*, die een object uniek identificeren. Tenslotte zijn er in het systeemontwerp nog een aantal andere zaken die dienen te worden vastgelegd, bijvoorbeeld zaken met betrekking tot het omgaan met schema-evolutie, historische gegevens, beveiliging, metagegevens, distributie, backup en recovery.

6.2 Detailontwerp

In het detailontwerp wordt het analysemodel uit de OMT-analysefase verder verfijnd en uitgebreid met oplossingsspecifieke constructies. Hierbij kan gebruik gemaakt worden van transformaties die de analysemodellen vertalen naar semantisch vergelijkbare modellen. Een voorbeeld van een dergelijke transformatie is het samenvoegen van twee klassen tot één klasse, waarbij alle attributen in de nieuwe klasse worden ondergebracht.

Tijdens het detailontwerp is er daarnaast een veelvoud aan andere database-specifieke beslissingen die genomen dienen te worden. Met betrekking tot attributen dienen bijvoorbeeld *keys* of *sleutels* te worden onderkend, *domeinen* te worden aangegeven en moet besloten worden of ook *null-waarden* voor kunnen komen. Daarnaast moet er gekeken te worden naar het verwachte aantal instanties van klassen en associaties, het toekennen van *indexen* en het voorspellen van query-gedrag. Er is een duidelijke afweging tussen *encapsulatie* en *optimalisatie*; kennis van de database-structuur kan leiden tot een betere prestatie.

6.3 Implementatie

In de implementatiefase tenslotte wordt een vertaling gemaakt van de gebruikte modellen naar een concrete database-representatie. Er is een duidelijk onderscheid tussen object-

georiënteerde en relationele databases, omdat er in het laatste geval een duidelijke *impedance mismatch* bestaat tussen de uitdrukkingskracht van de gebruikte object-georiënteerde modelleringstaal en de relationele database-representatie. Deze mismatch ontstaat ondermeer doordat object-georiënteerde concepten als *methoden*, *overerving* en *objectreferenties* ontbreken in het relationele paradigma.

Voor een relationele database dienen de klassen en relaties uit het objectmodel vertaald te worden naar tabellen. Een klasse wordt hierbij in het meest eenvoudige geval vertaald naar één tabel, en elk attribuut naar een kolom. Associaties kunnen worden vertaald naar een aparte tabel waarin de primaire sleutels van de gerelateerde entiteiten zijn opgenomen. Indien een associatie aan één of meerdere zijden multipliciteit één heeft dan kan deze het best vertaald worden door het opnemen van de sleutel in de gerelateerde tabel. Generalisatierelaties worden in het meest eenvoudige geval vertaald door aparte tabellen voor zowel sub- als superklassen, waarbij de primaire sleutel wordt gedeeld.

Naast het maken van een vertaling van het object-georiënteerde model dient er voor relationele databases aandacht te worden besteedt aan de *referentiële integriteit* van het systeem, *cascaded deletes* kunnen er voor zorgen dat niet alleen rijen uit één tabel, maar ook de corresponderende rijen uit gerelateerde tabellen worden verwijderd. Tenslotte kan gebruik gemaakt worden van *views* om tabellen die deel uitmaken van een generalisatiehiërarchie te bekijken en soms zelfs consistent te wijzigen, kunnen database-specifieke optimalisaties met betrekking tot ruimtegebruik van tabellen worden ingesteld en dient een duidelijke en robuuste SQL-stijl te worden gehanteerd.

Bij het gebruik van een object-georiënteerde database is de vertaling van het ontwerpmodel naar een database-representatie relatief eenvoudig. Zaken die wel aandacht verdienen zijn het onderscheid tussen *persistente* en *transient* objecten, het gebruik van *klassebibliotheken* van derden die niet in alle gevallen persistent gemaakt kunnen worden, de vaak slechts beperkte mogelijkheden tot toegang en gebruik van *metadata* tijdens executie en de soms nogal omslachtige *castingregels*.

6.4 Database-ontwerp met UML

[Blaha93] beschrijft hoe een object-georiënteerde ontwikkelmethode als OMT uitgebreid zou kunnen worden voor gebruik bij database-ontwerp. Hoe past UML in dit geheel?

Zoals eerder besproken bestaat UML uit een aantal verschillende views met bijbehorende diagrammen, zie figuur 2 in hoofdstuk 3. De verschillende views belichten elk een specifiek aspect van een applicatie. Het statische aspect, over het algemeen juist het belangrijkste aspect bij het ontwerp van databasesystemen, wordt hierbij uitgedrukt in classdiagrammen, waarbij ter illustratie eventueel gebruik gemaakt kan worden van objectdiagrammen. Deze diagrammen komen wat uitdrukkingskracht betreft sterk overeen met diagrammen uit

traditionele database-ontwerpmethoden als ER en NIAM en vergelijkbare diagrammen in bestaande ontwikkelmethoden als bijvoorbeeld OMT en Booch. In UML is dus nauwelijks sprake van extra uitdrukkingskracht; het gevolg hiervan is dat database-ontwerp met UML niet fundamenteel verschilt van database-ontwerp met behulp van bestaande notatiemechanismen. Het gebruik van UML heeft als grote voordeel dat een notatiestandaard voorhanden is, niet dat het database-ontwerpproces zelf vereenvoudigd of verbeterd wordt.

Overeenkomstig de observatie in het vorige hoofdstuk (UML is notatie, géén methode) dient UML ingepast te worden in een ontwikkelmethode voordat geprofiteerd kan worden van de nieuwe standaard. Eén mogelijkheid daartoe is de in dit hoofdstuk besproken uitbreiding op OMT.

7 Samenvatting en Conclusie

De Unified Modeling Language is de volgende stap in de evolutie van object-georiënteerde notatiemechanismen. Gebaseerd op de invloedrijke notaties uit ontwikkelmethoden als OMT, Booch en OOSE biedt UML een standaard representatie voor de meest gebruikte concepten en ideeën uit de object-georiënteerde praktijk. UML lijkt daarmee de oplossing voor de huidige onoverzichtelijke verzameling van verschillende notaties, zeker gezien de brede steun voor UML vanuit zowel de industrie als de standaardisatiewereld. Met het beschikbaar zijn van UML als standaard notatie zullen ook de ontwikkelmethoden meer overeenkomsten gaan vertonen, vooral doordat een eenduidige verzameling modelleringsconcepten voor handen is.

In dit artikel is een globaal overzicht gegeven van de belangrijkste kenmerken van UML. Door de belangrijkste onderdelen van UML aan de hand van voorbeelden te introduceren is antwoord gegeven op de vraag wat UML eigenlijk is en wat er allemaal mee kan. Een vraag die minstens zo relevant is bij de introductie van een nieuw mechanisme is wat dat mechanisme juist niet is. Wat is UML niet? UML is géén ontwikkelmethode, maar alleen een notatie. Dat is op zich geen bezwaar, het hebben van een standaard notatie is al enorm waardevol, maar het geeft wel aan dat UML nog ingepast dient te worden in een object-georiënteerde ontwikkelmethode als OMT of Objectory voordat voluit geprofiteerd kan worden van de nieuwe standaard. De plaats van UML binnen ontwikkelmethoden en, meer specifiek, binnen de ontwikkeling van database-systemen zal niet wezenlijk afwijken van de positie van de momenteel gebruikte datamodelleringstechnieken als ER-modellering en OMT. UML biedt op het gebied van statische modellen, de belangrijkste modellen voor het ontwerpen van databasesystemen, weinig extra uitdrukkingskracht.

De brede steun voor UML vanuit de industrie en door de OMG is essentieel voor het mogelijke succes van UML. Een andere voorwaarde is de beschikbaarheid van ondersteunende CASE-tools. Op het ogenblik ondersteunen bijna alle object-georiënteerde analyse en ontwerptools al in meer of mindere mate het modelleren met behulp van UML. Tools als Rational Rose, Paradigm Plus, ObjectTeam en Select/Enterprise bieden de mogelijkheid tot het ontwikkelen van verschillende soorten UML-diagrammen, de ondersteuning is veelal echter nog niet volledig.

De ontwikkelingen rond UML en ook Objectory zijn bij het verschijnen van dit artikel nog in gang. UML versie 1.1, door de het UML Partner consortium bij OMG ingebracht als antwoord op het *Request for Proposal*, is intussen aanvaard als standaard. Dit betekent echter niet dat UML niet meer aangepast zal worden. Een volledige beschrijving van UML 1.1 is voorlopig alleen beschikbaar als documentserie van de Rational Software Corporation [Rational97b], onder meer bestaande uit een redelijk formele beschrijving van de UML-semantiek en -notatie. In de loop van 1998 wordt een drietal boeken van Booch, Rumbaugh

en Jacobson , de grondleggers van UML, verwacht met een complete en meer uitgebreide beschrijving van zowel UML als Objectory ([Booch98], [Jacobson98] en [Rumbaugh98]). Recente informatie over UML is te vinden in het *UML Resource Center*, een pagina op de WEB-site van de Rational Software Corporation: `http://www.rational.com/uml/`.

8 Referenties

[Blaha93] Michael R. Blaha, William J. Premerlani, *Object-Oriented Concepts for Database Design*, Fifth Annual Software Technology Conference, Salt Lake City, USA, 1993.

[Blaha96] Michael R. Blaha, William J. Premerlani, *A Catalog of Object Model Transformations*, Third Working Conference on Reverse Engineering, Monterey, USA, 1996.

[Booch94] Grady Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, 1994.

[Eriksson98] Hans-Erik Eriksson, Magnus Penker, *UML Toolkit*, John Wiley & Sons,, New York, USA, 1998.

[Florijn95] Gert Florijn, Norbert van Oosterom, *Object-oriëntatie — Klaar voor gebruik?!*, Kluwer Bedrijfswetenschappen, 1995.

[Lee97] Richard C. Lee, William M. Tepfenhart, *UML and C++ - A Practical Guide to Object-Oriented Development*, Prentice Hall, New Jersey, USA, 1997.

[Fowler97] Martin Fowler, Kendall Scott, *UML Distilled - Applying the Standard Object Modelling Language*, Addison-Wesley, Reading, USA, 1997.

[Jacobson94] Ivar Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Reading USA, 1994.

[Rational97a] Rational Software Corporation, *The Rational Objectory Process Version 4.0*, Santa Clara, USA, 1997.

[Rational97b] Rational Software Corporation, *UML document series - version 1.1*, Santa Clara, USA, 1997.

[Rumbaugh91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modelling and Design*, Prentice-Hall, New York, USA, 1991.

[Troyer93] O. de Troyer, *Object-georiënteerde database-modellen*, Handboek Database Systemen, Array Publications, november 1993.

Verwachte literatuur

[Blaha98] Michael R. Blaha, William J. Premerlani, *Object Oriented Modeling and Design for Database Applications*, Prentice Hall, Englewood Cliffs, USA, verwacht 1998.

[Booch98] Grady Booch, James Rumbaugh, Ivar Jacobson, *Unified Modeling Language User Guide*, Addison-Wesley, verwacht voorjaar 1998.

[Elswijk98] Mark van Elswijk, *Software-patronen*, Handboek Database Systemen, Array Publications, verwacht begin 1998.

[Jacobson98] Ivar Jacobson, Grady Booch, James Rumbaugh, *The Objectory Software Development Process*, Addison-Wesley, verwacht zomer 1998.

[Rumbaugh98] James Rumbaugh, Grady Booch, Ivar Jacobson, *Unified Modeling Language Reference Manual*, Addison-Wesley, verwacht zomer 1998.

