design your future

Analysis & Design 2

Pieter Lust

handelswetenschappen en bedrijfskunde

bachelor in de toegepaste informatica

campus Kortrijk

academiejaar 2018-2019

vives katholieke hogeschool associatie KU Leuven

Legende van de gebruikte iconen

	Leerdoelen
	Extra informatie

	Niet vergeten

Inhoudsopgave

Inleiding

Het vak Analysis & Design 2 is een vervolg op en een verdieping van verschillende voorgaande vakken. Het is de bedoeling om dieper in te gaan op objectgeoriënteerd ontwerpen en programmeren.

Deze onderwerpen komen in het vak aan bod:

- Principes van goed objectgeoriënteerd ontwerp. Dit zijn een aantal vuistregels die een programma beter onderhoudbaar helpen maken.
- Design patterns (ontwerppatronen). Design patterns zijn standaardmanieren om bepaalde programmataken te implementeren, die in de praktijk hun deugdelijkheid bewezen hebben.

  Dit vak behandelt vooral de zogenaamde 'Gang of Four'-patterns, die gaan over eerder kleinschalige programmataken, zoals creatie van nieuwe objecten, of de verantwoordelijkheden van objecten aanpassen, of objecten met elkaar laten samenwerken.

  Er zijn ook patterns op architectuurniveau, waarvan MVC een van de bekendste is.
- Sequentie- en collaboratiediagrammen. Deze diagramtechnieken maken deel uit van UML. Het zijn diagrammen waarmee je modelleert hoe objecten met elkaar samenwerken.

Deze cursus behandelt de eerste twee onderwerpen. Design patterns komen aan bod in het handboek (Bevis, 2012).

Niet vergeten

In dit vak wordt voor de voorbeelden vooral Java gebruikt. Het is echter belangrijk te beseffen dat design patterns in eender welke objectgeoriënteerde taal toegepast kunnen worden! Dus niet enkel Java, maar ook C#, Visual Basic.NET, C++, … . JavaScript is in dit opzicht wat apart. Het is een taal die objecten kent, maar die breekt met een aantal conventies van het 'klassieke' objectgeoriënteerde paradigma. Daardoor zijn er in JavaScript vaak heel andere manieren om te doen wat je in de 'klassieke' talen zou doen met een Gang of Four-pattern.

Leerdoelen

Deze leerdoelen zijn voor het vak als geheel, niet enkel voor deze cursus.

Om te slagen voor dit vak moet je:

- ✓ van de behandelde design patterns de correcte Engelse naam kennen
- ✓ kunnen uitleggen wat het doel is van een bepaald design pattern
- ✓ kunnen uitleggen welk pattern het meest geschikt is voor een bepaald doel
- ✓ de structuur van elk pattern kunnen tekenen met een klassendiagram
- ✓ de verantwoordelijkheid van elke deelnemer in een pattern kunnen beschrijven
- ✓ de werking van een pattern kunnen modelleren met sequentiediagrammen en collaboratiediagrammen
- ✓ de patterns kunnen toepassen op een eenvoudige voorbeeldsituatie
- ✓ de verschillende vuistregels voor goed objectgeoriënteerd ontwerp kunnen uitleggen, en kunnen uitleggen waarom ze bijdragen tot een beter onderhoudbaar programma
- ✓ kunnen uitleggen hoe een design pattern een toepassing is van bepaalde vuistregels voor goed objectgeoriënteerd ontwerp

1 Interactiediagrammen: sequentie- en collaboratiediagrammen

Het dynamische gedrag van een applicatie kan voorgesteld worden met sequentiediagrammen en collaboratiediagrammen. Deze diagrammen zijn wat we noemen *interactiediagrammen*. In dit hoofdstuk wordt uitgelegd wat sequentie- en collaboratiediagrammen precies zijn, wat hun bedoeling is, en hoe je ze tekent.

1.1 Wat zijn interactiediagrammen?

In de cursus Analysis & Design 1 werden al een aantal diagramtechnieken behandeld. De statische structuur van een applicatie modelleren we met een klassendiagram, de toestandsveranderingen van een object met een toestandsdiagram, de functionele eisen met een use-cases-diagram, de workflow van een organisatie met een activiteitendiagram, de softwarearchitectuur met een component-diagram en de hardwarearchitectuur met een opstellingsdiagram.

Een klassendiagram modelleert de statische structuur van een applicatie, dus welke objecten er zijn, en welke relaties die objecten met elkaar hebben. Maar objecten zijn geen passieve, onveranderlijke dingen. In een werkend progamma reageren zij op bepaalde gebeurtenissen, zoals een gebruiker die een knop indrukt, een alarmsignaal van een machine of een ander object dat een boodschap stuurt met een bepaald verzoek (een operatie-aanroep). Het dynamisch gedrag van één dergelijk object tonen we met een toestandsdiagram, maar objecten sturen ook berichten naar elkaar, roepen elkaars operaties aan, doen beroep op elkaar voor het uitvoeren van bepaalde diensten. Hoe objecten met elkaar samenwerken, kunnen we tonen met *interactiediagrammen*.

Met een interactiediagram kunnen we een bepaald *scenario* modelleren. Een scenario is een opeenvolging van gebeurtenissen, bijvoorbeeld de concrete realisatie van een use-case of de uitvoering van een event. Interactiediagrammen tonen hoe de betrokken onderdelen van het systeem reageren in bepaalde situaties.

Interactiediagrammen zijn nuttig op verschillende niveaus. Ze kunnen tonen hoe verschillende systemen of subsystemen met elkaar samenwerken. Bijvoorbeeld: hoe een kassasysteem met een inventarissysteem samenwerkt om bij elke verkoop de inventaris up-to-date te houden. Ze kunnen ook tonen hoe concrete objecten (dus instanties van klassen) met elkaar samenwerken.

Niet vergeten

De term 'object' heeft in dit hoofdstuk een ruimere betekenis dan die van 'instantie van een klasse'. Het kan ook een instantie van een subsysteem zijn.

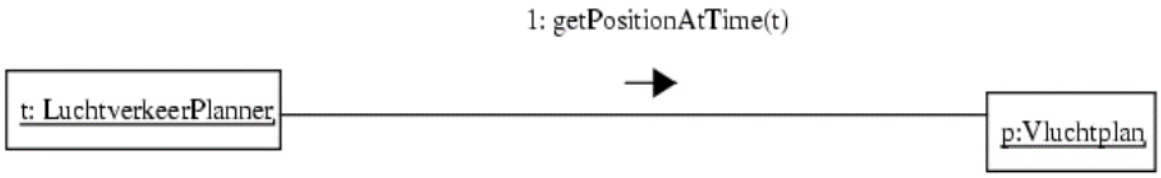
Bij het uitvoeren van acties werken objecten samen. Zij sturen *berichten* naar elkaar (in het Engels: *messages*). Wanneer een object A een bepaalde dienst van een object B nodig heeft om zijn taak uit

te voeren, dan zal A een bericht sturen naar B. Dat wil zeggen: A gebruikt de API van B, A roept methodes van B aan.

In het voorbeeld in figuur 1 stuurt het object `t` een bericht naar het object `p`. Het bericht heeft de naam `getPositionAtTime`, en het heeft een parameter `t`. In code (bijvoorbeeld in Java) zou dat er allicht zo uit zien:

```
position = p.getPositionAtTime(currentTime);
```

(`position` en `currentTime` zijn dan variabelen van het programma.)



Figuur 1: bericht versturen

Niet vergeten

Hoe zie je dat er in het diagram objecten staan, en geen klassen?

- In de rechthoeken staat tekst met een dubbelepunt. Objecten hebben een naam (het stuk voor de dubbelepunt) en een type (het stuk na de dubbelepunt). Klassen kunnen enkel een type hebben, dus daar geen dubbelepunt.

  “p:Vluchtplan” is een object van type Vluchtplan, met de naam p.

  “:Vluchtplan” is ook een object van type Vluchtplan, maar dan zonder naam.

  “Vluchtplan” is een klasse.

- Naam en type zijn onderstreept.

  Onderstrepen gebeurt niet altijd (zie bijvoorbeeld de meeste figuren in dit hoofdstuk). Sommige softwarepakketten laten de onderstreping weg (bijvoorbeeld: Visual Paradigm). Als je een diagram met de hand schetst wordt de onderstreping ook dikwijls weggelaten.

Bij het modelleren van scenario’s kunnen we de nadruk leggen op twee verschillende zaken:

- We kunnen ons concentreren op het tijdsverloop, op de opeenvolging van boodschappen in de tijd. In dat geval gebruiken we een *sequentiediagram (sequence diagram)*.
- We kunnen ons ook concentreren op het structurele aspect, op de relaties tussen objecten en hoe zij samenwerken. In dat geval gebruiken we een *collaboratiediagram (collaboration diagram)*.

Sequentiediagrammen en collaboratiediagrammen zijn de twee soorten interactiediagrammen. Van die twee soorten worden sequentiediagrammen veruit het meest gebruikt.

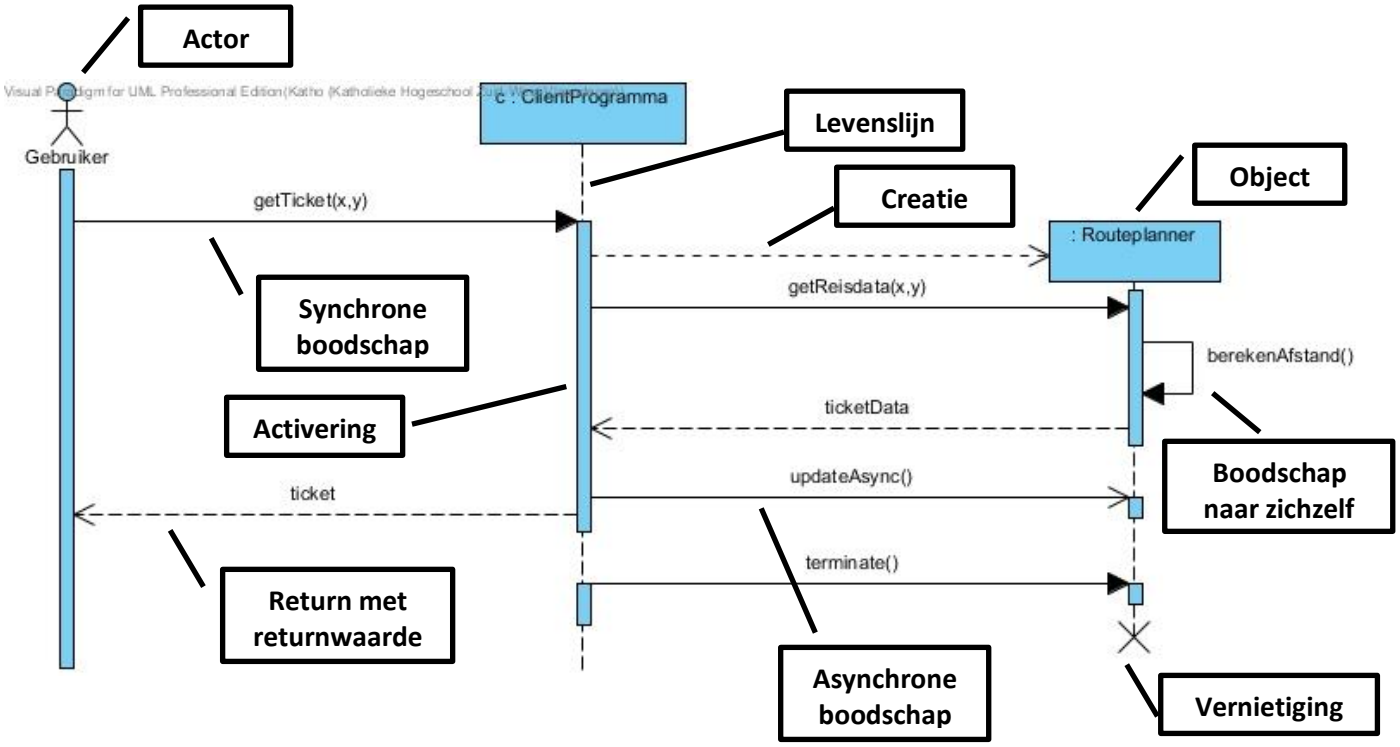
1.2 Sequentiediagrammen

1.2.1 Doel

Een sequentiediagram toont de samenwerking tussen objecten, waarbij de nadruk ligt op het tijdsverloop. Een sequentiediagram is dus heel geschikt om te tonen in welke volgorde de deelnemers in een interactie boodschappen sturen naar elkaar.

1.2.2 Basiselementen

Laten we de belangrijkste componenten van een sequentiediagram eens bekijken. Ze worden ook aangeduid op figuur 2.



Figuur 2: elementen van een sequentiediagram

Tijd

De tijd staat niet expliciet aangegeven op een sequentiediagram.

In een sequentiediagram loopt de tijd van boven naar onder. Wat hoger staat gebeurt eerst, wat lager staat gebeurt later.

Voorbeeld: in figuur 2 is het eerste wat gebeurt dat de actor Gebruiker een boodschap getTicket stuurt naar het object c:ClientProgramma. Het volgende is de creatie van een object :Routeplanner door het object c:ClientProgramma. Daarna stuurt c:ClientProgramma de boodschap getReisdata naar :Routeplanner.

Object

Een object wordt aangeduid in een rechthoek.

Voorbeeld: c:ClientProgramma is een object van het type ClientProgramma, met de naam c.

Wanneer we geen specifieke naam aan het object willen geven, dan laten we de naam gewoon weg. We schrijven dan enkel de klassenaam, voorafgegaan door een dubbelepunt.

Voorbeeld: :Routeplanner is een anoniem object van het type Routeplanner.

Actor

De start van een sequentie kan gegeven worden door een externe actor. We stellen een actor op dezelfde manier voor als in use-casesdiagrammen: een 'stickman' met de naam van de actor er onder.

Niet alle sequentiediagrammen hebben een actor.

Levenslijn

De levenslijn is een lijn die onder een object hangt. Elk object en elke actor heeft een levenslijn.

Het uitzicht van de lijn toont de toestand van het object. Een stippellijn geeft aan dat het object bestaat, maar niet actief is. Een dikke balklijn geeft aan dat het object actief betrokken is bij de sequentie (zie volgende item: activering).

Boodschappen vertrekken van een levenslijn en komen aan op een levenslijn.

Activering

Wanneer een object een boodschap krijgt, wordt het actief. De boodschap zorgt er voor dat één van de operaties van het object uitgevoerd wordt.

De activering van een object duiden we aan met balkje dat iets dikker is dan de levenslijn. De activering begint bij de operatieaanroep en eindigt wanneer de operatie ten einde is (en eventueel een returnwaarde teruggeeft).

Wanneer je een sequentiediagram tekent met software, dan zorgt de software voor het goede uitzicht van de levenslijnen. Wanneer je een sequentiediagram schetst op papier, dan kun je de balklijn eventueel vervangen door een dikke volle lijn.

Boodschap (message)

Een boodschap is een operatieaanroep, een verzoek aan een object om iets uit te voeren of om informatie te geven.

Een boodschap is een horizontale pijl. De pijl wijst naar het object dat de boodschap krijgt. De pijl vertrekt bij het object dat de boodschap stuurt.
Boven de pijl schrijf je de naam van de boodschap. Eventueel kun je de naam laten volgen door de lijst met parameters tussen haakjes. Anders schrijf je enkel de haakjes.

Voorbeeld: in figuur 2 is getReisdata() een boodschap die door c:ClientProgramma naar het object :Routeplanner gestuurd wordt. In softwaretermen: Clientprogramma roept de methode getReisdata()

van Routeplanner aan.
De boodschap heeft twee parameters: x en y.

Een *synchrone boodschap* (d.w.z. een boodschap waarbij het aanroepende object wacht om verder te gaan tot de aangeroepen operatie is uitgevoerd) duiden we aan met een pijl met een gesloten pijlpunt.
Methode-aanroepen in Java zijn een voorbeeld van synchrone boodschappen.

Een *asynchrone boodschap* (d.w.z. een boodschap waarbij het aanroepende object niet wacht op het einde van de operatie) duiden we aan met een pijl met een open pijlpunt.

Voorbeeld: in figuur 2 is getReisdata() een synchrone boodschap, en updateAsync() een asynchrone boodschap.

Creatieboodschap

Een constructor of creatieboodschap duiden we aan met een pijl in streeplijn die wijst naar het object dat gecreëerd wordt.

Een alternatief is om de creatie te tekenen met een gewone boodschap met de naam <<create>>.

Boodschap naar zichzelf

Een object kan ook een boodschap naar zichzelf sturen. We kunnen de activeringsbalk dan verdubbelen, maar dat hoeft niet.

Vernietiging

Wanneer een object vernietigd wordt, dan beëindigen we de levenslijn onderaan met een kruis.

Zo een kruis wordt meestal enkel getekend in het geval van een expliciete vernietiging, dus als in het programma instructies staan om het object te vernietigen. Voor automatische vernietiging door de garbage collector teken je geen kruis.

Return

Operaties geven vaak een returnwaarde. Een return duiden we aan met een stippelpijl.
Return mag je weglaten als dat het diagram beter leesbaar maakt.

Je kunt de returnwaarde bij een return aangeven, maar ook dat is niet verplicht.

Niet vergeten

Je hebt bij het tekenen van een sequentiediagram wat vrijheid over de mate van detaillering.
Hoeveel detail is nodig? Dat is een kwestie van gezond verstand. Het doel van het diagram is om te tonen hoe softwareobjecten met elkaar samenwerken. Als het diagram die boodschap duidelijk overbrengt, dan is het goed.

1.2.3 Oefening

Nieuw telefoonnummer

Teken het volgende scenario met een sequentiediagram:

Een object van de klasse `ContactControl` krijgt een message (van een onbekende bron):
`setTelnr('Pieters Jan','056805612').`

Dit betekent dat het telefoonnummer van de klant "Pieters Jan" gewijzigd moet worden en de waarde "056805612" moet krijgen. De gegevens van Jan Pieters zullen uit de database worden gehaald, gewijzigd worden en weer weggeschreven worden.

Daarvoor creëert `ContactControl` een object van de klasse `ConctactDb`. Aan dit object stuurt het de boodschap
`getContact('Pieters Jan')`.

`ContactControl` krijgt een object terug van de klasse `Contact`. Die klasse heeft een operatie
`setTelefoonnummer(telnr: String)`

Die operatie wordt uitgevoerd. Daarna wordt bij `ContactDb` de operatie
`storeContact(c: Contact)`
aangeroepen om Jan Pieters weer op te slaan.

Teken een sequentiediagram dat met dit scenario overeenkomt.

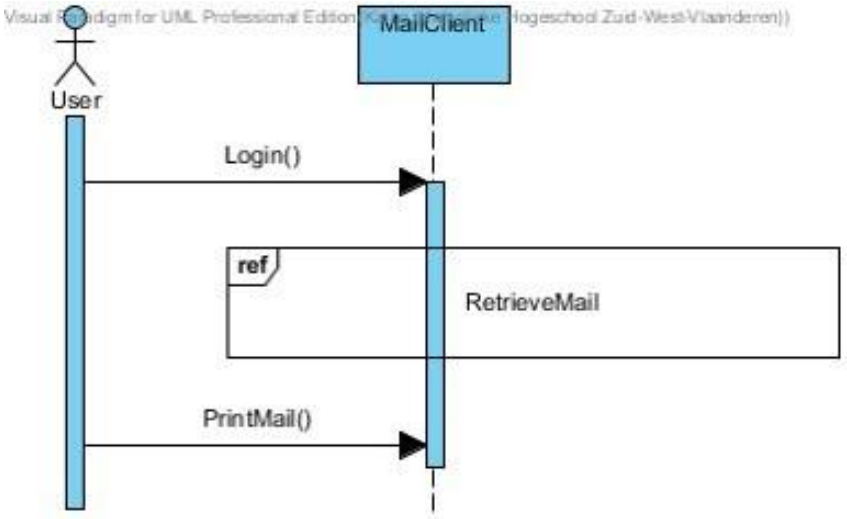
Design patterns

Maak diagrammen die de werking van de gang of four design patterns uit het handboek illustreren.

Voorbeeldoplossingen vind je op Toledo.

1.2.4 Verwijzing, tijdsbeperking, alternatieven, iteratie

Sequentiediagrammen kunnen **verwijzen** naar andere diagrammen, waar een deel van de interactie meer in detail uitgewerkt wordt. Een dergelijke verwijzing wordt getekend als een *frame*. Een frame is een rechthoek met een label in de linkerbovenhoek. Voor een verwijzing is het label 'ref'. De naam van het diagram waarnaar verwezen wordt staat in het frame. (Voorbeeld: figuur 3)



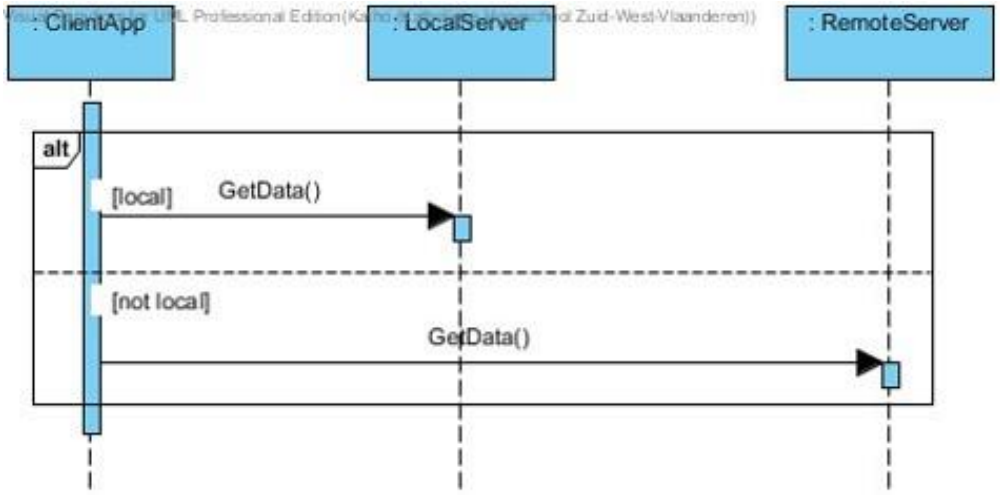
Figuur 3: verwijzing naar een ander sequentiediagram

Tijdsbeperkingen kun je aangeven als tekst tussen accolades. (Voorbeeld: figuur 4. De tijd tussen de boodschappen EnterTransaction en ConfirmTransaction mag hoogstens 300 seconden zijn.)

Figuur 4: tijdsbeperking

Soms heeft een scenario 2 **alternatieve verlopen**. Dat kun je modelleren met een frame met het label 'alt'. Het frame is met horizontale stippellijnen onderverdeeld in twee of meer delen. Elk deel is een alternatief verloop van de sequentie. De voorwaarde om voor het ene of het andere alternatief te kiezen (de 'guards') moet je schrijven als tekst tussen vierkante haken.

Figuur 5 toont een voorbeeld: :ClientApp kan ofwel GetData() aanroepen op een lokale server, ofwel GetData() aanroepen op een niet-lokale server. De guards ([local], [not local]) geven aan wanneer welk alternatief van toepassing is.

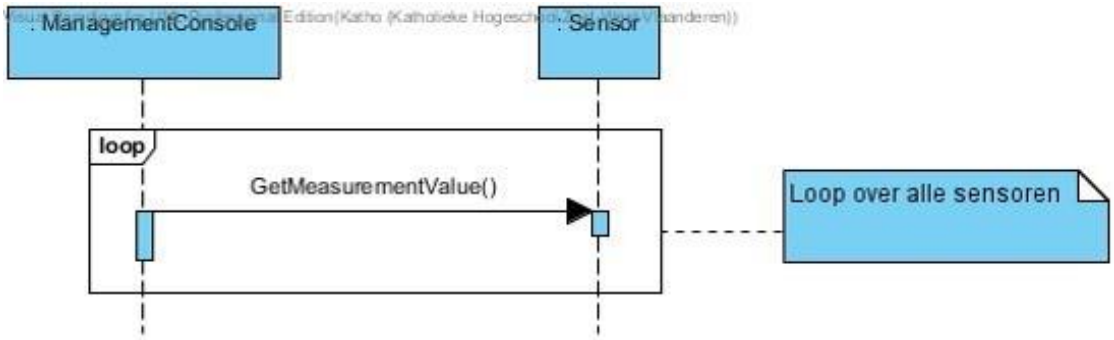


Figuur 5: alternatieve verlopen

In een scenario kan het soms gaan over een **iteratie** over een reeks van gelijkaardige objecten. Dat modelleer je met een frame met het label 'loop'. Eventueel kun je in een note wat meer uitleg geven over de iteratie.

Figuur 6 toont een voorbeeld: de ManagementConsole stuurt de boodschap GetMeasurementValue() niet naar 1 Sensor, maar naar alle leden van een collectie Sensor-objecten. In pseudocode:

```
for (sensor : Sensors) { sensor.GetMeasurementValue(); }
```



Figuur 6: een frame met een iteratie

Extra informatie

Niet alles wat je met sequentiediagrammen kunt doen is hier aan bod gekomen. Meer uitgebreide informatie kun je vinden in de literatuur (bijvoorbeeld: (Warmer & Kleppe, 2011)), of op het web (zoekterm: 'UML sequence diagram').

1.3 Collaboratiediagrammen

Een andere naam voor collaboratiediagram is *communicatiediagram*.

1.3.1 Doel

Een sequentiediagram toont de samenwerking tussen objecten, waarbij de nadruk ligt op de relaties tussen de verschillende objecten.

1.3.2 Elementen

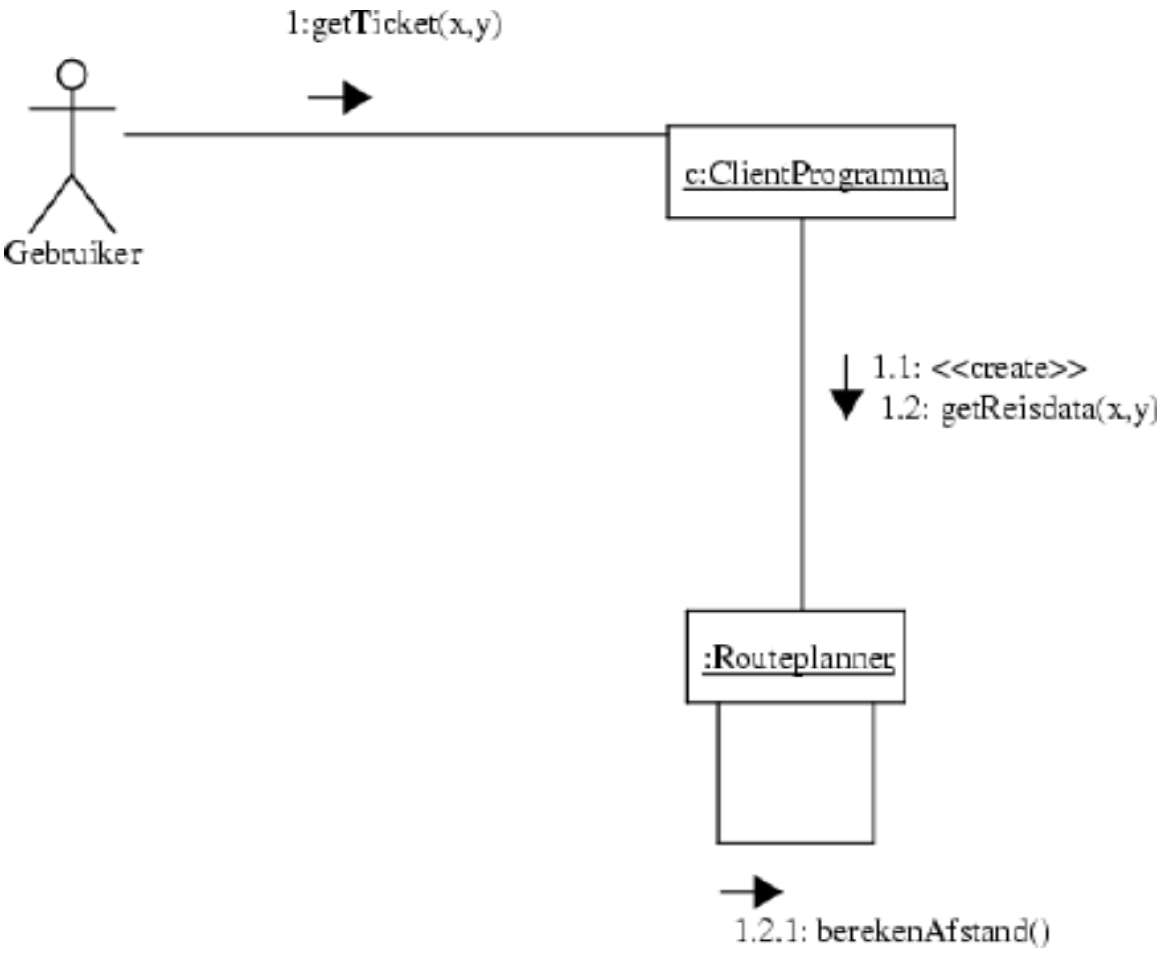
Een collaboratiediagram lijkt wat op een klassendiagram: rechthoeken verbonden met lijnen. Maar een klassendiagram toont klassen, en een collaboratiediagram toont objecten.

De boodschappen worden geschreven naast de associatielijnen, met een kort pijltje om duidelijk te maken welk object de boodschap stuurt en welk object de boodschap ontvangt.

De volgorde van boodschappen wordt aangegeven door ze te nummeren. Er zijn twee stijlen van nummeren mogelijk: ofwel doorlopend (1, 2, 3, …) , ofwel met niveaus (1, 1.1, 1.2, 1.2.1, 1.3, 2, …). De tweede stijl benadrukt de hiërarchie van functieaanroepen. Als boodschap 1 een object A activeert, dan is boodschap 1.1 de eerste boodschap die A als reactie aanroept op een ander object. Je bent vrij om de stijl te kiezen die je zelf het duidelijkste vindt.

Figuur 7 toont een voorbeeld. Het is een stuk van de interactie uit figuur 2 (de boodschappen updateAsync() en terminate() ontbreken hier).

Returnboodschappen zijn hier weggelaten: ze zouden de figuur te overladen maken.



Figuur 7: collaboratiediagram

De meeste case-tools kunnen sequentiediagrammen omzetten naar collaboratiediagrammen, en omgekeerd.

1.3.3 Oefening

Nieuw telefoonnummer

Zet de oefening uit 1.2.3 om naar een collaboratiediagram.

Een voorbeeldoplossing vind je op Toledo.

2 Objectgeoriënteerd werken en de SOLID-principes

In dit hoofdstuk worden kort een aantal basisprincipes van objectgeoriënteerd werken herhaald.

De SOLID-principes zijn een poging om een aantal basisprincipes voor goed objectgeriënteerd ontwerp samen te brengen in een letterwoord dat in het geheugen blijft hangen.

2.1 Wat is een object?

Een object is een stuk software, een samenhangend geheel van data en methodes.

1. Een object heeft een **identiteit**.

   Dat wil zeggen: elk object is uniek. In die zin zijn objecten net als mensen: we lijken allemaal op elkaar, maar elk mens heeft een eigen identiteit. Ik ben iemand, jij bent iemand anders, en dat verschil is voor iedereen duidelijk.

2. Een object heeft een **toestand.**

   Een object kan verschillende toestanden hebben. Die toestand wordt bijgehouden in de *attributen* van het object.

3. Een object heeft een **gedrag**.

   Een object kan dingen doen. Het gedrag wordt bepaald door de *methodes* van het object.

Objecten met gelijkaardige toestanden en gelijkaardig gedrag worden samengebracht in een *klasse*. Elk object is een *instantie* van 1 klasse, en een object kan nooit van klasse veranderen.

Van een object kun je enkel de toestand wijzigen tijdens de uitvoering van een programma. De structuur van het object (welke attributen en methodes het heeft) ligt vast. Het gedrag ligt ook vast. Om de structuur of het gedrag te wijzigen moet je het programma opnieuw compileren.

2.2 Het doel van object-oriëntering

Object-oriëntering is ontstaan uit een nood aan *onderhoudsvriendelijker* en *gemakkelijker te hergebruiken* software.

De onderhoudsvriendelijkheid van software is enorm belangrijk. De meeste softwaresystemen worden tijdens hun levensduur voortdurend aangepast. Er zijn meer informatici bezig met het onderhoud van bestaande systemen dan met het ontwikkelen van nieuwe. Onderhoudsvriendelijke software zorgt er voor dat dit efficiënt kan gebeuren.
Objecten moeten software overzichtelijker en gemakkelijker te begrijpen maken. In software die overzichtelijk is kun je snel terugvinden welke stukken aangepast moeten worden, en welke niet. Daardoor wordt onderhoud gemakkelijker.

Bij het ontwikkelen en onderhouden van software kun je ook veel tijd winnen als je bestaande code opnieuw kunt gebruiken. We vinden het bijvoorbeeld vanzelfsprekend om niet zelf een string-klasse te schrijven, maar om er een te gebruiken die al bestaat.

Objecten zijn bedacht met de bedoeling hergebruik gemakkelijker te maken: een object is een soort pakketje met daarin alles wat nodig is voor een bepaalde functionaliteit.

Zoals veel nieuwe dingen werden ook objecten in het begin heel erg gehypet. Met objecten zou programmeren kinderspel worden. Na een tijd kwam er ontnuchtering, maar het idee was zo goed dat het nu, bijna vijftig jaar later, onbetwist de meest gebruikte manier van programmeren is.

Tegenwoordig is er een trend naar hergebruik op een grotere schaal dan objecten, namelijk met frameworks en pakketten (bijvoorbeeld: een framework voor een webtoepassing, waar je via een package manager bijkomende functionaliteiten kunt aan toevoegen).

Ook functioneel programmeren is erg in opkomst. Objectgeoriënteerde programmeertalen worden vaak uitgebreid met functionele programmeermogelijkheden (bijvoorbeeld: lambda's in Java).
Het klassieke objectgeoriënteerde denken maakt een onderscheid tussen functies en gegevens. Dit onderscheid komt van het klassieke computermodel: er is een processor en een geheugen. De processor voert machine-instructies uit, en die instructies worden beschreven door de functies. Om van de functies machine-instructies te maken heb je een compiler nodig. Maar compilers zijn trage dingen. Toen objectgeoriënteerde talen bedacht werden was compileren enkel praktisch haalbaar als het vooraf gebeurde, voor dat de gebruiker de software krijgt. Gebruikers kregen een gecompileerde versie van het programma, want compileren tijdens het gebruik van het programma was niet haalbaar. Vandaar het onderscheid tussen enerzijds de data, dus de attributen van objecten, waarvan de inhoud gemakkelijk te wijzigen is, en de rest van het object, dus de structuur en de methodes, die niet wijzigbaar zijn.
Computers zijn sindsdien ongelooflijk veel sneller geworden, en men heeft ontdekt dat er een stijl van functies is (functies zonder neveneffecten) die zich goed leent tot parallelliseren. Daardoor is er nu meer mogelijk, en worden functies meer en meer zoals gegevens: iets wat je kunt wijzigen at runtime.
Veel van de patterns die in het handboek aan bod komen hebben hun oorsprong in problemen die ontstaan uit die beperking van objecten: namelijk dat hun structuur en hun methodes niet wijzigbaar zijn. Als die beperking er niet meer is kun je tot heel andere oplossingen komen. Dit valt echter buiten het bestek van dit vak.

2.3 Basisideeën van object-oriëntering

2.3.1 Modulariteit

Ieder object is als het ware een bouwsteen. Een stuk software wordt opgebouwd uit die bouwstenen, in plaats van als één massief geheel.

Goed passende bouwstenen geven een stuk software een *overzichtelijke structuur*.
Goede bouwstenen zijn *uitwisselbaar*.
Die twee zaken maken software gemakkelijker te begrijpen en te onderhouden.

Het is echter niet vanzelfsprekend een goede bouwsteen te maken! De richtlijnen in het vervolg van dit hoofdstuk proberen daarbij te helpen.

2.3.2 Encapsulering

Een goed object heeft een zichtbaar (public) deel en een verborgen (private) deel. Dat is een principe dat je niet enkel in software ziet. Een auto bijvoorbeeld heeft veel onderdelen die verborgen zijn: de motor, de remmen, de stuurinrichting … . Die zijn verborgen, omdat het zeer onpraktisch en gevaarlijk zou zijn ze rechtstreeks te bedienen. In de plaats daarvan bestuur je een auto met het stuur en de pedalen (het zichtbare deel). Bij objecten is het net zo: dingen die onpraktisch of gevaarlijk zijn om rechtstreeks te bedienen worden verborgen. Dat is *encapsulering*.

Ook encapsulering is goed voor onderhoudbaarheid. Het private deel van een object is onzichtbaar voor de rest van het programma. Je kunt dat private deel dus wijzigen zonder dat je je zorgen moet maken over de rest van het programma. Je moet enkel zorgen dat het publieke deel van het object blijft werken zoals voordien.

Het zichtbare deel van een object wordt ook wel de *interface* van het object genoemd (niet te verwarren met een interface in Java of C#, dat is iets anders).

2.3.3 Overerving

Het is mogelijk een object te nemen, en daar een nieuw object van af te leiden.

Dat wil zeggen dat het afgeleide object de functionaliteit van het andere object overneemt, of uitbreidt, of specialiseert.

Overerving is een van de manieren waarop je een object kunt hergebruiken.

2.3.4 Polymorfisme

Objecten met een gelijkaardige interface kunnen elkaars plaats innemen. Sterker nog: een object dat een functionaliteit nodig heeft van een ander object hoeft niet te weten welk concreet object dat is, zolang de functionaliteit er maar is. Dat betekent dat we als programmeur niet vooraf moeten zeggen welke concrete objecten welke taken zullen uitvoeren (‘early binding’). We kunnen die keuze uitstellen tot het allerlaatste moment, nl. de uitvoering van het programma (‘late binding’).

Ook dit is een principe dat in het dagelijks leven veel voorkomt. Als je een klaslokaal binnenkomt, moet je niet vooraf al weten op welke stoel je zult zitten (early binding). Je kunt dat kiezen op het moment zelf (late binding).

Polymorfisme is goed voor onderhoudbaarheid, omdat we met polymorfisme gemakkelijker een bepaalde taak door een nieuw object kunnen laten uitvoeren. Het volstaat dat het nieuwe object een gelijkaardige interface heeft aan het bestaande object dat we willen vervangen.

2.4 De SOLID-principes

Het letterwoord SOLID staat voor vijf principes:

Single responsibility

Open/Closed

Liskov substitution

Interface segregation

Dependency inversion

Het zijn principes waarvan de ervaring geleerd heeft dat ze helpen om goede objecten, goede softwarebouwstenen te maken. Dat wil zeggen: objecten die geschikt zijn om er onderhoudsvriendelijke software mee te maken.

Deze principes zijn richtlijnen, geen wetten. Bij het toepassen is het zoeken naar een gulden middenweg. De principes niet toepassen is niet goed, maar ze te strikt toepassen is ook niet goed. Door ervaring leer je deze principes goed doseren.

Er zijn nog andere principes dan de vijf SOLID-principes, vaak ook met mooie letterwoorden als naam, om te je te helpen ze te onthouden. (Bijvoorbeeld: DRY: 'do not repeat yourself'. M.a.w.: schrijf nooit twee keer dezelfde code. KISS: 'keep it short and simple'. Of de GRASP-principes: 'General Responsibility Assignment Software Patterns' – zie daarvoor bijvoorbeeld (Larman, 2005)). Je ontmoet er beslist nog tijdens je loopbaan.

Veel van het materiaal uit dit hoofdstuk is ontleend aan (Fowler, Clean Code: a Handbook of Agile Software Craftsmanship, 2009), een uitstekend boek over het schrijven van onderhoudsvriendelijke software. Ook de andere werken van Martin Fowler, en zijn blog (The Clean Code Blog) zijn zeer aan te bevelen.

2.4.1 Single responsibility principle

Elke klasse moet één, en niet meer dan één reden hebben om gewijzigd te worden.

Dit principe heeft veel te maken met het principe van hoge cohesie dat verder aan bod komt. Het principe van hoge cohesie zegt dat elke klasse één duidelijke, samenhangende verantwoordelijkheid moet hebben.

Als dit principe gevolgd wordt is het gemakkelijker om te begrijpen waar een klasse voor dient dan als een klasse veel vage verantwoordelijkheden heeft.
Software waarvan je gemakkelijk begrijpt waar elke klasse voor dient is veel gemakkelijker te onderhouden dan software waar klassen onduidelijke verantwoordelijkheden hebben. Je kunt immers sneller de delen terugvinden die je moet aanpassen, en er is minder kans dat de aanpassing ongewenste neveneffecten veroorzaakt.

Wat zijn aanwijzingen dat dit principe gevolgd wordt?

- klassen hebben een beperkte omvang
- de naam van de klasse maakt het doel van de klasse duidelijk
- je kunt de verantwoordelijkheid van de klasse omschrijven in een paar zinnen, liefst zonder gebruik van de woorden 'als', 'maar', 'en' en 'of'
- je kunt klassen gemakkelijk ordenen in groepen die bij elkaar horen

Bij klassen met te veel verantwoordelijkheden, of onduidelijke verantwoordelijkheden is het vaak zo dat

- de klasse erg groot is
- je zonder documentatie niet kunt begrijpen waar de klasse voor dient
- de klasse maar voor 1 specifieke toepassing bruikbaar is

Je kunt ook in de andere richting te ver gaan. Dan krijg je software met heel veel heel kleine klassen, waar je het overzicht in verliest.

Voorbeeld

```
public class SuperDashboard extends JFrame implements MetaDataUser {
  public Component getLastFocusedComponent();
  public void setLastFocused(Component lastFocused);
  public int getMajorVersionNumber();
  public int getMinorVersionNumber();
}
```

Deze klasse heeft twee redenen om te veranderen. Ten eerste doet ze iets met Java Swing componenten (ze is afgeleid van JFrame, een klasse die een venster voorstelt). Ten tweede doet ze iets met versie-informatie. Wanneer de componenten in het venster veranderen zal allicht ook het versienummer veranderen, maar andersom is het perfect denkbaar dat het versienummer verandert zonder dat de componenten in het venster veranderen.

Het zou dus beter zijn om de verantwoordelijkheid voor de versie-informatie af te splitsen in een aparte klasse.

2.4.2 Open/Closed principle

Een goede klasse is ‘open for extension’: ze kan uitgebreid worden. Een goede klasse is ook ‘closed for modification’: haar gedrag kan niet gewijzigd worden.

Op een iets andere manier gezegd: we moeten software zo ontwerpen dat ze gemakkelijk aangepast kan worden door nieuwe code toe te voegen, zonder de bestaande code te veranderen.

Waarom mag het gedrag van een klasse niet wijzigbaar zijn? Een klasse wordt dikwijls op verschillende plaatsen gebruikt. Het zou kunnen dat als je het gedrag van een klasse wijzigt omdat

dat voordelig is voor een bepaalde situatie, dat die wijziging er voor zorgt dat de klasse niet meer goed werkt in de andere situaties!
Wanneer je een klasse aanpast, dan moet je alle software hertesten die die aangepaste klasse gebruikt. Dat kan veel meer zijn dan alleen het programma of het deel van het programma waar je op dat moment mee bezig bent.
Een voorbeeld: in het bedrijf waar je werkt is er een klasse Product met een methode getPrice(). De returnwaarde van die methode is de prijs zonder BTW. Jij hebt voor het werk waar je mee bezig bent de prijs met BTW nodig. Je besluit getPrice() aan te passen zodat die methode voortaan de prijs met BTW retourneert. Je verandert dus het gedrag van de klasse Product. Op slag werken alle andere programma's die de klasse Product gebruiken niet meer correct.

Wat moet je dan wel doen als je een bestaande klasse wilt aanpassen aan je eigen noden? Je kunt ze uitbreiden, bijvoorbeeld door er een nieuwe klasse van af te leiden en in die nieuwe klasse de nodige aanpassingen te programmeren. In het geval van het voorbeeld: door een methode getPriceWithVAT() toe te voegen. De bestaande klasse blijft dan onveranderd, en alle programma's die die bestaande klasse gebruiken merken geen verandering en blijven dus werken.
Als je een klasse ontwerpt kun je ook voorzien dat bepaalde stukken aangepast zullen moeten worden. Door het gebruik van design patterns kun je aanpasbaarheid op een goede manier mogelijk maken. Het is wel niet gemakkelijk om vooraf te weten waar er aanpassing nodig zal zijn…

Hoe zie je dat dit principe toegepast wordt?

- functionaliteit wordt toegevoegd met nieuwe klassen, niet door het veranderen van bestaande klassen (dat zie je duidelijk in verschillende design patterns uit het handboek).
- er wordt overerving gebruikt
- er worden interfaces (in de Java-zin) gebruikt

2.4.3 Liskov substitution principle

Het principe met de moeilijkste naam, dat echter het meest vanzelfsprekend lijkt.

Een object moet je altijd kunnen vervangen door een meer gespecialiseerd object, zonder dat dat invloed heeft op bestaande gebruikers van dat object.

Objectgeoriënteerde talen als Java of C# maken het gemakkelijk om dit principe te respecteren. Als klasse A afgeleid is van klasse B, dan kun je objecten van A overal gebruiken waar objecten van B bruikbaar zijn.

Maar zelfs in objectgeoriënteerde talen is het mogelijk om dit principe te schenden. Een klasse biedt een aantal diensten aan zijn gebruikers aan, heeft als het ware een contract met zijn gebruikers om bepaalde dingen te doen. Als een afgeleide klasse het contract verbreekt, dan is het Liskov-principe geschonden.
Bijvoorbeeld: in java heeft klasse String een methode length(). Het ‘contract’ van klasse String met zijn gebruikers is dat length() het aantal tekens in de string geeft. Je maakt nu een klasse MyString, afgeleid van String, met een override van length() die altijd -1 teruggeeft. MyString houdt zich dus

niet aan het 'contract' van String, ook al kun je in principe in een programma een MyString gebruiken overal waar er een String staat. MyString schendt het Liskov subsitutieprincipe.

Hoe draagt dit principe bij aan onderhoudsvriendelijkheid? Als je dit principe respecteert, dan kun je een bestaande klasse vervangen door een afgeleide klasse, zonder dat je je zorgen moet maken dat die vervanging bestaande code zal breken.

2.4.4 Interface segregation principle

Een object mag niet afhankelijk zijn van methodes die het niet gebruikt.

Anders geformuleerd: een object mag enkel de methodes hebben die het echt nodig heeft.

Als je dit principe niet volgt, dan krijg je objecten met erg veel methodes. Die moeten allemaal correct werken, ook al gebruik je ze niet, want het zou kunnen dat jij of iemand anders die ooit toch eens zal gebruiken. Dergelijke objecten zijn dus lastiger te onderhouden dan objecten die het principe wel volgen.

Een manier om dit principe toe te passen die je dikwijls ziet zijn objecten die veel kleine interfaces (interfaces met weinig methodes) implementeren, eerder dan een paar heel grote. Door kleine interfaces te gebruiken heb je meer controle over welke methodes een object al of niet heeft. Bijvoorbeeld in de Java-klassenbibliotheek zie je dit dikwijls toegepast worden.

2.4.5 Dependency inversion principle

High-level modules mogen niet afhankelijk zijn van low-level modules, maar moeten afhankelijk zijn van abstracties. Abstracties mogen niet van details afhankelijk zijn. Details moeten afhankelijk zijn van abstracties.

Een wat extreem voorbeeld om dit principe te verduidelijken: met Word kun je documenten opstellen en die opslaan. Om een document lokaal op een harde schijf op te slaan moeten er een heleboel bits op een bepaalde plaats op de schijf gezet worden.
Word is een high-level module. Het systeem dat de harde schijf stuurt is een low-level module. Voor elk model harde schijf is het een beetje anders.
Het zou helemaal niet praktisch zijn als je een andere versie van Word nodig zou hebben voor elk model harde schijf dat er is. Daarom gebruikt Word een abstractie: het bestandssysteem. Het bestandssysteem is een abstractie van de bits die op een harde schijf staan. Op die manier heb je niet meer een versie van Word per model harde schijf, maar enkel nog een versie per soort bestandssysteem (dus per operating system).
Het bestandssysteem is een abstractie. Het zou niet praktisch zijn als het bestandssysteem (zoals gezien door de gebruikers van dat bestandssysteem) zou afhangen van het type harde schijf dat in een computer zit – het model harde schijf is een detail. Dat zou willen zeggen dat als je de harde schijf van een computer vervangt, je ook een nieuw bestandssysteem zou moeten installeren.

Alle software van enige omvang heeft high-level modules, zoals de user interface of de controller, en low-level modules, zoals data access objecten, of timers, of sockets. Als de high-level modules afhankelijk zijn van die low-level modules dan wil dat zeggen dat je die low-level modules niet zomaar kunt veranderen of vervangen. Dat maakt de software moeilijker te onderhouden, want een verandering in een low-level module (een detail) kan dan de werking van de high-level modules (het belangrijke) verstoren.

Dit principe is verwant met het principe van lage koppeling uit het volgende hoofdstuk. Een high-level module die van low-level modules afhankelijk is heeft een hoge koppeling.

2.5 Andere ontwerpprincipes

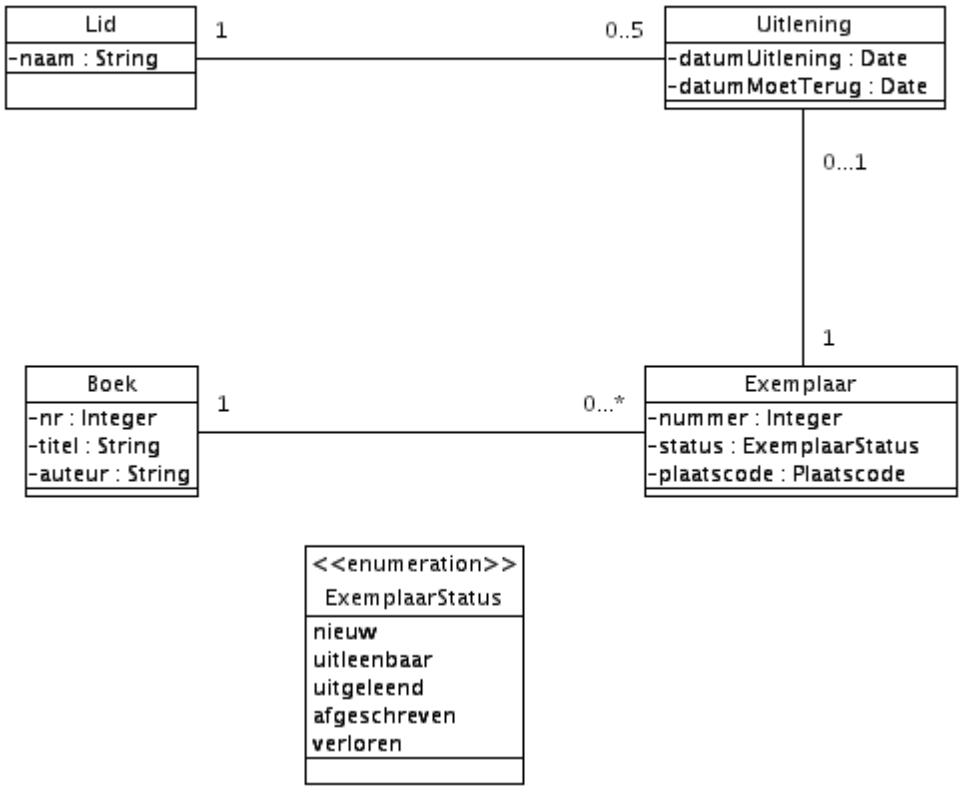
In dit hoofdstuk behandelen we enkele basisprincipes die ons kunnen leiden bij het detailontwerp: Information Expert, Creator, Lage Koppeling en Hoge Cohesie. Er is overlap tussen deze principes en de SOLID-principes. Dit hoofdstuk is dus voor een deel een variatie en herhaling van het vorige.

Om de functionele eisen van een programma te realiseren en om de business events uit te voeren, moeten we objecten met elkaar laten samenwerken. Voortdurend worden we dan geconfronteerd met de vraag:

Aan welk object kennen we de verantwoordelijkheid voor een bepaalde actie toe?

In dit hoofdstuk zullen we een aantal basisprincipes bespreken die ons bij deze keuze kunnen leiden.

Als voorbeeldsituatie nemen we de casus van de eenvoudige bibliotheek uit de cursus Analysis & Design 1. Figuur 8 is het domeinmodel, en figuur 9 de Object Event Table.



Figuur 8: domeinmodel eenvoudige bibliotheek

	Lid	Uitlening	Boek	Exemplaar
InschrijvenLid	C			
UitschrijvenLid	E			
Uitlenen	M	C	M	M
Terugbrengen	M	E	M	M
Verlengen	M	M	M	M
Verliezen	M	E	M	E
AankopenEx			M	C
ToekennenPlaatscodeEx			M	M
AfschrijvenEx			M	E
ToevoegenBoek			C	
VerwijderenBoek			E	

Figuur 9: object event table eenvoudige bibliotheek

2.5.1 Information Expert

Laten we als uitgangspunt het business event *uitlenen* nemen. De gedetailleerde specificatie van dit event ziet er als volgt uit:

```
Uitlening::uitlenen(l: Lid, e: Exemplaar, vandaag: Datum)
pre:
      l heeft niet meer dan 5 uitleningen
      e.status = uitleenbaar
post:
      new u:Uitlening
      u.datumUitlening = vandaag
      u.datumMoetTerug = vandaag + 3 weken
      u.lid = l
      u.exemplaar = e

Lid::uitlenen(u: Uitlening)
pre:
      self heeft niet meer dan 5 uitleningen
post:
      het aantal uitleningen van self verhoogt met 1

Exemplaar::uitlenen(u: Uitlening)
pre:
      self.status = uitleenbaar
post:
      self.status = uitgeleend
```

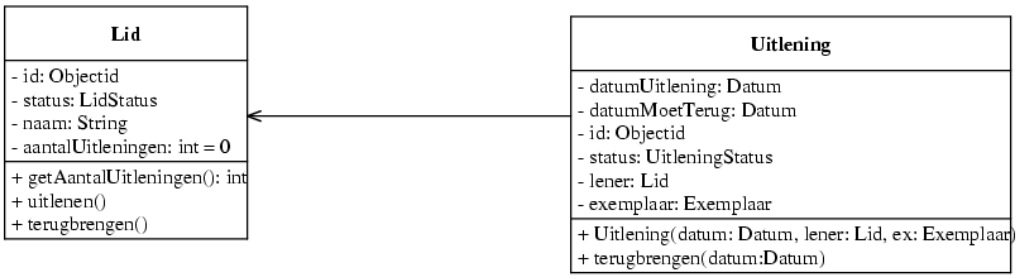
We stellen vast dat het object dat dit event moet uitvoeren eerst een aantal precondities moet testen en bijgevolg over bepaalde informatie moet beschikken. We moeten bijvoorbeeld weten hoeveel uitleningen een bepaald lid al heeft. De vraag is nu:

Wie moet er verantwoordelijk zijn voor het verschaffen van de informatie hoeveel uitleningen een bepaald lid heeft?

Het **principe van de information expert** zegt daarover:

Geef de verantwoordelijkheid aan dit object dat natuurlijkerwijze over de informatie zal beschikken.

We denken hierbij spontaan aan de domeinklasse Lid. De domeinklasse Lid heeft een associatie met Uitlening. We kunnen de verantwoordelijkheid voor het kennen van het aantal uitleningen van een Lid toekennen aan het lid zelf.



Figuur 10: ontwerpklassendiagram voor Uitlening en Lid

Dit heeft echter een aantal consequenties. Een object van de klasse Lid moet dan zijn uitleningen kennen. We moeten ten eerste een operatie

`getAantalUitleningen(): int`

toevoegen aan de klasse Lid.

Ten tweede moeten we ervoor zorgen dat Lid de mogelijkheid heeft om het aantal uitleningen te kennen. Dit kan op twee manieren opgelost worden:

- Ofwel is de navigeerbaarheid van de relatie tussen Lid en Uitlening in de richting van Lid (een uitlening kent zijn Lid) en geven we Uitlening de verantwoordelijkheid om dit attribuut op te hogen bij creatie en te verminderen bij het terugbrengen van een uitlening.
  Die situatie wordt getoond in figuur 10.
- Ofwel is de navigeerbaarheid van de relatie tussen Lid en Uitlening in de richting van Uitlening (een Lid kent zijn uitleningen) en kan Lid altijd nagaan hoeveel uitleningen het heeft.

Een deel van de code voor de klasse Lid vind je hieronder.

```
public class Lid {
    \\attributen
    private int nummmer;
    private String naam;
    private String voornaam;

    \\associaties met andere klassen
    private UitleningList uitleningen \\met Uitlening

    \\constructoren
    ...

    \\inspectiemethoden
    ...
    public int getAantalUitleningen()
    {
        return uitleningen.size();
    }
} // end class Lid
```

2.5.2 Creator

In deze paragraaf concentreren we ons op het volgende probleem:

Wie moet er verantwoordelijk zijn voor het creëren van een nieuwe instantie van een bepaalde klasse?

Wijs de verantwoordelijkheid voor het creëren van een nieuwe instantie van A toe aan de klasse B als één van de volgende beweringen waar is:

- B is een verzameling van objecten A.
- B bevat objecten van A.
- B registreert instanties van A.
- B maakt nauw gebruikt van objecten A.
- B beschikt over de initialisatiegegevens die nodig zijn bij de creatie van A.

Toegepast op het voorbeeld: welke klasse moet verantwoordelijk zijn voor het creëren van een nieuwe instantie van Uitlening? In ons geval zou Lid bijvoorbeeld de uitstekende kandidaat zijn. Veronderstellen we dat de navigeerbaarheid van de associatie tussen Lid en Uitlening in de richting van Uitlening gaat. In dat geval bevat Lid een lijst van uitleningen en maakt daar ook nauw gebruik van. Lid is bijgevolg de meest voor de hand liggende kandidaat om een Uitlening te creëren.

Het creëren van een uitlening vindt plaats bij het uitvoeren van het event ‘Uitlenen’. Lid is, zoals we eerder gezien hebben, ook de informatie-expert wanneer het gaat over de preconditie “Heeft dit lid nog minder dan vijf uitleningen?”. Dit alles impliceert dus dat we de uitvoering van het business event ‘Uitlenen’ zouden kunnen toekennen aan de klasse Lid.

2.5.3 Lage koppeling (low coupling)

Het voorbeeld uit de vorige paragraaf, waarbij Lid de uitvoerder van het event ‘Uitlenen’ wordt, is strijdig met een ander principe, het principe van de lage koppeling.

Het principe van lage koppeling moet een antwoord geven op de vraag:

Hoe zorgen we ervoor dat de afhankelijkheid tussen objecten laag blijft, dat de impact van veranderingen minimaal is en dat hergebruik van klassen mogelijk wordt?

Het antwoord op die vraag luidt:

Wijs de verantwoordelijkheden zo toe dat er een lage koppeling is tussen klassen.

Koppeling wil zeggen: afhankelijkheid. Klasse A is gekoppeld aan klasse B als klasse A klasse B nodig heeft om zijn werk te doen.
Een gevolg van die koppeling is dat als we klasse B veranderen, dat we op zijn minst klasse A moeten hertesten, en waarschijnlijk klasse A ook moeten aanpassen.
Veel koppeling wil dus zeggen dat een wijziging aan een deel van het programma veel impact zal hebben op de rest van het programma, en dat is slecht voor onderhoudbaarheid.

Veel koppeling wil ook zeggen dat om een klasse te kunnen hergebruiken, dat we alle afhankelijkheden mee moeten nemen. Hoge koppeling is dus slecht voor hergebruik.

Een element met lage koppeling is niet afhankelijk van te veel andere klassen. Een element met een hoge koppeling is afhankelijk van heel wat andere klassen. Dit heeft een aantal nadelen:

- Veranderingen in die andere klassen leiden tot veranderingen in de betrokken klasse.
- De klasse is moeilijk te begrijpen.
- De klasse kan moeilijker elders opnieuw gebruikt worden omdat ook de aanwezigheid van die andere klassen verondersteld wordt.

In dat opzicht zou het verstandiger zijn om de uitvoering van het event toe te kennen aan een derde klasse, een zogenaamde controllerklasse. Een controllerklasse is een klasse die instaat voor de uitvoering van een bepaalde gebeurtenis. De controllerklasse kent alle domeinklassen die bij die gebeurtenis betrokken zijn, weet welke precondities er getest moeten worden en weet welke operaties er bij de betrokken klassen uitgevoerd moeten worden om de gebeurtenis uit te voeren.

Uitgaande van de klassen en operaties in figuur 11 zou de uitvoering van de operatie execute() bij de klasse UitlenenController als volgt verlopen:

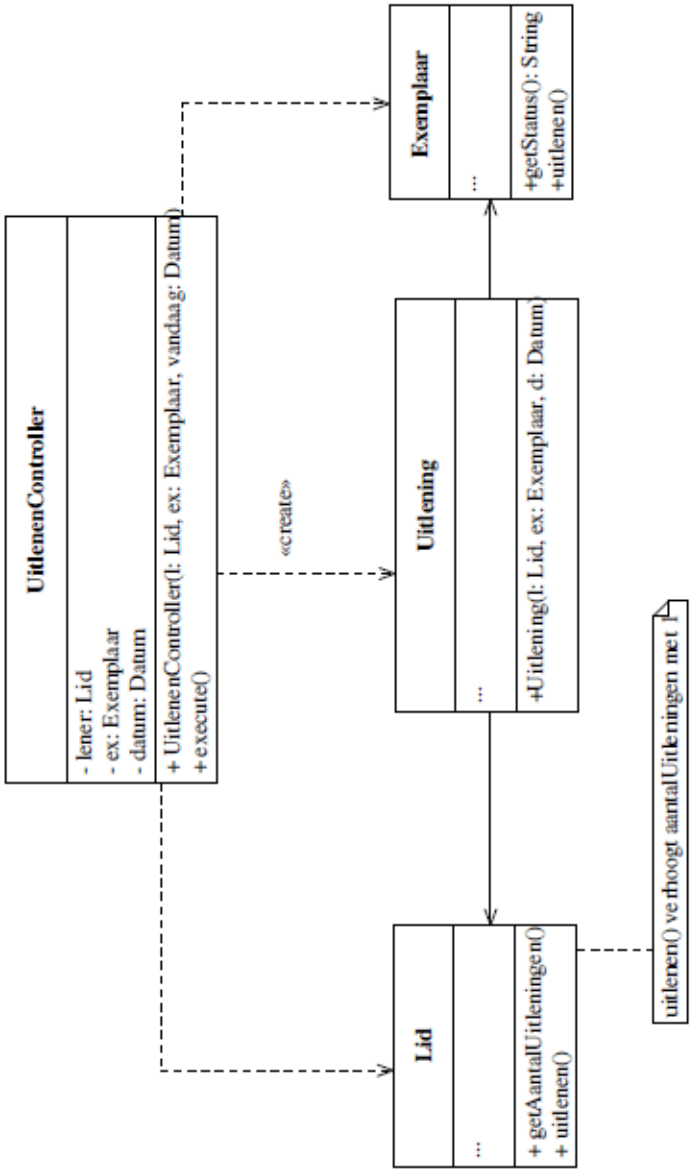
Eerst worden de precondities getest:

- Heeft het lid nog geen vijf uitleningen?
- Is het exemplaar uitleenbaar, dat wil zeggen het bevindt zich in de rekken, het is niet uitgeleend?

Wanneer aan één van de precondities niet voldaan wordt, wordt een passende exceptie geactiveerd.

Daarna wordt het exemplaar uitgeleend:

- Er wordt een uitlening voor dit lid en dit exemplaar gecreëerd.
- De status van het exemplaar wordt op uitgeleend gezet. Dit gebeurt met de operatie Uitlenen().



Figuur 11: ontwerpschema met controllerklasse

2.5.4 Hoge cohesie (high cohesion)

Hoge cohesie gedraagt zich ten opzichte van lage koppeling als yin en yang. Wanneer er een hoge cohesie is, leidt dit meestal tot een lage koppeling en omgekeerd.

Hoge cohesie biedt een antwoord op de vraag:

Hoe kunnen we complexiteit beheersbaar houden? Hoe kunnen we ervoor zorgen dat een complexe applicatie begrijpelijk en goed onderhoudbaar wordt?

Het antwoord luidt:

Zorg ervoor dat klassen een hoge cohesie vertonen, dit wil zeggen: zorg ervoor dat een klasse een beperkt aantal duidelijk samenhangende verantwoordelijkheden heeft.

Klassen met een lage cohesie zijn moeilijk te begrijpen en moeilijk te hergebruiken.

Het principe van hoge cohesie leidt ons opnieuw naar een oplossing waarbij het event 'Uitlenen' toegewezen wordt aan een aparte controllerklasse. Op die manier zullen de klassen Lid en Uitlening een beperkte verantwoordelijkheid krijgen: zij representeren één domeinobject en onderhouden de attributen van dit domeinobject. Zij hebben een lage koppeling met de andere domeinklassen en hoeven zich niet bezig te houden met de uitvoering van events.

De controllerklasse UitlenenController heeft natuurlijk wel een koppeling met de domeinklassen Exemplaar, Lid en Uitlening, maar heeft ook een duidelijk afgelijnde verantwoordelijkheid (een hoge cohesie dus). De klasse staat enkel in voor het uitvoeren van het event Uitlenen, d.w.z. het testen van de nodige precondities en het sturen van de juiste boodschappen naar de betrokken domeinobjecten.

Het principe van hoge cohesie is ook een sterk argument om het bewaren (of ophalen) van een domeinobject in de database niet aan de domeinklasse over te laten. Het bewaren van een object in de database veronderstelt immers heel wat specifieke acties: het leggen van een connectie, het omzetten van de objectattributen in tabelkolommen en het uitvoeren van SQL-code. We zullen deze verantwoordelijk aan aparte klassen toewijzen. We denken bijvoorbeeld aan een klasse 'LidDb' met operaties als:

```
bewaarLid(l: Lid)
haalOpLid(nummer: int): Lid
```

3 Design patterns van de gang of four

Deze patterns worden behandeld in het handboek (Java Design Pattern Essentials).

- Simple Factory
- Factory Method
- Abstract Factory
- Builder
- Adapter
- Composite
- Decorator
- Facade
- Flyweight
- Chain of Responsibility
- Command
- Iterator
- Memento
- Observer
- Strategy
- Template Method
- Null Object

Bibliografie

Bevis, T. (2012). *Java Design Pattern Essentials - Second Edition.* Ability First.

Fowler, M. (2003). *Patterns of Enterprise Application Architecture.* Addison-Wesley.

Fowler, M. (2009). *Clean Code: a Handbook of Agile Software Craftsmanship.* Prentice Hall.

Fowler, M. (sd). *The Clean Code Blog*. Opgehaald van http://blog.cleancoder.com

Freeman, E., & Freeman, E. (2007). *Train je hersens in Design Patterns.* LannooCampus.

Larman, C. (2005). *Applying UML and Patterns.* Pearson.

Warmer, J., & Kleppe, A. (2011). *Praktisch UML.* Pearson Education Benelux.