Objectgeoriënteerd ontwerpen

Introductie *17*

Leerkern *18*

1 Objectgeoriënteerd ontwerpen *18*
- 1.1 Softwareontwikkeling *18*
- 1.2 Wat is een goed programma? *24*
- 1.3 Objectkeuze *28*

2 UML-diagrammen *29*
- 2.1 Klassendiagrammen *29*
- 2.2 Objectdiagrammen *35*

3 Een eenvoudige simulatie van een bank *36*
- 3.1 Productomschrijving en gebruiksmogelijkheden *36*
- 3.2 Opstellen domeinmodel *37*
- 3.3 Opstellen ontwerpmodel *40*

4 Ontwerpen stap voor stap *47*

Samenvatting *50*

Zelftoets *51*

Terugkoppeling *52*

1 Uitwerking van de opgaven *52*

2 Uitwerking van de zelftoets *56*

Leereenheid 1

Objectgeoriënteerd ontwerpen

INTRODUCTIE

In de cursus Objectgeoriënteerd programmeren in Java 1 hebt u kleine, objectgeoriënteerde programma's leren schrijven. U werd daarbij aangemoedigd om bepaalde regels voor goed programmeren in acht te nemen.
U heeft bijvoorbeeld geleerd om de programmacode in de gebruikersinterface zo beperkt mogelijk te houden en al het echte werk over te laten aan instanties van andere klassen. In alle voorbeelden beperkte de taak van de gebruikersinterface zich daardoor tot het verzorgen van de interactie met de gebruiker van de applicatie. Ook hebben we u aangemoedigd om duidelijke namen te kiezen voor alles wat een naam moet krijgen: componenten uit de gebruikersinterface, klassen, attributen, methoden, lokale variabelen en parameters. En we wilden graag dat u de klassen als geheel en iedere methode afzonderlijk van commentaar voorzag.

In Objectgeoriënteerd programmeren in Java 1 is echter niet zo veel aandacht besteed aan de vraag waarom we dat willen. Wat is er tegen een applicatie waarin alles in de interfaceklasse (het frame) zelf gedaan wordt, waarin componenten button1, textfield1 en textfield2 heten, waarin alle andere attributen, variabelen en methoden zo kort mogelijke namen krijgen (a, b, c) en waarin geen regel commentaar staat? Met andere woorden: hoe ziet een goed programma er eigenlijk uit?

In paragraaf 1 van deze leereenheid gaan we deze vraag onderzoeken en er een gedeeltelijk antwoord op geven. We kijken nadrukkelijk niet alleen naar programma's die één persoon in uren of dagen kan ontwikkelen. We zoeken ook naar criteria voor programma's waar verschillende programmeurs aan werken en waarvan de ontwikkeling weken, maanden of zelfs jaren kost.

We houden ons in deze cursus, veel meer dan in Objectgeoriënteerd programmeren in Java 1, bezig met het ontwerp van objectgeoriënteerde programma's. We maken daarbij gebruik van diagramtechnieken die ook al in de cursus Objectgeoriënteerd programmeren in Java 1 aan de orde zijn geweest. Met behulp van klassendiagrammen wordt de structuur van een ontwerp beschreven: de klassen en de samenhang daartussen. Met objectdiagrammen kan de toestand van objecten op een bepaald moment tijdens het verwerken van een programma worden beschreven. Beide diagramtechnieken zijn afkomstig uit de unified modeling language (UML), een diagramtaal die speciaal is ontwikkeld voor objectgeoriënteerd ontwerpen.

In Objectgeoriënteerd programmeren in Java 1 werd, om de notatie zo dicht mogelijk aan te laten sluiten bij Java, afgeweken van de officiële UML-notatie. In deze cursus houden we ons echter wel aan die officiële notatie,die we in paragraaf 2 introduceren.

In paragraaf 3 gebruiken we deze diagramtechnieken en de criteria die in paragraaf 1 zijn opgesteld, in een ontwerp voor een eenvoudige simulatie van bankverkeer.
Aan het eind van deze leereenheid, in paragraaf 4, presenteren we een stappenplan, dat u kunt volgen wanneer u zelf een dergelijk ontwerp op moet stellen.

LEERDOELEN

Na het bestuderen van deze leereenheid wordt verwacht dat u
– kunt aangeven wat bedoeld wordt met softwareontwikkeling in het klein en met softwareontwikkeling in het groot
– de verschillende fasen in de ontwikkeling van een programma kunt aangeven
– weet wat bedoeld wordt met het technisch ontwerp van een programma
– vier doelstellingen kunt noemen die worden nagestreefd bij het maken van een technisch ontwerp
– drie concrete eigenschappen kunt noemen van een objectgeoriënteerd programma die bijdragen tot het verwezenlijken van die doelstellingen
– een eenvoudig klassendiagram kunt lezen en opstellen
– een objectdiagram in UML kunt lezen en tekenen
– weet welk stappenplan u kunt volgen bij het opstellen van een objectgeoriënteerd ontwerp
– de betekenis kent van de volgende kernbegrippen: voortraject, analyse, gebruiksmogelijkheid, domeinmodel, ontwerp, ontwerpmodel, implementatie, evaluatie, afronding, correctheid, robuustheid, gescheiden verantwoordelijkheden, lokaliteit, koppeling, generalisatie, associatie, rol, multipliciteit, link.

Studeeraanwijzingen
De studielast van deze leereenheid bedraagt circa 5 uur.

LEERKERN

1 Objectgeoriënteerd ontwerpen

1.1 SOFTWAREONTWIKKELING

Software-ontwikkeling in het klein

De cursussen Objectgeoriënteerd programmeren in Java 1 en Objectgeoriënteerd programmeren in Java 2 hebben als doel u te leren *software te ontwikkelen in het klein*. Dat betekent dat deze twee cursussen samen u voldoende basis moeten geven om, eventueel na wat extra oefening, zelfstandig kleine programma’s te kunnen ontwikkelen.
Of een programma klein of groot is, meten we niet af aan het aantal regels code in het eindproduct. Bepalend voor de ontwikkeling van een klein programma zijn de volgende kenmerken:
– Het wordt ontwikkeld door één programmeur.

– Het kan worden ontwikkeld in een beperkte tijd, die eerder gemeten zal worden in uren of dagen dan in maanden of jaren.
– Er is meteen al een duidelijke productomschrijving waaruit nauwkeurig valt op te maken wat het programma moet doen.
– Er is geen gebruikersgroep voor wie het programma per se onderhouden moet worden.

Software-ontwikkeling in het groot

Als de ontwikkeling van een programma geen van deze kenmerken heeft, is er duidelijk sprake van *softwareontwikkeling in het groot*. Een ontwikkeltraject dat aan slechts enkele van de eisen voldoet, valt in een overgangsgebied.

Voorbeeld: containerverhuur

In de rest van deze paragraaf gebruiken we als voorbeeld van een groot programma regelmatig een informatiesysteem voor een bedrijf dat verschillende typen containers verhuurt. De huurder neemt de containers op de plaats van vertrek in ontvangst en levert ze weer in op de plaats van bestemming; de verhuurder zorgt ervoor dat ze weer bij een volgende huurder terechtkomen. Het bedrijf heeft twintig kantoren in verschillende landen. Het informatiesysteem moet alle containers gaan volgen en de kantoren in staat stellen om snel te zien of aan een verzoek van een huurder voldaan kan worden en welke containers daar dan het best voor gebruikt kunnen worden. Uiteindelijk moet dat tot een efficiënter gebruik van containers leiden.

Deze cursus gaat niet over softwareontwikkeling in het groot; daar zijn andere cursussen voor. Wat we in deze cursus echter wel willen, is u een programmeerstijl bijbrengen die ook bruikbaar is voor het programmeren in het groot. In een cursus over objectgeoriënteerd programmeren ligt dat ook voor de hand, omdat deze programmeerstijl bij uitstek is ontwikkeld met het oog op grote programma's. In dat licht moeten de eisen worden gezien die we in deze cursus aan programma's zullen stellen. We zullen daarom in deze paragraaf over de grens van de cursus kijken, naar softwareontwikkeling in het algemeen.

OPGAVE 1.1

Het is niet helemaal duidelijk wanneer een programma door één programmeur is geschreven. In de cursus Objectgeoriënteerd programmeren in Java 1 bijvoorbeeld, hebt u verschillende applicaties ontwikkeld. In zekere zin was u niet de enige programmeur van deze applicaties. U gebruikte immers klassen die door andere programmeurs waren ontwikkeld.

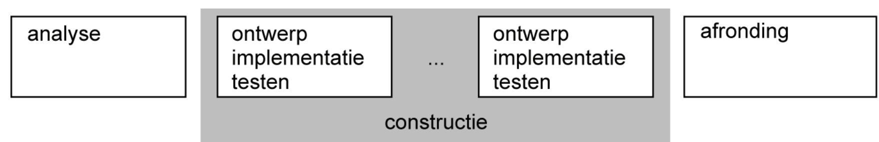
a Welke klassen waren dat zoal?
b Wat moest u van deze klassen weten om ze te kunnen gebruiken? Waar haalde u die informatie vandaan? Was deze altijd voldoende?

Hoe verloopt nu globaal de ontwikkeling van een groot softwaresysteem?

Voortraject

Voordat de ontwikkeling start, is er meestal al een heel traject afgelegd, het *voortraject*. Dit traject start bijvoorbeeld met het signaleren van een probleem of het ontstaan van een nieuwe markt.
Kijk als voorbeeld naar het systeem voor het containerverhuurbedrijf. Het voortraject kan daar begonnen zijn met de constatering dat de bedrijfsresultaten achteruit gingen. Toen onderzocht werd hoe dat kwam, bleken de tarieven aan de hoge kant. Een analyse van het bedrijfsproces leerde, dat er kosten bespaard konden worden door de containers efficiënter te

gebruiken. Het kwam te vaak voor dat er bijvoorbeeld een stel containers leeg van Rotterdam naar Liverpool werd gebracht, terwijl er in Birmingham vergelijkbare containers op een verhuurder stonden te wachten. Die vielen dan weer net onder een ander kantoor, zodat hun beschikbaarheid niet duidelijk werd. Eén informatiesysteem waarop alle kantoren zijn aangesloten, wordt als een mogelijke oplossing gezien. Als het management voldoende in het idee ziet om er geld in te steken, gaat het project pas echt van start.
Tijdens het voortraject wordt meestal ook het echte ontwikkeltraject opgezet: er wordt bijvoorbeeld bepaald wat het mag kosten en hoe lang het mag duren.
Figuur 1.1 toont een overzicht van een typische fasering vanaf dat moment.



FIGUUR 1.1 Typische fasering van een project waarin software wordt ontwikkeld

We gaan nu uitgebreider op de verschillende fasen in.

Analyse

In de eerste fase, de *analyse*, moet worden vastgesteld wat er nu eigenlijk precies ontwikkeld moet worden en aan welke eisen het product moet voldoen; pas daarna wordt besloten of het project verder gaat. Een precieze specificatie is nodig om vast te kunnen stellen of het systeem wel met bestaande methoden en technieken gerealiseerd kan worden en of dat kan binnen de randvoorwaarden die in het voortraject zijn bepaald.

Gebruiks-mogelijkheid

Engels: use case

Ontwikkelaars, opdrachtgevers en toekomstige gebruikers moeten dus rond de tafel gaan zitten om die specificatie op te stellen. Een manier om dat te doen, is het opsporen van alle gewenste *gebruiksmogelijkheden* (*use cases*) van het systeem. Iedere gebruiksmogelijkheid is een soort scenario waarin een typische interactie tussen een mogelijke gebruiker en het systeem wordt weergegeven.
Twee gebruiksmogelijkheden van het containersysteem zijn bijvoorbeeld de volgende:
– Het systeem verstrekt op verzoek van een medewerker, wiens taak het is om containers toe te wijzen, een overzicht van alle ongebruikte containers (aantal, type en locatie) op een gegeven datum in een gegeven gebied (een land, een groep landen of de hele wereld).
– Een medewerker voert een mogelijke order in (aantal containers van een bepaald type, plaats en datum van vertrek, plaats en datum van bestemming) en het systeem wijst op grond daarvan een beschikbare verzameling containers aan waarmee zo goedkoop mogelijk (volgens een gegeven kostenfunctie) aan de order kan worden voldaan.

Het is een kwestie van onderhandelen en afwegen welke gebruiksmogelijkheden in de uiteindelijke specificatie terechtkomen. In het voorbeeld kan het realiseren van de tweede gebruiksmogelijkheid zoveel kosten, dat het bedrijf besluit de uiteindelijke toewijzing toch met de hand te blijven doen en het systeem vooral te gebruiken om alle benodigde informatie centraal bij te houden en op een overzichtelijke wijze aan alle kantoren te leveren.

Een uitputtende lijst van gebruiksmogelijkheden vormt de basis van verdere gesprekken tussen ontwikkelaar en opdrachtgever.

Domeinmodel

Tijdens analyse (vaak tegelijkertijd met het opstellen van de lijst van gebruiksmogelijkheden) wordt ook een *domeinmodel* opgesteld. In een dergelijk model wordt het deel van de werkelijkheid beschreven waar het systeem betrekking op heeft. Bij een objectgeoriënteerde analyse wordt die beschrijving gegeven in termen van klassen, hun relaties en hun interacties. Oppervlakkig lijkt het domeinmodel daarom op een ontwerp voor een objectgeoriënteerd programma, maar dat is het niet! Bij het opstellen van het domeinmodel wordt nog helemaal geen rekening gehouden met de functionaliteit die de software moet gaan bieden. Een domeinmodel is bovendien vaak niet volledig: allerlei details worden weggelaten om het geheel begrijpelijk en hanteerbaar te houden. Ook de essentiële processen uit het domein worden tijdens de analyse beschreven, bijvoorbeeld: Hoe wordt een container gevolgd? Hoe wordt een order van een klant verwerkt?

Constructie

Als het project niet na de analysefase is afgeblazen, kan nu de feitelijke *constructie* van het systeem beginnen. Dat gebeurt vrijwel altijd in fasen. De lijst van gebruiksmogelijkheden kan hierbij als uitgangspunt dienen: in iedere fase wordt een deel van de gebruiksmogelijkheden gerealiseerd. Aan het eind van iedere fase is er dus een werkend systeem, dat echter nog niet alles doet wat het volgens de specificaties zou moeten doen. Iedere fase in de constructie bestaat zelf weer uit drie stappen: ontwerp, implementatie en testen.

Ontwerp

Tijdens het *ontwerp* van een objectgeoriënteerd systeem wordt vastgesteld uit welke klassen het programma zal gaan bestaan en hoe die klassen met elkaar samenhangen. Aan het eind van de ontwerpfase moet het volledig duidelijk zijn *welke* verantwoordelijkheden elke klasse in het programma zal hebben.
Wat dan nog niet vastligt, is *hoe* de klasse die verantwoordelijkheden zal realiseren. De resulterende klassenstructuur zullen we het *ontwerpmodel* noemen. Vaak zal het domeinmodel als uitgangspunt dienen voor het opstellen hiervan, maar het ontwerpmodel kan ook aanzienlijk afwijken van het domeinmodel. Er kunnen extra klassen aan worden toegevoegd die niet overeenkomen met elementen uit het domein, maar die louter een beheerstaak hebben. Ook worden er soms andere klassen gekozen, omdat de klassen uit het domeinmodel tot een onhandige of een inefficiënte implementatie zouden leiden.

Ontwerpmodel

In deze cursus verdiepen we ons niet in zulke gevallen, maar het is goed om u te realiseren dat bij programmeren in het groot het modelleren van het domein en het ontwerpen van de software gescheiden activiteiten zijn die tot verschillende resultaten kunnen leiden. Voor het softwareontwerp wordt daarom ook wel de term *technisch ontwerp* gebruikt.

Technisch ontwerp

Verder moet u zich realiseren dat een ontwerp vaak een uitbreiding is van een vorig ontwerp. De constructie verloopt immers in fasen; in elke fase worden gebruiksmogelijkheden toegevoegd. Soms kan daarvoor worden volstaan met het uitbreiden van bestaande klassen, soms moeten nieuwe klassen worden toegevoegd.

Implementatie

Als er een bevredigend ontwerp ligt, kan begonnen worden met de *implementatie*. Die implementatie kent haar eigen ontwerpfase: per klasse moet worden vastgesteld *hoe* deze klasse haar verantwoordelijkheden gaat verwezenlijken. Eventueel kan dit worden vastgelegd in een apart implementatiemodel waarin alle attributen met hun typen zijn vastgelegd en alle methoden met hun signatuur (de kop van de methode) en met een beschrijving van hun werking.
Vervolgens wordt de klasse gecodeerd. Omdat, tijdens de ontwerpfase, de interface van een klasse gedetailleerd is vastgelegd, kunnen in deze fase verschillende programmeurs onafhankelijk van elkaar aan verschillende klassen werken.

Testen

Het *testen* van een bepaalde constructiestap verloopt ook weer in fasen. Elke klasse wordt eerst afzonderlijk getest. De programmeur zal daarvoor een kleine testomgeving construeren. In Objectgeoriënteerd programmeren in Java 1 maakten we hiervoor gebruik van JUnit. Als alle klassen lijken te werken, worden ze samengevoegd en wordt het programma als geheel getest. Hierbij zullen eventuele misverstanden tussen de verschillende programmeurs aan het licht komen. Als het implementatiemodel volledig en eenduidig is en alle programmeurs volmaakt zijn, dan zijn die misverstanden er niet. De praktijk leert dat aan die voorwaarden vaak niet is voldaan.
Iedereen die regelmatig met computers werkt, weet hoe moeilijk testen is: in ieder softwarepakket van enige omvang blijken toch altijd weer fouten te zitten. Om dat aantal zo klein mogelijk te houden, is een goede teststrategie van zeer groot belang. Het zal duidelijk zijn dat voor ieder programma, hoe klein ook, slechts een miniem deel van alle mogelijke combinaties van invoerwaarden daadwerkelijk kan worden uitgeprobeerd. De kunst is nu dat deel zo te kiezen, dat het zo veel mogelijk representatief is voor alle invoer. In leereenheid 13 van Objectgeoriënteerd programmeren met Java 1 is aandacht besteed aan teststrategieën. Leereenheid 6 van deze cursus geeft daarop nog enkele aanvullingen en gaat ook in op zoeken naar de oorzaak van een fout zodat die hersteld kan worden (debuggen).

Afronding

Er komt een moment waarop ook de laatste gebruiksmogelijkheid is toegevoegd en getest en het systeem dus in principe af is. Er is dan vaak nog een *afrondingsfase* waarin de opdrachtgever bijvoorbeeld het systeem al ter beschikking krijgt om het uit te proberen. Of, als het een nieuw pakket betreft, wordt op dat moment misschien een gratis te downloaden bètaversie op internet gezet, zodat potentiële klanten deze uit kunnen proberen. Daar komen altijd nog fouten uit, die in de afrondingsfase verbeterd kunnen worden. Tot slot wordt het systeem dan echt overgedragen of vrijgegeven voor verkoop.

Het hier geschetste ontwikkeltraject is niet het enig mogelijke. Er zijn bijvoorbeeld andere faseringen mogelijk. Als ook de analyse en de afronding in fasen worden uitgevoerd, bestaat het hele traject uit het herhaaldelijk doorlopen van de stappen analyse, ontwerp, implementatie, testen en afronding van de huidige fase. De opdrachtgever heeft dan al snel een klein systeem ter beschikking. Na iedere volgende fase zullen de mogelijkheden zijn uitgebreid.

Prototype

Op deze wijze kan ook een *prototype* worden ontwikkeld: een experimentele versie van een te realiseren systeem. Meestal wordt een prototype gebruikt om inzicht te krijgen in de werking van een uiteindelijk programma. Aan de hand van zo'n prototype kan worden getoetst of aan alle eisen is voldaan en of het programma aan de verwachtingen van de opdrachtgever voldoet. Doordat de opdrachtgever in een vroeg stadium met het prototype kan 'spelen', is het mogelijk het ontwerp tijdig in een volgende fase bij te stellen.
Aan de andere kant zou het volledige ontwerp en de implementatie in een keer kunnen worden gedaan, zodat de constructie uit slechts één stap bestaat. Hoe groter het systeem is, hoe sterker dit laatste echter moet worden afgeraden: behoorlijk testen van heel veel code ineens is vrijwel onmogelijk. Op de voor- en nadelen van verschillende faseringen gaan we hier verder niet in.

Onderhoud

Als het systeem is opgeleverd of op de markt is gebracht, is daarmee de kous nog lang niet af. Het systeem moet namelijk ook *onderhouden* worden en in de praktijk is daarmee vaak veel meer tijd en geld gemoeid dan met de oorspronkelijke constructie.
Waar bestaat dat onderhoud uit? Ten eerste zullen er, hoe goed er ook getest is, nog steeds fouten in het programma zitten, die in een volgende versie verbeterd moeten worden. Ten tweede zullen er veranderingen in het domein optreden of in de omgeving waarin het programma draait, die aanpassingen in het systeem nodig maken. Ten derde kunnen er uitbreidingen van het programma nodig zijn, dat wil zeggen dat er nieuwe gebruiksmogelijkheden aan moeten worden toegevoegd.
Ook hiervan kan de oorzaak liggen in veranderingen in het domein, maar het kan ook zijn dat het regelmatige gebruik van het systeem de gebruiker op nieuwe ideeën brengt: 'het zou toch ook wel handig zijn als ...', of dat de voortdurende voortschrijdende techniek nieuwe mogelijkheden binnen bereik brengt.

OPGAVE 1.2
Bedenk eens een paar veranderingen in de omstandigheden die een wijziging of een uitbreiding van het containersysteem nodig maken.

Een programma kan op deze wijze vele jaren meegaan, in steeds weer nieuwe gedaanten, zonder ooit echt uit de roulatie te worden genomen. Uit het oogpunt van softwarearchitectuur zou het misschien verstandig zijn om programma's elke vijf of hoogstens tien jaar te vervangen door een volledig nieuwe versie die van de grond af aan is herontwikkeld. Daar zijn echter vaak zulke hoge kosten mee gemoeid dat een dergelijke oplossing economisch niet haalbaar is. Het millenniumprobleem, waar in de laatste jaren van de vorige eeuw veel aandacht voor was, kan hier gebruikt worden als illustratie: de oerversie van sommige programma's waarin jaartallen met slechts twee cijfers gerepresenteerd werden, dateerde nog uit de jaren zestig van de vorige eeuw!

In de totale levensloop van een groot systeem is onderhoud dus minstens zo belangrijke factor als de oorspronkelijke ontwikkeling.

1.2 WAT IS EEN GOED PROGRAMMA?

Welke eisen kunnen we nu, in het licht van de zojuist geschetste levensloop van softwaresystemen, stellen aan ontwerp en implementatie van een dergelijk systeem?

Correctheid

Allereerst moet een programma uiteraard *correct* zijn: het moet doen wat het volgens de specificaties zou moeten doen. Als gebruik gemaakt wordt van formele specificatietechnieken, kan de correctheid van een programma in principe met wiskundige methoden bewezen worden.
In deze cursus zullen we dat niet doen; in plaats daarvan zullen we, net als vrijwel altijd in de praktijk gebeurt, via testen fouten opsporen en die vervolgens herstellen. Realiseert u zich wel dat deze strategie beperkingen kent. Een goede teststrategie kan veel fouten aan het licht brengen, maar via testen kan nooit worden aangetoond dat het uiteindelijke programma correct is.
Correctheid is overigens een vrij beperkt begrip. In feite moet een programma doen wat de opdrachtgever wil of wat de toekomstige gebruikers verlangen. Als de specificatie daarmee achteraf niet in overeenstemming blijkt en de gebruikers ervaren het programma als onhandig of zelfs onbruikbaar, dan is het programma niet goed, zelfs niet wanneer bewezen is dat het correct is.

Robuustheid

Correctheid alleen is bovendien niet genoeg. Een programma moet ook *robuust* zijn, ofwel: het moet tegen mogelijke fouten bestand zijn. Dat kunnen fouten van de gebruiker zijn: bijvoorbeeld het invoeren van letters als een getal wordt verwacht, of het opgeven van een niet-bestaande datum, of het opvragen van gegevens uit een bestand die daar niet in zitten, of het kiezen van een verkeerde menuoptie waardoor een onbedoelde interactie wordt gestart of juist het abusievelijk afbreken van een lopende interactie.
In zulke gevallen verwachten we een redelijke respons van de software waarmee we werken: een geluid dat aangeeft dat we iets doen wat niet klopt, een heldere maar beknopte foutmelding, een mogelijkheid om de gestarte interactie meteen weer af te breken, of een waarschuwing dat we op het punt staan (veel) werk weg te gooien. Het programma moet bovendien bestand zijn tegen allerlei andere onvoorziene omstandigheden, zoals een harde schijf die vol is zodat er geen data kunnen worden weggeschreven.

In Objectgeoriënteerd programmeren in Java 1 hebben we aan deze eigenschap geen aandacht besteed. We hebben daar applicaties gemaakt die deden wat ze moesten doen, maar die niet werkten als de invoer niet aan de verwachtingen voldeed. Meestal leidde dat tot een aantal foutboodschappen, waarmee de gebruiker van een systeem bij voorkeur nooit geconfronteerd wil worden. In deze cursus zullen we aan dit aspect een aparte leereenheid wijden (leereenheid 7).

Begrijpelijke code

Wil een programma onderhouden kunnen worden, dan moet de code *begrijpelijk* zijn. De programmeur die een fout moet verbeteren of een wijziging aanbrengen, is vaak een andere dan degene die de code oorspronkelijk heeft geschreven. Als de tijd tussen schrijven en veranderen langer dan een paar maanden is, maakt dat bovendien weinig verschil meer: programmeurs staan dan net zo vreemd tegenover hun eigen code als tegenover die van ieder ander.

Een manier om de begrijpelijkheid van code te verbeteren, is een goede documentatie van het programma in de vorm van uitgebreid commentaar. In onze Java-cursussen gebruiken we daarvoor veelal javadoc, aangevuld met gewoon commentaar bij lastige stukken broncode.

Uitbreidbaarheid

Ook de structuur van het programma moet dusdanig zijn, dat het *makkelijk te wijzigen en uit te breiden* is. Een wijziging is makkelijker naarmate deze op minder delen van het programma invloed heeft. Wanneer op twintig plekken iets gewijzigd moet worden, zien we er gauw een over het hoofd. Een uitbreiding is gemakkelijker naarmate het nieuwe stuk zelfstandiger te ontwikkelen is en tot minder wijzigingen elders in het programma leidt.

Er zijn nog meer eisen waar een programma aan moet voldoen, maar daar besteden we in deze cursus minder aandacht aan. We noemen er nog twee.

Herbruikbaarheid

Een gewenste eigenschap die objectoriëntatie populair heeft gemaakt, is *herbruikbaarheid* van delen van het systeem. Soms kan het veel ontwikkeltijd besparen als een klasse die voor een bepaalde toepassing gemaakt is, ook in een andere toepassing kan worden ingezet. Het belang van deze eigenschap is u in feite al bekend: de Java API is in wezen niets anders dan een verzameling herbruikbare klassen.

Efficiëntie

Tot slot is voor sommige programma's *efficiëntie* heel belangrijk. Dit geldt bijvoorbeeld voor:
– toepassingen waarbij het programmaverloop een proces in de buitenwereld moet bijhouden, zoals een industriële robot die moet reageren op aanvoer van onderdelen op een band
– toepassingen waarbij een factor tien in geheugengebruik of tijd net de grens tussen haalbaar en niet haalbaar vormt: een weervoorspelling voor de komende week mag een dag rekentijd vragen, maar geen tien dagen
– pakketten waarbij de interactie met de gebruiker zeer intensief is en wachten al snel hinderlijk wordt: besturingssystemen, tekstverwerkers, ontwikkelomgevingen.

Al deze eisen zijn op zich redelijk voor de hand liggend. De vraag is echter, hoe we die eisen vertalen in eigenschappen van programmacode. Daarbij zullen we ons in de rest van deze paragraaf concentreren op begrijpelijkheid en gemak van wijzigen en uitbreiden. In tegenstelling tot bijvoorbeeld robuustheid en efficiëntie zijn dit namelijk de eisen waarmee van het begin af aan rekening moet worden gehouden.
Het is mogelijk om een niet-robuust, maar begrijpelijk en makkelijk te wijzigen programma, achteraf alsnog robuust te maken. Een ondoorgrondelijk en moeilijk te wijzigen programma achteraf alsnog begrijpelijk maken is niet echt onmogelijk, maar wel uiterst moeilijk. Vaak vereist het een volledig nieuw ontwerp en implementatie.

OPGAVE 1.3
Noem een paar eigenschappen van programmacode die de begrijpelijkheid ervan bevorderen.

De eigenschappen genoemd in de terugkoppeling bij opgave 1.3 liggen heel erg voor de hand, maar er zijn er meer.

Een heel belangrijke eigenschap die de begrijpelijkheid van code bevordert is *eenvoud*. In het algemeen zullen klassen en methoden moeilijker te begrijpen zijn naarmate ze groter zijn. Het is daarom goed om te streven naar *kleine klassen en methoden*. Zoals u bij het gebruik van de API specification gemerkt zult hebben, voldoen de klassen uit de Java API niet altijd aan deze norm. Ook is een methode begrijpelijker naarmate de *control flow eenvoudiger* is. Daarom proberen we een diepe nesting van while's, for's en if's te vermijden.

Kleine klassen en methoden

Eenvoudige control flow

Gescheiden verantwoordelijkheden

Een andere belangrijke manier om eenvoud te bereiken, is verwoord in het volgende principe: *ieder onderdeel van de code dient slechts één verantwoordelijkheid te hebben.* Dit principe geldt voor alle niveaus: opdrachten, methoden en klassen.

Opdrachten

– Op het niveau van individuele opdrachten gebruiken we vanwege dit principe nooit de waarden van expressies met neveneffecten. Java kent bijvoorbeeld opdrachten van de vorm *k*++ en ++*k*. Beide verhogen de waarde van de int-variabele *k* met 1. Deze uitdrukkingen hebben echter ook een waarde. De waarde van *k*++ is de waarde van *k* voor verhoging; de waarde van ++*k* is de waarde van *k* na verhoging. We kunnen bijvoorbeeld schrijven

```
int n = 3;
int m = 3;
int s = n++ + 1;
int t = ++m + 1;
```

Hierna zijn n en m beide gelijk aan 4, s is ook gelijk aan 4 (3 + 1) en t is gelijk aan 5 (4 + 1). Sommige programmeurs gebruiken dit soort opdrachten graag om compacte code te schrijven. De kleinste deler van een geheel getal n kan bijvoorbeeld gevonden worden met behulp van het volgende programmafragment:

```
deler = 1;
while ((n % ++deler) != 0);
```

In deze opdracht heeft de test twee verantwoordelijkheden: de variabele deler wordt verhoogd en er wordt bekeken of de rest van n bij deling door de nieuwe waarde van deze variabele ongelijk is aan 0. De while-opdracht bevat alleen een test en geen opdracht. Die test wordt dus eenvoudig herhaald tot de rest nul is en deler dus gelijk is aan de kleinste deler van n.
De volgende code is weliswaar iets langer, maar veel begrijpelijker:

```
deler = 2;
while (n % deler != 0) {
  deler++;
}
```

De kortste manier om iets op te schrijven, is dus niet bij voorbaat de eenvoudigste!

Methoden

– Op methodeniveau proberen we methoden te vermijden die zowel een waarde opleveren als een of meer attributen wijzigen. Een dergelijke methode heeft twee verantwoordelijkheden: iets aan de toestand van het object veranderen en een resultaat teruggeven.

Er zijn op deze regel overigens wel uitzonderingen. Het is bijvoorbeeld niet verkeerd om een methode een waarde terug te laten geven die laat zien of de actie die de methode moet uitvoeren, geslaagd is. Een voorbeeld hiervan is de methode

```
public boolean add(E e)
```

in de klasse ArrayList. Deze methode geeft de waarde true terug als het toevoegen van het element geleid heeft tot een verandering van de arraylist, en false als het toevoegen niet gelukt is.

Klassen

– Op klassenniveau zullen we het principe van gescheiden verantwoordelijkheden als een van de leidraden gebruiken bij de keuze van klassen in een ontwerp. Het is een goede gewoonte om bij het ontwerp het doel van een klasse – we kunnen ook zeggen: de verantwoordelijkheid van de klasse – in een kort zinnetje expliciet te vermelden. Als in zo'n zinnetje het woordje 'en' voorkomt, moet op zijn minst onderzocht worden of die klasse niet beter gesplitst kan worden, zoals in: deze klasse representeert een order en coördineert de toewijzing van containers aan die order. Het antwoord is in elk geval 'ja' als dat kan gebeuren zonder dat er veel extra interactie tussen die klassen nodig is.

Lage koppeling

Nog een andere manier om een programma eenvoudiger te maken, is het aantal associaties tussen klassen te beperken ofwel de *koppeling* tussen de klassen laag te houden. Als we een systeem hebben met tien klassen en iedere klasse is afhankelijk van alle andere, dan is dat systeem vrijwel zeker moeilijk te doorgronden en te wijzigen. Een wijziging in een klasse maakt dan al snel wijzigingen in alle andere klassen noodzakelijk. Hergebruik van één klasse in een ander systeem is al helemaal onmogelijk.

Gescheiden verantwoordelijkheden maken code gemakkelijker te begrijpen en alleen al daarom ook gemakkelijker te wijzigen en uit te breiden. Een wijziging of uitbreiding is echter ook des te gemakkelijker, naarmate deze op minder plaatsen in de code van invloed is. Stel bijvoorbeeld dat een wijziging invloed heeft op de signatuur van drie methoden uit drie verschillende klassen, en dat elk van die drie methoden op vijf verschillende plaatsen wordt aangeroepen. Om die ene wijziging door te voeren, moeten we dan op tenminste achttien plaatsen in de code iets veranderen: de drie methoden plus de vijftien aanroepen.
Niet alleen moet elk onderdeel van de code slechts één verantwoordelijkheid hebben, iedere verantwoordelijkheid van het systeem als geheel moet bij voorkeur ook slechts op één plek in de code gerealiseerd zijn. Als we dan iets wijzigen aan die verantwoordelijkheid, dan hoeven we alleen op die plek iets te veranderen. We noemen deze eis aan de code het principe van *lokaliteit*.

Lokaliteit

Constanten

Ook dit principe speelt op verschillende niveaus. De eis van lokaliteit is bijvoorbeeld een van de redenen dat we *constanten* definiëren. Stel er zijn in het containersysteem vijf verschillende typen containers en dat aantal komt op tien plaatsen in de code voor. Als we op al die tien plaatsen het getal 5 neerzetten en er komt een type container bij, dan moeten we tien wijzigingen aanbrengen. Is er ergens in het systeem een constante AANTALCONTAINERTYPEN gedefinieerd, dan hoeven we alleen de waarde van die constante te veranderen.

OPGAVE 1.4
Welke andere reden is er om constanten te definiëren?

Ook onze voorkeur voor attributen die van buiten de klasse niet gewijzigd kunnen worden (information hiding), is terug te voeren op het principe van lokaliteit: de verantwoordelijkheid voor het beheer van een dergelijk attribuut ligt geheel binnen de klasse, in plaats van verspreid door het hele programma. Met het oog daarop, maken we attributen vrijwel altijd private en zijn we voorzichtig met het exporteren van referenties naar attributen van een objecttype.

Het bereiken van lokaliteit is in de praktijk een van de moeilijkste zaken bij objectgeoriënteerd programmeren (en bij programmeren in het algemeen).

Voorbeeld

Op grond van het principe van lokaliteit willen we aan een klasse gemakkelijk een nieuwe subklasse kunnen toevoegen. Stel bijvoorbeeld dat er in het containersysteem een klasse Container is, met een subklasse voor elke soort container. Als we nu een nieuw soort container willen toevoegen, zullen we in elk geval een nieuwe subklasse van Container moeten definiëren. Daarnaast zijn wijzigingen nodig op plekken in de code waar instanties van de subklassen van Container gecreëerd worden en op plekken waar het nodig is na te gaan met welk type container we nu precies te maken hebben. Lokaliteit vereist dat dit op zo min mogelijk plaatsen voorkomt. In de volgende leereenheid zullen we zien hoe Java ons daarbij helpt.

1.3 OBJECTKEUZE

Het is lang niet gemakkelijk om programmacode te schrijven met al de gewenste eigenschappen. Het kiezen van goede namen is vooral een kwestie van zelfdiscipline, evenals het schrijven van commentaar. Maar voor het schrijven van code die ook aan de andere eisen voldoet, is ervaring nodig. Een beginnend programmeur ziet vaak maar één manier om een probleem aan te pakken. Hij zal daarom al gauw tot de conclusie komen dat déze grote klasse echt niet gesplitst kan worden, of dat déze methode echt zowel een waarde moet opleveren als de toestand van een object moet veranderen. Soms heeft een andere oplossing inderdaad meer nadelen dan voordelen, maar meestal ziet een programmeur met enige ervaring wel hoe het anders kan. Nog veel meer ervaring is nodig om klassen dusdanig te kiezen en uit te werken, dat de resulterende code voldoet aan de principes van gescheiden verantwoordelijkheden, lage koppeling en lokaliteit.

De verschillende eisen aan de code zijn bovendien soms met elkaar in conflict. Soms gaan een vergaande scheiding van verantwoordelijkheden en een grote lokaliteit ten koste van de eenvoud van een ontwerp. Er moet dan een afweging worden gemaakt. Het is daarbij heel belangrijk om te anticiperen op het soort wijzigingen dat later nodig kan zijn. Schrijven we bijvoorbeeld een schaakprogramma, dan mogen we de afmetingen van het bord, de aard van de stukken en de geoorloofde zetten rustig fixeren: het is een veilige aanname dat die niet zullen veranderen. Het is daarentegen zeer belangrijk om de generator van een zet

zo flexibel mogelijk te maken: we willen zeker in de toekomst nieuwe strategieën toevoegen.
Naarmate een programma groter is, de verwachte levensduur langer is en het minder voorspelbaar is welke wijzigingen er later noodzakelijk zijn, wordt het belangrijker om te kiezen voor scheiding van verantwoordelijkheden en lokaliteit. In deze cursus zullen we echter ook kiezen voor de eenvoud van het ontwerp.

2 UML-diagrammen

In Objectgeoriënteerd programmeren in Java 1 worden verschillende soorten diagrammen gebruikt: klassendiagrammen en een enkele keer sequentiediagrammen als hulpmiddel bij het ontwerpen van programma's en toestandsdiagrammen voor inzicht in het programmaverloop.

Unified Modeling Language (UML)

De *Unified Modeling Language*, afgekort UML, is een modelleertaal voor objectgeoriënteerde systeembeschrijvingen. UML biedt verschillende diagramtechnieken en is sinds de lancering in 1997 tot een standaard uitgegroeid. UML kan tijdens de volledige ontwikkelingscyclus gebruikt worden:
– tijdens de analyse voor het opstellen van een specificatie en van een domeinmodel
– tijdens het technisch ontwerp voor de specificatie van een programmastructuur (inclusief indeling in packages)
– tijdens de implementatie voor een volledige beschrijving van klassen en de interactie tussen objecten.

UML standaard: zie www.uml.org

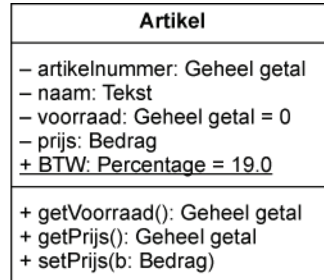
UML is ontworpen en wordt verder ontwikkeld door de Object management group (OMG), een volledige beschrijving kan worden geraadpleegd via de website www.uml.org.

In paragraaf 2.1 komen klassendiagrammen aan de orde. Ter herinnering: een klassendiagram laat zien welke klassen er zijn ontworpen, welke attributen en methoden deze hebben en in welke relatie de klassen tot elkaar staan. In Objectgeoriënteerd programmeren in Java 1 werden klassendiagrammen vooral gebruikt op implementatieniveau. In verband daarmee is in die cursus gekozen voor een notatie die zo dicht mogelijk bij Java staat, maar die niet overeenkomt met de UML-standaard. In deze cursus, waar we UML vooral op ontwerpniveau gebruiken, willen we ons wel conformeren aan die standaard.
In paragraaf 2.2 besteden we kort aandacht aan de objectdiagrammen uit UML. Deze beschrijven de toestand van objecten op een bepaald tijdstip en zijn de UML variant van de (complexere) toestandsdiagrammen die gebruikt worden in Objectgeoriënteerd programmeren in Java 1.
De sequentiediagrammen gebruikt in Objectgeoriënteerd programmeren in Java 1 volgden al de UML standaard. Deze worden daarom niet meer besproken.

2.1 KLASSENDIAGRAMMEN

Een klassendiagram toont klassen en hun samenhang. Klassendiagrammen kunnen tijdens analyse, ontwerp en implementatie gebruikt worden. In elke fase wordt alleen datgene in het klassendiagram opgenomen, wat in die fase belangrijk is. Het is dus belangrijk om te weten waarvoor een klassendiagram op een bepaald moment gebruikt wordt.

In deze cursus wordt analyse niet als aparte fase behandeld. Wel zullen we klassendiagrammen gebruiken bij ontwerp en implementatie van programma's.



FIGUUR 1.2 De klasse Artikel

Notatie van een klasse

We bekijken eerst de notatie voor een klasse. Figuur 1.2 toont een voorbeeld. Het diagram bestaat uit drie delen: in het bovenste deel staat vetgedrukt de naam van de klasse, in het middelste deel staan attributen en in het onderste deel staan methoden.

Het middelste deel van het diagram bevat attributen. Volgens de UML-standaard staat het type van de attributen na de attribuutnaam en niet zoals in Java ervoor. Naam en type worden gescheiden door een dubbele punt.
Op ontwerpniveau worden soms geen typeaanduidingen gebruikt die direct aan Java zijn ontleend, zoals int, double en String maar algemenere, zoals Geheel getal, Bedrag en Percentage in figuur 1.2. De precieze representatie kan dan later nog worden vastgesteld.

In figuur 1.2 heeft de klasse Artikel de attributen artikelnummer en voorraad van het type Geheel getal, naam van het type Tekst, prijs van het type Bedrag en BTW van het type Percentage. Dit laatste attribuut is een klassenattribuut: een attribuut dat hoort bij de klasse en dat dus voor alle instanties van die klasse dezelfde waarde heeft. In UML worden dergelijke attributen onderstreept. BTW is een constante en krijgt de waarde 19.0. Bij creatie van een instantie van Artikel krijgt het attribuut voorraad de waarde 0. Dit is de zogeheten standaardwaarde van dat attribuut.

Syntaxis attribuut

Voor attributen hanteert UML de volgende syntaxis:

```
[toegang] naam[: type] [= waarde]
```

waarbij alles tussen rechte haken mag worden weggelaten.
Toegang geeft aan of het attribuut public (+), private (–) of protected (#) is (ter herinnering: protected betekent dat het attribuut toegankelijk is voor alle klassen uit dezelfde package en voor alle subklassen). Wordt geen toegang vermeld, dan kan dat betekenen dat er in het diagram geen toegang gespecificeerd is; de toegang is immers niet verplicht. Dat zal gelden voor veel klassendiagrammen in deze cursus. Het kan echter ook betekenen dat het attribuut toegankelijk is voor klassen uit dezelfde package. Het verschil tussen deze twee gevallen is niet zonder meer te zien.

Naam en *type* geven de naam en het type van het attribuut weer. De aanduiding = *waarde* geeft een standaardwaarde aan, die het attribuut krijgt bij creatie van een instantie van de klasse.

UML laat de opsteller van het klassendiagram dus veel vrijheid: indien gewenst, kan van een attribuut enkel de naam worden vermeld.

OPGAVE 1.5
Geef een specificatie van een private-attribuut van de klasse Artikel met de naam verkrijgbaar en standaardwaarde onwaar.

Ook voor methoden laat UML de opsteller alle vrijheid om meer of minder informatie op te nemen. De minimale aanduiding bestaat uit de naam van de methode plus een paar haakjes. Desgewenst zullen we daar aanduidingen aan toevoegen van aantal en typen van de parameters, en van het type van de terugkeerwaarde. Ook hier staan typeaanduidingen altijd na de naam. In figuur 1.2 heeft de klasse Artikel methoden die de waarden van prijs respectievelijk voorraad teruggeven (getPrijs en getVoorraad) en een methode die een nieuwe waarde aan de prijs toekent (setPrijs).

Syntaxis methode

De volledige syntaxis voor methodeaanduidingen is

```
[toegang] naam([parameterlijst])[: resultaattype]
```

Voorbeeld

Een methode van een klasse Persoon die gebruikt kan worden om te testen of een persoon ouder is dan een ander, zou op implementatieniveau voluit als volgt kunnen worden genoteerd:

```
+ouderDan(p: Persoon): boolean
```

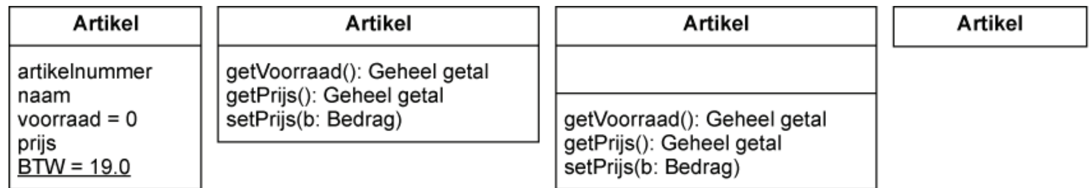
Informatie over de toegang tot attributen en methoden nemen we in het algemeen niet op in onze klassendiagrammen, zeker niet als alle niet-constante attributen private zijn en alle methoden public.

OPGAVE 1.6
Gegeven een klasse Cirkel, met een attribuut straal met standaardwaarde 1. De klasse heeft methoden om de omtrek en het oppervlak van de cirkel te berekenen, alsmede een methode om de cirkel in een vlak te plaatsen met het middelpunt op gegeven coördinaten. Zoals gewoonlijk zijn attributen private en methoden public. Teken een klassendiagram waarin deze informatie over attributen en methoden is opgenomen.

Methoden worden in UML overigens *operaties* genoemd. In deze cursus zullen we ons echter aan de Java-terminologie houden.

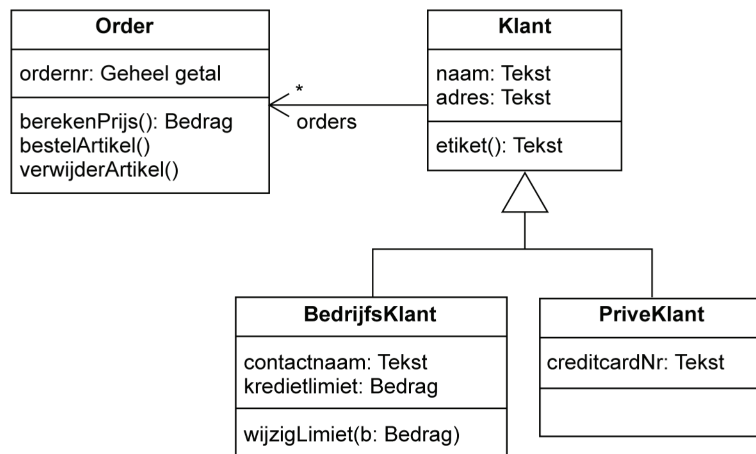
Attributen en/of methoden kunnen worden weggelaten uit een diagram. Figuur 1.3 toont enkele alternatieve notaties voor de klasse Artikel. Als in een diagram een klasse voorkomt zonder attributen of methoden, kan dat betekenen dat de klasse die niet heeft, dat ze nog niet bekend zijn of dat ze zijn weggelaten omdat ze in de context waarin het diagram gebruikt wordt, niet van belang zijn. In een diagram waarin tien klassen voorkomen, is het bijvoorbeeld goed om zich te concentreren op de

relaties tussen die klassen; attributen en methoden kunnen dan alleen bij naam worden aangeduid of zelfs geheel worden weggelaten. Uit de context waarin het diagram gebruikt wordt moet uiteraard wel duidelijk worden wat precies het geval is.



FIGUUR 1.3 Andere notaties voor de klasse Artikel

We bekijken vervolgens de *relaties* tussen de klassen. Figuur 1.4 toont een klassendiagram met twee soorten relaties.



FIGUUR 1.4 Generalisatie en associatie

Generalisatie

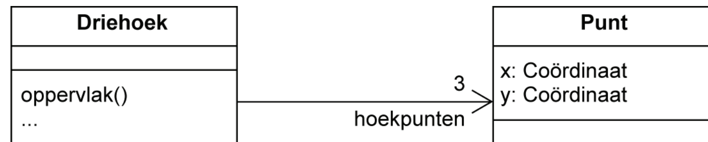
De overervingrelatie, de *generalisatie*, kent u uit Objectgeoriënteerd programmeren in Java 1. De klasse Klant uit figuur 1.4 heeft twee subklassen BedrijfsKlant en PriveKlant. De superklasse Klant heeft de attributen naam en adres en een methode etiket die worden geërfd door de subklassen. Een bedrijfsklant heeft nog twee extra attributen en een extra methode; een privé-klant heeft één extra attribuut.

Associatie

UML toont associaties niet als attributen

Bij een klant hoort een lijst orders. Dit is in het diagram weergegeven als een *associatie* van Klant naar Order. Merk echter op dat in het diagram van de klasse Klant, in tegenstelling tot hetgeen u op grond van de cursus Objectgeoriënteerd programmeren in Java 1 zou verwachten, *geen attribuut* orders van bijvoorbeeld het type ArrayList<Order> is opgenomen. De pijl in figuur 1.4 betekent dat iedere instantie van klasse Klant verwijst naar een aantal instanties van Order.
De reden dat we associaties niet (meteen) als attributen in klassen willen opnemen is dat klassendiagrammen worden gebruikt tijdens verschillende fasen van de ontwikkeling. Een associatie vertalen naar een attribuut betekent vaak al kiezen voor een bepaalde implementatie, op een moment dat dat nog helemaal niet nodig is.

Laten we als illustratie kijken naar een driehoek met drie hoekpunten. Figuur 1.5 toont een mogelijk klassendiagram, gemaakt in de ontwerpfase.



FIGUUR 1.5 Ontwerp van een klasse Driehoek

Op grond van dit klassendiagram kunnen we in de implementatie een attribuut hoekpunten opnemen van het type ArrayList<Punt> of van het type Punt[]. Maar we kunnen ook drie aparte attributen puntA, puntB en puntC van type Punt opnemen, of voor nog een andere implementatie kiezen. Java biedt meer mogelijkheden dan alleen de array of ArrayList om een meervoudige waarde te representeren. Tijdens het ontwerp willen we dat niet vastleggen.

Eenzijdige associatie

Tweezijdige associatie

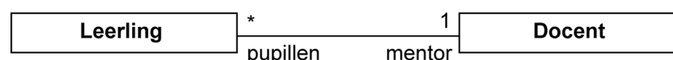
UML kent zowel eenzijdige als tweezijdige associaties, zie figuur 1.6. De *eenzijdige associatie*, een associatie in één richting, wordt getekend met een pijl. In dit geval verwijst een instantie van klasse A naar een instantie van klasse B, maar niet omgekeerd. Bij de *tweezijdige associatie*, een associatie in beide richtingen, verwijst een instantie van A naar een instantie van B, maar omgekeerd verwijst een instantie van B ook naar een instantie van A.

FIGUUR 1.6 Eenzijdige en tweezijdige associatie

Rol

Let op!

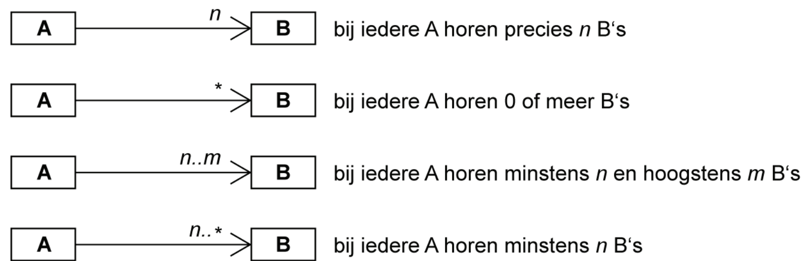
Anders dan in Objectgeoriënteerd programmeren in Java 1, zullen we aan associaties vrijwel altijd namen en multipliciteiten toevoegen. Kijk als voorbeeld naar figuur 1.7, die een tweezijdige associatie toont tussen de klassen Leerling en Docent. Die associatie kan bijvoorbeeld geïmplementeerd worden door de klasse Leerling een attribuut mentor te geven van type Docent en Docent een attribuut pupillen van type ArrayList<Leerling>. UML spreekt in dit geval niet van attributen: in plaats daarvan wordt gezegd dat de associatie tussen Leerling en Docent een *rol* mentor heeft (die wordt ingevuld door een instantie van Docent) en een *rol* pupillen (die wordt ingevuld door instanties van Leerling). De naam van een rol staat aan de kant van de klasse die de rol vervult, niet aan de kant van de klasse die de rol heeft.



FIGUUR 1.7 Een associatie met rolnamen en multipliciteitaanduidingen

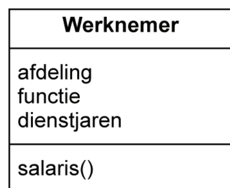
Multipliciteit

Figuur 1.7 toont ook aan beide kanten een aanduiding van de *multipliciteit* van de rol (ook wel kardinaliteit genoemd). Een leerling heeft precies één mentor (de aanduiding 1), maar een docent kan mentor zijn van 0 of meer verschillende leerlingen (aanduiding *).
Figuur 1.8 toont de verschillende aanduidingen voor multipliciteit. Als er geen multipliciteitaanduiding aan één van de zijden van de associatie staat, dan is de multipliciteit aan die zijde gelijk aan 1.



FIGUUR 1.8 Multipliciteit

OPGAVE 1.7
Het ontwerp van een informatiesysteem voor een salarisadministratie bevat een klasse Werknemer, met attributen en een methode als getoond in figuur 1.9.

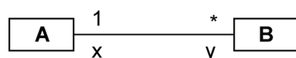


FIGUUR 1.9 Klasse Werknemer

Voor iedere werknemer moet bovendien worden bijgehouden wie daarvan de chef is (zelf ook een werknemer), en wie de eventuele ondergeschikten. Het betreft hier relaties tussen werknemers onderling en dus een associatie van de klasse Werknemer met zichzelf. Teken deze associatie, inclusief namen en multipliciteit.

De volgende opgave dient om u er nog eens duidelijk op te wijzen aan welke kant de naam en multipliciteit van een rol getekend moeten worden.

OPGAVE 1.8
Bekijk figuur 1.10.

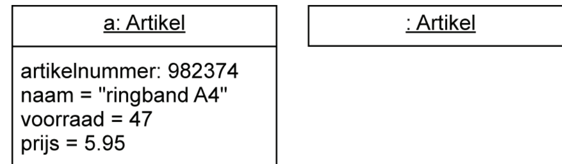


FIGUUR 1.10 Een associatie tussen A en B

Als deze associatie wordt geïmplementeerd door attributen in de klassen A en B, wat zijn dan voor de hand liggende namen en typen voor die attributen?

2.2 OBJECTDIAGRAMMEN

Uit Objectgeoriënteerd programmeren in Java 1 kent u de toestandsdiagrammen. Deze diagrammen komen in UML niet voor. UML kent wel een ander soort objectdiagram; zie de voorbeelden in figuur 1.11.



FIGUUR 1.11 Voorbeelden van objectdiagrammen van een instantie van klasse Artikel

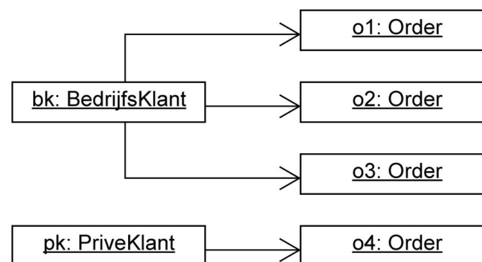
Het linkerdeel van figuur 1.11 toont een objectdiagram van een instantie a van klasse Artikel. De kop wordt *niet* vetgedrukt maar onderstreept en vermeldt in dit geval de naam van de instantie en het type. Het deel onder de streep toont de objectattributen en hun waarden. Klassenattributen worden in een objectdiagram niet getoond: ze horen immers bij de klasse en niet bij het object. Methoden worden in een objectdiagram evenmin getoond.

Anonieme instantie

Het rechterdeel van figuur 1.11 toont een zogenaamde *anonieme instantie*: in de kop staat alleen de klasse vermeld, voorafgegaan door een dubbele punt. In dit deel zijn bovendien de attribuutwaarden weggelaten; dat is toegestaan als we daar niet in geïnteresseerd zijn (ook bij niet-anonieme instanties).

Link

In een objectdiagram kunnen *links* voorkomen: een pijl van object a naar object b betekent dat object b bij object a hoort, of anders gezegd, dat object a object b kent en dus bijvoorbeeld een methode op object b kan aanroepen. In de implementatie betekent dit bijvoorbeeld dat object a een attribuut heeft met object b als waarde. Figuur 1.12 toont als voorbeeld links tussen twee klanten en vier orders.



FIGUUR 1.12 Een objectdiagram

Net als associaties kunnen links eenzijdig of tweezijdig zijn. De links uit figuur 1.12 zijn vanwege de pijl eenzijdig; Bedrijfsklant bk kent dus bijvoorbeeld order o1 maar het omgekeerde geldt niet. Als de link tussen deze twee objecten tweezijdig was (zonder pijl), dan zou order o1 ook BedrijfsKlant bk kennen.

OPGAVE 1.9
a Leg uit dat het objectdiagram uit figuur 1.12 in overeenstemming is met het klassendiagram uit figuur 1.4.
b Waarom staan er bij links tussen objecten geen multipliciteitaanduidingen?

3 Een eenvoudige simulatie van een bank

In de vorige paragraaf hebt u twee diagramtechnieken gezien die nuttig kunnen zijn bij het ontwerpen van een objectgeoriënteerd programma: het klassendiagram en het objectdiagram. Maar hoe komt u nu aan een dergelijk ontwerp? Hiervoor bestaat helaas geen eenvoudig recept; het maken van een goed ontwerp is vooral een kwestie van veel ervaring. Het kan wel helpen om de richtlijnen in het achterhoofd te houden die we in paragraaf 1 verstrekten.

In deze paragraaf laten we aan de hand van een voorbeeld zien hoe u te werk kunt gaan. In tegenstelling tot het containervoorbeeld uit paragraaf 1, past dit voorbeeld qua complexiteit goed in deze cursus. Het betreft een klein systeem voor simulatie van een eenvoudige bank.

3.1 PRODUCTOMSCHRIJVING EN GEBRUIKSMOGELIJKHEDEN

Productomschrijving

Ontwerp een programma voor de simulatie van het beheer van rekeningen door een bank.
Deze bank kent twee soorten rekeningen, te weten betaalrekeningen en spaarrekeningen. Bij iedere rekening moet de rekeninghouder en een uniek rekeningnummer worden bijgehouden. Van de rekeninghouder moet in elk geval de naam bekend zijn. Van een rekening dient het saldo opgevraagd te kunnen worden.
Een betaalrekening wordt gebruikt voor het dagelijkse betalingsverkeer. Er kan contant geld op de betaalrekening worden gestort en er kan contant geld van worden opgenomen. Bovendien kan geld worden overgemaakt van een betaalrekening naar een andere rekening (betaalrekening of spaarrekening). Opname en overmaken kunnen alleen worden uitgevoerd onder de voorwaarde dat het saldo op de betaalrekening niet negatief wordt. Er wordt geen rente betaald over de tegoeden op de betaalrekening.
Ook op een spaarrekening kan contant geld worden gestort. Verder heeft de spaarrekening een aantal beperkingen ten opzichte van de betaalrekening. Het is niet mogelijk geld over te maken van een spaarrekening naar een willekeurig andere rekening. Bij opening van een spaarrekening dient een betaalrekening te worden opgegeven, de zogenaamde tegenrekening. Bij opname van geld van de spaarrekening wordt het gevraagde bedrag niet contant uitgekeerd, maar gestort op deze tegenrekening. Per kalenderjaar mag er slechts € 10.000,– worden opgenomen. Over hogere opnamen moet een boete van 3 % betaald worden. Ook bij opname van een spaarrekening geldt dat het saldo niet negatief mag worden. Over het saldo op een spaarrekening wordt per jaar 5 % rente betaald. De rente wordt per maand berekend, op grond van het saldo op de eerste van die maand, maar slechts eenmaal per jaar bijgeschreven.

De gebruiker van de simulatie moet de bank opdracht kunnen geven geld te storten en op te nemen van rekeningen en geld over te maken van de ene naar de andere rekening (alles voor zover het soort rekening dat toestaat). Ook moet saldo-informatie over een rekening opgevraagd kunnen worden.
De simulatie houdt ook een datum bij. Bij de start van de simulatie is deze gelijk aan de echte huidige datum. De gebruiker moet de datum kunnen verzetten (altijd vooruit, dus naar de toekomst) zodat er acties op verschillende data kunnen worden uitgevoerd. Steeds als de datum wordt verzet, moet het systeem de rente berekenen die de spaarrekeningen hebben opgeleverd in de tussenliggende tijd. Als de nieuwe datum in een nieuw jaar ligt, moet de rente over het afgelopen jaar (of de afgelopen jaren) ook daadwerkelijk worden bijgeschreven.
Alle gebruikersacties worden mogelijk gemaakt via een grafische user-interface.

We gaan nu eerst een lijstje opstellen van de gebruiksmogelijkheden van de simulatie. Wat moet de gebruiker er zoal mee kunnen doen?

OPGAVE 1.10
Probeer, op grond van de productomschrijving, een lijst met gebruiksmogelijkheden op te stellen. Iedere gebruiksmogelijkheid beschrijft een interactie van de gebruiker met het systeem. Om u op weg te helpen, geven we hier twee voorbeelden:
– Maak een nieuwe betaalrekening aan.
– Geef het saldo van de rekening met gegeven nummer.

3.2 OPSTELLEN DOMEINMODEL

De volgende stap is nu, om op grond van de productomschrijving domeinklassen te kiezen die zeker nodig zijn. Iedere productomschrijving zal in elk geval een paar van deze klassen opleveren.

Welke domeinklassen kunt u, op grond van de productomschrijving, onmiddellijk benoemen?

In dit geval zijn dat er minimaal drie, namelijk Bank, Betaalrekening en Spaarrekening. U kunt ook Rekeninghouder als klasse beschouwen. De productomschrijving is vaag over wat er van een rekeninghouder moet worden bijgehouden (in ieder geval de naam). In een dergelijk geval is het handig om zo’n klasse dan toch op te nemen in het klassendiagram. Later, tijdens het verdere ontwerp, kan alsnog besloten worden dat deze klasse overbodig is. Misschien ziet u nog meer klassen, maar we zullen het hier voorlopig bij laten. We zullen zien dat er andere klassen naar voren komen door de eisen aan de software goed in de gaten te houden. Klassen die betrekking hebben op de gebruikersinterface worden op dit moment van het ontwerp nog niet beschouwd. We zijn namelijk alleen de domeinlaag aan het ontwerpen.

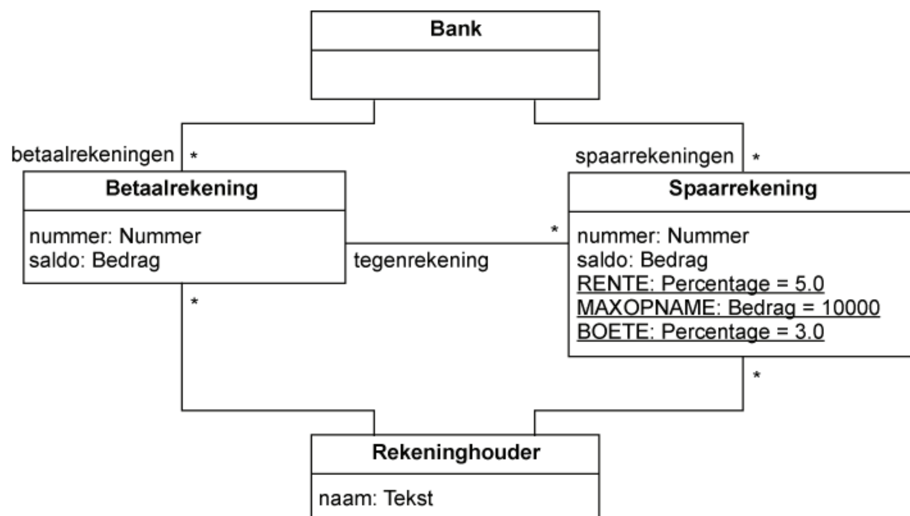
Vervolgens moet er worden geformuleerd welke globale verantwoordelijkheid iedere klasse heeft. We doen dat in de vorm van een kort zinnetje, dat later als commentaar in de kop van de klassendefinitie opgenomen kan worden.

OPGAVE 1.11
Formuleer de verantwoordelijkheid van elk van de vier tot nu toe benoemde klassen.

Als volgende stap kijken we of er attributen zijn die we op grond van de productomschrijving *zeker* nodig zullen hebben.

Welke attributen en associaties moeten de benoemde klassen *in elk geval* hebben?

Als we om te beginnen alleen de attributen en associaties opnemen waar we in geen geval buiten kunnen, dan komen we uit op figuur 1.13. Volgens de productomschrijving moet bij iedere rekening een uniek rekeningnummer worden bijgehouden. Ook wordt expliciet het saldo van de rekeningen genoemd en dus zullen we ook dat meteen als attribuut benoemen. Van een rekeninghouder moeten we in ieder geval de naam opslaan. Verder zien we bij de spaarrekening een aantal eigenschappen staan (rentepercentage, maximum op te nemen bedrag, en boetepercentage). De klasse Spaarrekening kan deze eigenschappen vastleggen in constanten; dit zijn ook attributen.
Relaties tussen klassen worden als associaties in een klassendiagram opgenomen, dus niet als attributen. De bank beheert verschillende betaalrekeningen en verschillende spaarrekeningen. Dit komt tot uiting in een associatie van Bank naar Betaalrekening en een associatie van Bank naar Spaarrekening, beide met multipliciteit * aan de kant van de rekeningklasse om aan te geven dat de bank 0 of meer rekeningen kan beheren. Verder behoort bij iedere betaalrekening en bij iedere spaarrekening een rekeninghouder, aangegeven door een associatie van Betaalrekening naar Rekeninghouder en een associatie van Spaarrekening naar Rekeninghouder. Bij iedere spaarrekening hoort een betaalrekening als tegenrekening. Deze relatie wordt getoond door een associatie van Spaarrekening naar Betaalrekening. De betaalrekening heeft in deze associatie de rol tegenrekening. Let ook op de multipliciteit bij deze associatie aan de kant van Spaarrekening. Deze is * omdat, hoewel bij iedere spaarrekening precies één betaalrekening hoort, een betaalrekening kan dienen als tegenrekening voor nul of meer spaarrekeningen. Waar geen multipliciteitsaanduiding staat, heeft deze de standaardwaarde 1 (elke betaal- of spaarrekening hoort bij één bank, elke rekening heeft één rekeninghouder en elke spaarrekening heeft één tegenrekening).

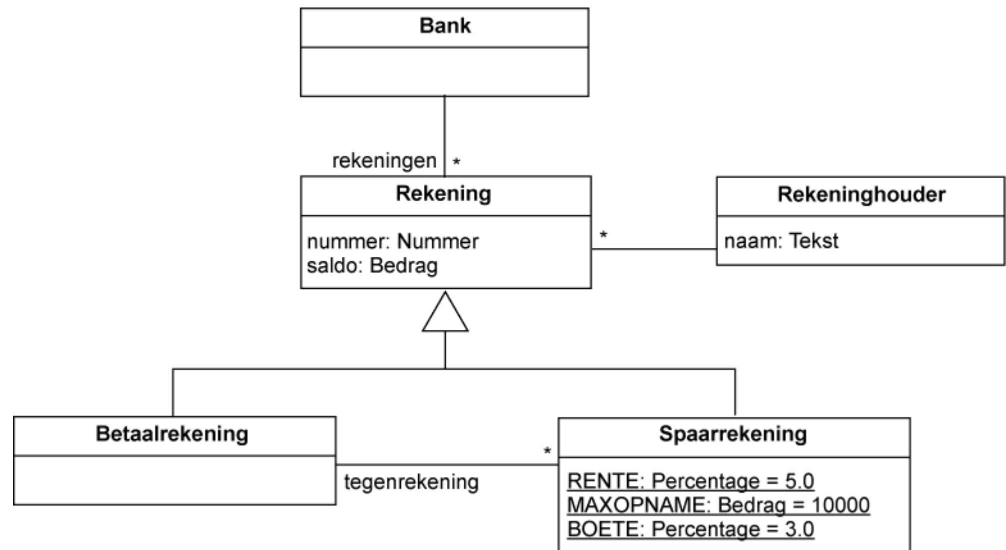


FIGUUR 1.13 Klassendiagram voor de bank, versie 1

Merk op dat tijdens het eerste ontwerp nog niet gekozen is voor een bepaalde representatie van de attributen. We gebruiken daarom algemene aanduidingen als Tekst, Nummer, Bedrag en Percentage.

Tegen welk eerder genoemd principe van een goed programma zondigt dit ontwerp en wat zou daar aan te doen zijn?

Dit ontwerp zondigt tegen het principe van lokaliteit. Het ontwerp van de klassen Betaalrekening en Spaarrekening is voor een groot deel gelijk. Keuzen voor een representatie van een nummer en een bedrag zullen in beide klassen zichtbaar zijn en als we een dergelijke doublure kunnen vermijden, dan moeten we dat doen. Bovendien delen beide klassen de associatie met Rekeninghouder.
In dit geval kan dat eenvoudig worden verholpen door de overeenkomstige delen van beide klassen in een superklasse Rekening onder te brengen. Uiteraard handhaven we Betaalrekening en Spaarrekening als subklassen. Uit de productomschrijving volgt immers dat deze wel degelijk zullen verschillen. Figuur 1.14 toont het nieuwe ontwerp.



FIGUUR 1.14 Klassendiagram voor de bank, versie 2

Opmerking

Misschien vraagt u zich af of in het uiteindelijke programma de betaalrekeningen en de spaarrekeningen niet gescheiden moeten blijven, zodat de klasse Bank in de implementatie uiteindelijke twee attributen zal krijgen: betaalrekeningen en spaarrekeningen. Dit is niet handig en we zullen het dan ook niet doen. Waarom dat zo is, zult u pas na de volgende leereenheid kunnen begrijpen.

3.3 OPSTELLEN ONTWERPMODEL

Het klassendiagram van figuur 1.14 gaf het domeinmodel. De volgende stap is het ontwikkelen van het ontwerpmodel. In dit model worden methoden toegevoegd aan de klassen en krijgen associaties een richting. Ook kan tijdens het verdere ontwerp blijken dat er nog meer attributen nodig zijn.

We gaan nu alle gebruiksmogelijkheden van de bank na. Ieder van die mogelijkheden zal leiden tot een methode in de klasse Bank. Als we namelijk iets met een rekening willen doen, dan geven we daar een opdracht voor aan de bank en niet direct aan een rekening, net als in het echte betalingsverkeer. Een gebruiker communiceert met de bank alleen via rekeningnummers en niet via instanties van rekeningen. De bank is verantwoordelijk voor het juist toepassen van de regels voor het betalingsverkeer en het beheren van de rekeningen. Het is niet gewenst dat gebruikers direct operaties op rekeningen uitvoeren, omdat de bank dan niet kan controleren of dat wel volgens de regels gebeurt.

Figuur 1.15 toont de klasse Bank met voor iedere gebruiksmogelijkheid een methode. Alleen het verzetten van de datum is nog niet opgenomen. Alle rekeningen worden door de gebruiker aangeduid door hun rekeningnummer; het is de verantwoordelijkheid van de bank om uit te zoeken welk type rekening bij een rekeningnummer hoort, en om de juiste acties daarmee uit te voeren.

Bank
maakBetaalrekening(naam: Tekst): Nummer maakSpaarrekening(naam: Tekst, tegen: Nummer): Nummer stort(nummer: Nummer, bedrag: Bedrag) neemOp(nummer: Nummer, bedrag: Bedrag) maakOver(van: Nummer, naar: Nummer, bedrag: Bedrag) geefSaldo(nummer: Nummer): Bedrag

FIGUUR 1.15 Klasse Bank met zijn methoden

Opmerking

Bij het ontwerp gaan we uit van de situatie waarbij geen fouten optreden. We negeren dus alle mogelijk uitzonderingsgevallen.

We gaan nu verder als volgt te werk.
– Voor elke gebruiksmogelijkheid (in dit geval dus voor elke methode van Bank) gaan we na welke acties er voor nodig zijn, met de nadruk op creatie van objecten en noodzakelijke interactie met andere objecten.
– Door deze acties nader te analyseren, kunnen we achterhalen welke constructoren en methoden de klassen uit figuur 1.14 nodig hebben.

maakBetaal-rekening

De methode maakBetaalrekening wordt aangeroepen als de gebruiker aangeeft een nieuwe betaalrekening te willen openen. Daarvoor dient de gebruiker zijn naam op te geven. Het resultaat van deze methode is dat de bank beschikking heeft gekregen over een nieuwe betaalrekening. De methode geeft het rekeningnummer van de nieuwe rekening terug aan de gebruiker, zodat deze in de toekomst dit nummer kan gebruiken voor zijn bankhandelingen.

OPGAVE 1.12

Som de acties op die in de methode maakBetaalrekening moeten worden uitgevoerd. Om u op weg te helpen, geven we vast de eerste twee:
– genereer een nog ongebruikt rekeningnummer
– creëer een instantie van Betaalrekening.

We gaan nu na wat deze acties betekenen voor het ontwerp.

Een betaalrekening bevat een nummer en een rekeninghouder. Deze informatie moet bij de creatie van een Betaalrekening-object via parameters meegegeven worden. Voor de rekeninghouder zijn er daarbij twee mogelijkheden:
– Als parameter wordt de naam van de rekeninghouder meegegeven. In dit geval dient de constructor van Betaalrekening een instantie van Rekeninghouder te maken.
– Als parameter wordt een instantie van Rekeninghouder meegegeven. In dit geval dient de Bank eerst een instantie van Rekeninghouder te maken.

We kiezen voor de tweede optie. Deze heeft als voordeel dat het dan ook mogelijk is voor de bank om eerst te zoeken of een rekeninghouder al bestaat en zo ja, de betreffende instantie als parameter aan de constructor mee te geven. Dit verhoogt de uitbreidbaarheid van het systeem (we laten het zoeken of de rekeninghouder al bestaat in deze applicatie echter verder buiten beschouwing).

We krijgen dan een extra stap in de lijst met acties:
– Creëer een instantie van Rekeninghouder.
– Genereer een nieuw rekeningnummer.
– Creëer een instantie van Betaalrekening.
– Voeg deze instantie toe aan de lijst van rekeningen.
– Geef het rekeningnummer terug.

Dit leidt tot de volgende constructoren in de klassen Rekeninghouder respectievelijk Betaalrekening

```
Rekeninghouder(naam: Tekst)
Betaalrekening(rekeninghouder: Rekeninghouder, nummer: Nummer)
```

Misschien denkt u ook al aan een methode om een ongebruikt rekening-nummer te genereren. Dat hoort eigenlijk nog niet in deze fase van het ontwerp. Het is een implementatiebeslissing om hiervoor een hulp-methode te maken. Hulpmethoden (die private horen te zijn) worden pas tijdens de implementatie ontworpen.

Deze constructoren worden opgenomen in het klassendiagram (zie figuur 1.16)

maakSpaarrekening

De methode maakSpaarrekening wordt op een gelijke wijze verwerkt in het ontwerp.

OPGAVE 1.13
Som de acties op die in de methode maakSpaarrekening moeten worden uitgevoerd. Tot welke constructor(en) en methode(n) in andere klassen zal dit leiden?

stort

De volgende methode is stort. De gebruiker stort daarmee geld op een rekening. De bank dient daarvoor achtereenvolgens de volgende stappen uit te voeren:
– Zoek de rekeninginstantie horend bij het rekeningnummer.
– Stort het bedrag op de gevonden rekening.

Om dit mogelijk te maken wordt de klasse Rekening uitgebreid met een methode stort(bedrag: Bedrag) waarmee het saldo van een rekening wordt verhoogd (zie figuur 1.16). Omdat de subklassen deze methoden erven, kan daarmee zowel op een betaal- als op een spaarrekening geld worden gestort.

neemOp

Met de methode neemOp neemt de gebruiker geld op van zijn rekening. De acties die de bank daarvoor moet uitvoeren zijn:
– Zoek de rekeninginstantie horend bij het rekeningnummer.
– Neem het bedrag op van de gevonden rekening.
Opnemen kan zowel van een betaalrekening als van een spaarrekening. Wat er precies bij het opnemen gebeurt, is echter afhankelijk van het type rekening. Beide typen rekening dienen daarom een eigen methode te krijgen, namelijk neemOp(bedrag: Bedrag) waarmee geld van de rekening wordt opgenomen (zie figuur 1.16). De implementatie van deze methoden zal verschillen.

maakOver

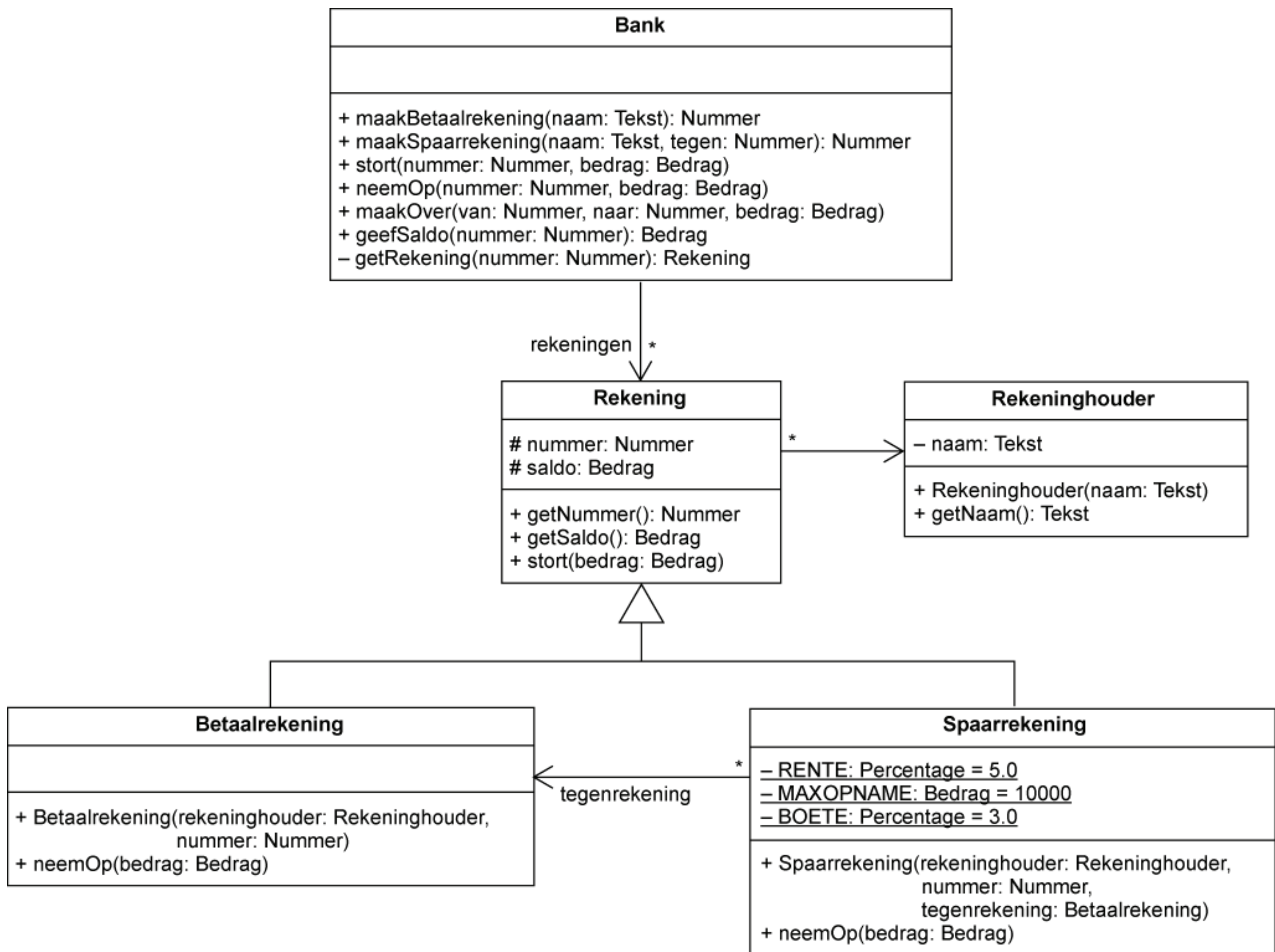
De methode maakOver is verantwoordelijk voor het overmaken van geld van een betaalrekening naar een andere rekening. De acties die de bank moet uitvoeren zijn:
– Zoek de rekeninginstanties bij de opgegeven rekeningnummers. De instantie van de rekening waarvandaan wordt overgemaakt moet een betaalrekening zijn.
– Neem het bedrag op van de ene rekening.
– Stort het bedrag op de andere rekening.
De methoden die we hiervoor nodig hebben, neemOp bij Betaalrekening en stort bij Rekening, zijn beiden al beschikbaar.

geefSaldo

De methode geefSaldo van de bank vraagt het saldo op van een rekening, zowel voor een betaal- als een spaarrekening. De acties van de bank daarvoor zijn:
– Zoek de rekeninginstantie bij het opgegeven rekeningnummer.
– Vraag aan de rekening zijn saldo.
– Geef het saldo terug.
Omdat de methode om het saldo op te vragen voor de betaal- en spaarrekening identiek zijn, en de benodigde informatie in de superklasse Rekening aanwezig is, kan de klasse Rekening worden uitgebreid met een methode getSaldo die het saldo teruggeeft.

Figuur 1.16 toont een nieuwe versie van het klassendiagram, waarin alle genoemde constructoren en methoden opgenomen zijn. Hierin zijn naast de besproken constructoren en methoden, bij enkele klassen ook get-methoden toegevoegd. Het is namelijk gebruikelijk om bij een attribuut een get-methode te maken waarmee de waarde van het attribuut opgevraagd kan worden (tenzij dat om een of andere reden onwenselijk is; maar daar is hier geen sprake van).

Verder ziet u dat bij alle attributen en methoden toegangsspecificaties zijn toegevoegd. We willen normaal gesproken attributen zoveel mogelijk private maken om te voorkomen dat er van buiten af ongewenste operaties op kunnen worden uitgevoerd. De attributen nummer en saldo van Rekening zijn protected gemaakt om ervoor te zorgen dat de subklassen Betaalrekening en Spaarrekening deze attributen wel direct kunnen gebruiken.



FIGUUR 1.16 Klassendiagram voor de bank, versie 3

We moeten nog een gebruiksmogelijkheid bekijken, namelijk het verzetten van de datum.

Wat moet er allemaal gebeuren als de gebruiker de datum verandert? Kijk naar de productomschrijving!

De gebruiker kan steeds, na een aantal acties, de datum vooruit zetten. Als de nieuwe datum in een nieuwe maand ligt, moet voor alle spaarrekeningen de opgebouwde rente over de tussenliggende maanden berekend en onthouden worden. Ligt de datum in een nieuw jaar, dan moet de rente over het afgelopen jaar (of over de afgelopen jaren) ook worden bijgeschreven. Bovendien moet dan het bedrag dat in het ‘huidige’ jaar is opgenomen, weer op nul worden gezet. Er zijn immers in dat jaar nog geen transacties op de spaarrekening geweest.

Ook hier moeten we ons eerst afvragen, welke klasse voor welke van deze taken verantwoordelijkheid krijgt.

– Welke klasse beheert het tijdsverloop in de simulatie?
– Welke klasse coördineert de acties die voor de spaarrekeningen ondernomen moeten worden?
– Welke klasse voert die acties daadwerkelijk uit?

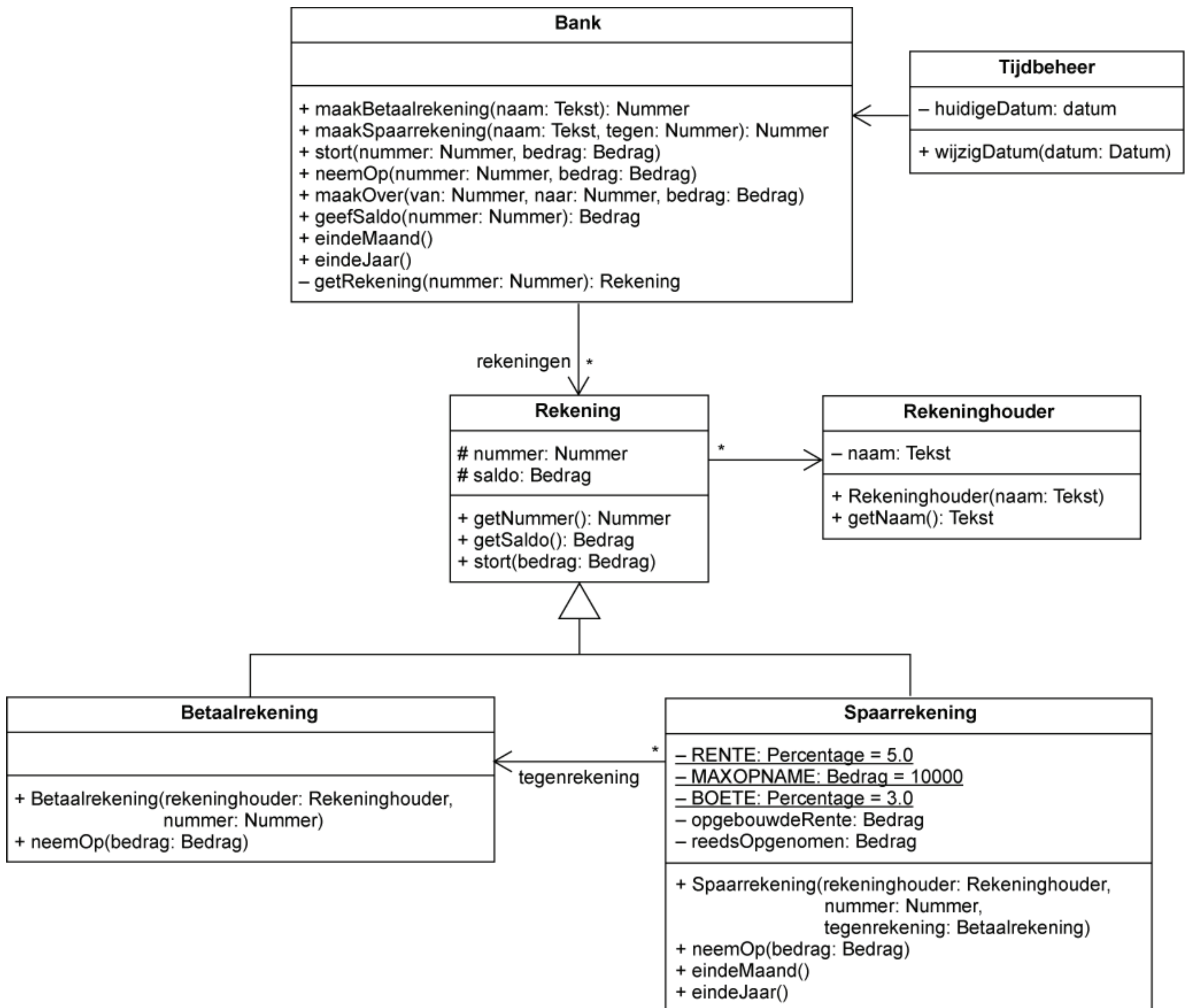
Het beheren van het tijdsverloop valt niet onder het beheer van de rekeningen en is dus geen taak voor de bank. We hebben dus behoefte aan een nieuwe klasse, die we Tijdbeheer noemen. Het coördineren van de acties voor de spaarrekeningen die uit het verlopen van de tijd voortkomen is wel een taak voor de klasse Bank. Het uitvoeren ervan hoort thuis bij de klasse Spaarrekening.

Probeer te bedenken hoe het proces, dat wordt gestart door een wijziging van de datum, moet verlopen.

De klasse Tijdbeheer bevat een attribuut dat de laatst opgegeven datum bijhoudt. Verder krijgt deze klasse een methode wijzigDatum die aangeroepen wordt met een nieuwe datum. Deze methode zet de klok maand voor maand vooruit, tot de nieuwe datum is aangebroken. Deze nieuwe datum wordt dan opgeslagen als huidige datum.
Aan het eind van iedere gesimuleerde maand wordt een methode eindeMaand aangeroepen op de bank, aan het eind van een gesimuleerd jaar wordt een methode eindeJaar aangeroepen. Omdat de datum willekeurig ver vooruit gezet kan worden, kunnen beide methoden verschillende keren worden aangeroepen.
De methode eindeMaand van Bank roept zelf dan weer voor alle spaarrekeningen hun methode eindeMaand aan. Deze methode berekent de rente over de zojuist verstreken maand. Bij het einde van een gesimuleerd jaar, gebeurt iets dergelijks: nu roept de bank voor alle spaarrekeningen de methode eindeJaar aan. Deze schrijft de rente over het afgelopen jaar bij.

We kunnen nu het klassendiagram gaan aanpassen. Om de beschreven procedure goed uit te kunnen voeren, heeft de klasse Spaarrekening een attribuut nodig voor de opgebouwde rente over het lopende jaar, die nog niet is bijgeschreven. Ook nemen we een attribuut op dat bijhoudt hoeveel geld er al is opgenomen in het huidige jaar; dit wordt in de methode eindeJaar weer op nul gezet. We hadden deze attributen ook al eerder kunnen introduceren omdat al uit de productomschrijving volgt dat ze nodig zullen zijn, maar pas bij het uitwerken van deze gebruiksmogelijkheid blijken ze ook echt relevant.

De wijzigingen in het ontwerp zijn getoond in figuur 1.17. Omdat een instantie van de klasse Tijdbeheer de bank opdracht geeft maanden en jaren af te sluiten, moet de klasse Tijdbeheer de klasse Bank kennen.



FIGUUR 1.17 Klassendiagram voor de bank, versie 4

OPGAVE 1.14

Een alternatief ontwerp voor het verwerken van het verzetten van de datum is als volgt. Tijdbeheer roept een methode wijzigDatum aan op de bank met de oude en nieuwe datum als parameter. Een methode met dezelfde signatuur wordt vervolgens aangeroepen op alle spaarrekeningen. In de spaarrekening zelf wordt pas het verstrijken van de tijd gesimuleerd.
Welke van de twee ontwerpen (het oorspronkelijke of het alternatieve) verkiest u en waarom?

Als laatste moet een gebruikersinterface aan het systeem worden toegevoegd. De overeenkomstige klasse heeft een associatie met zowel Bank, om de functies van bank aan te roepen, als met Tijdbeheer, om de datum te kunnen verzetten. We tonen geen nieuw klassendiagram.

Het ontwerp van dit systeem is hiermee afgerond.

We houden ons in deze leereenheid niet bezig met het omzetten van dit ontwerp in een implementatie. Wel kunnen we kort de stappen noemen die nog gedaan moeten worden.
– De gebruikersinterface moet worden ontworpen.
– Van iedere klasse moet de interface gespecificeerd worden: constructoren, publieke attributen en methoden. Van iedere methode wordt de signatuur gegeven en wordt omschreven wat de methode doet en/of welke waarde deze berekent.
Merk op dat hetgeen in figuur 1.17 getoond is, *niet* die interface is. Figuur 1.17 bevat immers ook de private attributen (en één private methode) die in een ontwerp onontbeerlijk zijn maar in een interface-specificatie van een implementatie niet thuishoren. Bovendien ontbreken in figuur 1.17 de omschrijvingen van de methoden. Een programmeur moet zowel over het klassendiagram als over de interfacespecificatie kunnen beschikken.
– Dan moet iedere klasse gecodeerd en getest worden. Het kan handig zijn om daarbij voor sommige klassen een aparte testomgeving te maken.

Een implementatie van dit ontwerp vindt u bij de bouwstenen in het project Le01Bank.

OPDRACHT 1.15
a Open het project Le01Bank.
Start de applicatie (methode main staat in de klasse BankFrame). Open enige rekeningen en voer wat transacties uit. Verzet de datum naar een volgend jaar en bekijk het saldo op de spaarrekeningen.
In welk opzichten schiet deze applicatie ernstig tekort?
b Bekijk de attributen en methoden van alle klassen uit het programma. Zijn er verschillen met het ontwerp?

4 Ontwerpen stap voor stap

Zie Objectgeoriënteerd programmeren in Java 1, leereenheid 6

In de cursus Objectgeoriënteerd programmeren in Java 1 hebben we een ontwerpmethode gepresenteerd die bestond uit de volgende zeven stappen:

1 productomschrijving
2 ontwerp van domeinklassen
3 implementatie van domeinklassen
4 testen van domeinklassen
5 ontwerp en specificatie van de gebruikersinterface
6 implementatie en testen van de gebruikersinterface
7 evaluatie

Splitsing in deeltaken

Deze methode voldeed voor kleine applicaties. Voor deze cursus brengen we er wat wijzigingen in aan. Zodra we iets grotere systemen gaan ontwerpen, is het soms handiger om de taak van het systeem op een andere manier *in deeltaken te verdelen*. In paragraaf 3.1 deden we dit door een lijst met gebruiksmogelijkheden op te stellen. Iedere gebruiksmogelijkheid komt dan overeen met een deeltaak van het systeem. Dit is vaak een goede strategie, maar niet altijd.

Denk bijvoorbeeld aan een productomschrijving die ons opdraagt een programma te maken waarmee de gebruiker tegen de computer kan schaken. De gebruiksmogelijkheden beperken zich dan mogelijk tot Nieuw spel en Doe een zet, hetgeen ons nauwelijks helpt om het probleem in deelproblemen te splitsen. In zulke gevallen moet dus een andere splitsing worden opgesteld.

Iteratief of cyclisch ontwikkelproces

De deeltaken worden vervolgens gebruikt om in opeenvolgende *iteraties* het programma te ontwerpen. Iedere iteratie wordt afgesloten met een implementatie van de betreffende deeltaak en de daaropvolgende iteratie bouwt weer voort op deze implementatie. Zo wordt het programma steeds verder uitgebreid.
Een dergelijke aanpak noemen we een *iteratief* of *cyclisch ontwikkelproces*. Als alle deeltaken op deze manier zijn ontworpen en geïmplementeerd, is het programma in principe af.

Objectkeuze

Bij het ontwerpen kunnen we, net als in het bankvoorbeeld, beginnen met de meest voor de hand liggende klassen en attributen van het totale systeem. Deze volgen direct uit de productomschrijving.

Per deeltaak een iteratie

Vervolgens kiezen we de deeltaak die we als eerste gaan aanpakken en bekijken in deze eerste iteratie welke klassen bij het uitvoeren van deze taak betrokken zijn. Hierbij zijn meestal ook nieuwe klassen nodig. Bij het bepalen daarvan dient het principe van gescheiden verantwoordelijkheden een belangrijke rol te spelen.
Vervolgens wordt vastgesteld hoe de interactie tussen de objecten verloopt voor de deeltaak onder beschouwing. Eventueel kan een sequentiediagram hierbij goede diensten bewijzen. Hierna kunnen we de (public) methoden specificeren en het klassendiagram aanpassen.

Documentatie

Bij een ontwerp hoort documentatie. Deze moet minimaal van iedere klasse aangeven welke verantwoordelijkheid deze heeft en bij elke methode specificeren welke actie deze uitvoert en/of welke waarde deze oplevert.

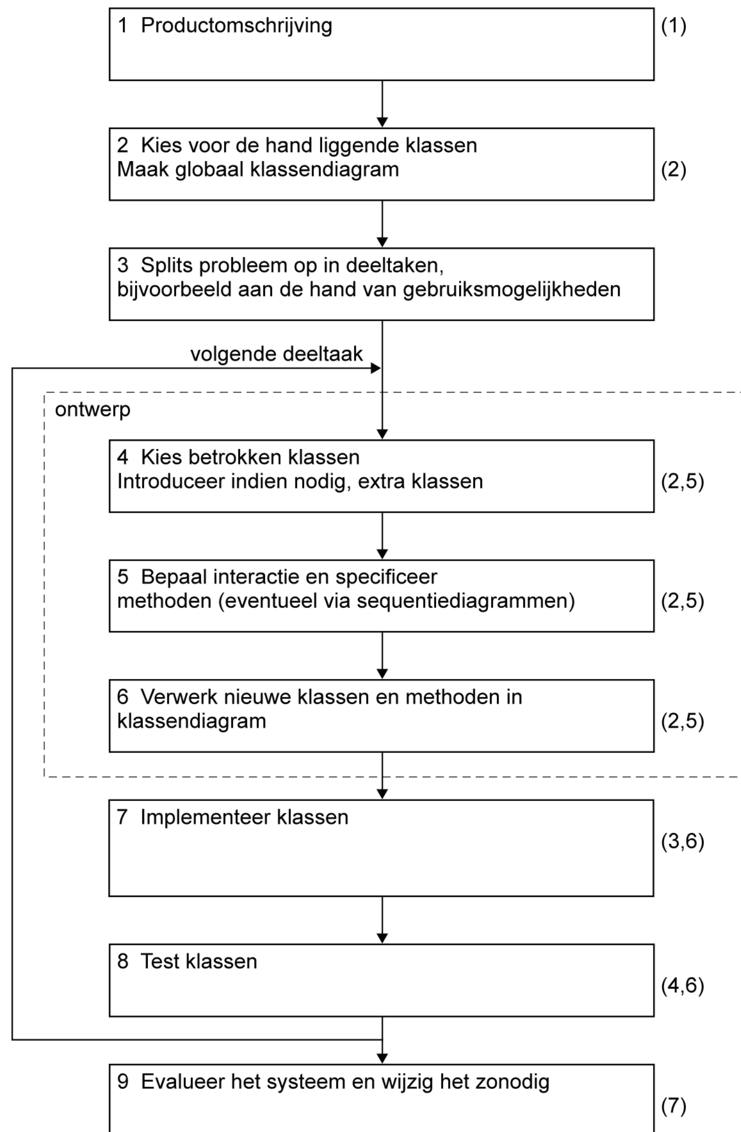
Implementatie

Vervolgens wordt het deelontwerp geïmplementeerd. De klassen, attributen en methoden die in de huidige iteratie naar voren zijn gekomen, worden geïmplementeerd en getest. Als de eerste versie van het programma geen fouten meer lijkt te bevatten, gaan we in de volgende iteratie een volgende deeltaak ontwerpen.

Evaluatie

Als alle iteraties zijn doorlopen, is het programma in principe af en moet het eindproduct worden geëvalueerd. Op grond van de evaluatieresultaten kunnen nog wijzigingen in het programma nodig zijn.

Figuur 1.18 geeft een overzicht van het gehele ontwikkelproces.



FIGUUR 1.18 Overzicht van het ontwikkelproces

Naast iedere stap staan tussen haakjes de overeenkomstige stappen uit de methode die we volgden in Objectgeoriënteerd programmeren in Java 1. In vergelijking met die methode zijn er de volgende wijzigingen:
– Het ontwerp wordt in verschillende stappen gedaan. Eerst wordt een globaal klassendiagram gemaakt waarin de meest voor de handliggende klassen met hun attributen en associaties worden weergegeven. Daarna wordt per deeltaak dit model uitgebreid en gewijzigd met methoden en eventueel nieuwe klassen.

– We maken gebruik van deeltaken. De stappen van objectkeuze tot en met implementatie worden nu in een iteratief proces voor iedere deeltaak herhaald. Dit is nodig omdat de programma's die we in deze cursus maken, iets complexer zijn dan die uit Objectgeoriënteerd programmeren in Java 1.
– Er wordt geen onderscheid meer gemaakt tussen de domeinklassen en de interfaceklassen. We zullen ons in deze cursus namelijk niet langer beperken tot programma's waarvan de gebruikersinterface uit slechts één venster (frame) bestaat. Het onderscheid is daarom niet meer zinvol. In het algemeen worden het ontwerp en de implementatie van de gebruikersinterface uitgevoerd als één (of meer, bij complexe interfaces) van de laatste deeltaken. Het is ook mogelijk om (delen van) de gebruikersinterface sneller te ontwikkelen om in een eerder stadium van het gehele ontwikkeltraject al de beschikking te hebben over een deelsysteem.

SAMENVATTING

Paragraaf 1

Wat is een goed programma? Deze vraag kan eigenlijk alleen zinvol beantwoord worden als we niet alleen kijken naar kleine programma's maar ook naar grote. Grote programma's worden ontwikkeld in een project dat typisch een voortraject, een analysefase, een constructiefase en een afrondingsfase kent.
– Tijdens het voortraject wordt een probleem gesignaleerd en ontstaat het plan een informatiesysteem te ontwikkelen om dat probleem op te lossen.
– Tijdens de analysefase wordt vastgesteld wat dat systeem precies moet doen en wordt een projectplan gemaakt.
– De constructie bestaat uit een aantal deelstappen: in elke stap kan bijvoorbeeld een deel van de gebruiksmogelijkheden van het systeem gerealiseerd worden.
– Iedere constructiestap omvat het ontwerp, de implementatie en het testen van de toevoegingen aan het systeem.
– In de afrondingsfase wordt het systeem als geheel door de gebruikers getest.
Als het systeem voltooid is, gaat het de onderhoudsfase in. Grote systemen gaan vaak jaren mee en behoeven voortdurende aanpassingen.
Een goed programma is correct (het voldoet aan de specificatie), robuust (het is bestand tegen gebruikersfouten en onverwachte situaties), het heeft begrijpelijke code die makkelijk te wijzigen en uit te breiden is en het springt efficiënt om met processortijd en geheugenruimte. Tot slot is het wenselijk om delen van de code te kunnen hergebruiken in andere toepassingen.

Bepaalde eigenschappen van de programmacode kunnen het onderhoud vergemakkelijken. Daartoe horen goed commentaar, goed gekozen namen en zo eenvoudig mogelijke onderdelen. Voor een objectgeoriënteerd programma zijn daarnaast vooral de volgende twee eigenschappen van belang.
– *Gescheiden verantwoordelijkheden*: elke klasse, methode en zelfs opdracht moet bij voorkeur één gemakkelijk te omschrijven verantwoordelijkheid in het geheel hebben.

– *Lokaliteit*: iedere verantwoordelijkheid van het systeem als geheel moet bij voorkeur op één plek gerealiseerd zijn, zodat wijziging van die verantwoordelijkheid op zo min mogelijk plaatsen in het programma tot wijziging van de code leidt. Ook kleine programma's willen we liefst aan deze eisen laten voldoen.

Paragraaf 2

Bij het ontwerpen van programma's maken we gebruik van enkele diagramtechnieken uit de unified modeling language (UML). Een klassendiagram toont de statische structuur van het programma: de klassen met hun attributen en methoden, en de relaties tussen de klassen. We bekeken twee soorten relaties:
– de overervingrelatie of (in UML-termen): generalisatie
– de associatie, waarbij instanties van de betrokken klassen elkaar kennen.
Bij associaties worden vaak rollen en multipliciteiten getoond.
Een objectdiagram toont de waarde van instanties op een gegeven moment en de links tussen deze instanties.

Paragraaf 3

In paragraaf 3 is als voorbeeld van een klein programma een eenvoudige banksimulatie ontworpen.

Paragraaf 4

Aan het eind van de leereenheid is getoond hoe men te werk kan gaan bij het realiseren van een dergelijk programma (zie figuur 1.18).

ZELFTOETS

1 a Noem vijf eisen waaraan een goed programma moet voldoen.
b Welke eigenschappen van de code zijn daarbij van belang?

2 Gegeven is de volgende productomschrijving van een snoepmachine.
Een snoepmachine bevat 12 dispensers (houders van artikelen) en een betaalmechanisme. Iedere dispenser bevat maximaal 20 artikelen met dezelfde prijs. Deze dispensers kunnen worden (bij)gevuld met nieuwe artikelen.
Een gebruiker kan het tegoed van de automaat verhogen. Het tegoed wordt bijgehouden in het betaalmechanisme. Dit betaalmechanisme kan geen geld teruggeven.
Een gebruiker van de automaat kan met behulp van het nummer van de dispenser het gewenste artikel kopen. Als het tegoed in het betaalmechanisme voldoende is, wordt het tegoed verlaagd en het artikel uitgeleverd.
a Welke klassen kunt u direct uit de productomschrijving halen. Teken een domeinmodel (zonder methoden).
b Geef de gebruiksmogelijkheden van het systeem.
c Stel een ontwerpmodel op.

TERUGKOPPELING

1 Uitwerking van de opgaven

1.1 a U heeft in Objectgeoriënteerd programmeren in Java 1 onder meer gebruik gemaakt van klassen uit de packages java.lang (Random, Math en natuurlijk ook String en de verpakkingsklassen), javax.swing (bijvoorbeeld JButton, JTextField en JLabel), java.awt (Color), java.util (ArrayList, GregorianCalendar) en java.text (DecimalFormat, SimpleDateFormat).
Daarnaast hebt u ook gebruik gemaakt van klassen die door het cursusteam waren geschreven, bijvoorbeeld de klassen uit de verkiezingenpackage.
b Om een klasse te kunnen gebruiken, moest u de interface van die klasse kennen: de publieke attributen (meestal constanten) en de methoden plus een beschrijving van hun betekenis of werking.
Deze werd beschreven in de specificatie. De beschrijving was soms niet volledig. Zo wordt binnen de API Specification bijvoorbeeld niet duidelijk uitgelegd welk coördinatenstelsel Swing gebruikt. Ook zult u vast wel eens de beschrijving van een methode verkeerd begrepen hebben, zodat deze heel iets anders bleek te doen dan u gedacht had. Eigenlijk werd u in zulke gevallen dus al geconfronteerd met een probleem dat hoort bij programmeren in het groot, namelijk de noodzaak voor een goede communicatie tussen de programmeurs van verschillende onderdelen.

1.2 Voorbeelden van wijzigingen zijn:
– Er komt een nieuw type container bij.
– Er wordt een nieuw tariefsysteem ingevoerd: er wordt bijvoorbeeld een nieuw type korting ingevoerd dat voorheen niet bestond.
– Er komt een nieuwe Java-versie, waardoor bepaalde functies met behulp van standaardklassen gerealiseerd kunnen worden. Men besluit het systeem daaraan aan te passen om zo de hoeveelheid intern te onderhouden code te verkleinen.
Een paar voorbeelden die nieuwe gebruiksmogelijkheden nodig of gewenst maken:
– Het bedrijf besluit niet alleen containers maar ook schepen te gaan verhuren.
– Een deel van het personeel van de verhuurder wordt ingezet bij het vervoer van de lege containers. Het bedrijf bedenkt dat het inroosteren van dat personeel ook best door het systeem ondersteund zou kunnen worden.

1.3 Twee zeer voor de hand liggende eigenschappen hebben we al in de inleiding genoemd: een programma wordt begrijpelijker als de programmeur betekenisvolle namen kiest en de programmacode voorziet van duidelijk verklarend commentaar. Verder is ook een standaardlay-out van het programma van belang (bijvoorbeeld de accolades altijd op dezelfde manier plaatsen en ieder blok twee spaties laten inspringen).

1.4 Definitie van constanten bevordert ook de begrijpelijkheid van de code en vermindert het risico van onopgemerkte tikfouten: het abusievelijk vervangen van 12 door 112 leidt niet tot een foutmelding, maar het abusievelijk vervangen van DOZIJN door DDOZIJN wel.

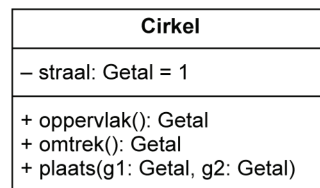
1.5 De volgende specificatie benadrukt dat dit een ontwerpmodel is en gebruikt daarom niet de typeaanduiding boolean en de standaardwaarde false:

– verkrijgbaar: Waarheidwaarde = onwaar

Een alternatief is

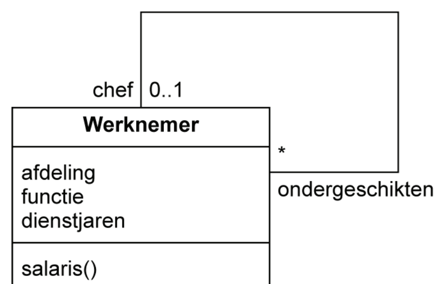
– verkrijgbaar: boolean = false

1.6 Figuur 1.19 toont de gevraagde klasse Cirkel.



FIGUUR 1.19 De klasse Cirkel

1.7 Het betreft een tweezijdige associatie, met als rolnamen bijvoorbeeld chef en ondergeschikten. Werknemers hebben in principe één chef, maar omdat de hiërarchie niet tot in het oneindige doorloopt, moet er ook minstens één werknemer zijn zonder chef. De multipliciteit van die rol is dus 0..1.
Een werknemer heeft 0 of meer ondergeschikten: de multipliciteit van die rol is dus *.
Figuur 1.20 toon het klassendiagram.



FIGUUR 1.20 De klasse Werknemer

1.8 Klasse A heeft een attribuut y, dat bijvoorbeeld van type B[] of van type ArrayList<B> is: er horen verschillende instanties van B bij iedere A. Klasse B heeft een attribuut x van type A: er hoort precies één instantie van A bij iedere B.

1.9 a Volgens het klassendiagram horen bij iedere klant (dus ook bij bedrijfsklanten en privé-klanten) nul of meer orders. In deze figuur zien we twee klanten: bij klant bk horen drie orders en bij klant pk hoort één order. Dat valt allebei onder nul of meer. Bovendien is de associatie tussen Klant en Order eenzijdig. Dat betekent dat alle links ook eenzijdig moeten zijn, en dat is ook zo.

b Een multipliciteitaanduiding bij de associatie tussen twee klassen geeft aan hoeveel links er tussen de bijbehorende instanties kunnen zijn. Een link tussen twee objecten is er of is er niet. Vanuit de implementatie bezien kunnen er meer links tussen dezelfde objecten zijn (object a heeft twee attributen die beide hetzelfde object b als waarde hebben), maar in UML geven we dat niet weer.

1.10 We hebben de volgende lijst gebruiksmogelijkheden:
– Maak een nieuwe betaalrekening.
– maak een nieuwe spaarrekening.
– Stort een bedrag op de rekening met gegeven nummer.
– Neem een bedrag op van de rekening met gegeven nummer .
– Maak een bedrag over van de rekening met gegeven nummer naar een andere rekening met gegeven nummer.
– Geef het saldo van de rekening met gegeven nummer.
– Wijzig datum (altijd naar later).

Bij de gebruiksmogelijkheden stort, neem op, maak over en geef saldo wordt geen onderscheid gemaakt tussen betaal- en spaarrekeningen. De gebruiker geeft alleen het nummer op van de rekening waarop de operatie moet worden uitgevoerd. Het is de verantwoordelijkheid van het systeem om te bepalen of de operatie op het desbetreffende type rekening is toegestaan en wat er dan precies moet gaan gebeuren. Het is echter niet fout als u bij het opsommen van de gebruiksmogelijkheden wel onderscheid heeft gemaakt tussen de rekeningtypen.

1.11 – De klasse Bank beheert alle rekeningen en regelt het gehele betalingsverkeer.
– De klasse Betaalrekening beheert het saldo van een rekeninghouder volgens de regels voor betaalrekeningen.
– De klasse Spaarrekening beheert het saldo van een rekeninghouder volgens de regels voor spaarrekeningen.
– De klasse Rekeninghouder beheert de gegevens van een rekeninghouder.

1.12 De volledige lijst van acties die de methode maakBetaalrekening moet uitvoeren, is:
– Genereer een nog ongebruikt rekeningnummer.
– Creëer een instantie van Betaalrekening.
– Voeg deze instantie toe aan de lijst van rekeningen.
– Geef het rekeningnummer terug.

1.13 De acties zijn voor een groot deel gelijk aan de acties voor het maken van een betaalrekening. Extra is het gebruik van de tegenrekening:
– Creëer een instantie van Rekeninghouder.
– Zoek de instantie van de tegenrekening aan de hand van het tegenrekeningnummer (dit moet een betaalrekening zijn).
– Genereer een rekeningnummer.
– Creëer een instantie van Spaarrekening.
– Voeg deze instantie toe aan de lijst van rekeningen.
– Geef het rekeningnummer terug.

Dit leidt tot de volgende constructor (in de klasse Spaarrekening):

```
Spaarrekening(rekeninghouder: Rekeninghouder,
              nummer: Nummer,
              tegenrekening: Betaalrekening)
```

In de omschrijving staat ook de stap om een rekeninginstantie te zoeken bij een rekeningnummer. We kunnen nu al voorzien dat ook andere operaties op de bank deze stap gaan gebruiken. De bank accepteert namelijk alleen rekeningnummers als invoer, maar om de operaties uit te voeren dienen methoden op rekeninginstanties te worden uitgevoerd. Het is de verantwoordelijkheid van de bank om bij een rekeningnummer de juiste rekening op te zoeken. Het is daarom zinvol om de klasse bank uit te breiden met de methode
getRekening(nummer: Nummer): Rekening
die gegeven een rekeningnummer de rekeninginstantie teruggeeft. Deze methode moet echter wel de toegangsspecificatie private krijgen. We moeten namelijk voorkomen dat gebruikers van de bank direct de beschikking krijgen over rekeninginstanties.
Het is niet fout als u deze methode nog niet opneemt in het ontwerp. Deze methode kan namelijk ook beschouwd worden als een implementatieoplossing.

1.14 In het alternatieve ontwerp simuleert de klasse Spaarrekening het verstrijken van de tijd. Dit is echter een taak die niet onder de verantwoordelijkheid van deze klasse hoort, maar die typisch thuishoort bij Tijdbeheer.
In dit ontwerp is daardoor de lokaliteit aangetast omdat een aspect van het systeem (het simuleren van het tijdsverloop) nu over meer klassen is verdeeld. Ook is de scheiding van verantwoordelijkheden aangetast omdat Bank en Spaarrekening nu meer doen dan alleen rekeningen beheren respectievelijk representeren.
Het oorspronkelijke ontwerp is daarom beter.

1.15 a De applicatie werkt volgens de productomschrijving. U zult echter merken dat het programma niet robuust is. Bij ongeldige invoer zal het programma exceptions opgooien. Later in de cursus zult u leren hoe deze afgevangen kunnen worden. Een andere ernstige tekortkoming is het feit dat alle rekeningen weg zijn als het programma gesloten wordt. Pas in leereenheid 9 en 10 leert u hoe dat voorkomen kan worden.
b Er zijn een aantal verschillen die illustratief zijn voor de overgang van een ontwerp naar een implementatie.
In het programma zijn representaties gekozen. Nummer is geïmplementeerd als int, Bedrag en Percentage zijn geïmplementeerd als double, Tekst als String en Datum als GregorianCalendar. Het attribuut rekeningen van de klasse Bank is geïmplementeerd met behulp van de API-klasse ArrayList, waarin gemakkelijk een lijst van objecten kan worden bijgehouden.
Verder heeft de methode neemOp in zowel Betaalrekening als Spaarrekening een boolean terugkeerwaarde gekregen om aan te geven of het opnemen geslaagd is. Bovendien heeft de klasse Rekening een methode neemOp, die vrijwel niets doet en die ook niet in het ontwerp voorkwam. In leereenheid 2 over overerving zullen we de noodzaak van deze methode duidelijk maken.

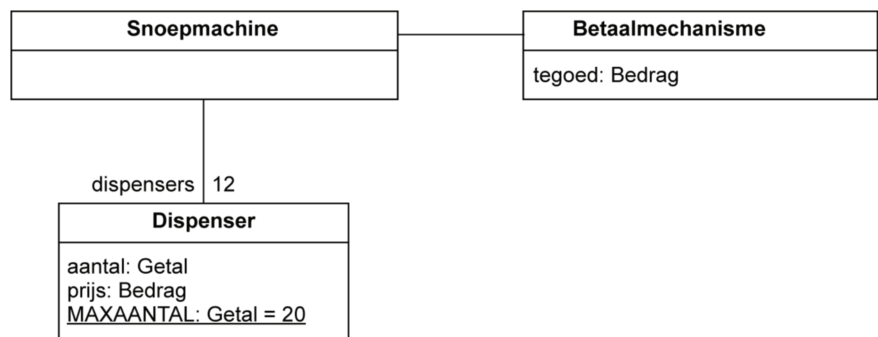
Van de klasse BankFrame was geen ontwerp gemaakt. Deze klasse heeft veel private methoden, zowel voor het opbouwen van de interface als voor event-handling. Deze methoden horen mogelijk in een implementatiemodel van de klassen thuis, maar zeker niet in het ontwerp.

2 Uitwerking van de zelftoets

1 a Een programma moet correct, robuust, makkelijk te wijzigen en makkelijk uit te breiden zijn. Verder is het gewenst dat onderdelen herbruikbaar zijn. Slechts in enkele gevallen is efficiëntie van doorslaggevend belang.

b De volgende eisen aan de code zijn genoemd.
– Een programma waarin de namen goed zijn gekozen en dat van geschikt commentaar is voorzien, is beter te begrijpen en dus makkelijker te wijzigen.
– Eenvoud van klassen en methoden (niet al te groot, geen diepe nesting) maakt code begrijpelijker en dus makkelijker te wijzigen.
– Ieder onderdeel van de code moet slechts één verantwoordelijkheid hebben (gescheiden verantwoordelijkheden).
– Iedere verantwoordelijkheid moet zoveel mogelijk op één plek in de code gerealiseerd zijn (lokaliteit).
– De onderlinge afhankelijkheid van klassen moet zo beperkt mogelijk zijn (lage koppeling).

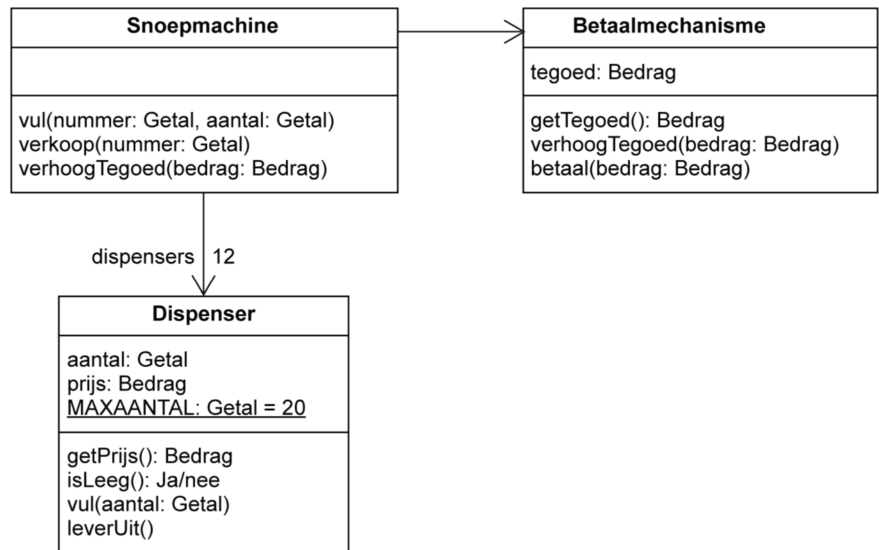
2 a Uit de productomschrijving volgen drie klassen: Snoepmachine, Dispenser en Betaalmechanisme. Het domeinmodel van de snoepmachine is gegeven in figuur 1.21.



FIGUUR 1.21 Klassendiagram snoepmachine, versie 1

b De gebruiksmogelijkheden zijn:
– vul de dispenser met gegeven nummer met een aantal artikelen
– koop een artikel uit dispenser met gegeven nummer
– verhoog het betaaltegoed.

c Een mogelijk ontwerpmodel is gegeven in figuur 1.22.



FIGUUR 1.22 Klassendiagram snoepmachine, versie 2

Uitleg:
Iedere gebruiksmogelijkheid leidt tot een methode in de snoepmachine.
– Met de methode vul wordt de dispenser met gegeven nummer gevuld. Deze methode zoekt de juiste dispenserinstantie en vult deze (methode vul in Dispenser).
– De methode verkoop wordt aangeroepen als een gebruiker een artikel uit de dispenser met gegeven nummer wil kopen. Deze methode zoekt de juiste dispenserinstantie en vraagt daaraan of deze nog artikelen bevat en wat de prijs van een artikel is (methoden isLeeg en getPrijs in Dispenser). Aan het betaalmechanisme wordt dan het tegoed gevraagd om te bepalen of er genoeg geld betaald is (methode getTegoed in Betaalmechanisme). Als hieruit blijkt dat de koop mogelijk is wordt het artikel door de dispenser uitgeleverd (methode leverUit in Dispenser) en wordt voor het artikel betaald (methode betaal in Betaalmechanisme).
– Met de methode verhoogTegoed wordt het betaaltegoed in de snoepmachine verhoogd. Omdat het Betaalmechanisme het tegoed bijhoudt moet de verhoging aan het Betaalmechanisme worden doorgegeven (methode verhoogTegoed in Betaalmechanisme).