Code Gorilla
Use Cases en functioneel ontwerp

Dagmar Hofman

Inhoudsopgave

1 Inleiding

Dit document gaat over **use cases** en **functioneel ontwerp**. Het doel is te *software ontwikkeling in het klein* te leren. Dit heeft de volgende kenmerken:

- Het wordt ontwikkeld door één programmeur
- Het kan worden ontwikkeld in een beperkte tijd (uren of dagen maar geen maanden of jaren)
- Er is meteen al een duidelijke productomschrijving waaruit nauwkeurig valt op te maken wat het programma doet
- Er is gen gebruikersgroep voor wie het programma per se onderhouden moet worden

2 Softwareontwikkeling

2.1 Voortraject

In de rest van deze paragraaf gebruiken we als voorbeeld van een groot programma regelmatig een informatiesysteem voor een bedrijf dat ver- schillende 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 vol- gende 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 program- meren 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.

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 verslechterden. 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.

2.2 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 pre- cieze 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. 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 sys- teem 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 con- tainers (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 gebruiksmoge- lijkheden 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 in- formatie centraal bij te houden en op een overzichtelijke wijze aan alle kantoren te leveren.

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

2.3 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 reke- ning gehouden met de functionaliteit die de software moet gaan bieden. Een domeinmodel is bovendien vaak niet volledig: allerlei details wor- den 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?

2.4 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 gereali- seerd. 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.

2.5 Ontwerp

Tijdens het ontwerp van een objectgeoriënteerd systeem wordt vast- gesteld uit welke klassen het programma zal gaan bestaan en hoe die klassen met elkaar samenhangen. Aan het eind van een ontwerpstap moet het volledig duidelijk zijn welke verantwoordelijkheden elke klasse in het programma tot dan toe zal hebben. Wat dan nog niet vastligt, is hoe elke klasse haar verantwoordelijkheden zal realiseren.

2.6 Ontwerpmodel

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 beheertaak hebben. Ook worden er soms andere klassen gekozen, omdat de klassen uit het domeinmodel tot een onhandige of een inefficiënte implementatie zouden leiden.
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.

2.7 Technisch ontwerp

Voor het softwareontwerp wordt daarom ook wel de term technisch ontwerp gebruikt. 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.

2.8 Implementatie

Als er een bevredigend ontwerp ligt, kan begonnen worden met de implementatie. 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 verschil- lende klassen werken.

2.9 Testen

Het testen van een bepaalde constructiestap verloopt ook weer in stappen. Elke klasse wordt eerst afzonderlijk getest. De programmeur zal daarvoor een kleine testomgeving construeren. In Objectgeoriënteerd program- meren 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 misver- standen 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 uitgepro- beerd. De kunst is nu dat deel zo te kiezen, dat het zo veel mogelijk representatief is voor alle invoer.

2.10 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 pak- ket 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 over- gedragen 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 (iteratief) doorlopen van de stappen analyse, ontwerp, implementatie, testen en afronding van de huidige fase. De opdracht- gever heeft dan al snel een klein werkend systeem ter beschikking. Na iedere volgende fase zullen de mogelijkheden zijn uitgebreid. Deze iteratieve aanpak is te beschouwen als een eenvoudige vorm van het Unified Process (UP). Kenmerkend voor het UP is het iteratief en incrementeel ontwerpen en implementeren van een systeem.

2.11 Prototype

Op deze wijze kan ook een prototype worden ontwikkeld: een experimen- tele versie van een te realiseren systeem. Meestal wordt een prototype gebruikt om inzicht te krijgen in de werking van een uiteindelijk pro- gramma. 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.

2.12 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 uit- breidingen van het programma nodig zijn, wat 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.

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.

3 Wat is een goed programma

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

3.1 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 honderd procent 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.

3.2 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 let- ters 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 onbe- doelde 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 omstandig- heden, zoals een harde schijf die vol is zodat er geen data kunnen worden weggeschreven.

3.3 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 oor- spronkelijk 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 commen- taar. In onze Java-cursussen gebruiken we daarvoor veelal javadoc, aangevuld met gewoon commentaar bij lastige stukken broncode.

3.4 Uitbreidbaarheid

Ook de structuur van het programma moet dusdanig zijn, dat het makke- lijk 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.

3.5 Herbruikbaarheid

Een gewenste eigenschap die objectoriëntatie populair heeft gemaakt, is herbruikbaarheid van delen van het systeem. Soms kan het veel ont- wikkeltijd 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

3.6 Efficiëntie

Tot slot is voor sommige programma's efficiëntie heel belangrijk. Dit geldt bijvoorbeeld voor:

- toepassingen waarbij het programmaverloop een proces in de buiten- wereld 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 bruikbaar en onbruikbaar 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 on- doorgrondelijk 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.

3.7 Kleine klassen en methoden

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.

3.8 Eenvoudige control flow

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. en klein maar vaak gezien voorbeeld van moeilijk te lezen code is het volgende:

```
if (!test) {
        return false;
} else {
        return true;
}
```

Dit kan korter en daarmee duidelijker opgeschreven worden:

```
return test;
```

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.

3.9 Lokaliteit

– 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. – 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 verantwoordelijk- heid 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. Gescheiden verantwoordelijkheden maken code gemakkelijker te begrij- pen 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 verschil- lende plaatsen wordt aangeroepen. Om die ene wijziging door te voeren, moeten we dan op tenminste achttien plaatsen in de code iets veran- deren: de drie methoden plus de vijftien aanroepen. Niet alleen moet elk onderdeel van de code slechts één verantwoorde- lijkheid 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.

3.10 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.

Ook onze voorkeur voor attributen die van buiten de klasse niet ge- wijzigd 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 alge- meen). 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.

3.11 Lege koppeling

Nog een manier die we hier willen noemen 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.

3.12 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 onder andere 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.