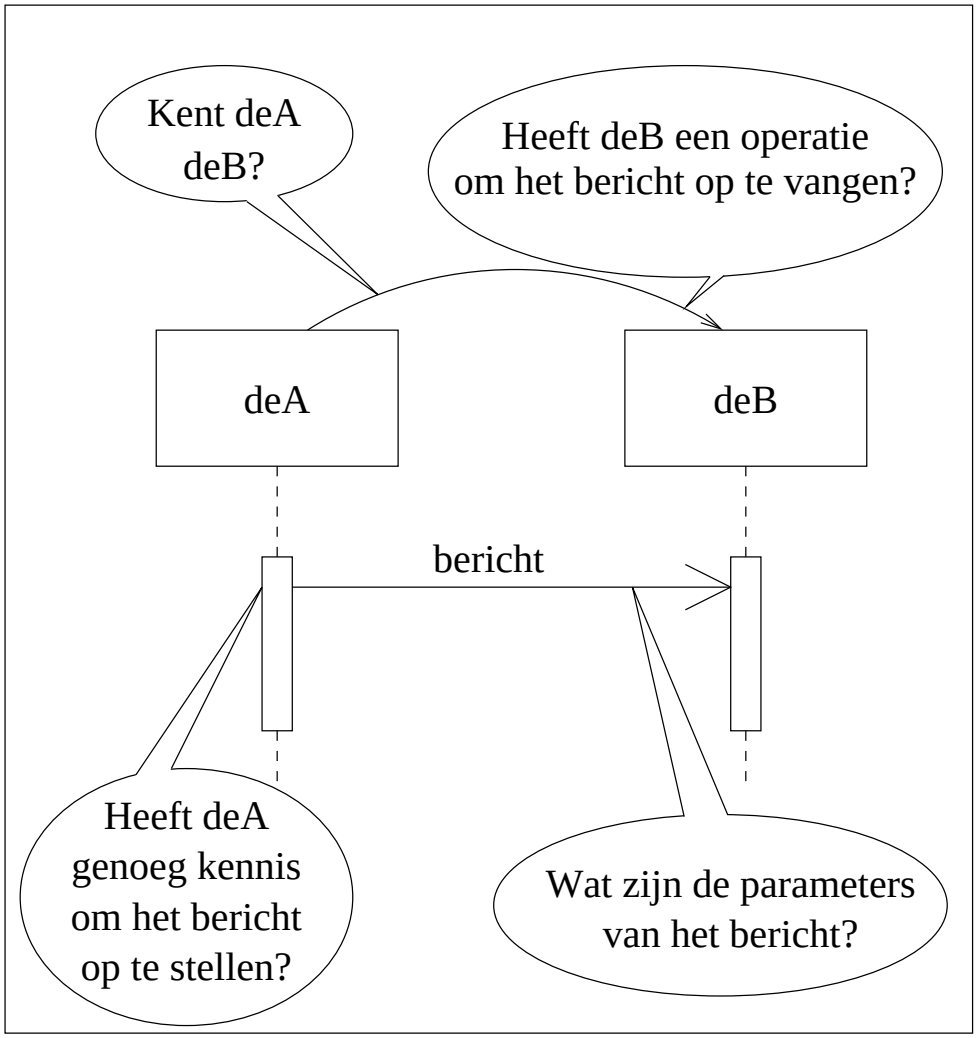
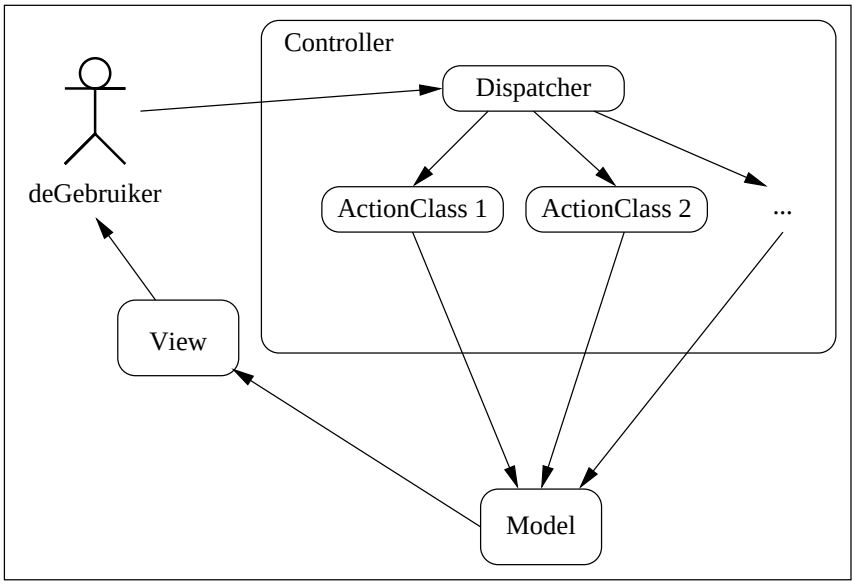


Figuur 1. De belangrijkste vragen bij het opmaken van een sequentiediagram

Bij het gebruik van het MVC-patroon (Model-View-Controller) dienen een aantal klassen te worden aangemaakt die niet in het model staan dat we hebben geconstrueerd. Deze klassen vallen onder de hoofding ‘technische klassen’.

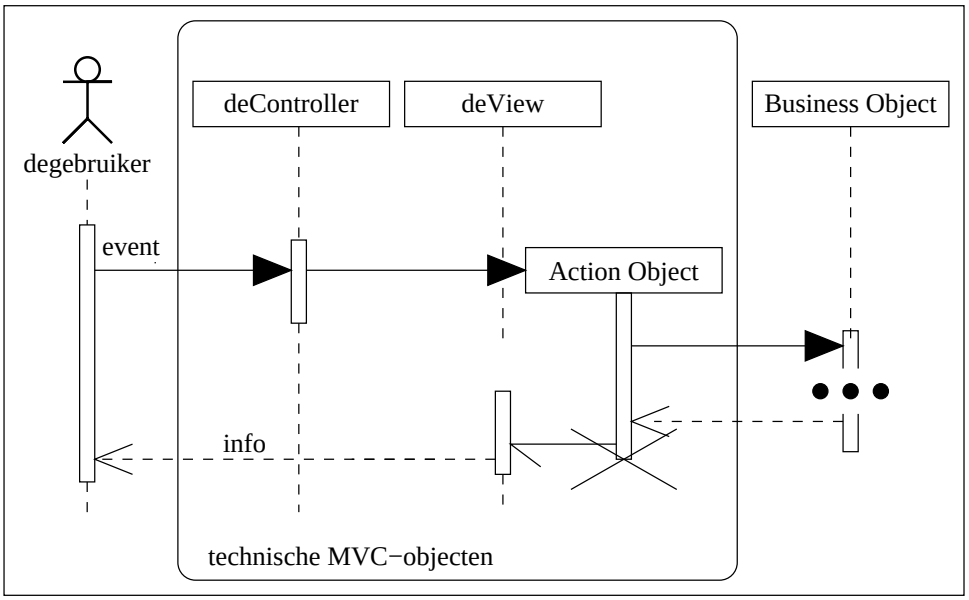
Bekijken we even het MVC-model aan de hand van een informeel schema dat de dataflow weergeeft. Er zijn heel wat variaties op het MVC-thema, Hier bekijken we



Figuur 2. Model-View-Controller met expliciete dispatcher en action classes

enkel een controller die bestaat uit een *dispatcher* en een aantal zgn. *action classes*. Een gebruiker die wil dat het systeem iets doet geeft een opdracht aan de dispatcher. Deze creëert een object van de juiste action class, en deze voert de opdracht uit. Informatie wordt doorgestuurd naar de gebruiker via de *view*. Er zijn heel wat varianten, zo is bij Java Swing de dispatcher verborgen, en zijn het de *event listeners* die de rol van action classes op zich nemen.

Het model van de realiteit dat we gemaakt hebben bepaalt de structuur van het modelgedeelte van bovenstaand schema. Controller en view komen niet expliciet voor. Als we de bovenstaande dataflow willen tekenen in een UML-sequentiediagram krijgen we ongeveer het volgende: Het is echter niet gebruikelijk in de modellering de technische klassen op te nemen: deze dragen immers niets bij aan het begrijpen van de structuur (behalve in de afbeelding, waar het de bedoeling is de MVC-structuur weer te geven). In het gebruikelijke UML-sequentiediagram van deze taak zal dus de gebruiker rechtstreeks onderhandelen met het business object van de tekening. De controller en de view mogen weggelaten worden uit het sequentiediagram, omdat ze gemeenschappelijk zijn aan alle sequentiediagrammen. Ze vermelden zou het diagram dus enkel maar ingewikkelder maken. Ook het action object moet niet vermeld worden. Het is gebruikelijk dat met elke taak uit het takendiagram één action class overeenkomt, zodat het tonen van het action object geen nuttige informatie oplevert. De regel



Figuur 3. Sequentiediagram zoals het normaal NIET getekend wordt

één taak t.o.v. één action class wordt niet altijd strikt toegepast, tenzij elke taak slechts één interventie van de gebruiker vereist, maar ook als de regel niet wordt toegepast gaat men geen action objects in het sequentiediagram plaatsen.

HOOFDSTUK 1
TESTEN EN DEBUGGEN

Debuggen is het halen van fouten uit code. Debuggen bestaat uit vier fasen:

1. Een fout wordt *ontdekt.*
2. De fout wordt *gelokaliseerd.*
3. De fout wordt *geïdentificeerd.*
4. De fout wordt *opgelost.*

Zoals vermeld in Hoofdstuk 17 kan het ontdekken van een fout gebeuren bij statische verificatie, bij testen of bij gewoon gebruik van de code. Bij statische verificatie is een fout automatisch gelokaliseerd; bij de andere manieren is het lokaliseren van de fout meestal de meest tijdrovende fase. We zullen hier dan ook deze fase uitgebreid bespreken.

1.1 TESTEN

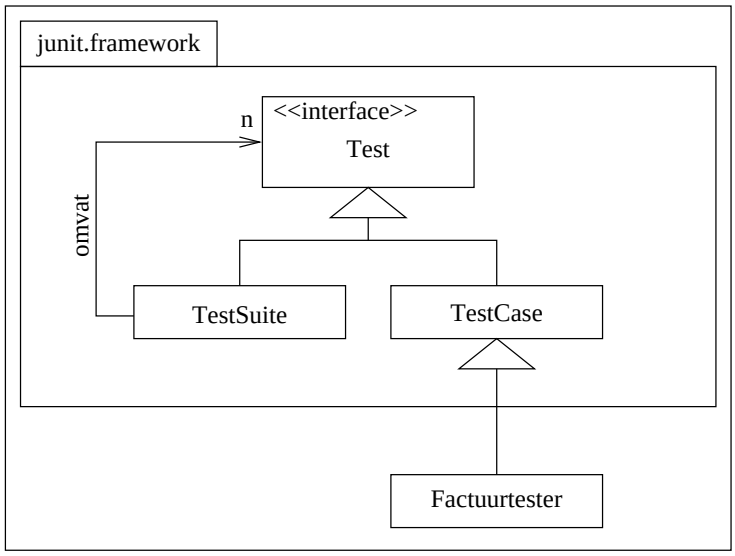
Bij een vrij klein systeem dat slechts enkele klassen omvat, gebeurt het testen op twee niveaus. Het laagste niveau (eenheidstesten, *unit testing*) is dat van individuele klassen. Testen hier zijn gericht op één methode, of op enkele methodes. Verder zijn er ook zogenaamde *integratietests*, die nagaan of de individuele klassen goed samenwerken. Traditioneel werden testen ontworpen en geschreven *nadat* de code gemaakt werd. Men ging er dan van uit dat deze code (op de bugs na) haar definitieve vorm had. Een test moest dan ook maar één keer uitgevoerd worden: ofwel doorstond de code de test, en dan was alles in orde, ofwel faalde de code en dan werd ze onmiddellijk aangepast. Het blijkt echter dat tests meerdere keren worden uitgevoerd. Daar zijn twee redenen voor:

1. Bij het debuggen van software is het mogelijk dat men bij een poging om een fout te verhelpen er andere introduceert. Bijgevolg moet men vorige tests, waar de code wel voldeed, herhalen.
2. Bij flexibele technieken, zoals extreme programming (XP), wordt ervan uitgegaan dat geschreven code steeds kan veranderd worden. Hier worden dan ook tests geschreven *voor* de code zelf. Bij het programmeren worden deze tests voortdurend (minstens ettelijke keren per dag) herhaald. Op deze manier wordt fouten vinden veel gemakkelijker: als een test die vroeger goed liep nu een fout oplevert, dan weet men bijna zeker dat dit iets te maken heeft met de recente veranderingen.

Er is dan ook een verschuiving van de klassieke vorm van testen, waarbij elke test apart gemaakt en manueel uitgevoerd werd, naar het maken van *testsuites*. Deze zijn volledig geautomatiseerd. De bedoeling is dat het testproces zo vlot mogelijk verloopt, om de programmeur aan te zetten deze zeer vaak uit te voeren, bijvoorbeeld elke keer als code gecompileerd wordt. Het is bijvoorbeeld zeer belangrijk dat de testsuite niet veel

uitvoer produceert, tenzij natuurlijk er iets misloopt. In het ideale geval schrijft een test alleen “OK” uit als het goed gaat. Dit ideaal kan natuurlijk alleen bereikt worden bij systemen en klassen die niet essentieel gebruik maken van het scherm als uitvoer- en toetsenbord en muis als invoerkanalen. Of iets juist op het scherm verschijnt moet manueel gecontroleerd worden, en ook de invoer verloopt manueel. Dit is een belangrijk obstakel bij een volledige automatisering van integratietests.

Op klassenniveau zijn er echter zeer veel tests die volledig geautomatiseerd kunnen worden, en er zijn verschillende hulpmiddelen die dit gemakkelijker maken. Zo is er voor Java bijvoorbeeld het *JUnit framework*. We schetsen hoe we hiermee een testsuite kunnen opzetten. Nemen we het voorbeeld van een klasse `Factuur` die een methode `berekenPrijs(Product p)` heeft. Wat we willen doen is het volgende: we willen deze methode laten lopen met een `Product` waarvan we weten dat de prijs 25 is, en controleren of het resultaat wel klopt. Hiervoor ontwikkelen we een klasse `Factuurtester`. Voor een test moeten we een omgeving opzetten: dit wordt een *fixture* genoemd. In ons geval bestaat de fixture uit een `Factuur`object en een `Productobject`. Binnen een fixture kunnen één of meerdere tests worden uitgevoerd; daarna moet de fixture worden afgebroken. Dit kan inhouden dat er beeldvensters moeten gesloten worden, of dat bestanden moeten worden opgeruimd, en zo verder. Binnen JUnit zijn er twee belangrijke klassen, die we laten zien in Figuur 1.1. Een



Figuur 1.1. Het JUnit framework.

`TestCase` biedt de mogelijkheid om een fixture met een of meerdere tests te definiëren. Een `TestSuite` kan een of meerdere `Tests` omvatten. Dit kunnen zowel `TestCases` zijn als andere suites. Onze `Factuurtester` ziet er dan ongeveer als volgt uit:

```
class Factuurtester extends TestCase{
    protected Factuur fc;
    protected Product p;
    protected void setUp(){
        ...
```

```
        fc=new Factuur(...);
        p=new Product(...);
        ...
    }
    protected void tearDown(){
        ...
    }
    public void testPrijs(){
        int prijs=berekenPrijs(p);
        assert(prijs==25);
    }
    ...
}
```

`setUp` en `tearDown` zijn de twee operaties van `TestCase` die toelaten de fixture op te bouwen en af te breken. De uitslag van een test wordt bepaald door de `assert`opdracht. Is aan de voorwaarde niet voldaan dan krijgen we een foutmelding in de aard van ‘‘`Factuurtester.testPrijs: error prijs != 25`’’. Is de voorwaarde wel voldaan dan krijgen we geen melding.

Er is uiteraard een nauw verband tussen de specificaties en het opzetten van een test: de precondities geven aan hoe een fixture opgezet wordt, terwijl de postcondities aangeven welke `assert`opdrachten er moeten geschreven worden.

Om de tests te laten lopen moeten we ze nog inbouwen in een testsuite, en deze suite laten lopen. De code hiervoor kan er als volgt uitzien:

```
TestSuite suite = new TestSuite();
suite.addTest(new FactuurTester("testPrijs"));
junit.textui.TestRunner.run(suite);
```

Merken we op dat hierbij de mogelijkheid van reflectie in Java, die in C++ ontbreekt, gebruikt wordt: de naam van een methode in stringvorm kan gebruikt worden om de methode zelf aan te spreken.

1.2 FOUTEN LOKALISEREN

Fouten kunnen optreden in de gegevensdefinities (bijvoorbeeld door het ontbreken van gegevensvelden), maar houden altijd verband met uitvoerbare code. Het is dus belangrijk de juiste plaats te vinden in de uitvoerbare code. Eerst zoekt men, zo nodig, in welke procedure (methode, lidfunctie, ...) de fout optreedt. Als de procedure zinvol kan ingedeeld worden met tussentoestanden (zie paragraaf 11.2), zoekt men het stuk code waar de fout haar oorsprong heeft. We noemen een stuk code tussen twee gedefinieerde toestanden een *sectie*. Een fout kan zichtbaar worden op een heel andere plaats dan waar ze ontstaan is. Bijvoorbeeld:

– Een sorteerprocedure bevat een fout, waardoor een tabel niet juist gesorteerd wordt.

– Een zoekprocedure gaat ervan uit dat de tabel correct gesorteerd is, en geeft daardoor een verkeerd resultaat terug.

Als we de fout willen verbeteren, dan moeten we uiteraard de sorteerprocedure veranderen, en niet de zoekprocedure. Algemeen veroorzaakt een fout in code een verkeerde toestand van het systeem, bijna altijd in de vorm van een aantal gegevensvelden die een verkeerde waarde aannemen. De code waarin deze verkeerde toestand aan het licht komt kan dicht bij of zeer ver van de code liggen waarin de fout ontstaan is. In het bovenstaande voorbeeld is er maar één tussenstap opgegeven, maar het verkeerde zoekresultaat hoeft niet onmiddellijk ontdekt te worden.

Als we een fout hebben, dan komt deze naar buiten als een bepaald stuk code wordt uitgevoerd, terwijl de fout niet noodzakelijk door deze code wordt veroorzaakt. Wat er juist verkeerd is kan altijd teruggebracht worden naar de toestand van het systeem. Er is een gegeven (dit kan een variabele zijn, of een waarde in een bestand, en zo verder) dat een verkeerde waarde heeft: het is dus belangrijk om dit nauwkeurig te omschrijven: *welk* gegeven is er verkeerd? Daarna gaan we de oorzaak zoeken: hoe komt het dat dit gegeven een verkeerde waarde heeft? We hebben dus altijd een van de volgende twee mogelijkheden:

1. We hebben een gegeven met een verkeerde waarde, en we zoeken de *plaats* in de code die verantwoordelijk is voor deze verkeerde waarde.
2. We hebben een gegeven met een verkeerde waarde, en we weten door welke sectie deze verkeerde waarde veroorzaakt wordt.

In het eerste geval hebben we een lokalisatieprobleem. We moeten opzoeken welke code de waarde verandert. Dit kan gebeuren door het gebruik van een debugger, waarmee we de code stap voor stap kunnen uitvoeren, waarbij we steeds de waarde van het gegevensveld in kwestie in het oog houden. Een andere methode is om code in te voegen die extra data uitschrijft. Dit kan de waarde zijn die in een foutief gegevensveld aanwezig is, of er kan aangegeven worden welke code juist wordt uitgevoerd. Dergelijke extra code kan met de hand worden toegevoegd, of door aspectgeöriënteerd programmeren. Eens we de plaats waar de fout veroorzaakt wordt hebben gevonden gaan we naar de tweede situatie: we hebben én een verkeerde waarde, én de code die ze veroorzaakt. Dan moeten we de fout identificeren. Het kan zijn dat de fout in de code zelf ligt (en dan kunnen we aan de verbetering ervan beginnen), maar het kan zijn dat de code zelf correct is, maar werkt met verkeerde gegevens. Als we de fout in deze gegevens geïdentificeerd hebben, dan zijn we terug in de eerste situatie.

1.3 IDENTIFICEREN VAN EEN FOUT

We hebben nu een sectie code en we weten dat het resultaat verkeerd is. In termen van paragraaf 11.2 moeten we dus rekening houden met het volgende:

1. De postcondities van de code. Deze zijn niet voldaan, want het resultaat is verkeerd. We moeten zien wat juist de overtreding van de postcondities is.
2. De precondities van de code. Een oorzaak van de fout kan zijn dat deze niet voldaan zijn. De toestand van het systeem aan het begin van de code is niet zoals ze moet zijn: we moeten dan identificeren wat er juist verkeerd aan is, en dan zijn we terug in situatie (1) uit de vorige paragraaf: we hebben een verkeerd gegeven maar weten niet waar ze veroorzaakt wordt.

3. De logica van de code zelf. Als aan de precondities voldaan is, maar niet aan de postcondities, dan is er iets mis met de code zelf.

Onderzoeken of aan de precondities voldaan is, is een vrij eenvoudig proces, tenminste als de precondities duidelijk omschreven zijn. De precondities bestaan uit een aantal toegelaten waarden van gegevens, en men moet deze gewoon aflopen om te zien of alles correct is.

Daarmee komen we bij het identificeren van een fout in de code zelf: precondities zijn voldaan, postcondities niet. De fout zelf vinden is vrij eenvoudig, vermits we een voorbeeld hebben van een situatie waarin het misgaat. We moeten enkel de code stap voor stap overlopen om te zien waar het juist misgaat. Dit overlopen kan op twee manieren gebeuren:

– We laten de code lopen. Tussenresultaten kunnen we controleren door met een debugger te werken, of door tussengevoegde uitschrijfoperaties, zoals we deden bij foutlokalisatie.

– We werken manueel: de code wordt niet uitgevoerd, maar we controleren wat er gebeurt door uit het hoofd, of met pen en papier, de werking van de code na te bootsen.

Het controleren van de code moet met de nodige zorg gebeuren. Vergeet nooit dat er al iemand is (de programmeur) vrij intensief met de code bezig is geweest. Zeer vaak is een fout het gevolg van het onzorgvuldig definiëren van precondities: de code komt dan terecht in een situatie die de programmeur niet voorzien had. Zorg ervoor dat je goed begrijpt wat de code doet (en wat ze moet doen). De toestand van het systeem verandert: je eerste taak is ervoor te zorgen dat je goed begrijpt wat de functies en eigenschappen zijn van elke variabele (en van elke constante, natuurlijk). In het oog te houden zijn:

1. De functie van de variabele. Deze zou normaal duidelijk moeten zijn aan de hand van de naam, vandaar het belang van goede naamgeving bij het schrijven van code. Soms is het niet mogelijk de functie volledig uit te leggen met een naam alleen: de functie zou dan moeten becommentarieerd worden bij de declaratie. Ga de declaratie opzoeken als deze niet staat in het stuk code dat je aan het analyseren bent.
2. Eventuele beperkingen op de toegestane waarden. Als de variabele een object voorstelt (denk hierbij aan het impliciete `this`object), dan kunnen er beperkingen zijn op de staat van het object. Als het gaat over elementaire variabelen (getallen, strings, . . .) kunnen er restricties zijn op de waarden.
3. Als de variabele een object is, let op de elementaire eigenschappen: de waarde van attributen; bij een tabel ook de lengte. Vooral als deze uitzonderlijk zijn (er is bijvoorbeeld een variabele `aantal`, en een tabel met lengte `aantal+2`) vormen ze een bron van fouten.

Verder is het nuttig om de zogenaamde *invariante condities* in het oog te houden. Het eenvoudigste voorbeeld krijgen we bij een dubbelgelinkte lijst: als de `volgend`pointer van een knoop `a` naar `b` wijst, dan moet de `vorig`pointer van `b` naar `a` wijzen. Bij het wijzigen van de lijst is deze voorwaarde gedurende een paar opdrachten niet voldaan,

maar dit mag zeker *niet buiten de sectie* komen[1]. Overtredingen op invariante condities moeten zo snel mogelijk hersteld worden.

We zeiden reeds dat de programmeur vrij intensief met de code bezig is geweest. Er zijn soms echter tekenen die erop wijzen dat de aandacht (of de kwaliteit) van de programmeur niet al te groot is geweest. Zulke tekenen wijzen vaak op fouten. Een lijstje:

– Elementaire fouten, zoals verkeerd commentaar, of een indentatie die niet overeenkomt met de indeling van de code. Een voorbeeldje van beide samen:

```
//Als x negatief is, wijzig dan y
    if (x>0)
        y-=3;
        z+=5;
```

– Stijlfouten. Programmeerstijl is bedoeld om code leesbaar te maken voor anderen, maar ook om fouten te voorkomen. Elke stijlfout is verdacht: zelfs eenvoudige fouten, zoals gebruik van `while` in de plaats van `for` of omgekeerd, verbergen vaak fouten. Zware fouten, zoals het afwijken van het vaste stramien

```
klaarzetten voorwaarde;
while (voorwaarde voldaan){
    behandel geval;
    zet volgende klaar;
}
```

verbergen zeer zelden geen fouten.

Bij het vinden van de fout is het nuttig een lijst in het hoofd te houden van de meest voorkomende fouten. Een klassieke lijst is de ADVB-classificatie:

A. Algoritme:

A.één-ernaast. De code berekent een getal dat één hoger is, of een lager dan de gewenste waarde.

A.logica. Er zit een logische fout in het algoritme. Vaak, maar niet altijd, is dit een gevolg van een verkeerde of onduidelijke beschrijving van pre- of postcondities.

A.validatie. Er wordt niet juist gecontroleerd of variabelen een correcte waarde hebben.

A.efficiëntie. De code loopt veel te traag, of heeft te veel geheugen nodig. Dikwijls te wijten aan een ontwerpfout, waardoor gegevens op een manier worden opgeslagen die niet interessant is voor de manier waarop ze gebruikt worden (bijvoorbeeld: een reeks gegevens opslaan in een tabel waarin lineair wordt gezocht), of omdat tussenresultaten herberekend worden in plaats van bijgehouden.

D. Data:

[1] Voor ingewikkelde invarianten is dit niet meer het geval: er kunnen zelfs verschillende private lidfuncties aan te pas komen om de invariant te herstellen.

D.index. De index van een tabel neemt verkeerde waarden aan.

D.grens. Er zijn fouten bij de speciale behandeling aan het begin of einde van een reeks gegevens. Voorbeeld: de waarden van een tabel moeten worden uitgeschreven met komma's tussen de waarden. Wat verschijnt is

```
, 1, 2, 3
```

of

```
1, 2, 3,
```

D.voorstelling. Een bug veroorzaakt door de manier waarop data worden voorgesteld in het geheugen. Dit soort fouten komt vaak voor bij programmatie dicht bij het machineniveau. Een voorbeeld: een programma dat werkt in een systeem dat ASCII gebruikt, maar niet werkt met Unicode.

D.geheugen. Het programma maakt fouten bij het geheugenbeheer. Voorbeelden zijn het niet aanmaken van objecten, en het vergeten van de `delete` in C++.

V. Vergeten:

V.init. Een variabele krijgt geen geldige beginwaarde.

V.ontbreekt. Een noodzakelijke opdracht ontbreekt.

V.locatie. Een opdracht staat op de verkeerde plaats.

B. Blunder:

B.Variabele. Er wordt een verkeerde variabele gebruikt.

B.uitdrukking. De berekening van een uitdrukking bevat een fout. Typisch voorbeeld in C++: er staat `if (a=b)` in plaats van `if (a==b)`.

B.taal. Een bug specifiek aan de syntax van de programmeertaal.

Aan deze classificatie moet zeker nog de de volgende klasse van fouten worden toegevoegd:

S. Symmetrie:

Sommige opdrachten hebben een tegenpool, waardoor een koppel ontstaat: openen/sluiten van bestanden, netwerkverbindingen en vensters, in C++ `new`/`delete`. De tweede helft van dit koppel staat dikwijls ver van de ene helft, en wordt dan ook gemakkelijk vergeten. Een goed programmeur schrijft dan ook *altijd* de tweede helft samen met de eerste (liefst zelfs voor de eerste), maar fouten tegen het symmetrieprincipe komen vaak voor.

Een voorbeeld van een B.taal-fout. Bekijken we het volgende C++-codefragment:

```
string str="aaaaaaaa";
for (int i=0;i<str.length();i++)
    if (str.substr(i,1)=="a")
        str.replace(i,1,"bb");
```

en vergelijken we dit met de volgende Perlcode

```
$str="aaaaaaaa";
foreach my $i (0..(length($str)-1)){
    if (substr($str,$i,1) eq "a"){
        substr($str,$i,1) = "bb";
```

```
        }
    }
```

We hebben ervoor gezorgd dat de opdrachten in de twee talen zo nauwkeurig mogelijk overeen komen, en de bedoeling is dat elke *a* in de string vervangen wordt door twee *b*'s. Maar eigenaardig genoeg is het resultaat in de twee talen zeer verschillend. De variabele `str` bevat op het einde van de C++-code de waarde "bbbbbbbbbbbbbbbb", maar met de Perlcode is het resultaat "bbbbbbbbaaaa"! Hoe is dit mogelijk? Het antwoord ligt in de verschillende interpretatie van een `for`opdracht in de twee talen. Bij C++ wordt gewerkt met een voorwaarde, die bij elke uitvoering van de lus gecontroleerd wordt. Omdat de string onderweg langer wordt, en dus de waarde van `str.length()` verandert, wordt de lus zestien keer uitgevoerd voor `i` even groot is als `str.length()`. Bij Perl echter wordt er *voor de eerste uitvoering van de lus* een lijst opgemaakt voor de waarden van de lusvariabele (die dus de waarden van 0 tot en met 7 aanneemt). Dat `length($str)`bij het uitvoeren verandert, doet niets meer aan deze lijst. De lus wordt 8 keer uitgevoerd, en de laatste 4 *a*'s worden niet meer bekeken.

HOOFDSTUK 2
REFACTORING

2.1 DE GESCHIEDENIS VAN NETSCAPE

De eerste veelgebruikte grafische webbrowser was Mosaic, gebouwd door een team van de universiteit van Illinois. In 1994 bouwden ze de opvolger Netscape, met versies voor verschillende platforms (Windows, Unix, Mac). Microsoft bracht de eerste versie van Internet Explorer uit halverwege 1995, maar tot ongeveer 1997 bleef Netscape met voorsprong de meest populaire internetbrowser. Het is pas met IE 3.0 dat Microsoft de kloof kon dichten. Daarna ging het met Netscape snel bergaf. Een geheel vernieuwde versie, Communicator 6.0, werd zelfs nooit afgewerkt, en er werd teruggegrepen naar een oude versie 4.0 met een poging om dit op te knappen. Veel is het nooit geworden, en uiteindelijk werd Netscape opgekocht door AOL.

Het probleem was onzorgvuldig design. De eerste versie van Netscape was een *quick and dirty* product: Het werkte, maar de code was slecht gestructureerd en dus moeilijk te onderhouden en uit te breiden. Naarmate de browser groter werd en het internet ingewikkelder (allerhande soorten plug-ins, nieuwe standaarden, animatie,...) stegen de kosten onevenredig veel: Mosaic was gemaakt door een team van 10 ontwikkelaars, Navigator 4.0 door 120 mensen. IE 1.0 was ook niet goed gestructureerd, maar Microsoft onderkende het probleem tijdig en herzag de code grondig voor IE 3.0.

Internetbrowsers bestaan in een snel evoluerende omgeving, zodat code dikwijls grondig moet herzien worden. Hierdoor degradeert de kwaliteit: rond 1996 crashte zowat elke browser regelmatig. Bovendien wordt de code moeilijker en moeilijker aanpasbaar. Vandaar de beslissing van Netscape in 1998 om van de grond af aan opnieuw te beginnen. Dit is uiteraard een zeer duur proces, en men is op zoek gegaan naar methodes om bestaande code te verbeteren. Dit heeft geleid tot het begrip *refactoring*: het *herwerken* of herschikken van code. Microsoft heeft de code voor IE succesvol herwerkt voor versie 3.0, Netscape probeerde eerst om vanaf nul te herbeginnen (versie 6.0), en herwerkte dan de bestaande code zonder veel succes. Dit lag o.a. aan het feit dat er voor de herwerking veel te weinig tijd voorzien was.

2.2 TOEPASBAARHEID VAN HERWERKEN

Bij het schrijven, uitbreiden en wijzigen van software is het altijd noodzakelijk om de organisatie van de software aan te passen. Eén van de basisprincipes van herwerken is om de twee processen te scheiden: men maakt eerst de bestaande code klaar om wijzigingen in aan te brengen, en *daarna* wijzigt men de functionaliteit. Herwerken is dus de *reorganisatie* van code *zonder* dat de functionaliteit wijzigt: de resulterende code dient op exact dezelfde manier te werken als de oorspronkelijke. De bedoeling hiervan is tweevoudig. Het eerste doel is om de code zo duidelijk en begrijpbaar mogelijk te maken; het tweede doel is om de software flexibel te maken. Het is duidelijk dat her-

werken geen nut heeft tenzij de code nog moet gewijzigd worden. Er zijn verschillende situaties waarbij herwerken aangewezen is:

– Bij het gebruik van *incrementele* ontwikkelingsmethodes. Hierbij kan het zowel gaan om een voorafgaand geplande incrementele methode met duidelijk afgelijnde builds, of, zoals in het geval van de browser, van een uitbreiding van bestaande software omwille van een wijziging in de omgeving.

– Bij het *hergebruiken* van code. Indien de code ongewijzigd kan overgenomen worden heeft herwerken niet veel zin, maar dikwijls wordt overgenomen code wel gewijzigd. Herwerken maakt dan deel uit van het leren kennen van de code. In plaats van vreemd uitziende constructies in de overgenomen code goed te leren kennen verwijdert men ze gewoon.

– Bij het ontwikkelingsproces. Vaak blijkt dat het ontwerp van de code gebreken vertoont en dat reorganisatie zich opdringt. Dit kan dan best zo snel mogelijk gebeuren, met het oog op testen en debuggen.

Reorganisatie van code is een proces dat uitgaat van de code zelf, en niet van een analyse- of ontwerpdocument. Het gaat dan ook uit van lokale mankementen van de code: elke keer dat code moeilijk te begrijpen is, of niet flexibel, zal men proberen te herstructureren. Het mankement wordt ook zo lokaal mogelijk opgelost: dikwijls binnen een methode. Slechts als men de noodzaak voelt om hogerop te gaan, zal dit gebeuren, en men zal op deze manier nooit verder geraken dan het niveau van ofwel een hiërarchie van van elkaar overervende klassen, ofwel een kleine groep van samenwerkende klassen.

De reorganisatie berust op een relatief klein aantal, welomschreven acties. Vermits het de bedoeling is om de functionaliteit van de software te behouden, dient gegarandeerd te worden dat dit inderdaad zo is. Het is dus uit den boze om de fundamentele structuur van de software te hertekenen: wie veel wijzigingen tegelijkertijd doorvoert loopt een groot risico op verandering van de functionaliteit. Ook is het niet de bedoeling om iets nieuws te proberen: men gebruikt best alleen acties uit de standaardlijst. Elk van deze acties heeft dan ook een vaste *naam*, zodanig dat het gemakkelijk is om ernaar te verwijzen. Als zulk een actie correct wordt uitgevoerd, dan is er gegarandeerd dat er niets verandert aan de functionaliteit. De acties zijn echter redelijk delicaat, en er kunnen onverwachte problemen opduiken. Nemen we als voorbeeld een van de eenvoudigste acties: het hernoemen van een methode van een klasse (de naam van deze methode is, zeer toepasselijk, *hernoemen van methode*[1]). Op het eerste gezicht is dit zeer eenvoudig: Als de oude naam `foo` was, en de nieuwe `bar`, verandert men gewoon overal in de code `foo` in `bar`. Hiermee kunnen verschillende problemen opduiken. Ten eerste is het best mogelijk dat er andere entiteiten zijn die ook `foo` heten, en waarvan het niet de bedoeling is dat ze van naam veranderen. Maar een nog groter probleem duikt op als de naam `bar` reeds gebruikt wordt voor andere entiteiten. In de meeste gevallen levert dit enkel compileerfouten op, maar er is een geval waar de fout

[1] In de terminologie die we vroeger gebruikten is een methode de implementatie van een operatie, en is het de operatie die een naam heeft. Zo is er in een hiërarchie van klassen bijvoorbeeld een *operatie* `foo`, waarbij de methode kan overschreven worden. Vanuit dit standpunt zou de naam *hernoem operatie* moeten zijn.

subtieler kan zijn, en dat is wanneer de naam `bar` ook al bestaat als de naam van de functie met dezelfde signatuur uit een of andere bovenklasse. In dit geval overschrijft de hernoemde operatie de andere `bar`operatie, en zijn de gevolgen onvoorspelbaar. Overigens kan het ook zijn dat `foo` een overschrijving was van een operatie die hogerop gedefinieerd was, en ook dan kunnen de gevolgen van hernoemen zeer vreemd zijn.

Herwerken steunt dan ook op het zeer dikwijls testen van de herschreven code. Het is volstrekt noodzakelijk om te werken met een automatische testomgeving, zodanig dat de wijzigende software zeer dikwijls kan onderworpen aan rigoureuze tests. Het veelvuldig laten lopen van de tests zorgt ervoor dat de oorzaak van bugs snel kan worden opgespoord. Immers, als na een kleine wijziging een bug wordt gesignaliseerd door de tests, dan is het duidelijk dat de bug te maken heeft met die kleine wijziging. Misschien is ze er niet door veroorzaakt, maar ze is dan wel op de voorgrond gekomen door die wijziging. Als men pas test na tientallen veranderingen te hebben aangebracht, moet men uitzoeken bij welke van die veranderingen die bug hoort.

Zoals reeds blijkt uit het voorbeeld van een methode met een nieuwe naam, is het met de hand uitvoeren van een herwerkingsoperatie delicaat en vervelend werk, en dus typisch iets dat we door een programma willen laten doen. Zo'n programma's kunnen bestaan, juist omdat het herwerkingsproces is opgebouwd uit welomschreven acties. Dergelijke programma's bestaan dan ook, en ze versnellen het werk ten zeerste. Merk op dat zulke programma's vrij ingewikkeld zijn, omdat ze niet kunnen volstaan met een oppervlakkige analyse van de code. Als, met bovenstaand voorbeeld, zo een programma ergens de uitdrukking `p.foo()` ziet staan, moet het weten of `p` tot de klasse behoort waarbij `foo` moet veranderd worden in `bar` of niet. Dergelijke programma's zijn dus niet veel eenvoudiger dan een compiler voor de betreffende taal. Niet alleen is zo een programma zeer nuttig om de actie zelf uit te voeren (waarbij er niet alleen tijdwinst is, maar waarbij de kans op fouten ook ten zeerste vermindert), ook kan zulk een programma controleren of de actie wel toelaatbaar is. Nemen we het voorbeeld van een andere actie: *extractie van code* uit een methode. Bij deze actie wordt een gedeelte van de code in een methode aangeduid. Deze wordt dan ondergebracht in een nieuwe, aparte methode. Op de oorspronkelijke plaats komt dan een oproep te staan naar de nieuwe methode. Hierbij moeten verschillende controles gebeuren op lokale variabelen. Indien een lokale variabele die gedeclareerd wordt binnen het aangeduid stuk code buiten het stuk nog gebruikt worden, dient men na te gaan of de waarde van die variabele met een uitvoerparameter moet teruggegeven worden. Als het uit te knippen stuk zelf lokale variabelen gebruikt die vóór het stuk werden geïnitialiseerd, dan moet men ook nagaan of de *waarde* gebruikt wordt, en eventueel een parameter voorzien in de oproep van de methode. Merk op dat men wel degelijk moet nagaan of de *waarde* belangrijk is, omdat domme variabelen vaak hergebruikt wordt. Nemen we bijvoorbeeld

```
int tmp;
tmp=a;a=b;b=tmp;
...
tmp=0;
for (int i=0;i<aantal;i++)
    tmp+=tab[i];
```

Hier wordt dezelfde variabele op twee plaatsen gebruikt, maar het is duidelijk niet nodig om de waarde van `tmp` mee te geven als parameter indien het tweede stuk apart wordt gezet: dit levert een volstrekt overbodige parameter op, en dus zinloze koppeling. Er is overigens een herwerkingsactie waarbij de `tmp`-variabele in het tweede stuk code vervangen wordt door een nieuwe variabele.

Door het bestaan van herwerkingstools is refactoring veel goedkoper en sneller geworden. Op deze manier worden herwerkingsacties meer en meer verweven in het proces van het maken of wijzigen van software. Men gaat niet meer eerst herwerken en daarna de aanpassingen doen, maar men voert die herwerkingsacties uit die nodig zijn voor een bepaalde aanpassing. Dit heeft als voordeel dat men code die men niet moet aanpassen niet eens moet bekijken. Nadeel is wel dat men op deze manier minder goed gestructureerde code krijgt: als een organisatiefout die wijziging moeilijk maakt niet direct in de buurt van het probleem ligt, dan zal men de refactoringsactie niet uitvoeren. Nemen we het voorbeeld van een operatie `foo()` die is ondergebracht bij een klasse A maar eigenlijk hoort bij klasse B. Als men nu de functionaliteit van B wil uitbreiden met een nieuwe methode en daarvoor `foo()` nodig heeft, dan zal men deze niet vinden, en bijvoorbeeld zelf een nieuwe `fookloon()` schrijven, eventueel inline in de nieuwe methode van B. Wie echter systematisch de code doorloopt zal bij A zien dat de methode `foo()` niet op zijn plaats staat, en deze overbrengen naar B. Welke methode het beste is: herwerken en wijzigen van code zeer duidelijk scheiden, of een vermenging van de twee, zal afhangen van een aantal factoren. Zo is herschikken zonder wijziging een goede techniek om code voor hergebruik te leren kennen, en zijn aanpassingen pas mogelijk als men de code enigszins kent. Als men verwacht dat men veel aanpassingen zal moeten doen is het dan ook interessant om eerst alle code te herschikken. Als men slechts enkele kleine wijzigingen verwacht moet men niet alle code leren kennen, en kan men gericht herschikken tot de wijzigingen kunnen worden aangebracht. Merken we tenslotte nog op dat het systeem van scheiden van herwerken en wijzigen niet altijd op veel bijval kan rekenen van het management, zeker niet wanneer er een strikt tijdschema te halen valt. De doelstellingen van herwerken, duidelijkheid en flexibiliteit worden dan vaak gezien als eigenschappen met weinig belang. Een periode uittrekken om code te veranderen in code die juist hetzelfde doet lijkt dan, speciaal voor mensen die niet rechtstreeks met het programmeerproces verbonden zijn, volledig nutteloos.

Herschikking van code maakt het tot op zekere hoogte mogelijk om fouten in het ontwerp van code op te vangen. Dit kan echter geen excuus zijn om onzorgvuldig te gaan ontwerpen. Zoals gezegd is refactoring een bottom-upproces: uitgaande van lokale eigenschappen van code wordt geherstructureerd. Hierdoor worden kleine ontwerpfouten gemakkelijk afgevlakt. Naarmate de fouten op een hoger niveau liggen komen er echter problemen. Ten eerste heeft men, door het werken vanop het niveau van code, niet veel overzicht: het is daarvoor dat UML-diagrammen dienen. Ten tweede is er op den duur veel code betrokken bij het herwerken, zodat het proces ingewikkeld en traag wordt.

2.3 UITWERKING

De volgende stappen worden ondernomen bij herschikking:

– Het overlopen van de code en het ontdekken van problemen.

– Het bepalen van de juiste actie of acties om te herschikken.

– Het uitvoeren van de acties en het testen van het resultaat.

Het overlopen van de code kan, zoals gezegd, gebeuren naar aanleiding van de adoptie van code voor hergebruik, of omwille van de noodzaak tot aanpassing. Bij nieuwe code is het ook nuttig aan herschikking te denken bij de acceptatie van code. Als er een review van code gebeurt door een groep bestaande uit de programmeur en anderen betrokken bij het project dan is dit het ogenblik om na te gaan of de code door degenen die ze niet geschreven hebben als duidelijk en natuurlijk ervaren wordt: zo niet dient herschikking overwogen te worden. Immers, op dit ogenblik is het waarschijnlijk dat er aan de code nog wijzigingen worden aangebracht, omdat er nog getest moet worden (eventueel alleen integratietests). Bijgevolg is verbetering van leesbaarheid en van flexibiliteit belangrijk. Merk op dat het hier misschien alleen gaat over flexibiliteit die te maken heeft met verbetering van fouten, en niet over flexibiliteit in verband met uitbreiding van de mogelijkheden. Zoals we zullen zien zijn sommige herschikkingsacties toegespitst op het laatste soort flexibiliteit. In een omgeving met een lage graad van volatiliteit van vereisten zullen deze acties een lage prioriteit krijgen, vooral omdat ze kunnen worden uitgesteld tot het ogenblik dat de noodzaak zich voordoet.

Een belangrijk aspect van de kennis van herschikkingstechnieken is dat ze de nadruk leggen op een goede lokale organisatie van code. Het vinden van potentiële probleempunten is van cruciaal belang. Vandaar dat het nuttig is om een aantal indicatoren van problemen in meer detail te bekijken, zelfs voor programmeurs die nauwelijks in contact zullen komen met herwerkingstechnieken. Immers, het vermijden van zulke probleempunten leidt tot goede code op zich. Een gids voor refactoring kan dan gelezen worden als een gids voor stijlvol programmeren, maar dan op een hoger niveau: waar een stijlgids het heeft over de code *binnen* een methode, gaat refactoring voornamelijk over de organisatie van code zoals ze verdeeld wordt over de operaties van een klasse, en over verschillende klassen.

Bij een goed ontwerp en uitwerking is herwerken van code niet zo dikwijls nodig. Soms worden de gevolgen van een aantal ontwerpbeslissingen pas duidelijk bij programmeren, en dan dient het ontwerp aangepast: de herwerkingsactie op codeniveau weerspiegelt zich dan in een wijziging in het ontwerp. Toch is dit soort aanpassingen vrij klein als het gaat om code die afgeleid wordt uit een ontwerp gebaseerd op stabiele vereisten. Het is pas als die vereisten volatiel zijn dat code vaak wordt herwerkt en zijn flexibiliteit verliest: bij Netscape was dat het geval na ettelijke versies. Dan is refactoring een hoge noodzaak.

Het is echter niet gemakkelijk om de symptomen van probleempunten zeer duidelijk te omschrijven. Er is een beperkt succes met het gebruik van softwaremetrieken. Zo kan het zijn dat een methode van een klasse vaak verwijst naar attributen en operaties van objecten in een bepaalde andere klasse. Dan kan men de vraag stellen of deze methode niet thuishoort in die andere klasse. Nu kan men het aantal van dergelijke verwijzingen in principe tellen met een programma (het is m.a.w. een softwaremetriek), en dus is het mogelijk een programma te schrijven dat dergelijke herwerkingsactie aanbeveelt. Er kunnen echter goede redenen zijn waarom de actie niet gewenst is. Herwerken van code is dus voornamelijk werk voor een ervaren programmeur. Het valt trouwens op dat bijna alle herwerkingsacties een tegengestelde actie hebben. Dit kan een gelijksoortige actie zijn: een methode hernoemen wordt teniet gedaan door de

methode terug de oude naam te geven, maar soms komen acties in paren. We hebben gezien dat we bij extractie van code een nieuwe methode creëren waarin code uit een lange methode wordt ondergebracht. Deze actie heeft zijn tegenpool: *inline brengen van methode*, waarbij een methode wordt verwijderd, terwijl haar code wordt ondergebracht in een oproepende methode. Bijgevolg is het niet zo dat men moet zoeken naar zoveel mogelijk gelegenheden om een herwerkingsactie toe te passen: meestal is het de kunst om de voor- en nadelen tegen elkaar af te wegen. Het is in verband met herwerken van code gebruikelijk om te spreken van de geur (Eng: *smell*) van probleemcode. Een ervaren programmeur is veel meer gevoelig voor dergelijke geur dan iemand die nieuw is in het vak. Toch is het mogelijk een aantal herkenningspunten op te geven.

We kunnen drie grote niveaus onderscheiden waarop kan gewerkt worden:

- Binnen een enkele klasse.
- Samenwerking tussen twee klassen, apart of binnen een hiërarchie van overerving.
- Tussen verschillende hiërarchieën van klassen.

Verder zijn er twee criteria die gebruikt worden:

- De begrijpelijkheid van code.
- De flexibiliteit van code.

Intuïtief kan men het tweede criterium als volgt beschrijven: code is flexibel als een wijziging in de functionaliteit weinig werk vraagt. Een van de principes hierbij is *eenheid van plaats*: als om een functie te veranderen code moet worden aangepast op verschillende plaatsen, dan is dit veel werk (al deze plaatsen moeten gezocht en gevonden worden). Bovendien is er veel kans op fouten: een plaats die gewijzigd moet worden kan vergeten worden. Bovendien is er een grote koppeling: voor elke te wijzigen plaats moet worden nagegaan of de wijziging de overige werking van het systeem niet stoort. Een tweede principe is *standaardisatie* van de aanpassing. Zo mogelijk moet de verandering gebeuren op een manier die *geen* inventiviteit van de programmeur vraagt. Een standaardmethode is gemakkelijker om juist uit te voeren en levert dus minder fouten op; bovendien is ze gemakkelijker te begrijpen als de code achteraf weer moet gewijzigd worden. Sommige herwerkingsacties kunnen verschillende soorten problemen oplossen. Het zal het nuttigste zijn om herwerken meer in detail te bespreken aan de hand van symptomen die op problemen wijzen: van daaruit kunnen we dan een aantal herwerkingsacties nader belichten. Het is belangrijk om te realiseren dat veel van de indicatoren van probleemcode al kunnen herkend worden bij het ontwerpproces. Dat geldt dan niet zozeer voor problemen binnen een klasse, maar wel voor klassenoverschrijdende problemen.

2.4 PROBLEMEN BINNEN EEN KLASSE

Veel van de problemen die optreden binnen een enkele klasse kunnen ook optreden met code die verspreid is tussen verschillende klassen.

In totaal onderscheiden we volgende probleempunten:

1. Onduidelijke naamgeving.
2. Onjuiste indeling private operaties.
3. Overmaat aan conditionele uitdrukkingen.
4. Onechte gegevensvelden/deelobjecten.
5. Te grote klasse.
6. Overdaad aan parameters.
7. Intelligente attributen.

Waarvan we het eerste reeds behandeld hebben. De eenvoudigste herwerkingsoperatie is het hernoemen van elementen: operaties, attributen, associaties, klassen. Dit verandert niets aan de functionaliteit en is dus puur gericht op betere begrijpelijkheid. Toch moet de nodige voorzichtigheid in acht worden genomen.

Een tweede probleempunt heeft te maken met private hulpoperaties. Bij het ontwerp van code hebben we gezien dat er twee redenen zijn om private operaties in een klasse aan te brengen. Eén ervan was het bestaan van gedupliceerde code: dezelfde handeling wordt in twee of meer operaties, of binnen een operatie, verscheidene malen toegepast. Dit is vaak iets wat op ontwerpniveau over het hoofd wordt gezien: pas bij het voor de tweede maal coderen heeft de programmeur het gevoel dat "hij die code al eens eerder heeft gezien". Dit is het ogenblik om te besluiten om de duplicaatcode onder te brengen in een aparte methode[2]. Wordt duplicaatcode gevonden in bestaande code dan wordt de methode *extraheer methode* gebruikt, en worden de verschillende kopieën van de code allemaal vervangen door een oproep naar de nieuwe operatie. De tweede reden om private operaties te creëren is dat een grote methode kan ingedeeld worden in verschillende deeloperaties. Soms is het mogelijk een deel van de code zinvol onder te brengen in een aparte operatie. Niet alle specialisten zijn enthousiast over zo'n maatregel. Onder het motto dat encapsulatie op alle niveaus maximaal moet zijn, eist men dat de code, vermits ze een deel is van die ene grote methode, binnen deze methode verborgen blijft. De maatregel om code te extraheren maakt de code van de grote methode leesbaarder, maar ze verdringt de andere (private) methodes uit de aandacht. In elk geval is hij niet gewenst als de lange code werkelijk samenhangend is, en bijvoorbeeld een enkel algoritme beschrijft, waarvan de losse onderdelen geen zin hebben. Vandaar er naast *extraheer methode* ook een tegengestelde actie is, *inline brengen van methode*.

Een derde probleempunt heeft te maken met het voorkomen van conditionele uitdrukkingen in de vorm van `if`- of `switch`opdrachten. Uiteraard is het niet de bedoeling om alle zulke opdrachten te verwijderen, maar ze kunnen een symptoom zijn van verschillende fouten. Conditionele uitdrukkingen zorgen ervoor dat één stuk code verschillende zaken doet, en men moet de vraag stellen of deze verschillende zaken niet moeten gescheiden worden. Het probleem kan zich voordoen op verschillende niveaus. Bij een ervan is er sprake van een methode die twee of meer geheel verschillende zaken doet. Geven we een (extreem) voorbeeld:

```
void zetAfmeting(int a, bool opzij){
    if (opzij)
        breedte=a;
    else
```

[2] In de praktijk blijkt dat programmeurs dit meestal pas doen als ze de code voor de *derde* maal intikken.

```
        lengte=a;
}
```

Deze code kan gemakkelijk vervangen worden door *twee* methodes; de herwerkoperatie heet dan ook *vervang parameter door expliciete methodes*:

```
void zetBreedte(int a){
    breedte=a;
}
void zetHoogte(int a){
    hoogte=a;
}
```

Het alternatief is veel eenvoudiger om te begrijpen: het gedrag wordt onmiddellijk afgeleid uit de procedurenaam. In het eerste geval moet de oproeper ook expliciet kiezen voor lengte of breedte door de `opzij`parameter in te vullen. Bovendien moet hij in de code kijken om te zien wat er juist gebeurt.

Een tweede probleem dat gekenmerkt wordt door conditionele uitdrukkingen is het bestaan van aparte staten en/of deelklassen. Zoals we weten is er een nauw verband tussen de twee: overerving met restrictie kan gemodelleerd worden met verschillende staten. Er zijn verschillende mogelijkheden:

- De conditie hangt niet af van de toestand van het object. Meestal is er dan geen probleem, maar als de conditie afhangt van de toestand van een ander object, kan het zijn dat in de andere klasse er staten en/of deelklassen zijn.

- De conditie hangt af van de toestand van het object, maar de toestand kan niet zo veranderen dat een ander alternatief gekozen wordt. Als er bijvoorbeeld twee alternatieven A en B zijn, dan zijn er objecten die altijd A kiezen en objecten die altijd B kiezen: welk van de twee het wordt is bepaald door de constructor. In dit geval zijn er eigenlijk twee klassen. Ofwel dient de ene een deelklasse te worden van de andere (en wordt de problematische methode overschreven), ofwel moeten ze beide deelklasse worden van een gemeenschappelijke bovenklasse. Waarschijnlijk is deze bovenklasse een abstracte klasse.

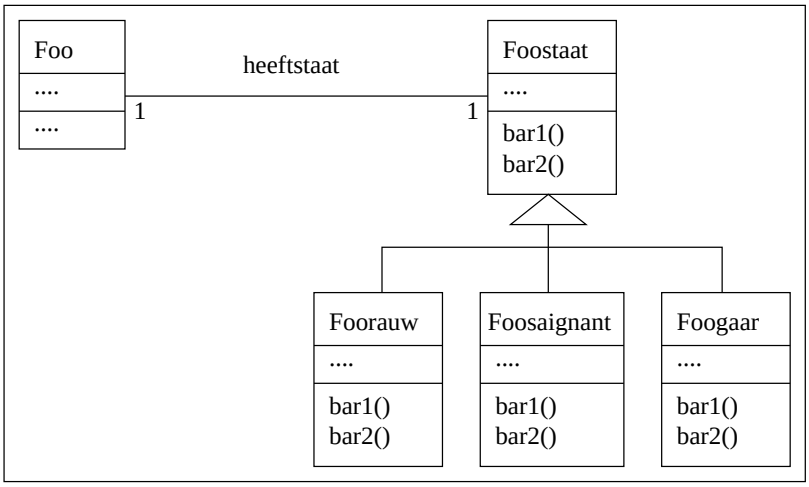
- De conditie hangt af van de toestand van het object, maar een object kan soms het ene alternatief en soms een ander kiezen. In dit geval zijn er staten.

Indien er duidelijk verschillende staten zijn, waarin verschillende methodes een andere reactie vertonen, dan kan het nuttig zijn om te werken met het *state pattern*. Hierbij worden de conditionele methodes ondergebracht in een hiërarchie van zogenaamde *statenklassen*. Veronderstellen we dat we een klasse `Foo` hebben met een `enum`veld met mogelijke waarden `rauw, gaar, saignant`. Er zijn twee methodes met condities:

```
void bar1(){
    switch(staat){
        case rauw:      actieA1();break;
        case saignant:  actieB1();break;
        case gaar:      actieC1();
    }
```

```
void bar2(){
    switch(staat){
        case rauw:      actieA2();break;
        case saignant:  actieB2();break;
        case gaar:      actieC2();
    }
```

Duidelijk hebben we hier drie staten. Met het statenpatroon verwijderen we nu de operaties `bar1()` en `bar2()` uit de klasse `Foo` en brengen ze onder in een abstracte klasse `Foostaat`. Er komen voor elke operatie drie methodes, die zijn ondergebracht in drie deelklassen van `Foostaat`. Het resultaat is te zien op Figuur 2.1. Merk op dat `Foostaat` en zijn deelklassen *technische* klassen zijn, in de zin dat er geen objecten uit de realiteit mee overeenkomen. Met een reëel object komt een gecombineerd paar van een `Foo` en een `Foostaat` overeen. Tenslotte moet vermeld dat het attribuut *staat* uit de klasse `Foo` zinloos is geworden, en dus ook verdwijnt. In plaats van dit attribuut te wijzigen wordt bij verandering van staat het `Foostaat`object vervangen.



Figuur 2.1. Een hiërarchie van statenklassen

Probleem vier is dat van onechte attributen en deelobjecten. Sommige lidvelden van een klasse hebben geen betekenis als een object “in rust” is, d.w.z. als er geen enkele operatie in uitvoering is. Dit wijst op een grote ontwerpfout: gegevensvelden moeten de toestand aanduiden, en een gegevensveld dat alleen zin heeft als een of andere operatie bezig is dient binnen die operatie te blijven. Een klassiek voorbeeld van zo’n ontwerpfout is een `Graaf`klasse die als deelobject een queue van knopen bevat die gebruikt wordt bij breedte-eerst zoeken. Deze queue is zinloos als er geen breedte-eerst zoeken aan de gang is, en mag dus geen deel uitmaken van het `Graaf`object.

Soms is het echter zo dat er een samenhangende groep van operaties is, meestal bestaande uit een publieke operatie en een aantal private operaties, die alleen door de publieke methode gebruikt worden. Vaak wisselen deze ook veel gegevens uit, zodat er óf veel parameters in de hoofdingen staan, óf lidvelden van de klasse gebruikt worden om de gegevens door te geven. Vanuit het standpunt van modulair ontwerp is dit geheel

een module met goede afscheiding: het is dan ook nodig om dit in een aparte structuur onder te brengen. In dit geval verdient het aanbeveling al deze methodes samen met de gegevensvelden onder te brengen in een aparte klasse. Dergelijke klasse wordt een *methodeklasse* genoemd. Strikt genomen is het geen klasse: vaak is de enige publieke operatie de constructor: deze voert de operatie uit, en daarna kan het object verdwijnen.

Probleem vijf doet zich voor als we een grote klasse hebben, d.w.z. een met veel gegevensvelden en/of operaties. Het is dan bijna steeds mogelijk om deze op te splitsen. Herinner je dat een vaste regel is dat op elk niveau van modulaire indeling er hoogstens vijf à tien verschillende deelmodules mogen zijn. Het kan zijn dat we een methodeobject kunnen maken om een deel af te splitsen; ook een gewone klasse kan soms gebruikt worden.

Zesde probleem is dat van lange lijsten parameters. Er is bijna steeds iets mis met een operatie die veel parameters heeft. De beschrijving van een grens van een module moet steeds zo klein mogelijk zijn, en een lange parameterlijst is een lange beschrijving. In veel gevallen zijn alle (of bijna alle) parameters gegevensvelden van één object. In dit geval kan men een referentie naar het object doorgeven in plaats van alle parameters. Merk op dat dit de code ook flexibeler maakt: als een operatie veel informatie nodig heeft van een bepaald object, is er veel kans dat een licht gewijzigde versie van de operatie ook nog andere informatie nodig heeft van dat object. Met de lange parameterlijst is dat niet mogelijk, met een referentie naar het object wel. Als er geen object is dat de gegevens samenbrengt, kan men zich afvragen of er geen zou moeten zijn: dit lijkt sterk op het methodeobject dat we bij het vorige probleem hebben ingevoerd.

Tenslotte is er het probleem van operaties die toegespitst zijn op een gegevensveld. Hierdoor heeft dit gegevensveld extra functionaliteit: het is een *intelligent attribuut* geworden. Nemen we het voorbeeld van een klasse `Persoon` met een attribuut `telefoonnummer` dat gedefinieerd staat als een element van de klasse string, en waarbij `Persoon` een operatie heeft om een zonenummer uit het telefoonnummer te isoleren. Alhoewel het hier alleen maar gaat over één attribuut met één operatie, is het toch beter deze apart te zetten. Immers, blijkbaar is een `telefoonnummer` een speciaal soort string, waaruit een zonenummer te halen valt. Deze operatie hoort duidelijk niet bij `Persoon`. Het is beter in dit geval een klasse `Telefoonnummer` te creëren en het zoeken naar het zonenummer daarin op te nemen. Merk op dat `Telefoonnummer` een deelklasse wordt van `string`. Een ander voorbeeld is dat van een geografische positie. Aanvankelijk zal die, waarschijnlijk in de vorm van twee getallen, als attributen worden geïmplementeerd. Als blijkt dat er verschillende operaties mee gebeuren (zoals overgaan van een coördinatenstelsel naar een ander, bijvoorbeeld van lengte- en breedtegraad naar plaats op een landkaart), dient dit overgeheveld te worden naar een aparte klasse.

2.5 PROBLEMEN TUSSEN TWEE KLASSEN

Veel van de problemen die optreden binnen een klasse kunnen ook voorkomen tussen twee klassen. De meeste problemen hebben echter te maken met een te grote koppeling tussen twee klassen. We onderscheiden volgende problemen:

1. Duplicaatcode.

2. Veel gebruik van gegevensvelden van de andere klasse.
3. Onvolledige klassen.
4. Onafhankelijke klassen met gelijkaardige functionaliteit.
5. Problemen met delegatie.
6. Het hagelverwijderensymptoom.
7. Divergente verandering.

Het eerste probleem is het bestaan van duplicaatcode. Het bestaan van duplicaten in verschillende klassen is niet zo wijd verspreid als duplicaten binnen een klasse, maar de problemen zijn hetzelfde. Het is mogelijk dat het dupliceren van code niet eenvoudig te vermijden is. Een goed criterium is hierbij of bij wijziging beide duplicaten moeten veranderd worden of niet. Als bijvoorbeeld twee klassen gebruik maken van binair zoeken in een gesorteerde tabel is er weinig reden om zich zorgen te maken: binair zoeken is een algoritme dat niet afhangt van de omstandigheden, alleen van het bestaan van een gesorteerde tabel. Als deze tabel in een van de klassen verdwijnt moet de code in de andere klasse niet aangepast worden. Als daarentegen twee klassen dezelfde code gebruiken om resultaten uit te schrijven, dan is het nodig om naar een oplossing te zoeken. Immers: als de uitschrijfmethode verandert (bijvoorbeeld niet meer naar het scherm maar naar een uit te printen rapport) dan moet de code op de twee plaatsen veranderd worden: dubbel werk, en dus gevoelig voor fouten. Voor het probleem zijn er twee, fundamenteel verschillende oplossingen. In beide gevallen gaat men zonodig eerst de gemeenschappelijke code in beide klassen in een aparte methode steken met *extraheer methode*, daarna gaat men proberen de twee methodes samen te brengen in een welbepaalde klasse.

– Als de twee klassen een gemeenschappelijke, vrij dichtbij liggende voorouder hebben, dan kan het mogelijk zijn om de code bij deze gemeenschappelijke voorouder te plaatsen. Doe dit alleen als de methode dan ook zinvol is voor de andere deelklassen (zelfs als ze niet wordt gebruikt). Uiteraard hebben in Java alle klassen een gemeenschappelijke voorouder, namelijk de klasse `Object`. Ga het dus niet te ver zoeken.

– Het is ook mogelijk dat de bovenklasse niet bestaat, maar dat het zinvol is ze te creëren. Vermits de bedoeling van herwerken is dat de functionaliteit van de code dezelfde blijft zal dit een abstracte klasse zijn: immers, instantiaties van deze bovenklasse zouden zich anders gedragen dan een object dat beschreven werd door de oude code.

– Als de twee klassen niet nauw verwant zijn is het mogelijk dat ze beide code hebben die aan een derde klasse toebehoort. In ons voorbeeld van de uitschrijfoperatie is het logisch de gemeenschappelijk code onder te brengen in een of andere uitschrijfklasse: de manier van uitschrijven hangt niet zozeer af van de gegevens, maar wel van het uitschrijfmedium. Op deze manier worden uitvoeroperaties gegroepeerd. Elk soort uitvoermedium bepaalt hoe de gegevens vorm krijgen: de klassen die uitvoer leveren dienen alleen te weten dat ze hun data naar een uitvoerobject moeten sturen.

Een tweede probleem is dat van een operatie die veel kijkt naar een ander object dan `*this`. Naar dit object kan verwezen worden door een gegevensveld van de eigen

klasse of door een parameter. Nemen we het voorbeeld van een klasse *Factuur*. In deze klasse is een operatie `berekenprijs(const Artikel&)` die de prijs van een item op de factuur berekent. Deze operatie kijkt uitgebreid naar een object van de klasse `Artikel`. Dit is slechte modulaire scheiding, en waarschijnlijk moet de methode worden ondergebracht bij de klasse `Artikel`.

Als er meerdere operaties zijn van klasse A die veel naar objecten van klasse B kijken en omgekeerd, dan is er iets grondig mis, maar de oplossing van het probleem is niet eenduidig. Soms is het nodig om de twee klassen samen te voegen. Dit is bijna zeker het geval als er een 1-1 associatie tussen de twee klassen bestaat. Soms is het nodig ettelijke methodes en/of gegevensvelden van de ene naar de andere klasse te verhuizen zodat twee minder gekoppelde klassen ontstaan, en wordt het probleem daarmee opgelost.

Een derde probleem is dat sommige klassen de indruk geven niet volledig te zijn: klassen met zeer weinig methodes en/of gegevensvelden. Als er geen andere methodes zijn dan get- en setoperaties dan is er een andere klasse, of zijn er meerdere, die zorgen voor de operaties. Er zijn fundamenteel twee verschillende oplossingen voor dit probleem: ofwel verhuizen we code uit de manipulerende klasse(n) naar de probleemklasse, ofwel schaffen we de probleemklasse af. Bij de eerste oplossing kan het zijn dat een methode volledig wordt overgebracht, of dat gedeelten uit methodes eerst worden geëxtraheerd, waarna de nieuwe methodes worden overgebracht. Zo wordt de probleemklasse een volwaardige klasse. Dit zal vooral het geval zijn als er meer dan een manipulerende klasse is: duidelijk is er een gemeenschappelijke factor in de vorm van de gegevensvelden. Als er maar een manipulerende klasse is dan kan het zijn dat deze oplossing gekozen wordt, maar het kan ook zijn dat de probleemklasse wordt opgeslorpt door de manipulerende klasse. Dit is niet altijd mogelijk: als er een 1-n associatie is tussen manipulerende klasse enerzijds en dataklasse anderzijds, dan is dit opslorpen niet mogelijk, en trouwens ook niet wenselijk. Zeer dikwijls gaat het dan om een associatieklasse. Zoals we gezien hebben wordt een associatieklasse niet verondersteld om echte operaties te hebben. Bovendien bevat ze informatie die essentieel niet hoort bij een object op zich, maar bij twee objecten. Denken we hierbij maar aan de punten die bij een student-vakpaar horen. Zelfs al zouden we in zo'n geval erin slagen om de punten bij een van de twee deelnemende klassen te zetten (als er bijvoorbeeld voor elk vak maar één student is) dan zou dit nog aan de duidelijkheid schaden, en bovendien de flexibiliteit verminderen. Dit is het tegengestelde van de doelstellingen van het herwerken van code.

Bij probleem vier hebben we twee onafhankelijke klassen die gelijkaardige functies vertolken. Met onafhankelijk bedoelen we dat ze niet nauw verwant zijn door overerving. In dit geval is het de zaak om na te gaan of ze niet op een of andere manier samenhoren. Let er eerst op dat operaties van de twee klassen die hetzelfde doen dezelfde naam moeten hebben: zo valt de gelijkenis meer op. Daarna kan men proberen ze in een hiërarchie te krijgen. Het is mogelijk dat de ene klasse een uitbreiding is van de andere, en dus meer eigenschappen heeft. In dat geval wordt ze de deelklasse, en de andere de bovenklasse. Het kan echter ook zijn dat beide klassen een gedeelte hebben dat niet gemeenschappelijk is. In dit geval wordt het gemeenschappelijke stuk ondergebracht in een gemeenschappelijke bovenklasse.

De vijfde groep problemen heeft te maken met delegatie. Dit is een van de fundamentele objectgerichte programmeertechnieken: een object kan verwerking van een

bericht delegeren naar een ander object. We spreken dan van een *delegerend* object en een *uitvoerend* object. Zo kan een delegerend object een aanvraag voor informatie doorspelen naar een ander object dat de informatie heeft, en het resultaat teruggeven aan de oorspronkelijke vrager. Dit is een goed mechanisme, maar het mag niet misbruikt worden. Soms zijn er klassen bij wie zowat alle methodes alleen maar delegeren en die zelf niets doen. Men kan dan de vraag stellen wat de zin is van deze klassen. Al wat ze doen is de berichtenketting langer maken.

Er dient opgemerkt te worden dat het gebeurt dat er klassen zijn die alleen maar delegeren, maar dan wel naar verschillende andere klassen. Dit gebeurt bij het *façade-patroon*. Dit wordt gebruikt om de modulariteit van een geheel te verbeteren. Er wordt een deelsysteem afgescheiden, dat uit een vrij groot aantal klassen kan bestaan. Om hun onderlinge interacties te beperken zet men een façade op, die bestaat uit een of enkele klassen. Hierin staan enkel die operaties van het deelsysteem publiek die belangrijk zijn voor gebruikers *buiten* het deelsysteem, terwijl de klassen van het deelsysteem publieke operaties kunnen hebben die alleen belangrijk zijn *binnen* het deelsysteem. De façade scheidt het deelsysteem af: dit is een vorm van inkapseling, en het is uiteraard niet de bedoeling van deze op te heffen.

Behalve bij een façade is het echter meestal niet de bedoeling om objecten te hebben die niets anders doen dan delegeren. Daarbij moet men rekening houden met wat de delegerende klasse toevoegt aan de delegatie: zijn er niet-delegerende methodes? Zo niet, dan is het waarschijnlijk beter om de klasse te schrappen. Dit houdt in dat in al de code moet gezocht worden naar oproepen naar methodes van de klasse, en dat deze moet omgeleid worden naar het uitvoerende object.

Als de klasse wel een meerwaarde heeft, dan zijn er twee standaardvormen om de delegatie te realiseren. De ene is om de klasse een deelklasse te maken van de uitvoerende klasse. De twee objecten, delegerend object en uitvoerend object, versmelten dus tot één enkel object. De tweede methode is om de originele klasse, of liever zijn objecten te *verpakken* in objecten van een omslagklasse (in het Engels spreekt men van een *wrapper class*). Elk object van de wrapperklasse *bevat* een object van de uitvoerende klasse. Er zijn dan wel twee objecten, de wrapper en het uitvoerend object, maar daarvan is er slechts ’één zichtbaar voor de buitenwereld. Er zijn duidelijke verschillen tussen de twee oplossingen:

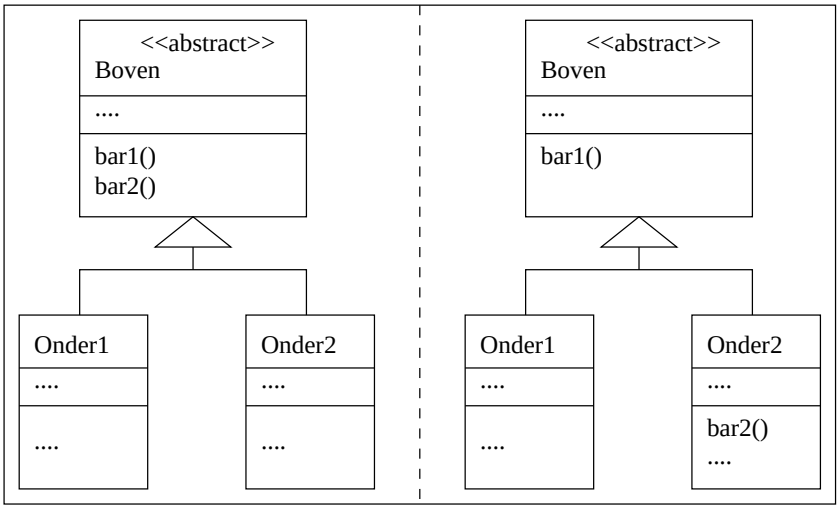
1. Bij (publieke) overerving zijn de originele methodes onmiddellijk beschikbaar voor de buitenwereld.
2. Bij een wrapper speelt het omvattend object de berichten door aan het ondergeschikt object

De tweede vorm wordt normaal gebruikt indien het gedrag van de oorspronkelijke klasse enigszins gewijzigd moet worden. We kunnen bijvoorbeeld een klasse `Telefoonboek` hebben die telefoonnummers teruggeeft. Als we objecten willen die de functies van een telefoonboek hebben met daarnaast ook nieuwe functies (zoals het opzoeken van een e-mailadres) gebruiken we overerving. Als we echter een object nodig hebben dat deze telefoonnummers in een iets andere vorm teruggeeft, dan gebruiken we een wrapper. Wrappers worden niet alleen om een aantal operaties van het omvatte object te wijzigen maar ook om operaties uit te schakelen als we niet alle publieke operaties van de uitvoerende klasse publiek willen maken. Als we een wrapper gebruiken moeten we immers de hele publieke interface expliciet definiëren: operaties

worden overgenomen uit een bovenklasse, maar niet van een deelobject.

Het afschermen van de publieke operaties kan ook gebeuren met private overerving. Private overerving lijkt dan ook meer op wrapping dan op gewone overerving. Het voornammste verschil met wrapping is dat de `protected` lidfuncties van het uitvoerend object beschikbaar zijn voor het delegerend object. De keuze tussen wrapping en private overerving is dus meer een praktische implementatiekeuze dan een echte designkeuze.

Soms worden de problemen veroorzaakt doordat men een verkeerde keuze heeft gemaakt tussen wrapper en overerving. Soms echter ontspruiten de problemen aan een verkeerd ingerichte overerving. Het probleem van een deelklasse die maar een gedeelte van het publieke gedrag van een bovenklasse overneemt, kan dan ook twee verschillende oplossingen krijgen. Het kan het een goed idee zijn deze deelklasse te vervangen door een wrapperklasse. Maar het kan ook zijn dat overerving een goede oplossing is, maar dat er iets mis is met de plaats waar bepaalde eigenschappen gedefinieerd zijn. Hiervoor verwijzen we naar de diagrammen op Figuur 2.2. De beginsituatie zien we



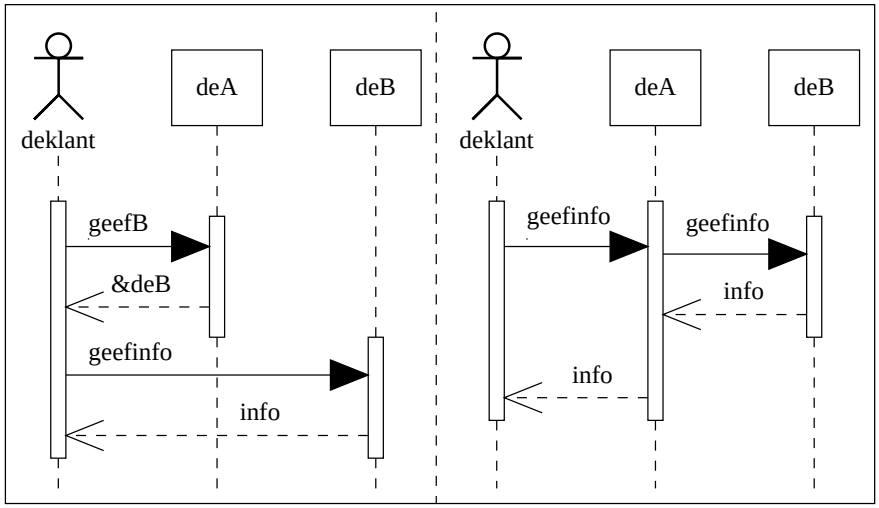
Figuur 2.2. Voor en na het naar beneden duwen van een operatie

aan de linkerkant. We hebben de abstracte klasse `Boven` met daarin twee operaties gedefinieerd. Er zijn twee deelklassen. De tweede deelklasse, `Onder2`, gebruikt beide operaties gedefinieerd in `Boven`. Als we het gebruik van de deelklasse `Onder1` bekijken zien we dat in ons gehele project de operatie `bar2` voor deze klasse nooit wordt opgeroepen en dus dat ze geen enkel nut heeft. Nu maakt `bar2` wel deel uit van de gedefinieerde interface van `Onder1`, waardoor deze nodeloos ingewikkeld is. Op het eerste gezicht, vermits `Onder1` maar een gedeelte van de interface van `Boven` gebruikt, lijkt het een goede oplossing te zijn om van `Onder1` een wrapperklasse te maken.

De juiste oplossing is echter om `bar2` naar beneden te duwen, naar de klasse `Onder2`: op deze manier wordt ze uit `Onder1` verwijderd, en staat ze op een veel logischer plaats.

Naast het gebruik van overerving, wrappers en façades zijn er echter gevallen waarin de delegatie echt is, in de zin dat het delegerende object en het uitvoerende

object duidelijk verschillend van aard zijn. Dit is zeker het geval als de associatie tussen de twee een multipliciteit heeft die verschilt van 1-1. In zo'n geval is het zo goed als onmogelijk om wrapper- en overervingstechnieken te gebruiken. Vaak is het toch de beste strategie om de uitvoerder te verbergen. Bekijken we hiervoor de twee sequentiediagrammen in Figuur 2.3. Aan de linkerkant zien we hoe een cliënt eerst



Figuur 2.3. Het verbergen van de uitvoerder

aan `deA` het adres vraagt van een object `deB`, en daarna aan dit object informatie gaat vragen. Dit vereist veel kennis van de cliënt: hij moet een inzicht hebben in het verband tussen `deA`, `deB` en de info die hij wil. In het tweede diagram is er veel meer ontkoppeling: al wat de cliënt moet weten is dat hij langs `deA` moet passeren om aan zijn informatie te geraken. In beide gevallen moet `deA` `deB` kennen, zodat de grens van `deA` in beide gevallen ongeveer even groot is. In bijna alle gevallen is het rechtse schema te verkiezen boven het linkse.

In sommige gevallen is er een *ketting van berichten*. De cliënt vraagt `deA` het adres van `deB`, en vraagt dan aan `deB` het adres van `deC`, enzovoorts. Het kan zijn dat het verbergen van de uitvoerder voor elke stap de oplossing is, maar als de keten erg lang is dient men zich af te vragen of dit wel correct is. Duidelijk wordt van de cliënt veel kennis verondersteld over de structuur van de verschillende objecten. Uiteindelijk is er aan het einde van de ketting een uitvoerend object. De cliënt kent diens klasse (want hij spreekt dit object direct aan), en meestal is het in dit geval beter om dan de afgeleide verbinding van cliënt naar uitvoerend object een directe verbinding te maken, zodat de hele ketting overbodig wordt. Merk op dat dit de tegengestelde actie is van het verbergen van de uitvoerder.

Ten slotte zijn er nog twee symptomen die voornamelijk opduiken als men reeds bezig is met de software te veranderen. Het eerste symptoom heet hagel verwijderen (shotgun surgery). Dit refereert naar het verschijnsel dat men bij iemand die getroffen is door een schot hagel op verschillende plaatsen loodbolletjes moet verwijderen. Als men een bepaalde aanpassing doet (zoals overschakelen op een nieuwe soort databank), en men moet op veel verschillende plaatsen veranderingen doorvoeren, dan

is dit het teken dat het bepaalde aspect dat men moet veranderen verspreid is over de code. Men gaat dan proberen methodes of datavelden te verplaatsen, en eventueel een klasse laten opslorpen door een andere om dit fenomeen te bedwingen. Nauw verwant is *divergente verandering*. Wanneer men een bepaalde klasse of operatie moet aanpassen bij een aantal niet-verwante veranderingen van de software, dan is het duidelijk dat dit item verschillende taken vervult. Waarschijnlijk is het dan beter om het item (meestal een klasse) op te splitsen in kleinere eenheden.

Veel van de herwerkingsacties zijn erop gericht om de flexibiliteit te verhogen. Hiervoor worden een aantal technieken gebruikt die het aantal elementen doen toenemen, zoals het gebruik van statenklassen. Deze zijn zeer nuttig als ze juist gebruikt worden: ze maken dan duidelijk welk soort flexibiliteit nodig was. Maar het toenemende aantal elementen maakt de software ingewikkelder dan voorheen. In een aantal gevallen zijn de technieken overbodig omdat ze flexibiliteit leveren op een punt waar ze niet gebruikt wordt. We krijgen dan bijvoorbeeld een hele hiërarchie van klassen die van elkaar overerven, maar waarvan er uiteindelijk maar enkele objecten voortbrengen. In dat geval kan het beter zijn om de structuur te vereenvoudigen, zelfs al wordt die daardoor meer rigide.

Geven we een voorbeeld van een herschikking van code die wel volgens de regels gebeurt, en die ook betere code oplevert, maar toch nog een loodzware structuur oplevert. Het voorbeeld komt uit het eerste lange werk over refactoring, en is een beetje aangepast. Het gaat over een menuprogramma. Een gebruiker geeft op welke dagen van de volgende week hij in de bedrijfscafetaria gaat eten, en krijgt dan het menu voor die dagen op het scherm. De interface met de gebruiker is met opzet zo eenvoudig mogelijk gehouden, omdat hij niet essentieel is voor de herwerking van code. De gebruiker schrijft gewoon een programma zoals dit:

```
#include "dagmenu.h"
int main(){
    menu.voegtoe(maandag,28);
    menu.voegtoe(woensdag,30);
    menu.print();
}
```

Als we dan in `dagmenu.h` gaan kijken dan vinden we de volgende code

```
enum dag={maandag,dinsdag,woensdag,donderdag,vrijdag,zaterdag};
class Menudag{
protected:
    int dagvanmaand;
    dag dagvanweek;
public:
    Menudag(dag d, int g):dagvanweek(d),dagvanmaand(g){};
    void printmenu(){
        cout<<dagvanmaand<<": ";
        if (dagvanweek==maandag)
            cout<<"Varkensgebraad met spruitjes"<<endl;
        else if (dagvanweek==donderdag)
            cout<<"Tofuspread met zeewier"<<endl;
```

```
        else
            cout<<"Friet met biefstuk"<<endl;
    }
};
class Menu{
protected:
    vector<Menudag> dagen;
    void voegtoe(dag d,int dagvanmaand){
        dagen.push_back(Menudag(d, dagvanmaand));
    }
    void print(){
        for (vector<Menudag>::iterator it=dagen.begin();
                it < dagen.end();it++)
            it->printmenu();
    }
};
```

De opvallende geur in dit voorbeeld wordt veroorzaakt door de conditionele uitdrukkingen in `Menudag::printmenu()`. Gaat het hier om staten of deelklassen? het laatste, want de toestand van een `Menudag` kan niet veranderen zodanig dat de uitkomst van de condities verandert (de toestand kan helemaal niet veranderen). Er is geen preferente dag die een bovenklasse zou kunnen worden. We maken dus een abstracte bovenklasse met zes deelklassen, een voor elke weekdag. Wat komt er in de bovenklasse? Het gegevensveld `dagvanweek` is overbodig geworden: voor elk van de zes deelklassen ligt de waarde ervan vast. `dagvanmaand` daarentegen blijft gemeenschappelijk. De `printmenu()`-operatie verschilt van deelklasse tot deelklasse: we maken ze dus virtueel, en definiëren ze pas in de deelklasse. We krijgen dan de volgende structuur:

```
class Menudag{
protected:
    int dagvanmaand;
public:
    Menudag(int g):dagvanmaand(g);
    virtual void printmenu()=0;
};
class Maandag:public Menudag{
public:
    Maandag(int g):Menudag(int g);
    void printmenu(){
        cout<<dagvanmaand<<": "
            <<"Varkensgebraad met spruitjes"<<endl;
    }
};
```

waarbij de dagen dinsdag tot en met zaterdag op analoge manier geprogrammeerd worden als maandag. Merk op dat we de `Menu`klasse moeten aanpassen. We kunnen geen vector van `Menudag`en meer gebruiken, maar moeten een vector van `Menudag*`-pointers gebruiken. Dit houdt in dat we bijvoorbeeld een destructor moeten schrijven:

```
    ~Menu(){
        for (vector<Menudag*>::iterator it=dagen.begin();
                it < dagen.end();it++)
            delete *it;
    }
```

Zijn we nu tevreden? Nog niet: in de deelklassen zit, in de functie `printmenu()` heel wat duplicaatcode. Veronderstel dat we de code willen veranderen om alles naar een HTML-pagina te schrijven: dan moeten we in zes klassen code veranderen. Dit is een typisch voorbeeld van hagel verwijderen. We brengen deze code dus samen op een plaats door een methode te extraheren met gemeenschappelijke code. Vermits de klassen met duplicaatcode allemaal een dichte voorouder hebben, `Menudag`, gaan we deze methode in de bovenklasse onderbrengen. We krijgen dus uiteindelijk

```
class Menudag{
protected:
    int dagvanmaand;
    virtual string gerecht()=0;
public:
    Menudag(int g):dagvanmaand(g);
    void printmenu(){
        cout<<dagvanmaand<<": "<<gerecht()<<endl;
};
class Maandag:public Menudag{
public:
    Maandag(int g):Menudag(int g);
protected:
    string gerecht(){
        return "Varkensgebraad met spruitjes";
    }
};
```

Indien we nu de code willen aanpassen om HTML-uitvoer te verkrijgen dan moeten we nog slechts de klasse `Menudag` aanpassen.

Alhoewel er een verbetering is van de kwaliteit van de code, hebben we toch een loodzware structuur gedefinieerd: een klasse met zes deelklassen, een voor elke dag van de week. Uiteindelijk hebben we daarmee ook nog een massa duplicaatcode: de code van de klasse `Maandag` beslaat 8 lijnen, en het enige verschil met de code voor `Dinsdag` is de naam van de klasse, en het gerecht van de dag. Bovendien, als ooit het menu verandert, een niet zo vreemde veronderstelling, moeten we elk van de zes klassen veranderen: duidelijk een geval van hagel verwijderen. In dit geval, hoewel de geur van slechte code zeer duidelijk aanwezig is, is er geen standaard herwerkingsactie die deze geur wegwerkt. De dagschotels voor de zes dagen moeten duidelijk samengebracht worden (hagel verwijderen), en de zes deelklassen zo mogelijk afgeschaft (duplicaatcode verwijderen). Een meer elegante oplossing van het probleem ziet er dan zo uit:

```
typedef int dagvanweek;
```

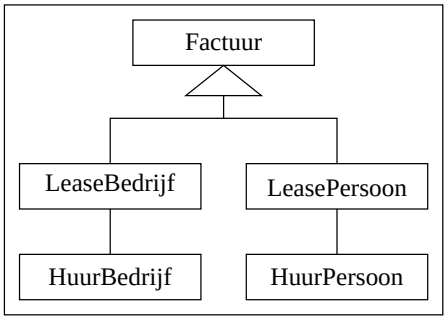
```
const dagvanweek MAANDAG=0;
...
const dagvanweek ZATERDAG=5;
class Menudag{
protected:
    dagvanweek dag;
    int dagvanmaand;
static const string gerecht[5];
public:
    Menudag(dagvanweek _dag, int g):dag(_dag)dagvanmaand(g);
    void printmenu(){
        cout<<dagvanmaand<<": "<<gerecht[dag]<<endl;
};
const string Menudag::gerecht[5]={"Varkensgebraad met spruitjes",
        "Friet met biefstuk", ...
```

En uiteraard zal onze main ook weer moeten aangepast worden: we krijgen terug de oorspronkelijke vorm, behalve dat de dagen van de week nu met hoofdletters geschreven worden.

2.6 PROBLEMEN TUSSEN HIËRARCHIEËN

In de vorige paragrafen bekeken we problemen waarbij hoogstens één hiërarchie van klassen een rol speelde, in deze paragraaf gaan we nog een stapje verder en bekijken we een probleem waarbij de oplossing twee hiërarchieën bevat. De actie heet *het ontwarren van hiërarchieën*, en het symptoom dat aanwijst dat dit nodig is, is dat van *parallelle hiërarchieën*. Nemen we een voorbeeld van een bedrijf dat auto's verhuurt op korte en lange termijn. Voor de facturatie heeft het bedrijf de hiërarchie van facturen die getoond wordt in Figuur 2.4. Er is een verschil in facturen naargelang de factuur wordt opgemaakt op naam van een bedrijf of op naam van een privépersoon, en naargelang het gaat over een lease- of over een huurcontract.

Opvallend aan deze hiërarchie is dat we er niet zinvol één enkele klasse aan kunnen

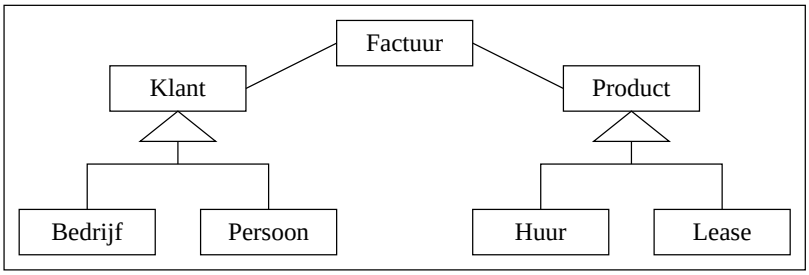


Figuur 2.4. Verstrengelde hiërarchieën

toevoegen: we moeten er altijd verschillende bijmaken, een geval van hagelverwijderen op hoog niveau. Nemen we bijvoorbeeld aan dat het bedrijf ook auto's gaat verko-

pen: dan moet het zowel een klasse `VerkoopBedrijf` als een klasse `VerkoopPersoon` maken. Als er een derde soort klant komt, bijvoorbeeld een vzw. met een ander BTW-stelsel als een gewoon bedrijf, dan moeten we onder `Factuur` een derde tak maken, met als klassen `LeaseVZW` en `HuurVZW`. We hebben hier twee hiërarchieën van klassen die in elkaar verstrengeld zijn geraakt. Merk op dat de ordening van de klassen in de hiërarchie die we hebben onnatuurlijk is, en dat de vier deelklassen dan ook anders zouden kunnen geordend zijn: bijvoorbeeld `HuurBedrijf` en `LeaseBedrijf` naast elkaar, eventueel als onderklassen van een klasse `Bedrijf`.

De oplossing is in dit geval duidelijk. Blijkbaar hangt de facturatie af van twee onafhankelijke gegevens: die van de klant (zoals de aanrekening van BTW, of misschien kortingen voor een categorie) en die van het product. Bijgevolg moeten er *twee* hiërarchieën zijn, een voor elke factor. Het resultaat ziet er dan uit zoals in Figuur 2.5. Merk op dat het mogelijk is dat deze oplossing niet echt voldoet, en dat we



Figuur 2.5. Ontwarde hiërarchieën

de verstrengelde oplossing moeten houden. Dit kan bijvoorbeeld het geval zijn als de prijzen voor producten voor privépersonen en voor bedrijven onafhankelijk van elkaar kunnen veranderen. In dit geval is er essentieel een sterke koppeling tussen klanten en producten in de buitenwereld. Deze koppeling weerspiegelt zich dan in het model van Figuur 2.4.

3.1 EEN VOORBEELD: FOUTAFHANDELING

Stel dat we een systeem ontwikkelen waarin we de volgende beleid uittekenen voor foutafhandeling:

- Er zijn twee soorten fouten die kunnen optreden: onhandelbare fouten en handelbare fouten.
- Beide worden afgehandeld met excepties.
- Onhandelbare fouten leiden tot het afsluiten van de applicatie met een melding die het soort fout aangeeft, evenals de totale oproeplijn (*call stack*), bijvoorbeeld

  ```
  Fatale fout: bestand "input.dat" bestaat niet.
  Fout gegenereerd in Graaf::leesBestand()
  Opgeroepen vanuit TreinSimulator::TreinSimulator()
  Opgeroepen vanuit main()
  ```

  Hiervoor wordt een klasse `OnhandelbareExeptie` gebruikt.
- Handelbare fouten. Voor elk zo'n fout wordt apart een ontwerp gemaakt waarin moet gedefinieerd worden:
  - Wat de naam is van de aparte klasse van exceptie gereserveerd voor de fout.
  - In welke omstandigheden zo een fout optreedt. Op deze manier wordt bepaald in welke lidfuncties van welke klassen de exceptie kan opgeworpen worden.
  - Waar de fout kan opgevangen worden.

De klassieke manier van werken is om dit beleid vast te leggen voor de code geschreven wordt. Bij implementatie dient de programmeur dan steeds in het oog te houden of er een kans is dat er excepties moeten gegenereerd, doorgegeven of opgevangen worden, en veel lidfuncties zullen dan voorzien moeten worden, waarbij veel procedures zullen voorzien worden van code van de vorm

```
try{
...
catch(FataleFout& FF){
    throw FataleFout(FF.what()
           +string("\nopgeroepen vanuit <naam van functie>"));
}
```

Deze manier van werken heeft een aantal nadelen:

1. De programmeur moet herhaaldelijk (bijna) dezelfde code schrijven.
2. De programmeur moet dit beleid steeds in het achterhoofd houden. Dit is het tegendeel van goede modulaire indeling, waarbij men steeds slechts met één ding tegelijk bezig is. Hierdoor vergroot de kans op fouten en weglatingen.
3. Er zijn problemen met hergebruik. Als men code binnentrekt in dit project met een ander beleid voor het aspect foutafhandeling, dan moet men die code aanpassen. Ook als een ander project code uit dit project wil binnenhalen kunnen zo'n aanpassingen nodig zijn.
4. Als men het beleid wil wijzigen dient men alle code te herzien.

Dit laatste komt vaak voor bij de overgang van ontwikkeling naar ingebruikneming. Het beleid zoals hierboven beschreven is aangepast aan ontwikkeling, maar een gewone gebruiker heeft weinig aan het uitschrijven van de *call stack* als er een probleem is. Men zal dan waarschijnlijk fatale fouten indelen in fouten waar de gebruiker iets aan kan verhelpen (bijvoorbeeld: als er een bestand ontbreekt dat door opnieuw installeren van de software teruggezet kan worden), waarbij een advies voor pechverhelping moet gegeven worden, en andere fatale fouten (bijvoorbeeld door bugs in de software) waarbij hem gevraagd wordt om de producent te contacteren.

In de jaren negentig groeide het besef dat dit soort situaties, waarbij een aspect van de te ontwikkelen software een invloed heeft op de code op veel verschillende plaatsen, vrij dikwijls voorkomen en dat de ermee verbonden problemen een negatieve invloed hebben op de kwaliteit van de software en op de efficiëntie van het ontwikkelingsproces. Het aspect foutafhandeling uit het voorbeeld waren we al tegengekomen bij de bespreking van de ontwerpfase, en ook andere aspecten die we daar gezien hebben, zoals beveiliging tegen misbruik en tegen het falen van hard- en software leiden tot analoge complicaties als hierboven beschreven. Ook een aantal andere aspecten werden geïdentificeerd, waarvan sommige alleen in specifieke situaties relevant zijn:

– **Klassenindeling.** Het gaat hier vooral over de indeling van geaggregeeerde structuren, die problemen oplevert bij het doorlopen. We komen hier later op terug.

– **Synchronisatie.** Bij parallelle berekeningen zijn er verschillende processen die een aantal resources delen. Elke keer als een proces zo een resource wil gebruiken, moet er gecontroleerd worden of het proces dit wel kan; zoniet moet het wachten.

– **Gedistribueerde locatie.** Bij een multiprocessorensysteem worden de objecten verdeeld over de verschillende processoren: dergelijke verdeling wordt een *configuratie* genoemd. Soms voorziet het operating system zelf in een configuratie, maar deze is niet altijd optimaal. Hierdoor moet de ontwikkelaar soms zelf aandacht besteden aan de configuratie, in de vorm van code. Deze code heeft hoogstens een indirect verband met de logica van de toepassing (meestal zal de toepassing op een éénprocessorsyteem ook werken, en daar is uiteraard geen configuratieprobleem) maar deze zal ook weer verspreid zitten.

– **Real-Time systemen.** Hierbij moet ervoor gezorgd worden dat bepaalde berekeningen op tijd klaar zijn. Voorbeelden zijn processturing, controle van voertui-

gen, telecommunicatie. Berekeningen die zulke tijdsbeperking hebben moeten prioriteit krijgen over andere. Soms is het natuurlijk onmogelijk om de constraints te eerbiedigen: dit betekent dat het systeem faalt. Een lichtere voorwaarde is dat bepaalde processen moeten starten binnen een bepaald tijdvenster. Zelfs als aan alle andere startvoorwaarden voor het proces voldaan is kan dit betekenen dat het proces moet wachten tot aan zijn laatst toegestane startmoment.

– **Herstel van hardwarefouten.** Waar een algemene back-up- en terugzetprocedure buiten de code valt, zijn er ook maatregelen op een laag niveau. Een voorbeeld daarvan is het gebruik van transacties in een databank.

– **Ontwikkeling versus gebruik.** Soms wil men dat de software zich anders gedraagt bij de ontwikkeling dan bij het dagelijks gebruik. Dit gebeurt in het voorbeeld van de foutafhandeling hierboven. Vaak wil men ook dat, tijdens ontwikkeling en vooral tijdens het debuggen, de software allerhande dingen logt. Dit kan een algemene norm zijn (bijvoorbeeld: de software registreert bepaalde acties), maar dat kan ook zeer tijdelijk zijn (bijvoorbeeld: bij het debuggen is er een probleem met een specifiek gegevensveld. Zolang dit probleem niet is opgelost moet de software een aantal zaken voor dit gegevensveld registreren, zoals de plaats waar het van waarde verandert, daarna is dit overbodig).

Dergelijke aspecten, die invloed hebben op de code op verschillende plaatsen, worden *dwarsaspecten* genoemd, in het Engels *cross concerns*[1].

Men realiseerde zich ook dat het klassieke paradigma van objectgericht ontwerp en objectgericht programmeren niet altijd de oplossing voor dit soort problemen gaf. Wie objectgericht werkt gaat zijn klassen indelen op basis van de realiteit. Dit geeft in het algemeen een indeling die de verschillende aspecten van de toepassingslogica (*business logic*) goed scheidt. De aspecten die problemen opleveren zijn aspecten die verband houden met de programmalogica, en deze staan dwars op de toepassingslogica. Een fijnere opdeling dan de indeling op basis van de toepassingslogica lijkt ook niet veel zin te hebben. Alle voorgestelde oplossingen hebben dan ook één ding gemeen: de behandeling van het dwarsaspect wordt apart gedefinieerd, en dan op een of andere manier verspreid naar verschillende plaatsen in de code. Dit lijkt eigenaardig, maar dit gebeurt ook bij objectgericht programmeren: in een klasse wordt code samengebracht die bij één aspect van de toepassingslogica hoort, maar de werking van de code zit, door het oproepen van lidfuncties van de klasse, verspreid over de hele toepassing.

Er zijn verschillende methodes om de werking van het dwarsaspect te verspreiden. De oudste manier is het werken met *metaobjecten*. Metaobjecten[2] controleren de acties van gewone objecten en passen deze aan. Metaobjecten zijn een bijzonder krachtig instrument, dat veel mogelijkheden biedt, maar daardoor ook vrij ingewikkeld is. Bovendien kan men, gezien de kracht van het instrument, ook veel verkeerds doen.

[1] Een probleem met de vertaling ‘aspect’ voor ‘concern’ zal later duidelijk worden: de term wordt ook gebruikt bij *aspectgericht programmeren*, en daar heeft het weven van aspecten een zeer specifieke betekenis. Een meer neutrale vertaling zou misschien ‘aangelegenheid’ zijn, maar die vlag dekt de lading minder goed.

[2] Meta komt van het Griekse μετα, dat ‘naast’ of ‘achter’ betekent, en op een vreemde manier ‘handelend over’ is gaan betekenen. Vergelijk metafysica, metataal.

Er was daarom behoefte aan meer beperkte methodes, die eenvoudiger toe te passen zijn. De twee belangrijkste zijn

– *Aspectgericht programmeren*.

– Het werken met *samengestelde filters* (Eng.: *composition filters*). Noteer dat dit paradigma soms beschouwd wordt als een speciale vorm van aspectgericht programmeren.

Voor het doorlopen van klassenstructuren heeft men nog een specifieke methode, die we verder zullen bespreken.

Het uitgangspunt is dus code waarin alleen toepassingslogica in verwerkt zit, met daarnaast de definitie van de dwarsaspecten. Er zijn verschillende mogelijkheden om deze twee te combineren:

– De code horend bij de dwarsaspecten wordt *verweven* met de code van de toepassingslogica. Traditioneel gebeurt dit voor het compileren (door een preprocessor zoals *AspectJ*), of tijdens het compileren (zoals bij *OpenC++*). Omdat Java reflectie heeft, blijft informatie zoals klassen- en operatienamen bewaard na compilatie, wat de mogelijkheid opent om bytecode te weven (zoals gebeurt met *Javassist*).

– Tijdens het lopen van het programma. Dit houdt in dat de werking van het programma met toepassingslogica gemonitord wordt door een metaprogramma dat de werking van het toepassingsprogramma kan onderbreken en wijzigen. Deze mogelijkheid wordt vooralsnog weinig of niet gebruikt.

3.2 METAOBJECTEN

In deze paragraaf bekijken we hoe OpenC++ werkt. OpenC++ is een voorbeeld van een systeem waarbij metaobjecten gebruikt worden om code aan te passen bij compilatie. Voor Java is er OJ (vroeger bekend als OpenJava) dat op hetzelfde stramien is opgebouwd.

Bij gebruik van van OpenC++ maakt de programmeur twee programma's: een basisprogramma en een metaprogramma. Het metaprogramma beschrijft hoe het basisprogramma moet aangepast worden, en is zelf geschreven in een uitbreiding van C++. Het gebruik verloopt als volgt:

1. Het metaprogramma wordt door een preprocessor[3] verwerkt, waarbij de uitbreidingen worden omgezet naar standaard C++.
2. Het metaprogramma wordt gecompileerd, waarbij een parser[4] wordt ingelinkt die een C++-programma kan analyseren.
3. Het basisprogramma wordt door de parser verdeeld in kleine eenheden. Tevens bepaalt de parser de structuur van het programma (welke klassen zijn er, welke erven van welke klassen, wat zijn de lidvelden, en zo voorts).
4. Op basis van deze gegevens zet het metaprogramma het basisprogramma om naar standaard C++-code, die gecompileerd wordt door een gewone C++-compiler.

[3] De OpenC++-documentatie spreekt van een compiler, maar deze vertaalt niet naar machinecode.
[4] Zie vorige voetnoot.

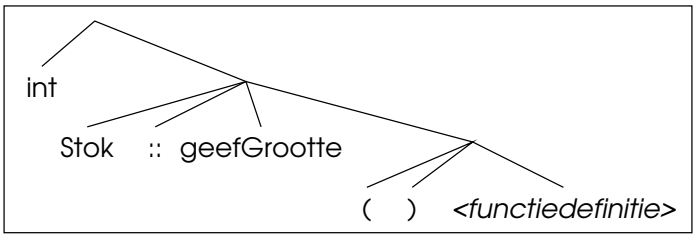
Nemen we het voorbeeld van ons foutafhandelingsbeleid. Om dit te implementeren moeten we een metaprogramma schrijven dat het basisprogramma doorzoekt, en dat bij de daarvoor in aanmerking komende lidfuncties vooraan een `try{`-lijn toevoegt, en achteraan de `catch`-opdracht.

Om te zien hoe het metaprogramma werkt is het nodig om te kijken welke data de parser ter beschikking stelt van dit metaprogramma. Deze zijn:

– De tekst van de code. De parser geeft echter deze tekst niet zomaar door: hij splitst hem op in de verschillende tokens (betekenisvolle stukken zoals operatoren en namen) en geeft deze terug in een boomstructuur (de *parseboom*). Zo zal het codefragment

```
int Stok::geefGrootte()
    <functiedefinitie>
```

ongeveer opgeslagen worden in de boom zoals getekend in Figuur 3.1.



Figuur 3.1. De boomstructuur van een codefragment

– De gegevens over de structuur van de code. Dit houdt in:

1. Een lijst van de klassen.
2. Per klasse: een lijst van alle bovenklassen, van alle onderklassen, van alle gegevensvelden, van alle operaties, en de naam van de metaklasse (zie verder).
3. Als verdere hulp kan voor elk stuk code ook de omgeving (*environment*) worden opgevraagd. Deze bevat een lijst van alle zichtbare variabelen, samen met hun type.

Voor elke klasse in de basiscode is er één metaobject. Dit heeft een dubbele functie: het dient als opslagplaats voor de info over de klasse (puntje 2 uit de opsomming), en het zorgt voor het omzetten van alle code die met die klasse te maken heeft. Een metaobject is een instantiatie van de klasse `Class`, die in OpenC++ gedefinieerd is, of een deelklasse die door de programmeur gedefinieerd is. Deze `Class` en haar deelklassen worden metaklassen genoemd. De programmeur geeft voor elke klasse in het basisprogramma op tot welke metaklasse het metaobject moet behoren met als defaultwaarde *Class*. Objecten van deze metaklasse geven de code behorend bij de klasse ongewijzigd door.

De OpenC++-parser zet het basisprogramma om in een parseboom. Voor elk element van de boom bepaalt hij of het een stuk code betreft dat als één geheel naar een

van de metaobjecten moet gestuurd worden voor vertaling door het oproepen van de passende lidfunctie van het metaobject. Zulke stukken zijn:

– De klassenhoofding. Hierdoor is het mogelijk om lidfuncties en gegevensvelden toe te voegen of te verwijderen, de toegangsrechten (`public, protected, ...`) te wijzigen, de naam of de argumenten van functies te veranderen, of zelfs de hele klasse te schrappen.

– Een definitie van een lidfunctie. Dit zou bijvoorbeeld de plaats zijn waar de `try`- en `catch`opdrachten worden toegevoegd.

– een oproep van een lidfunctie of een gegevensveld in code die tot de eigen klasse of tot een andere klasse behoort. De parser maakt daarbij het onderscheid tussen een gewone lidfunctie, een `new`opdracht, en zo verder.

Samen met het stuk code wordt ook een referentie naar een relevante `Environment` opgestuurd; elk metaobject kan ook de structurele informatie uit de andere metaobjecten raadplegen: de metaklasse `Class` houdt een `static` lijst bij van alle metaobjecten.

Op het eerste zicht is het nut van deze constructie beperkt. Immers, de aanpassing van de code is georganiseerd klasse per klasse, en binnen de klasse is ze opgedeeld naar hoofding/functieoproep/functiedefinitie. Dit is echter maar schijn. Elke klasse uit het basisprogramma heeft wel een eigen metaobject, maar verschillende metaobjecten kunnen tot dezelfde metaklasse behoren, en bovendien kan men door overerving de implementatie van een bepaald gedrag verdelen over verschillende metaklassen, vooral omdat C++ meervoudige overerving ondersteunt. Zo kan het foutafhandelingsbeleid uit ons voorbeeld vrij eenvoudig geïmplementeerd worden door alle metaobjecten te genereren met dezelfde metaklasse.

Het werken met metaobjecten blijkt een zeer krachtige techniek te zijn, die niet alleen voor dwarsaspecten kan gebruikt worden. Voorbeelden van toepassingen:

– Het inbouwen in C++-programma's van reflectie. Het is vrij eenvoudig om in elke klasse als statisch gegevensveld een namenlijst van lidfuncties en gegevensvelden op te nemen, en hierbij de nodige operaties te voorzien.

– Het inbouwen van persistentie. Indien een bepaalde klasse persistente objecten heeft (m.a.w. er is een "record" in de derde laag van ons vijflagenmodel) dan moet elke programma-instantiatie gebonden worden aan zo'n record. Deze record moet ingelezen worden bij creatie van het object, en weggeschreven na wijziging. Ook dit is eenvoudig te implementeren met een passende metaklasse, die deze keer niet met alle klassen wordt verbonden, maar alleen met de persistente klassen.

– Het uitbreiden van de taal, bijvoorbeeld met nieuwe sleutelwoorden, zoals bijvoorbeeld een `foreach(...)`-constructie. Dergelijke constructie kan dan door de vertaalfuncties worden omgezet naar echte C++-code.

Metaobjecten zijn dan wel een krachtig mechanisme, het is ook vrij ingewikkeld. Een van de grootste problemen is hier het manipuleren van code in de vorm van een parseboom. Het vereist heel wat kennis en ervaring om daar vlot mee te kunnen omspringen. Het lijkt moeilijk om dit probleem te omzeilen: de parseboom is nu eenmaal

de natuurlijke weergave van de structuur van code, en andere voorstellingen van deze structuur zullen zeker niet eenvoudiger zijn.

Het blijkt echter dat in de meeste gevallen er geen code moet omgevormd worden: het is voldoende om de code voor dwarsaspecten in te voegen[5], een proces dat 'verweven van code' (Eng: *code weaving*) wordt genoemd, terwijl de in te voegen code *adviescode* genoemd wordt. Een van de belangrijkste eigenschappen van een weefsysteem betreft de plaatsen waar eventueel adviescode kan worden toegevoegd. Dergelijke plaatsen worden verbindingspunten genoemd (Eng: *join points*). Alhoewel details kunnen verschillen, kan men toch typisch twee strategieën onderscheiden:

1. *Black box.* Hierbij worden de basiscodefragmenten in functiedefinities als ontoegankelijk beschouwd. De enige plaatsen waar adviescode kan worden ingevoegd is aan het begin en het einde van de uitvoering van publieke methodes.
2. *White box*, ook *glass box* of *clear box* genoemd. Adviescode kan ook intern binnen functiedefinities worden toegevoegd.

Het is duidelijk dat een whiteboxstrategie meer mogelijkheden biedt dan een blackboxmethode. Dit betekent echter niet dat ze alleen maar voordelen biedt. Zoals gezegd is het metaobjectprotocol (MOP) te algemeen om echt efficiënt het probleem van dwarsaspecten aan te pakken: hoe meer mogelijkheden, hoe complexer de methode. Black box is beperkter en dus eenvoudiger.

Bovendien is er de modulariteitseis. Bij een whiteboxmethode is er meer kans dat een programmeur een slechte afscheiding maakt tussen basiscode en adviescode, door bijvoorbeeld delen van de toepassingslogica onder te brengen in adviescode. Hierdoor ontstaat een aaneenhangend geheel, waarbij er geen wijzigingen aan de ene kant kunnen worden aangebracht zonder aanpassingen aan de andere kant. Bij een black box is dit weinig waarschijnlijk: adviescode wordt aangebracht rond de objecten, en gaat uit van de interface tussen deze objecten en de buitenwereld. Daarvoor is enkel het contract tussen object en buitenwereld belangrijk, en dit contract hangt niet af van de implementatie. Bovendien kunnen blackboxmethodes in een aantal gevallen gebruikt worden zonder dat men beschikking heeft over de broncode. Bij het debuggen van software zal men over het algemeen de voorkeur geven aan een whiteboxmethode. Debuggen is zelden zinvol zonder broncode, en de adviescode wordt slechts tijdelijk aangebracht voor debugspecifieke doeleinden (voornamelijk logging en het nagaan van de juistheid van de toestand van het systeem) die (bijna) nooit interfereren met de toepassingslogica.

Als typevoorbeeld van blackboxmethodes bekijken we samengestelde filters, terwijl we voor whiteboxtechnieken gaan kijken naar AspectJ/AspectC++, een van de meest verspreide tools voor aspectgericht programmeren.

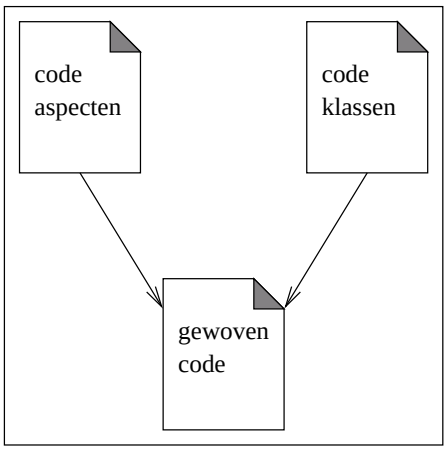
3.3 ASPECTGEÖRIËNTEERD PROGRAMMEREN

AspectJ en AspectC++ zijn beide pakketten voor aspectgericht programmeren waarbij de aspectcode toegevoegd wordt aan de basiscode voor compilatie[6]. Dit proces heet het

[5] Bij run-time weven wordt de adviescode niet echt ingevoegd, maar wel tussen de basiscode door uitgevoerd.

[6] AspectJ gebruikt een gemengd model, waarbij zowel naar sourcecode als naar bytecode kan gekeken

verweven van de code. Het resultaat is standaardcode die door een gewone compiler verwerkt wordt. In tegenstelling tot technieken die metaobjecten of filters gebruiken,



Figuur 3.2. Verweven aspecten.

waarbij de extra code geordend is rond de klassen van de basiscode, wordt hier de extra code ingedeeld op een manier die meer overeenkomt met de idee van dwarsaspecten die weinig te maken hebben met de klassenindeling. De basiseenheid van de aspectcode is een *aspect*. Noteer dat de term aspect hier een minder ruime betekenis heeft dan wat we tot nu toe gebruikt hebben. Een dwarsaspect zoals foutafhandeling is een vrij groot geheel, een AspectJ- of AspectC++-aspect behandelt slechts toevoeging van één soort code.

Gelijkaardige code, die op verschillende plaatsen in de originele code moet ingevoegd worden, dient op deze manier slechts op één plaats beschreven te worden. Dit gebeurt in een aspect. Een aspect bestaat uit verschillende delen:

1. De *pointcut*. Dit is een beschrijving van alle relevante *join points*. Dit zijn de punten in de originele code waar code moet toegevoegd worden. Een aantal mogelijke join points:
   - De *oproep* van een lidfunctie. De code wordt dan toegevoegd in de code van waaruit de lidfunctie wordt opgeroepen.
   - De *uitvoering* van een lidfunctie. De code wordt dan toegevoegd aan de lidfunctie zelf, in de klasse die eigenaar is van de lidfunctie. Code wordt toegevoegd aan het begin en/of aan het einde van de lidfunctie.
   - Wijziging of lezen van publieke variabelen.
2. Een advies (*advice*). Een advies is de code die moet ingevoegd moet worden bij elk join point van een pointcut.

Een pointcut kan worden gedefinieerd aan de hand van een aantal basiskeuzes (zogenaamde *primitive pointcuts*). Een eenvoudig voorbeeld in AspectJ:

worden voor het weven. Bovendien wordt in recentere versies *load-time weaving* ondersteund, waarbij de bytecode van een klasse pas geweven wordt als de klasse geladen wordt bij uitvoering van de toepassing. AspectC++ werkt zuiver als een preprocessor. In de rest van de tekst gaan we uit van het preprocessormodel.

```
pointcut services(Server s): target(s) && call(public * *(..))
```

`call(public * *(..))` is de pointcut die alle joinpoints selecteert waarbij een publieke functie wordt opgeroepen, terwijl `target(s)` alle joinpoints selecteert die horen bij de `Server`klasse. Zoals het voorbeeld doet vermoeden, kan men pointcuts combineren d.m.v. logische operatoren zoals EN, OF en NIET. `services` is de naam die de programmeur geeft aan deze pointcut. Het gebruik van een argumentenlijst is enigszins anders dan bij gewoon programmeren in Java. de uitdrukking `(Server s)` betekent *niet* dat de pointcut `services` kan opgeroepen worden met een argument, ze betekent *wel* dat het doel `s` kan gebruikt worden bij het definiëren van het `advice`: het advies kan gebruik maken van de argumenten in de pointcut.

Zoals te zien is aan de uitdrukking `call(public * *(..))` kan er gebruik worden gemaakt van wildcards. Advies kan code geven die uitgevoerd moet worden vlak *voor* het join point uitgevoerd wordt (**before()**), *na* het join point (**after()**), en *in plaats van* het join point (**around()**). De naam 'around' voor een advies dat uitgevoerd wordt in plaats van het join point mag eigenaardig lijken, maar het join point zelf kan opgeroepen worden vanuit het advies. Naast de adviescode die verweven wordt met de basiscode kan een aspect ook eigen lidvelden bevatten. In dit opzicht gedraagt het zich als een klasse, waarvan automatisch een (naamloos) object wordt gecreëerd. In het volgende voorbeeld (uit de documentatie van AspectC++) wordt een aspect gedefinieerd dat voor een bepaalde klasse nagaat hoe vaak constructor en destructor wordt opgeroepen. Dit is een handig hulpmiddel bij om zogenaamde *memory leaks* op te sporen, waarbij gereserveerd geheugen niet wordt vrijgegeven.

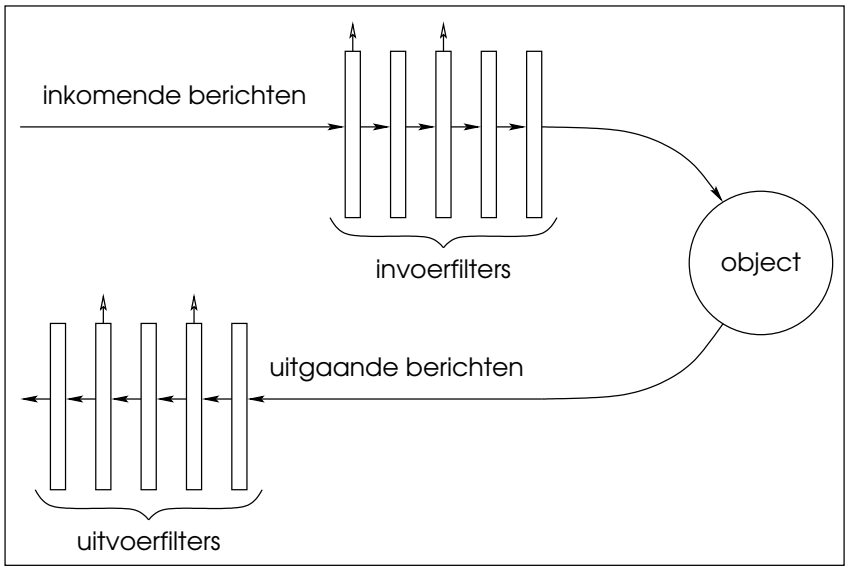
```
aspect InstanceCounting {
// the class for which instances should be counted
    pointcut observed() = "ClassOfInterest";
// count constructions and destructions
    advice construction (observed ()) :
        before () { _created++; }
    advice destruction (observed ()) :
        after () { _destroyed++; }
    public:
        InstanceCounting () {
             _created = _destroyed = 0;
        }
    private: // counters
        int _created; int _destroyed;
};
```

Bij AspectC++ is er de mogelijkheid om naast advies ook *slices* te definiëren. Een advies wordt uitgevoerd bij een bepaalde gebeurtenis (wat ook bij run-timeweven zinvol kan geïnterpreteerd worden), een slice voegt code toe op een bepaald punt in de basiscode. Zo kan een klassenhoofding worden uitgebreid met extra lidvelden. Bij het gebruik van metaobjecten hebben we gezien dat we op deze manier reflectie konden invoeren in C++. AspectC++ is in dit opzicht minder krachtig, omdat we niet de volledige broncode kunnen analyseren: dit maakt het moeilijk om bijvoorbeeld een lijst van lidfuncties in de slice op te nemen.

3.4 SAMENGESTELDE FILTERS

Deze manier van werken kan beschouwd worden als een verdere beperking van het metaobjectmodel. Waar we bij aspectgericht programmeren nog een vrij ruime keuze hebben wat betreft het toe te voegen advies, is dit bij filters ingekrompen tot een zeer beperkt aantal standaardacties. Bij het filtermodel wordt elk object voorzien van twee extra soorten van eigenschappen. Naast de klassieke gegevensvelden (attributen en associaties; deze zijn verborgen voor de buitenwereld) en de operaties wordt het object voorzien van

1. **Condities**. Condities zijn zuivere query's (zij hebben dus geen neveneffecten zoals verandering van toestand of het zenden van berichten naar andere objecten). In de meeste modellen wordt aangenomen dat ze een logische waarde teruggeven, maar dat is niet absoluut noodzakelijk.
2. **Filters**. Er zijn invoer- en uitvoerfilters.



Figuur 3.3. Het filtermodel

Een inkomend bericht (meestal een functieoproep) wordt niet onmiddellijk door het object verwerkt. In plaats daarvan wordt het door de (geordende) reeks invoerfilters gestuurd. Uitgaande berichten gaan door een reeks uitvoerfilters. Een filter kan een boodschap *accepteren* of *verwerpen*. Dit gebeurt op basis van de eigenschappen van het bericht (zoals afzender en bestemmeling, waarde van parameters, en zo verder) en op basis van de toestand van het object (die de filter opvraagt via de condities). Op basis van acceptatie of verwerping beslist de filter of hij zelf iets doet met de boodschap dan wel ze doorstuurt naar de volgende filter. Vreemd genoeg sturen sommige filters alleen de boodschappen die ze verwerpen naar de volgende filter, terwijl andere alleen berichten die ze accepteren doorsturen. De derde mogelijkheid is dat de filter alleen bij acceptatie of alleen bij verwerping een bepaalde actie uitvoert, en het daarna doorstuurt. Dit is het geval bij de hierna vermelde acties *vervangen*, *wachten* en *meta*.

De meest voorkomende acties die ondernomen worden zijn:

1. **Doorsturen.** Het bericht wordt doorgestuurd naar zijn bestemmeling (en dus niet naar de volgende filter).
2. **Vervangen.** Het bericht wordt gewijzigd, en dan doorgestuurd naar de volgende filter.
3. **Foutopvangen.** Genereert een exceptie.
4. **Wachten.** Het bericht wordt bijgehouden en periodiek worden de voorwaarden voor verwerping/acceptatie gecontroleerd, tot deze veranderen. Dan wordt het object doorgestuurd naar de volgende filter.
5. **Meta.** Het bericht wordt *gereïficeerd*. Dit wil zeggen dat er een object gemaakt wordt waarvan de attributen de eigenschappen van het bericht weergeven. Dit object wordt doorgestuurd naar een verwerkend object (dat binnen of buiten de filter gedefinieerd wordt). Dit kan bepaalde acties uitvoeren (bijvoorbeeld: loggen, aanpassen, ...). Daarna gaat het bericht naar de volgende filter.

Deze acties kunnen gecombineerd worden. Zo kan één filter een bericht wijzigen door de bestemmeling te veranderen, en dan het bericht doorsturen naar de nieuwe bestemmeling. Dit heeft natuurlijk alleen maar zin als de nieuwe bestemmeling een passende operatie heeft om het bericht op te vangen. Als een bericht na de laatste filter nog niet verwerkt is, wordt er een *FilterOntbeektExceptie* opgeworpen.

Uit dit overzicht blijkt dat er een vrij grote koppeling is tussen het object en de filters, in de vorm van de condities. Dit lijkt op het eerste gezicht niet wenselijk: de programmeur van de filters moet een aantal specifieke eigenschappen van het object kennen, en de programmeur van het object moeten weten welke condities er nodig zijn voor de filters. Nu is het echter niet nodig deze koppeling te gebruiken (ook bij AspectC++ kan dergelijke koppeling gemaakt worden, en nog in meer algemene zin, door lidfuncties van de basisobjecten op te roepen in de aspectcode). Bij een goed gebruik van de filtermethodiek wordt echter meestal alleen de *staat* van het object, zoals gedefinieerd in het statendiagram, opgevraagd. Deze staat is gedefinieerd bij het ontwerp van de objecten (en is dus gekend door de objectprogrammeur zonder rekening te houden met de filters).

Als voorbeeld van toepassing kunnen we kijken hoe er een controle kan ingebouwd worden dat de code aan bepaalde elementen van het ontwerp voldoet.

Een eerste controle is die op informatie bevat in de CRC-kaarten. Deze geeft, voor een klasse, niet alleen de operaties en de gegevensvelden, maar geeft ook aan welke operaties uit andere klassen mogen opgeroepen worden. De implementaties verschillen bij AspectC++-achtig aspectgericht programmeren en bij samengestelde filters:

– Aspectgericht: definieer voor elke klasse een CRC-aspect. Als pointcut heeft dit alle functieoproepen binnen code van de klasse die niet beantwoorden aan de lijst van de CRC-kaart (dit is gemakkelijk te implementeren: men maakt een pointcut van alle oproepen die wel aan de lijst beantwoorden door opsomming van de lijst, en neemt de negatie van deze pointcut). Het uit te voeren advies is dan het opwerpen van een `CRCOvertredingsExceptie`[7].

[7] AspectC++ heeft een mechanisme om bij verwerven van de code gewoon een foutmelding te geven, zodat de code niet verder wordt opgebouwd. Bij run-timeverweving is het melden van de fout bij

– Filters: men maakt een uitvoerfilter die een fout signaleert als het uitgaande bericht niet op de CRC-lijst staat.

Een tweede controle is die op het statenmodel. Hier is het natuurlijk nodig dat de staten van de objecten kunnen gecontroleerd worden. Bij het filtermodel wordt dit expliciet gedaan in de vorm van de condities, maar bij klassieke aspecten is het ook nuttig een of meerdere query's te hebben die de staat naar buiten brengen. Er is controle op twee zaken:

1. Sommige operaties mogen niet uitgevoerd worden in bepaalde staten. Dit is een controle die zeer gelijkaardig is aan de vorige, en de implementatie is analoog.
2. In een statendiagram wordt aangegeven door welke acties, en onder welke voorwaarden, de staat verandert. Met het `around()`-advies van een aspect is gemakkelijk te controleren of de verandering van staat inderdaad plaatsvindt. Bij de filtermethode is dit iets ingewikkelder: de meest zekere manier is om zowel een invoer- als een uitvoerfilter te maken die de `Meta`-actie implementeert, en de berichten naar een statencontroleobject stuurt.

Merk op dat deze controles niet kunnen uitgevoerd worden bij compilatie, omdat de toestand, en dus de staat, van een object dan niet gedefinieerd is.

3.5 DOORLOPEN VAN STRUCTUREN

Een van de terreinen waar aspecten vaak verweven raken is het doorlopen van complexe (data)structuren. Als voorbeeld nemen we een XML-schema.

Even kort samenvatten wat voor ons voorbeeld de essentie is van XML (waarbij we een aantal elementen weglaten, zoals de elementen in een XML-bestand die het bestand zelf beschrijven, en niet de gegevens die het bestand bevat). Elk XML-bestand bevat de gegevens van één element. Een element, per definitie, bestaat uit

– Nul of meerdere attributen. Een attribuut bestaat uit een naam en een waarde (beide strings). Twee attributen van hetzelfde element moeten een verschillende naam hebben.

– Een tekst (die de lege string kan zijn).

– Andere elementen. Twee elementen in hetzelfde element mogen dezelfde naam hebben.

Zo krijgen we een boomstructuur van elementen, met in elke knoop van de boom mogelijks tekst en/of attributen. Als voorbeeld nemen we een bestand met info over de programma's van een tv-zender voor een bepaald week: het element is de weekprogrammatie, dat zeven elementen bevat (een voor elke weekdag), die elk dan weer een aantal elementen bevatten (een voor elk programma), waarbij elk programma misschien nog verdere elementen kan bevatten.

Soms willen we de structuur van een XML-bestand beschrijven, bijvoorbeeld als we een programmeur willen uitleggen dat ons bestand (of juister, het rootelement ervan) altijd zeven deelelementen zal bevatten, dat elk programma een beginuur heeft

compilatie niet mogelijk.

(met uitleg over de juiste notatie: “20 uur 00” moet anders ingelezen worden dan “20::00”), en zo voorts. Dit kunnen we doen met een XML-schema (er zijn ook DTD’s, Document Type Definitions). Ruwweg is een typebeschrijving (`TypeDef`) een beschrijving van één element: welke attributen er (kunnen) zijn, welke deelelementen (met eventuele volgorde, multipliciteit), wat voor tekst. Deelelementen kunnen beschreven worden met een `TypeDef`, attributen met een `AttributeDeclaration`. Een type dat noch attributen noch deelelementen bevat is een `SimpleType`, anders hebben we een `ComplexType`. Een `ComplexType` heeft een `SequenceGroup`, een lijst van elementbeschrijvingen, en een lijst van attribuutbeschrijvingen. Een schema bestaat nu uit een aantal `TypeDef`-elementen (met als naam `ComplexType` of `SimpleType`) en een `ElementDeclaration` van het rootelement van een databestand (alles wat geen `TypeDef`-element is, en waarin de `TypeDefs` kunnen gebruikt worden[8]). Alles samen krijgen we het volgende klassenschema (gebaseerd op [3], hertekend om het duidelijker te maken): De operaties aangegeven in dit diagram zullen later worden uitgelegd. Het eerste wat aan dit klassendiagram opvalt is dat het zo goed als onontwarbaar lijkt, alhoewel het aantal klassen (10) voor één diagram nog net haalbaar is. Dit komt omdat er drie aspecten verweven zijn:

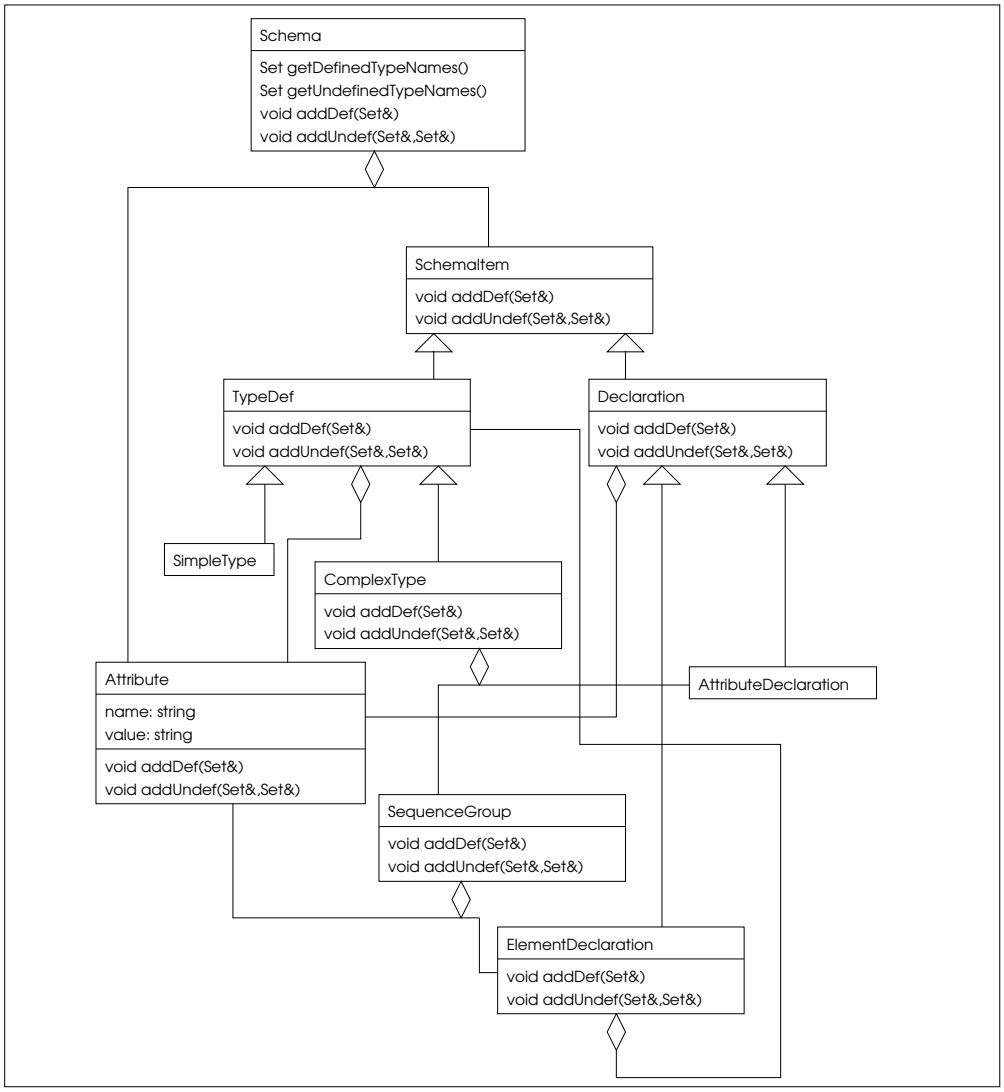
– Het aspect `Attribute`, duidelijk een algemene dienstmodule die niet thuis hoort in de klassenindeling die aspecten moet scheiden. Objecten van `Schema`, `TypeDef`, `SequenceGroup` en `Declaration` kunnen alle een onbepaald aantal `Attribute`s bevatten.

– Het aspect *aggregatie*: wat maakt deel uit van wat?

– Het aspect *abstractie*: welke klasse is deelklasse van welke?

Als we de laatste twee aspecten apart op een diagram zetten wordt plotseling veel duidelijk: Merk op dat, zoals gezegd, een `Schema` verschillende `ElementDeclarations` kan bevatten, wat een overbodige complicatie is die we dan ook verder niet meer bekijken. Nu dit probleem uit de weg is kunnen we terugkomen op ons hoofdprobleem: dit van het doorlopen van een datastructuur. Zoals gezegd kan een XML-schema een aantal `TypeDef`s bevatten, elk met een naam (in XML geïmplementeerd doordat de `Typedef` een `Attribute attr` bevat met `attr.name="name"`, waar `attr.value` dan de naam aangeeft). Ook kan een element beschreven worden door op te geven dat het aan een `TypeDef` voldoet. Dit wordt geïmplementeerd door het element een `Attribute attr` te geven met `attr.name="type"`, waarbij `attr.value` de naam van de `TypeDef` is. Het gegeven probleem is nu om te ontdekken of er in een bepaald schema types gebruikt worden die niet in het schema gedefinieerd zijn. Bij klassiek objectgericht programmeren krijgen we de code die volgt uit het klassenschema uit Figuur 3.4. Eerst worden met de `addDef()`-functies alle typedefinities in het schema opgehaald, daarna worden met de `addUndef()`-functies plaatsen opgehaald waarin een typedefinitie gebruikt wordt die niet in de set van gedefinieerde namen zit (de meer volledige signatuur van de `addUndef()`-functie is `addUndef(const Set& definedTypes, Set& undefinedTypes)`). De meeste `addDef`functies doen niets

[8] Omdat dit te eenvoudig was, heeft men beslist dat een schema *verscheidene* `ElementDeclaration`s kan bevatten. De laatste `ElementDeclaration` beschrijft het rootelement van het databestand dat we willen beschrijven, de voorgaande doen dienst als `TypeDef`s.



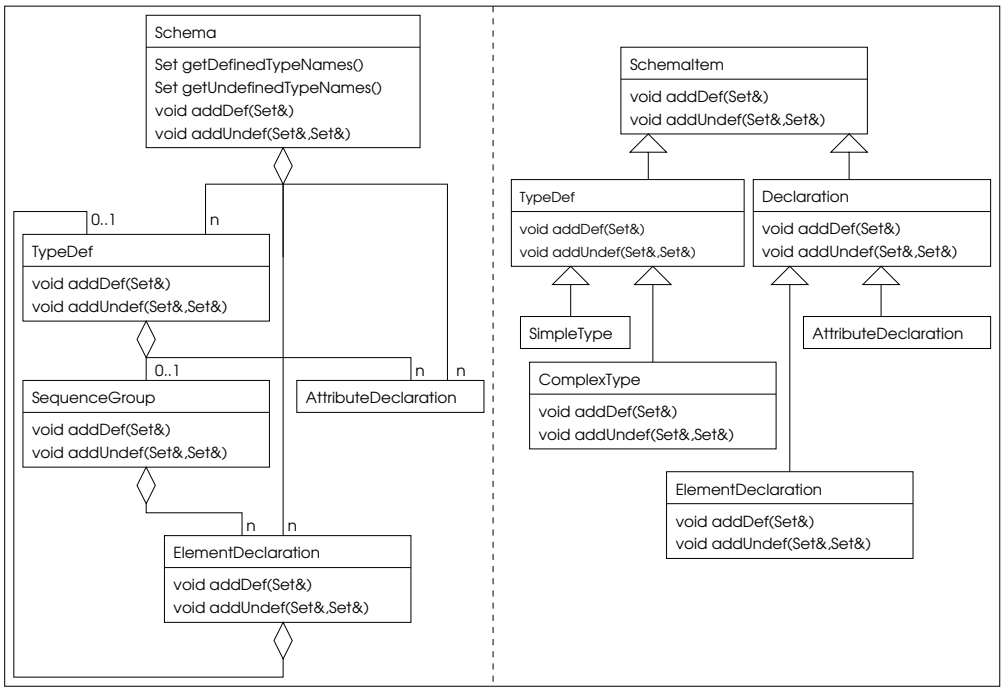
Figuur 3.4. Klassendiagram voor XML-schema's

anders dan andere `addDef`functies oproepen: alleen de `TypeDef::addDef`functie voegt iets toe aan de `Set`.

Bekijken we in meer detail het ophalen van de gedefinieerde types. Dit ophalen heeft twee regels:

1. Als een object een `TypeDef` is, dan zet het zijn naam in de verzameling gedefinieerde types.
2. Elk object geeft de navraag door aan al zijn delen die `TypeDef`s kunnen bevatten, of zelf een `TypeDef` zijn.

Het implementeren van deze twee regels heeft echter tot gevolg dat we een stuk of vijf `addDef`methodes moeten implementeren: duidelijk het hagelverwijderensyndroom dat typisch is voor een dwarsaspect. Wel blijft dit dwarsaspect beperkt tot één enkele



Figuur 3.5. Scheiding van de aspecten aggregatie en abstractie.

samengestelde structuur, in dit geval een `Schema`.

We hebben gezien dat delegatie gebruikt wordt om de uitvoerder te verbergen: het aanvragend object kan een vraag stellen aan een delegerend object, zonder te weten dat dit delegerend object niet zelf de gevraagde dienst levert. Deze overweging heeft geleid tot het opstellen van de zogenaamde *wet van Demeter*. Deze wet stelt dat een methode van een bepaald object `self` alleen methodes mag oproepen van vier groepen objecten:

1. `self` zelf.
2. Objecten doorgegeven met de parameters van de methode.
3. Objecten gecreëerd binnen de methode.
4. Objecten geassocieerd aan `self`, door een gewone associatie of door aggregatie.

De constructie getoond aan de linkerkant van Figuur 2.3 op pagina 26 overtreedt duidelijk deze wet.

In veel gevallen is het echter zo dat het aanvragend object niet zozeer onwetend is over *welk* object uiteindelijk de dienst moet leveren, maar wel over *waar* het object zich bevindt. In veel gevallen (zoals in ons voorbeeld) gaat het ook over meer dan een uitvoerend object: we moeten alle `TypeDef`s vragen hoe ze heten. En eigenlijk zijn er twee verweven aspecten:

1. Waar zitten de objecten die de dienst moeten leveren?
2. Wat moet er met die objecten gebeuren?

Hoezeer deze aspecten loodrecht op elkaar staan wordt duidelijk als we één van de twee wijzigen, en het andere constant laten.

1. Als we de aggregatiestructuur wijzigen (onderstel bijvoorbeeld dat we binnen een `AttributeDeclaration` ook een `TypeDef` zouden hebben), dan moeten we `addDef` functies toevoegen in bepaalde elementen, en de andere `addDef`functies aanpassen om in de goede richting te gaan. De enige `addDef`functie die echt iets doet is deze van de `TypeDef`, en de actie (naam toevoegen aan de `set`) verandert niet.
2. Onderstel dat we de aggregatiestructuur constant houden, maar dat we iets anders willen vragen aan de `TypeDef`s, bijvoorbeeld dat we het totaal aantal `Attribute`s willen weten in alle `TypeDef`s samen. Dan moeten we naast elke `addDef`functie een `countTypeAttr`functie zetten. De delegerende `countTypeAttr`s zullen exact dezelfde structuur hebben als de delegerende `addDef`s.

In het eerste geval hebben we een hagelprobleem, in het tweede duplicaatcode.

Het tweede probleem is vrij eenvoudig op te lossen door de twee aspecten te scheiden zodat het doorloopaspect gescheiden wordt van het andere. Hiervoor zijn twee fundamenteel verschillende manieren in gebruik.

1. Het gebruik van *iteratoren*. Een iterator is een object dat referenties naar elementen in een bepaalde verzameling één na één teruggeeft. De index van een element in een tabel kan als een soort iterator beschouwd worden: als we een tabel doorlopen en elk element een functie `foo` laten uitvoeren schrijven we

   ```
   for (int i=0; i<n ;i++)
         tab[i].foo();
   ```

   We kunnen nu een klasse `TypeDefIterator` definiëren die alle `TypeDef`s in een `Schema` opzoekt zodat de volgende code zinvol wordt:

   ```
   for (TypeDefIterator it=hetschema.begin();@
                        it!=hetschema.end(); it++)@
        it->addDef(definedTypes);@
   ```

   We kunnen zulk een iterator gebruiken, zowel voor de `addDef`- als voor de `countTypeAttr`functies. Iteratoren worden veel gebruikt bij klassieke containers.
2. Het gebruik van `Visitor`s en doorloopfuncties[9]. Een iterator brengt een referentie van elk object in de verzameling naar buiten, een doorloopfunctie brengt een object van buiten (de visitor) bij elk object in de structuur. In ons voorbeeld krijgen we een doorloopfunctie `Schema::bezoekTypeDefs(Visitor)`, en definiëren we een `AddDefVisitor` en een `CountAttrVisitor`, beide als deelklassen van `Visitor`.

Essentieel voor beide oplossingen is dat de iterator of de doorloopfunctie toegang kan krijgen tot de interne structuur van alle objecten van de aggregatie. Dit kunnen we implementeren, ofwel door de structuur bloot te stellen (in C++ is er daarvoor het typewoord `friend`: als we een iterator als `friend` van een klasse declareren, kan hij aan de `private` lidvelden), of door de objecten in de aggregatie lidfuncties te geven die de nodige informatie kunnen doorgeven. Het is duidelijk dat, bij beide oplossingen, alle code overeen moet komen met de aggregatiestructuur, en dat het schrijven van die code

[9] Zoals bij doorloopaspect ligt de nadruk bij doorloopfunctie op de tweede lettergreep.

geen denkwerk van de programmeur vereist, alleen het toepassen van de regels. Het is dus een taak kan geautomatiseerd worden, en waarbij automatisatie de kans op fouten verkleint. Bovendien moeten we naargelang van de gekozen oplossing veranderingen aanbrengen in de `TypeDefIterator` dan wel in de `bezoekTypeDefsFunctie` als we de aggregatiestructuur aanpassen, ook iets dat best automatisch gebeurt.

Vermits we de aggregatiestructuur moeten kennen zijn de geziene methodes van aspectgericht programmeren en van samengestelde filters te beperkt voor dit probleem. Metaobjecten hebben echter de beschikking over de nodige structuurinformatie, en het is dan ook mogelijk het probleem daarmee op te lossen. Gezien de specifiteit van het probleem heeft men ook echter tools kunnen ontwikkelen die deze taak aankunnen (zie bijvoorbeeld [3]). Deze laten toe om de te bezoeken verzameling in een beperkte formele taal te beschrijven (bijvoorbeeld: bezoek alle `TypeDef`s die door aggregatie in een `Schema` zitten, maar niet deze die deel zijn van een `ElementDeclaration`).

[1] **Shigeru Chiba:** A Metaobject Protocol for C++, Proceedings OOPSLA 1995, pp. 285–299

[2] **R.E. Filman and D.P. Friedman:** Aspect-Oriented Programming is Quantification and Obliviousness, Workshop on Advanced Separation of Concerns, OOPSLA 2000.

[3] **Doug Orleans , Karl J. Lieberherr:** DJ: Dynamic Adaptive Programming in Java, Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, September 25-28, 2001, p.73-80.