Design af genbrugeligt objektorienteret software

Evaluering af software ved hjælp af statiske mål.

Speciale
Januar 2004

af: **Søren Gaardbo Jensen** <soren@gaardbo.dk>

Vejleder: Eric Jul

DIKU, Datalogisk Institut
Københavns Universitet

Resumé

Dette speciale omhandler design af objektorienteret software, som imødekommer genbrug og vedligeholdelse. En række softwaremål undersøges, med det formål at finde statiske mål, der kan måle nogen af de egenskaber, der menes at karakterisere et design som beskrevet ovenfor. Karakteristikken af det gode design findes i gennemgangen af tretten designprincipper, som sammenholdes med softwaremålene. Der blev konstrueret et nyt mål der måler graden af brug af interfaces i Javakildekode.

Der er en sammenhæng mellem overholdelse af designprincipperne og softwaremålene. Det konkluderes at det kan lade sig gøre at foretage en automatisk evaluering af software, som kan identificere klasser der potentielt er dårligt designet.

Abstract

This thesis deals with the design of object oriented software, that meet requirements of reusability and maintainability. Some software metrics are investigated, in the quest of finding static measures, that will be able measure properties of software that are important to reusability and maintainability. The characterization of good design is sought in an investigation of thirteen design principles, which are compared to the design measures. A new metric was constructed, that measures the degree of use of interfaces in the Java programming language.

There is a connection between the application of the design principles, and the software measures. It is concluded, that it is possible to automatically evaluate software in the purpose of identifying potentially badly designed classes.

Forord

Dette speciale er blevet til i perioden fra marts 2003 til januar 2004. Emnet, *design af genbrugeligt og vedligeholdelsesvenligt software*, er inspireret af kontakt med mængder af dårligt designet software. Denne software var umulig at vedligeholde og håbløs at forsøge at genbruge. Noget af denne software var mit eget. Andre gange måtte jeg forbløffes over smuk og velstruktureret kode. Let at vedligeholde og mulig at genbruge. Sjældent var dette software mit eget.

En undersøgelse af hvad der gør software genbrugeligt og vedligeholdelsesvenligt, samt muligheden for automatisk at afgøre dette, var derfor nærliggende.

Flere personer har været involveret i dette projekt. Jeg vil derfor takke min vejleder Eric Jul, for gode råd og for at give mig mulighed for at deltage på ECOOP 2003. Jeg vil også gerne takke min kæreste, Kira, dels for korrekturlæsning og dels for overbærenhed i travle perioder. Tak til Mikkel Rasmussen for korrektur og uvurderlige kommentarer, til Robert Bialek for korrektur og gode diskusioner og til Frank Grandjean for lån af ekstra hardware, da ressourcerne blev små.

København, 15 januar 2004.

Søren Gaardbo Jensen

Indhold

Kapitel 1

Indledning

Når der udvikles software, stilles som udgangspunkt funktionelle krav til det udviklede system. Det vil sige krav til de opgaver systemet skal være i stand til at løse for brugere eller andre systemer. Mindre fokus er der på de *ikke-funktionelle* krav. Disse krav kan være til f.eks. størrelse, sikkerhed, stabilitet osv. I dette speciale fokuseres der på to relaterede ikke-funktionelle krav: Mulighed for genbrug, og vedligeholdelsesvenligheden, altså hvor let koden i et system er at vedligeholde.

Når det er interessant at beskæftige sig med disse problemer, skyldes det ønsket om at kunne konstruere software af en høj kvalitet, som dels kan bruges i flere sammenhænge, og dels er designet så de ændringer og tilføjelser til systemet, som uundgåeligt kommer, kan implementeres så smertefrit som muligt.

Undersøgelser har vist at størstedelen, 50-75%, af den tid der bruges på softwareudvikling går til vedligeholdelsen af systemet [HK81]. Således er det ikke den initielle udvikling, men vedligeholdelsen af systemet der er mest ressourcekrævende. Det giver derfor god mening, at prøve at designe systemerne så de bliver så vedligeholdelsesvenlige som muligt. Booch [Boo91] skelner mellem *software maintenance* og *software evolution*. Vedligeholdelsen er processen at rette fejl i systemet, mens evolution er processen at tilpasse systemet til nye krav. Når vedligeholdelse nævnes i det følgende, skelnes der ikke mellem vedligeholdelse *(maintenance)* og evolution, da distinktionen ikke er relevant i specialet.

De føromtalte egenskaber: At et delsystem er designet på en måde der understøtter genbrug og vedligeholdelse, og at delsystemet er vedligeholdelsesvenligt, hænger uadskilleligt sammen. Hvor let et delsystem kan vedligeholdes, hviler direkte på de samme egenskaber som gør et delsystem genanvendeligt [Coc93]. Med andre ord er det nogen af de samme principper for udvikling, og de samme egenskaber ved systemet, der gør det muligt let at vedligeholde og genbruge dette system, eller dele deraf. Et godt design som tilgodeser genbrug vil således også være positivt for hvor let systemet er at

vedligeholde.

Der er mange elementer i softwareudvikling som bruger begrebet design. Derfor er det vigtigt at beskrive hvad begrebet betyder i forbindelse med *design af genbrugeligt og vedligeholdelsesvenligt objektorienteret software*. I sin enkelthed dækker begrebet i denne sammenhæng over:

> Dekompositionen af et system, til en mængde samarbejdende elementer.

Designet har betydning for hvilke ansvarsområder har disse komponenter har, og hvordan komponenterne kommunikerer med hinanden.

I det strukturerede paradigme, kan en funktion eller procedure betragtes som et element i designet, ligesom et helt modul med samarbejdende funktioner, kan betragtes som et sådant. I det objektorienterede paradigme, er det oplagt at betragte klasser som de samarbejdende elementer, men når programkode skal analyseres med henblik på genanvendelighed og vedligeholdelse, er det nødvendigt at betragte systemet på flere abstraktionsniveauer. Der foregår kommunikation inde i det enkelte objekt, f.eks. gennem delte variable og gennem metodekald. På et lidt højere niveu, kommunikeres mellem objekter og der foregår kommunikation mellem delsystemer. Samarbejdende elementer er derfor nødvendigvis en lidt bred definition.

Det er fordelingen af ansvarsområder, og kommunikationen mellem elementer i softwaresystemet der er de vigtigste parametre i designet af software der er muligt at genbruge og lettere at vedligeholde. De enkelte elementer i systemet kan kommunikere på forskellige måder, og det er blandt andet dette der afgør vedligeholdelsesvenligheden og genanvendeligheden.

1.1 Problemformulering

Det overordnede problem kan formuleres på følgende måde:

> Hvad karakteriserer "det gode design" i forhold til genbrug og vedligeholdelse, og kan vi automatisk evaluere software så problematiske, eller specielt gode, dele identificeres?

Dette overordnede problem rejser en mængde delspørgsmål:

- Hvilke strategier kan anvendes for at opnå genbrug?
- Hvilke mekanismer i programmeringssproget understøtter genbrug og vedligeholdelse?
- Hvilke softwaremål er beskrevet?

- Hvad betragtes som værende godt objektorienteret design?
- I hvilken udstrækning afspejles det gode design i softwaremålene?
- Er de eksisterende softwaremål tilstrækkelige?

1.2 Metode

En måde at beskrive "det gode design", som understøtter genbrug og som er vedligeholdelsesvenligt, er ved at undersøge beskrevne designprincipper. Designprincipperne antages at være udtryk for god programmeringsskik og kan derfor fortælle hvad godt design er. Derfor vil en række designprincipper blive beskrevet. Da design af software afhænger af det programmeringsparadigme der benyttes, afgrænes der til det objektorienterede paradigme.

For at kunne evaluere software, skal der findes metoder at evaluere efter. Ønsket er at finde mål der kan automatisere evalueringen. Derfor gennemgås en række softwaremål. Først gennemgås nogle traditionelle softwaremål fra det strukturerede paradigme. Dernæst beskrives en mængde mål fra det objektorienterede paradigme. Det er forventningen at disse mål kan være med til at afsløre designproblemer i objektorienterede programmer.

Designprincipperne sammenholdes med softwaremålene, for at finde ud af hvordan softwaremålene gør udslag i forhold til designprincipperne. Dette gøres for at kunne afgøre om designproblemer, i forhold til principperne, vil vise sig i softwaremålene. Der skal tages stilling til om der er egenskaber ved kildekode der ikke bliver berørt af nogen af de gennemgåede softwaremål.

Sidst vil der blive implementeret udvalgte softwaremål, som der bliver eksperimenteret med. Der bliver udviklet noget eksempelkode, som bryder med designprincipperne. Den samme kode omskrives så den overholder designprincippet, og målene bliver gennemført igen. Målene for de to versioner sammenlignes derefter for at se hvordan målene opfører sig i praksis.

Overordnet er strategien at:

- Beskrive strategier for at opnå genbrug.
- Beskrive mekanismer i det objektorienterede paradigme, som understøtter genbrug og vedligeholdelse.
- Gennemgå og beskrive traditionelle softwaremål.
- Gennemgå og beskrive objektorienterede softwaremål.
- Gennemgå et udvalg af designprincipper fra det objektorienterede paradigme.
- Sammenholde designprincipperne med nogle af de softwaremål der er blevet beskrevet.

- Gennemføre nogle mindre eksperimenter med softwaremålene.

1.3 Mål

Målet med specialet er at finde metoder til automatisk identifikation af problematisk kildekode, i det objektorienterede paradigme. Dette kræver opfyldelse af en mængde delmål. Dels er målet at præsentere eksisterende softwaremål og diskutere et udvalg af disse. Dels er et mål at præsentere et udvalg af designprincipper og finde en mulig sammenhæng mellem softwaremålene og designprincipperne.

Yderligere ønskes det at finde ud af om eksisterende softwaremål er tilstrækkelige til at afdække de egenskaber der karakteriserer et godt design.

1.4 Resultater

I dette afsnit præsenteres en kort oversigt over de resultater der blev fundet i specialet.

1.4.1 Genbrug

I forbindelse med genbrug konkluderes at:

- Der skelnes i litteraturen mellem systematisk og usystematisk genbrug. Definitionen af systematisk genbrug er for snæver i forhold til det der var ønsket at beskrive i specialet.
- Kopiering af kildekode kan betragtes som en genbrugsstrategi, men forkastes af vedligeholdelseshensyn.
- Genbrug kan opnås gennem kald til allerede implementerede dele, gennem brug af komponenter og gennem framework.
- Mekanismerne polymorfi og nedarvning understøtter genbrug.

1.4.2 Softwaremål

Gennemgangen af softwaremål gav følgende resultater:

- En mængde forskellige softwaremål blev gennemgået. En mængde mål der blev defineret i det strukturerede paradigme blev beskrevet. Elementer af disse mål kan genfindes i mål for software i det objektorienterede paradigme.
- To komplette *metric suites* blev gennemgået. Chidamber & Kemerers samling af mål og *Metrics of Object-Oriented Design - MOOD*, af e Abreu et al. Chidamber & Kemerers mål er defineret på klasseniveau, mens e Abreu et al.s mål er defineret på et samlet system.

- Gennemgangen af softwaremål afslørede to egenskaber som synes meget relevante for genbrug og vedligeholdelse: Kobling og kohæsion. Kobling er et udtryk for afhængigheder mellem dele af et system, f.eks. mellem klasser. Kohæsion er et udtryk for sammenhæng indenfor et enkelt modul eller en klasse. Det ønskes at systemer indeholde en lav grad af kobling, og en høj grad af kohæsion.

1.4.3 Designprincipper

Gennemgangen af designprincipper gav følgende resultater:

- Tretten designprincipper blev gennemgået og sammenholdt med softwaremålene.
- Nogle af designprincipperne giver udslag i målene
- Nogle principper for design af komponenter og frameworks blev beskrevet. Ingen af de gennemgåede mål kan benyttes her.
- Princippet som siger at der skal designes mod et interface, og ikke mod en klasse, bliver ikke afspejlet i nogen af målene.

1.4.4 Mangler ved eksisterende mål

Der blev identificeret følgende mangler ved de eksisterende mål:

- Der mangler et mål der forsøger at identificere klasser der burde "smeltes sammen". Konstruktionen af et sådan blev opgivet, da det ville give for mange resultater. Årsagen er at principper som Model-View-Control, anbefaler en opdeling af data og præsentation af disse. Et mål som det nævnte ville identificere kode der var designet efter Model-View-Control.
- Det blev klart under gennemgangen af designprincipperne, at princippet om at designe mod et interface ikke blev afspejlet i nogen af softwaremålene. Derfor blev to nye mål konstrueret. Begge mål udtrykker i hvor høj grad et system benytter sig af interfaces. Det ene mål benytter *kendskab* mellem klasser. Det andet abstraktionsmål er baseret på koblingen. Målene er forholdet mellem brugen/kendskabet til interfaces, i forhold til brugen/kendskabet for hele klassen.

1.4.5 Eksperimenter

Eksperimenterne gav følgende resultater:

- For at se hvordan nogle af softwaremålene opfører sig, blev der gennemført et eksperiment med målene CBO, $LCOM*$ og kendskabet.

Eksperimentet blev udført på de tre designprincipper *information expert*, *Polymorphism* og *Law of Demeter*. Resultatet var, at det var muligt at se forbedringer i koblingsmålet CBO og i kendskabet.

- Eksperimentet viste sig generelt for småt til at der kan siges noget om kohæsionsmålet $LCOM*$.
- Abstraktionsfaktorene blev afprøvet i to systemer. Eclipse, og POS. Det viste sig at Eclipse har mere kendskab til interfaces i forhold til det samlede kendskab, end POS har. Det samme gør sig gældende når kobling bruges i stedet for kendskab. Dette indikerer at Eclipse er mere fleksibelt end POS. Det indikerer også at Eclipse er mere vedligeholdelsesvenligt og strukturelt giver bedre mulighed for genbrug.

1.5 Læsevejledning

I specialet behandles forskellige områder indenfor design af genbrugeligt og vedligeholdelsesvenligt objektorienteret software. Nedenfor beskriver kort de forskellige områder i specialet, med henvisning til kapitlerne hvor området bliver behandlet.

Der er forskellige måder hvorpå man kan genbruge eksisterende kode. Forskellige strategier for genbrug, og forskellige mekanismer der kan udnyttes til at opnå genbrug. (Se kapitel 2.)

En mængde begreber er opstået under beskrivelser af egenskaber ved programmer. Kohæsion og kobling er nogle af de mest centrale. Forskellige softwaremål som beskriver kompleksiteten i programmer, eller mål der beskriver kobling eller kohæsion, er blevet konstrueret. En del af disse begreber og mål gennemgås. (Se kapitlerne 3 og 4.)

Erfaring med softwareudvikling har resulteret i forskellige regler eller principper som menes at give "god" software. Et udvalg af disse principper, både generelle og til komponent- og framworkdesign bliver beskrevet. (Se kapitlerne 5 og 6.)

Designprincipperne vurderes i forhold til nogle af softwaremålene for at finde en mulig sammenhæng. Sammenhængen er interessant for automatisk at kunne finde problematisk eller god kode. Målene bliver gennemgået sammen med de enkelte principper (Se kapitel 5.)

Egenskaber, som ikke bliver berørt af nogen af de gennemgåede mål, bliver beskrevet, og et eksperimentelt mål for abstraktioner bliver opstillet. (Se kapitel 7.)

Et lille udvalg af softwaremålene, samt det eksperimentelle abstraktionsmål, bliver implementeret og afprøvet. (Se kapitel 8.)

Kapitel 2

Genbrug

I dette kapitel gennemgås begrebet genbrug. For det første skal det afgøres hvad vi forstår ved genbrug. Dernæst skal forskellige måder hvorpå man kan opnå genbrug beskrives.

Gennemgangen af genbrug som begreb, og mekanismer der hjælper til genbrug, er baseret på litteraturgennemgang.

Konklusionen på kapitlet er, at der skelnes mellem flere former for genbrug. En form, *systematisk genbrug*, sætter snævre grænser for hvad der kan opfattes som genbrug. Gentagne kald til et modul, der allerede hører til programmet, karakteriseres ikke som systematisk genbrug. Den form for genbrug som er interessant i specialet, behøver ikke at kunne karakteriseres som systematisk genbrug.

Der er flere strategier for at genbruge kode. Genbrug gennem kopiering af kildekode forkastes som strategi, da det giver dårlige muligheder for vedligeholdelse. Genbrug kan opnås gennem kald til dele af systemet, genne brug af komponenter og frameworks.

Det objektorienterede paradigme tilføjer muligheder for genbrug. Nedarvning f.eks. giver mulighed for at genbruge implementation.

2.1 Introduktion

Begrebet genbrug i forbindelse med softwareudvikling bruges vidt. Derfor skal det først præciseres hvad der i det følgende betragtes som genbrug. Generelt er der tale om genbrug, når "noget" bliver brugt flere gange. Derfor kan man argumentere for, at man praktisk set ikke kan vide om noget er genanvendeligt *før* det faktisk har indgået i en anden kontekst end den det oprindeligt blev udviklet i. Genbrug kan således defineres som en mængde af kode, som på en eller anden måde indgår i flere kontekster. Det rejser naturligvis spørgsmål om hvornår der er tale som den samme mængde kode.

Hvis to projekter har brug for den samme, eller næsten den samme, funktionalitet, er det nærliggende blot at kopiere kode fra det ene projekt til det andet. Der er tale om genbrug, i den forstand at der er kopieret en intellektuel indsats, men det er ikke de samme linier der bliver eksekveret i det to færdige systemer. Omvendt er det klart, at et klassebibliotek som inkluderes i forskellige projekter intuitivt må karakteriseres som genbrug, da det er de *samme* klasser som benyttes i flere projekter. I litteraturen er der forskellige bud på hvad der faktisk bør karakteriseres som genbrug. *Systematisk genbrug*, har en meget snæver definition på hvad der karakteriseres som genbrug [EMT02]:

- Komponenten som genbruges må ikke modificeres. Med andre ord, tæller kun *black-box* genbrug.
- Komponenten der genbruges skal komme fra et *repository*, en "genbrugsdatabase" som indeholder de genanvendelige komponenter.
- Komponenten skal indgå direkte i det udviklede system, eller indirekte ved at indgå i værktøjer udviklet til systemet.

Ovenstående definition er konstrueret som forsøg på at opstille mål for omfanget af genbrug i et system, og er som følge deraf for snæver i forhold de problemstillinger der her skal behandles. Ezran afgrænser sig bevidst fra *ad hoc* genbrug, altså den genbrug som opstår gennem "god programmeringsskik".

2.2 Strategier for genbrug

I det følgende gennemgås en række forskellige måder at realisere genbrug på. Overordnet kan man betragte følgende strategier:

- Kopiering af kodelinier.
- Kald til inkluderede klassebiblioteker.
- Kald til egne allerede implementerede dele i samme system.
- Gennem konfigurerbare komponenter.
- Gennem frameworks.

I Ezran [EMT02] defineres yderligere granulariteten for genbrug:

- En funktion eller procedure.
- En klasse.
- En mængde klasser (samlet komponent)

- Et delsystem, et framework.
- En applikation.

Som det ses, er den mindste enhed der ifølge [EMT02] kan genbruges, en funktion eller procedure. Der er overlap mellem begreberne i strategi og granul. F.eks. er det en strategi for genbrug at implementere og genbruge et komponent, mens det i følge Ezran et. al. også er et granul. Det samme gør sig gældende for frameworks. Endnu et element i typen af genbrug, er hvor meget viden om det genbrugte elements interne struktur der er nødvendig for at kunne bruge komponenten. Som før skrevet er *black-box* genbrug brugen af et element uden viden om interne strukturer. Det eneste der kendes til et black-box element, er elementets *interface* dvs. de metoder elementet stiller til rådighed udadtil. Omvendt er det nødvendigt at kende til interne egenskaber når man benytter *white-box* genbrug. Ved denne type genbrug er det nødvendigt for udvikleren at have viden om komponentets interne struktur, da white-box genbrug f.eks. realiseres ved nedarvning og overskrivning af dele af komponentets metoder.

Det skal bemærkes at Poulin karakteriserer white-box genbrug som genbrug hvor det er tilladt at modificere kildekoden [Pou97]. Derudover definere Poulin *Gray-box* genbrug, som en form for white-box genbrug, men hvor det ikke er tilladt at modificere kildekoden.

Det er ikke alle kombinationer af ovenstående dimensioner der giver mening. F.eks. vil kopiering af kodelinier altid udgøre en form for white-box genbrug, mens klassebiblioteker ofte vil være black-box. Både komponenter og frameworks kan dog findes i begge former.

Granulariteten siger intet om *hvordan* genbruget finder sted. Således kan genbrug af de forskellige granuler foregå på forskellige måder, som angivet ovenfor. F.eks. kan en klasse (granul) genbruges ved blot at kopiere koden (strategi) til klassen, eller ved at indkludere et klassebibliotek (strategi).

2.2.1 Kopiering

En simpel form for genbrug er *kopiering af kildekode*. Et allerede løst problem, eller dele deraf identificeres. Herefter kopieres de ønskede kildekodelinier. Genbruget kan finde sted på tværs af projekter, eller inden for det samme projekt. Metoden kan være fristende da en løsning findes meget hurtigt. Der er dog flere problemer ved denne strategi. Blokke af kode er meget ofte afhængige konteksten den indgår i; af data som ikke er defineret i blokken selv. Derfor kan det være vanskeligt at isolere den mængde kode der er interessant for genbruget [Cou98]. Et andet problem er at denne form af genbrug på sigt fører til systemer som vil være meget svære at vedligeholde [HT00]. En rettelse af en fejl i systemet kan betyder at systemet skal vedligeholdes flere forskellige steder. Dette betyder flere ting: Der opstår meget let fejl i koden, da det kan være svært at have overblik over alle de steder der

skal rettes i koden. Det tager meget tid at gennemføre rettelser i systemet. I øvrigt vil det have en indflydelse på ressourceforbruget af programmet, idet dublering af kode, indenfor samme projekt, vil forøge pladsforbruget. Denne form for kodegenbrug er ikke interessant fordi den dels betyder at vedligeholdelsen bliver meget problematisk, og dels at man ikke kan drage fordel af den centrale videreudvikling og fejlretning [Cou98, p.163].

Metoden ses også brugt, blot hvor større dele kopieres. En hel applikation kopieres og der modificeres så systemet passer til nye krav. På denne måde kopieres en hel arkitektur, og man undgår de problemer som opstår når man kopierer enkelte metoder ud af en kontekst. Dog er denne form for genbrug også problematisk i forhold til vedligeholdelse, så det er ikke denne type genbrug der skal stræbes efter. Skal de dele som er genbrugt ved denne metode udvides eller fejlrettes, skal samtlige steder rettelserne skal implementeres både findes og rettes. Alene problemet at finde de relevante steder er problematisk.

2.2.2 Klassebiblioteker

Genbrug gennem brug af klassebiblioteker ligner i høj grad den type genbrug vi kender fra de ikke objektorienterede sprog. En klasse kan i denne sammenhæng betragtes som et individuelt run-time bibliotek, da det giver mulighed for at samle en mængde funktioner i en enhed [Coc93]. De fleste programmeringssprog stiller klassebiblioteker til rådighed, for udvikleren. Brugen af disse biblioteker bliver ikke af alle betragtet som genbrug. Poulin skriver at kald til metoder i standardbiblioteker som f.eks. *stdio* i C ikke kan betragtes som genbrug, da man stort set ikke kan konstruere programmer uden dette bibiotek [Pou97]. Grundlæggende matematiske operationer og lignende falder ind under samme kategori. Omvendt mener Poulin, at klassebiblioteker som ikke er en del af standard, skal betragtes som genbrug. I sammenhæng med design af genanvendelige softwarekomponenter, er det ikke nødvendigt at skelne mellem standard og ikke standard. Det interessante i denne sammenhæng er designet af komponenterne, og brugen af disse.

Vedligeholdelsen af klasser fra klassebiblioteker foregår kun ét sted, hvilket fra et vedligeholdelsessynspunkt er godt. Det betyder at en fejl som rettes i et klassebibliotek, retter fejl i alle de systemer den aktuelle klasse indgår i. Det medfører naturligvis, at fejl der evt. bliver introduceret ved rettelsen, også introduceres i alle de systemer som bruger biblioteket.

Det er ikke nødvendigvis uproblematisk at genbruge klasser fra klassebiblioteker. Hvis klassen er dårligt designet så den f.eks. har sideeffekter som det oprindelige system er afhængigt af, er klassen svær at genbruge.

2.2.3 Kald til dele af systemet selv

Fornuftigt designet software implementerer sine funktioner så generelt at de kan kaldes og bruges flere steder fra. Genbruget minder meget om genbrug gennem klassebiblioteker, men adskiller sig herfra ved kun at være del af det aktuelle system under udvikling. På sigt bør de generelle klasser flyttes til et klassebibliotek, så andre projekter også kan udnytte implementationerne.

2.2.4 Komponenter

En komponent kan indeholde en mængde gensidigt afhængige klasser, som arbejder sammen for at løse et givent problem. Afhængigheden kan bestå i at forskellige dele af komponenten har brug for at kalde andre dele af komponenten, eller gennem delte data. Denne definition minder om Coulanges definition på en *package*. En package (skal ikke forveksles med en Java-pakke), er i følge Coulange en mængde delprogrammer som deler data [Cou98]. Udadtil skal en package opfylde principperne om information hiding, men en package, ifølge Coulange, er ikke speciel for det objektorienterede paradigme. Coulange definerer et genbrugeligt komponent som

> ...an entity which can be used, possibly with modifications, in a new software development. [Cou98, p.51]

Denne definition er meget bred, og afgrænses lidt at Poulin [Pou97] i følge følgende definition:

> ...we define a *reusable component* as a group of functionally related software modules and their associated documentation. [Pou97, p.2]

Som tidligere skrevet, kræver Poulin, at der ikke foretages ændringer i et genbrugt element, for at kunne karakterisere det som genbrug. Samtidig skal et komponent være ledsaget af dokumentation, som i øvrigt kan være begrænset til kommentarer i kildekoden. I følge Jacobson et al. er et komponent blot en type, klasse *eller andet produkt* som er designet til at være genanvendeligt [JGJ97a]:

> A **component** is a type, class or any other workproduct that has been specifically engineered to be reusable. [JGJ97a, p.85]

Jacobson et al.s definition er dog ikke eksklusiv for kildekodekomponenter, men kan lige så godt være use cases og analyse. Baseret på ovenstående definitioner, vil et komponent i det følgende være betegnelsen for en samling samarbejdende klasser som er funktionelt relaterede, og som samlet løser et afgrænset problem. For ikke at få en for flydende grænse mellem de forskellige strategier for genbrugeligt design, kræves det, at komponenten ikke modificeres. Altså adopteres dele af Poulins definition, hvor black-box genbrug er et krav. Yderligere diskussion af komponenter og design af disse findes i kapitel 6.

2.2.5 Frameworks

Et *framework* er et genbrugeligt design. I et framework genbruges der kode, men ideen med et framework er i høj grad at strukturen i designet genanvendes. Der er lidt forvirring om begrebet, da nogle betragter frameworks som en slags komponenter, og andre betragter dem som store designmønstre [Joh97]. Frameworks indeholder også elementer fra begge dele. De er mindre køreklare end komponenter, men indeholder kode i modsætning til designmønstre. Der er givet flere definitioner på hvad et framework er og hvad det indeholder. Fælles for disse definitioner er, at et framework består af en mængde klasser og interfaces, mellem hvilke der er defineret relationer. Således giver frameworks mulighed for at genbruge hele arkitekturer, hvilket i følge [WBJ90], formentligt vil være vigtigere for softwareudviklingen end blot genbrug af kode. Grunden til dette er, at interfacedesignet og den funktionelle dekomposition udgør det største intelektuelle arbejde, som er langt sværere at konstruere eller genkonstruere, end kode er [WBJ90, p.116].

I Viljamaa gives definitionen [Vil97, p. 1]

> ...a set of objects - an application core - that captures the special expertice in some application domain to a reusable form.

Denne definition, hvis den tages helt bogstaveligt, er relativt snæver, da den beskriver frameworks som en *"application core"*. Dette begrænser os i hvordan vi kan tænke på frameworks. Hvis udvikling af frameworks kun overvejes under udviklingen af *kerneområdet* i applikationen, er der en risiko for, at designet af perifære delsystemer, som ellers kunne være oplagte at implementere som frameworks, bliver implementeret uden de muligheder som frameworks giver. Denne traditionelle opfattelse af frameworks: At en applikation baseres på ét framework som udvides med applikationsspecifik kode, er problematisk fordi det ofte viser sig at være ønskværdigt a integrere flere frameworks med hinanden [MBF99]. Når et framework implementeres i den tro at de kun skal udvides, og ikke integreres, gøres antagelser som ikke er hensigtsmæsseige når frameworket evt. alligevel skal integreres med andre delsystemer.

Frameworks kan opdeles i de domænespecifikke [PSAD97] og de generelle. Et domænespecifikt framework er et som kun kan anvendes indenfor et bestemt domæne, som f.eks. et framework som implementerer behandlingen af ordrer i et finanssystem. Et generelt framework kan bruges på tværs af domæner. Tænkes f.eks. på *applets* i Java, implementeres disse i et framework som sørger for at metoder som initialiserer, starter og stopper appletten bliver kaldt på de rigtige tidspunkter, når appletten eksekveres i en webbrowser. I dette applet-frameworket kan der implementeres systemer til mange forskellige domæner.

Generelt beskrives frameworks som en mængde af klasser som danner et skelet af et system eller delsystem. Samlet indeholder frameworket en

mængde klasser, samt en ansvarsfordeling, eller *infrastruktur* mellem disse klasser [Coc93, GHJV95, Jia00]. Frameworks, og designet af disse, bliver gennemgået tilbunds i kapitel 6.

2.3 Genbrugsmekanismer i det objektorienterede paradigme

Det objektorienterede paradigme er ikke nødvendigt for at konstruere genanvendeligt software. Samtidig er det ikke garanteret, at delsystemer implementeret i det objektorienterede paradigme, er genanvendelige. Der er dog egenskaber ved det objektorienterede paradigme, som kan udnyttes til at opnå en højere grad af genanvendelighed og vedligeholdelsesvenlighed. Muligheder som nedarvning, polymorfisme og indkapsling, som traditionelt nævnes som specielle for objektorienterede paradigme, giver en fleksibilitet som gavner muligheden for genbrug [EMT02, p.45] [Coc93]. I det følgende beskrives disse mekanismer.

2.3.1 Indkapsling

Indkapsling (eng.: *encapsulation*) er det forhold, at det er muligt at repræsentere data, og de operationer der bruges sammen med disse data, i en samlet enhed [vdL99, Rog01]. Eksempelvis er javaklassen `java.lang.String` en klasse som kan repræsentere en tekststreng. Data, som er selve tekststrengen, og nogle af de operationer der kan give mening sammen med en tekst, f.eks. `substring()`, er samlet i samme enhed. På denne måde er der adgang til operationer på data, når man fat i selve data. Det er positivt for genbrugsmulighederne at disse er sammenhængende.

En vigtig følge af indkapsling, er muligheden for at skjule information, om interne data og strukturer for brugere af objektet: *Information hiding*. I litteraturen bruges begreberne *encapsulation* og *information hiding* ofte som synonymer [GHJV95, Boo91], men ikke alle deler denne opfattelse [Rog01]. I følge Rogers er indkapsling en facilitet som programmeringssproget tilbyder, mens information hiding er et designprincip [Rog01]. Stroustrup støtter dette synspunkt, da han skriver at det er muligt at bryde principperne om information hiding i sprog som tilbyder indkapsling, og det er muligt at praktisere information hiding i sprog uden indkapsling [Str91]. På grund af denne generelle forvirring, vil der blive skelnet mellem de to begreber.

I nogen sammenhænge nævnes *implementation hiding* specifikt, men da både information hiding og implementation hiding handler om at skjule detaljer om et objekts interne dele, vil der i det følgende ikke blive skelnet mellem disse to begreber.

Tanken med information hiding er som sagt, at skjule detaljer om implementationen af klassen fra brugeren af klassen. Alle operationer på klassen,

og tilgange til data i klassen, skal i følge princippet, tilgås gennem dertil implementerede metoder i klassen, og det bliver derved unødvendigt at kende til andre forhold vedrørende klassen, end det *interface* klassen stiller til rådighed.

Skjules metoder som ikke er relevante for komponentens virkemåde, bliver interfacet reduceret, hvormed kompleksiteten af interfacet ligeledes reduceres. Dermed bliver en bruger af komponenten bedre i stand til at overskue komponenternes interface, og dermed brugen af disse.

Princippet bliver brugt sammen med designprincippet at implementere mod et interface i stedet for at bruge viden om interne strukturer i klassen der benyttes. En gennemgang af dette princip sammen med en beskrivelse af de problemer princippet løser, findes i afsnit 5.4 i kapitel 5.

Indkapsling og information hiding støtter udviklingen af black-box komponenter, og gør det lettere at genbruge klasser, da data og operationer inkluderes som en enhed. Bruges information hiding sikrer man sig at klienter til komponenten ikke *kan* være afhængig af detaljer i implementationen [Lis88].

2.3.2 Nedarvning

Nedarvningsmekanismen gør det muligt at genbruge implementationen fra en forældreklasse, og implementere nye klasser ved specialisering af eksisterende. Når en klasse `C` nedarver fra en klasse `P`, bliver mange af egenskaberne[1] i `P` tilgængelige i `C`. Genbruget består i at nedarvede klasser bruger den samme implementation som findes i forældreklassen.

Som det kommer til at fremgå af designprincipperne senere, er det ikke uvæsentligt hvordan nedarvning i praksis bruges. I forbindelse med ændringer i klasser som har subklasser, er der en risiko for at ændringerne kommer til at påvirke subklasserne. Det er ikke realistisk at rettelser i superklasser altid propagerer gennem klassehierarkiet til de nedarvede klasser, således at subklasserne bliver ved med at virke. [Lis88]. Der er afhængigheder mellem klasser og eventuelt nedarvede klasser, så ændringer i superklasser er risikable. Ændringer i en superklasse som ikke medfører ændringer i subklasser, kræver at de forventninger subklasserne har til superklassen stadig holder. Præcis som i tilfælde af afhængigheder mellem andre klasser.

Chidamber & Kemerer målte objekthierarkiers dybde og antal umiddelbare børn til klasser. Fra deres data sås det, at dybden af de målte objekthierarkier ikke var særlig stor, og at en klasse typisk havde meget få børn. Dette antyder at genbrug gennem nedarvning ikke bliver brugt i særlig stor udstrækning i de målte systemer [CK94, p.486]. En forklaring i følge [CK94] kunne være at virksomhedernes designmetoder dikterede et lavt hierarki, eller at der mangler kommunikation mellem de forskellige objektdesignere.

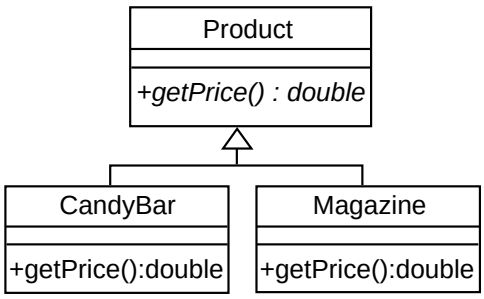
[1] Synligheden af metoder og attributter i nedarvede klasser er, i Java, afhængig af deklarationen af disse, i forældreklassen.

Hvis designmetoden dikterer et lavt objekthierarki kan det være af forskellige årsager. Et dybt hierarki er ofte svært at forstå. Et dybt hierarki giver mere genbrug af kode, men på bekostning af fleksibilititen i muligheden for at modificerer en klasse højt oppe i hierarkiet. Jo dybere et objekthierarki, jo større er risikoen for uønskede sideeffekter i forbindelse med modifikationer. I [CSAD02] er graden af genanvendelighed af en komponent sammenhængende med generaliteten af komponentet. Et generelt komponent ligger højere i objekthierarkiet end et specifikt. Derfor kan dybe objekthierarkier have negativ indflydelse på komponentgenbrug, selvom det har en positiv indflydelse på genbrug af antal linier af kode.

Et alternativ til genbrug af eksisterende klasser gennem nedarvning, er genbrug ved hjælp af komposition. Ved komposition sammensættes nye klasser af eksisterende. Fordele og ulemper ved nedarvning og komposition, bliver gennemgået i et senere kapitel.

2.3.3 Polymorfisme

Polymorfisme referer til muligheden for at en reference i koden kan pege på forskellige typer. Gennem *late binding* afgøres, på afviklingstidspunktet, hvilke typer der refereres til, og dermed hvilke metoder der faktisk kaldes, når metoder på et objekt kaldes.



Figur 2.1: Eksempel på polymorfi

I eksemplet 2.1 er to varer `Magazine` og `Candybar` begge nedarvet fra `Product`. `Product` definerer en metode `getPrice` som returnerer prisen på en vare. Denne metode overskrives i både `Candybar` og `Magazine`, således at de forskellige produkter har mulighed for at returnere forskellige priser. Uanset om den aktuelle reference er til en `Candybar` eller et `Magazine`, eksiterer en metode som returnerer prisen på varen. Udvikleren behøver således ikke at skelne mellem de to typer af produkter. Det skal bemærkes at det netop beskrevne problem, at kunne returnere forskellige priser afhængigt af produktet i et virkelighedstro system formentligt ville være implementeret anerledes. Eksemplet viser dog alligevel grundideen i polymorfisme.

2.4 Sammenfatning

Kapitlet beskrev begrebet genbrug, og hvordan genbrug kan opnås når man udvikler software. Yderligere blev der gennemgået en række mekanismer fra det objektorienterede paradigme der bidrager til genbrug og vedligeholdelse.

- Der skelnes mellem *systematisk* genbrug og *ad hoc* genbrug. Denne skelnen er ikke interessant for specialet, da genbrug gennem "god programmeringsskik" er ligeså væsentlig når det "gode design" skal identificeres.
- Der findes flere strategier for genbrug. Fra ren kopiering af kodelinier til udarbejdelse af komponenter og frameworks. Kopiering forkastes som genbrugsstrategi da det giver dårlige muligheder for vedligeholdelse.
- Genbrug kan opnås gennem kald til allerede implementerede dele i systemet, gennem brug af komponenter og gennem framework.
- Der er egenskaber som er specielle ved det objektorienterede paradigme, der øger muligheden for vedligeholdelse og genbrug.
- Indkapsling binder data og operationer på disse, sammen i en enhed. Begrebet forveksles ofte med *information hiding* som er et princip der siger at en klasses (eller en komponents) interne dele skal skjules for omverdenen.
- Nedarvning giver mulighed for genbrug af implementation i en superklasse.
- Polymorfi giver mulighed for at arbejde med objekter af forskellig type uden at kende deres nøjagtige type.

Kapitel 3

Traditionelle softwaremål

I dette kapitel gennemgås forskellige softwaremål som er blevet defineret i det strukturerede paradigme. Disse mål gennemgås af to årsager: I blandt de gennemgåede er nogle af de første softwaremål der blev defineret. For det andet er der stadig elementer fra disse mål, der indgår i softwaremål der senere er blevet defineret. Kapitlet tjener som oversigt over de mest omtalte mål i det strukturerede paradigme.

Konklusionen på kapitlet er, at der er gjort flere forsøg på at opstille softwaremål.

Kobling og kohæsion er beskrevet, og forskellige grader af kohæsion er blevet beskrevet. Koblingen udtrykker afhængigheder mellem moduler, og kohæsionen er et begreb der udtrykker sammenhængen inde i et modul. Der er ikke fundet nogen softwaremål der skelner mellem de forskellige former for kohæsion.

3.1 Introduktion

Der er, af forskellige årsager, gjort en del forsøg på at karakterisere forskellige egenskaber i kildekode. Mængden af kildekode i forhold til en programmørmåned har været brugt som mål for produktivitet [Som92], og andre mål har været opstillet for at prøve at udlede kvaliteten og kompleksiteten af et stykke kildekode.

I forbindelse med genbrug og vedligeholdelse, resulterer høj kompleksitet ofte i, at udvikleren begrænses i sine muligheder for at genbruge. Det skal bemærkes at der her tales om kompleksitet i strukturer i koden og lignende, og ikke om kompleksitet i problemdomænet. Et delsystem kan implementere et meget komplekst domæne uden at være komplekst ud fra de mål vi her taler om. Dog kan der være en tendens til at komplekse domæner er svære at forstå, og på denne måde skaber problemer i nogen typer genbrug.

Et kompleksitetsmål er et objektivt mål for, hvor svært det vil være for en programmør, at udføre almindelige programmeringsopgaver, som f.eks. vedligeholdelse af kildekode [LC]. Et kompleksitetsmål forsøger altså at måle relationer mellem egenskaber ved programkode, og problemer der opstår ved arbejdet med denne programkode. Graden af hvor let det er at modificere og genbruge programkode, afspejles i nogen grad af hvor kompleks koden er. Derfor er det interessant at betragte kompleksitetsmål i forbindelse med genbrug og vedligeholdelse.

Det skal derfor undersøges hvilke egenskaber der har indflydelse på graden af hvilken, det er muligt at genanvende og vedligeholde systemer. Dette problem kan betragtes fra to sider: Hvilke attributter skal et system have for at give mulighed for genbrug, og hvilke attributter begrænser os fra at kunne genbruge et delsystem. Typisk opnås førstnævnte ved at følge nogle principper for design af systemer, mens begrænsningerne for genbruget af et system kan vise sig gennem evaluering af systemet. Denne evaluering kan foregå gennem mål af de egenskaber ved et system, som antages at afgrænse udvikleren fra at genbruge.

Der skal skelnes mellem attributter der *hjælper til* en øget genanvendelighed, og de attributter der *understøtter* eller er direkte nødvendige for genbrug. Egenskaber som fornuftig variabelnavngivning, f.eks. Ungarsk notation [SH91] og kildekodeformatering hjælper til øget genbrug, da kildekoden bliver lettere at læse og lettere forståelig. Disse attributter betyder dog ikke, at det ikke er teknisk muligt at genanvende systemet uden store tilpasninger, men genbrug som kræver at udvikleren sætter sig ind i den eksisterende kode, kan blive udelukket hvis koden er svært at forstå og dermed svær at inkorporere

Andre attributter kan derimod have indflydelse på om et system overhovedet kan genbruges i en anden kontekst uden meget store tilpasninger af systemet. Hvis der er meget store afhængigheder mellem dele af et system, kan komponenter dårligt isoleres og dermed dårligt genbruges.

Begreberne kohæsion *(eng: cohesion* eller *strength*) og kobling *(eng: coupling)*, er accepteret som nøgleegenskaber i forbindelse modulariteten i et system. Kohæsion er et begreb som dækker over hvor sammenhængende de enkelte dele i af softwaremodul er i forhold til hinanden. Der stræbes efter høj kohæsion, da dette indikerer en stor sammenhæng mellem elementerne. Ideen er, at de elementer som optræder i samme modul, skal være relaterede til hinanden. Koblingsbegrebet udtrykker afhængigheder mellem moduler. Der stræbes efter lav kobling, da afhængigheder mellem moduler ofte betyder, at modifikationer i et modul medfører rettelser i alle afhængige moduler. Samtidig er kobling i perspektivet genanvendelighed problematisk, da det kan være svært at isolere de moduler der er interessante at genbrug.

Selvom de klassiske kompleksitetsmål har rødder i procedurale programmeringssprog, er der elementer i målene som er relevante i objektorienterede sprog. Der kan argumenteres for, at udviklingen af en enkelt metode i en

klasse ligger så tæt op af traditionelle måder at udvikle på, at et klassisk mål kan være anvendeligt som del af et mål for kompleksiteten af en klasse [CK94]. Af denne årsag og fordi nogle objektorienterede softwaremål benytter traditionelle kompleksitetsmål, bliver et antal traditionelle kompleksitetsmål gennemgået i dette kapitel.

3.2 Størrelsesmål

Størrelsesmål har været anvendt til forskellige formål, og typisk har størrelsen af et system været opgivet i antal kodelinier *(LOC - eng: Lines of Code)*. Dette mål har blandt andet været anvendt til f.eks.:

- Kvalitet, målt i antal fejl pr. tusinde linier kode.
- Produktivitet, målt i antal kodelinier pr. måned.
- Dokumentation, målt i antal kommentarer pr. tusinde kodelinier.
- Omkostninger, målt i pris pr. tusinde kodelinier.

Som det ses, er LOC blevet brugt til normalisering af andre egenskaber ved systemer, således at det bliver muligt, at sammenligne data på tværs af forskellige projekter. LOC er dog ikke den eneste måde størrelsen af et system er blevet opgivet på. Antal tokens og antal funktioner er brugt som størrelsesangivelse, og antal variabelerklæringer er blevet brugt som kompleksitetsmål i sig selv.

I genbrugs- og vedligeholdelsessammenhænge, er det en tanke, at store moduler kan være svære at genbruge, fordi størrelsen gør dem uoverskuelige. Samtidig er der en tendens til at store moduler er vokset fordi de har fået tildelt for mange ansvarsområder. Dette gør dem svære at genbruge. Derfor kan tilstedeværelsen af store moduler eller funktioner anvendes som advarselsindikator. Bliver moduler eller funktioner store, kan det være et symptom på uhensigtsmæssigt design.

Størrelsesmål er nødvendigt for at kunne normalisere data, og målene kan indikere at funktioner bør splittes op, men størrelsen alene er ikke velegnet som mål for genanvendelighed og vedligeholdelsesvenlighed.

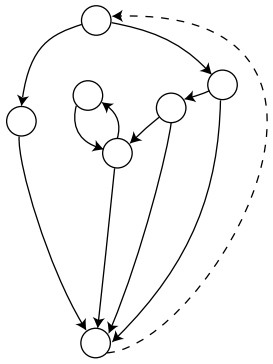
3.3 McCabes cyklomatiske kompleksitet

McCabe opstiller et kompleksitetsmål, cyklomatisk kompleksitet, som bygger på et programs kontrolgraf [McC76]. Tanken er, at jo mere kompleks en kontrolstruktur, jo mere komplekst er programmet. Målet er et udtryk for hvor meget kontrollogik der er i programmet. Målet kan sige noget om hvormange afprøvninger der mindst skal til, for at få dækket hele kontrolgrafen i et modul, men i denne sammenhæng er kompleksiteten interessant fordi et

komplekst modul er mere besværligt at forstå og vedligeholde. Cyklomatisk kompleksitet blev udviklet til strukturerede sprog, men er stadig interessant i objektorienterede sprog, når en enkelt metode betragtes.

```
   .
   .
   .

13:public int someCalculation(int option, int value) {
14:   int retValue = 1;
15:
16:   if (option == McCabe.FAHRENHEIT2CELSIUS)
17:      retValue = (int) ((value-32)/1.8);
18:   else if ((option == McCabe.FACULTY) && (value > 1))
19:      while(value > 1) {
20:         retValue *= value;
21:         value--;
22:      }
23:   return retValue;
24:}


   .
   .
   .
```

Figur 3.1: Eksempel på cyklomatisk kompleksitet.



Figur 3.2: Kontrolgraf for kildekoden fra figur 3.1.

McCabes mål stammer fra grafteorien, hvofra har vi det cyklomatiske tal γ, for en orienteret graf, defineret på følgende måde [Mat]:

$$\gamma = e - n + 1 \tag{3.1}$$

I 3.1 er e antal kanter, og n antal knuder i grafen. Grafen skal være sammenhængende. Dette cyklomatiske tal, er udtryk for hvor *få* kanter der skal fjernes for at der ikke længere findes cykler i grafen.

I følge et teorem givet i [McC76] gælder det, at det cyklomatiske tal er lig med det største antal mulige lineært uafhængige cykler i grafen, i en stærkt sammenhængende graf. Disse uafhængige cykler danner en basismængde, fra hvilken lineære kombinationer kan danne hvilket som helst gennemløb af programmet.

Hvis det cyklomatiske tal skal kunne beregnes, kræves det at flowgrafen er stærk forbunden, hvilket vil sige, at der findes en sti fra enhver knude i grafen til enhver anden knude i grafen, ved at følge kanterne i grafen. For at opnå at flowgrafen opfylder dette, medregnes en imaginær kant fra den sidste knude i grafen, tilbage til den første knude. McCabes kompleksitetsmål bliver således:

$$v(G) = e - n + 2 \tag{3.2}$$

hvor e igen er antal kanter i grafen og n er antallet af knuder. Beregningen af kompleksiteten kan dog simplificeres, så det ikke er nødvendigt at konstruere en kontrolflowgraf for programmet. Ifølge McCabe, kan man i praksis blot tælle prædikater i koden. Sammensatte prædikater, som

```
if ((option == McCabe.FACULTY) && (value > 1))
```

bidrager med to til kompleksiteten[1], da ovenstående er ækvivalent med

```
if(option == McCabe.FACULTY)
   if(value > 1)
```

Det er således muligt at finde den cyklomatiske kompleksitet, udelukkende ved at betragte syntaktiske konstruktioner i koden, ved at tælle prædikater.

$$v(G) = \pi + 1 \tag{3.3}$$

π er i denne sammenhæng antal prædikater i koden.

I figur 3.1 vises et, upraktisk, modul der kan foretage to forskellige beregninger. Kontrolgrafen for denne ses i figur 3.2. Den cyklomatiske kompleksitet $v(G)$ for grafen i figur 3.2, er $v(G) = 10 - 7 + 2 = 5$. Det samme resultat kan fås ved at tælle prædikaterne i kildekoden i figur 3.1. Optællingen af prædikater kan ses i tabel 3.1.

Linie		Bidrag
16	`(option == McCabe.FAHRENHEIT2CELSIUS)`	1
18	`((option == McCabe.FACULTY) && (value > 1))`	2
19	`(value > 1)`	1
		$\pi = 4$

Tabel 3.1: Sammentælling af prædikater fra figur 3.1.

Resultatet fra tabel 3.1 er π, som indsættes i formel 3.3, hvilket giver fem, ligesom tidligere.

McCabe fastsætter en vejledende øvre grænse for kompleksiteten på 10. Det fremhæves dog, at dette ikke er et "magisk" tal, men at det virker rimeligt.

3.4 Halstead's mål

I Halstead's *Software Science* [Hal77] er tanken at måle nogle grundelementer i programmer, og udfra disse konstruere mål for en mængde attributter i kildekode. Grundelementerne er *operatorer* og *operander*, og Halsted argumenterer for at algoritmer kun består af disse to elementer, ved at betragte computere, hvis instruktionsformat netop kun består af operatorer og data disse opererer på; operander [Hal77, p.8]. Operander er alle variable og konstanter, og operatorer er de elementer som påvirker værdien i operanderne. Halstead identificerer følgende tælbare attributter fra et program:

[1]Det skal bemærkes at flere kommercielle værktøjer, bla.a *JStyle* fra "Man Machine Systems" [JSt] og *Understand for Java* fra "Scientific Toolworks, Inc." [Und] beregner McCabe uden dette hensyn. Eksemplet i figur 3.1 vil i disse systemer have v(G)=4

η_1 = Antal unikke operatorer i programmet.

η_2 = Antal unikke operander i programmet.

N_1 = Total antal operatorer i programmet.

N_2 = Total antal operander i programmet.

Fra dette defineres forskellige mål som *vokabulariet* $\eta = \eta_1 + \eta_2$, og *programlængden* $N = N_1 + N_2$. Da et program ikke består af andet end operatorer og operander, kan længden af programmet opgøres på denne måde.

Halstead definerer et andet størrelsesmål, *program volume*, som er størrelsen af programmet opgjort i antallet af bits der skal bruges for at repræsentere programmet. Da det samlede vokabolarium er η, skal der $log_2\eta$ bits til at repræsentere samtlige elementer i vokabolariet. Længden af programmet er N, så hele programmet, *volumenet* defineres som:

$$V = N\ log_2\ \eta \tag{3.4}$$

Yderligere defineres det *potentielle volumen* som er den mest kortfattede måde at beskrive en operation på. Den potentielle volumen findes ved følgende formel:

$$V^* = (N_1^* + N_2^*)\ log_2\ (\eta_1^* + \eta_2^*) \tag{3.5}$$

Denne mest kortfattede beskrivelse, vil være et kald til en implementation af operationen, og denne vil ikke kræve gentagelse af hverken operatorer eller operander. Derfor gælder det at $N_1^* = \eta_1^*$ og $N_2^* = \eta_2^*$. I følge Halstead, skal der bruges præcis to operatorer til kaldet, netop en til navngivningen af operationen, og netop en til at repræsentere tildelingen. Derfor vil antallet af unikke operatorer, η_1^* altid være 2. Tilbage er blot at finde antallet af input *og* output parametre, som er antallet af unikke operander η_2^* i kaldet. Samlet bliver det potentielle volumen:

$$V^* = (2 + \eta_2^*)\ log_2\ (2 + \eta_2^*) \tag{3.6}$$

Fra ovenstående kan *program level* L defineres, som forholdet mellem det potentielle volumen V^* og volumen V af implementationen:

$$L = \frac{V^*}{V} \tag{3.7}$$

Program level udtrykker hvor tæt et program er på sit potentielle volumen, og kun den mest kortfattede implementation kan have program level på én. Jo mere volumniøse implementationer bliver, jo lavere program level værdi vil de opnå.

I tabel 3.2 ses optællingen af η_1, η_2, N_1 og N_2 for modulet i figur 3.3. Optællingen foregår på Javakildekode, og Halstead definerede oprindeligt sit

```
  ⋮

public int someCalculation(int option, int value)
{
   int retValue = 1;

   if (option == Halstead.FAHRENHEIT2CELSIUS)
      retValue = (int) ((value-32)/1.8);
   else if ((option == Halstead.FACULTY) && (value > 1))
      while(value > 1) {
         retValue *= value;
         value--;
      }
   return retValue;
}

  ⋮
```

Figur 3.3: Eksempelkildekode til Halstead's mål

Operator	Antal	Operand	Antal
`public`	1	`someCalculation`	1
`int`	5	`option`	3
`(...)`	9	`value`	6
`,`	1	`retValue`	4
`{...}`	2	`1`	3
`=`	2	`Halstead.FAHRENHEIT2CELSIUS`	1
`;`	5	32	1
`if...else`	1	1.8	1
`==`	2	`Halstead.FACULTY`	1
`-`	1		
`/`	1		
`if`	1		
`&&`	1		
`>`	2		
`while`	1		
`*=`	1		
`--`	1		
`return`	1		
$\eta_1 = 18$	$N_1 = 38$	$\eta_2 = 9$	$N_2 = 21$

Tabel 3.2: Optælling til Halstead's mål for modulet i figur 3.3.

mål på Fortrankildekode. Da det kan være problematisk at afgøre *præcis* hvad der skal tælles med som operander og hvad der skal medtælles som operatorer, kan der være forskelle i de resultater forskellige værktøjer rapporterer, fordi disse tæller forskelligt. Optalte værdier for to forskellige værktøjer fremgår af tabel 3.3. De forskellige resultater afgøres bla. af hvad der betragtes som operatorer. En operator som `*=` kan tælles som en unik operator, eller den kan tælles som `*` og `=`. Dette har indflydelse både på antallet af unikke operatorer η og på det samlede antal operander N. De forskellige resultater betyder blot, at kun værdier beregnet på samme måde kan sammenlignes. Desværre oplyser de afprøvede produkter ikke hvordan deres optælling fungerer. Det skal bemærkes at *program level L* er meget lav i eksempelkoden, hvilket betyder at koden er meget stor i forhold til den korteste måde at udtrykke problemet. En del af problemet kan være, at mange kompleksitetsmål fra det strukturerede paradigme, generelt rapporterer højere kompleksiteter for de objektorienterede sprog, end for de strukturerede [JFF+02][2], men da målet kun beregnes på en enkelt metode, som jo ligger tæt på strukturen for et enkelt modul, er dette næppe hele forklaringen. Resultater fra [SXC99] viser, at der var en sammenhæng mellem programmøres opfattelser af hvor vedligeholdelsesvenlige moduler var, og værdien af modulernes mål. I forsøget i [SXC99], blev et antal programmører spurgt om deres mening med hensyn til vedligeholdelsesvenligheden af en mængde moduler. Forskellige mål blev beregnet for modulerne, bla. LOC, McCabe og Halstead, og det viste sig at der var høj korrelation mellem kompleksiteten og programmørenes opfattelse af modulerne. I følge Morasca [Mor] er Halstead's mål ikke brugt i stor udstrækning, selvom der findes en mængde værktøjer der er i stand til at beregne målet. Som eksempel, sætter Morasca en grænse for volumenet: $V \leq 4000$. En grænse der virker vilkårlig, og som der ikke argumenteres for.

Attribut	JStyle [JSt]	Understand for Java [Und]	Egen optælling
η_1	-	19	18
η_2	-	10	9
η	25	29	27
N_1	-	41	38
N_2	-	23	21
N	64	53	59
Vol	215	246.12	280.54
L	0.01	0.09	0.01

Tabel 3.3: Forskellige mål af Halstead.

[2]Jorgensen et al. betragtede pseudokode for hhv. strukturerede og objektorienterede sprog, og konkluderede at de objektorienterede sprog var mere komplicerede målt med LOC, McCabe og Halstead. Målet var at vise at indlæringen for de objektorienterede sprog er sværere end for de strukturerede [JFF+02].

3.5 Fan-in/fan-out

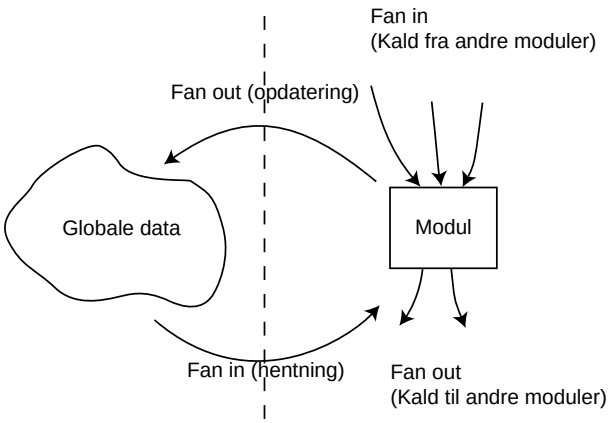
Et program kan betragtes som en mængde moduler som kalder hinanden. Et element i kodegenbrug er genbrug af moduler som udfører en begrænset veldefineret opgave. Jo mere afgrænset og veldefineret opgaven er, jo nemmere er modulet at genbruge. Identificeres en fejl i modulet, skal denne rettes ét og kun ét sted, hvilket tydeligt er en fordel. Til gengæld koster denne genbrug noget. Ændringer som finder sted i det kaldte modul, kan have effekt på samtlige kaldende moduler, hvilket betyder at kalderen er sårbar med hensyn til ændringer i det kaldte modul. Der er altså en afhængighed fra kaldende til kaldte modul. Denne attribut afspejles i et moduls *fan-in*. Fan-in defineres af Yourdon & Constantine som antallet af moduler som refererer til det specificerede modul [YC78]. Betragtes et strukturdiagram, som afspejler et program, svarer fan-in til et modul, til antallet af linier der går ind i modulet. Dette svarer til antallet af andre moduler som kalder det aktuelle modul [Som92]. En høj fan-in antyder at der er høj kobling, da fan-in er et mål for afhængigheder mellem moduler [Som92].

Fan out defineres tilsvarende som antallet af moduler som det aktuelle modul kalder. Igen svarer det til at se på antallet af andre moduler som bliver kaldt fra det modul fan-out skal findes for. Høje fan-out værdier kan være udtryk for høj kompleksitet i modulet, på grund af den kontrollogik der skal koordinere kaldene til de underliggende moduler [YC78].

Betragtes ovenstående fan-in/fan-out definition som et udtryk for kobling mellem moduler, har den en brist idet den ikke tager hensyn til ændringer i globale data, der deles mellem moduler. En anden definition af fan-in/fan-out tager højde for denne form for kobling [HK81]. I denne definition (informational fan-in/fan-out [Som92]) tælles antallet af lokale dataflows sammen med antallet af tilgange til globale data. Et lokalt dataflow findes fra modul A til modul B, hvis en af følgende betingelser er opfyldt [HK81]:

1. A kalder B direkte.
2. B kalder A som returnerer en værdi v til B, hvor v efterfølgende bliver brugt i B
3. C kalder både A og B, og sender en returværdi fra A videre til B

Henry og Kafura definerer *fan-in* af en procedure A, som antallet af lokale dataflows der går ind i A plus antallet af datastrukturer som A henter information fra. *Fan-out* af en procedure A, er antallet af lokale flows som går ud af A plus antallet af datastrukturer som A opdaterer [HK81]. Konsekvensen af denne definition er, at koblinger mellem moduler som skabes gennem delte data, også får en effekt i fan-in/fan-out. Figur 3.4 viser fan-in og fan-out for et modul. Hele figuren viser Henry og Kafuras definition, mens figuren til højre for den stiplede linie viser Yourdon og Constantines definition.



Figur 3.4: Fan-in/fan-out

Hvor Yourdon og Constantine antyder at der er en sammenhæng mellem et moduls interne kompleksitet og dets fan-out, går Henry og Kafura et skridt videre og definerer et kompleksitetsmål baseret på fan-in og fan-out. Kompleksiteten defineres som: $complexity = length \times (fan\text{-}in \times fan\text{-}out)^2$ I Henry og Kafura benyttes et simpelt mål for *length*, idet der blot tælles linier i kildekoden. Dette inkluderer eventuelle kommentarer. Argumentet for dette længdemål er, at man ved at der er en korrelation mellem antal kodelinier og hvor mange fejl koden indeholder, selvom korrelationen ikke er den stærkeste og at det simple længdemål er benyttet fordi det var nemt af finde. En mulighed er også at benytte McCabes cyklomatiske kompleksitet, eller Halsteads length som *length* i ovenstående [HK81].

Når et modul har høje fan-in/fan-out værdier, viser det at modulet har mange forbindelser til andre dele af systemet. Dette indikerer blandt andet, at modulet har for mange ansvarsområder. Yderligere viser en høj kompleksitet i et modul, at modulet definerer et *stress point* i systemet, hvilket er defineret som et afgrænset område i koden, hvor store mængder af information flyder [HK81]. Disse områder er svære at modificere på grund af mulige følgeeffekter andre steder i systemet. Henry og Kafura mener at en høj kompleksitet i et modul også kan være et udtryk for, at modulet er for generelt, og at det derfor burde have været designet som flere moduler. Med andre ord, er det et tegn på at modulet har en lav kohæsion. Et modul, i forhold til en datastruktur D, defineres i Henry og Kafura, som alle de procedurer som enten direkte eller indirekte opdaterer eller bruger information fra D. Kompleksisteten af hele modulet defineres som summen af kompleksiteterne af alle de procedurer som indgår i modulet.

Set i forhold til objektorienterede sprog betyder det, at en klasse og et modul ikke er ækvivalente. En klasse kan indeholde metoder som ikke bruger data der er defineret i klassen selv. Dette betyder at metoden falder

uden for mængden af metoder som definerer et modul. Hvis principperne om *information hiding* ikke overholdes, kan data fra et objekt tilgås fra en metode i en anden klasse. Dette betyder at ikke alle metoder som tilgår data, og derfor definerer modulet, er med i modulet. Selvom praksis med information hiding udøves, kan en metode i en fremmed klasse indirekte opdatere information i et objekt gennem dets metoder. Igen vil modulet mangle metoder i forhold til Henry og Kafuras definition.

Imidlertid kan det alligevel give god mening af beregne fan-in/fan-out og kompleksiteten på klasseniveau. Grunden til dette er, at den mindste enhed man er interesseret i at genbruge i det objektorientede paradigme *er* en klasse. I den kontekst Henry og Kafura beskriver kompleksitet, er den mindste interessante genbrugsdel en procedure.

3.6 Kohæsion

Kohæsionsbegrebet i software har sin oprindelse i en artikel af Stevens, Meyers & Constantine [SMC74]. I artiklen betragtes kohæsion indenfor et *modul*.

Et modul defineres som

> ...a set of one or more contiguous program statements having a name by which other parts of the system can invoke it and preferably having its own distinct set of variable names. [SMC74, p.116].

Det som karakteriserer et modul er altså at det kan afgrænses, er sammenhørende og kan kaldes fra andre dele af systemet. Både procedurer og subrutiner generelt karakteriseres i [SMC74] som moduler.

I praksis er dette ækvivalent med funktioner og procedure. Begrebet *element* introduceres, som værende en del af et modul, som f.eks. et enkelt programudtryk. Dette begreb er nødvendigt for at kunne tale om kohæsionen i et modul, altså sammenhængen mellem de enkelte elementer i modulet. Kohæsionen i en funktion er et udtryk for sammenhængen i funktionen. Jo mere kohæsiv funktionen er, jo mere afgrænset og præcis er funktionen i sin opgave. Derfor stræbes efter så høj kohæsion som muligt.

Kohæsionen i et modul er associeret med hvor god modulariteten i modulet er. Jo højere kohæsion, jo bedre modularitet. Man stræber efter høj modularitet, da dette gør det lettere at vedligeholde systemet [YC78]. Yderligere er der tilsyneladende en sammenhæng mellem hvor stærk kohæsionen i et modul er, hvor fejlbehæftet modulet er, og hvor besværligt det er at rette fejlene [McC93]. Jo højere kohæsion, jo færre fejl og jo lettere er fejl at rette.

Yourdon & Constantine har senere behandlet emnet og udvidet den oprindelige taksonomi med begrebet "procedural kohæsion" [YC78]. Den samlede taksonomi indeholder således syv kategorier, rangeret fra lavest til højest, som det ses i tabel 3.4. Kohæsionsbegreberne i tabellen er ikke oversat af

1.	Coincidental cohesion.	Værst
2.	Logical cohesion.	.
3.	Temporal cohesion.	.
4.	Procedural cohesion.	.
5.	Communicational cohesion.	.
6.	Sequential cohesion.	.
7.	Functional cohesion.	Bedst

Tabel 3.4: Yourdon & Constantines kohæsionskategorier [YC78]

hensyn til referencer. Eksemplerne i det følgende, som illustrerer kohæsion, er konstrueret i Java, selvom kohæsionen som gennemgås er defineret i det strukturerede paradigme. Dette udgør ikke et problem, da en metode fra en klasse, og en funktion i et modul kan sidestilles i denne sammenhæng [CK94].

3.6.1 Coincidental cohesion

Coincidental cohesion, eller "tilfældig kohæsion", repræsenterer den dårligste form for kohæsion. I denne kategori er der ingen rationelle relationer mellem elementerne i modulet, og man kunne med rette kalde denne form "ingen kohæsion" i stedet, da der netop ingen sammenhæng er mellem elementerne i modulet. Moduler som har tilfældig kohæsion kan være blevet til, som

```
⋮

private void doStuff(int fahrenheit, int fluidounces) {
   System.out.print(
      fahrenheit + " fahrenheit is " +
      (int)((fahrenheit-32)/1.8) + " celsius"
   );
   System.out.println(
      "  ...and " + fluidounces + " fluid ounces is " +
      fluidounces * 30 + " grams."
   );
}

⋮
```

Figur 3.5: Eksempel på tilfældig kohæsion.

følge af "modularisering", hvor en mængde urelaterede linier kode, som ofte optræder sammen, er blevet flyttet ind i et modul [SMC74]. I forbindelse med genbrug det tydeligt at typen af kohæsion er problematisk. I figur 3.5 vises et eksempel på tilfældig kohæsion.

I eksemplet udskriver metoden `doStuff( ...)` både omregningen af Fahrenheit til Celsius, og fluid ounces til gram. Beregningerne er fuldstændigt

uafhængige af hinanden, men alligevel bundet sammen i en metode. Hvis funktionen, at omregne og udskrive Fahrenheit til Celsius, ønskes genbrugt, følger den uønskede udskrivning af omregning fra fluid ounces til gram med. I forbindelse med vedligeholdelse opstår problemer hvis metoden kaldes flere steder, men det ønskes at ændre rækkefølgen omregningerne bliver udskrevet fra ét af stederne. Ændringen kan ikke umiddelbart foretages, da den vil påvirke alle kald til metoden. Dette forhold er naturligvis altid et problem, men risikoen for at problemet opstår er større, hvis modulet har tilfældig kohæsion [SMC74].

Samtidig er det svært at forstå hvad modulets opgave er. Tilfældig kohæsion betragtes som værende uacceptabel i et system, og er let at fjerne. Dette gøres ved at splitte modulet op i det antal af opgaver som modulet løser.

3.6.2 Logical cohesion

Logisk kohæsion optræder, når en mængde relaterede opgaver er implementeret i samme funktion, men den specifikke opgave funktionen løser i et enkelt kald, er bestemt af en parameter til funktionen. Størst er problemet, hvis de forskellige elementer af funktionalitet deler kode i funktionen, hvilket i følge Stevens et. al. er sandsynligt, da de er implementeret samme sted.

```
⋮

private int convert(int type, int value) {
   int result = 0;

   switch(type) {
      case FAHRENHEIT_2_CELSIUS:
         result = (int) ((value - 32)/1.8); break;
      case FLUIDOUNCES_2_GRAMS:
         result = value * 30; break;
   }

   return result;

}

⋮
```

Figur 3.6: Eksempel på logisk kohæsion.

Som det ses i eksemplet i figur 3.6, er to konverteringsrutiner implementeret i en funktion, hvor et argument til funktionen afgør hvilken konvertering der vil finde sted. I det tilfælde hvor en øget præcision, af f.eks. Fahrenheit til Celsius, ønskes, og returværdien derfor skal ændres til `float` fra `int`, kan

dette ikke umiddelbart lade sig gøre, da begge konverteringer er bundet af hinandens returtype. Det kan ikke lade sig gøre kun at modificere den ene. Dette, at implementationen af ellers urelaterede elementer kommet til at afhænge af hinanden, er et problem i logisk kohæsion. Af denne grund er vedligeholdelse svær i tilfælde med logisk kohæsion. Af samme grund som for tilfældig kohæsion, er genbrug også problematisk. Den ønskede funktionalitet er ikke isoleret, og kan derfor ikke genbruges uden sin kontekst. Yderligere kan interfacet til et modul med logisk kohæsion være sværere at forstå på grund af både den længere parameterliste, men også fordi der kræves viden om de parametre der bruges til at vælge funktion med.

3.6.3 Temporal cohesion

Temporær kohæsion findes når en mængde opgaver er grupperet i samme modul, fordi de skal udføres "på samme tidspunkt". Typiske eksempler på denne form for kohæsion er initialiserings - og oprydningsrutiner, hvor en mængde funktioner skal udføres. I eksemplet i figur 3.7 vises en metode

```
⋮

private void startUp() {
   printer = new Printer();
   printQueue = new PrintQueue();
   numberOfJobsSinceStart = 0;
   currentPrintJob = null;
   createUserInterface();
}

⋮
```

Figur 3.7: Eksempel på temporær kohæsion.

med temporær kohæsion. Det eneste der binder de forskellige elementer i modulet sammen, er at de skal udføres på samme tidspunkt. Formen for kohæsion er dårlig i forbindelse med genbrug, da elementerne i modulet ikke nødvendigvis har noget med hinanden at gøre, og derfor ikke nødvendigvis skal bruges sammen i senere projekter.

3.6.4 Procedural cohesion

I procedural kohæsion, har rækkefølgen i hvilken elementerne bliver udført en betydning, som når der følges en procedure. De enkelte elementer i modulet deler ikke data. Elementerne i modulet udfører urelaterede opgaver, men er bundet sammen fordi de samlet svarer til en sekvens af trin i en samlet

operation. I figur 3.8 har metoden procedural kohæsion, da hvert element svarer til et trin i en procedure der henter oplysninger fra en bruger.

```
⋮

 private void userInputConversion() {
    String input;
    BufferedReader keyboard = new BufferedReader(
       new InputStreamReader(System.in));

    System.out.print(
       "Input integer number to convert (fahrenheit -> celsius): ");

    try {

       input = keyboard.readLine();

       int fahrenheit = Integer.parseInt(input);
       System.out.println(
          fahrenheit + " fahrenheit is " +
          (int)((fahrenheit - 32)/1.8) + " celsius.");

    }
    catch(NumberFormatException nfe) {
       System.out.println("input was  not an integer.");
    }
    catch(IOException ioe) {
       System.out.println("Could not read from keyboard.");
    }
 }

⋮
```

Figur 3.8: Eksempel på procedural kohæsion.

Moduler med procedural kohæsion er ikke gode i forbindelse med genbrug og vedligeholdelse. Hvis modulet i fig 3.8 skulle genanvendes, kan det kun ske i den komplette form, hvilket vil sige med samme brugerinterface, input fra keyboard osv. Med andre ord er moduler med procedural kohæsion meget applikationsafhængige. I eksemplet er det let at se at modulet kan gøres mere kohæsivt ved at konstruere moduler for hver afgrænsede opgave i modulet. Ved at konstruere et `getUserInput()`-modul og et `Fahrenheit2Celsius()`-modul, kan der let laves et modul med *sekventiel* kohæsion.

3.6.5 Communicational cohesion

Communicational cohesion, eller kommunikationskohæsion, er tilstede når elementerne i et modul er relaterede gennem brugen af den samme mængde

input data, eller arbejder på de samme output data. I figur 3.9, bruges det

```
...

private void convert(int value) {

   int celsius = Fahrenheit2Celsius(value);
   int grams = FluidOunces2Grams(value);
   System.out.println(
      value + " fahrenheit = " +
      celsius +
      " celsius.");
   System.out.println(
      value + " fluid ounces = " +
      grams +
      " grams.");
}

...
```

Figur 3.9: Eksempel på kommunikationskohæsion.

samme argument til metoden til beregning af to forskellige ting. Beregningerne er relateret på den måde at begge beregninger baseres på samme input. I forbindelse med genbrug, er denne form for kohæsion stadig for svag. I det tilfælde hvor man ønsker at udskrive konvertering fra fluid ounces til gram fra figur 3.9, men ikke ønsker udskrivning af Fahrenheit til Celsius, umuliggøres genbruget af funktionen. Et andet, ofte brugt, eksempel på kommunikationskohæsion, er hvor et modul modtager en post fra en database som argument, opdaterer posten og derefter gemmer denne i databasen. Forestiller man sig at dette modul bliver brugt gennem systemet, og vedligeholdelsesprogrammøren, på et tidspunkt, får brug for at få opdateret posten med modulet, men i nogen situationer ønsker at undlade at skrive posten tilbage i databasen, opstår der problemer da skrivningen ligger inde i modulet.

3.6.6 Sequential cohesion

Der er sequential cohesion, eller sekventiel kohæsion, i et modul, hvis output fra et kald bruges som input i et efterfølgende kald. Dette er illustreret i 3.10. Her er det mellem elementet `Celsius = Fahrenheit2Celsius(Fahrenheit);` og udskrivningen af den beregnede værdi `Celsius`, der giver sekvensiel kohæsion. Sekvensiel kohæsion lider af, at proceduren er indlejret i modulet. Dette gør modulet applikationsspecifikt og dermed sværere at genbruge.

```
⋮

private void printFahrenheit2Celsius(int fahrenheit) {

   int celsius = fahrenheit2Celsius(fahrenheit);

   System.out.println(
      fahrenheit + " fahrenheit = " +
      celsius +
      " celsius.");
}

⋮
```

Figur 3.10: Eksempel på sekvensiel kohæsion.

3.6.7 Functional cohesion

Functional cohesion, eller funktionel kohæsion, er den stærkeste form for kohæsion. I denne form for kohæsion, er alle elementer relateret til hinanden på den måde, at de alle bidrager til at udføre den samme enkelte funktion. I figur 3.11 ses et eksempel på dette. Som det ses i figur 3.11, har modulet én

```
⋮

private int fahrenheit2Celsius(int fahrenheit) {
   int celsius = (int)((fahrenheit-32)/1.8);
   return celsius;
}

⋮
```

Figur 3.11: Eksempel på funktionel kohæsion.

afgrænset opgave; at konvertere grader Fahrenheit til Celsius. Modulet har ingen andre opgaver og ingen sideeffekter. Alle elementer i modulet bidrager til løsningen af den samme opgave. Dette betyder at modulet er let at genanvende.

3.6.8 Diskussion af kohæsion

Det er afgjort at alle moduler i et system ikke kan have funktionel kohæsion. Flere af kohæsionsformerne er netop problematiske i forbindelse med genbrug, fordi de indeholder applikationspecifik kode og procedurer, for løsning af applikationsspecifikke problemer. En applikation vil altid i nogen grad in-

deholde applikationsspecifik kode, og kan derfor ikke bestå udelukkende af moduler med funktionel kohæsion. Pointen med kohæsion er heller ikke at alle moduler skal være funktionelt kohæsive. Der skal blot stræbes efter så høj kohæsion som muligt.

Yourdon & Constantine mener at moduler som har en af de tre svageste former for kohæsion; tilfældig, logisk og temporær kohæsion, generelt må betragtes som uacceptable, mens de tre stærkeste; kommunikationskohæsion, sekvensiel og funktionel, er acceptable. De skriver intet eksplicit om procedural kohæsion. Temporær kohæsion bliver imidlertid betragtet som acceptabel af McConnell [McC93]. Grunden til at Yourdon & Constantine betragter temporær kohæsion som uacceptabel er, at de elementer som indgår i moduler med temporær kohæsion, som Yourdon & Constantine betragter, er på et ganske lavt abstraktionsniveau. Dette medfører, at ændringer til moduler, som bliver initialiseret i et `startUp()`- modul, kan kræve vedligeholdelse i `startUp()`. Dette giver problemer i forbindelse med vedligeholdelse. McConnell skriver, at hvis man betragter moduler som f.eks. `startUp()` som moduler der organiserer andre aktiviteter, og flytter initialiseringen af relaterede elementer ud i moduler for sig selv, undgås problemerne ved vedligeholdelse, og temporær kohæsion bliver, på dette niveau, acceptabel. I forhold til genbrug er temporær kohæsion dog stadig problematisk, fordi de elementer det initialiseres i et modul med temporær kohæsion vil være applikationsspecifikke. Betragtes moduler som *bruges sammen*, kan temporær kohæsion være et mindre problem for genbrug, hvis modulerne som har temporær kohæsion kun initialiserer moduler som bruges sammen med det genbrugte.

Selvom kohæsion er blevet inddelt i syv kategorier, skelnes der stadig mellem acceptabel og uacceptabel kohæsion. Derfor kunne man spørge om det er nødvendigt at opdele kohæsion i hele syv forskellige kategorier. Skelnen mellem flere kategorier er vigtig, da de enkelte kategorier specificerer *hvad* der er årsagen til graden af kohæsion, og hvor alvorligt problemet er. Yderligere kan opdelingen være med til at give en dybere forståelse af hvordan strukturmæssig kohæsion opstår. Opmærksomhed på dette er relevant under udvikling af software.

Set i forhold til systemdesign, indikerer lav kohæsion, at dekompositionen af systemet er uhensigtsmæssig. Udfra de definitioner der gives af kohæsion, ses det at moduler med lav kohæsion foretager flere uafhængige opgaver. Derfor bør disse moduler opsplittes til flere uafhængige moduler. Denne opsplitning vil have en positiv effekt på både genbrug og vedligeholdelse fordi nogle af disse moduler vil være mindre applikationsspecifikke, og vil have en mere afgrænset funktion.

3.7 Kobling

Kobling kan karakteriseres som *graden af afhængighed mellem dele af et system*. Oprindeligt blev dette begreb også defineret, af Stevens, Myers & Constantine, i det strukturerede paradigme som et mål for styrken af forbindelser mellem moduler i et system [SMC74].

Generelt for koblinger mellem dele af et system er problemet at ændringer som foretages i en del, som følge af vedligeholdelse eller videreudvikling, let kan kræve at der foretages ændringer i de tæt koblede dele. Yderligere betyder kobling, at en del som er interessant at genbruge, ikke kan isoleres fra de dele der ikke skal genbruges.

Hvis der er en høj grad af kobling mellem moduler i et system, skaber dette nogle problemer. Mange forbindelser mellem forskellige dele i et system giver risiko for, at fejl eller ændringer i et modul påvirker mange andre moduler i systemet, og dermed gør omfanget af fejlen, eller omfanget af den mængde kode der skal rettes som følge af en ændring, større end nødvendigt. Samtidig vil stærke koblinger til andre moduler medføre at et modul bliver sværere at genbruge, da de koblede moduler også vil skulle være tilgængelige i det nye projekt, selvom de ikke nødvendigvis er relevante i den nye sammenhæng. Koblingen mellem moduler har således indflydelse på hvor let det er at vedligeholde kode, og hvor let det er at genbruge kode. Forståelsen af et program er også påvirket af kobling, idet et system med mange koblinger er mere komplekst fordi der kræves viden om både de moduler der er koblede og til meningen med koblingerne. Det bliver derved sværere at forstå et system, da en større del skal overskues. En reduktion af koblingerne i systemet vil således også reducere kompleksiteten af systemet [SMC74], og gøre systemet lettere at genbruge. Man er altså interesseret i at minimere koblinger mellem moduler af hensyn til forståelighed, vedligehold og genbrug. Minimeringen af koblingerne består dels i helt at fjerne unødige koblinger og dels i at nødvendige koblinger gøres så svage som muligt.

Som antydet kan et modul kan være afhængig af et andet modul på flere forskellige måder. Alvorsgraden af kobling er forskellig, afhængig af forskellige faktorer. F.eks. er koblingen til et modul lavere hvis der refereres til et moduls interface, end hvis der refereres til interne data i modulet. Hvis flere moduler deler data eller eksterne enheder, deler disse et *fælles miljø*. Som eksempel giver [SMC74] globale variable. Når flere moduler deler det samme fælles miljø, opstår der en kobling mellem disse moduler. Ny brug af, eller ændringer til, et fælles miljø kan påvirke alle de moduler som bruger dette fælles miljø. Myers [Mye78] opdeler kobling i syv kategorier, se tabel 3.5. Af hensyn til referencer, er begreberne fra Myers ikke oversat.

Kategorierne i tabel 3.5 viser en prioriteret ønskeliste med hensyn til kobling mellem moduler. Således er det bedre at have datakobling mellem to moduler end det er at have ekstern kobling. Dog overlapper kategorierne på en måde som gør at en ekstrem form for ekstern kobling kan være bedre

1.	No direct coupling.	Best case
2.	Data coupling.	.
3.	Stamp coupling.	.
4.	Control coupling.	.
5.	Extern coupling.	.
6.	Common coupling.	.
7.	Content coupling.	worst case

Tabel 3.5: Myers koblingskategorier [Mye78]

end en svag form for kontrolkobling.

3.7.1 No direct coupling

Myers skriver at definitionen på "ingen direkte kobling" er, at ingen af de andre koblinger er til stede. I sig selv tilstræbes "ingen direkte kobling" ikke, da det i praksis ikke kan opnås at der ingen kobling er mellem moduler der kommunikerer med hinanden. Enhver form for kommunikation mellem moduler, vil resultere i en form for kobling.

3.7.2 Content coupling

Denne form for kobling optræder hvis et modul refererer til interne dele af et andet modul. Myers beskriver forholdet, at et modul refererer til interne data i et andet modul gennem et offset til modulet. Med andre ord reference til data med direkte offset i begyndelsen af modulets kode og data. Denne kobling er meget stærk, da stort set enhver ændring i data eller kode, før de data i modulet der refereres til, vil kræve ændringer i de kaldende moduler. Content coupling optræder også ved kald som modificerer et andet moduls kode, og hop til labels inde i et andet modul. Som Myers skriver er det i praksis er svært at konstruere denne form for kobling i højniveausprog [Mye78, p.42].

3.7.3 Common/External coupling

Common og external coupling ligner hinanden meget, og opstår begge som følge af brugen af globale variable. Common coupling opstår, når der er et antal moduler som alle refererer en global datastruktur, mens external coupling opstår når de globale referencer er til, hvad Myers kalder homogene data. Homogene data er primitive typer i denne sammenhæng, som f.eks. en `int`. Hvis to, eller flere, moduler kan tilgå og modificere de samme globale data, findes der kobling mellem disse moduler. Myers identificerer en række problemer ved brugen af globale data, som alle påvirker genbrug og vedligeholdelse:

```
⋮

private int value = 0;
private class Person {
   String name;
   String phone;
};
private Person person = new Person();

public void increaseValue() {
   value++;
}
public void decreaseValue() {
   value--;
}

public void setName(String n) {
   person.name = n;
}
public void setPhone(String p) {
   person.phone = p;
}

⋮
```

Figur 3.12: Eksempel på external og common kobling. Metodeparet `increaseValue`/`decreaseValue` udgør external coupling, mens `setName`/`setPhone` udgør common coupling.

- Globale data gør det svært at læse og forstå koden, da ændringer til data, som modulet er afhængigt af, kan ændres i hele programmet. Fejlfinding besværliggøres også af dette.
- Ændringer i globale data fra et modul kan have sideeffekter, som ikke er tydelige i interfacet til modulet. Et modul kan således gøre mere end brugeren af modulet regner med.
- Specielt common kobling giver kobling mellem tilsyneladende urelaterede moduler. To, ellers uafhængige moduler, kan bruge forskellige felter i den samme globale datastruktur, hvilket binder de to moduler sammen.
- Det er svært at genbruge moduler i forskellige kontekster indenfor samme program, da "parametre" til modulet findes i globale data. Det vides ikke på kaldetidspunktet, om disse globale data bruges i en anden sammenhæng, derfor skal de globale data gemmes midlertidigt mens kaldet står på.
- Genbrug i fremtidige projekter besværligøres, da udveksling af data er bundet gennem variabelnavngivningen.
- I common kobling, hvor de globale data ligger i en datastruktur, er man i forbindelse med genbrug blive tvunget til at implementere denne datastruktur selvom datastrukturen ikke er relevant for et nyt projekt.
- Common kobling eksponerer data til moduler som ikke skal bruge dem, fordi en hel datastruktur sendes som parameter selvom om kun en lille del af denne er relevant for modulet.
- Globale data afgrænser udvikleren fra at kontrollere hvorfra data er tilgængelige, så information hiding er umuliggjort.

I eksemplet i figur 3.12, ses både common og external kobling. Modulerne `increaseValue()` og `decreaseValue()` er external koblet, da de deler den samme globale homogene variabel `value`. Modulerne `setName(...)` og `setPhone(...)` er common koblet, da de begge bruger en global variabel `person` af datastrukturen `Person`.

3.7.4 Control coupling

I Myers beskrives control coupling i to afskygninger. Hvis enten et modul kalder et andet, med argumenter som kontrollerer logikken i det kaldte modul, eller hvis en returværdi fra et kaldt modul bruges til at kontrollere logikken i det kaldende modul, optræder control coupling. Typen af kobling er problematisk, da der fra kalderens side kræves viden om intern logik i det modul som kaldes.

Da værdien af parametre til en funktion ofte har betydning på control-flowet i modulet, er det ikke nok at betragte brugen af en parameter for at afgøre om der er tale om control coupling. Det der ifølge Myers er afgørende er, hvordan kalderen betragter den parameter der gives til det kaldte modul. Hvis afsenderen betragter parametren som en enhed der afgør kontrollen i det kaldte element, er der tale om control coupling.

Det skal bemærkes at logisk kohæsion, og control coupling er relateret på den måde, at hvis der findes logisk kohæsion i et modul, vil der ofte være control coupling mellem det modul der kalder modulet med logisk kohæsion, og modulet selv. Et modul som kalder `convert(...)` fra figur 3.6, vil være control coupled til dette modul, fordi kalderen ved at en parameter bestemmer kontrollen i det kaldte modul. Ofte bestemmer værdien af en parameter hvordan modulet reagerer, men det er ikke nødvendigvis kontrol kobling af denne årsag. I figur 3.13 ses et eksempel på et modul, `abs(...)`, som returnerer den absolutte værdi af værdien givet til modulet. Værdien af parametren har indflydelse på kontrol flowet i modulet, men der er ikke kontrol kobling mellem modulerne, da kalderen ikke bevidst styrer kontrollen i `abs(...)`.

```
⋮

public int abs(int value) {
   if (value < 0)
      return -value;
   else
      return value;
}


public NotControlCoupled() {
   for(int i=-2; i < 5; i++)
      System.out.println("Abs(" + i + ")=" + abs(i));
}

⋮
```

Figur 3.13: Eksempel uden kontrol kobling.

Control coupling er ikke svær at undgå. Der argumenteres i [SMC74] for, at selve den logik der afgør hvilken kontrolparameter der skal sendes til funktionen skal implementeres under alle omstændigheder, så i stedet for at implementere en funktion med en kontrolparameter, bør man implementere flere forskellige funktioner, som hver varetager en funktion. Kompleksiteten af den kaldende funktion bliver ikke større, mens kompleksiteten af de kaldte funktion bliver lavere. Samtidig bliver de kaldte funktioner mindre, lettere at forstå og da en opsplitning fjerner afhængigheder mellem de enkelte elementer, bliver disse lettere at genbruge.

```
.
.
.

private int convert(int type, int value) {
   int result = 0;

   switch(type) {
      case FAHRENHEIT_2_CELSIUS:
         result = (int) ((value - 32)/1.8);
         break;
      case FLUIDOUNCES_2_GRAMS:
         result = (int) value * 30;
         break;
   }

   return result;

}
public ControlCoupling() {
   int cmd = getCommand();
   int val = getValue();
   int result;

   result = convert(cmd,val);

   System.out.println(
      val + " converted is: " +
      result
      );
}


.
.
.
```

Figur 3.14: Eksempel indeholdende kontrolkobling.

```
.
.
.

public int fahrenheit2Celsius(int fahrenheit) {
   return (int) ((fahrenheit - 32)/1.8);
}

public int fluidOunces2Grams(int fluidOunces) {
   return (int) fluidOunces * 30;
}

public ControlCouplingRemoved() {
   int cmd = getCommand();
   int val = getValue();
   int result = 0;

   switch(cmd) {
      case FAHRENHEIT_2_CELSIUS:
         result = fahrenheit2Celsius(val);
         break;
      case FLUIDOUNCES_2_GRAMS:
         result = fluidOunces2Grams(val);
         break;
   }

   System.out.println(
      val + " converted is: " +
      result
      );
}


.
.
.
```

Figur 3.15: Eksempel hvor kontrolkobling er fjernet.

I figur 3.14 ses et eksempel hvor et modul `ControlCoupling` er kontrolkoblet til et andet modul `convert(...)`. Som Myers skriver, skal kontrollogikken implementeres under alle omstændigheder, så denne kan flyttes til kalderen, som det illustreres i figur 3.15.

3.7.5 Stamp coupling

Stamp coupling minder noget om common coupling, idet koblingen finder sted gennem en delt datastruktur. Forskellen er, at stamp coupling ikke foregår gennem globale data, men gennem argumenter mellem funktioner. Hvis et modul kalder en funktion i et andet modul, med et argument som er en datastruktur, er der stamp coupling mellem disse moduler. Koblingen har flere konsekvenser. Ændringer i formatet i datastrukturen kan betyde ændringer i de moduler der bruger strukturen. Det skal her bemærkes at moderne sprog som Java ikke har dette problem. En ændring til en klasse som bruges som argument til en metode i en anden klasse, kræver ikke nødvendigvis genoversættelse af klassen som tager argumentet. Problemet er afhængigt af hvordan oversætteren fungerer. I det tilfælde hvor oversætteren genererer referencer relativt til en pointer til en struktur, vil en udvidelse af strukturen kræve genoversættelse af alle brugere af denne struktur, og det er dette Myers beskriver.

Et andet problem ved stamp coupling er, at der eksponeres flere data fra

strukturen end der er nødvendigt. En funktion der modtager en datastruktur som argument har adgang til alle felterne i strukturen selvom kun enkelte data fra strukturen er relevante for funktionen.

```
⋮

public int getAge(Person p) {
   int thisYear = Calendar.getInstance().get(Calendar.YEAR);
   return thisYear - p.birthYear;
}

public StampCoupling() {
   Person person = new Person("Søren","Gaardbo",1971);

   System.out.print(person);
   System.out.println(" Age: " + getAge(person));

}

⋮
```

Figur 3.16: Eksempel på Stamp coupling.

Stamp coupling går ud over generaliteten af det modul der er stamp coupled til. I figur 3.16 ses en metode `getAge(...)`, som tager en `Person` som argument, for at beregne alderen på den person der gives med som argument. I eksemplet skal der ses bort fra, at `getAge()` i et objektorienteret sprog hører hjemme i `Person`-klassen selv. Modulet `getAge` er koblet til `Person`, fordi denne gives med som argument til modulet. En løsning på `getAge()` som tager et fødselsår som argument, vil kunne bruges i andre sammenhænge også.

3.7.6 Data coupling

Der er data coupling mellem to moduler, hvis disse moduler kommunikerer direkte med hinanden, og alle data der udveksles mellem dem er homogene data. Igen er homogene data primitive typer, og der kommunikeres kun gennem disse. Dette er den kategori i Myers taksonomi der er bedst, da der er lavest mulig kobling mellem modulerne, bortset fra *ingen kobling*.

For at reducere stamp coupling til data coupling, kan man konstruere sine funktioner således at de tager alle de data de skal bruge med som homogene argumenter, i stedet for at overføre en struktur. Stamp coupling eksemplet fra figur 3.16 er netop reduceret til data coupling i figur 3.17, hvor `getAge(...)` nu beregner alder på grundlag af et fødselsår og ikke en person. Reduktionen af kobling fra stamp coupling til data coupline betyder typisk, at antallet af parametre til en funktion vil vokse. Myers mener, at antallet af parametre i

```
⋮

public int getAge(int birthYear) {
   int thisYear = Calendar.getInstance().get(Calendar.YEAR);
   return thisYear - birthYear;
}

public DataCoupling() {
   Person person = new Person("Søren","Gaardbo",1971);

   System.out.print(person);
   System.out.println(" Age: " + getAge(person.birthYear));

}

⋮
```

Figur 3.17: Eksempel på datakobling.

et system der er veldekompositioneret, dog vil begrænse sig til højst seks til syv parametre. Det skal bemærkes, at antallet af parametre til en funktion, også bidrager til kompleksiteten af interfacet til funktionen [YC78]. Således kan der være en konflikt imellem forsøget på at reducere koblingen mellem moduler, og den, af brugeren, perciperede kompleksitet i interfacet.

3.7.7 Diskussion af kobling

I forhold til genbrug er kobling interessant, da lav kobling til et modul betyder, at modulet, i hvert fald strukturelt, er lettere at bruge i en ny kontekst. Jo lavere kobling i et modul, jo lettere er det for andre moduler at kalde dette modul. Vedligeholdelsen af moduler med lav kobling kan blive nemmere, da ændringer foretaget i moduler med lav kobling kan betyde færre rettelser i de moduler som er koblet til det der modificeres.

Grænsen for hvilke grader af koblinger der er acceptable er flydende omkring stamp kobling [McC93][3] Ingen koblinger som bruger globale data er acceptable, og kontrolkobling er ikke ønskværdig af samme årsager som for logisk kohæsion; der kræves viden om et moduls indre virkemåde.

McConnell [McC93] mener at en stamp kobling hvor alle, eller næsten alle, felter i en struktur benyttes, er acceptabel, mens en stamp kobling som kun bruger dele af en struktur ikke er acceptabel [McC93]. Der argumenteres ikke for dette synspunkt i, men begrundelsen er formentlig, at hvis et modul bruger næsten alle de informationer der ligger i en struktur, er modulet og data bundet stærkt sammen. Samtidig eksponeres der ikke mere data for modulet end nødvendigt, og endeligt vil en reduktion til datakobling

[3]McConnell kalder denne form for kobling for *Data-Structure Coupling.*

betyde at interfacet til modulet kan blive meget stort på grund af de mange parametre til modulet. Da mange parametre også komplicerer et interface, kan der være fornuft i at acceptere denne form for kobling.

3.8 Sammenfatning

I dette kapitel blev en mængde softwaremål for det strukturerede paradigme gennemgået. Følgende elementer opsummerer kapitlet:

- Størrelsesmål i forskellige kombinationer har været brugt til at udtrykke flere forskellige egenskaber ved kildekode og udviklingen af disse. F.eks. programmørers produktivitet. Størrelsesmålene er ikke anvendelige til at sige noget om genbrug og vedligeholdelse.

- McCabes kompleksitetsmål er baseret på kontrolstrukturen i kildekoden. Målet kan sige hvor mange afprøvninger der skal til for at få afprøvet alle veje i kontrolstrukturen. Samtidig siger målet noget om hvor kompleks programkoden er, og dermed hvor svær koden er at forstå og vedligeholde.

- Halstead's mål er baseret på optælling af operatorer og operationer i et program. Ud fra disse optællinger defineres forskellige mål. Selve optællingen af de værdier der indgår i målet er kun vagt formuleret hvilket betyder at forskellige kommercielle produkter oplyser forskellige mål for Halstead. Uden klare regler for optællingen er målet ikke brugbart.

- Fan-in/fan-out er mål for dels hvor mange andre moduler der kalder det aktuelle modul (fan-in), og dels hvor mange moduler det aktuelle modul kalder (fan-out). Målet siger noget om dels hvor skøbelige afhængige moduler er af det aktuelle modul (fan-in), og dels hvor skrøbeligt det aktuelle modul er overfor ændringer i andre dele af systemet (fan-out). Af de gennemgåede mål er det dette der har det største potentiale i forbindelse med identifikation af kode der er god eller dårlig i sammenhæng med genbrug og vedligeholdelse.

- Kohæsionen siger noget om sammenhængen i et modul. Der stræbes efter høj kohæsion da dette indikerer at modulet har en klar sammenhængende funktion. Lav kohæsion er et udtryk for at et modul har får mange ansvarsområder og bør opdeles i mindre moduler med hvert sit ansvarsområde. Kohæsionen som den er blevet præsenteret er ikke umiddelbart målbar.

- Koblingen siger noget om afhængighederne mellem forskellige moduler. Det er ønsket at have så lav kobling som muligt, da dette begrænser

effekten af ændringer i et modul. Koblingen er tæt relateret til fan-in/fan-out.

Kapitel 4

Softwaremål i OOP

I dette kapitel gennemgås nogle softwaremål for det objektorienterede paradigme. Ønsket er, at finde mål der kan identificere kode, som enten er god eller dårlig i forhold til genbrug og vedligeholdelse. Disse mål skal samtidig være uafhængige af metadata, da det er ønsket at en analyse af kildekoden kun skal være afhængig af kildekoden selv.

Konklusionen på kapitlet er, at der er gjort en del forsøg på at opstille softwaremål i det objektorienterede paradigme. Der blev gennemgået to komplette *metrics suites*: Chidamber & Kemerer's [CK94] og *MOOD* af e Abreu et al. [eAM96].

Der er gjort specielt mange forsøg på at opstille mål for kohæsionen i en klasse. Kohæsionen er interessant, fordi et kohæsionsmål kan udtrykke at en klasse indeholder urelaterede ansvarsområder, og derfor bør opdeles i flere klasser. Det er ikke selve kohæsionen der måles, men egenskaber ved kildekoden der menes at være udtryk for kohæsion. På trods at kohæsions "abstrakthed" virker begrebet interessant til automatisk evaluering af kildekode.

Kobling i systemer kan være en hindring for genbrug og vedligeholdelse. Derfor virker koblingsmål interessante i forbindelse med udarbejdelsen af et system til automatisk evaluering af kildekode.

4.1 Introduktion

Det objektorienterede paradigme introducerer nogle muligheder for softwaredesign, som ikke findes i de strukturerede sprog. Derfor har der været behov for specifikt at betragte det objektorienterede paradigme, for at finde de problemer som de nye muligheder fører med sig. I det følgende gennemgås et antal mål som er opstillet specifikt for det objektorienterede paradigme. Det skal bemærkes at der findes en stor mængde mål som udtrykker forskellige egenskaber ved et objektorienteret system. I det følgende er udvalgt to *metric*

suite's, som bliver gennemgået: Chidamber & Kemerer's kompleksitetsmål [CK94], og MOOD målene fra e Abreu et al. [eAC94, eAGE95, eAM96]. Grunden til at Chidamber & Kemerer's mål gennemgås er, at denne mængde mål formentlig er et af de bedst kendte, og mest refererede i litteraturen. MOOD målene gennemgås fordi de gennemgår mål som afspejler egenskaber som er specifikke for det objektorienterede paradigme, som samtidig ikke behandles af Chidamber & Kemerer.

Gennemgangen af følgende mål, skal ikke betragtes som udtømmende for mål i det objektorienterede paradigme.

4.2 Chidamber & Kemerer's kompleksitetsmål

Den formentlig bedst kendte samling af kompleksitetmål for objektorienterede systemer, er Chidambrer & Kemerer's *metrics suite* [CK94]. Udgangspunktet for Chidamber & Kemerer var, at hvis det skal kunne lade sig gøre at spore forbedringer i et system, skal der findes en måde at foretage målinger i systemet, som afspejler forbedringerne. Målene i deres *metrics suite*, bliver gennemgået i det følgende, dels sammen med nogle af de problemer og kritikpunkter der er blevet rejst, og dels sammen med andre forslag til hvordan relaterede egenskaber kan findes i software.

4.2.1 Weighted Methods Per Class

Weighted Methods per class, WMC, eller vægtede metoder pr. klasse, er et mål for den samlede kompleksitet for en klasse. Betragtes en klasse C med metoderne $M_1, ..., M_n$, med kompleksiteter for hver metode $c_1, ..., c_n$ tilhørende C, da defineres WMC som:

$$WMC = \sum_{i=1}^{n} c_i$$

Kompleksitetsværdierne $c_1, ..., c_n$ bliver ikke defineret i [CK94], men det foreslås at kompleksiteten blot sættes til én for hver metode, hvilket betyder at målet er det samme som antalet af metoder i klassen, eller at der benyttes et traditionelt mål for kompleksiteten i metoderne. Der kan argumenteres for blot at sætte kompleksiteten til én og lade målet afspejle antallet af metoder, da mange metoder i en klasse kan være en indikation af, at klassen er meget specialiseret, og dermed svær at genbruge [CK94]. Da det at konstruere en metode minder meget om at konstruere et proceduralt program, kan et traditionelt mål benyttes, f.eks. McCabes cyklomatiske kompleksitet, eller Halsteads mål [CK94, Ben98].

Målet er blevet kritiseret af Churcher & Shepperd af flere årsager. Dels rejses der tvivl om hvorvidt det er rimeligt at bruge traditionelle mål for

metoderne [CS95b]. Den primære anke er at metoder i objektorienterede systemer typisk er meget mindre end funktioner i det strukturerede paradigme, hvilket i følge [CS95b] gør det usandsynligt at f.eks. McCabe er anvendelig i denne sammenhæng. Problemet uddybes ikke nærmere i [CS95b], men hvis metoder generelt er mindre end de funktioner der findes i det strukturerede paradigme, kan der være mindre kontrollogik i metoderne, hvilket giver en lavere cyklomatisk kompleksitet. *Getter* og *setter*-metoder indeholder ofte slet ingen kontrollogik, derfor vil disse have cyklomatisk kompleksitet på én. Spørgsmålet er om forskellen på paradigmerne er et reelt problem. Da WMC netop summerer kompleksiteten for samtlige metoder i klassen, kan det tænkes at nogle af de traditionelle mål stadig giver et udmærket billede af kompleksiteten i en klasse. I øvrigt mener Jorgensen et al. at kompleksiteten for objektorienterede systemer generelt er højere end den er for strukturerede [JFF+02]. Der er altså enighed om, at der kan være en forskel på resultaterne på tværs af paradigmerne, men ikke helt enighed om hvordan forskellen giver sig udslag. Hvis der ikke sammenlignes kompleksiteter på tværs af paradigmer, må nogle af de traditionelle mål stadig kunne benyttes.

Curcher & Shepperd kritiserer yderligere Chidamber & Kemerer for ikke at tage sprogspecifikke hensyn [CS95a]. Specifikt kritiseres det, at Chidamber & Kemerer ikke forholder sig til problemstillingen at antallet af metoder i en klasse, kan opgøres på flere måder. Problemet der rejses er, om *nedarvede metoder* skal tælles med i en klasse, hvordan man forholder sig til *operatorer* som defineres i klassen etc. En del af pointen er, at det kan være at måden man opgør antallet at metoder, skal være afhængig af hvad målet skal bruges til. I forbindelse med koblingsmål, er kun synlige metoder, dvs. dem der er en del af klassens interface, relevante.

I *Authors' Reply* til [CS95a], skriver Chidamber & Kemerer, at der tælles efter det princip, at hvor der er gjort en designindsats, medtælles metoden. Med andre ord medtælles nedarvede metoder ikke, men operatorer som bliver defineret for en klasse, skal tælles med.

Churcher & Shepperd foreslår en mængde måder hvorpå antallet af metoder kan optælles [CS95b]. Tabel 4.1 fra [CS95b] summerer måderne, med forslag til mulig navngivning af hver optællingsstrategi.

Chidamber & Kemerers' svar på kritikken indeholder deres bud på hvordan antallet af metoder i en klasse skal opgøres. Deres optællingsstrategier er markeret med *kursiv* i tabel 4.1. Det skal bemærkes at *message degeneray* i tabel 4.1, dækker over det forhold, at flere metoder kan være navngivet ens, men have forskellige signatur. Dette er kendt som *overloading*. Et eksempel er metoderne `max` i `java.lang.Math`-klassen. Metoderne `max` har følgende forskellige signaturer:

```
static double max(double a, double b)
static float max(float a, float b)
static int max(int a, int b)
```

Concept	**Candidate Term**	**Note**
Inheritance	Gross, full, extended	All generations included.
	Nett, compact	Only the local increment included.
	n-extended	n generations included.
Access Mode	restricted	Language-dependent restrictions on acess.
	unrestricted	Access mode ignored.
Overriding	inclusive, distinguished	Both overriding and overridden method included
	exclusive, non-distinguished	Overridden methods excluded
Message Degeneracy	condenced	Treating degenerate members as one.
	expanded	Treating degenerate members as one.
Operators	*augmented*	Including operators.
	reduced	Excluding operators.

Tabel 4.1: Mulige metodeoptællingsstrategier [CS95b]

```
static long max(long a, long b)
```

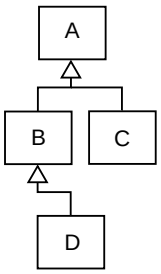
`max(...)` returnerer den største af parametrene `a` og `b`. Den af de ovenstående metoder der bliver kaldt, er afhængig af typen metoden kaldes med. Spørgsmålet som rejses i [CS95b] er om ovenstående bør tælles med som fire distinkte metoder, eller som en metode.

Der står intet i Chidamber & Kemerer om hvorvidt deklarationen af en metode (public, protected etc.), har indflydelse på optællingen af metoder i en klasse, men eftersom der i svaret til [CS95a] står, at alt der udgør en *ekstra designindsats* skal medtælles, antages det at deklarationen af en metode er underordnet i følge Chidamber & Kemerer, såfremt metoden er erklæret i den klasse der tælles metoder for.

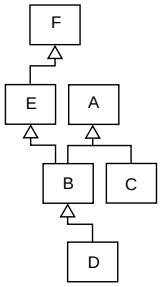
4.2.2 Depth of Inheritance tree (DIT)

Depth of Inheritance tree, *DIT* eller, dybden af klassehierarkiet, måles som den største afstand fra en knude (klasse), til roden i hierarkiet. Hvis der, i det aktuelle sprog, er mulighed for multipel nedarvning, måles dybden som den størst mulige afstand fra noden til en rod. Se figur 4.1 og 4.2.

Chidamber & Kemerer argumenterer for tre synspunkter i forbindelse med dybden af klassehierarkiet.



Figur 4.1: DIT for hierarki uden multipel nedarvning. DIT(D)=2



Figur 4.2: DIT for hierarki med multipel nedarvning. DIT(D)=3

1. Virkemåden af en klasse bliver sværere at forudsige, jo dybere i klasseshierarkiet den befinder sig, da klassen kan nedarve et større antal metoder.

2. Klassehierarkiet vil være mere komplekst fordi flere metoder og klasser er involveret.

3. Jo dybere en klasse ligger i hierarkiet, jo større er potentialet for genbrug genem nedarvede metoder.

I to systemer, undersøgt af Chidamber & Kemerer, viste det sig, at klassehierarkierne generelt var relativt lave og *top-tunge*, hvilket betyder at der er relativt mange klasser i toppen af hierarkiet. I et senere forsøg [CDK98], hvor tre systemer blev undersøgt, viste den samme tendens sig. Klasserne der blev undersøgt havde en tendens til at have små DIT-værdier. På grundlag af disse data spekuleres det, at designerne måske fravælger muligheden for genbrug gennem nedarvning, mod at få en simplere og lettere forståelig model at arbejde med.

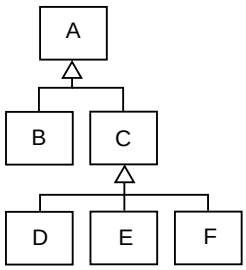
4.2.3 Number of Children

Number of Children *NOC* eller, antal børn, til en klasse defineres som antallet af umiddelbart efterfølgende subklasser til en klasse. I figur 4.3 vises et klassehierarki, med følgende værdier for NOC: NOC(A)=2, NOC(B)=0, NOC(C)=3, NOC(D,E,F)=0. Chidamber & Kemerer nævner tre interessante synspunkter i forbindelse med NOC:

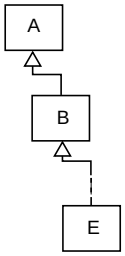
1. Jo flere børn til en klasse, jo mere genbrug gennem nedarvning.

2. Mange børn til en klasse kan tyde på at nedarvning bruges forkert.

3. Mange børn giver indtryk af hvor stor en effekt en klasse har på det samlede design. Mange børn kan kræve flere afprøvninger af metoderne i forældreklassen.



Figur 4.3: NOC for klasser.



Figur 4.4: "Sekventielt" klassehierarki.

Som det ses af definitionen, tages der ikke hensyn til, at en klasses børn også kan have børn så alene kan resultater fra målet virke misvisende. Et klassehierarki som det illustreret i figur 4.4 kan give indtrykket at en ændring i den øverste klasse `A` i hierarkiet ikke vil påvirke mange andre klasser, da NOC for `A` kun er én. Som det ses i figuren, er dette ikke tilfældet, da mange klasser inddirekte nedarver fra `A`. Sammen med DIT kan NOC give et overordnet billede af klassehierarkierne, men det kan ikke ses fra disse mål, hvor mange andre klasser der potentielt bliver påvirket af en ændring i en enkelt klasse.

4.2.4 Coupling between object classes

Coupling between objects CBO, eller koblingen mellem klasser, defineres som antallet af andre klasser en specifik klasse er koblet til. En klasse `A` er koblet til en klasse `B`, hvis `A` refererer til en metode eller en instansvariabel som er defineret i `B`. $CBO(C)$ findes ved blot at tælle antallet af distinkte klasser som C refererer til. Dette er identisk med *fan-out* som beskrevet i kapitel 3.

Tre problemer ved kobling fremhæves i [CK94]:

1. Høj kobling hindrer genbrug. Jo mindre koblingen er fra en klasse til en anden, jo større er muligheden for at genbruge klassen.

2. Jo større kobling der er mellem klasser, jo større følsomhed for forandringer er der i designet. Derfor bliver vedligeholdelsen af systemet sværere.

3. Kobling kan bruges som mål for hvor besværligt afprøvningen af systemet vil være. Jo højere kobling, jo stærkere en afprøvning skal systemet gennemgå.

Koblingsmål kan ifølge Chidamber & Kemerer sige noget om genanvendeligheden og vedligeholdelsesvenligheden af en klasse.

Da kobling er et vigtigt punkt i forsøg på at måle muligheden for genbrug og vedligeholdelse, bliver kobling og forskellige måder at måle dette gennemgået i et senere afsnit.

4.2.5 Response For a Class

Response For a Class RFC, eller udfaldsrummet for en klasse, defineres som:

$$RFC = |RS|$$

hvor RS er response set for klassen. Det kan udtrykkes som:

$$RS = \{M\} \cup_{all\ i} \{R_i\}$$

Hvor $\{R_i\}$ er mængden af alle de metoder der kaldes i metoden i, og $\{M\}$ er mængden af metoder i klassen [CK94]. Response-mængden er mængden af metoder som potentielt kan blive kaldt når der sendes en besked til en klasse. Elementerne i respons-mængden indeholder, som det ses, kun kald af metoder på første niveau. De kald fra en klasse til en anden klasse, som igen kalder nye metoder, tælles af praktiske årsager ikke med. Et højt mål for RFC tyder på at en klasse er kompleks, fordi det antal metoder der potentielt kan kaldes er stort [CK94]. Hvis argumentet er at kompleksiteten af en klasse er stor fordi den potentielt kan kalde mange andre metoder, er det en fejl ikke at medtælle metodekald i flere led. Hvis en klasse kun kalder én metode i én anden klasse bliver RFC-værdien lav. Dette er på trods af at den kaldte metode kan have en meget stor RFC-værdi. Ses der bort fra RFC i dybere niveauer, gives der ikke et reelt billede af hvor hvor mange metoder der potentielt kan kaldes.

Man kan argumentere for at en stor RFC-værdi kan være et problem for genbrug og vedligeholdelse, da klassen enten er meget stor, målt i antal metoder, eller kommunikerer med mange andre klasser, og dermed er koblet til disse. Yderligere kan RFC være en indikation af, at klassen har mange ansvarsområder, og dermed har lav kohæsion. Det er dog ikke umiddelbart muligt at se hvad årsagen til den høje RFC skyldes.

4.2.6 Lack of Cohesion in Methods (LCOM)

Lack of Cohesion in Methods $LCOM$, eller manglende kohæsion mellem metoder, siger noget som sammenhængen i en klasse. Kohæsionen i en klasse siger noget om hvor sammenhængende funktionaliteten i en klasse er. LCOM er formelt defineret i på følgende måde [CK94]: Lad C være en klasse med n metoder $M_1, ..., M_n$, og $\{I_j\}$ være mængden af instansvariable brugt i metoden M_i. I alt er der n sådanne mængder $\{I_1\}, ..., \{I_n\}$. Lad $P = \{(I_i, I_j) | I_i \cap I_j = \emptyset\}$ og $Q = \{(I_i, I_j) | I_i \cap I_j \neq \emptyset\}$. Hvis alle n mængder $\{I_1\}, ..., \{I_n\}$ er $\emptyset$, dvs. der bruges ingen instansvariable i nogen af metoderne, sættes $P = \emptyset$.

$$\begin{aligned} LCOM &= |P| - |Q|, hvis |P| > |Q| \\ &= 0\ ellers \end{aligned} \tag{4.1}$$

Med andre ord; P er mængden af metodepar, som *ikke har* brugen af nogen instansvariable fælles og Q er mængden af metodepar som *har* brugen af instansvariable til fælles. Jo større LCOM bliver, jo flere ikke-relaterede metoder findes i klassen, målt efter deres fælles brug af instansvariable. Jo mindre LCOM er, jo mere kohæsiv er klassen. LCOM bliver 0 når der enten er flere metodepar der gør brug af samme instansvariable, end der er metodepar der ikke gør, eller når mængderne er lige store $|P| = |Q|$. Med andre ord betyder en klasse med LCOM=0 ikke at, klassen har maksimal kohæsion, da to klasser, begge med LCOM=0, kan have forskellig kohæsion. LCOM er baseret på "ensartethed" mellem metoder i en klasse. I Chidamber & Kemerer, har man brugt referencer til de samme instansvariable som et udtryk for ensartethed. Hvis et antal metoder alle arbejder på de samme data, er disse metoder mere ensartede end metoder der arbejder på disjunkte mængder af instansvariable.

Chidamber & Kemerer observerer følgende i forbindelse med LCOM:

- Jo højere kohæsion i en klasse, jo bedre er mulighederne for indkapsling.
- Lav kohæsion tyder på at en klasse har urelaterede ansvarsområder og derfor bør splittes op i flere klasser.
- Lav kohæsion giver høj kompleksitet, og dermed større risiko for fejl.

Da antallet af metoder bruges i beregningen af LCOM, lider LCOM under nogle af de samme problemer som WMC, da selve opgørelsen af antallet af metoder i en klasse, kan gøres på flere måder. Spørgsmålet om hvilke metoder der skal medtælles bliver ikke behandlet af Chidamber & Kemerer.

Som det er tilfældet for kobling, er kohæsion interessant når mulighederne for genbrug skal belyses. Samtidig er LCOM blevet kritiseret af flere årsager, og der er opstillet flere forskellige kohæsionsmål i det objektorienterede paradigme. Derfor bliver begrebet gennemgået grundigt i afsnit 4.5.

4.3 MOOD - Metrics of Object-Oriented Design

e Abreu & Carapuça definerer en mængde mål, som de kalder *MOOD - Metrics of Object-Oriented Design* [eAC94, eAGE95, eAM96]. Denne samling mål, blev konstrueret for at kunne evaluere egenskaber som er specifikke for det objektorienterede paradigme, som nedarvning, indkapsling og polymorfisme. Da flere af disse egenskaber netop skulle give mulighed for øget genbrug, er disse mål interessante at betragte. Resultaterne af samtlige mål i MOOD, er faktorer, som ligger mellem nul og én. Dette betyder at målene kan sammenlignes på tværs af systemer, da de er størrelsesuafhængige.

I målene benyttes følgende begreber [eAC94]:

Children Count $CC(C_i)$ Antal børn til C_i. Disse er klasser som nedarver *direkte* fra C_i.

Descendants Count $DC(C_i)$ Antallet af efterkommere til C_i, dvs. alle klasser der nedarver *direkte* eller *indirekte* fra C_i.

Parents Count $PC(C_i)$ Antal forældre til C_i. I Java vil denne værdi altid være én, da Java ikke tillader multibel nedarvning og alle klasser nedarver fra `Object`.

Ancestor Count $AC(C_i)$ Antal forfædre til C_i. Bemærk at dette Java vil være det samme som dybden af klassen i klassehierarkiet, eller $DIT(C_i)$.

$M_d(C_i)$ Antallet af metoder der bliver *defineret* i $C(_i)$.

$M_n(C_i)$ Antal *nye* metoder i C_i). Nye metoder er metoder som ikke overskriver eksisterende metoder fra en forfader.

$M_i(C_i)$ Antal *nedarvede* metoder i C_i. Dette er antallet af tilgængelige metoder i C_i som er defineret i en forfader til C_i, og ikke overskrives i C_i.

$M_o(C_i)$ Antal overskrevne metoder i C_i, dvs. antallet af metoder som overskriver en metode defineret i en forfader til C_i.

$M_a(C_i)$ Antallet af tilgængelige metoder i C_i, dvs. antallet af metoder som kan kaldes i klassen C_i.

TC er det totale antal klasser.

4.3.1 Method Hiding Factor & Attribute Hiding Factor

Method Hiding Factor MHF og Attribute Hiding Factor AHF, eller metode - og attributsynlighed, beregnes i forhold til hvor mange metoder, hhv. attributter der samlet findes i systemet.

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)} \tag{4.2}$$

hvor,

$$V(M_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(M_{mi}, C_j)}{TC}$$

og

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1 & hvis\ C_j\ kan\ kalde\ M_{mi} \\ 0 & ellers \end{cases}$$

I ovenstående bliver tælleren den samlede "usynlighed" af metoderne, og nævneren er det samlede antal metoder defineret i hele systemet som undersøges. Hvis alle metoder er synlige alle steder fra, vil MHF blive nul.

	$V(M_{mi})$
public	1
protected	$\frac{1+DC(C_i)}{TC}$
private	$\frac{1}{TC}$

Tabel 4.2: $V(M_{mi})$ for forskellige deklarationer af metoder i C++ [eAM96]

e Abreu et al. viser hvordan synligheden kan findes i C++ kode, alene ud fra deklarationen af den enkelte metode eller attribut. De samme regler gælder for Java. Synlighederne kan ses i tabel 4.2 fra [eAM96]. Som det ses, kan $V(M_{mi})$ findes udfra deklarationen af metoden, antallet af samtlige klasser i systemet og antallet af efterfølgere til klassen. I tabel 4.2 er $DC(C_i)$ antallet af efterkommere til klassen C_i.

Synligheden af attributter i en klasse, findes på samme måde som det er tilfældet for metoder. Attribute Hiding Factor (AHF) beregnes på følgende måde:

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)} \tag{4.3}$$

hvor,

$$V(A_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(A_{mi}, C_j)}{TC}$$

og

$$is\_visible(A_{mi}, C_j) = \begin{cases} 1 & hvis\ C_j\ kan\ referere\ A_{mi} \\ 0 & ellers \end{cases}$$

Som for $V(M_{mi})$ kan $V(A_{mi})$ findes for $C++$ som beskrevet i tabel 4.3

	$V(A_{mi})$
public	1
protected	$\frac{1+DC(C_i)}{TC}$
private	$\frac{1}{TC}$

Tabel 4.3: $V(M_{mi})$ for forskellige deklarationer af attributter i C++ [eAM96]

Målene *MHF* og *AHF* er interessante i genbrugssammenhænge af samme årsag som de er det i vedligeholdelsessammenhænge. Målene siger noget om hvor sårbar en klasse er for ydre påvirkning. Samtidig viser målene om principperne for *information hiding* er brudt. Hvis *AHF* er lav, er mange interne

attributter af en klasse eksponeret for omverdenen. I princippet bør *AHF* være én for samtlige klasser, da dette indikerer total information hiding. M-HF bør i følge e Abreu et. al. ikke være for lav (dette betyder at et stort antal metoder er synlige i forhold til det samlede antal metoder), da en klasse ofte indeholder en mængde metoder, som er implementeret udelukkende til "internt brug". Gemmes disse metoder i klassen, i stedet for at eksponere dem i interfacet, begrænses interfaces, hvilket gør klassen nemmere at overskue og forstå.

4.3.2 Method Inheritance Factor & Attribute Inheritance Factor

Method Inheritance Factor MIF og Attribute Inheritance Factor AIF, eller metode - og attributnedarvningsfaktor angiver hvor mange af metoderne hhv. attributterne der er nedarvet, i forhold til hvor mange metoder hhv. attributter der totalt findes i systemet. *Method Inheritance Factor (MIF)* og *Attribute Inheritance Factor (AIF)* defineres således [eAM96]:

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} \tag{4.4}$$

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)} \tag{4.5}$$

Det gælder at$M_a(C_i) = M_d(C_i) + M_i(C_i)$. Overskrevne metoder i en klasse tæller med som en defineret metode for pågældende klasse, og ikke som nedarvede metoder.

Det totale antal nedarvede metoder, og totale antal tilgængelige metoder, kan beregnes på følgende måde i et klassehierarki [eAC94]:

$$TM_i = \sum_{k=1}^{TC} \Big[M_n(C_k) \times DC(C_k) - M_o(C_k) \Big] \tag{4.6}$$

$$TM_a = \sum_{k_1}^{TC} \Big[M_n(C_k) \times \Big[1 + DC(C_k) \Big] \Big] \tag{4.7}$$

I formel 4.6 skal det bemærkes, at $M_n(C_k) \times DC(C_k)$ beregner antallet af klasser hvor nye metoder i C_k potentielt kan bruges, dvs. de klasser hvor metoderne ikke bliver overskrevet.

Det ekstreme tilfælde hvor MIF=0, er der ingen genbrug gennem nedarvning. Enten er der intet klassehierarki i systemet, eller også bliver samtlige metoder i subklasserne overskrevet. Er MIF=1, bliver der ikke tilført nye

metoder i subklasser. Det vil ikke fremgå af MIF, om der bliver tilføjet attributter til klassen, men funktionaliteten i klassen bliver ikke udbygget.

I e Abreu et al. argumenteres for, at "fornuftige" værdier af MIF hverken vil befinde sig omkring nul, eller omkring maksimumværdien én [eAC94, eAM96]. Bliver værdien for lav, udnyttes genbrug gennem nedarvning slet ikke, og hvis værdien er for stor, kan det være et tegn på, at klassehierarkiet er meget stort, og dermed besværliggør genbrug, vedligeholdelse og afprøvning. e Abreu et al. finder en moderat negativ korrelation med fejltætheden, så fejltætheden går lidt ned, når MIF bliver større. Samtidig findes en høj negativ korrelation mellem værdien, og hvor meget tid der skal bruges på at rette systemet. Med andre ord, tyder forsøget gennemgået i [eAM96] på, at jo større værdien af MIF bliver, jo mindre tid skal der bruges på rettelser.

AIF, nedarvning af atributter, viser ikke den sammenhæng med "fejltætheden". e Abreu et. al. kan ikke konkludere noget om AIF udfra de data de kom frem til i [eAM96]. I [eAGE95] skriver de at en fornuftig værdi af AIF formentlig ligger i et interval mellem ekstremerne snarere end i den ene eller anden ende af spektret.

4.3.3 Koblingsfaktor

Coupling Factor COF, eller koblingsfaktoren, defineres af e Abreu et al. for at udtrykke koblinger i et system. Som tidligere er en kobling en reference fra en klasse til en anden. Denne reference kan være et kald til en metode i klassen, eller brugen af en variabel fra klassen. Blandt referencer tæller ikke nedarvede metoder og attributter, så brugen af en metode som er arvet fra en forældreklasse bidrager ikke til kobling. Koblingsfaktoren COF er defineret for *hele* systemet med følgende formel:

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC} \tag{4.8}$$

hvor:

$$is\_client(C_c, C_s) = \begin{cases} 1 & hvis\ C_c\ er\ klient\ til\ C_s \\ 0 & ellers \end{cases}$$

Som det ses kræves at $C_c \neq C_s$, således at et kald fra en klasse til en metode der er defineret i klassen selv ikke bidrager til koblingen. COF er altså forholdet mellem antallet af faktiske koblinger i systemet og antallet af mulige koblinger i systemet. Tælleren i 4.8 er det totale antal koblinger mellem klasserne, mens nævneren er antallet af teoretisk mulige koblinger. Koblinger kan i følge e Abreu et. al. opstå på følgende måde i $C++$ [eAM96]:

- Kald til en metode (*member function*) i en anden klasse gennem dennes interface.

- Kald til en synlig eller skjult metode i en anden klasse, som følge af brugen af *friend*
- Gennem allokering/deallokering gennem et objekts *constructor / destructor*.
- Gennem reference til en serverklasse som *datamember* eller som *formel parameter* i et kald til en funktion.

Koblingsfaktoren siger således noget om hvor høj koblingen i systemet som helhed er. Der stræbes efter en lav koblingsfaktor i forbindelse med genbrug og vedligeholdelse. Dog er en koblingsfaktor på nul hverken ønskværdig eller realistisk. Dette ville betyde at ingen klasser kendte til nogen andre klasser. I et ekstrem kunne man forestille sig én klasse som implementerer et helt system. Denne klasse vil have en koblingsfaktor på nul, da der ikke findes andre klasser at have koblinger til. Til gengæld ville denne ene klasse formentlig være en stor monolitisk og meget kompleks klasse med meget lav kohæsion. Dette vil være et udtryk for en ufornuftig dekomposition af systemet.

e Abreu et al. introducerer i deres oprindelige artikel begrebet *Clustering Factor*, som udtrykker antallet af *clusters* i forhold til antallet af klasser i systemet [eAC94]. Tanken er, at hvis man betragter koblinger som en graf, hvor knuderne i grafen er klasser, og kanterne mellem knuderne repræsenterer koblinger mellem de to klasser, vil der findes disjunkte mængder af usammenhængende klasser [eAC94]. Det totale antal *clusters* i systemet, vil være antallet af grafer i det samlede system. Hver af disse grafer vil repræsentere et potentielt genbrugeligt modul, fordi det ikke er koblet til andre dele af systemet. Desværre behandler e Abreu et al. kun dette overfladisk, og det har ikke været muligt at finde en dybere behandling af emnet fra e Abreu et al. i senere artikler. Måske har det ikke været muligt at argumentere for at clustering faktoren hænger sammen med kvaliteten i systemet. Formentlig vil de fleste systemer også bestå af mindst én kobling, da ethvert delsystem i det samlede system vil kommunikere med det omkringliggende system på den ene eller den anden måde.

4.3.4 Polymorphism Factor

e Abreu et al.'s Polymorphism Factor POF eller, polymorfismefaktor, er et mål for hvor meget der gøres brug af polymorfi i systemet. Målet er defineret således [eAM96]:

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]} \tag{4.9}$$

I formel 4.9 er $M_o(C_i)$ antallet af overskrevne metoder i C_i, $M_n(C_i)$ er antallet af nye metoder i C_i og $DC(C_i)$ er antallet af nedarvede klasser til

C_i. Således angiver formlen et mål for hvor mange overskrevne metoder der findes i systemet i forhold til det samlede antal teoretisk mulige overskrivninger, da nævneren i 4.9 er produktet af antallet af nye metoder i C_i og antallet af nedarvede klasser til C_i. Det skal bemærkes at formel 4.9 ikke er fra den oprindelige artikel, da bedre forståelse af målet blev opnået senere [eAGE95].

En høj polymorfismefaktor er et udtryk for at mange metoder overskrives i underklasser i hele systemet. Jo højere værdi af POF, jo flere metoder bliver overskrevet i systemet. Der må derfor være en sammenhæng mellem POF og MIF, da MIF netop bliver mindre (der er mindre nedarvning i systemet), jo flere metoder der bliver overskrevet. e Abreu et al. kommenterer ikke denne sammenhæng.

e Abreu et al. skriver intet konkret om ønskede værdier for POF, men mener, som for MIF, at ingen af ekstremværdierne er ønskværdige [eAC94, eAGE95]. Det som taler for en værdi højere end nul, hvor ingen metoder overskrives, er at nedarvningen, og raffineringen af klasserne gennem klassehierarkiet, er med til at simplificere implementationen af de enkelte klasser. Hvis værdien af POF findes i det andet ekstrem, bliver fejlfinding i følge e Abreu et. al. besværliggjort [eAGE95].

4.4 Price & Demurjians genanvendelsesevaluering

Price og Demurjian præsenterer en teknik til at analysere komponenter med henblik på at finde ud af, hvor genanvendelige disse komponenter er [PSAD97, PNSAD01, CSAD02]. I modsætning til andre mål der er blevet gennemgået, kræver Price & Demurjian's evaluering metadata. De ekstra informationer der kræves er hvilke klasser der forventes at skulle bruges sammen. Dette er i erkendelse af, at komponenter som regel ikke arbejder alene. En komponent defineres som en mængde af klasser, som forventes at blive genbrugt som en samlet gruppe [PNSAD01, p.42]. Evalueringen af genanvendeligheden af komponenten, er baseret på informationer om koblinger fra den komponent der analyseres, til andre dele af systemet. I evalueringen tages der hensyn til det faktum, at mange klasser er tænkt til at arbejde sammen med andre klasser i systemet og at koblinger til disse klasser derfor ikke nødvendigvis udgør et strukturelt problem for genanvendeligheden. I erkendelse af at der som regel findes klasser i et system som er udviklet udelukkende til det specifikke system, skelner Price og Demurjian mellem generelle og specifikke klasser. Tanken er, at generaliteten af klasserne har indflydelse på hvor alvorlig en kobling mellem klasser er, for genanvendeligheden.

Således kræver evalueringen, at hver klasse markeres som enten *generel* eller *specifik*. De *generelle* klasser er tænkt som klasser der har genbrugspotentiale, og som derfor skal opfylde visse kriterier. De *specifikke* klasser er klasser som laves specielt til det enkelte system, og som derfor ikke be-

tragtes som havende potentiale som genbrug. Markeringen af hvorvidt en klasse er genanvendelig (G) eller specifik (S), blev i [PNSAD01] erstattet med en markering af *genanvendelsesniveauet*, et heltal som angiver i hvor høj grad en klasse menes at være genanvendelig. Den mest generelle klasse bliver markeret med 0, som er det højeste generalitetsniveau. Mere specifikke klasser bliver markeret med højere tal. I det følgende vil der blot blive skelnet mellem to genanvendelsesniveauer, specificeret ved G og S.

I et klassehierarki bliver klasserne mere specifikke, som man arbejder sig ned i hierarkiet. En klasse på øverste niveau tænkes at have et større genbrugspotentiale på grund af generaliteten. Generelt gælder at en subklasse har det samme eller lavere generalitetsniveau end dens forældre.

Når de subjektive karakteristika er anført, beregnes et mål for samtlige klasser. Målet er baseret på koblinger mellem forskellige typer af klasser med hensyn til relationer til andre klasser. I [CSAD02] defineres parvise koblinger i systemet som *metodekald* fra en klasse til en anden. I denne forbindelse skal det bemærkes, at definitionen antager at alle adresseringer af et andets objekts eventuelle `public` variable, tilgås gennem *get/set* metoder. Hvis principperne for *information hiding* brydes, tages der i målet, ikke hensyn til den kobling der måtte opstå som følge af en direkte reference til en anden klasses `public` variable. Dette problem er dog ikke noget modellen ikke kan løse. Det er muligt at opsamle referencer fra en klasse til variable i en anden klasse, og dermed registrere koblingen mellem de to klasser.

Der tages ikke eksplicit hensyn til de koblinger som findes mellem en superklasse og dets subklasser. Det skyldes at de er uinteressante i forbindelse med Price og Demurjians mål, da en subklasse aldrig kan bruges uden dets superklasse. En eksplicit markering af disse koblinger er derfor ikke nødvendige for målet.

Der findes således *generelle* og *specifikke* klasser, og *relaterede* og *urelaterede* klassehierarkier i et system. Dette betyder at der findes otte typer koblinger mellem klasser i systemet, som kan ses i tabel 4.4. I tabellen er G en generel klasse og S en specifik klasse. Genanvendelsesniveauet er ikke markeret i tabellen, da ideen med genanvendelsesniveauet blot er at vise, at nogen klasser er på et højere genanvendelsesniveau end et andet. Dette illustrerer G og S tilstrækkeligt.

1. $G \rightarrow G$ *relaterede hierarkier*	5. $S \rightarrow G$ *relaterede hierarkier*
2. $G \rightarrow G$ *urelaterede hierarkier*	6. $S \rightarrow G$ *urelaterede hierarkier*
3. $G \rightarrow S$ *relaterede hierarkier*	7. $S \rightarrow S$ *relaterede hierarkier*
4. $G \rightarrow S$ *urelaterede hierarkier*	8. $S \rightarrow S$ *urelaterede hierarkier*

Tabel 4.4: Price & Demurjian's typer af kobling mellem klasser

Om klasser er generelle eller specifikke, relaterede eller urelaterede kan ikke afgøres automatisk, så udvikleren er nødt til at angive både om en klasse forventes at skulle genbruges, og hvilke dele af systemet der forventes at

skulle bruges sammen. Dette er formentlig den største hindring for en udbredt brug af metoden. Desværre er det en forudsætning at informationer om relaterede klassehierarkier og de enkelte klassers generalitetsniveau er tilgængelig. Uden generalitetsniveauet vil det ikke være muligt at afgøre om en kobling mellem to klasser er et problem for genbruget, og uden relationerne mellem klassehierarkier, er det ikke muligt at sige om koblinger mellem flere generelle klasser udgør et problem. Desværre er det ikke muligt at automatisere processen. I forbindelse med angivelse af generalitetsniveauet, er det oplagt at forsøge at udnytte den forventede korrelation mellem generalitetsniveau og niveau i klassehierarkiet. Desværre kan dette ikke lade sig gøre, da en klasse sagtens kan være meget specifik selvom den befinder sig højt oppe i et klassehierarki. Ligelede kan det forekomme, at en klasse er generel selvom den findes et stykke nede i klassehierarkiet, typisk i et meget dybt hierarki. Relationerne mellem klassehierarkier kan heller ikke findes automatisk, da hele evalueringen er baseret på koblinger. Det vil ikke være muligt at afgøre om en relation mellem to klassehierarkier er acceptabel uden udviklerens subjektive bedømmelse. Dog vil det være muligt at støtte udvikleren i at vurdere acceptable relationer. Da hele teknikken baseres på fundne koblinger mellem klasser, vil det være muligt at spørge udvikleren om en relation er acceptabel eller ej, i stedet for at udvikleren på forhånd skal give denne information.

Resultatet af analysen af kildekoden er, for hver klasse, antallet af koblinger af de forskellige typer som er angivet i tabel 4.4. Ikke alle koblinger mellem klasser er en hindring for genbrug. Koblinger mellem generelle klasser, som er tænkt at skulle bruges sammen er ikke en hindring for genbrug. Der argumenteres for at denne type af kobling er positiv, da den bidrager til at en større mængde genbruges. Koblinger mellem specifikke klasser er heller ikke nogen hindring for genbrug, da disse klasser ikke er tænkt til at skulle genbruges overhovedet. Koblinger fra specifikke klasser til generelle klasser er ikke nogen hindring, da de specifikke klasser ikke bliver genbrugt. Koblinger fra generelle klasser til specifikke klasser er derimod et problem da en generel klasse herved bliver afhængig af en klasse som ikke er tænkt til genbrug. Yderligere er koblinger på tværs af urelaterede hierarkier et problem for genbrug, da disse hierarkier ikke skal bruges sammen. Tabel 4.5 sammenfatter effekten af de forskellige typer kobling.

Price & Demurjians tilgangsvinkel til genanvendelighedevaluering af klasser er interessant fordi de betragter alvorsgraden af koblinger i et system. Desværre er evalueringen baseret på metainformation, så det er besværligt at benytte det. Et problem som Price & Demurjian ikke kommer ind på er vedligeholdelsen af systemet. Selvom to klasser er tænkt til at arbejde sammen, er generelle og derfor ikke er et problem for genbrug, kan koblingerne være problematiske i forbindelse med rettelse af fejl og den almindelige evolution

1. $G \rightarrow G$ *mellem relaterede hierarkier:* Denne kobling er ikke et problem for genanvendelse. Derimod betragtes den som god, da den giver mulighed for genbrug af en større mængde programkode.

2. $G \rightarrow G'$ *mellem urelaterede hierarkier:* Denne kobling er en hindring for genbrug, da klasserne ikke er tænkt at skulle bruges sammen. For at fjerne koblingen, refaktoreres ved at konstruere specialiserede nedarvede klasser af både G og G'. Disse specifikke klasser genbruges ikke.

3. $G \rightarrow S$ *mellem relaterede hierarkier* Denne kobling er en hindring for genbrug, da en generel klasse afhænger af en specifik klasse. For at fjerne koblingen, refaktoreres ved enten at nedarve fra G og lave en specifik klasse til systemet så man får en $S \rightarrow S$ kobling. Alternativt kan man forsøge at generalisere S, så der fås en $G \rightarrow G$ kobling.

4. $G \rightarrow S$ *mellem urelaterede hierarkier* Denne kobling er en hindring for genbrug, både da klasserne ikke er tænkt til at skulle bruges sammen, og fordi en generel klasse er afhængig af en specifik klasse. Refaktoreres ved at forsøge at flytte den generelle klasse til en specifik ditto. Desværre er der stadig en afhængighed på tværs af hierarkier som også er problematisk for genbrug.

5. $S \rightarrow G$ *mellem relaterede hierarkier* Dette er ingen hindring for genbrug, da den specifikke klasse ikke forventes genbrugt. En muligt refaktorering for at øge muligheden for genbrug, er at forsøge at flytte koblingerne til superklassen for S, så der fås en $G \rightarrow G$ kobling.

6. $S \rightarrow G$ *mellem urelaterede hierarkier* Dette er ingen hindring for genbrug, da koblingen går fra en specifik til en generel klasse. Der er ingen måde at øge muligheden for genbrug.

7. $S \rightarrow S$ *mellem relaterede hierarkier* Igen er dette ingen hindring for genbrug, da det ikke forventes at nogen af klasserne skal genanvendes. Dog kan det muliggøre genbrug hvis begge klasser kunne generaliseres.

8. $S \rightarrow S$ *mellem urelaterede hierarkier* Ingen hindring for genbrug, da klasserne ikke er tænkt at skulle genbruges.

Tabel 4.5: Price & Demurjian's karakteristika ved relationer

i systemet.

4.5 Kohæsion i det objektorienterede paradigme

Kohæsionen i en klasser er et udtryk for sammenhængen i klassen. Hvis kohæsionen i en klasse er lave, kan det være et udtryk for at klassen har urelaterede ansvarsområder og bør opdeles i mindre klasser med hvert sig ansvarsområde. En klasse med høj kohæsion er ønskværdig, da den for det første er mere overskuelig, og dermed lettere at forstå, og fordi den er lettere at genbrug. Når en klasse har høj kohæsion, har den et mere afgrænse og specifikt ansvarsområde, hvilket gør den nemmere at genbruge [Lar02].

Kohæsion virker derfor som et begreb, der kan være med til at identificere designmæssigt problematiske klasser. Kohæsionsbegrebet og forskellige kohæsionsmål for det objektorienterede paradigme bliver gennemgået her.

Det skal bemærkes at det ikke er *kohæsionen* selv der måles. I stedet måles forskellige fakta som menes at være udtryk for god eller dårlig kohæsion i systemet. Ligesom for kohæsion i det strukturerede paradigme, anses høj kohæsion at være udtryk for god modularitet. Denne modularitet har en positiv indflydelse på vedligeholdelsen af systemer. Samtidig er softwaredele med høj kohæsion lettere at genbruge fordi de er begrænsede.

Chidamber & Kemerer's mål $LCOM$ er baseret på antagelsen at en delt variabel mellem flere metoder, er et udtryk for sammenhæng mellem metoderne. Jo mere sammenhængende metoderne i en klasse er, jo mere kohæsiv er klassen.

$LCOM$ er blevet udskældt af forskellige årsager, så andre har forsøgt at opstille mål som bedre skulle afspejle kohæsionen i en klasse. I Abounader et al. påpeges at $LCOM$ ikke er følsom nok i tilfælde af at kohæsionen i en klasse er høj [AL97]. En høj $LCOM$ værdi er tegn på dårlig kohæsion i klassen, men en $LCOM$ værdi på nul, siger ikke nødvendigvis at der er god kohæsion i klassen. Derimod viser $LCOM$ udmærket når der er meget lav kohæsion i en klasse.

$I_i \cap I_j$	I_1	I_2	I_3	I_4
I_1	x	$\neq \emptyset$	$\emptyset$	$\neq \emptyset$
I_2	-	x	$\emptyset$	$\emptyset$
I_3	-	-	x	$\neq \emptyset$
I_4	-	-	-	x

Tabel 4.6: Foreningsmængder mellem mængder af instansvariable

Følgende eksempel på beregningen af $LCOM$ er taget fra Chidamber & Kemerers artikel [CK94]: Betragt en klasse C med tre metoder $M_1, M_2 og M_3$. Lad mængden af instansvariable, I_i som bruges i M_i være følgende: $\{I_1\} = \{a, b, c, d, e\}$ $\{I_2\} = \{a, b, e\}$ og $\{I_3\} = \{x, y, z\}$. $\{I_1\} \cap \{I_2\} \neq \emptyset$, $\{I_1\} \cap \{I_3\} =$

$\emptyset$ og $\{I_2\} \cap \{I_3\} = \emptyset$. LCOM er i dette eksempel 1, da $|P| = 2$ og $|Q| = 1$. Tilføjes en metode M_4 til klassen med $\{I_4\} = \{x, y, z, d\}$ bliver LCOM=0, da $|P| = 3$ og $|Q| = 3$, se tabel 4.6. Det forhold at $LCOM$ i ovenstående eksempel bliver 0, er problematisk, da klassen i virkeligheden indeholder to mængder af kohæsive metoder, den ene indeholdende M_1 og M_2, og den anden indeholdende M_3 og M_4. Kohæsionsmålet burde afspejle det forhold at der i klassen findes to mængder af metoder der hver især arbejder på hver deres disjunkte mængde af instansdata.

Endnu et problem ved $LCOM$ er at der ikke opstilles nogen regler for fortolkningen af værdien. Det kan være svært at vide om en given $LCOM$-værdi udgør et problem, og svært at sammenligne værdier for forskellige klasser.

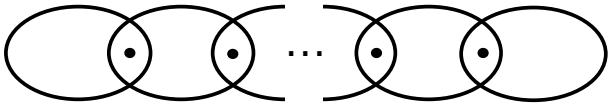
Henderson-Sellers opstiller et kohæsionsmål $LCOM*$, som præsentere kohæsionen i procent [HS96]. Dette kræver at man definerer "perfekt" kohæsion og den "ringeste" form for kohæsion. Den perfekte kohæsion er når målet bliver nul, mens den ringeste kohæsion er når målet bliver én. $LCOM*$ er, som de andre kohæsionsmål baseret på brugen af attributter i en klasse. Den stærkeste kohæsion der kan findes i en klasse bliver når alle attributter bruges i alle metoder i klassen. I det andet ekstrem findes den dårligste form for kohæsion, når en enhver metode i klassen refererer netop én attribut, og denne attribut ikke refereres i andre metoder. Med mængden af metoder $\{M_i\}$ $(i = 1, ..., m)$ og mængden af attributter $\{A_j\}$ $(j = 1, ..., a)$ ser målet således ud [HS96]:

$$LCOM* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1 - m} \tag{4.10}$$

I formel 4.10 er m antallet af metoder i en klasse, som tilgår en mængde attributter, A_j er en attribut og $\mu(A_j)$ er antallet af metoder der tilgår attributten A_j. Tælleren i formel 4.10 er det gennemsnitlige antal metoder der refererer en attribut, minus det totale antal metoder i klassen.

Henderson-Sellers beskriver ikke hvordan tilfældet hvor der er nul eller én metode i en klasse håndteres. Det er dog klart at en klasse med kun én metode må være stærkt kohæsiv, hvilket bør give værdien nul. Et andet, tænkt, eksempel er en klasse som indeholder en mængde attributter, men hvor ingen metoder i klassen tilgår disse. Klassen kan ikke karakteriseres som kohæsiv, men det vil ikke fremgå af målet, da $LCOM*$ vil være nul. Henderson-Sellers opremser et antal design guidelines (se tabel 5.1). Blandt disse findes punktet: "Enhver operation der findes i klassen, skal enten tilgå eller ændre data fra klassen". Hvis dette overholdes, vil problemet med optælling af være løst. En beregning af $LCOM*$ må derfor se bort fra metoder der ikke refererer attributter fra klassen. Således bliver m fra formel 4.10 antallet af metoder

der tilgår mindst én attribut fra klassen. Hvis en klasse har færre end én klasse der tilgår attributter fra klassen, bliver $LCOM* = 0$.



Figur 4.5: Sekvensiel kohæsion

Hitz & Montazeri introducerer begrebet *sekventiel kohæsion* [HM96]. Sekvensiel kohæsion er en fiktiv struktur, hvor en "kæde" af metoder i en klasse deler en instansvariabel. Strukturen er illustreret i figur 4.5, hvor en elipse repræsenterer en metode, og en prik repræsenterer en instansvariabel der bruges i metoden. Hitz & Montazeri viser at $LCOM$ for denne specielle kohæsion kan findes for n metoder ved:

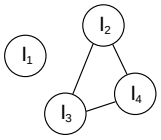
$$LCOM = \left\lceil \binom{n}{2} - 2(n-1) \right\rceil^{+} \tag{4.11}$$

$$hvor \lceil k \rceil^{+} \ er \ k, \ hvis \ k > 0 \ og \ 0 \ ellers.$$

For $n < 5$ er $LCOM = 0$, men for $n = 5, 6, 7$ bliver $LCOM$ 2, 5 og 9.

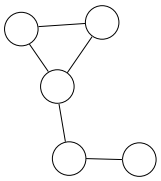
En $LCOM \geq 0$ bør indikere, ifølge Chidamber & Kemerer, at klassen sandsynligvis bør opslittes [CK94], men strukturen i ovenstående eksempel er den samme. Derfor undrer det, at denne indikation af opsplitning kun viser sig for klasser der har mere end fem metoder [HM96]. Dette viser et brud på forståelsen af $LCOM$ værdierne. Hitz & Montazeri mener at den intuitive opfattelse af LCOM kan koges ned til at være antallet af mængder af metoder, som arbejder på en delt mængde af instansvariable. Hvis X er en klasse, I_X mængden af instansvariable der bruges i X, og M_X er metoderne tilhørende X defineres grafen G_X således: $G_X(V, E)$, hvor $V = M_X$ og $E = \{< m, n > \in V \times V | \ \exists \ i \in I_X : (m \ access \ i) \wedge (n \ access \ i)\}$. To knuder i grafen, som repræsenterer to metoder, vil kun være forbundne, hvis der findes en variabel i klassen, som benyttes af begge de metoder som knuderne repræsenterer. Hitz & Montazeri definerer nu $LCOM$ til at være antallet af sammenhængende komponenter i G_X, hvilket svarer til antallet af mængder af metoder, som arbejder på disjunkte mængder af instansvariable. I det følgende kaldes målet $LCOM_{HM}$, for at kunne skelne.

Figur 4.6 viser et eksempel på $LCOM_{HM}$. Eksemplet har tre metodepar, som ikke har nogen brug af en variabel tilfælles ($\{(I_1, I_2), (I_1, I_3), (I_1, I4)\}$) og tre metodepar som *har* fælles brug af variable ($\{(I_2, I_3), (I_2, I_4), (I_3, I_4)\}$). Chidamber & Kemerer's $LCOM$ vil i dette tilfælde blive nul, hvilket er den højeste kohæsion målet kan registrere. I $LCOM_{HM}$ bliver målet, som vist i figur 4.6 to, hvilket ikke er maksimal kohæsion. Bemærk at maksimal kohæsion i $LCOM_{HM}$ er én og ikke nul, som for Chidamber & Kemerer's mål.

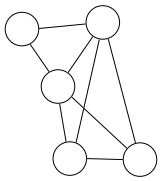


$$I_1 = \{x, y\}$$
$$I_2 = \{a, b\}$$
$$I_3 = \{b\}$$
$$I_4 = \{b, c\}$$

Figur 4.6: Illustration af $LCOM_{HM}$. $LCOM_{HM} = 2$



Figur 4.7: $LCOM_{HM} = 1$, men intuitivt mindre kohæsiv.



Figur 4.8: $LCOM_{HM} = 1$, men intuitivt mere kohæsiv.

Der kan stadig være stor forskel i hvor kohæsiv en klasse er, selvom den har "maksimal" kohæsion med $LCOM_{HM} = 1$. I det ene ekstrem, findes den sekvensielle kohæsion som illustreret i figur 4.5. I det andet ekstrem, bruges alle instansvariable af alle metoder i klassen, hvilket vil give en komplet graf. Sidstnævnte tilfælde har, intuitivt, større kohæsion end første, selvom $LCOM_{HM}$ er den samme i de to eksempler. I figurene 4.7 og 4.8, ses et eksempel på to klasser som begge har $LCOM_{HM} = 1$, men som, kohæsivt, intuitivt virker forskellige. For at kunne differentiere mellem disse tilfælde, bruges følgende mål på klasser med samme kohæsion:

$$C = 2\frac{|E| - (n-1)}{(n-1)(n-2)} \tag{4.12}$$

Målet C i formel 4.12, kommer fra iagtagelsen at det største antal kanter der kan findes i en sammenhængende graf med n knuder, er $\frac{n(n-1)}{2}$, og det mindste antal kanter der kan findes er som i eksemplet i figur 4.5, $(n-1)$. Forholdet mellem det faktiske antal af kanter, og det teoretiske antal, er derfor $\frac{|E|}{(n(n-1))/2}$. Af praktiske hensyn afbildes værdien ned i intervallet fra $[0; 1]$ ved at trække minimum antallet af kanter fra i tæller og nævner. Dermed bliver formlen som i 4.12. En komplet graf vil give værdien 1, mens det andet ekstrem, sekventiel kohæsion, vil give resultatet 0. Målet er altså et udtryk for "hvor komplet" grafen er, og dermed hvor sammenhængende metoderne i klassen er. Det skal bemærkes, at målet i formel 4.12 kun kan bruges på sammenhængende grafer og derfor ikke kan bruges til at differentiere kohæsionen mellem to klasser som har $LCOM_{HM}$-værdi større end én. I modsætning til selve $LCOM_{HM}$, stræbes der efter at C bliver så høj som muligt.

Et andet kohæsionsmål *Coh*, af Chen & Lu (ref. i [AL97]) bruger antal disjunkte mængder af argumenter til metoder, i stedet for instansvariable, som grundlag for beregning af kohæsionen. En klasse med N metoder $M_1 \ldots M_N$ har N mængder af argumenter $I_1 \ldots I_N$. M sættes til antallet af disjunkte mængder af argumenter, formet af foreningsmængder af argumenterne. En klasse har f.eks. fire metoder, med følgende argumentmængder: $I_1 = \{a, b\}$, $I_2 = \{a, c\}$, $I_3 = \{d, e\}$ og $I_4 = \{f\}$ Antallet af disjunkte mængder er tre: $\{a, b, c\}$, $\{d, e\}$ og $\{f\}$. Kohæsionen er defineret som $\frac{M}{N} \times 100$. Igen er en lav værdi udtryk for høj kohæsion. Jo færre disjunkte mængder af variable, jo mere kohæsiv er klassen. Samtidig betyder det, at en klasse bliver mere kohæsiv hvis antallet af disjunkte mængder argumenter holdes fast, mens antallet af metoder i klassen stiger. Med andre ord er en klasse med mange metoder, som alle arbejder på den samme mængde parametre, mere kohæsiv end en klasse med få metoder der også alle arbejder på den samme mængde parametre. Da målet er baseret på argumenter til metoderne, og ikke instansvariable, mener forfatterne at det er muligt at benytte målet tidligere i udviklingen, allerede inden systemet er implementeret.

Instansvariable, som Chidamber & Kemerer bruger i LCOM, er ikke tilgængelige i designfasen, og kan derfor ikke beregnes ligeså tidligt i processen. Til gengæld er der et praktisk problem i at beregne *Coh* automatisk på kildekode. Problemet består i at identificere argumenter som *er* de samme. Selv om viden om både argumentnavn og type er tilgængelig, kan det ikke garanteres, at argumenterne har samme betydning. En fuldstændig sikker automatisk beregning af *Coh* på kildekode er således ikke mulig uden en form for metainformation om parametrene til metoderne. Dette gør målet uegnet i et automatisk evalueringsværktøj.

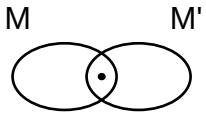
Bieman & Kang [BK95] skriver, at metoder i en klasse kan være knyttet sammen på to forskellige måder, og konstruerer et kohæsionsmål som afspejler dette. Enten deler metoderne en eller flere instansvariable fra klassen, eller også findes der kald fra den ene metode til den anden. Således kan brugen af en instansvariabel findes direkte i en metode, eller brugen af instansvariablen kan findes i en metode som kaldes af den aktuelle metode. Tilknytningen fra en metode til en instansvariabel kan således findes gennem brugen af en anden metode. På grund af *late binding*, kan det ikke altid lade sig gøre, statisk, at finde alle relationer mellem metoder og instansvariable. Dette problem vælger [BK95] at se bort fra, da deres erfaring er at *late binding* i praksis kun har lille indflydelse på kohæsionen i en klasse.

Da *constructors* og *destructors* bruges til initialisering og oprydning, refererer disse specielle metoder ofte en stor del, hvis ikke alle, af en klasses instansvariable. For ikke at dette skal have indflydelse på kohæsionsmålet, vælges det at udelukke både constructor og destructor fra dataopsamlingen til målet. Yderligere adskiller Bieman & Kangs klassekohæsionsmål sig fra

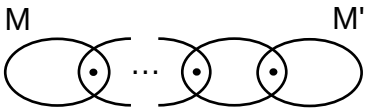
andre ved kun at inkludere metoder som er synlige for brugere af klassen. Private elementer i en klasse bliver kun inkluderet i målet gennem indirekte brug gennem synlige elementer.

Som det er tilfældet for Chidamber & Kemerers definition af $LCOM$ er det vigtigt at forholde sig til hvilke elementer af en klasse som skal tælles med i beregningerne i forbindelse med nedarvning. Bieman & Kang skriver at der er tre muligheder for at medtage metoder og instansvariable når beregningen af kohæsionen finder sted: 1) Alle nedarvede elementer fra alle forfædre medtælles. 2) Kun metoder og instansvariable som er defineret i klassen selv tælles med eller 3) Nedarvede instansvariable tælles med, men nedarvede metoder tælles ikke med. Målet kan bruges uanset hvilken af de tre strategier man vælger.

Målet som Bieman & Kang definerer, er baseret på direkte og indirekte brug af fælles variable. Der opstillet to mål: TCC *Tight Class Cohesion* og LCC *Loose Class Cohesion*. $TCC(C)$ er antallet af direkte forbundne metoder i en klasse i forhold til antallet af mulige forbundne metoder i klassen, og $LCC(C)$ er antallet af inddirekte eller direkte forbunde metoder i forhold til antallet af mulige forbundne metoder. I det følgende er $NDC(C)$ antallet af direkte forbindelser af metoder i C, $NIC(C)$ er antallet af inddirekte forbundne metoder i C og $NP(C)$ er antallet af mulige forbindelser mellem metoder i C. I en klasse med i alt N metoder, vil der være i alt $N \times (N-1)/2$ mulige forbundne metodepar.



Figur 4.9: Direkte forbundne metoder



Figur 4.10: Inddirekte forbundne metoder

To metoder er direkte forbundne, hvis de deler brugen af en instansvariabel, og inddirekte forbundne hvis metoderne inddirekte referer til den samme instansvariabel. Med andre ord er metoden M inddirekte forbundet til M' hvis M kalder en metode der er direkte eller inddirekte forbundet med M'. Figurerne 4.9 og 4.10 illustrerer dette.

$$TCC(C) = NDC(C)/NP(C) \tag{4.13}$$

$LCC(C)$ er antallet af forbundne metoder, de kan være enten direkte eller indirekte forbundne, igen i forhold til det samlede mulige antal forbundne metoder.

$$LCC(C) = (NDC(C) + NIC(C))/NP(C) \tag{4.14}$$

Både TCC og LCC har værdier der ligger mellem 0 og én. Jo mere brug de forskellige metoder gør af hinanden, jo større viser kohæsionen sig.

4.5.1 Diskussion af kohæsionsmål

Når man vil måle egenskaber ved kildekode som skal udtrykke kohæsionen i koden, er det fundamentalt at definere området indenfor hvilket kohæsionen måles. I det strukturerede paradigme taltes om *moduler* som enhed, og i det objektorienterede paradigme er det oplagt at betragte klasser som enhed. Som beskrevet i kapitel 2 beskriver Ezran granulariteten af genbrugelige dele. Derfor kunne man undersøge kohæsion indenfor hvert af disse granuler. Granulerne, som nævnes af Ezran er; funktioner, klasser, mængde klasser (samlet komponent), delsystem (framework) og applikation. Det er ikke interessant at betragte kohæsion på en hel applikation. En applikation har ofte mange ansvarsområder og vil derfor have lav kohæsion. Komponenter og frameworks gennemgås for sig i et senere kapitel. Tilbage har vi klasser og funktioner.

Oprindeligt blev kohæsion undersøgt på modulniveau, hvilket ud fra Stevens, Meyers og Constantines definition, i det objektorienterede paradigme er metoder. Det er ikke muligt at genbruge en metode uden at klassen den indgår i også bliver genbrugt. Nogle klasser kan dog blot betragtes som samlinger af funktioner, der er, eller bør være, relaterede til hinanden. En klasse som f.eks. `java.lang.Math` er et eksempel på dette. Udfra et kompleksitets - og vedligeholdelsesvenligt synspunkt er det relevant at se på kohæsionen indenfor en enkelt metode, ligesom det er gøres i den oprindelige beskrivelse af kohæsion. Det har ikke været muligt at finde kohæsionsmål der er i stand til at klassificere metoder i en klasse, i kategorierne defineret af Stevens, Meyers og Constantine. Fokus for kohæsionen er løftet et abstraktionsniveu op så kun kohæsion i klasser behandles.

Da kohæsion udtrykker sammenhængen i en klasse, og lav kohæsion kan være en indikation af at klassen bør opdeles, er kohæsionsmål interessante til automatisk evaluering af kildekode.

4.6 Kobling i det objektorienterede paradigme

Ligesom det gælder for kohæsion, er der bred enighed om, at kobling har indflydelse på genbrug og vedligeholdelse. Kobling er tilsyneladende lettere tilgængelig end kohæsion. Det er relativt let at finde og definere afhængigheder mellem klasser i et system.

Det er ingen overraskelse at alle koblinger i det strukturerede paradigme, ikke direkte lader sig oversætte til det objektorienterede. Myers definition af content coupling, hvor udtryk modificerer udtryk i andre moduler, er slet ikke mulig i ret mange sprog, heller ikke sprog fra det strukturerede paradigme. Samtidig introducerer det objektorienterede paradigme konstruktioner som ikke er mulige i det strukturerede. Koblingen mellem en forældreklasse og en nedarvet klasse, er et eksempel på dette.

I Chidamber & Kemerer's definition af kobling *Coupling Between Objects*, *CBO* [CK94], findes koblingen for en klasse blot ved at tælle antallet af referencer fra den aktuelle klasse, til andre klasser i systemet. Koblingsmålet tæller kun distinkte klasser, og alle klasser tæller lige meget i målet. I målet, tæller alle former for kobling lige højt. I følge Hitz & Montazeri er dette et problem, da nogle koblinger må betragtes som mere alvorlige end andre [HM96]. F.eks. giver direkte reference til en *instansvariabel* i en fremmed klasse, større kobling end kald til en *metode* i en fremmed klasse, og adgang til instansvariabel i fremmede klasser, giver større kobling end adgang til instansvariable i en superklasse af den aktuelle klasse.

I det følgende kaldes en klasse som indeholder en data eller metoder som andre kalder, for C_s, og klasser som refererer til metoder eller data i C_s kaldes C_c .

For at illustrere Hitz & Montazeri's pointe med at graduere de forskellige typer af kobling, betragtes koblingen fra en klasse C_c direkte til en instansvariabel i i klassen C_s, og en reference til en *getter*-metode der returnerer i. Da problemet er afhængigheden til i i C_s, betragtes måder i kan ændre na-

Ændring	Kobling direkte til i	Kobling til metode `get_i()`
Variablen i kan helt forsvinde fra systemet, hvis værdien i repræsenterer, går fra at være en gemt værdi, til at være en indlæst, beregnet eller lign. værdi.	Referencer til variablen kan ikke længere løses. Dette medfører rettelser alle steder hvor i refereres.	`get_i()`-metoden rettes i C_s så en beregnet værdi returneres i stedet for indholdet af en variabel. Der skal ikke ændres andre steder.
Variablen flyttes, på grund af flytning af ansvar mellem klasser, f.eks. på grund af refactoring.	Som ovenstående.	Som ovenstående.
Indholdet af variablen ændrer mening. F.eks. kunne et i repræsentere en værdi i procent. På grund af nye ønsker til systemet, kommer i til at indeholde promille i stedet, som nye dele af systemet skal bruge.	Alle de steder i anvendes skal rettes. Dette tager tid, og der er en risiko for at systemet vil indeholde fejl fordi ikke alle stederne i bruges, bliver rette.	`get_i()`-metoden rettes til at returnere værdien omregnet til promille.

Tabel 4.7: Mulige ændringer af en instansvariabel, og effekten på de koblede klasser.

tur, gennem evolutionen af systemet. Disse metoder og deres indflydelse på kobling ses i tabel 4.7

Der er andre måder i kan ændre sig, f.eks. kan programmøren af hensyn til et nyt krav til systemet, vælge at ændre typen på i. Dette er også et problem, da systemet derved skal ændres flere steder, men samme problem kunne eksistere, selvom der fandtes en `get_i()` metode i C_s. Problemet er derfor ikke specifikt for kobling til instansvariable.

En koblings styrke er også afhængig af antallet af parametre til den metode som kaldes. Hitz & Montazeri mener, at koblingen til et andet objekt er stærkere, jo flere parametre metoden tager [HM96].

En anden mangel ved Chidamber & Kemerer's definition er, at de ikke forholder sig til om nedarvning af klasser påvirker koblingen i systemet. En klasse kan kalde metoder i en forældreklasse, som om det var dens egen metode, så spørgsmålet er, hvor alvorlig koblingen mellem en klasse `A` og et barn `B` af klassen `A`, som kalder en metode i `A`. Hitz & Montazeri mener at kobling til superklasser for en klasse, er mindre alvorlig end koblinger til fremmede klasser, men at der faktisk findes kobling mellem super- og subklasser.

I forbindelse med genbrug, bliver dette synspunkt støttet af Price & Demurjian [PSAD97], da de netop inddeler grupper i klasser som skal bruges sammen. Koblinger til forældreklasser er ikke alvorlig, i deres mål, da det alligevel ikke er muligt at genbruge en nedarvet klasse C uden at genbruge hele klassehierarkiet ned til C.

Det er tydeligt at der forskel på hvor alvorlig en kobling er. Ikke bare baseret på hvor tæt koblingen er, men også hvilke klasser koblingerne findes mellem. Generelt er en kobling til en klasse som tit ændrer sig, eller hvor det er forventet at den ændrer sig, mere alvorlig end en kobling til en klasse der slet ikke forventes at skulle ændres. Eksempelvis vil en kobling til klassen `String` i Java slet ikke være noget problem, da denne klasse er *meget* stabil (dette på trods, sker det at metoder i Java standardbiblioteket bliver forældede og afskrevne (*depricated*).

e Abreu et. al.'s koblingsfaktor COF citeabreu96, viser forholdet mellem antallet af faktiske koblinger i et system, og antallet af teoretisk mulige koblinger. Derfor siger målet noget om kobling generelt i systemet, og kan ikke bruges til at identificerer problematiske klasser.

4.6.1 Diskussion af koblingsmål

Det er tydeligt at der findes flere varianter af kohæsionsmål end koblingsmål. Dette skyldes formentligt at koblinger er nemmere at identificere. Kobling er mere præcist defineret og nemmere at måle. Igen har man valgt at fokusere på kobling mellem klasser, og ignoreret både kobling mellem metoder

indenfor en klasse, og forskellige koblingtyper mellem klasser. Der er flere af de koblingsformer beskrevet af Stevens, Meyers & Constantine der kan konstrueres i det objektorienterede paradigme.

En af grundene til at der fokuseres på kobling mellem klasser kan være, at disse koblinger udgør et større problem for genbrug og vedligeholdelse end kobling mellem metoder i den samme klasse. Koblingerne er begrænset til elementer indenfor den samme enhed, og dermed er konsekvenserne ved en rettelse i klassen mere overskuelig i forhold til når en rettelse kan påvirke en større mængde klienter. Alene problemet at identificere påvirkede klasser kan være et stort problem. Derfor er koblinger mellem klasser et større problem end koblinger mellem metoder i en klasse.

Hitz & Montazeri er inde på at der findes forskellige former for kobling og at disse former ikke bør tælle lige højt når den samlede kobling for en klasse beskrives. En praktisk løsning på problemet er dog ikke beskrevet i detaljer.

Da stor kobling er en hindring for både genbrug og vedligeholdelse, er koblingsmål interessante i forbindelse med automatisk evaluering af kildekode.

4.7 Sammenfatning

I dette kapitel blev et antal forskellige softwaremål for det objektorienterede paradigme gennemgået, og kohæsion og kobling i det objektorienterede paradigme er blevet uddybet. Følgene punkter opsummerer kapitlet:

- Mange softwaremål, som skulle afspejle forskellige forhold i et objektorienteret system, er blevet konstrueret.
- Nogle mål fra det strukturerede paradigme, er stadig relevante i det objektorienterede paradigme, fordi de indgår som en del af målet.
- Chidamber & Kemerer [CK94], opstillede som de første softwaremål for det objektorienterede paradigme. Af disse mål er specielt kobling CBO interessant i forbindelse med udarbejdelse af softwaremål der kan identificere problematiske områder i kildekode.
- Chidamber & Kemerer's kohæsionsmål $LCOM$ [CK94], er ikke følsomt nok når kohæsionen er høj, dvs. når $LCOM = 0$. Der er derfor brug for andre kohæsionsmål. Chidamber & Kemerers kohæsionsmål er ikke anvendeligt.
- Hitz & Montazeris kohæsionsmål er også baseret på brug af attributter i en klasse. Deres mål betragter antal sammenhængende grafer, baseret på metoders brug af attributter. Samtidig definerer de et mere præcist mål når kohæsionen er høj (den dannede graf er sammenhængende).

- Henderson-Sellers kohæsionsmål giver værdier mellem nul og ét. Derfor kan det bruges til finde "graden af kohæsion" i procent. Dette gør det velegnet til sammenligninger mellem klasser. Samtidig er det relativt let at beregne.

- Kohæsion virker specielt interessant fordi et kohæsionsmål kan afsløre at et klasse har for mange ansvarsområder og derfor måske er problematisk. Opsplitningen af sådanne klasser kan øge muligheden for genbrug og vedligeholdelse. Lav kohæsion i en klasse kan være et udtryk for at klassen har for mange ansvarsområder og derfor bør opsplittes. Når en klasse har høj kohæsion, er den mere overskuelig og har et mere specifikt ansvarsområde. Dette gør den nemmere at genbruge.

- Det er ikke selve kohæsionen der måles, men i stedet egenskaber ved kildekoden som menes at have indflydelse på kohæsionen. Dette er ikke ligetil, hvilket også viser sig i antallet af forsøg på at måle kohæsion.

- Da kohæsion udtrykker sammenhængen i en klasse, og lav kohæsion kan være udtryk for at en klasse skal opdeles, er kohæsionsmål interessante i forbindelse med automatisk evaluering af kildekode.

- Kobling er interessant da mange koblinger betyder at en klasse er svær at tage ud og benytte en ny kontekst. Forskellige styrker af kobling bør findes, da koblingsmål herved måske kan afsløre problematisk design, hvor klasser kender "for meget" til hinanden.

- Koblingsmål er mere "ligetil" at konstruere end kohæsionsmål. Det er de da koblinger er bedre defineret end kohæsion er.

- e Abreu et al.s mål [eAM96] kan ikke bruges til at identificere problematiske klasser i et system, da det måler systemet som helhed. Dernæst har det ikke været muligt for e Abreu et al. at konkludere hvilke værdier for deres mål der er hensigtsmæssige.

Kapitel 5

Designprincipper i objektorienteret programmering

I dette kapitel gennemgås en række designprincipper som er defineret i det objektorienterede paradigme. Gennemgangen, som er baseret på litteratur, foretages for at få overblik over hvad der generelt betragtes som godt design i dette paradigme.

Undervejs diskuteres princippets indflydelse på softwaremålene, for at forsøge at finde frem til mål, som afspejler overholdelse eller brud på designprincipperne.

Konklusionen på kapitlet er, at der findes en sammenhæng mellem nogle af designprincipperne og nogle af softwaremålene. Der findes designprincipper der ikke bliver berørt af nogen af de gennemgåede softwaremål. Liskov Substitution Principle er et eksempel på dette.

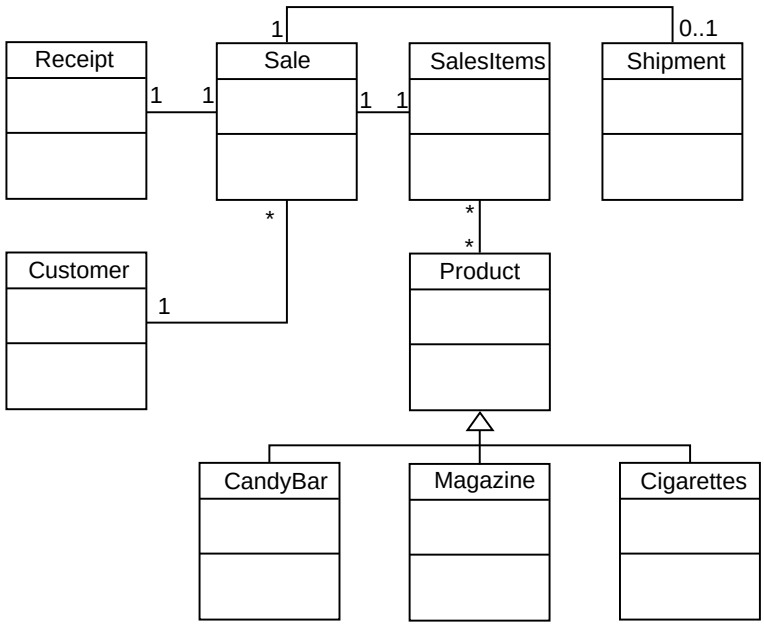
Yderligere er der principper som må kunne identificeres af statiske mål, men som ikke bliver berørt af nogen af de gennemgåede softwaremål. Princippet om at designe mod et interface i stedet for mod en konkret klasse, er et sådan eksempel

5.1 Introduktion

Der findes mange principper for hvordan objektorienteret kode bør designes. I dette kapitel gennemgås nogle generele designprincipper som skulle hjælpe til at designe softwaresystemer der er lettere at vedligeholde og genbruge. Undervejs vil de gennemgåede principper bliver sammenholdt med nogle af de softwaremål der er blevet beskrevet. Der fokuseres på kohæsions- og koblingsmål, da disse virker specielt relevante i forbindelse med genbrug og vedligeholdelse.

Stort set alle designmønstre, som beskrevet af bla. Gamma et al. [GHJV95] er relevante at betragte i lyset af softwaremålene. Det vil i midlertid være for omfattende at gennemgå alle disse mønstre.

Forventningen er, at brugen af designprincipperne giver delsystemer som har lav kompleksitet og gode værdier for kobling og kohæsion. Det skal bemærkes at principperne der gennemgås ikke altid kan betragtes isoleret. Derfor kan det i nogle tilfælde se ud som om et designprincip kan give et dårligere design når det sammenholdes med softwaremålene. Designprincipperne kan ikke slavisk anvendes, men kræver at udvikleren forholder sig til en given problemstilling. I nogle tilfælde kan forskellige designprincipper være i direkte modstrid med hinanden.



Figur 5.1: UML diagram som viser klasser som er involveret i et salg.

Gennem dette kapitel er der brug for eksempler. For mange af disse eksemplers vedkommende, vil de tage udgangspunkt i et fragment af et system til håndtering af salg. UML diagrammet i figur 5.1 viser delsystemet. De enkelte klasser har følgende betydning i systemet:

Sale samler informationer om et salg.

SalesItems er en klasse der indholder de varer der indgår i salget.

Product er en superklasse, fra hvilken alle produkter der kan sælges skal nedarves fra.

Candybar, Magazine og Cigarettes er konkrete implementationer af `Product` og er de eneste varer systemet kan sælge.

Receipt er en klasse der kan generere en kvittering for salget.

Customer indeholder kundeoplysninger om kunden til salget.

Shipment indeholder informationer der skal bruges hvis salget skal sendes til en modtager f.eks. nummer til sporing af forsendelsen, firmaet der står for udbringing etc.

Et salgssystem som eksemplet viser, ville i praksis nok have en lidt anden struktur så hvert produkt ikke var implementeret som en klasse for sig, men eksemplet er udmærket til at illustrere forskellige designprincipper.

I dette kapitel refereres til Javas `interface`-begreb. Nøgleordet `interface` i Java gør det muligt at definere et interface som en klasse senere kan forpligte sig til at implementere. Et `interface` indeholder ikke implementation, men blot en samling *signaturer* for metoder. En signatur er blot en beskrivelse af en operation; et navn, parameterliste og returværdi. For at kunne skelne mellem det generelle interfacebegreb og javas `interface`-begreb, skrives, som alle andre nøgleord, Javas begreb i `courier`.

5.2 GRASP Patterns

Larman beskriver en mængde mønstre som bruges når der skal tildeles *ansvarsområder* mellem forskellige klasser [Lar02]. Disse mønstre (eller principper) kaldes *GRASP* for **G***eneral* **R***esponsibility* **A***ssignment* **S***oftware* **P***atterns*.

Flere af disse mønstre viser principper for godt design i forhold til genbrug og vedligeholdelse. For Larman's *GRASP* mønstre gælder det, at de beskriver ideerne og tankerne i Gamma et al.s [GHJV95] mønstre på et mere abstrakt plan. Flere af Gamma et al.s mønstre dækkes derfor af et enkelt af Larmans. Larmans mønstre vil kort blive beskrevet i det følgende.

5.2.1 Low coupling & High cohesion

"Mønstrene" *low coupling* og *high cohesion* er abstrakte. De siger blot, at når der tildeles ansvarsområder skal der tages hensyn til hvordan tildelingen vil påvirke koblinger og kohæsionen i systemet [Lar02]. En række guidelines, fra Henderson-Sellers, der alle kan følges for at opnå lav kobling og høj kohæsion kan ses i tabel 5.1 [HS96, p.110].

De enkelte punkter i tabel 5.1 vil ikke blive gennemgået. I stedet henvises til punkterne når det er nødvendigt.

5.2.2 Information Expert

Dette mønster siger blot, at når der skal tildeles et ansvarsområde i en mængde af klasser, skal ansvarsområdet tildeles den klasse der råder over den information der skal bruges for at kunne udføre opgaven [Lar02].

1. Alt der indgår i en klasses eksponerede (public) interface, skal være metoder.
2. En klasse må aldrig eksponere detaljer om sin implementation. Heller ikke gennem public metoder.
3. En operation på en klasse må kun være i det eksponerede interface, hvis den skal være tilgængeligt for brugere af klassen.
4. Enhver operation der findes i klassen skal enten tilgå eller ændre data fra klassen.
5. Enhver klasse skal være afhængig af så få andre klasser som muligt.
6. Interaktionen mellem to klasser, må kun bestå af beskedudveksling.
7. Enhver subklasse skal udvikles som en specialisering af sin superklasse således at superklassens eksponerede interface bliver en delmængde af subklassens.
8. Den øverste klasse i et hierarki skal altid være en abstrakt model af et koncept.
9. En mængde genbrugelige klasser skal bruge nedarvning så meget som muligt, til modelering af relationer mellem elementer i domænet.
10. Antallet af metoder i en klasse som har viden om datarepræsentationen i klassen skal begrænses.

Tabel 5.1: Guidelines som støtter høj kohæsion og lav kobling efter [HS96].

Dette princip vil betyde at kommunikationen mellem forskellige klasser vil mindskes, da operationer og data ligger sammen. Dermed bliver koblingen i systemet mindre. Yderligere skal princippet i følge Larman støtte høj kohæsion, af samme årsag: Operationerne på data ligger sammen med data, så de operationer der giver mening på data vil være grupperet sammen.

Følges princippet slavisk, kan det have direkte negativ effekt på designet. Konsekvensen kan være, at alle tænkelige operationer der kan udføres på den mængde data en klasse indeholder, er implementeret i klassen selv. Larmans eget eksempel er en klasse som skal kunne gemme sine data i en database. Ifølge princippet skal denne funktionalitet implementeres i klassen selv, men dette vil betyde at klassen bliver mindre kohæsiv da den kommer til at indeholde databasefunktionalitet. Yderligere kan det føre til duplikeret kode, da flere andre klasser formentlig også skal kunne gemme data i databasen. En "databasekommunikationsklasse" vil derfor være at foretrække selvom denne klasse ikke indeholder de data der skal gemmes i databasen.

Udslag i målene

Hvis kobling bliver mindre og kohæsion større ved at følge princippet, bør dette give udslag i de relaterede softwaremål.

For koblings vedkommmende er det klart, at hvis data og operationer på samme ligger i forskellige klasser, vil det øge dem samlede kobling i systemet, da der uundgåeligt vil være referencer mellem operationsklassen og

dataklassen. På denne baggrund vil koblingen blive mindre hvis der måles med f.eks. koblingsmålet, Coupling Between Objects CBO. Dette er for princippet *isoleret* set.

Som beskrevet kan princippet, hvis det bruges uden omtanke, føre til duplikeret kode. Dette kan have en negativ indflydelse på koblingen, da denne kopierede kode kan have referencer til andre klasser. Hvis den samme, duplikerede kode, ligger i flere forskellige klasser, kan koblingerne for systemet *totalt* set blive højere.

e Abreu et al.s koblingsfaktor, Coupling Factor COF vil, ligesom CBO blive mindre. COF er som bekendt defineret for det totale system, og kan derfor ikke bruges til at sige noget om den enkelte klasse.

Samlet vil det formentlig vise sig, at koblingen i det samlede system, og i de enkelte klasser vil blive reduceret når designprincippet overholdes. Hvis operationer der før var implementeret i en ekstern klasse bliver flyttet ind i den klasse der besidder data, vil koblinger til "operationsklassen" blive fjernet.

Princippet siger at operationer der arbejder på data fra en klasse, skal implementeres i denne klasse. Det må betyde at de operationer der implementeres i klassen, for at følge princippet, faktisk benytter nogle af de attributter der findes i klassen. Dette er dog slet ikke ensbetydende med at de data der ligger i klassen er kohæsive. Hvis de ikke er det, bliver klassen ikke mere kohæsiv af, at der bliver tilføjet metoder der arbejder på hvert sin mængde af attributterne.

Derfor vil brud eller overholdelse af princippet ikke nødvendigvis vise sig i målet for den klasse der analyseres.

Til gengæld er det muligt at den klasse der, mens princippet bliver brudt, indeholder operationerne vil vise sig mere kohæsiv efter princippet bliver overholdt. Dette skyldes at metoder der ikke er relateret til de attributter der findes i klassen bliver fjernet. Derfor kan kohæsionsmålene blive bedre for disse klasser.

5.2.3 Creator

Objekter skal konstrueres et sted. *Creator*-mønsteret tildeler ansvaret for konstruktionen af et objekt [Lar02]. Konstruktion af et objekt introducerer koblinger mellem objektet som er ansvarlig for konstruktionen; `A` og objektet der bliver konstrueret; `B`. Derfor er det væsentligt hvordan ansvarsfordelingen af konstruktioner af objekter er. Mønsteret siger at `A` skal konstruere `B`, hvis et eller flere af følgende punkter er sandt:

- `A` aggregerer `B`-objekter (*aggregates*).
- `A` indeholder `B`-objekter (*contains*).

- `A` registrerer `B`-objekter (*records*).
- `A` er tæt forbunden med `B`-objekter gennem brug (*closely uses*).
- `A` indeholder data som `B`-objekter skal initialiseres med.

Hvis flere af ovenstående punkter er opfyldt for forskellige klasser, prioriteres aggregering og indhold højere end de øvrige. Hvis mønsteret overholdes, skulle det resultere i lavere kobling, god indkapsling og genanvendelighed.

Udslag i målene

For ovenstående punkter: Aggregering, indhold, registrering, brug og initielle data, gælder det at koblingen mellem den konstruerende klasse og den klasse der bliver konstrueret ikke bliver større på grund af konstruktionen, fordi der i forvejen vil være en kobling mellem konstruktør og konstruerede objekt. Hvis reglerne overholdes vil det ikke give en øget kobling mellem klasserne.

Som eksempel på princippet om konstruktion, betragtes konstruktionen af et `Product` (se figur 5.1). I følge princippet bør det være `SalesItems` der står for konstruktionen af et `Product`, da `SalesItems` indeholder produkter. For at bryde princippet kan vi vælge at lade `Sale` have ansvar for at konstruere `Product`-objekter og derefter sende dem til `SalesItems`. Dette vil betyde at `Sale` skal kende til `Product`-klassen, hvilket muligvis slet ikke er nødvendigt hvis `SalesItems` står for kontruktionen af produkter. Derfor tyder det på at et brud på princippet kan føre til øget kobling i systemet, hvilket burde kunne ses i både koblingsmålet CBO og koblingsfaktoren COF.

Det er væsentligt at bemærke at konstruktøren af et objekt i denne forbindelse er det objekt der sætter konstruktionen i gang, og får en instans af et nyt objekt. Selve konstruktionen kan foregå gennem et *factory*-mønster, hvor det er factory-klassen der sørger for konstruktionen. I denne forbindelse vil det være objektet der kalder factory-metoden der betragtes som konstruktør, og ikke selve factory-klassen.

Intuitivt kan man argumentere for at kohæsionen i systemet vil blive svagere når princippet ikke følges. Hvis et objekt `A`, er ansvarlig for at konstruere andre objekter som `A` ellers ikke har nogen interaktion med, vil konstruktionen af objektet gøre kohæsionen i `A` svagere fordi `A` ikke ellers er relateret til det konstruerede objekt.

De kohæsionsmål der er blevet gennemgået er baseret på brugen af et objekts attributter, og det er ikke sikkert at det konstruerede objekt er attribut i konstruktøren, men blot findes som lokal variabel. Derfor er det ikke sikkert at brud på reglen vil kunne ses i kohæsionsmålene. Ser man på eksemplet med konstruktion af `Product`-objekter, er alternativerne som sagt at enten `Sale` eller `SalesItems` konstruerer objekterne. I begge tilfælde skal tilføjelsen af produkter til salget muligvis gå gennem `Sale`. Derfor er spørgsmålet

om der skal findes en metode i `Sale` som konstruerer er produkt og sender det til `SalesItems`, eller om `Sale` blot beder `SalesItems` om at tilføje et produkt. I begge tilfælde refereres der til attributten af typen `SalesItems` i `Sale`. Derfor er der ikke forskel i hverken antallet af metoder eller antallet af attributter i `Sale` som kan have indflydelse på kohæsionen.

5.2.4 Polymorphism

Det er ofte et ønske at kunne udvide eller erstatte elementer af et system med nye dele, uden at dette kræver ændringer i klienten til de dele der erstattes eller udvides. En mekanisme der kan bruges til at nå dette mål er polymorfisme. Dette bruges når der er brug for at objekter skal opføre sig forskelligt afhængigt af deres type, eller når det skal kunne lade sig gøre at indsætte *pluggable components* [Lar02].

De konkrete produkter repræsenteret ved klasserne `Candybar`, `Magazine` og `Cigarettes` i figur 5.1 er alle nedarvede fra den fælles forældreklasse `Product`. I det sandsynlige tilfælde hvor det ønskes at der kan sælges andre end disse tre produkter, er det vigtigt at denne tilføjelse til systemet kan ske så let som muligt. Når et nyt produkt skal tilføjes, skal dette blot understøtte det samme interface som de øvrige produkter for at virke. Bruges polymorfi til at opnå dette, skal der ikke ændres nogen steder i systemet for at tilføje et nyt produkt, andet end den nye klasse som repræsenterer produktet. I eksemplet i figur 5.1 implementerer hvert konkret produkt deres egen version af metoden `getPriceInclVat()` da priserne på produkterne er forskellige, og det er meningen at produkterne skal kunne sælges med forskellig momssats. Var systemet i stedet implementeret uden brug af polymorfi, ville det kræve

```
⋮

public double getPriceInclVat() {
   switch(type) {
      case Magazine : return calcPrice(magazinePrice, magazineVat);
      case Cigarettes : return calcPrice(cigarettesPrice, cigarettesVat);
      case Candybar : return calcPrice(candybarPrice, candybarVat);
   }
   return 0;
}

⋮
```

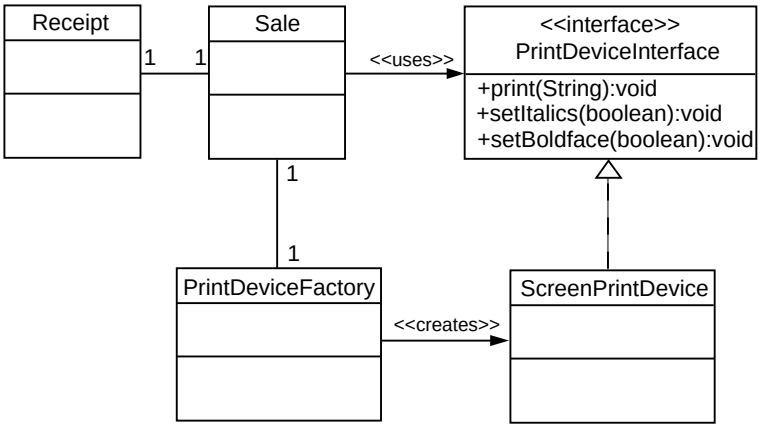
Figur 5.2: Metode fra klassen `ProductNoPoly` til returnering af prisen på produktet.

at enhver funktion som var afhængig af typen af produktet, skulle foretage sammenligninger for at identificere typen af produkt. I figur 5.2 vises et fragment af koden i `ProductNonPoly`-klassen, som ikke benytter polymorfi. Her

ses det at metoden som returnerer prisen på en vare, gør dette på baggrund af værdien af attributten `type` i `ProductNonPoly`.

En lignende fremgangsmåde vil være nødvendig hver gang typen af produkt har betydning for funktionaliteten, derfor vil konstruktioner som i eksemplet i figur 5.2 kunne findes flere steder i koden. I nogle sammenhænge vil det ikke være klassen `ProductNonPoly` der indeholder den funktionalitet der er specifik for typen af produkt, så `switch`-konstruktioner som i eksemplet vil muligvis også kunne findes spredt gennem resteren af systemet. Et eksempel kunne være hvis `Receipt` skulle håndtere bestemte varer på specielle måder. Tilføjelse af et nyt produkt uden brug af polymorfi vil derfor kræve ændringer mange steder i koden. Selvom tilføjelsen kunne afgrænses i klassen selv, vil logikken som vælger mellem typerne af produkt gøre klassen uoverskuelig og give klassen lavere kohæsion.

I forbindelse med *pluggable components* bruges polymorfisme til at definere et samlet interface for en mængde klasser. I Java kan både nedarvning og interfaceimplementation, vha. `interface` bruges til at definere dette interface. Tages der igen udgangspunkt i delsystemet i figur 5.1 kan man forestille sig at klassen `Sale` har brug for at kunne skrive en kvittering fra `Reciept` ud. Udskrivningen kan foretages på forskellige enheder som alle foretager udskrivningen på forskellig måde. Enhederne implementerer tekstattributter som fed og kursiv på forskellige måder. For at kunne udskifte udskrivningsenheden ud uden at skulle ændre i implementationen af `Sale`, kan man bruge nedarvning, eller som i følgende eksempel et `interface` som beskriver hvad klassen som minimum udbyder af funktionalitet, og som `Sale`, andre klienter, derfor kan benytte sig af.



Figur 5.3: UML diagram som viser `PrintDeviceInterface` som *pluggable* enhed.

I figur 5.3, som viser strukturen, er `SalePluggable` ikke bundet til en konkret udskrivningsklasse, men udelukkende til et interface som definerer hvad `SalePluggable` kan forvente af udskrivningsenheden. Derfor kan det lade sig gøre at variere udskrivningsenheden uden at det er nødvendigt at

ændre i `SalePluggable`. Det skal bemærkes at `SalePluggable` stadig er koblet til interfacet `PrintDeviceInterface`, som det ses i figur 5.4

```
⋮

private PrintDeviceInterface printDevice;

public SalePluggable() {
   salesItems = new SalesItems();
   printDevice = PrintDeviceFactory.createPrintDevice();
}

public void printReceipt() {
   printDevice.setItalics(true);
   printDevice.print(new ReceiptPluggable(this).getText());

}

⋮
```

Figur 5.4: Klassen `SalePluggable` er afkoblet fra udskrivningsenheden og kun koblet til interfacet `PrintDeviceInterface`.

Fordelen ved at bruge polymorfi er at det er let at lave variationer i programmet, og lettere at lave nye implementationer som kan tages i brug uden at klienterne i den eksisterede kode bliver påvirket.

Ved brug af polymorfisme er det vigigt at superklasser ikke har viden om eventuelt nedarvede klasser. Grunden til dette er at hvis viden om subklasser bliver brugt i superklassen, risikere man at skulle ændre superklassen når det bliver tilføjet nye nedarvede klasser til klassehierarkiet. Pr. definition har nedarvede klasser kendskab til sin superklasse, og denne afhængighed er ikke et problem for genbrug. Dog findes der forskellige former for afhængighed fra en subklasse til superklassen, hvor nogle af den kan udgøre et problem for vedligeholdelsen af systemet. Dette bliver gennemgået senere.

Udslag i målene

Hvis et system implementerer variationer som det blev illustreret i eksemplet med produkterne der ikke benytter polymorfi (se koden i figur 5.2), og dette senere refaktoreres således at polymorfi benyttes, vil det formentlig påvirke kompleksitetsmålene.

Det forventes at koblingen i systemet vil være mindre eller uændret. I den udstrækning der i det oprindelige system uden polymorfisme fandtes logik der skelnede mellem de forskellige klasser, vil koblingen i systemet blive lavere, da koblingerne til de konkrete klasser vil forsvinde og blive erstattet af en enkelt kobling til klassernes fælles superklasse.

Koblingen kan også vise sig uændret i det tilfælde hvor systemet kun forberedes på alternative implementationer af en klasse, og der derfor kun er implementeret en enkelt konkret klasse. Koblingerne vil være fra det eksisterende system til den abstrakte klasse i stedet for til den konkrete klasse. Koblingsmålene vil derfor vise en uændret kobling.

Det skal bemærkes at brud på principperne for brug af polymorfi kan konstrueres på forskellig måde. F.eks. kan en attribut der angiver typen for en klasse bruges i logikken i en superklasse til de klasser der skelnes imellem. Dette vil formentlig ikke øge koblingen i systemet, da attributten der testes på findes i superklassen. En anden konstruktion kunne være at lade en anden (hjælpe) klasse teste attributten i de nedarvede klasser. I dette tilfælde vil der opstå kobling til hjælpeklassen.

Det kan yderligere forventes at kohæsionen i systemet vil blive højere eller uændret. Hvert af de nedarvede klasser indeholder nu kun data og metoder som giver mening for den enkelte klasse. I det tidligere eksempel indeholdt den fælles klasse `Product` momsprocenter for flere forskellige varer.

Det er klart at der er andre mål der bliver påvirket af omskrivningen fra brud på princippet til overholdelse af samme. Antal børn, NOC, og selvfølgelig dybden af klassehierarkiet, DIT, vil begge blive større. Der er dog intet i designprincippet der kræver at DIT bliver stor, mens NOC vil være meget afhængig af konteksten i hvilken princippet bruges. I forbindelse med forskellige `PrintDevices` er det tvivlsomt at NOC vil blive specielt stor, mens strategien for implementation af forskellige varer formentligt vil betyder at NOC bliver meget stor.

5.2.5 Indirection

Indirection er et princip der bruges for at undgå at klasser bliver tæt koblede til hinanden [Lar02]. I det tidligere eksempel kommunikerede klassen `SalePluggable` med interfacet `PrintDeviceInterface` i stedet for direkte med en konkret implementation af interfacet. Samtidig er `SalePluggable` afkoblet fra konstruktionen af en konkret implementation af et "PrintDevice". Disse er begge eksempler på *indirection*. Der er flere af mønstrene fra Gamma [GHJV95] der bygger på princippet om "indirektehed", blandt andre *Façade* og *Adapter* [Lar02].

Udslag i målene

Mønsteret er meget generelt og det er derfor svært specifikt at sige hvordan målene vil blive påvirket af mønsteret. Det afhænger af konteksten som mønsteret anvendes i. Mønsteret undgår direkte kobling mellem elementer

således at muligheden for genbrug forøges. Derfor må man forvente at koblingsmålene viser dette. Tages et eksempel med brugen af et façademønster er det oplagt at koblingen i systemet vil blive mindre. Façademønsteret bruges til at simplificere et interface til en mængde relaterede klasser, som arbejder sammen om et fælles problem. Kommunikation med de forskellige relaterede klasser går gennem et dedikeret façadeobjekt, som videresender beskeder til andre objekter. Klienter til dette samlede komponent vil derfor kun skulle kommunikere med façadeobjektet. Jo flere klienter til det samlede komponent jo større udslag vil det have i koblingsmålene. De klienter der er, eller ville være, koblet til flere klasser, vil med façademønsteret kun være koblet til ét. Dette eksempel viser at koblingen under nogen omstændigheder kan blive mindre, men det er svært at sige noget generelt fordi mønsteret er meget abstrakt

Ligesom for kobling er det svært at sige noget specifikt om hvordan kohæsionen vil være påvirket af at mønsteret overholdes. Hvis façademønsteret betragtes igen, er kohæsionen afhængig af hvor tæt relateret klasserne som façadeklassen bruger er. Hvis façadeklassen udbyder metoder fra klasser som kun knyttes sammen i meget få metoder, vil kohæsionen i façadeklassen være lav. Igen er målet meget afhængigt af konteksten det indgår i og det er derfor ikke muligt at sige noget generelt om kohæsionen når mønsteret overholdes.

5.2.6 Pure fabrication

Dette princip beskriver det problem, at man, når man tildeler ansvarsområder til klasser, kan komme til at implementere mange ansvarsområder til samme klasse, som følge af brugen af *information expert* [Lar02]. *Information expert* siger at et ansvarsområde skal tildeles den klasse der har informationerne til at kunne udføre opgaven. For at undgå at klasserne får for mange ansvarsområder med risiko for at blive svagt kohæsive, bruges *pure fabrication*. Princippet siger at nogle ansvarsområder bør implementeres i en klasse for sig, som er meget kohæsiv og meget genanvendelig. Følges *information expert*, kunne man argumentere for at metoden `abs()` som returnerer den absolutte værdi af et tal, skulle implementeres i samtlige klasser som repræsenterer et tal det giver mening at tage `abs()` af, f.eks. Javaklassen `Integer`. Dette ville føre til store svagt kohæsive klasser, og genbruget af kode i de mange ens metoder vil blive besværliggjort. I stedet findes klassen `java.lang.Math` som indeholder matematiske funktioner. Klassen `Math` er blevet til ved *pure fabrication*, da det er en "tænkt" klasse. Deraf navnet *pure fabrication*. Det er ikke en klasse man ville kunne finde i domænemodellen til et system.

Et andet eksempel på en klasse konstrueret efter princippet, kunne være en klasse til kommunikation med en database. Klasser kunne indeholde den logik der skal til at skabe kontakt til databasen, læse og skrive i den etc.

Denne er konstrueret som alternativ til at lade hver klasse implementere sin egen databasekommunikation.

Princippet lover i følge Larman højere kohæsion fordi princippet gruppere relaterede metoder i samme klasser, og øget mulighed for genbrug, da små funktionsorienterede klasser kan bruges i flere forskellige applikationer.

Udslag i målene

Mønsteret faktoriserer ens funktionalitet ud af de enkelte klasser og ind i en fælles klasse. Dette kan både øge og mindske koblingen i systemet. Det kan mindske koblingen i systemet hvis de metoder der bliver udfaktoreret indeholder mange koblinger til andre klasser. Derved bliver koblingerne fjernet fra alle klienter og erstattet med en kobling til en enkelt klasse. Dette ligner meget hvad der kan ske ved brug af *indirection*-mønsteret.

Koblingen kan i nogle sammenhænge blive større hvis de metoder der udfaktoreres ikke er koblet til andre klasser. I dette tilfælde vil der opstå kobling mellem klienter og den nyoprettede klasse.

Det er meningen af mønsteret kun skal bruges på operationer der er relateret til hinanden, men som tidligere nævnt, kan operationer været relateret på forskellig måde. I databaseeksemplet fra før vil de forskellige operationer i klassen sikkert dele instansvariable der indeholder forbindelsen til databasen. Derfor vil de gennemgåede kohæsionsmål formentlig vise at kohæsionen i systemet bliver større. Omvendt er det ikke sikkert at de forskellige operationer deler data selvom de er tæt relaterede. Problemet er, som tidligere nævnt, at den semantiske kohæsion ikke kan måles ved hjælp af de gennemgåede mål. De matematiske operationer der tideligere blev nævnt, er et eksempel på dette.

Den semantiske kohæsion må dog ventes at blive højere for systemet.

5.2.7 Protected variations

Dette mønster beskriver muligheder for at designe delsystemer, således at dele af systemet kan varieres uden at det påvirker andre dele af systemet på en uønsket måde [Lar02]. Kernemekanismerne som giver denne mulighed for variationer, er de objektorienterede mekanismer; indkapsling og polymorfisme. Interfaces og indirektion er tillige vigtige.

Ganske kort er løsningen på variationer, at man identificerer områder i systemet som er sandsynlig genstand for ændring, og derefter konstruere et *stabilt* interface rundt om disse. Implementationen af variationspunkter i koden kan være på grund af ønsket om at have mulighed for at variere elementer i systemet, eller de kan være implementeret på grund af forventet evolution i systemet. Klassiske eksempler på elementer der kan variere er operativsystemspecifikke og hardwarespecifikke dele.

Variationspunkterne kan etableres på mange forskellige måder. *Indirection* og *polymorphism* som er gennemgået tidligere, er eksempler på principper der understøtter muligheden for at variere bestemte elementer i koden. Specielt kan *Open-Closed principle*, *Liskov Substitution Principle* og *Law of Demeter* nævnes som principper for design af systemer som er robuste overfor ændringer og videreudvikling. Disse bliver gennemgået senere i kapitlet, og en diskussion af deres indvirkning på kompleksiteten i systemet vil findes der.

Relation til målene

Dette mønster er abstrakt og implementeres i praksis ved hjælp af nogle af de principper der er gennemgået andre steder i specialet. Mønsteret har til hensigt at afkoble konkrete klasser fra hinanden og lade kommunkationen mellem klasser være afhængig af interfaces til klasserne. Der er ingen af de gennemgåede mål der skelner mellem kobling mellem konkrete klasser og mellem klasser og interfaces.

5.3 Open-Closed principle

Meyer beskriver designprincippet *Open-Closed Principle* [Mey97]. Et modul skal være åbent for udvidelse og tilpasning af modulet, men lukket i forhold til ændringer i selve modulet. Essensen i princippet er, at modulet er stabilt i sit interface.

Grunden til formuleringen af princippet er, at en komponent, eller et modul, kan indgå i mange sammenhænge. Hvis der på et tidspunkt er brug for at modificere komponenten for at imødekomme nye krav, må dette ikke påvirke de klienter der allerede bruger komponenten.

I princippet skal komponenter derfor designes så de *aldrig* ændres. I stedet skal de designes så komponenten giver mulighed for tilpasning og udvidelser gennem tilføjelse af kode. Nøglen til løsningen på dette problem er abstraktioner. Konkrete klasser bør ikke have referencer til hinanden [Mar00]. I stedet skal der refereres til abstrakte klasser eller interfaces. Disse abstrakte klasser eller interfaces kan erstattes med ny kode der opfylder de nye krav. Samtidig bliver det unødvendigt at ændre på de eksisterende klienter.

Principperne *Open-Closed Principle* og *Protected Variants* i store træk formuleringer af det samme princip. Meyer lægger vægt på evolutionen i komponenten, mens Larman taler om variationen. Begge principper gør brug af abstraktioner, polymorfisme og information hiding for at opnå den ønskede effekt.

Princippet er udmærket demonstreret i afsnit 5.2.4, hvor et `PrintDevice-Interface` gør det muligt at erstatte en del af systemet som udskriver en kvittering. Det største problem er at finde ud af præcis hvor det er nødvendigt at implementere mulighed for variation og evolution. Dette skal afgøres i analysen af systemet, og ligger derfor uden for specialets område. Således

er den gennemgåede løsning heller ikke forberedt, hvis det på et tidspunkt blev et krav, at der, udover den traditionellle kvittering, også skulle printes en faktura på en anden type printer. Dette kan ikke umiddelbart lade sig gøre, da der kun kan være ét `PrintDevice` tilknyttet et `Sale`. `Sale` er lukket overfor ændringer, men ikke tilstrækkeligt åben overfor udvidelser. Havde det været tydeligt fra start at det kunne være nødvendigt med ovenstående udvidelse, kunne det have været løst med en liste af `PrintDevices` som alle fik kaldt deres `print()`-metode. Dette havde løst *dette* problem. Senere kunne det blive nødvendigt at udvide systemet med en kasseskuffe der automatisk går op når kvitteringen printes. Umiddelbart kan dette ikke lade sig gøre. Igen er `Sale` lukket fordi der ikke var tænkt på denne evolution[1] En mere generel løsning på problemet er selvfølgelig at implementere et *observer pattern* [GHJV95]. Pointen er blot, at man ikke kan sikre sig mod enhvert tænkeligt ønske om ændring til systemet.

Udslag i målene

Princippet er meget generelt og siger ikke noget om fordeling af ansvarsområder som kan have indflydelse på f.eks. kohæsion og kobling. Princippet skal give mulighed for at udvide eller erstatte dele af en implementation uden at eksiterende klienter skal modificeres.

Der er umiddelbart ingen af de gennemgåede softwaremål der bliver påvirket af brugen af princippet. Der bliver senere gennemgået et mål der menes at give udslag i forbindelse med brugen af princippet.

5.4 Design mod et interface

Dette princip indgår som del af flere af de allerede gennemgåede principper. Det er dog vigtigt, og bliver derfor eksplicit nævnt her.

Grunden til at princippet er, at man ikke ønsker at der ikke opstår koblinger mellem konkrete klasser. De koblinger der bør findes i koden skal være mellem en klient og et interface. Effekten er, at en klient ikke kan være afhængig af dele af leverandørklassens interne struktur, og at en leverandør altid kan udskiftes af en anden som tilbyder det samme interface.

Liskov mener, at for at abstraktioner, som interfaces jo er, skal kunne virke, *skal* implementationerne være indkapslet og principperne for information hiding skal overholdes [Lis88, p.19]. Dette betyder, i ekstremet, at der skal defineres interfaces for alle klasser i et system. I Java kunne dette realiseres ved at definere et `interface` for alle klasser der implementeres. Interfacet skal ikke indeholde alle de metoder klassen indeholder, blot de metoder der

[1] Nogle kunne måske blive fristet til at implementere kasseskuffen som et *PrintDevice*, og bruge metoden `print()` til at åbne kasseskuffen. Dette ville virke, men i vores søgen efter det gode design, er dette en meget dårlig strategi. Løsningen ville i øvrigt bryde med *Liskov Substitution Principle (se afsnit 5.7)*

skal være tilgængelige fra klienter. Konsekvensen er også at ingen attributter nedarves til subklasser. Har en subklasse brug for en attribut i superklassen, skal denne attribut hentes gennem interfacet.

Gamma et al. anbefaler ligeledes at man så vidt muligt undgår at erklærer instanser af en konkret type, men at man kun binder sig til et interface som beskrives af en abstrakt klasse [GHJV95, p.18]. Dette er også er en anbefaling fra D'Sourza [DW99]. Steder i koden er det naturligvis uundgåeligt at instantiere objekter, og dermed instantiere en konkret klasse. Dette skal begrænses til klasser der er konstrueret til dette formål. For en uddybning af dette refereres til Larmans *Creator* mønster [Lar02] og *Creational Patterns* i Gamma et al. [GHJV95].

Både D'Souza et al. og Meyer skriver, at man skal forsøge at minimere interfacet, dvs. mindske antallet af metoder i interfacet. Årsagen er, at koblingen mellem klasser stiger jo mere information der udveksles mellem disse klasser. Jo større interfacet er, jo mere information udveksles potentielt mellem klasserne. En relateret regel er at mindske antallet af interfaces til en klasse [Mey97]. Igen er grunden til reglen at man ønsker at minimere koblinger mellem klasser. Jo færre interfaces en klasse har, jo "kommunikationskanaler" er der til at koble klasser sammen. Derfor vil denne regel føre til færre koblinger [Mey97].

Relateret til dette princip er en regel fra Meyer, *explicit interfaces* der siger, at alle moduler der kommunikerer med hinanden, skal gøre dette på en tydelig måde [Mey97]. Reglen illustreres bedst gennem et modeksempel:

En klasse `A` modificerer en variabel `v`, i en klasse `B`, gennem en metode i `B`. I en metode i en klassen `C` læses `v` fra `B`, og `C` agerer på baggrund af værdien i denne variabel. Dette eksempel giver kobling mellem klasserne `A` og `C` uden at denne kobling er gennemskuelig for en udvikler.

Udslag i målene

Der er ingen af de gennemgåede mål der tager hensyn til koblinger mellem elementer i et system er mellem konkrete klasser, eller mellem konkrete klasser og interfaces. Derfor er der ingen af de gennemgåede mål der vil blive påvirket af om en reference til en konkret klasse bliver erstattet af en reference til et interface. Koblinger til interfaces er en bedre form for kobling, da et interface kan implemeteres af nye klasser som kan erstatte den gamle, uden at dette har indflydelse på klienten til interfacet. Det bør kunne lade sig gøre at opstille et statisk mål der afspejler om dette designprincip benyttes.

Hvis princippet om at *alt* skal indkapsles, må der ikke findes referencer til superklassers attributter fra subklasser. Dette betyder at målet *Attribute Inheritance Factor*, AIF, skal være nul. Dette svarer imidlertid ikke til de anbefalinger e Abreu et al. kommer frem til. Det er uvist hvad grunden til

uenigheden er, men som for alle andre principper, må der foretages en afvejning af fordele og ulemper ved at bruge princippet. Hvis man insisterer på en AIF på nul, vil det betyder at der skal implementeres flere metoder fordi nedarvede klasser stadig kan have brug for at tilgå attributter i superklassen. Dette giver et større interface, hvilket er i strid med D'Souza et al. og Meyers anbefalinger om at forsøge at minimere interfacet af hensyn til kompleksiteten.

Anbefalingerne om at minimere interfacene til klasserne bliver ikke direkte mål af nogen af de gennemgåede mål. Vægtede metoder, WMC, måler samtlige metoder i en klasse, og tager således ikke hensyn til om metoden er en del af (public) interface til klassen. Målet Method Hiding Factor, MHF, bliver påvirket hvis minimeringen af klassens (public) interface betyder at metoder bliver flyttet fra at være offentligt tilgængelige, til at være private. Dette vil betyde at MHF bliver større, altså et udtryk for at flere af metoderne i systemet er gemt for klienter.

5.5 Nedarvning

Selvom nedarvning og komposition, som gennemgås i næste afsnit, ikke er decideret designprincipper, bliver de beskrevet her fordi begge begreber bruges for at opnå kodegenbrug og brugen af begreberne påvirker nogle af de gennemgåede softwaremål.

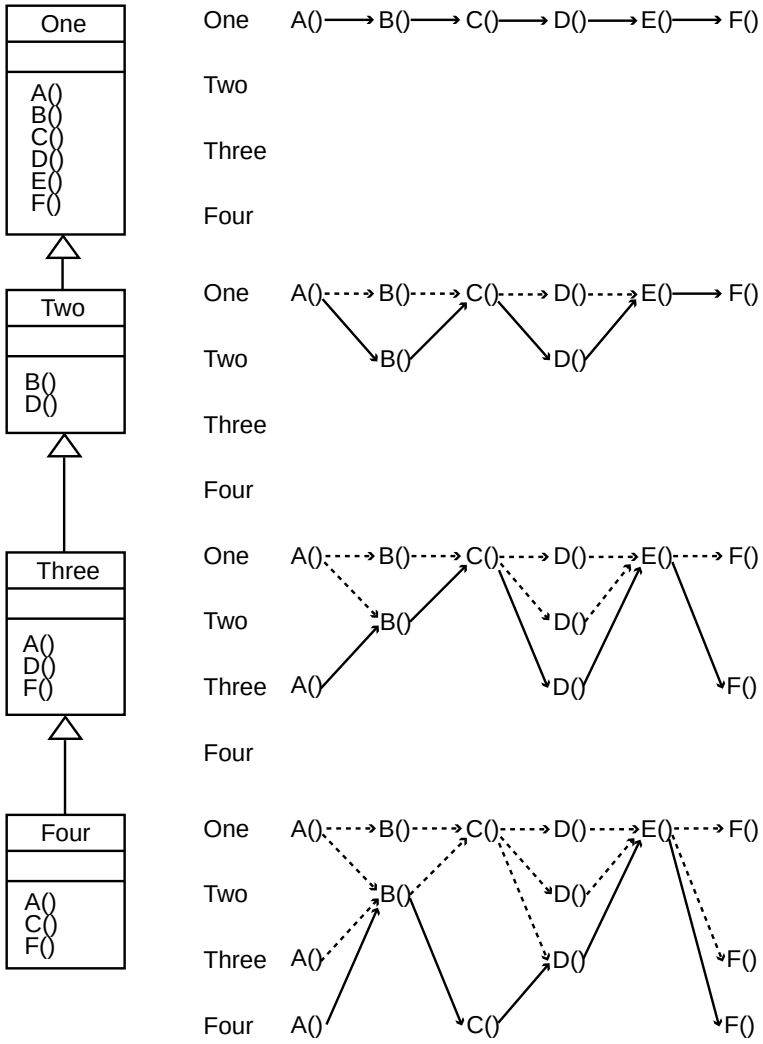
Nedarvning opnås ved, at en klasse beskrives på baggrund af en eksisterende klasse. På denne måde specialiseres klasser ned gennem klassehierarkiet, og der opnås kodegenbrug fordi klasser nede i hierarkiet arver metoder og attributter implementeret højere oppe i hierarkiet. Fordelen ved at specialisere klasser gennem et klassehierarki er netop at der opnås genbrug.

Som bliver beskrevet i det følgende, er der en del problemer forbundet med brugen af nedarvning.

Et problem er at nedarvning kan ødelægge indkapslingen. Detaljer fra en superklasse er ofte synlige i nedarvede klasser, hvilket kan betyde at en eventuel ændring i superklassen kan kræve ændringer i de nedarvede klasser. Dette er årsagen til Liskovs anbefaling om at alt skal indkapsles.

Et andet problem ved nedarvning, er når der ønskes at genbruge en nedarvet klasse. Hvis superklassen til den genbrugte klasse indeholder dele der ikke er ønskværkdige i den nye kontekst, kan det være nødvendigt at omskrive forældreklassen. Et eksempel kunne være, at dele af superklassens interface ikke er fornuftigt i den nye kontekst. Det er ikke muligt at skjule dele af et nedarvet interface, så ændringerne skal foretages i forældreklassen. Dette er ikke godt for genbrug eller vedligeholdelse. I høj grad kan dette problem være forårsaget af et uhensigsmæssigt konstrueret nedarvningshierarki, da nedarvede klasser skal være specialisering af deres superklasse, og derfor bør have mindst det samme interface som superklassen.

Et problem forbundet med nedarvning er når klassehierarkierne bliver for dybe. I et dybt hierarki hvor metoder bliver overskrevet i dybereliggende klasse, bliver det svært at overskue og forstå hvilke operationer der bliver kaldt på hvilke klasse [JGJ97b]. Dette beskrives som *yo-yo* problemet. Problemet er, at udvikleren er nødt til nøje at studere klassehierarkiet op og ned, for at forstå kontrol-flowet i progammet, fordi det, på grund af polymorfien, ikke umiddelbart er klart hvilke metoder der bliver kaldt. Dette er også et argument for at holde klassehierarkier lavere, og bruge komposition i stedet for nedarvning.



Figur 5.5: Viser yo-yo-problemet. Det kan være problematisk at forstå hvilke metoder der bliver kaldt, når metoder overskrives gennem klassehierarkiet.

I figur 5.5 vises et klasse hierarki med fire klasser. For metoderne gælder det, at `A()` kalder `B()`, som kalder `C()` osv. Overskrivninger af nogle af me-

toderne i subklasserne til `One` betyder at det er forskellige implementationer af metoderne der bliver kaldt. I figuren illustreres hvordan kaldesekvensen er for instancer af de forskellige klasser. Er objektet, som `A()` kaldes på en instans af klassen `One`, ser kaldene ud som i øverste linie. Alle metoderne der kaldes er implementeret i `One`. Anderledes ser det ud for objekter af typer længere nede i hierarkiet. Det bliver mere komplekst at finde frem til de metoder der faktisk bliver kaldt. Jo dybere klassehierarki, jo, potentielt, flere overskrevne metoder.

Et andet problem ved nedarvning i forbindelse med genbrug/vedligeholdelse er skrøbelige basisklasser (*eng: fragile base classes*) [MS98]. I klassehierarkier kan ændringer i basisklassen have indflydelse på de klasser der nedarver fra denne basisklasse. Det er klart at ændringer i en basisklasse kan have indflydelse på virkemåden i subklasser til basisklassen. Nogle af de følgeeffekter en ændring i en basesklasse kan have på sine subklasser kan være svære at forudse. Selv i de tilfælde hvor en subklasse kun er afhængig af forældreklassens interface kan der opstå problemer. I [MS98] beskrives forskellige måder disse problemer kan opstå. Betragt kildekoden[2] i figur 5.6 og 5.7. Klassen `FragileBag` definerer en klasse der kan gemme værdier af typen `String`. Den indeholder to metoder til at tilføje elementer til sin interne struktur; `add(element)` og `addAll(elements[])`. Metoden `addAll(..)` kalder, som det ses i figuren `add(..)` for samtlige elementer i det array der modtages som parameter. En bruger ønsker at konstruere en ny type, `CountingBag`, baseret på `FragileBag`. Denne inkrementerer en intern variabel hver gang der tilføjes et element via `add(..)`. Da `addAll(..)` i superklassen `FragileBag` kalder `add(..)` for alle elementer i det array der gives med som parameter, virker `addAll(..)` uden problemer for instancer af `CountingBag`. På et tidspunkt ændres `FragileBag`, af den ene eller anden årsag, så elementerne lægges direkte i den interne datastruktur uden at `add(..)` kaldes, som det ses i figur 5.8. Konsekvensen er at `addAll(..)` ikke længere virker efter hensigten i den nedarvede klasse `CountingBag`. Årsagen er, at `CountingBag` antog at `add(..)` altid ville blive kaldt når elementer blev tilføjet `FragileBag`. Koblingen mellem superklasser og deres nedarvede klasser kan være stærk. Man kan ikke antage at alle nedarvede klasser virker efter ændringer foretaget i superklassen. I princippet betyder dette at alle nedarvede klasser skal inspiceres og testes efter en rettelse i en klasse, fra hvilken andre klasser nedarver.

Selvom der er knyttet en del problemer til nedarvning, skal brugen ikke afskrives som teknik til genbrug. I følge Gamma et al. fører komposition dog ofte til design der er simplere og lettere at genanvende [GHJV95, p. 20].

[2] Dette er et tilpasset eksempel fra [MS98]

```
⋮

public class FragileBag {

   public void add(String str) {
      bag.add(str);
   }
   public int size() {
      return bag.size();
   }
   public void addAll(String[] strArray) {
      // This works because of the call to 'add', and the fact
      // that 'add' is overridden in subclass 'CountingBag'
      for(int i=0; i<strArray.length; i++) {
         add(strArray[i]);
      }
   }

⋮
```

Figur 5.6: Superklassen *FragileBag*'s metode `addAll(..)` kalder *add(..)* for alle elementer.

```
⋮

public class CountingBag extends FragileBag {
   public void add(String str) {
      count++;
      super.add(str);
   }
   public int size() {
      return count;
   }

⋮
```

Figur 5.7: Subklassen *CountingBag* nedarver fra *FragileBag*, og overskriver `add(..)`-metoden. Efter ændring i superklassen *FragileBag*, virker denne klasse ikke længere efter hensigten.

```
⋮

public void addAll(String[] strArray) {
   // Subclasses no longer works, because they expect
   // 'add' to be called
   for(int i=0; i<strArray.length; i++) {
      internalColl.add(strArray[i]);
   }
}

⋮
```

Figur 5.8: Superklassen *FragileBag* efter ændring i metoden `addAll(..)`

Udslag i målene

Alle de mål der er relateret til klassehierarkier vil blive påvirket når nedarvning benyttes. Store værdier for DIT og NOC viser at nedarvning bliver brugt intensivt i systemet der evalueres.

Andre mål som Method Inheritance Factor, MIF, og Attribute Inheritance Factor, AIF, kan også blive påvirket, men er afhængig af deklarationen af metoder og attributter. Derfor kan der ikke siges noget generelt om disse mål.

En undersøgelse af Bieman & Kang viste, at klasser som blev brugt intensivt i nedarvning havde relativt lav kohæsion [BK95]. Dette tyder på at den form for genbrug der gøres brug af i forbindelse med nedarvning, ikke er besværet af lav kohæsion.

5.6 Law of Demeter

Law of Demeter er en designregel, der begrænser mængden af klasser en metode kan sende beskeder til [LHR88, LH89]. Begrundelsen for reglen er at jo mere en klasse kender til det omkringliggende system, jo mere skrøbeligt er dette objekt i forhold til ændringer i systemet. Der opstår med andre ord uønskede koblinger mellem dele af systemet.

Essensen af Law of Demeter er: Metoderne i en klasse ikke bør være afhængige af strukturen i *andre* klasser. Yderligere bør metoderne i en klasse kun sende beskeder til en *begrænset* mængde af andre klasser [Sak88]. Følgende definitioner fra Lieberherr bruges til at beskrive Law of Demeter [LH89]:

Klient. Metoden M er klient til klassen C, hvis M kalder en metode f tilgængelig i C. Hvis f er specialiseret i en nedarvet klasse, er M kun klient til den "højeste" klasse i hierarkiet som specialiserer f.

Leverandør. Hvis M er klient til C, er C leverandør til M.

Foretrukken bekendt. En foretrukken bekendt (*eng: preferred acquaintance*) til metoden M, er et objekt som enten er konstrueret direkte i M eller et globalt objekt der kan bruges i M.

Foretrukken leverandør. Klassen B er foretrukken leverandør (*eng: preferred supplier*) til metoden M defineret i C, hvis B er leverandør til M og et af følgende holder:

- B findes i deklarationen af C
- B findes i deklarationen af et argument til metoden M
- B er foretrukken bekendt til C

For at Law of Demeter skal være overholdt, kræves det at enhver leverandør er en *foretrukken leverandør*. Mere uformelt kan loven udtrykkes således: Ethvert kald fra en metode M i et objekt C, skal være til et objekt af en af følgende klasser:

- Klassen selv (*this* i Java).
- Argumentobjekter til metoden M.
- Underdele af C, dvs. klasser C består af.
- Et objekt som konstrueres i M.

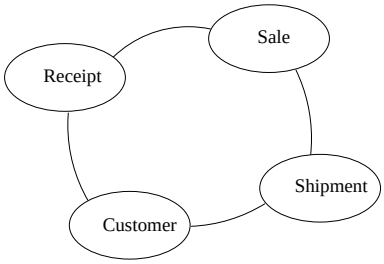
Et brud på Law of Demeter ses i figur 5.9, hvor metoden `getText()` i klassen `Receipt` kalder `getName()` og `getAddress()` i linierne 24 og 25. Fowler kalder fænomenet *Message Chains* [Fow99]. `Receipt` er en klasse som, givet et `Sale`-objekt, skal konstruere en kvittering som kan hentes af klientobjektet gennem `getText()`. Kundens navn og adresse skal fremgå af kvitteringen. Disse oplysninger fås gennem kundeobjektet `Customer` som returneres af `Sale`-objektet. `Receipt` kommunikerer derfor både med `Sale` og med `Customer` objekterne. Dette er et brud på Law of Demeter, da `Sale` kommunikerer med `Customer`-objektet, og ikke er foretrukken leverandør til `Receipt`.

Samme situation findes i klassen `Shipment`, som er en eventuel forsendelse af varerne i salget. I `Shipment`-klassen findes en metode `getAddress()` som returnerer adressen som forsendelsen skal leveres på. `getAddress` finder også adressen ved at kalde gennem objektet `Sale`, med konstruktionen: `sale.getCustomer().getAddress()`.

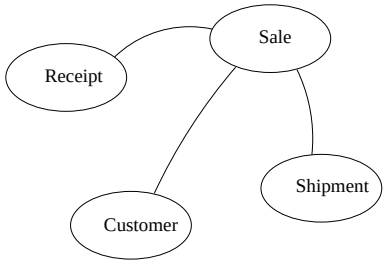
Der findes derfor koblinger mellem `Receipt` og `Customer`, og mellem `Shipment` og `Customer`. Disse afhængigheder illustreres i figur 5.10. For at bringe ovenstående kode i overensstemmelse med Law of Demeter, konstrueres metoderne `getCustomerName()` og `getCustomerAddress()` på klassen `Sale`. Klasserne `Receipt` og `Shipment` kender således ikke længere til `Customer` objektet. Afhængighederne efter rettelsen illustreres i figur 5.11.

```
  ⋮

20:   public String getText() {
21:      float total=0;
22:      String receipt;
23:
24:      receipt = sale.getCustomer().getName() + "\n";
25:      receipt += sale.getCustomer().getAddress() + "\n\n";
26:
27:      SalesItems salesItems = sale.getSalesItems();
28:      Product[] p = salesItems.getProducts();
29:
30:      for(int i=0; i<salesItems.getProductCount();i++) {
31:         receipt += p[i].getName() + "\t" + p[i].getPriceInclVat() +"\n";
32:         total += p[i].getPriceInclVat();
33:      }
34:      receipt += "\ntotal:\t\t" + total + "\n";
35:      return receipt;
36:
37:   }

  ⋮
```

Figur 5.9: I klassen `Receipt` bryder metoden `getText()` Law of Demeter



Figur 5.10: Afhængigheder mellem objekter ved brud på Law of Demeter



Figur 5.11: Afhængigheder mellem objekter hvor Law of Demeter overholdes

Fordelen ved at overholde Law of Demeter er at der bliver færre koblinger mellem klasser. Hvis ændringer i interfacet til `Customer` ændrer sig, er det en mere begrænset mængde af klasser der bliver påvirket. Hvis f.eks. adressefeltet i `Customer` ændrer repræsentation, så de ikke længere blot er en tekst, kan ændringer begrænses til `Sale`-objektets `getCustomerAddress()`. Koden som ikke overholder Law of Demeter, skal rettes flere steder, både i `Receipt`-klassen og `Shipment`-klassen.

```
  ⋮

20:   public String getText() {
21:      float total=0;
22:      String receipt;
23:
24:      receipt = getCustomerName(sale.getCustomer()) + "\n";
25:      receipt += sale.getCustomer().getAddress() + "\n\n";
26:
27:      SalesItems salesItems = sale.getSalesItems();
28:      Product[] p = salesItems.getProducts();
29:
30:      for(int i=0; i<salesItems.getProductCount();i++) {
31:         receipt += p[i].getName() + "\t" + p[i].getPriceInclVat() +"\n";
32:         total += p[i].getPriceInclVat();
33:      }
34:      receipt += "\ntotal:\t\t" + total + "\n";
35:      return receipt;
36:   }
37:
38:   public String getCustomerName(Customer c) {
39:      return c.getName();
40:   }

  ⋮
```

Figur 5.12: Omskrevet version som ikke bryder Law of Demeter, men stadig indeholder kobling.

Det ses udfra definitionen af foretrukne leverandører, at koden kan omskrives så den overholder Law of Demeter uden at koblingerne i systemet reduceres. I figur 5.12 ses en omskrivning af koden i figur 5.9. Linie 24 i den oprindelige kode, er omskrevet så der kaldes en lokal metode, `getCustomerName()`, som henter kundens navn. Da der kun refereres til foretrukne leverandører i klassen, overholder den nu Law of Demeter, men der er stadig koblinger fra `Receipt` til `Customer`. Selvom loven er overholdt, strider omskrivningen mod idéen i Law of Demeter.

Law of Demeter er yderligere opdelt i en *stærk* og en *svag* form. Den stærke form stiller strengere krav til de objekter et givent objekt må kommunikerer med. Forskellen på de to former er, at den stærke form kræver af

der kun kommunikeres med instansvariable som er defineret i *klassen selv*. Med andre ord må der ikke refereres direkte til instansvariable som er defineret i en superklasse til klassen selv. Denne restriktion findes ikke i den svage form [LHR88]. Grunden til skelnen mellem de to former illustreres i følgende eksempel.

Som det fremgår af figur 5.1 nedarves tre forskellige produkter fra klassen `Product`; `CandyBar`, `Magazine` og `Cigarettes`. Disse er i eksemplet *konkrete* produkter. `Product` indeholder attributten `price`, som repræsenterer prisen, eksklusive moms, på varen. Da alle produkter har en pris, er denne attribut implementeret i `Product`. I eksemplet skal der være mulighed for at have forskellig moms på de forskellige produkter, så hvert konkret produkt implementerer en metode `getPrice()`, som returnerer prisen inklusive moms. I figur 5.13 overholdes kun den svage form af loven, da der refereres direkte til den nedarvede attribut `price`, mens koden i figur 5.14 overholder den stærke version af loven, da prisen på produktet returneres gennem metoden `getPrice()` på `Product`.

```
 ⋮

private double vat = 0.25;
public double getPriceInclVat() {
  return price * (1 + vat);
 }

 ⋮
```

Figur 5.13: Den svage form for Law of Demeter er overholdt.

```
 ⋮

private double vat = 0.15;
public double getPriceInclVat() {
 return getPrice() * (1 + vat);
}

 ⋮
```

Figur 5.14: Den stærke form for Law of Demeter er overholdt.

Fordelen ved den stærke form af loven viser sig i de tilfælde hvor strukturen i den nedarvede attribut ændrer sig. Forestiller man sig at det skal være muligt at angive prisen på produkterne i forskellige valutaer, er datatypen `double` ikke længere tilstrækkelig. Af denne årsag kunne man indføre en klasse `PriceCurrency` som indeholder både prisen på varen, og den valuta prisen er opgivet i. Overholdes kun den svage version af loven, skal der ændres i alle de subklasser som bruger værdien `price`. I den version der overholder den stærke version, skal der kun ændres i superklassen, altså i `Product` i eksemplet. Metoden `getPrice()` skal blot ændres så klienterne til metoden stadig modtager prisen som `double` i en referencevaluta.

Som det fremgår af ovenstående, stiger antallet af koblinger mellem klasser i systemet, hvis Law of Demeter ikke overholdes. Dette forhold menes at være betydende for mulighederne for både vedligeholdelse og genbrug.

I forbindelse med vedligeholdelse ses det, at overholdelsen af både den strenge og den svage form af loven, har indflydelse på hvor let det er at foretage ændringer i systemet. Nogle af de problemer der kan opstå i forbindelse med evolutionen af systemet begrænses eller fjernes, når loven overholdes.

I de tilfælde hvor man ønsker at genbruge klasser eller delsystemer, har overholdelsen af Law of Demeter også betydning, på grund af antallet af koblinger i systemet. Forstiller man sig, at klassen `Receipt` fra ovenstående eksempel var ønskværdig at genbruge, kan dette kun ske sammen med genbrug af klassen `Sale`, da `Receipt` henter sine data fra `Sale`. I eksemplet som ikke overholder Law of Demeter, *skal* der yderligere implementeres en `Customer`-klasse, da `Receipt` også henter data herfra. Det bliver altså mere omstændigt at genbruge `Receipt` hvis Law of Demeter ikke er overholdt.

Udslag i målene

Da det nu er sandsynliggjort at overholdelse af Law of Demeter har indflydelse på hvor let det er at genbruge og vedligeholde software, er det interessant at overveje hvordan de gennemgåede kompleksitetsmål vil opføre sig, på det, semantisk set, samme system, ved og uden overholdelse af Law of Demeter. I det følgende vil der blive argumenteret for hvorledes nogle af disse mål må opføre sig.

Som det ses af ovenstående eksempler, vil en overholdelse af Law of Demeter betyde flere implementerede metoder. Antallet af argumenter til de enkelte metoder kan også blive større [LHR88]. Som eksempel kunne man forestille sig at man har brug for en metode som i `char`-præsentation, returnerer værdien af den n'te mest betydende bit, af den binære repræsentation af tallet i, kunne denne findes på på følgende måde:

```
myInteger.toBinaryString(i).charAt(n);
```

Den lokale viden som findes det sted hvor kaldet foretages skal sendes med som argumenter til den metode som erstatter ovenstående besked-kæde, så resultatet kunne komme til at se således ud:

```
myInteger.nthPositionOfBinary(i,n);
```

Law of Demeter kan derfor resultere i længere parameterlister til de enkelte metoder.

Prisen for at få den lavere kobling som Law of Demeter tilbyder, er tilsyneladende flere metoder, og flere parametre til metoder. Disse mange metoder kan komme til at forringe overskueligheden i koden.

Hvis et program, som ikke overholder Law of Demeter, transformeres så det kommer til at overholde loven[3], er det interessant at se hvordan de forskellige

[3]Lieberherr et al. viser at ethvert objektorienteret program kan omskrives så det overholder Law of Demeter uden at semantikken i programmet ændres [LH89].

kompleksitetsmål bliver påvirket. Law of Demeter ændrer ikke ved strukturen i klassehierarkiet, så målene DIT og NOC er ikke relevante.

Law of Demeter giver flere metoder og flere argumenter til nogle af metoderne. Dette forhold vil påvirke de mål som bruger disse elementer som basis for beregningen af kompleksiteten. Weighted Methods per Class, WMC, er et af de mål der afhænger af antallet af metoder. Som beskrevet i kapitel 4 er der flere måder at opgøre WMC. Chidamber & Kemerer's oprindelige metode var at tillægge alle metoder vægten én. Dermed vil WMC blive større efter applikationen af Law of Demeter. Det er foreslået at bruge et traditionelt mål for kompleksiteten i en metode, og bruge dette som vægt i WMC.

Bruges andre vægte vil WMC også gå op, totalt set. Betragtes før og efter applikationen af Law of Demeter ses det, at antallet af metoder på leverandørsiden vil gå op. Hos klienten er den eneste ændring at eksisterende beskedkæder (*message chains* [Fow99]), bliver fjernet. McCabe's cyklomatiske mål er baseret på kontrolstrukturen i metoden, og bliver derfor ikke påvirket af at Law of Demeter overholdes. Halstead's program volume vil blive lavere hvis man *isoleret* ser på den kaldende funktion. Til gengæld vil den tilføjede metode gøre program volumen større for systemet totalt set. Den *potentielle volumen* ændrer sig naturligvis ikke, da den, i følge Halstead, mest kortfattede måde at udtrykke problemer ikke ændrer sig, da antallet af operander ikke ændrer sig. Program level kommer nærmere til én, da koden som overholder Law of Demeter er tættere på det mest kortfattede udtryk af problemet. Alt i alt vil WMC derfor vise et system som en anelse mere komplekst efter brugen af Law of Demeter, i forhold til før.

Et andet mål som baseres på antal metoder i en klasse er Response For Class, RFC. Værdien af denne forventes derfor også at ændre sig. I klientklasserne, vil RFC dog ikke blive påvirket, fordi RFC i Chidamber & Kemerer's definition kun ser på første niveau af kald. Til gengæld vil leverandørklassens RFC-værdi stige, da der kommer en metode mere til klassen. Samlet for et system, vil RFC således blive større.

Hvis vi antager at det typiske tilfælde vil være at den nye metode vil være erklæret `public`, må Method Hiding Factor, MHF, blive lavere for systemet. Method Inheritance Factor, MIF, vil forblive den samme eller blive større, afhængigt af niveauet i klassehierarkiet hvor den nye metode i systemet bliver implementeret.

De fleste kohæsionsmål er baseret på brugen af instansvariable i klassen. I det gennemgåede eksempel, vil ændringerne ikke ændre på brugen af disse instansvarianble, og dermed vil kohæsionen ikke ændre sig. Den eneste ændring der kan have indflydelse på kohæsionen er i leverandørklassen `Sale`. Her bliver en metode tilføjet, som refererer til instansvariablen `customer` som tilhører sale. Hvis denne metode refererer til *andre* instansvariable, kan kohæsionen blive større, men i det mest enkle tilfælde vil kohæsionen være stabil. Kohæsionsmålet af Bieman & Kang [BK95] bruger, som beskrevet,

metodekald som grundlag for kohæsionsberegningen. Efter brugen af Law of Demeter, er det sandsynligt at klassen selv ikke bruger den nye implementerede metode. Det tætte kohæsionsmål (se formel 4.13) vil derfor vise at kohæsionen i klassen er blevet lavere, da antallet af mulige metodekald er blevet større (fra $(N \times (N-1))/2$ til $((N+1) \times N)/2$), mens antallet af faktisk forbundne metodepar vil være uændret. Det samme viser sig i Bieman & Kang's løse kohæsionsmål, som yderligere indeholde inddirekte metodekald.

Det forventes til gengæld at koblingerne i systemet er blevet lavere. CBO er blot antallet af andre klasser en given klasse refererer til. Derfor bliver koblingen i ovenstående eksempel lavere. Da argumentet for at bruge Law of Demeter netop er at det reducerer koblinger i systemet, er det ikke overraskende. I følge Hitz & Montazeri [HM96], vil en lang parameterliste betyde en stærkere kobling til en klasse, og lange parameterlister er blevet karakteriseret som "bad smells" af Fowler [Fow99]. Da Law of Demeter netop kan betyde at parameterlister bliver længere, kan der opstå stærkere koblinger på denne bekostning. Desværre har det ikke været muligt at finde konkrete kompleksitetsmål som tager hensyn til lange parameterlister. Det er dog langt fra sikkert at den kobling der opstår som følge af lange parameterlister er væsentlig i forhold til at et antal klasser er blevet afkoblet fra hinanden.

Hvis vi skal tro de gennemgåede kompleksitetsmål, vil en ændring i et system fra ikke at overholde Law of Demeter, til at overholde loven, gøre systemet mere komplekst, bortset fra de mål som forsøger at beskrive koblingen i systemet. Det er dog opfattelsen at Law of Demeter bidrager mere positivt til genbrug og vedligeholdelse end den gør negativt. Dette hænger sammen med, at de forskellige kompleksitetsmål formentlig kun vil afspejle en lille forøgelse af kompleksiteten sammenlignet med fordelene ved afkoblingen. I de gennemgåede kompleksitetsmål er der ikke gjort forsøg på at graduere vigtigheden eller alvorsgraden af de forskellige mål. Når Law of Demeter alligevel må anses som værende et godt princip, er det fordi lav kobling mellem klasser er vigtig for genbrug og vedligeholdelse.

5.7 Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP) er formuleret på følgende måde:

> "What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T." [Lis88]

Dette betyder bl.a. at et program der benytter objekter af en klasse `T`, skal virke som forventet hvis det i stedet bruger klasser `S` nedarvet fra `T`. Med

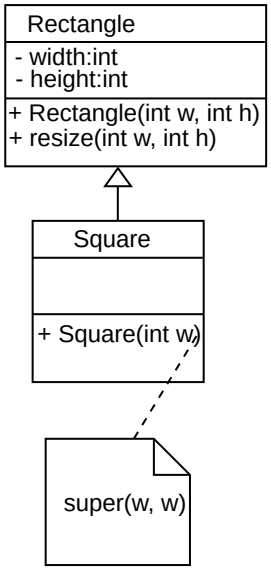
andre ord, skal alt man kan gøre med en forældreklasse kunne gøres med en nedarvet klasse, så en nedarvet klasse må ikke have begrænset interface i forhold til forældreklassen.

Implikationen er, at enhver nedarvet klasse skal indeholde mindst det samme interface som den klasse den er nedarvet fra. Dette sikres i det objektorienterede paradigme. Dette er imidlertid ikke nok. Hver metode i en nedarvet klasse skal også udføre den samme opgave som den klasse den er nedarvet fra, så semantikken af de nedarvede metoder skal være den samme som for forældreklassen.

Grunden til princippet er at klienter til en klasse ikke har, og ikke skal have, bekymringer om hvilken eksakt instans af en klasse der aktuelt bruges. Objektet skal blot gøre hvad der forventes af det.

```
  ⋮

public class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
      width = w;
      height = h;
   }
   public void resize(int w, int h) {
      width = w;
      height = h;
   }
   public int getArea() {
      return width * height;
   }
   public String toString() {
      return "(" + width + ", " + height + ")";
   }
}

  ⋮

   public class Square extends Rectangle {
      public Square(int w) {
         super(w, w);
      }
   }

  ⋮
```

Figur 5.15: Eksempel på brud på Liskov Substitution Principle



Figur 5.16: UML for strukturen i figur 5.15

Eksemplet i figur 5.15 illustrerer problemet. På grund af visse lighedstegn mellem rektangler og kvadrater, er kvadrat implementeret som en subklasse til kvadrat. Da `Rectangle` har en metode `resize` til at ændre størrelsen på rektanglet, har `Square` automatisk denne metode. Problemet er at metoden `resize(width, height)` kan kaldes på kvadratklassen, hvilket bryder med et kvadrats definition; at alle sider er lige lange. Det er derfor et brud på LSP at nedarve kvadrat fra rektangel. Da et kvadrat og et rektangel har fællestræk, er en løsning at nedarve fra en fælles forældre, f.eks. `Polygon` eller lign, som kunne indeholde fælles metoder, f.eks. beregning af arealet.

Brydes LSP, får det en konsekvens for klienterne til de klasser der bryder princippet. I figur 5.17 vises et fragment af en klient til `Rectangle` fra figur 5.16.

```
⋮
    if (o instanceof Square)
       o.resize(10,10);
    else if (o instanceof Rectangle)
       o.resize(12,10);
⋮
```

Figur 5.17: Et fragment af en klient til `Rectangle`.

Som det ses er klienten nødt til at sikre invariansen i `Square` ved at kalde `resize()` med argumenter med ens værdier. En konstruktion som den der ses i figur 5.17 er ødelæggende for vedligeholdelse, da tilføjelsen af en ny type vil kræve modifikationer i klienterne.

Udslag i målene

Argumentet for at overholde LSP er at sematikken ikke må ændre sig i nedarvede klasser. Hvorvidt LSP er overholdt kan ikke identificeres af nogen af de gennemgåede softwaremål.

5.8 Sammenfatning

Kapitlet gennemgik en række principper som hjælper til et godt vedligeholdelsesvenligt design som øger muligheden for genbrug. Undervejs er der argumenteret for hvordan forskellige mål kan blive påvirket af brugen af designprincipperne.

- Larmans *GRASP*-mønstre giver på et abstrakt plan principper for design i det objektorienterede paradigme. Nogle af disse principper vil påvirke softwaremålene.

- Et delsystem skal være åbent for udvidelse og tilpasning, men lukket for ændringer. Et delsystem der bruges af klienter må aldrig modificeres. Nøglen til open-closed principle er abstraktioner (`interface`'s i Java), så kode kan erstattes uden at modificere koden i det delsystem der ønskes genbrugt.

- Design mod et interface. Dette gennemgående princip siger at det så vidt muligt skal undgås at implementere et system så det er afhængigt af konkrete implementationer, men i stedet så det kun er afhængigt at et interface. Dette betyder også at principperne for information hiding skal overholdes, så alle data bør være private.

- Ingen af de gennemgåede mål er i stand til at registrere om princippet om at designe mod et interface bliver brudt, selvom det bør kunne lade sig gøre at opstille sådan et mål.

- Law of Demeter sætter begrænsninger for hvilke objekter et givent objekt må kommunikere med. Loven sikre lavere kobling mellem objekter.

- Liskov Substitution Principle siger at subklasser til en superklasse skal opføre sig "som forventet". Overskrevne metoder i en subklasse må ikke variere fra det der forventes af metoden der overskrives. Der er ingen af de gennemgåede softwaremål der kan registrere om princippet er brudt.

- Brud på nogle af principperne vil give udslag i softwaremålene, så en dybere undersøgelse af målene er vigtig for at kunne konstatere om softwaremålene kan identificere problematisk kode.

Kapitel 6

Komponenter & Frameworks

I dette kapitel gennemgås principper for design af komponenter og frameworks. Da årsagen til at komponenter og frameworks udvikles er at de skal kunne genbruges, er det vigtigt med en gennemgang af principperne for designet af disse.

Gennemgangen foretages for at se om designprincipperne for udviklingen af komponenter og frameworks giver udslag i softwaremålene.

Konklusionen af kapitlet er, at ingen af de gennemgåede mål umiddelbart kan bruges til at evaluere designet af komponenter og frameworks.

6.1 Introduktion

De generelle designprincipper som er beskrevet i kapitel 5, er beskrevet i litteraturen fordi de menes at give et godt modulært design, som er lettere at vedligeholde end hvis principperne er brudt. I dette kapitel vil nogle principper for design af kode som specifikt er af hensyn til genbrug blive gennemgået. Både komponenter og frameworks bliver udviklet fordi det er meningen at de skal indgå i forskellige kontekster. Derfor er det interessant at beskrive nogle principper for design af disse sammenholdt med de gennemgåede softwaremål.

Mange designmønstre som er beskrevet, af f.eks. Gamma et al. [GHJV95], er relevante at beskrive i forbindelse med design af komponenter, frameworks og generelt design. Plads forbyder at beskrive dem alle, så enkelte mønstre som er meget anvendt indenfor komponent og frameworkdesign er blevet udvalgt, og der fokuseres på disse.

6.2 Komponenter

For at kunne identificere gode principper for design af genbrugelige komponenter, er det nødvendigt at diskutere begrebet komponent.

Når der i det følgende skrives om komponenter, er det *softwarekomponenter* der menes. I nogen sammenhænge defineres genbrugelige komponenter som alle *workproducts*; use cases, testspecifikationer etc., der er designet så de kan bruges igen [JGJ97b]. Dette er relevant for generelt genbrug, men ligger udenfor specialets område.

Object Management Group, OMG, skriver følgende om begrebet komponent:

> "A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces" [Gro03].

En komponent er således et modulært delsystem, som indkapsler løsningen på et problem, således at delsystemet kan indsættes og udskiftes i en større sammenhæng. Kommunikationen med komponenten foregår gennem et eller flere interfaces.

D'Souza et al. beskriver kodekomponenter på følgende måde:

> "A coherent package of software implementation that (a) can be independently developed and delivered, (b) has explicit and well-specified interfaces for the services it provides, (c) has explicit and well-specified interfaces for services it expects from others, and (d) can be composed with other components, perhaps customizing some of their properties, whithout modifying the components themselves" [DW99].

D'Souza et al.'s definition er ikke i strid med OMG definitionen, men tilføjer: At en komponent kan være sammensat af andre komponenter, og kan give mulighed for tilpasning (punkt d), og at en komponent kan specificere hvilke interfaces den regner med at få stillet til rådighed fra omgivelserne (punkt c). Tilpasningen skal dog foregå uden at der ændres i selve komponenten.

Samlet kan vi stille følgende krav til komponenter:

- Nogle egenskaber skal kunne tilpasses for at øge genanvendeligheden af komponenten.
- Komponenten skal være kohæsivt, dvs. der skal være en sammenhæng mellem de opgaver komponenten varetager.
- Komponenten skal kunne udvikles uafhængigt af de systemer det kommer til at indgå i.
- Komponenten skal kunne indsættes i en kontekst.
- Komponenten skal kunne erstattes af et andet komponent, som tilbyder mindst det samme interface som forgængeren.

- Implementationen skal være indkapslet, og der kommunikeres kun med komponenten gennem interfacet.

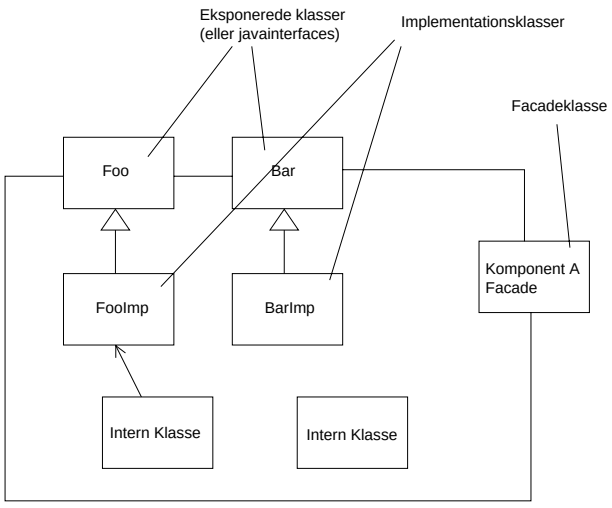
Det interessante spørgsmål er hvordan ovenstående opnås, og om der er en sammenhæng mellem nogle af de gennemgåede mål og ovenstående krav. Derfor beskrives i det følgende nogle designprincipper som benyttes i komponentdesign.

Da det ikke er realistisk, eller ønskværdigt, at implementere en komponent så alle facetter kan tilpasses, er det væsentligt at identificere de steder i komponenten som det er sandsynligt vil ændre sig. Selve identifikationen af disse variationspunkter er et af de største problemer ved godt komponentdesign [Sch99]. De mekanismer der kan bruges til at implementere *variabiliteten* [JGJ97b], polymorfi etc. er beskrevet allerede.

6.2.1 Designprincipper

De principper der allerede er gennemgået bør også bruges internt i en komponent. Det som adskiller komponenter fra f.eks. klasser eller frameworks er, at komponenter udgør et mere komplet delsystem. En komponent kan bestå af flere samarbejdende dele, men dette faktum er skjult for klienter. Samtidig skal en komponent imødekomme tilpasninger for at øge muligheden for genbrug af komponenten.

Et princip som bruges i konstruktionen af komponenter er at definere interfacene til komponenten som abstrakte klasser (eller interfaces), og implementere en *façade* som giver adgang til instanser af disse klasser [Sch99].



Figur 6.1: Design af en komponent (fra [Sch99]).

Som det ses i figur 6.1, er der tre typer af klasser i komponenten: Abstrakte klasser, eller interfaces som er synlige for en bruger af komponenten,

en façadeklasse som giver adgang til konkrete klasser i komponenten, og konkrete klasser som implementerer funktionaliteten.

Når det bliver nødvendigt at foretage ændringer i selve komponenten, kan dette ofte gøres uden at det påvirker klienterne til komponenten. Kun hvis ændringerne medfører modificering af interfacene der publiceres af façadeklassen kan det betyde at ændringer i klienterne skal gennemføres. Samtidig gives der mulighed for at publicere nye interfaces, som kan eksistere sammen med ældre interfaces. Det betyder at det er muligt at udvide funktionaliteten i en komponent, og samtidig beholde kompatibiliteten til ældre systemer som er afhængig af komponenten.

6.2.2 Tilpasning

For at en komponent kan tilpasses, kan nogle af de variabilitetsmekanismer der tidligere er omtalt anvendes. Hvorvidt disse mekanismer er anvendt kan ikke afsløres af nogen af de mål der er blevet gennemgået. Der er derfor næppe muligt at bruge nogle af de gennemgåede mål til af afgøre om en komponent er muligt at tilpasse i tilstrækkelig grad.

En mulig måde at forsøge at afsløre noget om fleksibiliteten i en komponent, kunne være at betragte ind -og uddata fra et komponenet. Udfra tidligere beskrevne principper må man forvente at fleksibiliteten af en komponent vil være større jo højere grad komponenten er afhængig af interfaces og abstrake klasser i forhold til konkrete klasser. En analyse af typen af ind -og uddata kan måske indikere noget om fleksibiliteten.

6.2.3 Uafhængig udvikling

Kravet om uafhængig udvikling er et krav om afgrænsning og dekomposition. Kravet er relateret til analyse og design og må betragtes før der findes kildekode. Derfor giver det ikke mening at betragte dette krav i forbindelse med statisk analyse af kildekode.

Et andet perspektiv kan dog anlægges på kravet, idet uafhængig udvikling kræver at komponenten er afkoblet og veldefineret i sit interface. Dette behandles i afsnittet om information hiding.

6.2.4 Erstatning

Erstatning betyder at en komponent bliver udskiftet med et andet i en fast kontekst. Kravet om muligheden for erstatning af en komponent betyder at evolutionen i en komponent ikke må ændre i interfacet til komponenten. Overholdes princippet for komponentudvikling som beskrevet betyder det at evolutionen af komponenten skal foregå således at interfacet *aldrig* ændrer sig. Dette er fuldstændig som *open-closed-principle*.

Det er dog sandsynligt at der på et tidspunkt vil blive behov for at ændre i interfacet til komponenten. Ovenstående konstruktion af komponenter

giver mulighed for at publicere forskellige interfaces, og derved give udvidede interfaces til de klienter der kender til disse. Således kan komponenter vedligeholdes og videreudvikles, samtidig med at klienter som er afhængige af tidligere versioner af komponenten ikke skal modificeres.

Små ændringer, som fejlrettelser, behøver ikke at betyde at interfacet til komponenten ændrer sig. Derfor vil nogle rettelser ikke betyde noget for erstatningen af en komponent. Andre rettelser vil tilføje metoder til interfacet. Dette betyder en udvidelse af interfacet, men behøver heller ikke at medføre ændringer i klienter til tidligere versioner af komponenten, da "ældre" klienterne ikke bruger disse nye metoder. Udvidelsen af et interface behøver derfor heller ikke at medføre ændringer i klienterne. Til gengæld opstår der problemer hvis der bliver nødvendigt at *ændre* et interface. Som tidligere nævnt, siger opened-closed principle, at softwaredele skal være åbne for udvidelser og lukkede for ændringer. En komponent må derfor ikke ændre i sit interface, men i stedet eksponere et nyt interface som nye klienter kan benytte, samtidig med at de gamle interfaces bevares.

Et eksempel på et brud på disse principper er fra Sun's GUI-bibliotes *Swing*. I Swing findes komponenten `JFrame`, som er et vindue til præsentation af data for en bruger, og som er en udvidet version af `Frame`. `JFrame` nedarver i øvrigt fra `Frame`. På `Frame` tilføjes elementer; knapper, tekster etc. med metoden `frame.add(...)`, mens man på et `JFrame`-komponent skal bruge `jframe.getContentPane().add(...)`. Hvis en udvikler ønsker at erstatte `Frame`-komponenter med `JFrame`, skal der ændres alle de steder der tilføjes noget til `Frame`-komponenten. Designerne af `JFrame` kunne have valgt at implementere `JFrame` som en façade, og dermed givet mulighed for at anvende komponenten på samme måde som dets forgænger `Frame`. Det er interessant at bemærke at implementationen af `JFrame` bryder med både Liskov Substitution Principle og Law of Demeter.

6.2.5 Information Hiding

Kravet om *information hiding* er vigtigt fordi implementation bag et interface giver mulighed for at modificere implementationen af komponenten uden at det bliver nødvendigt med ændringer i de klienter der bruger komponenten. Der er allerede argumenteret for princippet i kapitel 5.

6.2.6 Kohæsion

Kravet om høj kohæsion i en komponent er ikke anderledes end kravet til høj kohæsion i en klasse på et lavere abstraktionsniveau. Ønsket om høj genanvendelighed for komponenten betyder at komponenten ikke skal indeholde mange urelaterede ansvarsområder.

Da abstraktionsniveauet er højere for komponenten end for klasser, er det interessant at diskutere om de eksisterende kohæsionsmål er tilstrækkelige

eller i det hele taget passende for hele komponenter.

I følge D'Souza et al.s komponentdefinition, kan en komponent være sammensat af flere andre komponenter. Disse indlejrede komponenter kan i princippet arbejde på den samme mængde data, men det er sandsynligt at de i meget høj grad også har egne instansvariable som de ikke deler med andre komponenter. Dette betyder at der formentlig vil være en mængde af data der er eksklusiv for et indlejret komponent.

I princippet kan en komponent godt bestå af en enkelt klasse [DW99], men det vil normalt indeholde flere klasser af to årsager: For det første er det sjældent en enkelt klasse er meningsfuld når den står alene [SB98], og for det andet er fordelene ved genbrug af små enkelt-klassekomponenter mindre end ved genbrug af en større mængde samarbejdende klasser.

Da de gennemgåede kohæsionsmål for klasser, typisk er baseret på delt brug af instansvariable i klassen, kan disse mål ikke bruges i forbindelse med beregning af kohæsion på en komponent. Dette minder om de problemer der findes i beregning af kohæsion på pakkeniveau.

Alternativt kunne kohæsionsmålet CoH af Chen & Lu (ref. i [AL97]), som er baseret på mængder af argumenter til metoder være interessant i evalueringen af kohæsionen i en komponent. Beregningen kunne begrænses til at betragte de metoder der findes i interfacet til komponenten, og kunne derfor indikere noget om sammenhængen mellem de metoder interfacet tilbyder. Ideen forkastes dog at to grunde: Målet kan ikke foretages automatisk af et evalueringsværktøj, uden metainformation fra udvikleren om hvilke argumenter der repræsenterer samme data. Selvom der ikke er et stort overlap mellem de argumenter der sendes til en komponent, er det ikke ensbetydende med at komponenten ikke er kohæsivt. Forestiller man sig en komponent der kommunikerer med en seriel port, ved brug af en given protokol, kan komponenten have en `openPort(String comPort)` metode som specificerer porten der benyttes. Dette argument behøver ikke findes i andre af argumenterne til metoder i komponenten, men komponenten kan ikke opfattes mindre kohæsivt af denne grund.

Et generelt problem for kohæsion i komponenter er grænserne for komponenten. Der er intet syntaktisk der fortæller hvad der er en del af komponenten og hvad der ligger udenfor. I Java er det en mulighed at vedtage at ethvert komponent skal findes i en `package` for sig og derved definere komponenten, men denne løsning er sprogspecifik. Der er samtidig ingen garanti for at reglen overholdes, så der kan f.eks. ligge flere komponenter i en `package`. Dette afgrænsningsproblem skal løses for at det overhovedet giver mening at tale om kohæsion i en komponent.

Hvis projektet er at udvikle en komponent, er det klart at komponenten er afgrænset af alle de klasser der indgår i projektet, da intet andet end komponenten selv indgår i projektet.

De mål der er blevet gennemgået er alle baseret på fælles brug af data, og de er konstrueret for klasser og ikke samlinger af samarbejdende klasser.

Derfor skal målene tilpasses så de kan anvendes på komponenter.

Når afgrænsningsproblemet er løst, kunne kohæsionsmål for komponenter konstrueres på forskellig måde. En mulighed kunne være blot at finde gennemsnitskohæsionen for samtlige klasser som indgår i komponenten. Dette vil imidlertid ikke afsløre sammenhængen mellem de klasser der indgår i komponenten. Derfor er en mere interessant løsning at bruge de eksisterende kohæsionsmål, men gå et abstraktionsniveau op, og betragte samarbejdende klasser indenfor en komponent, på samme måde som vi har betragtet samarbejdende metoder indenfor en klasse. Afhængigt af programmeringssproget vil det ligeledes være muligt at betragte data som deles mellem klasser, på samme måde som data kan deles mellem metoder indenfor en klasse. I Java f.eks. er *default*-synligheden for en attribut pakkesynlighed og kunne derofr afsløre delte data i komponenten. Det skal bemærkes at designprincipperne anbefaler at alle attributter erklæres `private`, så i lyset af nogle af designprincipperne er metoden for svag.

Det må konkluderes at kohæsionsmålene ikke uden videre kan bruges i forbindelse med måling af kohæsion i komponenter. Det ene problem er afgrænsningsproblemet, og det andet er at de eksisterende kohæsionsmål er baseret på fælles brug af data. Hvis de klasser der indgår i kompontet overholder principperne om information hiding, kan kohæsionsmålene ikke baseres på fælles data.

6.2.7 Kobling

Kobling skal, som for det øvrige design og af samme årsager, minimeres i komponenter. Et krav der kunne stilles til en komponent er at der ikke findes koblinger fra komponenten til elementer der ligger uden for komponenten selv. Dette fremgår ikke af nogen af de gennemgåede mål, men følgende kunne valideres statisk for komponenter: Ingen argumenter til, eller returværdier fra en komponent må være andet end interfaces. Valideringen af kravet vil være sprogspecifikt. I Java ville kravet være at ingen argumenter til eller returværdier fra en komponent, må være andet end: Primitive typer, abstrakte klasser eller interfaces.

6.3 Frameworks

Hvor man kan tænke på komponenter som genbrugelige, primært funktionelle, dele: Kodegenbrug, kan man tænke på frameworks som genbrugelige strukture: Designgenbrug. I komponentgenbrug skriver udvikleren kode som kalder komponenterne, som derved bliver genbrugt. Dette forholder sig omvendt i genbrug af frameworks. Strukturen i programmet ligger fast, og udvikleren skriver kode som frameworket kalder på passende tidspunkter. Der tales derfor ofte om *inversion of control* når frameworkprincipper forklares.

Framworks er allerede defineret i kapitel 2, hvor der blev argumenteret for udviklingen af frameworks. I det følgende behandles designet af frameworks.

Følgende punkter karakteriserer et framework [Lar02]:

- Frameworks er en kohæsiv samling af *interfaces* og *klasser* der sammen udbyder en service som er central for en ikke-varierende del af et delsystem.
- Et framework indeholder konkrete og abstrakte klasser som definerer interfaces, samt invariante interaktioner mellem objekter.
- Et framework kan tilpasses f.eks. gennem nedarvning af eksisterende klasser fra frameworket.
- Frameworket kan indeholde klasser med både abstrakte og konkrete metoder.
- Frameworks gør intensiv brug af *The Hollywood Principle*.

The Hollywood Principle er et princip der bygger på at frameworket kalder metoder i klasser som kun er defineret abstrakt, eller gennem et interface. Brugeren af frameworket er ansvarlig for at implementere klasser der overholder interfacene, og for at registrere disse implementationer i frameworket. Princippet er navngivet *The Hollywood Principle* da de konkrete implementationer brugeren udvikler bliver kaldt når der bliver brug for dem. *"Don't call us, we'll call you."*

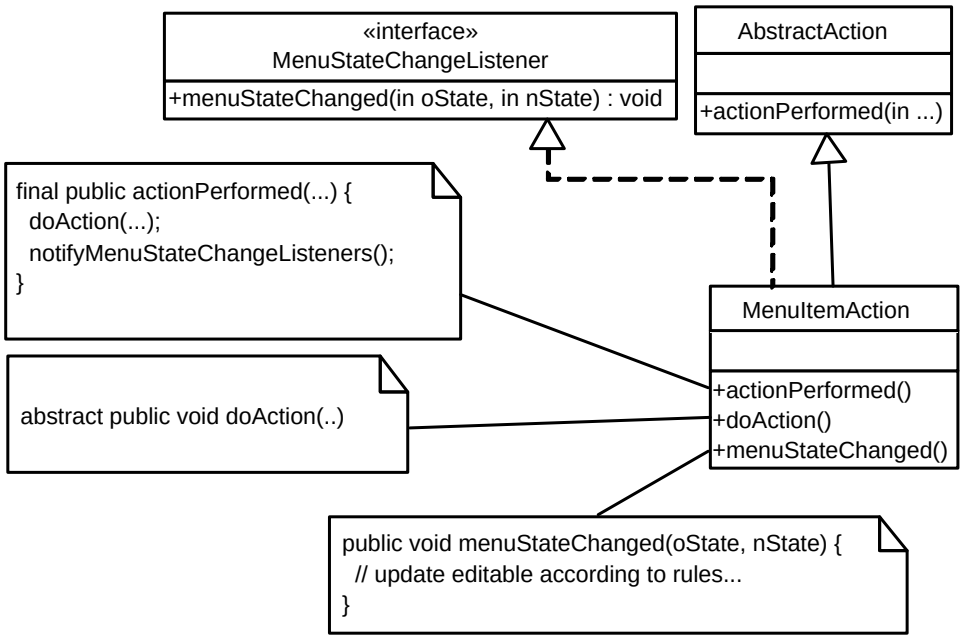
6.3.1 Design

En af nøgleteknikkerne i design af framworks, er brugen af *abstrakte* klasser [Joh97]. Abstrakte klasser er klasser der ikke kan konstrueres instanser af og derfor kun kan bruges som superklasse for andre klasser. Superklassen indeholder typisk mindst én abstrakt metode, som kræves overskrevet i subklasserne. Et mønster som beskriver denne metode, *template* mønsteret, bliver gennemgået senere.

En anden konsekvens af abstrakte klasser er at klasserne, som er erklæret abstrakte, definerer et interface for subklasserne til denne. Dette betyder at teknikker som benytter polymorfi kan bruges.

Template pattern

Template mønsteret [GHJV95] er meget anvendt i frameworkdesign. Mønsteret giver mulighed for at definere en algoritme i en superklasse, og lade dele algoritmen være overladt til abstrakte metoder, som skal implementeres i subklasser. På den måde kan strukturen i algoritmen genbruges så man har en løsning på et generelt problem. Konkret brug af algoritmen kræver implementation af de variable dele af algoritmen.



Figur 6.2: Eksempel på template mønsteret

I figur 6.2 ses et UML diagram over et framework til menuhåndtering i et program. Menusystemet skal håndtere tilgængeligheden af menupunkter. Et menupunkt har en attribut som angiver om menupunktet kan aktiveres. Attributten kan have to tilstande: Tilgængelig, (*enabled*) og ikke tilgængelig (*disabled*). Denne attribut på menupunktet skal menusystemet ændre afhængig af den status menusystemet har. Tilgængeligheden af de forskellige menupunkter ændres når brugeren aktiverer menupunkterne. For at implementere dette, kan et template mønster benyttes. Superklassen `MenuItemAction` implementerer algoritmen for test og eksekvering af menupunkterne.

For at lave et nyt menupunkt, nedarves fra `MenuItemAction`, og den abstrakte metode `doAction()` fra `MenuItemAction` overskrives. Metoden `actionPerformed` er den metode der bliver kaldt når der trykkes på en knap. Denne metode er implementeret i superklassen, `MenuItemAction`, og erklæret `static` så den ikke kan overskrives i subklasserne. `actionPerformed` kalder den egentlige funktionalitet i subklassen, og kalder alle klasser der er registreret som `MenuStateChangeListener`, hvilket alle konkrete `MenuItemAction` skal være. Således bliver alle funktioner informeret om den nye tilstand af menuen, og man er garanteret af *enabled*/*disabled* attributten bliver opdateret.

Den metode der definerer algoritmen, i ovenstående tilfælde `actionPerformed` i `MenuItemAction` er *template metoden*, de abstrakte metoder der bliver kaldt fra template metoden kaldes *hooks* eller *hot-spots*. Sidstnævnte metoder bestemmer hvor variationen i delsystemet kan forekomme. Præcis

dette er også en svaghed ved genbrug gennem *hook*-metoder: Alle de variable dele af algoritmen skal identificeres og implementeres således at de kan overskrives i subklasser. Bliver de ikke det, er delsystemet begrænset i genbrug. Dette er et alvorligt problem, da udviklingen af et successfuldt framework kræver identifikaton af hot-spots [PPK, p.3].

Nøglen til frameworkdesign er fleksibilitet gennem *hot-spots*, som f.eks. implementeres gennem et *template*-mønster. Templatemønsteret giver mulighed for at fastholde en overordnet algoritme og variere elementer i algoritmen. I nogle sammenhænge er det ønskværdigt med endnu større fleksibilitet, hvor hele algoritmen kan variere. Til dette kan et *strategy*-mønster anvendes. Her bliver er større ansvarsområde uddelegeret til en klasse som nedarver fra en abstrakt klasse der definerer det interface klienten må have kendskab til.

6.3.2 Kobling

Template mønsteret giver, på grund af nedarvningen, klasser som er koblet til hinanden. Disse koblinger påvirker ikke Chidamber & Kemerer's koblingsmål CBO citeChid94 fordi CBO betragter nedarvede metoder som klassens egne metoder. Hitz & Montazeri [HM96] opstiller ikke et egentligt mål, men konstaterer bare at koblingen til en superklasse ikke er alvorlig. Price & Demurjian [PSAD97] mener heller ikke at denne form for kobling udgør et problem for genbrug, da subklassen ikke skal kunne bruges uafhængigt af superklassen.

Mål som bliver påvirket af template mønsteret er dybden af klassehierarkiet, DIT, samt antallet af børn til en klasse, NOC. Dette betyder ifølge, Chidamber & Kemerer [CK94], at måden de klasser der indgår i klassehierarkiet virker på, bliver sværere at forudsige. Dette viser sig også i ovenstående eksempel hvor kaldet til en metode `doAction` medfører opdatering af menuens tilstand, uden at den implementerede brugerfunktion kender til dette.

6.3.3 Kohæsion

Kohæsionen i frameworks, er som for de andre softwaredele, et spørgsmål om sammenhængen mellem de elementer der indgår i frameworket. Da et framework definerer en større samarbejdende mængde af elementer, er det sandsynligt at kohæsionen i nogle klasser i et framework vil blive målt lav. Det er dog ikke muligt at sige noget generelt om dette.

6.3.4 Problemer med frameworks

Paradoksalt nok, kan udviklingen af frameworks have en negativ indflydelse på vedligeholdelsen af et system. Da frameworks gør meget brug af templatemønsteret, og dermed nedarvning, kan frameworks have nogle af de problemer der også kan opleves i forbindelse med nedarvning. Specielt må nævnes problemet med skrøbelige basisklasser, som tidligere er blevet gennemgået,

da template-mønsteret er baseret på deling af ansvarsområder mellem super- og subklasser. Forståelsen af et framework kan være svær af samme årsag. Dette er beskrevet tidligere som *yo-yo*-problemet. Konstrueres frameworks i Java, er der yderligere det problem, at enhver `public`-metode, som ikke er erklæret `final`, potentielt kan overskrives. Et udgivet framework bliver derfor meget låst i forhold til videreudvikling.

Ligesom for anden udvikling bør afhængigheder mellem klasser kun være baseret på interfaces. Dette betyder at metoder der overskrives i nedarvede klasser som en del af templatemønsteret ikke bør referere direkte til attributter i superklassen.

Da applikationer der er konstrueret udfra et framework er meget afhængige af interfacene i frameworket, er applikationerne nødt til at udvikle sig sammen med den videre udvikling i frameworket. Derfor er lav kobling vigtig i framework design. Hvis koblingen mellem elementer i frameworket og de applikationer der bruger frameworket er høj, vil en ændring i frameworket kunne kræve ændringer i samtlige applikationer der benytter frameworket [GHJV95]. Price & Demurjian [PSAD97] skriver at koblinger fra en nedarvet klasse til en superklasse ikke er væsentlig i forbindelse med genbrug, fordi en subklasse ikke skal genbruges uden sin superklasse. Dette er korrekt for simpel genbrug, men koblinger fra subklasser til superklasser er væsentlig i forbindelse med vedligeholdelse, når det er sandsynligt at der er behov for videreudvikling i superklasserne. Dette er sandsynligt i frameworks fordi et framework kan betragtes som et selvstændigt system.

6.4 Sammenfatning

Der er en forskel på den måde man kan genbruge komponenter og frameworks, men der er mange overlappende principper. Disse overlappende principper gør at det til tider kan være svært at skelne mellem en komponent og et framework fordi grænsen mellem disse to begreber bliver udvisket. Det er dog ikke væsentligt at skelne mellem komponenter og frameworks for at implementere genanvendeligt og vedligeholdelsesvenligt software. Det væsentlige består netop i at resultatet bliver genanvendeligt og vedligeholdelsesvenligt.

- Identifikationen af variationspunkter er et af de største problemer ved godt komponentdesign.
- Komponenter bør implementeres ved brug af *façader*.
- Komponenter bør gøre brug af information hiding for bedre at kunne skjule effekten af en ændring i komponenten for klienterne til komponenten.
- Frameworks lægger vægt på *design genbrug* i højere grad end kodegenbrug.

- Frameworks bruger *Inversion of control* (Hollywood principle).
- Frameworks giver meget høj grad af genbrug.
- Frameworks kræver lav kobling begge veje i klassehierarkiet.
- Kohæsionsmålene kan ikke uden videre bruges til at finde kohæsion i komponenter.
- Koblingsmålene kan ikke uden metadata bruges til evaluering af komponenter, da viden om hvad der tilhører komponenten og hvad der ligger udenfor, skal haves i koblingsmålet.
- Chidamber & Kemerer's koblingsmål CBO kan ikke bruges til at evaluere frameworks, da målet ikke tager hensyn til koblinger mellem super- og subklasser.
- Det er ikke muligt at finde kohæsionen i et helt framework med de gennemgåede kohæsionsmål.

Kapitel 7

Mangler ved eksisterende mål

I dette kapitel diskuteres det om der er åbenbare områder der beskrives i designprincipperne, der ikke berøres af nogen af softwaremålene. Dette er vigtigt, da vi gerne vil kunne identificere problematisk kode.

Konklusionen på kapitlet er, at det ikke vil være muligt at konstruere et mål der identificerer klasser der bør smeltes sammen. Dette "inverse" kohæsionsmål vil blive påvirket af gode designmønstre som f.eks. Model-View-Controller. Klasser der f.eks. er udviklet efter dette mønster, vil give udslag i målene, hvilket ikke er hensigtsmæssigt.

Der blev konstrueret kendskabsmål der tæller antallet af klasser eller interfaces en klasse er bekendt med. Dette mål kan bruges til at evaluere halvfærdige systemer med i stedet for koblingsmål.

Der blev konstrueret to abstraktionsfaktormål, som afspejler brugen af interfaces i koden. Det ene mål er baseret på kendskabsmålet, mens det andet er baseret på koblingsmålet, CBO, fra Chidamber & Kemerer [CK94].

7.1 Introduktion

I gennemgangen af softwaremål og designprincipper er der mindst to emner der ikke er blevet nævnt i litteraturen. Første problem er at stærke koblinger mellem to klasser kan betyde at disse klasser bør lægges sammen, på samme måde som lav kohæsion i en klasse kan være en indikation af at klassen bør splittes op. Det andet problem er at et væsentligt designprincip siger at der skal udvikles mod en klasses interface for at opnå stor fleksibilitet. Dette princip afspejles ikke i nogen af de gennemgåede mål. I dette kapitel diskuteres disse to problemer. Målene tager udgangspunkt i sproget Java.

7.2 Det "inverse" kohæsionsmål

Når to klasser er meget tæt koblet til hinanden, kan det være et udtryk for at nogle af designprincipperne ikke er overholdt. Larman beskriver mønsteret *information expert* [Lar02], som tildeler ansvarsområder til klasser afhængigt af de data de har til rådighed. Hvis man forestiller sig brud på dette princip, kan det føre til at to klasser kommunikerer meget med hinanden, fordi den ene klasse har data som den anden klasse skal bruge for at udføre sine ansvarsområder.

Betragtes to sådanne klasser, kan de begge godt have høj kohæsion efter de mål vi kender, fordi kohæsionsmålene som udgangspunkt bruger instansdata som basis for beregningen. Data som bliver hentet fra en anden klasse indgår ikke i kohæsionsberegningen. Koblingsmålene kan, men behøver ikke at blive påvirket i særlig stor grad. En klasse `A` der ofte henter data fra en klasse `B`, behøver ikke at være koblet til mange andre klasser. Samtidig tæller koblingen til `B` kun én enkelt gang, så den intensive brug af `B` vil ikke nødvendigvis påvirke koblingsmålene i særlig stor grad.

Ét af problemerne ved ovenstående eksempel er, at der er kobling mellem `A` og `B`, så ændres der i den ene klasse, kan det medføre ændringer i den anden. Dette er generelt for kobling. Et andet problem er, at `B` kan være nødt til at eksponere metoder til at hente og gemme data i sit interface, fordi `B` indeholder data som kun `A` skal bruge. Dette gør `B` skrøbelig i forhold til resten af systemet, fordi andre klasser også får mulighed for at modificere `B`'s data. Samtidig bliver `B`'s interface større end det behøver at være, hvilket giver en større perciperet kompleksitet fordi udviklere skal overskue det større interface.

Ovenstående kan opstå i et uhensigtsmæssigt design, og kan rettes ved at placere data og operationer i samme klasse. Det skal bemærkes at konstruktioner som ovenstående kan være implementeret med fuldt overlæg og at det ikke er en universel regel at konstruktioner som ovenstående er "forkerte".

Med udgangspunkt i ovenstående, kunne man forestille sig en analyse af koden som forsøgte at samle klasser i stedet for at identificere klasser der bør splittes op. Denne form for "inverst kohæsionsmål" diskuteres i det følgende.

Det er ønsket at konstruere et softwaremål der afspejler en klasses intensive brug af en anden klasses data, for at identificere kode der bør flyttes. En fremgangsmåde kunne være at prøve at finde en klasse `A`, som bruger data fra en klasse `B` mere intensivt end `B` selv gør. En sådan situation kunne inddikere at attributter i `B` i stedet bør være attributter i `A`.

Selvom tanken om et sådan mål er besnærende er der problemer forbundet anvendelsen af målet. Et kendt designprincip er *Model View Controller*

[GHJV95]. Dette designprincip adskiller netop præsentation af data fra data selv. Derfor ville målet vise at data burde flyttes fra modellen til præsentationsklasserne. Samme problem vil kunne findes i andre designmønstre, f.eks. *visitor*, det vil ikke være muligt at skelne mellem de konstruktioner der udgør en designfejl, og de der er implementeret for at opnå stor fleksibilitet. Derfor vil ideen om målet forlades og ikke blive behandlet yderligere.

7.3 Abstraktionsfaktorene

Et gennemgående princip er at der skal programmeres mod en klasses interface. Det er afhængigt af programmeringssproget hvordan interfaces defineres. I Java er der to muligheder. Enten nedarves fra en abstrakt klasse (`extends`), eller også bruges Javas `interface`-konstruktion (`implements`). Fordelen ved at bruge `interface` frem for nedarvning er, at en klasse i Java har mulighed for at implementere mange interfaces, og dermed understøtte mange forskellige typer. Dette svarer til at man i C++ har mulighed for multipel nedarvning. Fordelen ved nedarvning fra den abstrakte klasse er, at nogle dele af klassen kan være implementeret og dermed give en standardimplementation af disse dele, for alle nedarvede klasser. Størst fleksibilitet opnås ved at implementere et `interface`, fordi en klasse kan implementere flere interfaces, men ikke nedarve fra flere klasser.

På trods af at princippet er vigtigt for fleksibilitet, vedligeholdelse og genbrug, er der ingen af de gennemgåede mål der tager abstraktioner i betragtning. Det skyldes at der tilsyneladende har været fokus på at forsøge at måle kompleksiteten af software, og ikke andre former for vedligeholdelsesvenlighed og genanvendelighed. Samtidig har der været et ønske om at konstruere sproguafhængige softwaremål. Da der her er afgrænset til Java, ses der bort fra sprogspecifikke problemer for andet end Java.

Det er ønsket at konstruere softwaremål der udtrykker i hvor høj grad en klasse har kendskab eller kobling til abstraktioner. Dette er relevant fordi dele af et system er nemmere at udskifte og genbruge hvis delene kun afhænger af abstraktioner. Dette er der argumenteret for i kapitel 5. I Java implementeres abstraktionerne bedst ved hjælp af `interface`. Andre sprog, som f.eks. C++ bruger abstrakte klasser. Da abstraktioner implementeres forskelligt i forskellige sprog afgrænses der til at se på Java, og dermed `interface`'s.

Målet defineres i to former: *Kendskab* til interfaces i forhold til det totale kendskab i en klasse, og *kobling* til interfaces i forhold til det totale antal koblinger i klassen.

Koblingsabstraktionsfaktoren er graden af koblinger til interfaces i en klasse. Da kobling, specielt til konkrete klasser, er beskrevet som et problem for genbrug og vedligeholdelse, er det interessant at se hvor mange af

disse koblinger der er til interfaces, som udgør et mindre problem, eller direkte et bedre design. I princippet ønskes så høj koblingsabstraktionsfaktor som muligt: Så mange af de eksisterende koblinger som muligt, skal være til interfaces i stedet for til konkrete klasser.

Kendskab til en klasser eller et interface, er viden om klassens eller interfacets eksistens. Det betyder at mængden af klasser og interfaces klassen `A` kender til, indeholder de klasser som `A` er koblet til, men også de klasser og interfaces som `A` f.eks. modtager i som parameter til en metode, hvor parameteren blot bliver videresendt til en delegeret klasse. Kobling mellem `A` og `B` opstår i følge Chidamber & Kemerer [CK94], når der findes en reference fra en metode i `A` til en metode eller en attribut i `B`. Hvis en metode i `A` modtager en parameter af typen `B`, men ikke udføre nogen af `B`'s metoder, eller refererer til nogen af `B`'s attributter, er der ikke kobling mellem `A` og `B`.

Grunden til at et mål baseret på *kendskabet* til klasser og interfaces er interessant, er at målet bliver anvendeligt på delsystemer. Hvis der f.eks. udvikles en klasse `A` som indeholder en metode `m` der returnerer en klasse `R`, vil dette forhold ikke afpejles i noget koblingsmål, da der ikke nødvendigvis findes nogen brug af metoden i resten af systemet. Bruges kendskab til returtypen afspejles det i abstraktionsfaktoren om der returneres et interface eller en konkret klasse. Hvis systemet der evalueres er ukomplet, påvirker dette derfor ikke kendskabsabstraktionsfaktoren. Koblingen, og dermed koblingsabstraktionsfaktoren, vil være påvirket af at det er et delsystem, og ikke et komplet systmen der evalueres.

7.3.1 Kendskab til typer i Java

Generelt findes der kendskab til en klasse eller et interface hver gang der refereres til klassen eller interfacet i kildekoden. I Java kan dette foregå på følgende måder, hvor `A` er en klasse der kender til typen `B`:

- Der findes en attribut af typen `B` i `A`.
- Der findes en metode i `A` der tager et argument af typen `B`.
- Der findes en metode i `A` der returnerer typen `B`.
- Der findes en lokal variabel i en metode i `A` af typen `B`.
- Der findes en *type cast* i en metode i `A` til typen `B`.
- `A` nedarver fra `B`.
- `A` implementerer `B`.
- Der findes en `catch(B)` i en metode i `A` (`B` er nedarvet fra `java.lang.Exception`)
- Typen `B` er nedarvet fra `java.lang.Exception`, og kan "kastes" fra `A`.

- Det konstrueres et objekt af typen `B` i `A`.
- Der findes kobling fra `A` til `B`.

Java er ikke 100% objektorienteret idet der findes primitive typer som f.eks. `boolean`, `int`, `float` osv. Disse udgør ikke et problem i forhold til kendskabet, da deres "interface" aldrig ændrer sig. Derfor tælles primitive typer ikke med i målene.

Tilbage er der typer fra standardklassebiblioteket, og typer som udvikleren eller tredjepart har udviklet. Ofte udgør kendskab er til standardklassebiblioteket ikke noget problem, da deres interface er meget stabilt. En klasse som f.eks. `java.lang.String` ændrer kun meget sjældent sit interface. Dog har vi set eksempler på dele af Sun's standard SDK der faktisk har ændret sig, og det har yderligere været et ønske at softwaremålene skulle være uafhængige af metadata. Derfor vil referencer til klasser fra SDK'et også indgå i beregningerne af abstraktionsfaktorene.

Når en klasse implementerer et interface, betyder det at den er nemmere at erstatte i den kontekst den indgår i, fordi andre implementationer let kan overtage den aktuelle klasses plads i systemet. Dette forhold bliver afspejlet i målet, fordi det kræves at det omkringliggende system er afhængigt af interfacet til klassen. Implementerede interfaces og en eventuel superklasse til den klasse der evalueres vil derfor ikke blive medtaget i målet. Med andre ord betragtes kun *brugen* af klasser og interfaces. Dette menes at være nok til at afgøre graden af abstraktion i systemet. I denne sammenhæng skal det præciseres at *brugen* af en klasse f.eks. er at finde i kodelinien: `if (o instanceof someClass)...`. En sådan reference til `someClass` vil ikke fremgå af koblingsmålet CBO.

Det er ønsket at målet skal udtrykke noget om fleksibiliteten i en enkelt klasse, relateret til kendskabet mellem klasser. Målet kan dog udvides så det virker på en mængde af klasser, og dermed være interessant til evaluering af komponenter. Hvis målet bruges på komponenter vil afgrænsningsproblemet igen opstå, altså det forhold at det skal kunne afgøres hvad der indgår i kompontet og hvad der ikke gør.

Hvis ID (interface definitions) er mængden af interfaces i systemet, kan selve målet, kendskabsabstraktionsfaktoren, $KAF(C_i)$ for klassen C_i, udtrykkes på følgende måde, hvor IK er mængden af *interface kendskab* i en klasse, og TK er det totale kendskab i en klasse.

$$KAF(C_i) = \frac{|IR(C_i)|}{|TR(C_i)|} \tag{7.1}$$

hvor,

$$TR(C_i) = \{r \mid r \textit{ kender til i } C_i\}$$

og

$$IR(C_i) = \{r \mid r \in TR(C_i) \wedge (r \in\ ID)\}$$

Med andre ord er KAF forholdet mellem antallet af distinkt kendskab til interfaces i forhold til det samlede antal distinkte kendskaber i klassen. Målet vil derfor blive mellem nul og ét. Som tidligere nævnt ses der bort fra primitive typer. Målet baseret på kobling i stedet for kendskab defineres ækvivalent hermed, som

$$CAF(C_i) \quad = \quad \frac{|IC(C_i)|}{|TC(C_i)|} \tag{7.2}$$

hvor,

$$TC(C_i) = \{r \mid r \textit{ er koblet til i } C_i\}$$

og

$$IC(C_i) = \{r \mid r \in TC(C_i) \wedge (r \in\ ID)\}$$

Da abstrakte klasser ikke giver den samme høje fleksibilitet som interfaces, vælges det at se bort fra abstrakte klasser. Således vil referencer til abstrakte klasser blive talt med som alle andre klasser.

7.3.2 Diskussion af abstraktionsfaktorene

Målene KAF (Kendskabsabstraktionsfaktoren) og CAF (Coblingskabstraktionsfaktoren) er direkte relateret til brugen af interfaces i klasserne. Når KAF bliver nul indikerer det at klassen der undersøges kun har kendskab til konkrete klasser. I det andet ekstrem, når KAF er én, betyder det at den aktuelle klasse kun har kendskab til interfaces. Det samme gør sig gældende for koblingsabstraktionsfaktoren.

En stor værdi af koblingsabstraktionsfaktoren CAF skulle betyde at de andre dele klassen er koblet til, let kan udskiftes med nye implementationer. Dette er et argument for at CAF skal være så stor som muligt. Det er dog ikke sandsynligt at der vil findes ret mange klasser der har meget høje KAF, eller CAF-værdier. Det skyldes blandt andet brugen af Javas SDK. Mange af standard-klasserne, som f.eks. `java.lang.String`, `java.lang.Integer` osv. implementerer slet ikke noget javainterface. Derfor vil klienter til disse klasser altid være afhængige af den konkrete klasse. De fleste af disse klasser vil formentlig slet ikke ændre deres interface og kendskabet til disse er derfor ikke alvorlig. Det har blot andre konsekvenser at en klasse ikke implementerer et interface: Det bliver ikke muligt at udskifte en klasse med en anden der tilbyder det samme interface. Dermed bliver brugere af klasserne mindre fleksible.

Spørgsmålet er også hvilke negative konsekvenser der kan være ved at implementere klasser der i meget høj grad er baseret på interfaces i stedet for konkrete klasser. Det er klart at en adskillelse af interface og klasser i

Java kræver mere programmeringsarbejde. Dette betyder at mere vedligeholdelsesarbejde og samtidig kan overskueligheden af systemet blive påvirket. Disse ting betyder muligvis at systemet opfattes som mere komplekst.

De to abstraktionsmål vil blive afprøvet sammen med nogle af de øvrige mål i kapitel 8.

7.4 Sammenfatning

Kapitlet beskrev to forhold der slet ikke blev berørt af nogen af de gennemgåede softwaremål. Dette er på trods af at det ene af disse forhold, udvikling mod et interface i stedet for en konkret klasse, er vigtigt for genbrug og vedligeholdelse. Samlet beskriver kapitlet:

- En evaluering af kildekode der samler klasser i stedet for at splitte dem op, en form for "inverst kohæsionsmål" bliver ikke berørt af nogen af målene.
- Et "inverst kohæsionsmål" kan ikke konstrueres uden at give for mange resultater, da mange designmønstre adskiller data og nogle af operationerne på data, f.eks. Model-View-Control. Klasser der f.eks. er udviklet efter dette mønster, vil give udslag i målene, hvilket ikke er hensigtsmæssigt.
- Der blev konstrueret et abstraktionsmål der beskriver graden af interfaceafhængigheder i forhold til afhængigheder til klasser. Dette mål skal i princippet være så stort som muligt for at give fleksibel kode.
- Der blev konstrueret et kendskabsmål der giver en supermængde af koblingsmålet. Kendskabsmålet er ikke afhængigt af om det system der evalueres er et delsystem eller et ufærdigt system.

Kapitel 8

Eksperiment med mål

I dette kapitel eksperimenteres der med udvalgte softwaremål og designprincipper. Eksperimenterne går ud på at udføre målinger med softwaremålene på kode hvor designprincipperne er brudt og hvor de overholdes. Dette gøres for at registrere hvordan principperne påvirker målene.

Yderligere gennemføres nogle eksperimenter med kendskabs- og koblingsfaktorene. Disse mål afprøves på to større systemer, og resultaterne sammenlignes.

Det konkluderes at koblingen i systemet bliver påvirket positivt (mindre kobling), når designprincipperne overholdes. Bortset fra et eksperimentet med Law of Demeter, viste det sig at kendskabet i systemet blev også lavere efter applikationen af designprincipperne. Kildekoden til eksperimentet var generelt for lille til at man kan konkludere noget om kohæsionen målt med $LCOM*$.

Eksperimentet med abstraktionsfaktorene viste en forventet forskel mellem to evaluerede systemer.

8.1 Beskrivelse af eksperiment

Eksperimentet består af to dele. Første del består i at udvikle mindre kodeeksempler der bryder med et bestemt designprincip, evaluere koden ved hjælp af nogle udvalgte mål og sammenligne resultaterne med resulater fra evaluering af ækvivalent kode som ikke bryder med det samme designprincip. Dette gøres for at afsløre hvordan brugen af designprincipperne påvirker de udvalgte mål. Der vil blive udviklet et lille system, der er så overskueligt at det lader sig gøre at kontrollere hvilke brud der findes på designprincipperne. I det følgende kaldes dette lille system for *ExpPos*, da det er et *eksperimentelt point-of-sale system* Koden til dette system, kan ses i bilag A.1. Yderligere vil der blive udviklet små ikke-realistiske programmer som i ekstrem grad bryder med principperne.

Anden del består i at evaluere et større sammenhængende system med de udvalgte mål. Dette gøres for at se hvordan målene opfører sig i virkelige systemer i stedet for eksempelkode. Nogle af de dele af systemet der identificeres som problematiske i forhold til målene, vil blive undersøgt med henblik på at finde brud på designmålene.

Det forventes at kode der overholder designprincipperne generelt vil vise bedre resultater når den evalueres med målene. Hvis dette viser sig korrekt, betyder det at designprincipperne målbart højner kvaliteten i kildekoden i forhold til de udvalgte mål, og derfor er væsentlige i forbindelse med udvikling af software af høj kvalitet.

8.2 Valg af mål og principper

Det blev tydeligt ved gennemgangen af både softwaremål og designprincipper, at kobling og kohæsion er meget væsentlige, set i forhold til genbrug og vedligeholdelse. Derfor er det oplagt at fokusere på mål der afspejler kohæsion og kobling i et system. Yderligere er der forsøgsvis blevet opstillet to mål: Abstraktionskendskabet og abstraktionskoblingen, der måler graden af kendskab og kobling til abstraktioner i forhold til kendskabet og koblingen i hele systemet. De mål der derfor er blevet udvalgt til nærmere undersøgelse er CBO fra Chidamber & Kemerer [CK94] (se afsnit 4.2.4, $LCOM*$ [HS96] (se afsnit 4.5) og abstraktionsmålene der blev defineret i kapitel 7, afsnit 7.3. Valget af $LCOM*$ i stedet for f.eks. Hitz & Montazeri's kohæsionsmål [HM96] er på grund af tilgængeligheden af en eksisterende implementation.

Man skal være opmærksom på at de udvalgte mål har nogle "væsensforskelle". Hvor $LCOM*$ er uafhængig af klassens størrelse og abstraktionsfaktorene er uafhængig af klassens eller systemets størrelse, indgår størrelsen på systemet ikke i beregningen af CBO. Muligvis ændrer CBO sig derfor alene på grund af systemets størrelse. Imidlertid er dette ikke noget problem for anvendelsen af målet. Uanset hvor stort et system er, vil mange koblinger fra en klasse til andre være et problem, så absolutte værdier for kobling imellem klasser er anvendeligt som mål.

Det vil ikke være en overkommelig opgave at sammenholde samtlige designprincipper med de valgte mål. Derfor udvælges en delmængde af de gennemgåede principper. Nogle principper, som f.eks. *Liskov Substitution Principle*, er baseret på sematikken i de metoder der implementeres i en klasse, så et brud på princippet vil formentlig ikke vise sig i målene. Derfor fravælges dette princip. Yderligere skal princippet der undersøges være tilpas afgrænset til at man kan holde andet end princippet der undersøges invariant.

De valgte mål virker alle på klasser. Derfor afgrænses der fra at undersøge principper for udvikling af komponenter og frameworks. Følgende designprincipper udvælges:

- Information Expert [Lar02] (se afsnit 5.2.2).
- Polymorphism [Lar02] (se afsnit 5.2.4).
- Law of Demeter [LHR88, LH89] (se afsnit 5.6).
- Design mod et interface (se afsnit 5.4).

Det skal bemærkes at abstraktionsfaktoren (afsnit 7.3) kun er relevent i forbindelse med princippet om design mod et interface. Derfor bliver abstraktionsfaktorene behandlet sammen med princippet om interfaces på anden vis end de øvrige principper og mål

8.3 Værktøjer til måling

Der skal implementeres et værktøj til at beregne de valgte softwaremål. Eclipse, som er et open-source udviklingsmiljø, giver mulighed for udvikling af plug-ins[ecl, GB04]. Plug-ins udviklet til Eclipse kan gøre brug af kildekodeparser og andre værktøjer som Eclipse stiller til rådighed. Eclipse vælges derfor som platform for implementationen af softwaremålene.

Et andet open-source projekt, *Metrics* [met] implementerer et framework i hvilket softwaremål kan tilføjes. Dette framework implementerer i forvejen $LCOM*$, men implementationen viste sig desværre af være fejlbehæftet og kunne derfor ikke bruges. Hverken CBO eller abstraktionsfakorene (kendskab og kobling til interfaces) implementeres af *Metrics*, så alle fire mål implementeres i en ny plug-in, der gør brug af strukture fra Eclipse og frameworket fra *Metrics*. I forbindelse med programmeringen af målene, viste sig nogle tekniske vanskeligheder som betyder, at direkte attributtilgang i en klasse ikke tælles med i koblingen til denne klasse. Således bliver en reference på formen `aClass.anAttribute` ikke talt med. I forhold til CBO er dette en begrænsning, men i forhold til eksperimenterne er det ikke et problem, da eksempelprogrammerne kun benytter `private` deklarationen. Enhver attribut i en klasse tilgås således gennem et metodekald, `aClass.getAnAttribute()`, hvilket bliver registreret i systemet.

For de udvalgte principper måles derfor: Kohæsionen ved hjælp af $LCOM*$, koblingen ved hjælp af CBO og kendskabet i absolut værdi. Som gennemgået i kapitel 7 er kendskabet i en klasse `A` alle de klasser eller interfaces der er kendt i `A`, ikke kun dem `A` er koblet til.

8.4 Sammenhæng mellem designprincipper og mål

I dette afsnit gennemføres eksperimenter med de udvalge designprincipper og softwaremål. Der implementeres en mængde kode der bryder med designprincipperne. Samme kode omskrives så den kommer til at overholde det

designprincip der aktuelt betragtes, og de forskellige softwaremål før og efter modifikationen sammenlignes. Når koden modificeres kan det ske at metoder eller hele klasser bliver overflødigjort. Disse metoder og klasser bliver fjernet før måling.

8.4.1 Information Expert

Information Expert brydes hvis en klasse har ansvar for en operation der bør høre til i en anden klasse på grund af placeringen af de data operationen skal bruge. Et ekstremt tilfælde af brud på Information Expert vil derfor være hvis alle operationer på data fra en klasse findes i en anden klassen end den der indeholder data.

Klasse	Mål	*Før* Inf. Expert	*Efter* Inf. Expert
`DataA`	CBO	0	1
	$LCOM*$	0,5	0,2
	$Kendskab$	0	1
`OpAdd`	CBO	1	
	$LCOM*$	0	*slettet*
	$Kendskab$	1	
`OpMul`	CBO	2	
	$LCOM$	0	*slettet*
	$Kendskab$	2	
`OpSub`	CBO	1	
	$LCOM$	0	*slettet*
	$Kendskab$	1	
`Run`	CBO	4	1
	$LCOM$	0	0
	$Kendskab$	5	2
Hele systemet	$\sum(CBO)$	8	2
	$gns(CBO)$	1,6	0,4
	$gns(LCOM*)$	0,1	0,1
	$Kendskab$	9	3

Tabel 8.1: Mål før og efter 'Information Expert'.

I tabel 8.1 ses resultatet af et forsøg hvor koden netop er implementeret som ovenfor beskrevet. Det er ikke overraskende at koblingen i systemet bliver reduceret efter princippet overholdes. Alle koblinger til operationsklasserne, `OpAdd`, `OpMul` og `OpSub`, forsvinder fra `Run`. Kohæsionen i klassen `DataA`, som nu indeholder både data og operationer på disse, bliver også stærkt forbedret da der bliver tilføjet en mængde metoder på klassen som alle bruger instansvariable fra klassen selv. Som forventet giver Information Expert bedre kohæsion og mindre kobling.

Kendskabet til andre klasser bliver også mindre efter overholdelse af princippet. Klassen `Run`, som bruger `DataA` og operationsklasserne, og dermed binder disse sammen, har ikke længere kendskab til de forskellige operationsklasser.

Ved konstruktion af et ekstremt tilfælde af brud på Information Expert, skal man være opmærksom på hvordan målene fungerer. $LCOM*$ er her et problem, da kun metoder der bruger instansvariable fra klassen medtælles i målet. Det betyder at en klasse som indeholder en mængde metoder der kaldes fra andre klasser og som ikke tilgår egne instansdata, ikke medtælles i kohæsionsmålet. Derfor vil det ekstreme, tilfælde hvor der findes en dataklasse med data, og en operationsklasse som indeholder metoder der arbejder på data fra dataklassen, ikke have lav kohæsion i følge $LCOM*$. Dette kan betragtes som en svaghed ved $LCOM*$. Det skal bemærkes at stort set alle de omtalte kohæsionsmål vil have denne egenskab. Kun kohæsionsmålet CoH af Chen & Lu (ref. i [AL97]) ville muligvis give et andet resultat.

Kildekoden til brud på Information Expert kan ses i bilag A.5.

I ExpPos brydes der med Information Expert. Metoden `getSubTotal(...)` på klassen `Sale` returnerer subtotalen for en `SalesEntry`. Denne metode bør i stedet være implementeret på `SalesEntry`-klassen selv. Yderligere er genereringen af kvitteringen flyttet fra en dedikeret klasse `StdReceiptLayout`. Data til kvitteringen findes i `Sale`, og man kunne derfor bruge Information Expert til at argumentere for at genereringen af kvitteringen skal foregå i en metode til `Sale`. Rettelserne i systemet betyder ændringer i to klasser: `Sale` og `SalesEntry`. Klassen `StdReceiptLayout` bliver overflødig og derfor slettet. Kildekoden til disse ændrede klasser kan ses i bilag A.2.

Værdierne for softwaremålene fremgår af tabel 8.2. For at bringe koden til at overholde Information Expert er to metoder blevet flyttet. Klassen `Sale` havde tidligere en metode der returnerede subtotalen på en salgslinie. Denne metode er blevet flyttet til `SalesEntry` da det er denne klasse der indeholder viden om subtotalen, dvs. viden om pris og mængde en vare er solgt i. Yderligere er metoden til at printe en kvittering flyttet fra den dedikerede klasse `ReceiptLayout`, med subklasse, til klassen `Sale`, da `Sale` indeholder data om hvad der er solgt. Klasserne `ReceiptLayout` og `StdReceiptLayout` er efterfølgende blevet slettet. Resultatet viser at der er en øget koblingen (forringelse) for `Sale`, og en øget kohæsion (forbedring) for `SalesEntry`. Udslaget i kohæsionen skyldes at der blev tilføjet en metode til klassen `SalesEntry`, som benytter to af attributterne fra klassen. Da kohæsionsmålet er baseret på brugen af attributter i metoderne i klassen, giver det en øget kohæsion i følge $LCOM*$. Koblingen i `Sale` er blevet højere og dermed dårligere end den var før systemet blev ændret. Forøgelsen skyldes at ansvarsområder som tidligere var tildelt den dedikerede klasse `StdReceiptLayout` nu er blevet flyttet ind i `Sale`.

Klasse	Mål	*Før* Inf. Expert	*Efter* Inf. Expert
`Sale`	CBO	2	5
	$LCOM*$	0,5	0,5
	$Kendskab$	4	8
`SalesEntry`	CBO	1	1
	$LCOM*$	0,5	0,34
	$Kendskab$	1	1
`ExpPos`	CBO	7	5
	$LCOM$	0	*slettet*
	$Kendskab$	9	*slettet*
`StdReceiptLayout`	CBO	6	*slettet*
	$LCOM$	0	*slettet*
	$Kendskab$	9	*slettet*
`ReceiptLayout`	CBO	2	*slettet*
	$LCOM$	0	*slettet*
	$Kendskab$	9	*slettet*
Hele systemet	$\sum(CBO)$	12	9
	$gns(CBO)$	0,86	0,75
	$gns(LCOM*)$	0,16	0,18
	$Kendskab$	29	20

Tabel 8.2: Mål før og efter 'Information Expert' i *ExpPos*.

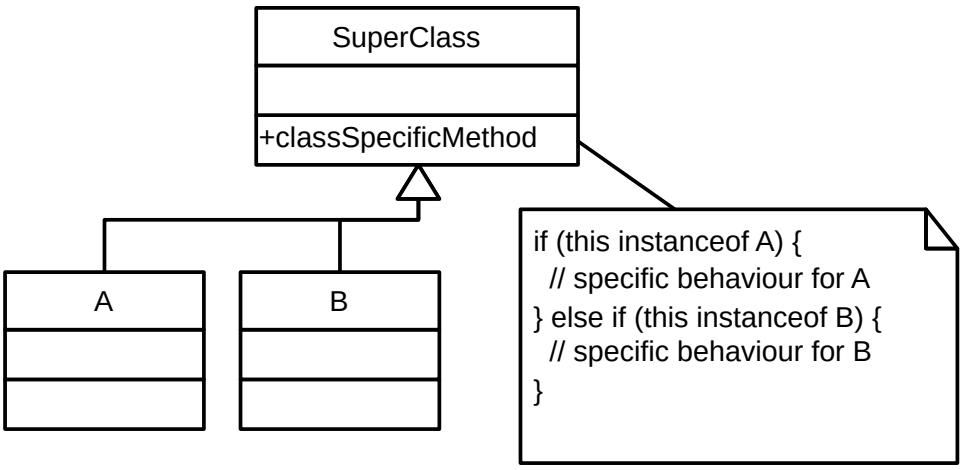
For systemet samlet set, er koblingen blevet bedre efter Information Expert bliver overholdt, da det totale antal koblinger er blevet mindre. Kohæsionen i systemet er i gennemsnit blevet en anelse forringet. Dette skyldes at de klasser der blev overflødiggjort efter omskrivningen havde en kohæsion på 0 (nul). Denne høje kohæsion skyldtes at klasserne indeholdt en enkelt instansvariabel, som blev tilgået fra samtlige metoder i klassen. Dette giver efter $LCOM*$ perfekt kohæsion.

Kendskabet i systemet går ned i det samlede system, men op i klassen `Sale`. Forøgelsen af kendskabet i `Sale` skyldes også at ansvaret fra `StdReceiptLayout` er flyttet til `Sale`. Yderligere betyder sletningen af superklassen `ReceiptLayout`, at det samlede kendskab i systemet bliver mindre.

Det ses at overholdelse af Information Expert har indflydelse på softwaremålene. Både koblingen og kohæsionen blev påvirket efter implementation af Information Expert. Koblingen blev større for en enkelt klasse, men mindre for systemet totalt set. Kohæsionen blev mindre for den ene af de påvirkede klasser, men større i gennemsnit. Den, marginalt dårligere, gennemsnitlige, kohæsion skyldes at to klasse med perfekt kohæsion blev slettet fra systemet. Som beskrevet i kapitel 5, svarer disse resultater til hvad der blev forventet.

8.4.2 Polymorphism

Principperne for polymorfisme kan brydes på forskellig måde. Udgangspunktet for brugen af polymorfi er klassespecifik opførsel. Polymorfismeprincippet siger at klassespecifikke operationer skal ligge i klassen selv. Én måde at bryde princippet på, er at lade superklassen vælge mellem forskellige implementationer afhængig af den aktuelle type klassen har.



Figur 8.1: Brud på princippet om polymorfi.

Figur 8.1 illustrerer en af måderne princippet for polymorfi kan brydes. Den klassespecifikke operation udføres i superklassen afhængigt af typen på den aktuelle klasse. Som tidligere nævnt er dette et stort problem i forhold til udvidelser, da enhver tilføjelse af nedarvede klasser kræver ændringer i superklassen. Princippet kan også brydes ved at konstruere en hjælpeklasse som indeholder den klassespecifikke metode. Metoden tager en parameter som er superklassen for en mængde klasser, og udføre funktioner afhængig af den faktiske type instansen af parameteren har.

Ovenstående brud på polymorfi er implementeret og kan ses i bilag A.6. I tabel 8.3 ses mål for de to forskellige måder at bryde princippet, samt for en ækvivalent implementation der overholder princippet.

I eksemplet er der intet udslag at finde i kohæsionsmålet. En grund til dette er, at systemet er for småt. Der er ingen forskel i de samlede koblinger i systemet. Koblingerne er blot fordelt forskelligt i de tre implementationer. Det skal bemærkes at udslagene i koblingsmålet er afhængigt af hvad de forskellige klassespecifikke metoder indeholder. Hvis de klassespecifikke metoder i de tre nedarvede klasser `B1, B2` og `B3` havde indeholdt koblinger til de samme eksterne klasser, ville den samlede kobling være øget i systemet ved at overholde princippet om nedarvning. Grunden til dette er, at én kobling i f.eks. superklassen i stedet bliver implementeret i alle tre nedarvede klasser. Dette vil betyder at `B1, B2` og `B3` alle ville være koblet til den samme klasse

Klasse	Mål	Brud i super-klasse	Brud i hjælpe-klasse	Intet brud
`S` (superklasse)	CBO	3	0	0
	$LCOM*$	0	0	0
	$Kendskab$	6	0	0
`B1` (nedarvet)	CBO	0	0	1
	$LCOM*$	0	0	0
	$Kendskab$	0	0	1
`B2` (nedarvet)	CBO	0	0	1
	$LCOM*$	0	0	0
	$Kendskab$	0	0	1
`B3` (nedarvet)	CBO	0	0	1
	$LCOM*$	0	0	0
	$Kendskab$	0	0	1
`Run` (main)	CBO	1	1	1
	$LCOM*$	0	0	0
	$Kendskab$	5	6	5
`Tool` (hjælpe)	CBO	-	3	-
	$LCOM*$	0	0	0
	$Kendskab$	-	7	-
Hele systemet	$\sum(CBO)$	4	4	4
	$gns(CBO)$	0,80	0,67	0.80
	$gns(LCOM*)$	0	0	0
	$Kendskab$	11	13	8

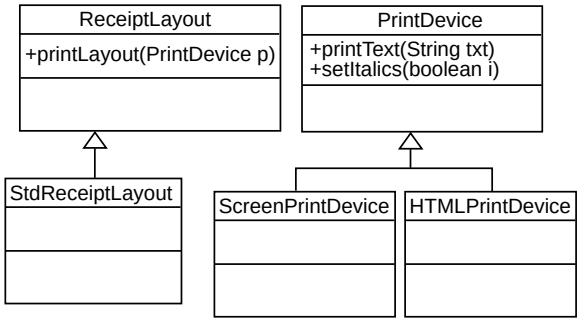
Tabel 8.3: Mål før og efter 'Polymorphism'

som superklassen tidligere var alene om at være koblet til.

Hvis man betragter CBO og polymorfi isoleret, er det derfor et spørgsmål om distribution af koblingerne i systemet. Fordelen ved at bruge polymorfi er således ikke et spørgsmål om at undgå kobling.

Kendskabet i systemet ændrer sig afhængigt af implementationen. Den implementation der overholder princippet om polymorfisme har generelt mindre kendskab til andre klasser end de øvrige to implementationer. Dette hænger sammen med testen af typen på klassen. En test på klassens type med `instanceof` operatoren betyder, at klassen hvori operatoren benyttes har kendskab til alle de typer der testes for.

I ExpPos bliver princippet om polymorfi brudt. Til udskrivning af kvitteringer findes en mængde klasser. Udskrivningen er overordnet opdelt i layout'et af udskriften og den 'enhed' der kan udskrives på. I ExpPos er der implementeret subklasser til begge, se figur 8.2.



Figur 8.2: Struktur for kvitteringsudskrivningsklassernei.

Der er implementeret et enkelt layout, `StdReceiptLayout` og to forskellige 'enheder', `ScreenPrintDevice` og `HTMLPrintDevice`. Layout'et bestemmer hvordan udskriften ser ud og hvilke data der bliver udskrevet, mens 'enheden' definerer forskellige måder at udskrive på. Man kan forestille sig forskellige typer af printere har forskellige kommandoer for fed, kursiv etc. `ScreenPrintDevice` og `HTMLPrintDevice` har netop forskellige måder at håndtere disse tekstegenskaber på.

I implementationen der ikke bruger polymorfi, er `PrintDevice` reduceret til en type der testes på i udskrivningsrutinerne i `StdReceiptLayout`'s printmetode. I printmetoden udskrives forskellige formatteringstegn afhængigt af den type af `PrintDevice` der gives til metoden. Kildekoden til de ændrede klasser kan ses i bilag A.3.

I implementationen der bruger polymorfi er udskrivningen af tekst og formatteringstegn flyttet ned i de forskellige `PrintDevice`'s, og `printLayout`-metoderne kalder metoder i det aktuelle `PrintDevice` i stedet for at teste på dets type.

Klasse	Mål	*Før* Polymorphism	*Efter* Polymorphism
`StdReceiptLayout`	CBO	6	6
	$LCOM*$	0,0	0,0
	$Kendskab$	9	6
`ScreenPrintDevice`	CBO	0	1
	$LCOM*$	0,0	0,0
	$Kendskab$	0	2
`HTMLPrintDevice`	CBO	0	1
	$LCOM*$	0,0	0,0
	$Kendskab$	0	2
Hele systemet	$\sum(CBO)$	12	14
	$gns(CBO)$	0,86	1,00
	$gns(LCOM*)$	0,16	0,16
	$Kendskab$	29	27

Tabel 8.4: Mål før og efter 'Polymorphism' *ExpPos*.

Eksperimentet med *ExpPos* gav resultater som de ses i tabel 8.4. Som det ses blev koblingen en anelse større efter implementationen af polymorfi. Dette skyldes at brugen af `System.out` (som er af typen `PrintStream`), forsvinder fra superklassen `StdReceiptLayout`, men til gengæld bliver erstattet af en kobling til `PrintDevice`. Føromtalte kobling til `PrintStream` bliver tilføjet i både `ScreenPrintDevice` og `HTMLPrintDevice`. Derved forbliver koblingen i `StdReceiptLayout` konstant, mens koblingen for de nedarvede klasser `ScreenPrintDevice` og `HTMLPrintDevice` begge forøges.

Havde de nedarvede printklasser faktisk gjort brug af forskellige udskrivningsenheder i stedet for begge at være koblet til `PrintStream`, var koblingen blevet lidt mindre i `StdReceiptLayout`. Igen ville de eksisterende koblinger bliver fordelt anderledes i systemet.

Det samlede kendskab i systemet går marginalt ned. Kendskabet i `StdReceiptLayout` går ned da klassen ikke længere behøver at kende til `HTMLDevice` og `ScreenPrintDevice`. Det øgede kendskab i de to nedarvede klasser `HTMLDevice` og `ScreenPrintDevice` skyldes at begge disse klasser efter omskrivningen har kendskab til `PrintStream` fra `System.out`.

8.4.3 Law of Demeter

Argumentet for at overholde Law of Demeter er at dette princip giver en lavere kobling i systemet. Et ekstremt tilfælde af brud på Law of Demeter bør derfor vise sig i stor grad i koblingsmålet.

Et ekstremt tilfælde af brud på Law of Demeter kan ses i bilag A.7. I hovedklassen `ExtrLod` findes en metode indeholdende en længere beskedkæde. Omskrivningen af koden så Law of Demeter fjernes betyder at denne besked-

kæde forsvinder. Forsøget gennemføres både hvor der overføres parametre til de enkelte metodekald og hvor der ikke gør. I første eksempel er alle metoder parameterløse.

Klasse	Mål	*Før* LoD	*Efter* LoD
`ExtrLod`	CBO	4	1
	$LCOM*$	0,0	0,0
	$Kendskab$	5	2
`A,B,C`	CBO	0	1
	$LCOM*$	0,0	0,0
	$Kendskab$	1	1
`D`	CBO	1	1
	$LCOM*$	0,0	0,0
	$Kendskab$	1	1
Hele systemet	$\sum(CBO)$	5	5
	$gns(CBO)$	1,0	1,0
	$gns(LCOM*)$	0,0	0,0
	$Kendskab$	9	6

Tabel 8.5: Mål før og efter Law of Demeter (*uden* brug af parametre)

Softwaremålene anvendes, og i tabel 8.5 ses resultatet af målingen af eksperimentet hvor ingen parametre bliver overført i beskedkæden. Som det ses er koblingen samlet set ikke ændret. Igen er det et spørgsmål om at koblingerne blot er blevet distribueret anderledes i systemet. Ligesom det var tilfældet for polymorfi, betyder Law of Demeter at koblingerne bliver fordelt mellem klasserne i stedet for at være samlet i en klasse.

Kendskabet i klasserne bliver lidt mindre. Dette skyldes at klassen der indeholdt beskedkæden ikke længere kender til alle typerne i beskedkæden, hvilket er samme årsag som for nedgangen i koblingen. Igen ses det at overholdelse af Law of Demeter betyder at der findes mindre kendskab i klasserne.

I eksperimentet hvor der blev overført parametre gennem beskedkæden, bliver den samlede CBO heller ikke påvirket. Igen er det kun fordelingen af koblingerne der ændrer sig. Til gengæld viser det sig at kendskabet i klasserne bliver større. Det hænger sammen med, at de paramtre der bliver sendt ned gennem beskedkæden kommer til at påvirke kendskabet for alle de klasser der indgår i beskedkæden.

Law of Demeter bliver brudt i systemet ExpPos. Klassen `Receipt` bryder med princippet i to tilfælde. I det ene for at finde en kundes navn og adresse, og i det andet for at finde navnet på en given solgt vare. Omskrivningen kræver ændringer i tre klasser, `StdReceiptLayout`, `Sale` og `SalesEntry`. Disse tre omskrevne klasser kan ses i bilag A.4. Som beskrevet ved gennemgangen af princippet, er det forventet at koblingen ville falde mens kohæsionen ville

Klasse	Mål	*Før* LoD	*Efter* LoD
`ExtrLod`	CBO	4	1
	$LCOM*$	0,0	0,0
	$Kendskab$	9	6
`A`	CBO	1	2
	$LCOM*$	0,0	0,0
	$Kendskab$	2	5
`B`	CBO	1	2
	$LCOM*$	0,0	0,0
	$Kendskab$	2	4
`C`	CBO	1	2
	$LCOM*$	0,0	0,0
	$Kendskab$	2	3
`D`	CBO	2	2
	$LCOM*$	0,0	0,0
	$Kendskab$	2	2
`ParmA..ParmD`	CBO	0	0
	$LCOM*$	0,0	0,0
	$Kendskab$	0	0
Hele systemet	$\sum(CBO)$	9	9
	$gns(CBO)$	1,00	1,00
	$gns(LCOM*)$	0,0	0,0
	$Kendskab$	17	20

Tabel 8.6: Mål før og efter Law of Demeter (*med* brug af parametre)

uændret eller måske lidt højere. I tabel 8.7 vises resultaterne af målene før og efter Law of Demeter overholdes.

Klasse	Mål	*Før* LoD	*Efter* Lod
`StdReceiptLayout`	CBO	6	2
	$LCOM*$	0,00	0,00
	$Kendskab$	9	5
`Sale`	CBO	2	3
	$LCOM*$	0,50	0,50
	$Kendskab$	4	5
`SalesEntry`	CBO	1	1
	$LCOM*$	0,50	0,50
	$Kendskab$	1	2
Hele systemet	$\sum(CBO)$	12	9
	$gns(CBO)$	0,86	0,64
	$gns(LCOM*)$	0,16	0,16
	$Kendskab$	29	27

Tabel 8.7: Mål før og efter LoD på det *ExpPos*.

Som det ses i tabel 8.7 sker der en ændring i målene når Law of Demter overholdes. Klassen `StdReceiptLayout` som brød med Law of Demeter får en væsentlig lavere kobling med det omkringliggende system. `Sale`, som nu har fået tildelt ekstra ansvar, har fået øget sin kobling med én.

Kohæsionen bliver ikke påvirket af ændringerne. Måske skyldes dette at systemet er for småt. Det var forventningen at klasserne fik øget sin kohæsion.

8.5 Samlet diskussion af resultater

I dette afsnit sammenholdes resultaterne fra eksperimentet. De evaluerede systemer er ikke særligt store, så det er svært at sige noget helt entydigt om hvordan softwaremålene ændrer sig når designprincipperne overholdes.

8.5.1 Kohæsion

Generelt har systemerne været for små til at det er muligt at sige noget om kohæsionsmålet $LCOM*$. I flere af eksperimenterne var $LCOM*$ nul. Yderligere skal man ved brug af $LCOM*$, eller de gennemgåede kohæsionsmål generelt, være opmærksom på følgende problem: Når der i en klasse findes attributter som det omkringliggende system skal kunne bruge eller modificere, er det god skik at implementere *getter* og *setter* metoder. Disse metoder vil typisk kun tilgå en enkelt variabel. Resultatet af dette vil være at kohæsionen i klassen vil blive lavere efter $LCOM*$. Problemet kunne delvist løses ved at ekskludere metoder fra en klasse som er navngivet efter konventionen

i JavaBeans. Konventionen siger at en metoder der henter en attribut `X` skal navngives `getX()`, og tilsvarende for *setter* metoden `setX(X)`. Der er dog probeblemer ved at udelukke metoder ud fra dette kriterie: For det første kan det ikke garanteres at denne konvention overholdes. For det andet giver det mening at navngive en metode som `getX`, selvom `X` er en beregnet værdi. I sidstnævnte tilfælde vil metoden bidrage positivt til kohæsionen, hvis beregningen er baseret på instansvariable fra klassen selv.

De dele af eksperimentet der viste en ændring i $LCOM*$, viste at kohæsionen var blevet bedre i systemet efter brugen af designprincippet. Sammen med argumenterne givet i kapitel 5 menes kohæsionen stadig at blive forbedret af brugen af designprincipperne.

8.5.2 Kobling

Eksperimenterne viste udslag i koblingsmålet begge veje. I Information Expert blev koblingen tydeligt reduceret. I et eksperiment blev koblingen større, mens tre eksperimenter ikke viste udslag i den samlede kobling i systemet.

I eksperimentet der viste en øget kobling (Polymorfi i ExpPos afsnit 8.4.2) skyldtes forøgelsen af hver af de nedarvede klasser fik de koblinger superklassen havde. I et mere realistisk system vil subklasserne formentlig være mere forskellige og derfor have koblinger til forskellige andre klasser. Det vil betyde at en omskrivning af koden fra brud på polymorfiprincippet til overholdelse af samme, vil give en fordeling af superklassens koblinger mellem subklasserne.

Generelt ses det af eksperimentet, at de koblinger der findes i systemet før omskrivningen, bliver fordelt blandt klasserne i systemet, så koblingerne bliver mere jævnt distribueret. Fra at have få klasser med mange koblinger, giver omskrivningerne systemer med flere klasser med få koblinger. Da koblinger betragtes på klasseniveau, er denne ændring en forbedring af systemet.

Det konkluderes at designprincipperne generelt har en positiv indflydelse på koblingerne i systemet. Dels på grund af en samlet nedgang i koblingerne, men også på grund af den bedre distribution af koblingerne.

8.5.3 Kendskab

Bortset fra et enkelt eksperiment (Law of Demeter *med* parametre se afsnit 8.4.3), viste det sig meget klart at kendskabet mellem klasser blev reduceret når designprincipperne bliver overholdt. Grunden til forøgelsen i dette enestående eksempel er, at parametrenes typer bliver kendt af alle klasserne der findes i beskedkæden.

Det konkluderes at designprincipperne generelt har en positiv indflydelse på kendskabet mellem klasser, dvs. kendskabet mellem klasser bliver mindre. Det forventes derfor at kendskabet kan bruges som en indikator på designproblemer. En grundig undersøgelse af dette ligger udenfor specialets rammer.

8.6 Abstraktionsfaktorene

I dette afsnit udføres eksperimenter med abstraktionsfaktorene: Kendskabsabstraktionsfaktoren (KAF) og koblingsabstraktionsfaktoren (CAF). Det er ønsket at se hvordan disse mål ser ud i virkelige systemer.

Eksperimentet foregår på en anden måde end de øvrige eksperimenter. Et eksperiment som dem gennemført for de øvrige mål giver ikke mening for abstraktionsfaktorene, da en erstatning af f.eks. kobling til en konkret klasse med en kobling til et interface, pr. definition vil vises sig som en ændring i abstraktionsfaktorene.

I stedet eksperimenteres der med abstraktionsmålene i to større systemer. Disse systemer evalueres med målene og sammenlignes derefter. Det ene system er udviklingsmiljøet *Eclipse* som er et open-source projekt med IBM som hovedbidragyder. Eclipse er et system på godt 600.000 linier javakode, og indeholder over 7.500 javaklasser. Det andet system er noget mindre. Systemet er et kommercielt *point-of-sale*-system, herefter blot kaldet POS. Dette system er på ca. 350 klasser og indeholder omkring 25.000 linier kode.

Da der findes operativspecifikke dele af Eclipse, som skal vælges ved oversættelsestidspunktet, er der dele af implementationen der er blevet fjernet inden oversættelse og evaluering. De ovenfor opgivne antal linier og klasser er efter denne mængde kode er blevet fjernet. Det anslås at der i alt er blevet fjernet omkring 100 klasser. Det menes ikke at dette forhold har nogen særlig indflydelse på resultaterne.

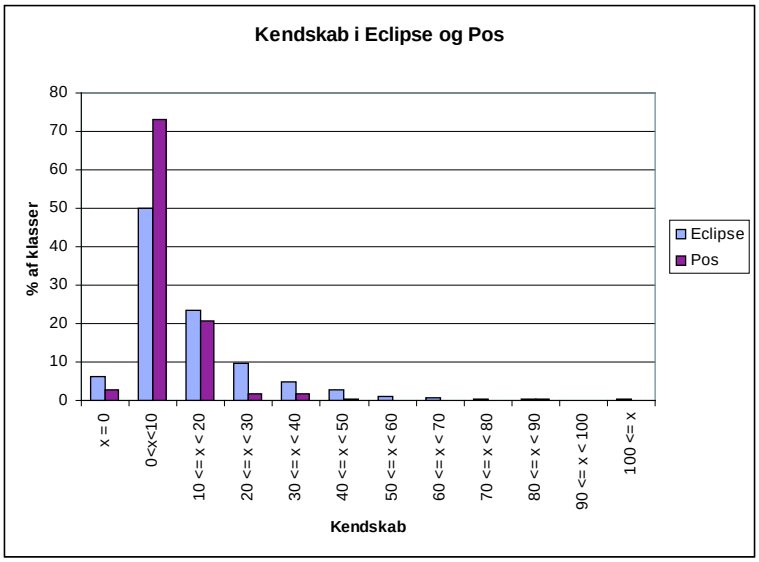
For at få en forståelse af de data der ligger til grund for faktorene, bliver kendskab og koblingen også gennemgået her. Kendskabsmålene i de to systemer sammenholdes og diskuteres.

Det forventes at abstraktionsmålene er højere i Eclipse end de er i POS. Dette hænger blandt andet sammen med, at Eclipse er udviklet så det er let at udvidde med ny funktionalitet. Eclipse indeholder en reletivt lille kerne og er ellers baseret på plug-ins. Denne struktur forventes i høj grad at være baseret på interfaces, for at gøre systemet så fleksibelt som muligt. Plug-in strukturen er ikke i samme grad gennemført i POS.

8.6.1 Kendskabet i systemerne

Kendskabet er defineret som alle de klasser en klasse `A` er koblet til, eller kender til gennem parametre, oprettelse af objekter, typecasts osv. Kendskabet

i Eclipse og POS fremgår af grafen i figur 8.3. Data som ligger til grund for grafen kan ses i appendiks B.6 og B.7.

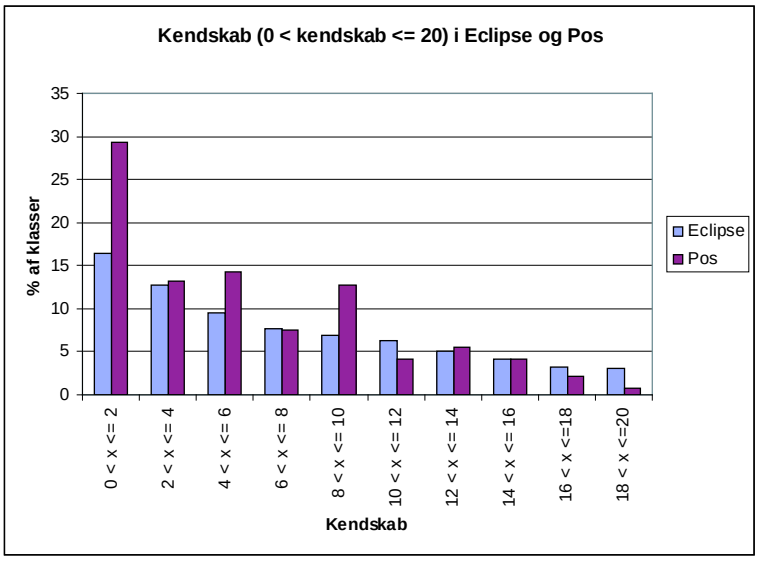


Figur 8.3: Kendskab for Eclipse og Pos.

Figur 8.3 viser kendskabet i systemerne. Som det ses har de fleste klasser kendskab til mellem nul og tyve klasser og interfaces. Dette udsnit vises på en finere skale i figur 8.4. Denne figur viser antallet af kendskaber mellem én og tyve. Som det ses er der generelt fald i mængden af klasser og interfaces jo større kendskabet er i klasser og interfaces. Det var forventet mængden af klasser ville blive mindre, jo større kendskabet blev.

Generelt ses det i figur 8.4, at kendskabet i klasserne og interfacene i POS er højere end kendskabet i Eclipse. Eclipse har enkelte klasser og interfaces som har et højt kendskab. Samlet har godt 20% af klasserne et kendskab som er større end 30. Det tilsvarende tal for POS er godt 3,6% (se data i bilag B). Disse enkelte klasser med højt kendskab er ikke nødvendigvis dårligt designet. Eclipse gør f.eks. brug af et *Visitor*-mønster [GB04], til operationer på parsede syntakstræer. Denne visitor har kendskab til mange klasser, da visitorens metoder skal kunne kaldes med de mange forskellige typer af noder i syntakstræet. Dette betyder et stort kendskab fra klassen til det omkringliggende system. Til gengæld er det begrænset til ret få klasser.

Ved sammenligningen af kendskabet i Eclipse og POS, skal man være opmærksom på størrelsen af systemerne. Eclipse indeholder godt tyve gange flere klasser end POS og dette kan være årsagen til at kendskabet er større i Eclipse.



Figur 8.4: Kendskab (0 < kendskab <= 20) for Eclipse og Pos.

8.6.2 Koblingen i systemerne

Koblingen er som bekendt en delmængde af kendskabet i systemet. I dette afsnit gennemgås resultater for koblingen i de to systemer.

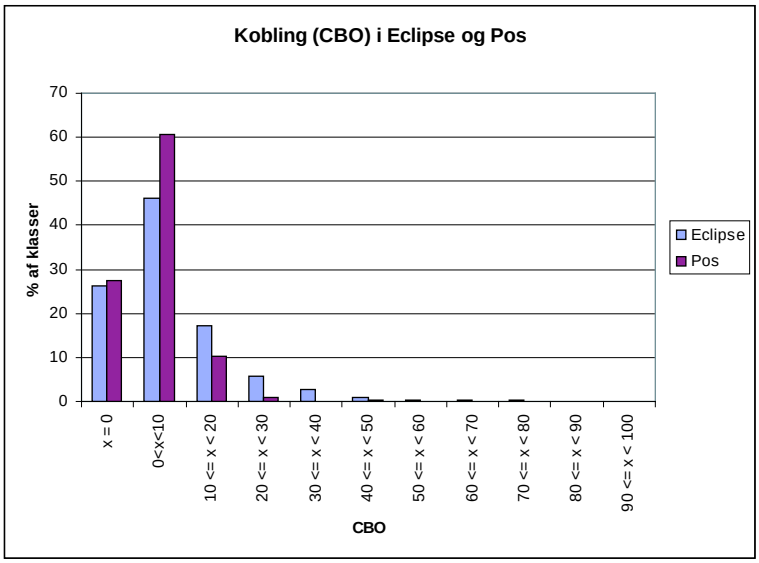
Koblingen i systemerne sammenlignes i figur 8.5. Det, ligesom for kendskabet, at de fleste af klasserne i systemet har kobling der ligger under tyve. Derfor er disse også præsenteret på en finere skala i figur 8.6.

Som det ses i figur 8.6, har Eclipse et pænt faldende antal af klasser med koblinger, jo større koblingsmålet bliver. I POS bliver mængden af klasser med kobling større det første stykke. Der er, som det ses, en større mængde klasser med kobling på fem og seks, end der er med en kobling på én og to. Det må konkluderes at koblingen i POS er værre end Eclipse, og dermed mere problematisk at vedligeholde og genbruge dele af. Dette er som forventet.

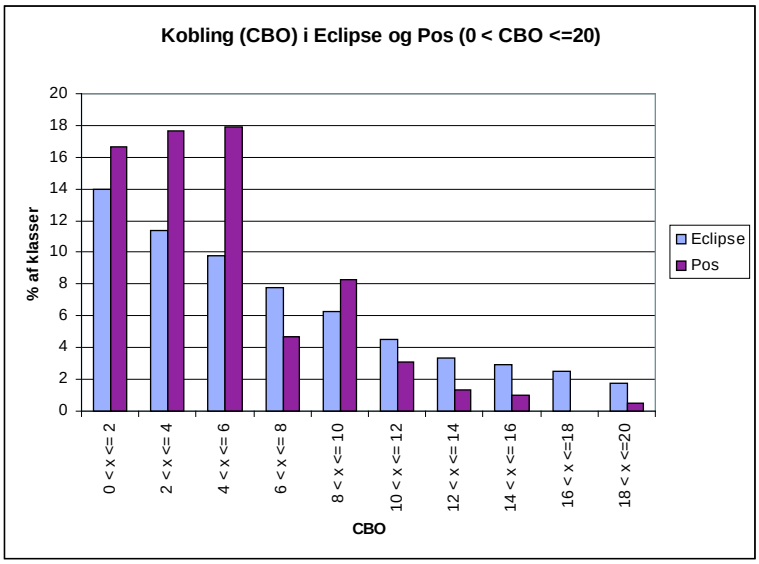
8.6.3 Kendskabsabstraktionsfaktoren

Vi har set hvordan kendskabet ser ud i Eclipse og POS. I dette afsnit sammenlignes kendskabsabstrationsfaktoren KAF, altså andelen af kendskabet der er til interfaces i stedet for til klasser.

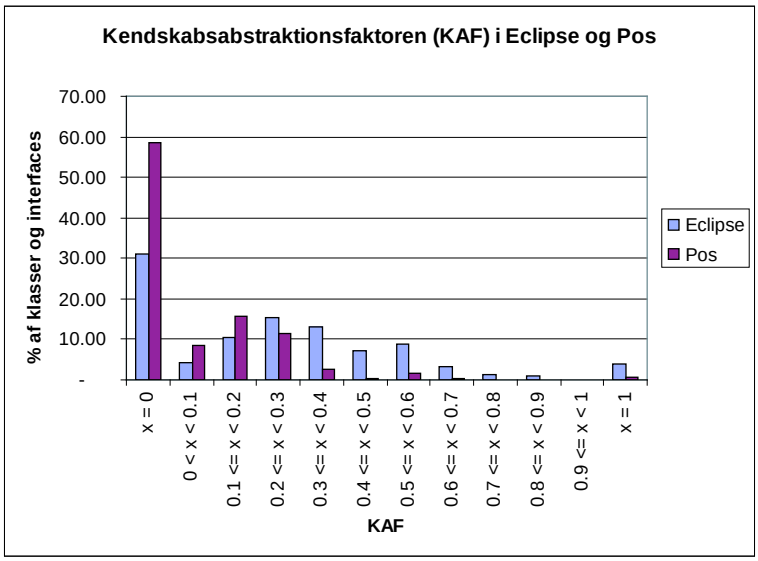
Grafen i figur 8.7 viser kendskabsabstraktionsfaktoren for de to systemer. Som det ses, er der en markant større del af klasser og interfaces i POS, i forhold til Eclipse, der har $KAF = 0$, og dermed slet ikke kender til interfaces. Dette kan være et udtryk for manglende opmærksomhed om interfaces i det hele taget, under udvliklingen. Klasserne er problematiske i forbindelse med vedligeholdelse og genbrug, da det er svært at skifte de dele ud, som



Figur 8.5: Koblingen CBO for Eclipse og Pos.



Figur 8.6: Koblingen ($0 < \text{CBO} <= 20$) for Eclipse og Pos.



Figur 8.7: Kendskabsabstraktionsfaktoren for Eclipse og Pos.

klasserne er afhængige af.

Generelt ses det i figur 8.7 at Eclipse har et markant større antal klasser som mere intensivt har kendskab til interfaces i forhold til kendskabet totalt. Dette betyder at klasserne lettere kan genbruges da delene som klasserne har kendskab til er interfaces som let kan udskiftes.

Fra data ses det at Eclipse tilbyder en større fleksibilitet end POS, og i forhold til genbrug og veligeholdelse af koden er af bedre kvalitet.

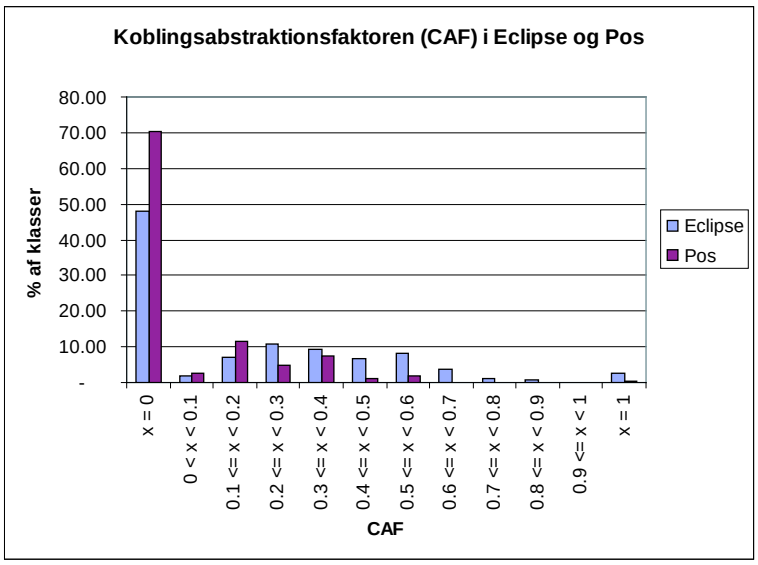
8.6.4 Koblingsabstraktionsfaktoren

Koblingsabstraktionsfaktoren CAF viser samme tendens som kendskabsabstraktionsfaktoren. Det tyder derfor på at klasserne i høj grad er koblet til de klasser og interfaces de har kendskab til.

I figur 8.8 ses koblingsabstraktionsfaktoren for systemerne. Det ses at en markant større del af systemet i POS ikke er koblet til interfaces ($CAF = 0$), i forhold til hvordan dette ser ud i Eclipse. Det ses at Eclipse, stort set, er ene om at have klasser med meget høj koblingsabstraktionsfaktor, altså klasser som i høj grad er koblet til interfaces i stedet for til klasser. Igen bekræftes forventningen om at Eclipse er mere fleksibelt designet end POS.

8.7 Sammenfatning af abstraktionsfaktorene

Begge abstraktionsfaktorer viste sig højere i systemet Eclipse end i POS. Dette svarede helt til forventningerne, fordi Eclipse i meget høj grad er designet til at udvidet, og bortset fra en relativt lille kerne, er baseret på plug-ins



Figur 8.8: Koblingssabstraktionsfaktoren for Eclipse og Pos.

[GB04]. Det menes at abstraktionsfaktorene viser graden af fleksibilitet i et system.

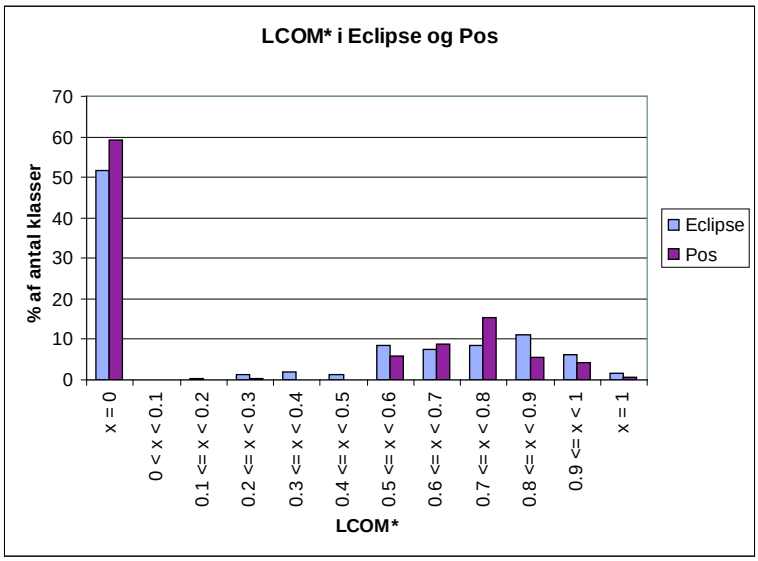
8.8 Kohæsionen

Der blev opsamlet data om kohæsionen i systemerne Eclipse og POS. Disse data vises i figur 8.9.

Forventningen var, at Eclipse generelt indeholder høj grad af kvalitet, og at dette generelt ville vise sig ved sammenligning med POS. Som det ses i figur 8.9 følges de to systemer nogenlunde ad, når man sammenligner $LCOM*$. Det kan derfor ikke konkluderes alene ved hjælp af $LCOM*$, at kvaliteten i Eclipse skulle være større end kvaliteten i POS.

Kun en analyse af de enkelte klasser som har høj $LCOM*$-værdi, kan fortælle om klasserne faktisk bør opdele. Som tidligere nævnt kan en klasse betragtes som værende kohæsiv af andre årsager end at dens metoder bruger attributter fra klassen.

Både Eclipe og POS har et brugerinterface som samler en mængde komponenter som skal præsenteres for brugeren. Disse klasser vil sandsynligvis have lav kohæsion, da komponenterne ikke behøver at være relateret gennem de data de præsenterer eller bruger, og dermed ikke bidrager positivt til kohæsionsmålet $LCOM*$. Det er meget sandsynligt at nogle af de klasser har en høj $LCOM*$-værdi er kohæsive i et andet perspektiv.



Figur 8.9: Kohæsionen LCOM* for Eclipse og Pos.

8.9 Sammenfatning

Målet med dette kapitel var at se om der var en sammenhæng at finde mellem designprincipper og softwaremål, samt at afprøve kendskabsabstraktionsfaktoren og koblingsabstraktionsfaktoren. For at nå dette blev en mængde kildkode udviklet som brød med nogle udvalgte designprincipper. Principperne blev et for et implementeret i det oprindelige system og evalueret med målene efter og før brud med principperne. Yderligere blev abstraktionsfaktorene afprøvet i et mindre eller middelstort kommercielt system, og i et stort system.

Resultatet er, at der er ændringer at spore i målene, når designprincipperne brydes. Dog har eksperimentsystemerne været for små til at man kan sige særligt meget om kohæsionen.

- De dele der viste en ændring i $LCOM*$, viste at kohæsionen i systemet blev højere (bedre) når principperne blev overholdt.
- Der bør måske tages hensyn til *getter* og *setter* metoder ved beregning af kohæsionen.
- Koblingen i systemet blev generelt lavere i systemet efter designprincipperne blev overholdt, så det konkluderes at designprincipperne har en positiv indflydelse på koblinger i systemet.
- Bortset fra eksemplet med Law of Demeter med parametre, blev kendskabet i systemet lavere. Jo lavere kendskab i klasserne, jo mindre skrø-

belige er klasserne overfor ændringer, så det konkluderes at systemerne bliver mindre skrøbelige når designprincipperne overholdes.

Samlet giver det derfor systemer af højere kvalitet ud fra de givne softwaremål, når designprincipperne overholdes.

Afprøvningen af abstraktionsfaktorene viste:

- Klasserne i Eclipse har mere kendskab til interfaces i forhold til det samlede kendskab, end klasserne i POS har. Dette betyder at Eclipse er mere fleksibelt med hensyn til vedligeholdelse og genbrug.
- Klasserne i Eclipse har en større koblingsabstraktionsfaktor end POS. Dette er igen en indikation af at Eclipse er mere fleksibelt designet end POS.
- Abstraktionsfaktorene på de evaluerede systemer viser nogenlunde det samme. Det menes at kendskabsfaktoren vil være mere anvendelig når kun delsystemer evalueres.

Kapitel 9

Konklusion og fremtidigt arbejde

I dette speciale er design af objektorienteret software, som imødekommer genbrug og vedligeholdelse blevet studeret. Det overordnede mål har været, at beskrive elementer af godt design, og at afgøre om nogle af disse elementer lader sig identificere automatisk af statiske softwaremål.

9.1 Konklusion

Konklusionen er, at der er en sammenhæng mellem brugen af designprincipper, som menes at give et godt design, og nogle af softwaremålene. De designprincipper der blev eksperimenteret med viste at koblinger i klasser blev færre. Man kan evaluere software automatisk, som kan identificere klasser der potentielt er dårligt designet.

Yderligere blev der konstrueret et abstraktionsmål som kan udtrykke graden af brug af interfaces i Javakildekode. Det menes at målet kan bruges til at sige noget om et systems generelle fleksibilitet. Eksperimentet der gennemførtes i specialet er dog for lille til at der kan siges noget eksakt om målets anvendelse. Det tyder dog på at målet kan bruges til at sammenligne fleksibiliteten i systemer.

Konklusionen er opdelt i de delproblemer specialet søgte at finde svar på.

9.1.1 Genbrug

Der skelnes i litteraturen mellem systematisk og usystematisk genbrug. Den systematiske genbrug er karakteriseret ved, at den genbrugte komponent ikke må modificeres, skal hentes fra et *repository* og skal indgå direkte i det nyudviklede system, eller inddirekte gennem anvendelse i værktøjer som systemet gør brug af. Denne definition af genbrug var for snæver til at den

kunne bruges i specialet, da det var ønsket også at beskrive design, som er vedligeholdelsesvenligt.

Strategier for genbrug inkluderer derfor f.eks. kopiering af kildekode. Denne strategi forkastes dog af vedligeholdelseshensyn. Genbrug kan opnås gennem kald til allerede implementerede dele af systemet, igennem komponenter eller frameworks.

Yderligere er der nogle mekanismer i det objektorienterede paradigme der kan benyttes til at opnå genbrug: Nedarvning giver mulighed for at arve implementation der findes i en klasses superklasse. Polymorfi giver mulighed for at lade et delsystem arbejde med forskellige typer uden at kende den konkrete type.

9.1.2 Softwaremål

En mængde forskellige softwaremål blev gennemgået. En mængde mål der blev defineret i det strukturerede paradigme blev beskrevet. Nogle af disse mål eksisterer som dele af nogle objektorienterede mål. McCabes kompleksitetsmål kan f.eks. benyttes som vægt i Chidamber & Kemerers mål: *Weighted Methods Per Class*, WMC. Målet fan-out, kan genfindes i Chidamber & Kemerers koblingsmål: *Coupling Between Object Classes*, CBO.

To komplette *metric suites* blev gennemgået. Chidamber & Kemerers samling af mål og *Metrics of Object-Oriented Design - MOOD*, af e Abreu et al. Chidamber & Kemerers mål er defineret på klasseniveau, mens e Abreu et al.s mål er defineret på et samlet system. Da det har været ønsket at identificere problematiske områder i et system, blev *MOOD* fra e Abreu et al. ikke behandlet videre.

Gennemgangen af softwaremål afslørede to egenskaber som synes meget relevante for genbrug og vedligeholdelse: Kobling og kohæsion. Kobling er et udtryk for afhængigheder mellem dele af et system, f.eks. mellem klasser, og kohæsion er et udtryk for sammenhæng indenfor et enkelt modul, f.eks. en klasse. Det ønskes at systemer indeholde en lav grad af kobling, og en høj grad af kohæsion. Høje koblinger kan dels betyde, at det kan være svært at isolere delsystemer så de kan tages ud til genbrug, og dels kan de betyde, at ændringer et sted i systemet kan give udslag, eller kræve rettelser et andet sted i systemet. Begge disse egenskaber er problematisk i forbindelse med genbrug og vedligeholdelse.

Lav (dårlig) kohæsion, kan være et udtryk for at en klasse har fået tildelt for mange ansvarsområder. Har en klasse mange ansvarsområder, er den sværere at genbruge og vedligeholde. Dels er klassen sværere at forstå, og dels er meget kohæsive klasser ofte nemmere at genbruge, fordi de har et specifikt ansvar.

Chidamber & Kemerer definerer mål for begge disse begreber. Kohæsionsmålet fra Chidamber & Kemerer, *LCOM*, er blevet udskældt, specielt fordi det ikke giver særligt følsomme værdier når der er høj kohæsion i en klasse. To klasser med, intuitivt set, vidt forskellig kohæsion, kan begge give "perfekt" kohæsion med Chidamber & Kemerers kohæsionsmål. Derfor er der gjort andre forsøg på at opstille mål der udtrykker kohæsionen i en klasse. Selve kohæsionen kan ikke måles. Når der tales om kohæsionsmål, er det egenskaber som man mener siger noget om kohæsion der måles. Ikke kohæsionen selv.

9.1.3 Designprincipper

Tretten designprincipper blev gennemgået og sammenholdt med softwaremålene. Designprincipperne er udtryk for godt design, og er derfor interessante at diskutere sammen med softwaremålene.

Nogle af designprincipperne giver udslag i målene, mens andre ikke gør det. Liskov Substitution Principle siger, at subklasser til en klasse skal opføre sig "som forventet". En subklasse må ikke omdefinere meningen af en del af superklassen. Om dette princip er brudt, lader sig ikke afgøre ved hjælp af nogen af de gennemgåede softwaremål.

Andre designprincipper, som f.eks. *Information Expert*. Vil give udslag i nogle af målene. Princippet siger at et ansvar for en operation skal tildeles den klasse der har data til at udføre operationen. Brud på dette princip vil vise sig i kohæsions- og koblingsmål.

Nogle principper for design af komponenter og frameworks blev beskrevet. Ingen af de gennemgåede mål kan benyttes til evaluering af disse.

Princippet som siger at der skal designes mod et interface, og ikke mod en klasse, bliver ikke afspejlet i nogen af målene. Dette er problematisk da det er et meget vigtigt princip. Overholdes princippet giver det større fleksibilitet i systemet. Det forholder sig sådan, fordi klassen som implementerer et interface, kan udskiftes af en anden klasse, sålænge den nye klasse tilbyder det samme interface.

9.1.4 Mangler ved eksisterende mål

Da kohæsionsmålet forsøger at udtrykke hvor sammenhængende en klasse er, så man kan identificere klasser der bør opsplittes, virker det oplagt at undersøge muligheden for at konstruere det "inverse" mål, og forsøge at identificere klasser der bør smeltes sammen. Et sådan mål ville imidlertid give mange resultater. Betragte f.eks. princippet Model-View-Controller, siger princippet netop at data og præsentation af data skal adskilles. Konstruktionen af et "inverst" kohæsionsmål opgives dermed.

Det blev klart, under gennemgangen af designprincipperne, at princippet om at designe mod et interface ikke blev afspejlet i nogen af softwaremålene. Derfor blev to nye mål konstrueret. Begge mål udtrykker i hvor høj grad et system benytter sig af interfaces. Det ene mål benytter *kendskab* mellem klasser. Kendskabet i en klasse `A` indeholder alle de andre klasser som `A` har kendskab til. Kendskabet kan være gennem en parameter der bliver overført til `A`, brug af Java operationen `instanceof` osv. Kendskabet indeholder også de klasser `A` er koblet til. Kendskabet kan bruges til at evaluere delsystemer, da målet også tager hensyn til klasser der ikke er koblet til. Et delsystem kan f.eks. indeholde en klasse med en metode der returnerer noget til kalderen. Hvis dette "noget", f.eks. klassen `B`, ikke bliver brugt i resten af det delsystem der evalueres, hvilket kan ske når et system f.eks. ikke er færdigt, tælles `B` alligevel med i kendskabsmålet, da metoden der returnerer `B`, og dermed har kendskab til `B`.

Det andet abstraktionsmål er baseret på koblingen, CBO, som defineret af Chidamber & Kemerer, i stedet for kendskabet. Målene er forholdet mellem brugen/kendskabet til interfaces, i forhold til brugen/kendskabet for hele klassen.

9.1.5 Eksperimenter

For at se hvordan nogle af softwaremålene opfører sig, blev der gennemført et eksperiment med målene CBO, $LCOM*$ og kendskabet. Eksperimentet blev udført på de tre designprincipper *information expert*, *Polymorphism* og *Law of Demeter*. Resultatet er, at det var muligt at se forbedringer i koblingsmålet CBO og i kendskabet. Kendskabet blev dog øget i forbindelse med eksperimentet med *Law of Demeter*, når parametre blev sendt igennem beskedkæden.

Eksperimentet viste sig generelt for småt til at der kan siges noget om kohæsionsmålet $LCOM*$.

Abstraktionsfaktorene blev afprøvet i to systemer. Eclipse, som er et system indeholdende over 600.000 linier javakode, og et kommercielt *point-of-sale* system (POS), blev evalueret med målene. Det viste sig at Eclipse har mere kendskab til interfaces i forhold til det samlede kendskab, end POS har. Det samme gør sig gældende når kobling bruges i stedet for kendskab. Dette indikerer at Eclipse er mere fleksibelt end POS. Det indikerer også at Eclipse er mere vedligeholdelsesvenligt og strukturelt giver bedre mulighed for genbrug.

9.2 Fremtidigt arbejde

Arbejdet med emnet har givet inspiration til at undersøge flere aspekter ved softwaremål og designprincipper. Følgende punkter nævner områder der ville

være interessant at undersøge nærmere:

- Det kunne være interessant at undersøge forholdet mellem koblingen og kendskabet. Kan man sige noget om en klasse hvis klassen er koblet til alt den har kendskab til?
- En undersøgelse af sammenhængen mellem kendskabet og koæsionen i en klasse. Man kunne forestille sig at stort kendskab ville være korreleret med lav kohæsion.
- Kobling og kendskab *fra* en klasse er blevet undersøgt. Dette er et udtryk for hvor skrøbelig den enkelte klasse er overfor forandringer. Det modsatte, altså hvormange klasser der har kendskab til den aktuelle klasse bør også undersøges. Dette vil vise hvor stor risiko der vil være ved at ændre i klassen. Jo flere klasser der afhænger af den aktuelle klasse, jo større er risikoen ved at ændre i den (Dette svarer til *fan-in*).
- Et større eksperiment med abstraktionsfaktorene bør gennemføres, da eksperimentet i dette speciale er for lille til at man kan sige noget sikkert om målets anvendelse.

Litteratur

[AL97] Joe Raymond Abounader and David Alex Lamb. A Data Model for Object-Oriented Design Metrics. Technical Report ISSN-0836-0227-1997-409, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1997.

[Ben98] Per Olof Bengtsson. Towards Maintainability Metrics on Software Architecture: An Adaption of Object-Oriented Metrics. In *First Nordic Workshop on Software Architecture (NOSA'98)*, Aug 1998.

[BK95] James M. Bieman and Byung-Kyoo Kang. Cohesion and Reuse in an Object-Oriented System. In *ACM SIGSOFT Symposium on Software Reusability*, pages 259–262, 1995. Tilgængelig på: citeseer.nj.nec.com/bieman95cohesion.html.

[Boo91] Grady Booch. *Object-Oriented Design with Applications*. Benjamin Cummings Publishing Company, 1991.

[CDK98] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer. Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. *IEEE Transactions on Software Engineering*, 24(8), August 1998.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6), June 1994.

[Coc93] A. A. R. Cockburn. The Impact of Object-Orientation on Application Development. *IBM Systems Journal*, 32(3), 1993.

[Cou98] Bernhard Coulange. *Software Reuse*. Springer-Verlag, 1998.

[CS95a] Neville I. Churcher and Martin J. Shepperd. Comments on "A Metrics Suite for Object Oriented Design". *IEEE Transactions on Software Engineering*, 21(3):263–265, March 1995.

[CS95b] Neville I. Churcher and Martin J. Shepperd. Towards a Conceptual Framework for Object Oriented Software Metrics. *ACM SIGSOFT Software Engineering Notes*, 20(2), April 1995.

[CSAD02] Rodrigo E. Caballero and Sr. Steven A. Demurjian. Towards the Formalization of a Reusability Framework for Refactoring. *Lecture Notes in Computer Science*, 2319:293–308, 2002.

[DW99] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML, the Catalysis Approach*. Addison Wesley, 1999.

[eAC94] Fernando Brito e Abreu and Rogério Carapuça. Object-Oriented Software Engineering: Measuring and Controlling the Development Process (Revised version). *4th International Conference on Software Quality*, October 1994. Tilgængelig på: http://www-ctp.di.fct.unl.pt/QUASAR/Resources/Papers/-Metrics/4ICSQ.pdf.

[eAGE95] Fernando Brito e Abreu, Miguel Goulão, and Rita Esteves. Toward the Design Quality Evaluation of Object-Oriented Software Systems (Revised version). *Proceedings of the 5th International Conference on Software Quality*, October 1995.

[eAM96] F. Brito e Abreu and W. Melo. Evaluating the Impact of Object-Oriented Design on Software Quality. In *3rd Int'l S/W Metric Symposium*, march 1996. Tilgængelig på: http://www2.umassd.edu/SWPI/ESEG/3IntSoftMetSymp.pdf.

[ecl] Eclipse: http://www.eclipse.org.

[EMT02] Michael Ezran, Maurizio Morisio, and Colin Tully. *Practical Software Reuse*. Springer-Verlag, 2002.

[Fow99] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 1999.

[GB04] Erich Gamma and Kent Beck. *Contributing to eclipse, Principles, Patterns and Plug-Ins*. Addison Wesley, 2004.

[GHJV95] Eric Gamma, Richard Helm, Ralph Johnsson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addion Wesley, 1995.

[Gro03] Object Management Group. OMG Unified Modeling Language Specification version 1.5, March 2003.

[Hal77] Maurice H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[HK81] Sallie Henry and Dennis Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, SE-7(5), Sep 1981.

[HM96] Martin Hitz and Behzad Montazeri. Chidamber & Kemerer's Metrics Suite: A Measurement Theory Perspective. *IEEE Transactions on Software Engineering*, 22(4), April 1996.

[HS96] Brian Henderson-Sellers. *Object-Oriented Measures. Measures of Complexity.* Prentice Hall, 1996.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer.* Addison Wesley, 2000.

[JFF+02] Dr. Paul Jorgensen, David Fernandez, Al Fischer, Michael Greco, Bradly Hussey, Steven Kuchta, Hong Li, Steven Overkamp, Douglas Rodenberger, and Richard VanderWal. Has the Object-Oriented Paradigm Kept Its Promise?, December 2002. Tilgængelig på: http://www.csis.gvsu.edu/Academics/-MastersFall2002/OOPromisesAndReality.pdf.

[JGJ97a] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse.* Addison Wesley, 1997.

[JGJ97b] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Sofware Reuse, architecture process and organization for business success.* Addison Wesley, 1997.

[Jia00] Xiaoping Jia. *Object-Oriented Software Development Using Java.* Addison Wesley, 2000.

[Joh97] Ralph E. Johnson. Components, Frameworks, Patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pages 10–17, 1997.

[JSt] Jstyle: http://www.mmsindia.com/jstyle.html.

[Lar02] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall PTR, 2 edition, 2002.

[LC] Al Lake and Curtis R. Cook. Use of Factor Analysis to Develop OOP Software Complexity Metrics. Technical Report 94-60-04, Oregon State University Corvallis, Computer Science Department. Tilgængelg på: citeseer.nj.nec.com/lake94use.html.

[LH89] Karl J. Lieberherr and Ian M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, 6(5):38–48, 1989.

[LHR88] Karl J. Lieberherr, Ian Holland, and Arthur J. Riel. Object-Oriented Programming: An Objective Sense of Style. Number 11, pages 323–334, San Diego, CA, September 1988.

[Lis88] Barbara Liskov. Data Abstraction and Hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, 1988.

[Mar00] Robert C. Martin. Design Principles and Design Patterns, 2000. Tilgængelig på: http://www.objectmentor.com-/resources/articles/Principles_and_Patterns.PDF.

[Mat] Mathworld:http://mathworld.wolfram.com/-cyclomaticnumber.html.

[MBF99] Michael Mattsson, Jan Bosch, and Mohammed E. Fayad. Framework Integration - Problems, Causes, Solutions. *Communications of the ACM*, 42(10):81–87, Oct 1999.

[McC76] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(2), Dec 1976.

[McC93] Steve McConnell. *Code Complete*. Microsoft Press, 1993.

[met] Metrics: http://metrics.sourceforge.net.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2 edition, 1997.

[Mor] Sandro Morasca. Software measurement. Tilgængelig på: http://citeseer.nj.nec.com/498984.html.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. *Lecture Notes in Computer Science*, 1445:355–382, 1998.

[Mye78] Glenford J. Myers. *Composite/Structured Design*. Van Nostrand Reinhold Company, 1978.

[PNSAD01] Margaretha W. Price, Donald M. Needham, and Sr. Steven A. Demurjian. Producing Reusable Object-Oriented Components: A Domain-and-Organization-Specific Perspective. In *Proceedings of the 2001 symposium on Software reusability*, pages 41–50. ACM Press, 2001.

[Pou97] Jeffrey S. Poulin. *Measuring Software Reuse*. Addison Wesley, 1997.

[PPK] Wolfgang Pree, Gustav Pomberger, and Franz Kapsner. Framework Component Systems: Concepts, Design Heuristics, and Perspectives. Tilgængelig på: http://www.softwareresearch.net/site/publications/C014.pdf.

[PSAD97] Margaretha W. Price and Sr. Steven A. Demurjian. Analyzing and Measuring Reusability in Object-Oriented Design. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 22–33. ACM Press, 1997.

[Rog01] Paul Rogers. Encapsulation is not Information Hiding. *JavaWorld*, pages 3–30, May 2001.

[Sak88] Markku Sakkinen. Comments on "Law of Demeter" and C++. *SIGPLAN Notices*, 23(12), 1988.

[SB98] Yannis Smaragdakis and Don Batory. Implementing Reusable Object-Oriented Components. In *5th International Conference on Software Reuse*, Victoria, Canada, 1998.

[Sch99] Steffen Schaefer. Design for Maintenance, 1999. OOPSLA 99 Design for Maintanence Workshop.

[SH91] Charles Simonyi and Martin Heller. The Hungarian Revolution. *Byte*, Aug 1991.

[SMC74] W. P. Stevens, G. J. Meyers, and L. L. Constantine. Structured Design. *IBM Systems Journal*, 2(13):115–139, 1974.

[Som92] Ian Sommerville. *Software Engineering*. Addison Wesley, fourth edition, 1992.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 2 edition, 1991.

[SXC99] Dimitris Stavrinoudis, Michalis Xenos, and Dimitris Christodoulakis. Relation between Software Metrics and Maintainability. In *Proceedings of the FESMA99 International Conference, Federation of European Software Measurement Associations, Amsterdam, The Netherlands*, pages 465–476, 1999.

[Und] Understand for java: http://www.scitools.com/uj.html.

[vdL99] Peter van der Linden. *Just Java 2*. Sun Press, 1999.

[Vil97] Antii Viljamaa. Application Frameworks in the Java Environment. Technical Report C-1997-24, University of Helsinki, Department of Computer Science, 1997.

[WBJ90] Rebecca J. Wirfs-Brock and Ralp E. Johnson. Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, 33(9), 1990.

[YC78] Edward Yourdon and Larry L. Constantine. *Structured Design*. Yourdon Press, 1978.

Bilag A

Kildekode til eksperiment

Dette appendiks indeholder en udskrift af den kildekode der blev brugt til eksperimentet med softwaremålene og designprincipperne. Første afsnit indeholder den kode der bryder med samtlige valgte designprincipper, og derfor fungerer som referencekode. I de efterfølgende afsnit vil de relevante ændrede klasser findes.

A.1 Original kode

A.1.1 CandyBar.java

```
1  /*
2   * Created on 26-12-2003
3   *
4   */
5  package violatesall;
6
7  /**
8   * @author Søren Gaardbo
9   *
10  * Violations:
11  * -----------
12  */
13 public class CandyBar extends SalesItem {
14
15   public CandyBar() {
16     super("Candybar", 6, 25);
17   }
18 }
```

A.1.2 CreateTestData.java

```
1  /*
2   * Created on 26-12-2003
3   *
4   */
```

```
5  package violatesall;
6
7  /**
8   * @author SGJ
9   *
10  */
11 public class CreateTestData {
12
13   public static void createSalesItems(Sale sale) {
14     sale.addSalesItem(1, new CandyBar());
15     sale.addSalesItem(2, new Magazine());
16     sale.addSalesItem(1, new Stamp());
17
18   }
19 }
```

A.1.3 Magazine.java

```
1  /*
2   * Created on 26-12-2003
3   *
4   */
5  package violatesall;
6
7  /**
8   * @author Søren Gaardbo
9   *
10  * Violations:
11  * -----------
12  * Should implement an interface 'ISalesItem'.
13  */
14 public class Magazine extends SalesItem {
15
16   public Magazine() {
17     super("Magazine", 32, 25);
18   }
19 }
```

A.1.4 Sale.java

```
1  /*
2   * Created on 26-12-2003
3   *
4   */
5  package violatesall;
6
7  import java.util.Vector;
8
9  /**
10  * @author Søren Gaardbo
11  * Violations:
12  * -----------
13  * Use Interface instead of concrete class (SalesItem, salesEntries...)
```

```
14   * Violates Information Expert. Calculation of subtotals should reside
15   *    in SalesEntry.
16   */
17  public class Sale {
18    private Vector salesEntries;
19    private Customer customer;
20
21    public Sale() {
22      salesEntries = new Vector();
23      customer = new Customer("Søren", "Gaardbo", "Blankavej␣28");
24    }
25
26    public Vector getSalesEntries() {
27      return salesEntries;
28    }
29    public void addSalesItem(int qty, SalesItem salesItem) {
30      SalesEntry salesEntry = new SalesEntry(qty, salesItem);
31      salesEntries.add(salesEntry);
32    }
33    public double getTotal() {
34      double total = 0;
35      for(int i = 0; i < salesEntries.size(); i++) {
36        SalesEntry salesEntry = (SalesEntry) salesEntries.get(i);
37        total += getSubTotal(salesEntry);
38      }
39      return total;
40    }
41    public double getSubTotal(SalesEntry salesEntry) {
42      return (salesEntry.getQty() * salesEntry.getSalesItemPriceInclVat());
43    }
44
45    public Customer getCustomer() {
46      return customer;
47    }
48  }
```

A.1.5 Customer.java

```
1   package violatesall;
2   /*
3    * Created on 26-12-2003
4    *
5    */
6
7   /**
8    * @author Søren Gaardbo
9    *
10   * Violations:
11   * -----------
12   * Should implement an interface 'ICustomer'
13   */
14  public class Customer {
15    private int customerNumber;
```

```
16    private String firstName;
17    private String lastName;
18    private String address;
19
20    public Customer(String fName, String lName, String adr) {
21      firstName = fName;
22      lastName = lName;
23      address = adr;
24      customerNumber = 12;  // replace with 'real' assignment.
25    }
26    public String getName() {
27      return firstName + "␣" + lastName;
28    }
29    public String getAddress() {
30      return address;
31    }
32  }
```

A.1.6 ExpPos.java

```
1   /*
2    * Created on 26-12-2003
3    *
4    * Violations:
5    * -----------
6    * Use interface instead of concrete class : ScreenPrintDevice
7    *
8    */
9   package violatesall;
10
11  /**
12   * @author Søren Gaardbo
13   *
14   * Violations:
15   * -----------
16   * Should, as much as possible, depend on interfaces.
17   *    e.g. 'ISale', 'IPrintDevice' etc.
18   */
19
20  public class ExpPos {
21    Sale sale;
22    ScreenPrintDevice printDevice;
23    HTMLPrintDevice htmlDevice;
24
25    public ExpPos() {
26      sale = new Sale();
27      CreateTestData.createSalesItems(sale);
28
29      printDevice = new ScreenPrintDevice();
30      htmlDevice = new HTMLPrintDevice();
31      ReceiptLayout layout = new StdReceiptLayout(sale);
32      layout.printLayout(printDevice);
33      layout.printLayout(htmlDevice);
```

```
34    }
35    public static void main(String[] args) {
36      new ExpPos();
37    }
38  }
```

A.1.7 HTMLPrintDevice.java

```
1   /*
2    * Created on 01-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package violatesall;
8
9   /**
10   * @author Søren Gaardbo
11   *
12   * Violations:
13   * -----------
14   * Should implement 'IPrintDevice' instead of extension
15   *    of 'PrintDevice'.
16   */
17  public class HTMLPrintDevice extends PrintDevice {
18    /*
19     * Class is empty. The class is reduced to
20     * a 'type'. Polymorphism is not used.
21     */
22
23  }
```

A.1.8 Magazine.java

```
1   /*
2    * Created on 26-12-2003
3    *
4    */
5   package violatesall;
6
7   /**
8    * @author Søren Gaardbo
9    *
10   * Violations:
11   * -----------
12   * Should implement an interface 'ISalesItem'.
13   */
14  public class Magazine extends SalesItem {
15
16    public Magazine() {
17      super("Magazine", 32, 25);
18    }
19  }
```

A.1.9 PrintDevice.java

```
/*
 * Created on 01-01-2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package violatesall;

/**
 * @author Søren Gaardbo
 *
 * Violations:
 * -----------
 * Should be replaced with an interface
 */
public abstract class PrintDevice {

  public PrintDevice() {
  }
  /*
   * This package illustrates the advantege of using
   * polymorphism. The following is commented out
   * because the class is reduced to a 'type'
   */

/*
  public abstract void setItalics(boolean italics);
  public abstract void setBoldface(boolean boldface);
  public abstract void printHeader();
  public abstract void printFooter();
  public abstract void print(String text);
  public abstract void newLine();
*/
}
```

A.1.10 ReceiptLayout.java

```
/*
 * Created on 01-01-2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package violatesall;

/**
 * @author Søren Gaardbo
 *
 * Violations:
 * -----------
 * None, but could be replaced with an interface.
 */
```

```
16 public abstract class ReceiptLayout {
17   private Sale sale;
18   public ReceiptLayout() {
19     sale = null;
20   }
21   public ReceiptLayout(Sale _sale) {
22     sale = _sale;
23   }
24   public Sale getSale() {
25     return sale;
26   }
27   public abstract void printLayout(PrintDevice device);
28
29 }
```

A.1.11 Sale.java

```
1  /*
2   * Created on 26-12-2003
3   *
4   */
5  package violatesall;
6
7  import java.util.Vector;
8
9  /**
10  * @author Søren Gaardbo
11  * Violations:
12  * -----------
13  * Use Interface instead of concrete class (SalesItem, salesEntries...)
14  * Violates Information Expert. Calculation of subtotals should reside
15  *    in SalesEntry.
16  */
17 public class Sale {
18   private Vector salesEntries;
19   private Customer customer;
20
21   public Sale() {
22     salesEntries = new Vector();
23     customer = new Customer("Søren", "Gaardbo", "Blankavej␣28");
24   }
25
26   public Vector getSalesEntries() {
27     return salesEntries;
28   }
29   public void addSalesItem(int qty, SalesItem salesItem) {
30     SalesEntry salesEntry = new SalesEntry(qty, salesItem);
31     salesEntries.add(salesEntry);
32   }
33   public double getTotal() {
34     double total = 0;
35     for(int i = 0; i < salesEntries.size(); i++) {
36       SalesEntry salesEntry = (SalesEntry) salesEntries.get(i);
```

```
37        total += getSubTotal(salesEntry);
38      }
39      return total;
40    }
41    public double getSubTotal(SalesEntry salesEntry) {
42      return (salesEntry.getQty() * salesEntry.getSalesItemPriceInclVat());
43    }
44
45    public Customer getCustomer() {
46      return customer;
47    }
48  }
```

A.1.12 SalesEntry.java

```
1   /*
2    * Created on 26-12-2003
3    *
4    */
5   package violatesall;
6
7   /**
8    * @author Søren Gaardbo
9    *
10   * Violations:
11   * -----------
12   * Should depend on interfaces e.g. ISalesItem etc.
13   * Should implement an interface 'ISalesEntry'
14   */
15  public class SalesEntry {
16    double qty;
17    SalesItem salesItem;
18
19    public SalesEntry(double quantity, SalesItem item) {
20      qty = quantity;
21      salesItem = item;
22    }
23    public SalesItem getSalesItem() {
24      return salesItem;
25    }
26    public double getQty() {
27      return qty;
28    }
29    public double getSalesItemPriceInclVat() {
30      return salesItem.getPriceInclVat();
31    }
32  }
```

A.1.13 SalesItem.java

```
1   /*
2    * Created on 26-12-2003
3    *
```

```
 */
package violatesall;

/**
 * @author Søren Gaardbo
 *
 * Violations:
 * -----------
 * Should implement an interface 'ISalesItem'
 */
public abstract class SalesItem {
  private double vatPct;
  private String name;
  private double price;

  public SalesItem(String prodName, double _price, double vat) {
    name = prodName;
    vatPct = vat;
    price = _price;
  }

  public double getPrice() {
    return price;
  }
  public double getVatPct() {
    return vatPct;
  }

  public double getPriceInclVat() {
    return getPrice() * (1 + (getVatPct() / 100));
  }
  public String getName() {
    return name;
  }
}
```

A.1.14 ScreenPrintDevice.java

```
/*
 * Created on 26-12-2003
 *
 */
package violatesall;

/**
 * @author Søren Gaardbo
 *
 * Violations:
 * -----------
 * Should implement 'IPrintDevice' instead of extension
 *    of 'PrintDevice'.
 */
public class ScreenPrintDevice extends PrintDevice {
```

```
16    /*
17     * Class is empty. The class is reduced to
18     * a 'type'. Polymorphism is not used.
19     */
20  }
```

A.1.15 Stamp.java

```
1   /*
2    * Created on 26-12-2003
3    *
4    */
5   package violatesall;
6
7   /**
8    * @author Søren Gaardbo
9    *
10   * Violations:
11   * -----------
12   * Should implement an interface 'ISalesItem'
13   * instead of extension of 'SalesItem'.
14   */
15  public class Stamp extends SalesItem {
16
17    public Stamp() {
18      super("Stamp", 4.25, 0);
19    }
20  }
```

A.1.16 StdReceiptLayout.java

```
1   /*
2    * Created on 01-01-2004
3    *
4    */
5   package violatesall;
6
7   import java.util.*;
8   /**
9    * @author Søren Gaardbo
10   * * Violations:
11   * -----------
12   * Polymophism: Specific behaviour should be delegated to
13   *     the different 'PrintDevices'
14   *
15   * Law of Demeter: Statements like
16   *     salesEntry.getSalesItem().getName() should
17   *     be replaced.
18   *
19   * Note: This is a *very* 'brute force' implementation of
20   *     layout. The class 'PrintDevice' is actually
21   *     reduced to a 'selection attribute'.
22   *     The class is an ugly implementation with
```

```
23   *    lots of duplicated code.
24   */
25  public class StdReceiptLayout extends ReceiptLayout {
26    public StdReceiptLayout(Sale _sale) {
27      super(_sale);
28    }
29    public void printLayout(PrintDevice device) {
30
31      if (device instanceof ScreenPrintDevice) {
32        System.out.print("ReceiptHeader");
33
34        Vector salesEntries = getSale().getSalesEntries();
35        System.out.print(getSale().getCustomer().getName());
36        System.out.println();
37        System.out.print(getSale().getCustomer().getAddress());
38        System.out.println();
39        System.out.println();
40
41        for(int i = 0; i < salesEntries.size(); i++) {
42          SalesEntry salesEntry = (SalesEntry) salesEntries.get(i);
43          System.out.print(salesEntry.getQty() + "\t");
44          System.out.print(salesEntry.getSalesItem().getName() + "\t");
45          System.out.print(getSale().getSubTotal(salesEntry));
46          System.out.println();
47        }
48        System.out.print("--------------------");
49        System.out.println();
50        System.out.print("*");    // Boldface
51        System.out.print("Total:␣\t" + getSale().getTotal());
52        System.out.print("*");
53        System.out.println();
54        System.out.print("End␣of␣Receipt.");
55
56      }
57
58      if (device instanceof HTMLPrintDevice) {
59        System.out.print("<HTML><BODY>");
60
61        Vector salesEntries = getSale().getSalesEntries();
62        System.out.print(getSale().getCustomer().getName());
63        System.out.print("<BR>");
64        System.out.print(getSale().getCustomer().getAddress());
65        System.out.print("<BR>");
66        System.out.print("<BR>");
67
68        for(int i = 0; i < salesEntries.size(); i++) {
69          SalesEntry salesEntry = (SalesEntry) salesEntries.get(i);
70          System.out.print(salesEntry.getQty() + "\t");
71          System.out.print(salesEntry.getSalesItem().getName() + "\t");
72          System.out.print(getSale().getSubTotal(salesEntry));
73          System.out.print("<BR>");
74        }
75        System.out.print("--------------------");
76        System.out.print("<BR>");
```

```
77        System.out.print("<B>");    // Boldface
78        System.out.print("Total: \t" + getSale().getTotal());
79        System.out.print("</B>");
80        System.out.print("<BR>");
81        System.out.print("</BODY></HTML>");
82      }
83    }
84
85  }
```

A.2 Overholdelse af 'Information Expert'

A.2.1 Sale.java

```
1   /*
2    * Created on 26-12-2003
3    *
4    */
5   package fixInfExp;
6
7   import java.util.Vector;
8
9   /**
10   * @author Søren Gaardbo
11   */
12  public class Sale {
13    private Vector salesEntries;
14    private Customer customer;
15
16    public Sale() {
17      salesEntries = new Vector();
18      customer = new Customer("Søren", "Gaardbo", "Blankavej 28");
19    }
20
21    public Vector getSalesEntries() {
22      return salesEntries;
23    }
24    public void addSalesItem(int qty, SalesItem salesItem) {
25      SalesEntry salesEntry = new SalesEntry(qty, salesItem);
26      salesEntries.add(salesEntry);
27    }
28    public double getTotal() {
29      double total = 0;
30      for(int i = 0; i < salesEntries.size(); i++) {
31        SalesEntry salesEntry = (SalesEntry) salesEntries.get(i);
32        /*
33         * Following line fixes Information Expert.
34         */
35        total += salesEntry.getSubTotal();
36      }
37      return total;
38    }
39    /*
```

```
40      * getSubTotal is no longer used. getSubTotal is now
41      * implemented in the 'SalesEntry' class.
42      */
43     /*
44     public double getSubTotal(SalesEntry salesEntry) {
45       return (salesEntry.getQty() * salesEntry.getSalesItemPriceInclVat());
46     }
47     */
48
49     public Customer getCustomer() {
50       return customer;
51     }
52
53     /*
54      * PrintReciept moved to here from 'StdReceipt'
55      *
56      */
57     public void printLayout(PrintDevice device) {
58
59       if (device instanceof ScreenPrintDevice) {
60         System.out.print("ReceiptHeader");
61
62         Vector salesEntries = getSalesEntries();
63         System.out.print(getCustomer().getName());
64         System.out.println();
65         System.out.print(getCustomer().getAddress());
66         System.out.println();
67         System.out.println();
68
69         for(int i = 0; i < salesEntries.size(); i++) {
70           SalesEntry salesEntry = (SalesEntry) salesEntries.get(i);
71           System.out.print(salesEntry.getQty() + "\t");
72           System.out.print(salesEntry.getSalesItem().getName() + "\t");
73           System.out.print(salesEntry.getSubTotal());
74           System.out.println();
75         }
76         System.out.print("--------------------");
77         System.out.println();
78         System.out.print("*");    // Boldface
79         System.out.print("Total: \t" + getTotal());
80         System.out.print("*");
81         System.out.println();
82         System.out.print("End of Receipt.");
83
84       }
85
86       if (device instanceof HTMLPrintDevice) {
87         System.out.print("<HTML><BODY>");
88
89         Vector salesEntries = getSalesEntries();
90         System.out.print(getCustomer().getName());
91         System.out.print("<BR>");
92         System.out.print(getCustomer().getAddress());
93         System.out.print("<BR>");
```

```
94        System.out.print("<BR>");
95
96        for(int i = 0; i < salesEntries.size(); i++) {
97          SalesEntry salesEntry = (SalesEntry) salesEntries.get(i);
98          System.out.print(salesEntry.getQty() + "\t");
99          System.out.print(salesEntry.getSalesItem().getName() + "\t");
100         System.out.print(salesEntry.getSubTotal());
101         System.out.print("<BR>");
102       }
103       System.out.print("--------------------");
104       System.out.print("<BR>");
105       System.out.print("<B>");   // Boldface
106       System.out.print("Total:␣\t" + getTotal());
107       System.out.print("</B>");
108       System.out.print("<BR>");
109       System.out.print("</BODY></HTML>");
110     }
111   }
112
113 }
```

A.2.2 SalesEntry.java

```
1   /*
2   * Created on 26-12-2003
3   *
4   */
5  package fixInfExp;
6
7  /**
8   * @author Søren Gaardbo
9   *
10  */
11 public class SalesEntry {
12   double qty;
13   SalesItem salesItem;
14
15   public SalesEntry(double quantity, SalesItem item) {
16     qty = quantity;
17     salesItem = item;
18   }
19   public SalesItem getSalesItem() {
20     return salesItem;
21   }
22   public double getQty() {
23     return qty;
24   }
25   /*
26    * getSalesItemPriceInclVat is no longer called
27    * because 'getSubTotal' is implemented.
28    */
29   /*
30   public double getSalesItemPriceInclVat() {
```

```
31      return salesItem.getPriceInclVat();
32    }
33    */
34    public double getSubTotal() {
35      return salesItem.getPriceInclVat() * qty;
36    }
37  }
```

A.3 Overholdelse af Polymorfi

A.3.1 PrintDevice.java

```
1   /*
2    * Created on 01-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package fixPoly;
8
9   /**
10   * @author Søren Gaardbo
11   *
12   * To change the template for this generated type comment go to
13   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14   */
15  public abstract class PrintDevice {
16
17    public PrintDevice() {
18    }
19    public abstract void setItalics(boolean italics);
20    public abstract void setBoldface(boolean boldface);
21    public abstract void printHeader();
22    public abstract void printFooter();
23    public abstract void print(String text);
24    public abstract void newLine();
25  }
```

A.3.2 ScreenPrintDevice.java

```
1   /*
2    * Created on 26-12-2003
3    *
4    */
5   package fixPoly;
6
7   /**
8    * @author Søren Gaardbo
9    *
10   */
11  public class ScreenPrintDevice extends PrintDevice {
12
```

```
13    public void setBoldface(boolean boldface) {
14       System.out.print("*");
15    }
16
17    public void setItalics(boolean italics) {
18       System.out.print("/");
19    }
20    public void printFooter() {
21      System.out.print("End␣of␣Receipt.");
22    }
23    public void printHeader() {
24      System.out.print("ReceiptHeader");
25    }
26
27    public void newLine() {
28      System.out.println("");
29    }
30
31    public void print(String text) {
32      System.out.print(text);
33    }
34  }
```

A.3.3 HTMLPrintDevice.java

```
1   /*
2    * Created on 01-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package fixPoly;
8
9   /**
10   * @author Søren Gaardbo
11   *
12   * To change the template for this generated type comment go to
13   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14   */
15  public class HTMLPrintDevice extends PrintDevice {
16
17    public void setItalics(boolean italics) {
18      if (italics)
19        System.out.print("<i>");
20      else
21        System.out.print("</i>");
22
23    }
24
25    public void setBoldface(boolean boldface) {
26      if (boldface)
27        System.out.print("<b>");
28      else
```

```
29        System.out.print("</b>");
30
31    }
32
33    public void printHeader() {
34      System.out.println("<HTML><BODY>");
35
36    }
37
38    public void printFooter() {
39      System.out.println("</BODY></HTML>");
40
41    }
42
43    public void print(String text) {
44      System.out.print(text);
45
46    }
47
48    public void newLine() {
49      System.out.print("<BR>");
50
51    }
52
53  }
```

A.4 Overholdelse af Law of Demeter

A.4.1 Sale.java

```
1   /*
2    * Created on 26-12-2003
3    *
4    */
5   package fixLoD;
6
7   import java.util.Vector;
8
9   /**
10   * @author Søren Gaardbo
11   * Violations:
12   * -----------
13   * Use Interface instead of concrete class (SalesItem, salesEntries...)
14   * Violates Information Expert. Calculation of subtotals should reside
15   *    in SalesEntry.
16   */
17  public class Sale {
18    private Vector salesEntries;
19    private Customer customer;
20
21    public Sale() {
22      salesEntries = new Vector();
23      customer = new Customer("Søren", "Gaardbo", "Blankavej␣28");
```

```
24    }
25    /*
26     * 'getSalesEntries' is no longer called.
27     */
28    /*
29    public Vector getSalesEntries() {
30      return salesEntries;
31    }
32    */
33    public void addSalesItem(int qty, SalesItem salesItem) {
34      SalesEntry salesEntry = new SalesEntry(qty, salesItem);
35      salesEntries.add(salesEntry);
36    }
37    public double getTotal() {
38      double total = 0;
39      for(int i = 0; i < salesEntries.size(); i++) {
40        SalesEntry salesEntry = (SalesEntry) salesEntries.get(i);
41        total += getSalesItemSubtotal(i);;
42      }
43      return total;
44    }
45
46    public Customer getCustomer() {
47      return customer;
48    }
49
50    /*
51     * Following methods was added to confirm to the
52     * Law of Demeter.
53     *
54     * Note: The methods getSalesItemName and getSalesItemSubtotal
55     *      does actually NOT confirm to LoD.
56     *      This is because of the use of Vector (SalesEntries).
57     */
58    public String getCustomerName() {
59      return customer.getName();
60    }
61    public String getCustomerAddress() {
62      return customer.getAddress();
63    }
64    public int getSalesItemCount() {
65      return salesEntries.size();
66    }
67    public double getSalesItemQty(int index) {
68      return ((SalesEntry) salesEntries.get(index)).getQty();
69    }
70    public String getSalesItemName(int index) {
71      return ((SalesEntry) salesEntries.get(index)).getSalesItemName();
72    }
73    public double getSalesItemSubtotal(int index) {
74      SalesEntry se = (SalesEntry) salesEntries.get(index);
75      return se.getQty() * se.getSalesItemPriceInclVat();
76    }
77  }
```

A.4.2 SalesEntry.java

```
1  /*
2   * Created on 26-12-2003
3   *
4   */
5  package fixLoD;
6
7  /**
8   * @author Søren Gaardbo
9   *
10  */
11 public class SalesEntry {
12   double qty;
13   SalesItem salesItem;
14
15   public SalesEntry(double quantity, SalesItem item) {
16     qty = quantity;
17     salesItem = item;
18   }
19   public SalesItem getSalesItem() {
20     return salesItem;
21   }
22   public double getQty() {
23     return qty;
24   }
25   public double getSalesItemPriceInclVat() {
26     return salesItem.getPriceInclVat();
27   }
28   /*
29    * Following method was added to confirm to the
30    * Law of Demeter.
31    */
32   public String getSalesItemName() {
33     return salesItem.getName();
34   }
35 }
```

A.4.3 StdReceiptLayout.java

```
1  /*
2   * Created on 01-01-2004
3   *
4   * To change the template for this generated file go to
5   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6   */
7  package fixLoD;
8
9  /**
10  * @author Søren Gaardbo
11  * * Violations:
12  * -----------
13  * Polymophy: Specific behaviour should be delegated to
14  *    the different 'PrintDevices'
```

```
15   *
16   * Law of Demeter: Statements like
17   *    salesEntry.getSalesItem().getName() should
18   *    be replaced.
19   *
20   * Note: This is a *very* 'brute force' implementation of
21   *    layout. The class 'PrintDevice' is actually
22   *    reduced to a 'selection attribute'.
23   *    The class is an ugly implementation with
24   *    lots of duplicated code.
25   *
26   * Note: Some of the code appears to violate LoD. Statements
27   *    as eg. getSales().getCustomerName(). This is not a
28   *    problem because the class is already coupled to
29   *    'Customer' through its superclass. The alternative
30   *    would be to make 'Customer' protected in the
31   *    superclass.
32   */
33  public class StdReceiptLayout extends ReceiptLayout {
34    public StdReceiptLayout(Sale _sale) {
35      super(_sale);
36    }
37    public void printLayout(PrintDevice device) {
38
39      if (device instanceof ScreenPrintDevice) {
40        System.out.print("ReceiptHeader");
41
42        System.out.print(getSale().getCustomerName());
43        System.out.println();
44        System.out.print(getSale().getCustomerAddress());
45        System.out.println();
46        System.out.println();
47
48        for(int i = 0; i < getSale().getSalesItemCount(); i++) {
49          System.out.print(getSale().getSalesItemQty(i) + "\t");
50          System.out.print(getSale().getSalesItemName(i) + "\t");
51          System.out.print(getSale().getSalesItemSubtotal(i));
52          System.out.println();
53        }
54        System.out.print("--------------------");
55        System.out.println();
56        System.out.print("*");    // Boldface
57        System.out.print("Total:␣\t" + getSale().getTotal());
58        System.out.print("*");
59        System.out.println();
60        System.out.print("End␣of␣Receipt.");
61
62      }
63
64      if (device instanceof HTMLPrintDevice) {
65        System.out.print("<HTML><BODY>");
66
67        System.out.print(getSale().getCustomerName());
68        System.out.print("<BR>");
```

```
69        System.out.print(getSale().getCustomerAddress());
70        System.out.print("<BR>");
71        System.out.print("<BR>");
72
73        for(int i = 0; i < getSale().getSalesItemCount(); i++) {
74          System.out.print(getSale().getSalesItemQty(i) + "\t");
75          System.out.print(getSale().getSalesItemName(i) + "\t");
76          System.out.print(getSale().getSalesItemSubtotal(i));
77          System.out.println();
78          System.out.print("<BR>");
79        }
80        System.out.print("--------------------");
81        System.out.print("<BR>");
82        System.out.print("<B>");   // Boldface
83        System.out.print("Total: \t" + getSale().getTotal());
84        System.out.print("</B>");
85        System.out.print("<BR>");
86        System.out.print("</BODY></HTML>");
87      }
88    }
89
90  }
```

A.5 Isoleret forsøg med Information Expert

A.5.1 Kode der bryder med Information Expert

DataA.java

```
1   /*
2    * Created on 05-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package infExp.violation;
8
9   /**
10   * @author SGJ
11   *
12   * To change the template for this generated type comment go to
13   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14   */
15  public class DataA {
16    private int a;
17    private int b;
18    public DataA(int _a, int _b) {
19      a = _a;
20      b = _b;
21    }
22
23    public int getA() {
24      return a;
```

```
  }
  public int getB() {
    return b;
  }
}
```

Run.java

```
/*
 * Created on 02-01-2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package infExp.violation;

/**
 * @author Søren Gaardbo
 *
 */
public class Run {
  public Run() {
    DataA dataA = new DataA(1,2);
    OpAdd opAdd = new OpAdd();
    OpMul opMul = new OpMul();
    OpSub opSub = new OpSub();

    opAdd.addAndPrint(dataA.getA(), dataA.getB());
    opSub.subAndPrint(dataA.getA(), dataA.getB());
    opMul.mul(dataA);

  }
  public static void main(String[] args) {
    new Run();
  }
}
```

OpAdd.java

```
/*
 * Created on 05-01-2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package infExp.violation;

/**
 * @author SGJ
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
```

```
15 public class OpAdd {
16   public void addAndPrint(int a, int b) {
17     int d = a + b;
18     System.out.println("Result:␣" + d);
19   }
20 }
```

OpMul.java

```
1  /*
2   * Created on 08-01-2004
3   *
4   * To change the template for this generated file go to
5   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6   */
7  package infExp.violation;
8
9  /**
10  * @author SGJ
11  *
12  * To change the template for this generated type comment go to
13  * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14  */
15 public class OpMul {
16   public void mul(DataA data) {
17     int d = data.getA() * data.getB();
18     System.out.println("Result:␣" + d);
19   }
20
21 }
```

OpSub.java

```
1  /*
2   * Created on 08-01-2004
3   *
4   * To change the template for this generated file go to
5   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6   */
7  package infExp.violation;
8
9  /**
10  * @author SGJ
11  *
12  * To change the template for this generated type comment go to
13  * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14  */
15 public class OpSub {
16   public void subAndPrint(int a, int b) {
17     int d = a - b;
18     System.out.println("Result:␣" + d);
19   }
20 }
```

A.5.2 Kode der overholder Information Expert

DataA.java

```
/*
 * Created on 05-01-2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package infExp.confirms;

/**
 * @author SGJ
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
public class DataA {
  private int a;
  private int b;
  public DataA(int _a, int _b) {
    a = _a;
    b = _b;
  }
  public void addAndPrint() {
    int d = a + b;
    System.out.println("Result: " + d);
  }
  public void mul() {
    int d = a * b;
    System.out.println("Result: " + d);
  }
  public void subAndPrint() {
    int d = a - b;
    System.out.println("Result: " + d);
  }

  public int getA() {
    return a;
  }
  public int getB() {
    return b;
  }
}
```

Run.java

```
/*
 * Created on 02-01-2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
```

```
7  package infExp.confirms;
8
9  /**
10  * @author Søren Gaardbo
11  *
12  */
13 public class Run {
14   public Run() {
15     DataA dataA = new DataA(1,2);
16     /*
17     OpAdd opAdd = new OpAdd();
18     OpMul opMul = new OpMul();
19     OpSub opSub = new OpSub();
20
21     opAdd.addAndPrint(dataA.getA(), dataA.getB());
22     opSub.subAndPrint(dataA.getA(), dataA.getB());
23     opMul.mul(dataA);
24     */
25     dataA.addAndPrint();
26     dataA.subAndPrint();
27     dataA.mul();
28
29   }
30   public static void main(String[] args) {
31     new Run();
32   }
33 }
```

A.6 Isoleret forsøg med polymorfi

A.6.1 Brud på polymorfi i superklasse

S.java

```
1  /*
2   * Created on 05-01-2004
3   *
4   * To change the template for this generated file go to
5   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6   */
7  package polyviolationSuper;
8  import java.io.*;
9  /**
10  * @author SGJ
11  *
12  * To change the template for this generated type comment go to
13  * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14  */
15 public class S {
16   public void classSpecificBehaviour() {
17     if (this instanceof B1) {
18       System.out.println("Class␣is␣B1");
19     }
```

```
20      else if (this instanceof B2) {
21        try {
22          FileWriter filewriter =
23            new FileWriter("C:\\polyv.txt");
24          filewriter.write("Class␣is␣B2");
25        } catch (IOException ioe) {}
26      }
27      else if (this instanceof B3) {
28        try {
29          FileOutputStream fileOutput =
30            new FileOutputStream("C:\\poly3.txt");
31          fileOutput.write(64);
32        } catch(IOException ioe) {}
33      }
34    }
35  }
```

B1.java

```
1   /*
2    * Created on 05-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package polyviolationSuper;
8
9   /**
10   * @author SGJ
11   *
12   * To change the template for this generated type comment go to
13   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14   */
15  public class B1 extends S {
16
17  }
```

B2.java

```
1   /*
2    * Created on 05-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package polyviolationSuper;
8
9   /**
10   * @author SGJ
11   *
12   * To change the template for this generated type comment go to
13   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14   */
```

```
15 public class B2 extends S {
16
17 }
```

B3.java

```
1  /*
2   * Created on 05-01-2004
3   *
4   * To change the template for this generated file go to
5   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6   */
7  package polyviolationSuper;
8
9  /**
10  * @author SGJ
11  *
12  * To change the template for this generated type comment go to
13  * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14  */
15 public class B3 extends S {
16
17 }
```

Run.java

```
1  /*
2   * Created on 05-01-2004
3   *
4   * To change the template for this generated file go to
5   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6   */
7  package polyviolationSuper;
8
9  /**
10  * @author SGJ
11  *
12  * To change the template for this generated type comment go to
13  * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14  */
15 public class Run {
16
17   public Run() {
18     S s;
19     s = new B1();
20     s.classSpecificBehaviour();
21     s = new B2();
22     s.classSpecificBehaviour();
23     s = new B3();
24     s.classSpecificBehaviour();
25
26   }
27   public static void main(String[] args) {
```

```
28      new Run();
29    }
30  }
```

A.6.2 Brud på polymorfi med hjælpeklasse

S.java

```
1   /*
2    * Created on 05-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package polyvariationExtern;
8
9   /**
10   * @author SGJ
11   *
12   * To change the template for this generated type comment go to
13   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14   */
15  public class S {
16  }
```

B1.java

```
1   /*
2    * Created on 05-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package polyvariationExtern;
8
9   /**
10   * @author SGJ
11   *
12   * To change the template for this generated type comment go to
13   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14   */
15  public class B1 extends S {
16
17  }
```

B2.java

```
1   /*
2    * Created on 05-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
```

```
7  package polyvariationExtern;
8
9  /**
10  * @author SGJ
11  *
12  * To change the template for this generated type comment go to
13  * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14  */
15 public class B2 extends S {
16
17 }
```

B3.java

```
1  /*
2  * Created on 05-01-2004
3  *
4  * To change the template for this generated file go to
5  * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6  */
7  package polyvariationExtern;
8
9  /**
10  * @author SGJ
11  *
12  * To change the template for this generated type comment go to
13  * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14  */
15 public class B3 extends S {
16
17 }
```

Run.java

```
1  /*
2  * Created on 05-01-2004
3  *
4  * To change the template for this generated file go to
5  * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6  */
7  package polyvariationExtern;
8
9  /**
10  * @author SGJ
11  *
12  * To change the template for this generated type comment go to
13  * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14  */
15 public class Run {
16
17   public Run() {
18     S s;
19     Tool t = new Tool();
```

```
20      s = new B1();
21      t.classSpecificBehaviour(s);
22      s = new B2();
23      t.classSpecificBehaviour(s);
24      s = new B3();
25      t.classSpecificBehaviour(s);
26
27    }
28    public static void main(String[] args) {
29      new Run();
30    }
31  }
```

Tool.java

```
1   /*
2    * Created on 05-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package polyvariationExtern;
8
9   import java.io.*;
10  /**
11   * @author SGJ
12   *
13   * To change the template for this generated type comment go to
14   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
15   */
16  public class Tool {
17    public void classSpecificBehaviour(S s) {
18      if (s instanceof B1) {
19        System.out.println("Class␣is␣B1");
20      }
21      else if (s instanceof B2) {
22        try {
23          FileWriter filewriter =
24            new FileWriter("C:\\polyv.txt");
25          filewriter.write("Class␣is␣B2");
26        } catch (IOException ioe) {}
27      }
28      else if (s instanceof B3) {
29        try {
30          FileOutputStream fileOutput =
31            new FileOutputStream("C:\\poly3.txt");
32          fileOutput.write(64);
33        } catch(IOException ioe) {}
34      }
35
36
37      if (s instanceof B1) {
38        System.out.println("Class␣is␣B1");
```

```
39      }
40      else if (s instanceof B2) {
41        System.out.println("Class␣is␣B2");
42      }
43      else if (s instanceof B3) {
44        System.out.println("Class␣is␣B3");
45      }
46    }
47
48  }
```

A.6.3 Brud på polymorfi fjernet

S.java

```
1   /*
2    * Created on 05-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package polyconfirms;
8
9   /**
10   * @author SGJ
11   *
12   * To change the template for this generated type comment go to
13   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14   */
15  public abstract class S {
16    public abstract void classSpecificBehaviour();
17  }
```

B1.java

```
1   /*
2    * Created on 05-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package polyconfirms;
8
9   /**
10   * @author SGJ
11   *
12   * To change the template for this generated type comment go to
13   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14   */
15  public class B1 extends S {
16    public void classSpecificBehaviour() {
17      System.out.println("Class␣is␣B1");
```

```
  }
}
```

B2.java

```
/*
 * Created on 05-01-2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package polyconfirms;

import java.io.*;
/**
 * @author SGJ
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
public class B2 extends S {
  public void classSpecificBehaviour() {
    try {
      FileWriter filewriter =
        new FileWriter("C:\\polyv.txt");
      filewriter.write("Class␣is␣B2");
    } catch (IOException ioe) {}
  }
}
```

B3.java

```
/*
 * Created on 05-01-2004
 *
 * To change the template for this generated file go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
package polyconfirms;

import java.io.*;

/**
 * @author SGJ
 *
 * To change the template for this generated type comment go to
 * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
 */
public class B3 extends S {
  public void classSpecificBehaviour() {
    try {
      FileOutputStream fileOutput =
        new FileOutputStream("C:\\poly3.txt");
```

```
22        fileOutput.write(64);
23      } catch(IOException ioe) {}
24    }
25  }
```

Run.java

```
1   /*
2    * Created on 05-01-2004
3    *
4    * To change the template for this generated file go to
5    * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
6    */
7   package polyconfirms;
8
9   /**
10   * @author SGJ
11   *
12   * To change the template for this generated type comment go to
13   * Window&gt;Preferences&gt;Java&gt;Code Generation&gt;Code and Comments
14   */
15  public class Run {
16
17    public Run() {
18      S s;
19      s = new B1();
20      s.classSpecificBehaviour();
21      s = new B2();
22      s.classSpecificBehaviour();
23      s = new B3();
24      s.classSpecificBehaviour();
25
26    }
27    public static void main(String[] args) {
28      new Run();
29    }
30  }
```

A.7 Isoleret forsøg med Law of Demeter

A.7.1 Brud på Law of Demeter

ExtrLod.java

```
1   /*
2    * Created on 29-12-2003
3    *
4    */
5   package lod.before;
6
7   /**
8    * @author Søren Gaardbo
9    *
```

```
 */
public class ExtrLod {
/*
 * Law of Demeter is violated in the
 * constructor 'ExtrLod' with the
 * message chain a.getB()....
 */
  public ExtrLod() {
    A a = new A();
    a.getB().getC().getD().opOnD();
  }
  public static void main(String[] args) {
    new ExtrLod();
  }
}
```

A.java

```
/*
 * Created on 29-12-2003
 *
 */
package lod.before;

/**
 * @author Søren Gaardbo
 *
 */
public class A {
  private B b;
  public A() {
    b = new B();
  }
  public B getB() {
    return b;
  }

}
```

B.java

```
/*
 * Created on 29-12-2003
 *
 */
package lod.before;

/**
 * @author Søren Gaardbo
 *
 */
public class B {
  private C c;
```

```
13    public B() {
14      c = new C();
15    }
16    public C getC() {
17      return c;
18    }
19  }
```

C.java

```
1   /*
2    * Created on 29-12-2003
3    *
4    */
5   package lod.before;
6
7   /**
8    * @author Søren Gaardbo
9    *
10   */
11  public class C {
12    private D d;
13    public C() {
14      d = new D();
15    }
16    public D getD() {
17      return d;
18    }
19
20  }
```

D.java

```
1   /*
2    * Created on 29-12-2003
3    *
4    */
5   package lod.before;
6
7   /**
8    * @author Søren Gaardbo
9    *
10   */
11  public class D {
12    public void opOnD() {
13      System.out.println("Operation␣on␣D␣called.");
14    }
15
16  }
```

Overholdelse af Law of Demeter

ExtrLod.java

```
1  /*
2   * Created on 29-12-2003
3   *
4   */
5  package lod.after;
6
7  /**
8   * @author Søren Gaardbo
9   *
10  */
11 public class ExtrLod {
12
13   public ExtrLod() {
14     A a = new A();
15     // a.getB().getC().getD().opOnD();
16     a.opOnB();
17   }
18   public static void main(String[] args) {
19     new ExtrLod();
20   }
21 }
```

A.java

```
1  /*
2   * Created on 29-12-2003
3   *
4   */
5  package lod.after;
6
7  /**
8   * @author Søren Gaardbo
9   *
10  */
11 public class A {
12   private B b;
13   public A() {
14     b = new B();
15   }
16   public B getB() {
17     return b;
18   }
19   public void opOnB() {
20     b.opOnC();
21   }
22 }
```

B.java

```
/*
 * Created on 29-12-2003
 *
 */
package lod.after;

/**
 * @author Søren Gaardbo
 *
 */
public class B {
  private C c;
  public B() {
    c = new C();
  }
  public C getC() {
    return c;
  }
  public void opOnC() {
    c.opOnD();
  }
}
```

C.java

```
/*
 * Created on 29-12-2003
 *
 */
package lod.after;

/**
 * @author Søren Gaardbo
 *
 */
public class C {
  private D d;
  public C() {
    d = new D();
  }
  public D getD() {
    return d;
  }
  public void opOnD() {
    d.opOnD();
  }

}
```

D.java

```
1  /*
2   * Created on 29-12-2003
3   *
4   */
5  package lod.after;
6
7  /**
8   * @author Søren Gaardbo
9   *
10  */
11 public class D {
12   public void opOnD() {
13     System.out.println("Operation␣on␣D␣called.");
14   }
15
16 }
```

Bilag B

Data fra eksperimenter

Her følger de data som ligger til grundlag for graferne i 8

B.1 Kendskabsabstraktionsfaktoren (KAF)

Kategori	Antal	%
x = 0	2339	31.00
0 < x < 0.1	317	4.20
0.1 <= x < 0.2	791	10.49
0.2 <= x < 0.3	1164	15.43
0.3 <= x < 0.4	998	13.23
0.4 <= x < 0.5	543	7.20
0.5 <= x < 0.6	657	8.71
0.6 <= x < 0.7	255	3.38
0.7 <= x < 0.8	101	1.34
0.8 <= x < 0.9	72	0.95
0.9 <= x < 1	3	0.04
x = 1	304	4.03
Sum	7544	100.00

Tabel B.1: KAF i Eclipse.

Kategori	Antal	%
x = 0	226	58.70
0 < x < 0.1	33	8.57
0.1 <= x < 0.2	61	15.84
0.2 <= x < 0.3	44	11.43
0.3 <= x < 0.4	10	2.60
0.4 <= x < 0.5	1	0.26
0.5 <= x < 0.6	6	1.56
0.6 <= x < 0.7	1	0.26
0.7 <= x < 0.8	0	-
0.8 <= x < 0.9	0	-
0.9 <= x < 1	0	-
x = 1	3	0.78
Sum	385	100.00

Tabel B.2: KAF i Pos.

B.2 Koblingssabstraktionsfaktoren (CAF)

Kategori	Antal	%
x = 0	3615	47.92
0 < x < 0.1	148	1.96
0.1 <= x < 0.2	542	7.18
0.2 <= x < 0.3	820	10.87
0.3 <= x < 0.4	708	9.38
0.4 <= x < 0.5	496	6.57
0.5 <= x < 0.6	610	8.09
0.6 <= x < 0.7	284	3.76
0.7 <= x < 0.8	88	1.17
0.8 <= x < 0.9	44	0.58
0.9 <= x < 1	2	0.03
x = 1	187	2.48
Sum	7544	100.00

Tabel B.3: CAF i Eclipse.

Kategori	Antal	%
x = 0	271	70.39
0 < x < 0.1	10	2.60
0.1 <= x < 0.2	44	11.43
0.2 <= x < 0.3	18	4.68
0.3 <= x < 0.4	29	7.53
0.4 <= x < 0.5	5	1.30
0.5 <= x < 0.6	7	1.82
0.6 <= x < 0.7	0	-
0.7 <= x < 0.8	0	-
0.8 <= x < 0.9	0	-
0.9 <= x < 1	0	-
x = 1	1	0.26
Sum	385	100.00

Tabel B.4: CAF i Pos.

B.3 Kohæsionen LCOM*

Kategori	Antal	%
x = 0	3881	51.57
0 < x < 0.1	9	0.12
0.1 <= x < 0.2	35	0.47
0.2 <= x < 0.3	93	1.24
0.3 <= x < 0.4	155	2.060
0.4 <= x < 0.5	104	1.38
0.5 <= x < 0.6	635	8.44
0.6 <= x < 0.7	566	7.52
0.7 <= x < 0.8	629	8.36
0.8 <= x < 0.9	826	10.98
0.9 <= x < 1	468	6.23
x = 1	125	1.66
Sum	7526	100

Tabel B.5: LCOM* i Eclipse.

Kategori	Antal	%
x = 0	206	59.37
0 < x < 0.1	0	0
0.1 <= x < 0.2	0	0
0.2 <= x < 0.3	1	0.29
0.3 <= x < 0.4	0	0
0.4 <= x < 0.5	0	0
0.5 <= x < 0.6	20	5.76
0.6 <= x < 0.7	31	8.93
0.7 <= x < 0.8	53	15.27
0.8 <= x < 0.9	19	5.48
0.9 <= x < 1	15	4.32
x = 1	2	0.59
Sum	347	100

Tabel B.6: LCOM* i Pos.

B.4 Kobling (CBO)

Kategori	Antal	%
x = 0	1988	26.35
0<x<10	3480	46.13
10 <= x < 20	1303	17.27
20 <= x < 30	433	5.74
30 <= x < 40	201	2.66
40 <= x < 50	70	0.93
50 <= x < 60	26	0.34
60 <= x < 70	23	0.30
70 <= x < 80	12	0.16
80 <= x < 90	5	0.07
90 <= x < 100	3	0.04
Sum	7544	100.00

Tabel B.7: CBO i Eclipse.

Kategori	Antal	%
x = 0	106	27.53
0<x<10	234	60.78
10 <= x < 20	40	10.39
20 <= x < 30	4	1.04
30 <= x < 40	0	-
40 <= x < 50	1	0.26
50 <= x < 60	0	-
60 <= x < 70	0	-
70 <= x < 80	0	-
80 <= x < 90	0	-
90 <= x < 100	0	-
Sum	385	100.00

Tabel B.8: CBO i Pos.

B.5 Kobling (0 < CBO <= 20)

Kategori	Antal	%
0 < x <= 2	1053	13.96
2 < x <= 4	860	11.40
4 < x <= 6	739	9.80
6 < x <= 8	589	7.81
8 < x <= 10	474	6.28
10 < x <= 12	340	4.51
12 < x <= 14	254	3.37
14 < x <= 16	218	2.89
16 < x <=18	190	2.52
18 < x <=20	130	1.72
Sum	4847	62.53

Tabel B.9: CBO i Eclipse (0 < CBO <= 20).

Kategori	Antal	%
0 < x <= 2	64	16.62
2 < x <= 4	68	17.66
4 < x <= 6	69	17.92
6 < x <= 8	18	4.68
8 < x <= 10	32	8.31
10 < x <= 12	12	3.12
12 < x <= 14	5	1.30
14 < x <= 16	4	1.04
16 < x <=18	0	-
18 < x <=20	2	0.52
Sum	274	71.17

Tabel B.10: CBO i Pos(0 < CBO <= 20).

B.6 Kendskab

Kategori	Antal	%
x = 0	478	6.35
0<x<10	3759	49.95
10 <= x < 20	1776	23.60
20 <= x < 30	738	9.81
30 <= x < 40	364	4.84
40 <= x < 50	201	2.67
50 <= x < 60	87	1.16
60 <= x < 70	58	0.77
70 <= x < 80	36	0.48
80 <= x < 90	17	0.23
90 <= x < 100	12	0.16
100 <= x	17	0.23
Sum	7543	100.00

Tabel B.11: Kendskab i Eclipse.

Kategori	Antal	%
x = 0	10	2.60
0<x<10	281	72.99
10 <= x < 20	80	20.78
20 <= x < 30	6	1.56
30 <= x < 40	6	1.56
40 <= x < 50	1	0.26
50 <= x < 60	0	-
60 <= x < 70	0	-
70 <= x < 80	0	-
80 <= x < 90	1	0.26
90 <= x < 100	0	-
100 <= x	0	-
Sum	385	100.00

Tabel B.12: Kendskab i Pos.

B.7 Kendskab (0 < kendskab <= 20)

Kategori	Antal	%
0 < x <= 2	1236	16.42
2 < x <= 4	962	12.78
4 < x <= 6	720	9.57
6 < x <= 8	575	7.64
8 < x <= 10	523	6.95
10 < x <= 12	476	6.32
12 < x <= 14	379	5.04
14 < x <= 16	311	4.13
16 < x <=18	241	3.20
18 < x <=20	228	3.03
Sum	5651	75.09

Tabel B.13: Kendskab i Eclipse (0 < kendskab <= 20).

Kategori	Antal	%
0 < x <= 2	113	29.35
2 < x <= 4	51	13.25
4 < x <= 6	55	14.29
6 < x <= 8	29	7.53
8 < x <= 10	49	12.73
10 < x <= 12	16	4.16
12 < x <= 14	21	5.45
14 < x <= 16	16	4.16
16 < x <=18	8	2.08
18 < x <=20	3	0.78
Sum	361	93.77

Tabel B.14: Kendskab i Pos (0 < kendskab <= 20).

Bilag C

Indhold af cd-rom

I dette appendiks beskrives indholdet af den medfølgende cd-rom. Beskrivelsen er inddelt i indholdet af katalogerne på cd-romen.

PlugIn indeholder den plug-in der blev udviklet til Eclipse, inklusive kildekode. Bemærk at "metrics" skal være installeret for at plug-in'en vil virke.

Metrics indeholder en plug-in hvis framework abstraktionsmålene brugs. Hentet fra `http://metrics.sourceforge.net`

Eclipse indeholder Eclipse version 2.1.1 til windows. Den er hentet fra `http://www.eclipse.org`

Scr indeholder kildekoden til eksperimenterne.

Results indeholder data fra de eksperimenter der blev foretaget på kildekoden til Eclipse, og POS.

Speciale indeholder specialet i .dvi,- .ps- og .pdf-format.