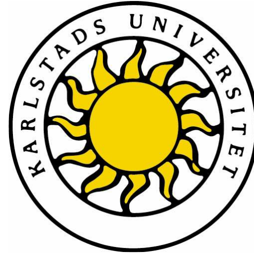


Avdelning för datavetenskap

Andréas Jonsson

Införande av objektorienterade mönster för ökad förändringsbarhet i mjukvarusystem

Introduction of object oriented patterns to increase software modifiability

Examensarbete (20p)
Civilingenjör Informationsteknologi

Datum:	07-01-17
Handledare:	Donald Ross
Examinator:	Martin Blom
Löpnummer:	D2007:01

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

Införande av objektorienterade mönster för ökad förändringsbarhet i mjukvarusystem

Andréas Jonsson

© 2007 Författaren och Karlstads Universitet

Denna uppsats är skriven som en del av det arbete som krävs för att erhålla en magisterexamen i datavetenskap. Allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Andréas Jonsson

Godkänd, 2007-01-17

Opponent: Eivind Nordby

Handledare: Donald Ross

Examinator: Martin Blom

Sammanfattning

Objektorienterade mönster och omkonstruktion är två olika designstrategier som har ett gemensamt mål: att göra mjukvarusystem mer förändringsbara och mindre komplexa. Mönster tillämpas för att förebygga utvecklingen av komplexitet i mjukvara. Omkonstruktioner görs för att reducera komplexitet som redan uppstått i mjukvarans inre struktur. Denna rapport identifierar fyra grundläggande strukturproblem som gör mjukvara onödigt komplex och svår att förändra: duplicerad kod, villkorslogik, långa metoder och bristande inkapsling. Rapporten visar hur objektorienterade mönster kan införas i mjukvara genom omkonstruktion och göra mjukvara mer förändringsbar genom att reducera de fyra nämnda strukturproblemen. Som en fallstudie om mönsterbaserade omkonstruktioner, omkonstrueras en del av systemet INCA genom att tillämpa mönstren Template Method och Strategy.

Introduction of object oriented patterns to increase software modifiability

Object oriented patterns and refactoring are two different approaches to software design that both share a common objective: to increase software modifiability and reduce its complexity. Patterns are applied to prevent the development of software complexity. Refactorings are applied to reduce complexity that has already found its way into the internal structures of software. This report identifies four fundamental structural problems that make software unnecessarily complex and hard to maintain: duplicated code, conditional logic, long methods and poor encapsulation. The report shows how object oriented patterns can be introduced in software by means of refactoring and make software more modifiable by reducing the four mentioned structural problems. As a case study of pattern based refactorings, a part of the INCA system is refactored by applying the patterns Template Method and Strategy.

Innehåll

1 Inledning **1**

1.1 INCA - Informationsnätverk för Cancerforskning 3

1.2 Rapportstruktur och läsanvisningar 5

2 Bakgrund **7**

2.1 Inledning 7

2.2 Objektorienterad programmering 8

2.3 Mönster 10

2.3.1 Arkitektur och design, en subjektiv gräns? 11

2.3.2 Mönster är halvbakade 11

2.3.3 Planerad design med mönster 12

2.4 Omkonstruktion 12

2.4.1 Omkonstruktion och designskulder 13

2.5 Mönsterinförande omkonstruktion 13

2.6 Förändringsbarhet 15

2.7 Strukturproblem 23

2.7.1 Duplicerad kod 23

2.7.2 Utbredd villkorslogik 23

2.7.3 Långa metoder 24

2.7.4 Bristande inkapsling 25

2.8 Kategorisering av mönster 27
2.9 Mönster och kommunikation 29
2.10 Mönster, överdesign och underdesign 29
2.11 Mönster och inlärningstider 30
2.12 Klassnormalisering 32
2.13 Mönster, omkonstruktion och prestanda 33
2.14 Användningen av mönster i INCA 34
2.15 Kapitelsammanfattning 34

3 Strukturproblem och mjukvarumått 37

3.1 Inledning 37
3.2 Kända mjukvarumått 38
3.3 Duplicerad kod 39
3.3.1 Antal kodrader (LOC) 40
3.3.2 Antal satser 40
3.4 Villkorslogik 43
3.4.1 Cyklomatisk komplexitet 43
3.4.2 Kodblocksdjup 44
3.5 Långa metoder 45
3.5.1 Antal satser/metod 45
3.5.2 Antal metoder med fler än x satser 46
3.6 Bristande Inkapsling 46
3.6.1 Coupling 47
3.7 Kapitelsammanfattning 48

4 Strukturproblem och mönsterbaserade lösningar 49

4.1 Inledning 49
4.2 Duplicerad kod 50

4.2.1 Icke objektorienterade lösningar 50
4.2.2 Objektorienterade mönster 51
4.3 Villkorslogik 56
4.3.1 Icke objektorienterade lösningar 56
4.3.2 Objektorienterade mönster 56
4.4 Långa metoder 65
4.4.1 Icke objektorienterade lösningar 65
4.4.2 Objektorienterade mönster 66
4.5 Bristande inkapsling 66
4.5.1 Icke objektorienterade lösningar 66
4.5.2 Objektorienterade mönster 67
4.6 Kapitelsammanfattning 80

5 Mönsterbaserade tillämpningar i INCA 81

5.1 Inledning 81
5.2 FormView och TestaFormular 81
5.2.1 Beskrivning av delsystem och strukturproblem 81
5.2.2 Omkonstruktion: Steg 1 83
5.2.3 Omkonstruktion: Steg 2 85
5.2.4 Omkonstruktion: Steg 3 86
5.2.5 Omkonstruktion: Steg 4 89
5.3 Fler omkonstruktioner i INCA 91
5.4 Kapitelsammanfattning 91

6 Resultat och mätningar 93

6.1 Inledning 93
6.2 FormView och TestaFormular 93
6.2.1 Mjukvarumått för omkonstruktionen 93

6.2.2 INCA-gruppens tyckande om omkonstruktionen 96
6.2.3 Likheter med mönstret Bridge 97
6.3 Kapitelsammanfattning 97

7 Slutsats **99**
7.1 Författarens reflektioner 102
7.1.1 Är polymorfism som alternativ till villkorslogik tillräckligt betonat i undervisningen? 102
7.1.2 Är det lättare att lära sig mönster ur ett omkonstruktionsperspektiv? 103
7.1.3 Finns det utrymme för en kurs i omkonstruktion, mönster och testning? 104
7.2 Framtida arbete 106

Referenser **109**

A Mjukvarumätningsverktyg **113**
A.1 SourceMonitor 114
A.2 DevMetrics 115
A.3 Resource Standard Metrics (RSM) 116

B INCA **117**
B.1 Driftsmiljö 117
B.2 Skiktning och arkitekturbeskrivning 118

Ordförklaringar **119**

Förkortningar **121**

Objektorienterade Mönster **123**

Figurer

1.1 Beskrivning av examensarbete 2

2.1 Kostnaden för underhåll utgör 70 % av all mjukvaruutveckling 8

4.1 Exempel på Template Methods: GetObject och Load 52
4.2 Exempel på Template Methods: GetObject och Load 53
4.3 Inkapsling av sammansatta villkor 56
4.4 Uppdelning av villkorslogik med hjälp av beslutsmetod 57
4.5 Exempel på villkorslogik i en Front Controller utan Command 59
4.6 Front Controller med Command och Missing Object 61
4.7 Exempel där villkorslogik helt ersätts med polymorfism 63
4.8 Strukturen för State och Strategy liknar Command 64
4.9 Exempel på lagerindelade arkitekturer 68
4.10 Iteratorer kan kapsla in vad som itereras över och hur 74
4.11 Mönstret Decorator tillämpas för strömmar i .NET 77
4.12 Separated Interface gör affärslogiklagret oberoende av dataåtkomstlagret . 78
4.13 En tillämpning av mönstret Bridge. Källa: Ulf Bilting [10]. 79

5.1 Schematisk pseudokod för hur webbkontroller skapas i INCAs formulär . . 83
5.2 Omkonstruktion av FormView och TestaFormular 84
5.3 Delegerande anrop från FormView och TestaFormular 85
5.4 Delegerande anrop från FormView och TestaFormular 86

5.5 Mönstret Strategy ersätter villkorstester med polymorfism 87
5.6 Factory Method för att tvinga initiering av objektreferens 88

Tabeller

2.1 Indelning av designmönster enligt Gamma et al [24] 28

3.1 Mjukvarumått för hur omkonstruktioner påverkar strukturproblem 37
3.2 SATC Mjukvarumått för objektorienterade system 38
3.3 SATC Mål för mjukvarumått och effekter på kvalitetsaspekter 39
3.4 Gränsvärden för cyklomatisk komplexitet. Källa: SEI [38] 44

4.1 Exempel på mönster och de primära strukturproblem de angriper. 50

6.1 Mjukvarumått för delsystemet före och efter omkonstruktion 94

Kapitel 1

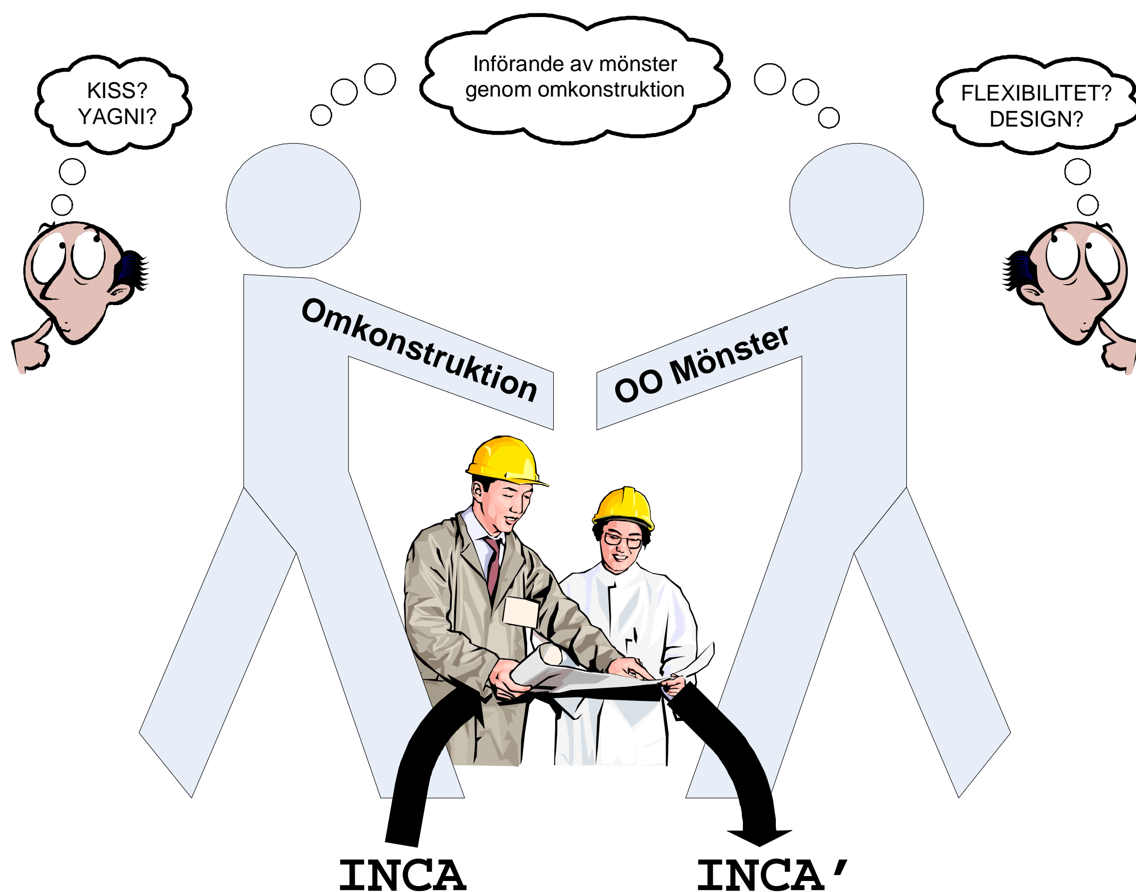
Inledning

Förändringsbarhet (eng. modifiability), användbarhet och prestanda är exempel på kvalitetsmål för mjukvaruutveckling [4]. Denna rapport fokuserar på hur förändringsbarhet kan uppnås i mjukvarusystem. För att utveckla förändringsbara system finns olika strategier:

- En strategi är att *försöka förutse* vilka ändringar ett system kommer att utsättas för och *planera* för förväntade ändringar genom att välja en design som gör systemet förändringsbart. Denna strategi kallas för *planerad* [21] design eller *"upfront"* [22] design.

- En annan strategi är att börja med den enklaste lösningen som fungerar och *gradvis förbättra* lösningen när möjligheter till förbättringar upptäcks under utvecklingens gång. Denna strategi kallas för *framväxande* (eng. evolutionary) [21] design och tillämpas bland annat av förespråkare för Extreme Programming (XP) [5].

Designmönster [24] och arkitekturmönster [23], i denna rapport kallade objektorienterade mönster, beskriver beprövade och eleganta lösningar på återkommande problem inom objektorienterad mjukvaruutveckling. Syftet med dessa mönster (eng. patterns) är att skapa flexibla, återanvändbara, utökningsbara och förändringsbara mjukvarusystem. En god



Figur 1.1: Beskrivning av examensarbete

kunskap om mönster är värdefull för planerad design, då mönster representerar beprövade lösningar som använts förr.

Vilken plats har då mönster i den strategi som kallas framväxande design? Förespråkare för framväxande design betonar vikten av kontinuerlig omkonstruktion (eng. refactoring) för att utveckla och underhålla en enkel design [5]. En omkonstruktion är en förändring av mjukvarans inre struktur för att göra den lättare att förstå och billigare att förändra, utan att förändra mjukvarans yttre beteende [22]. Förespråkare för framväxande design är inte nödvändigtvis avogt inställda till användningen av objektorienterade mönster. Då mönster tillämpas i en framväxande design i samband med omkonstruktion ligger inte

fokus på flexibilitet och återanvändbarhet. Mönster kan tillämpas som en lösning på ett befintligt strukturproblem. Denna rapport identifierar fyra mätbara strukturproblem och visar hur mönster kan tillämpas genom omkonstruktion för att reducera eller eliminiera dessa strukturproblem. De fyra strukturproblem som beskrivs är duplicerad kod, villkorslogik, långa metoder och bristande inkapsling. Som en fallstudie tillämpas en mönsterinförande omkonstruktion på ett delsystem i systemet INCA. Omkonstruktionen berör tre av de fyra strukturproblem som beskrivs i rapporten. Mjukvarumått används för att visa hur strukturproblemen påverkas av omkonstruktionen.

1.1 INCA - Informationsnätverk för Cancerforskning

INCA står för INformationsnätverk för CAncerforskningen och är ny nationell IT-plattform för hantering av nationella register kring cancerpatienter avseende vård och forskning. INCA utvecklas av IT-konsultföretaget Sogeti i Karlstad. Parallellt med skrivandet av denna examensrapport har klassdiagram för INCA tagits fram på önskemål av Sogeti. Innehållet i denna examensrapport ligger också till grund för en strukturanalys av INCA, där omkonstruktioner föreslås och diskuteras för att öka systemets förändringsbarhet. Följande text är hämtat från Sogetis webbplats [32] och ger en översikt av INCA (se ordförklaringar och förkortningar för ord i texten):

> "IT-plattformen kallad INCA (Informationsnätverk för cancerforskningen) är helt baserad på Microsoft-teknik. INCA är ny nationell IT-plattform för hantering av nationella register kring cancerpatienter avseende vård och forskning.
>
> INCA-systemet består av ett konstruktionsverktyg för att skapa och underhålla register och rollbaserade informationsformulär, samt att koppla dem samman genom dynamiskt konfigurerbara ärendeflöden.
>
> Tillämpningsdelen av INCA består av ett webbgränssnitt för inmatning, validering/monitorering samt datauttag för statistik. Validering av kvalitetsregistren

sker bland annat genom integration med regionala cancerregister (web services). Som teknisk grund för INCA ligger Microsoft .NET Framework. Lösningen är uppdelad i två delar – en Windows-applikation framtagen i C#.NET (konstruktionsverktyget) samt en webbapplikation framtagen i ASP.NET (tillämpningsdelen). Som webbserver används IIS och som databashanterare, SQL Server. För att knyta INCA till andra system inom vården nyttjas BizTalk Server som integrationsplattform.

IT-konsultföretaget Sogeti var det företag som efter en offentlig upphandling fick uppdraget att utveckla denna nya nationella plattform.

Stort affärsvärde

Insamling av cancerrelaterad data har hittills samlats i separata register och system inom varje OC, baserat på blanketter och formulär i pappersformat. Dessa register har utvecklats med hjälp av olika tekniska plattformar. Datat sammanfördes sedan en gång per år i ett nationellt register, i efterhand.

Genom den nya lösningen kan tidskrävande pappersexercis elimineras samt administrativa kostnader och kostnader för förvaltning av de många olika register och system som tidigare använts för insamling och bearbetning av data sänkas. Insamlingen av data ska i framtiden göras elektroniskt via webbformulär vilket konkret innebär att läkare på klinikerna och forskare runt om i landet snabbare och enklare kan få tillgång till informationen. Målet är att tillgängligheten till nationella data ska öka för både vården och forskningen så att verksamheten inom våra sex svenska OC, samt på klinikerna runt om i landet, kan bedrivas på ett mycket mer effektivt sätt än tidigare.

Kundnytta

Med INCA får kunden, det vill säga Sveriges sex OC, möjligheten att bygga upp en gemensam nationell databas. Med INCA undviker man pappersexercis

och hanteringen av insamling av cancerrelaterad data snabbas upp dramatiskt, vilket i sin tur kan ge ett bättre underlag för kundens kund, det vill säga landets alla kliniker och forskare, att utveckla behandlingsmetoder mot cancer i olika former. Med ett större och mer aktuellt underlag kan utvärderingen av exempelvis nya behandlingsmetoder för att bota olika former av cancer bli mer effektiv, vilket innebär att nyttan med INCA-lösningen berör både våra sex svenska OC, Sveriges alla kliniker samt samtliga patienter drabbade av cancer i Sverige.

I ett bredare perspektiv finns här ett generiskt konstruktionsverktyg, en plattform, som skulle kunna användas för att skapa en mängd olika registertillämpningar inom landstingsvärlden."

INCAs driftsmiljö, skiktning och arkitektur illustreras i bilaga B.

1.2 Rapportstruktur och läsanvisningar

För att förstå innehållet i denna rapport bör läsaren vara väl förtrogen med grundläggande begrepp inom objektorienterad programmering, såsom begreppen arv, inkapsling, polymorfism och virtuella metoder. Grundläggande förståelse för UML förutsätts. Läsaren rekommenderas att läsa igenom ordförklaringarna och förkortningarna som listas i slutet av denna rapport. Kapitlen bör läsas i den ordning de är listade, dvs. från kapitel 2 till kapitel 7. Rapporten är strukturerad på följande sätt:

- Kapitel 2: Bakgrund

  I detta kapitel introduceras begreppen mönster, omkonstruktion och förändringsbarhet. Fyra större strukturproblem beskrivs, för vilka omkonstruktioner är rekommenderade: Duplicerad kod, utbredd villkorslogik, långa metoder och bristande inkapsling. Införande av mönster genom omkonstruktion diskuteras och begreppet klassnormalisering berörs i korthet.

- Kapitel 3: Strukturproblem och mjukvarumått

  Utvalda mjukvarumått presenteras, vilka kan användas för att identifiera förekomster av de fyra strukturproblem som beskrevs i kapitel 2, samt för att mäta vilken effekt mönsterbaserade omkonstruktioner har på dessa strukturproblem.

- Kapitel 4: Strukturproblem och mönsterbaserade lösningar

  Mönster presenteras, vilka vart och ett kan tillämpas för att angripa ett eller flera av de fyra strukturproblem som beskrevs i kapitel 2.

- Kapitel 5: Mönsterbaserade tillämpningar i INCA

  En omkonstruktion av ett delsystem i INCA beskrivs, där mönstren Template Method och Strategy införs för att reducera tre av de strukturproblem som presenterats i kapitel 2: duplicerad kod, villkorslogik och långa metoder.

- Kapitel 6: Resultat och mätningar

  Mjukvarumått som presenterats i kapitel 3 används för att mäta effekterna av den omkonstruktion som beskrevs i kapitel 5.

- Kapitel 7: Slutsats

  Slutsatser av arbetet dras, författarens reflektioner presenteraras och förslag till fortsatt arbete ges.

- Ordförklaringar

  Förklaringar ges till vissa ord i rapporten, där ordens innebörd inte bedöms som självklar för en läsare som annars förväntas förstå rapportens innehåll.

- Förkortningar

  Fullständiga namn ges till de förkortningar som förekommer i rapporten.

- Objektorienterade Mönster

  De mönster som omnämns med eller utan beskrivning i rapporten, listas med namn och källa.

Kapitel 2

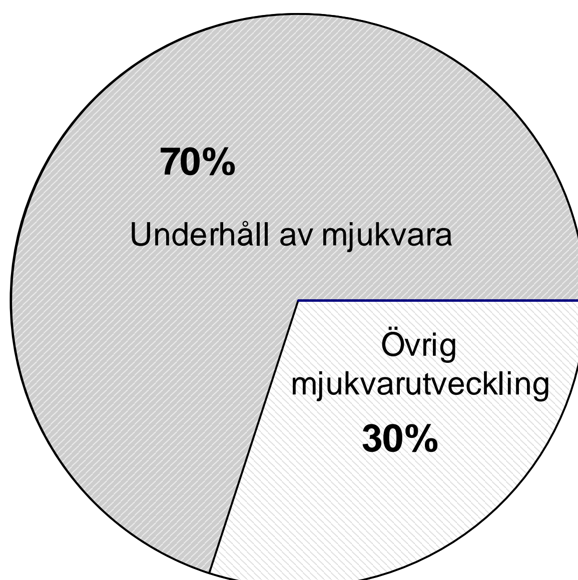
Bakgrund

2.1 Inledning

> "A design that doesn't take change into account risks major redesign in the future. Those changes might involve class redefinition and reimplementation, client modification, and retesting. Redesign affects many parts of the software system, and unanticipated changes are invariably expensive. Design patterns help you avoid this by ensuring that a system can change in specific ways."

Eric Gamma et al, *Design Patterns* [24]

Ett av målen för tillämpningen av objektorienterade mönster är att skapa förändringsbara mjukvarusystem. Långt ifrån alla utvecklare har nöjet att påbörja nyutveckling av system. Många utvecklare blir tvärtom involverade i mjukvaruprojekt som pågått under en längre tid. Då 70% av kostnaden för all mjukvaruutveckling uppskattas bestå av underhåll av mjukvarusystem [29] finns det anledning att fråga sig om och hur objektorienterade mönster kan införas i ett redan existerande system. Om objektorienterade mönster bara kan tillämpas i början av en designfas eller då nya tillägg görs i gamla system, då är tillämpbarheten för mönster starkt begränsad. Detta kapitel introducerar begreppen mön-



Figur 2.1: Kostnaden för underhåll utgör 70 % av all mjukvaruutveckling

ster, omkonstruktion och förändringsbarhet, samt diskuterar införande av mönster genom omkonstruktion.

2.2 Objektorienterad programmering

"If you don't use virtual functions, you don't understand OOP yet."

Bruce Eckel, *Thinking in C++* [17]

Under 1960- och 1970-talen var goto-eliminering och strukturerad programmering ett aktuellt diskussionsämne [14]. Bohm och Jacopini [12] visade att alla program kan byggas upp av tre kontrollstrukturer: sekvens, selektion och repetition. Nyckeln till framgångarna för strukturerad programmering var att programmen blev tydligare och lättare att felsöka och förändra [14].

Objektorienterad programmering (OOP) har också sina rötter från 1960-talet genom språket Simula 67, men det var först på 1980-talet som OOP användes i större omfattning

[14]. OOP betonar en programmeringsprincip som inte är unik för objektorienterad programmering, nämligen inkapsling (eng. encapsulation). Inkapsling av datastrukturer kan göras med hjälp av funktioner. OOP inför ytterligare en nivå av inkapsling där datastrukturer och metoder kan kapslas in i klasser. Här slutar tyvärr många att tala om inkapsling, enligt Shalloway och Trott [31]. Inkapsling likställs ofta med begreppen dataabstraktion och/eller informationsdöljande (eng. information hiding), ett misstag som lyfts fram och klargörs i en webbartikel av Edward V. Berard [9].

Utöver inkapsling tillför OOP två nya begrepp: arv och polymorfism [13]. Dessa begrepp används i alla böcker om OOP och borde därför vara bekant för alla som gått en grundkurs i objektorientering. Att känna till arv och polymorfism och kunna redogöra i detalj för hur virtuella metoder fungerar är inte liktydigt med att faktiskt använda virtuella metoder. Eckel [17] menar att en C-programmerare lär sig C++ i tre steg:

1. Först som ett bättre C-språk. Fel i ett C-program kan ofta hittas genom att kompilera det med en C++-kompilator.

2. Det andra steget är “objektbaserad” C++. Fördelar ses med att gruppera data och funktioner i en klass. Objektbaserad programmering kan även inbegripa enkla former av arv.

3. Eckel påpekar att det är lätt att stanna vid objektbaserad programmering och inte använda virtuella metoder, vilket är det tredje och sista steget en C-programmerare tar i processen att lära sig C++.

Eckel framhåller att de flesta egenskaper i C++ har en analogi i procedurella språk, medan virtuella metoder inte har någon analogi i procedurella språk. Funktionspekare då? Kanske finns vissa likheter, men Shalloway och Trott framhåller: “function pointers/delegates cannot retain state on a per-object basis” [31]. Fowler [22] berättar om två krav som framfördes från en av hans kunder:

- Ni måste använda Java
- Ni får inte använda objekt

Det är fullt möjligt att göra en övergång från ett procedurellt språk som C till ett objektorienterat språk som Java, utan att programmera vare sig objektbaserat eller objektorienterat. En klass är då bara ett sätt att paketera data och funktioner. Ett sätt att lära sig använda virtuella metoder och behärska objektorienterad programmering är att studera och tillämpa objektorienterade mönster.

2.3 Mönster

Objektorienterade mönster, både på arkitekturnivå [23] och designnivå [24] förmedlar kunskap om hur inkapsling, arv och polymorfism kan användas för att uppnå förändringsbara mjukvarusystem och återanvändbara lösningar på designproblem.

Både Fowler [22] och Gamma et al [24] citerar Cristopher Alexanders definition av ett mönster, vilken enligt dem inte bara gäller byggnader, utan också mjukvara:

> “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” [2]

Designmönster är inte begränsade till den uppsättning mönster som beskrivs av Gamma et al [24], även kallade “Gang of Four” (Gamma, Helm, Johnson and Vlissides). De skriver själva att deras mönsterkatalog inte är komplett. De arkitektur- och designmönster som beskrivs i kapitel 4 i denna rapport är hämtade från fler än en källa. Begreppet objektorienterade mönster används i rapporten för att representera både arkitektur- och designmönster.

2.3.1 Arkitektur och design, en subjektiv gräns?

Vad är då skillnaden mellan arkitektur och design? Fowler menar att det inte finns någon objektiv gräns mellan arkitektur och design, men ger ändå exempel på vad många anser vara kännetecknande för arkitektur [23]:

- “the highest-level breakdown of a system into its parts.”
- “decisions that are hard to change.”

Fowler gör ingen stor sak av vad som är arkitektur och vad som är design, eftersom gränsen är subjektiv. Han framhåller att vissa av hans beskrivna mönster rimligtvis kan kallas arkitekturmönster, medan andra handlar mer om design. “In the end architecture boils down to the important stuff — whatever that is” [23].

Ett sätt att fly undan begreppsdefinitioner och gränsen mellan designmönster och arkitekturmönster, är att kalla dem för objektorienterade mönster eller bara mönster. Alla mönster som beskrivs eller omnämns i rapporten finns listade med namn och källa i bilagan “Objektorienterade Mönster”.

2.3.2 Mönster är halvbakade

Fowler [23] menar att mönster är halvbakade, vilket går helt i linje med Alexanders [2] definition tidigare i detta avsnitt. Mönster kan enligt Fowler inte tillämpas blint. Mönster måste anpassas till sitt sammanhang. En av författarna till Design Patterns [24], John Vlissides, skriver [39] att ett mönsters strukturdiagram bara är ett exempel, inte en specifikation.

Då mönster inte utgör kompletta lösningar, utan lämnar mycket utrymme för kreativt tänkande, finns det anledning att fråga hur mycket som är ad-hoc i tillämpningen av mönster. Kan tillämpningen av mönster automatiseras? Utan att nämna exempel kallar Fowler “pattern tools” i allmänhet för miserabla misslyckanden. [23].

2.3.3 Planerad design med mönster

Gamma et al uppmanar sina läsare att försöka förutse vilka förändringar mjukvaran kommer att utsättas för och välja en design som gör mjukvarusystemet förändringsbart.

> "A design that doesn't take change into account risks major redesign in the future." [24]

För att undvika kostnader i samband med nya eller förändrade krav på ett mjukvarusystem, uppmanas systemutvecklare att planera för förändring genom att välja flexibla, anpassningsbara och förändringsbara designlösningar. Strategin att försöka förutse vilka ändringar som mjukvaran kommer att utsättas för, kritiseras av förespråkare för Extreme Programming (XP) [5], vilka betonar den alternativa designstrategin omkonstruktion (eng. refactoring).

2.4 Omkonstruktion

> "You Aren't Gonna Need It."
>
> Ron Jeffries et al, *Extreme Programming Installed* [25]

YAGNI (You Aren't Gonna Need It) är ett uttryck som används flitigt inom Extreme Programming. Uttrycket syftar främst på utveckling av programfunktionalitet som inte behövs för stunden, men som kanske kommer att behövas längre fram. Uttrycket syftar också på utveckling av lösningar som är mer flexibla än vad som krävs för stunden. Extreme Programming förespråkar en framväxande (eng. evolutionary) design, istället för en planerad design. En framväxande design åstadkoms genom att utvecklarna regelbundet omkonstruerar mjukvaran. Martin Fowler [22] definierar omkonstruktion (eng. refactoring) och dess motsvarande verbform på följande sätt:

> "Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior."
>
> "Refactor (verb): to restructure software by applying a series of refactorings without changing its observable behavior."

Definitionen av omkonstruktion omfattar ett stort antal aktiviteter. Omkonstruktion kan innebära att stora delar av ett delsystem skrivs om, men en omkonstruktion kan också utgöras av en så trivial uppgift som att byta namn på en variabel eller en metod. Då en design växer fram och underhålls genom omkonstruktion är det av största vikt att omkonstruktion är en regelbunden aktivitet, integrerad i utvecklingsprocessen.

2.4.1 Omkonstruktion och designskulder

Komplexitet som utgörs av otillräcklig design kan ses som designskulder [27]. Planerade flexibla designlösningar går ut på att förebygga att ett system belastas med designskulder. Omkonstruktion handlar istället om att betala av de designskulder som uppstår då systemet utvecklas. Små designskulder utgör inte någon stor risk för utvecklingen av ett system, men då omkonstruktion försummas växer designskulden. Om systemutveckling pågår under en längre tid utan att omkonstruktion görs, riskerar systemet att drabbas av allvarliga skulder. Den omkonstruktion som då krävs blir mer omfattande och riskfylld.

2.5 Mönsterinförande omkonstruktion

> "Our design patterns capture many of the structures that result from refactoring. Using these patterns early in the life of a design prevents later refactorings. But even if you don't see how to apply a pattern until after you've built your system, the patterns can still show you how to change it. Design patterns thus provide targets for your refactorings."

Erich Gamma et al, *Design Patterns* [24]

Mönster är generella designlösningar som förebygger utvecklingen av komplexitet i ett system. Omkonstruktion är en aktivitet som utförs efter att en mindre komplex lösning har identifierats för ett problem i ett system. Martin Fowler är författare till ledande litteratur om både mönster [23] och omkonstruktion [22]. De ideliga frågorna om hur mönster och XP går ihop, fick honom att skriva en artikel med en fråga som titel, "Is Design Dead?" [21]. Efter en längre diskussion svarar han bestämt nej på frågan och sammanfattar artikeln genom att lista vad som krävs för att upprätthålla en design enligt XP. Fowler medger att detta är en diger lista av krav och att XP inte gör mjukvarudesign lättare. Design enligt XP kräver [21]:

- En ständig strävan att hålla designen så tydlig och enkel som möjligt.
- En förmåga att kunna omkonstruera, så att förbättringar kan göras då behov uppstår.
- En god kunskap om mönster: inte bara om lösningarna, utan också när de ska användas och hur de kan införas.
- En design med en blick för framtida förändringar, i vetskap om att de beslut som tas nu, kommer att ändras senare.
- En förmåga att kunna förklara designen för dem som behöver förstå den, med hjälp av kod, diagram och framför allt, samtal.

Filosofin bakom omkonstruktion är att välja enkla lösningar före flexibla, men att välja de enkla lösningarna så att de lätt kan ersättas då behovet av mer flexibla lösningar uppstår [22]. Det är omöjligt att förutse alla ändringar som kommer att göras på den mjukvara vi utvecklar. Däremot kan vi förvänta oss att ändringar kommer att ske och vi kan planera för kontinuerlig omkonstruktion. Få system utvecklas rätt från början, kravspecifikationer är ofta ofullständiga, felaktiga och vilseledande [10].

Gamma et.al. [24] skriver att designmönster utgör mål för omkonstruktion. Kerievsky har tagit fasta på detta påstående och har skrivit en hel bok [27] om att införa designmönster genom omkonstruktion. Fowler [22] skriver:

> "There is a natural relation between patterns and refactorings. Patterns are where you want to be; refactorings are ways to get there from somewhere else."

Förhållningssättet att införa designmönster genom omkonstruktion (dvs. i efterhand) är motiverat för designmönster. Då arkitektur handlar om beslut som är svåra att ändra på [23], borde arkitekturmönster vara svårare att införa i efterhand, vilket i sin tur talar för att kunskap om arkitekturmönster i inledningen av ett projekt är mycket viktig. Fowler [23] skriver också:

> "Architectural refactoring is hard, and we're still ignorant of its full costs, but it isn't impossible."

2.6 Förändringsbarhet

Förespråkare för planerad design med mönster och förespråkare för framväxande design har ett gemensamt mål, förändringsbarhet. PerOlof Bengtsson [8] definierar ett mjukvarusystems förändringsbarhet (eng. modifiability) som den enkelhet med vilken ett mjukvarusystem kan förändras eller anpassas till ändringar i omgivningen, kraven eller funktionsbeskrivningen.

Stephen T. Albin [1] säger att ett system är förändringsbart om utökningar av systemets funktionalitet är mer kostnadseffektiva än att bygga ett nytt system. Hur kan förändringsbarhet mätas? Albin skriver:

> "The measure of modifiability is the cost and effort required to make a change to an application."

En mätning av förändringsbarhet innebär enligt Albin en mätning av tiden och kostnaden för verkställandet av systemets inkommande ändringskrav. Hur kan mätningar genomföras för att avgöra om en omkonstruktion resulterat i ett mer förändringsbart system? Ett sätt är att dela in systemutvecklare i två grupper, där den ena gruppen verkställer ändringskrav i ett omkonstruerat system och den andra gruppen verkställer samma ändringskrav i "samma" icke omkonstruerade system. Mätningar av tiden och kostnaden för de båda grupperna att verkställa samma ändringskrav kan sedan jämföras. Detta sätt att mäta förändringsbarhet är opraktiskt och dessutom förenat med störande element, såsom gruppmedlemmarnas skicklighet och deras förmåga att samarbeta. Om ändringarna i systemets två versioner utförs av samma person eller grupp störs mätningarna istället av att erfarenheten från genomförandet av ändringskravet i den ena versionen av systemet underlättar ändringen i den andra systemversionen.

Finns det något annat objektivt sätt att mäta eller uppskatta hur förändringsbarheten i ett system påverkas av en omkonstruktion? Om det finns objektiva indikationer som talar om att omkonstruktioner bör genomföras för att öka förändringsbarheten, då skulle mätningar kunna genomföras före och efter omkonstruktioner för att beräkna omkonstruktionens effekt på samma objektiva indikationer som motiverade omkonstruktionen.

Finns det då objektiva indikationer som talar om att omkonstruktioner bör genomföras eller är omkonstruktioner för ökad förändringsbarhet endast motiverade av systemutvecklarnas subjektiva tyckande? Fowler och Beck [22] menar att förståelsen för *när* omkonstruktioner bör genomföras och avslutas är lika viktig som kunskapen om *hur* omkonstruktioner genomförs. Det är dock svårare att tala om när en omkonstruktion bör genomföras än att tala om hur den kan genomföras. Omkonstruktioner motiveras av vissa typer av strukturer. Fowler och Beck beskriver dessa strukturer som dåliga lukter (eng. Bad Smells) och använder dem som indikatorer för behov av omkonstruktioner. De lukter som nämns är (om de sammanfattande beskrivningarna inte förstås, hoppa över dem tills vidare och läs igenom rapporten först):

- "Duplicated Code"

  Duplicerad kod anses vara den mest unkna stanken. Begreppet duplicerad kod representerar mer än exakta kopior av kodavsnitt i samma system. Duplicerad kod försvårar ändringar i systemet, vilket diskuteras mer senare.

- "Long Method"

  Ju längre en metod är, desto svårare är den att förstå. Korta metoder ökar läsbarheten och gör koden enklare att dela och återanvända.

- "Large Class"

  Stora klasser med mycket kod och många instansvariabler utgör en grogrund för duplicerad kod.

- "Long Parameter List"

  Parametrar är ett bättre alternativ än global data, men långa parameterlistor är svåra att läsa och måste underhållas då behovet av data ändras. Långa parameterlistor kan kortas ner genom inkapsling i objekt.

- "Divergent Change"

  Denna dåliga lukt uppstår då en typ av ändring inte kan genomföras i en enda tydlig punkt i systemet, utan omfattar flera metoder eller klasser.

- "Shotgun Surgery"

  Denna dåliga lukt uppstår då *en typ* av ändring i en klass leder till *andra typer* av ändringar i andra klasser.

- "Feature Envy"

  En metod vars kod är mer involverad i en annan klass än sin egen kan vara ett tecken på att metoden befinner sig i fel klass.

- "Data Clumps"

Relaterade data fält som förekommer i ett flertal klasser eller relaterade parametrar i ett flertal metoder är ofta tecken på att en ny klass bör skapas.

- "Primitive Obsession"

  Primitiva typer är programmerarens byggblock och dess användning nödvändig. Att skapa små klasser (som exempelvis Money och ZipCode) kan hjälpa till att sudda ut den tydliga linje som kan dras mellan primitiva typer och stora klasser.

- "Switch Statements"

  Att hantera variationer med switch-satser leder ofta till duplicerad villkorslogik. Switch-satser kan ersättas med polymorfism, vilket beskrivs senare i denna rapport. Att ersätta switch-satser som endast omfattar en enskild metod där inga förändringar förväntas är överarbete till liten nytta.

- "Parallel Inheritance Hierarchies"

  Denna lukt är en variant av "Shotgun surgery" och uppstår då införandet av en subklass tvingar fram skapandet av en ny klass i en annan arvshierarki.

- "Lazy Class"

  Ett ansvarsområde kan vara så litet att det inte bör tilldelas en egen klass.

- "Speculative Generality"

  Då utvecklare spekulerar om vad ett system kommer att behöva någon gång i framtiden, finns en risk att systemdesignen görs mer flexibel än den behöver vara. Om spekulationerna träffar rätt kan den flexibla designen betala för sig, men om inte finns strukturer vars komplexitet gör systemet onödigt svårt att förstå och underhålla.

- "Temporary Field"

  Temporära fält eller attribut i en klass läggs ibland till för att undvika att skicka

parametrar mellan metoder. De temporära fälten kan bara förstås om implementationen för enstaka metoder granskas. I alla andra sammanhang är det svårt att se varför de existerar i klassen. Fälten bör därför flyttas till en egen klass tillsammans med de metoder som använder dem.

- “Message Chains”

  Denna lukt uppstår då en klient ber ett objekt returnera ett annat objekt, som klienten sedan ber returnera ett annat objekt, osv. Klienten är involverad i andra klassers relationer till andra klasser. Delegering sker utan inkapsling.

- “Middle Man”

  Delegering och inkapsling kan överdrivas då objekt (av en “Middle Man”-klass) vidarebefordrar för mycket ansvar till objekt av andra klasser. Delegering är inte det enda sättet att utöka klassers beteende. Det kan också göras genom arv (även om delegering ofta är att föredra [24].

- “Inappropriate Intimacy”

  Då klasser är så intima att de använder varandras privata delar, är det dags att bryta förhållandet mellan dem. Om det finns gemensamma intressen finns det anledning att skapa en ny klass. Arv kan leda till att subklasser “känner till” för mycket om sina föräldrar, mer än föräldrarna “önskar”. Om så är fallet bör arv ersättas med delegering.

- “Alternative Classes with Different Interfaces”

  Denna lukt uppstår då två eller flera klasser existerar vars ansvarsområden är lika, men klassernas gränssnitt är olika. Denna lukt ger uttryck för en mer subtil kodduplicering.

- “Incomplete Library Class”

  Utvecklare av klassbibliotek kan inte förväntas förutse alla användningsområden för

sina klasser. Dåliga lukter uppstår lätt då klienter använder biblioteksklasser vars gränssnitt inte är komplett. De dåliga lukterna kan angripas antingen genom att klientklassen berikas med metoder som kompletterar hålen i biblioteksklasserna eller att biblioteksklasserna utökas genom arv.

- "Data Class"
  Dataklasser är klasser som endast innehåller enkla metoder för att hämta och spara data i objekt. Problemet är att sådana klasser blir hårt knutna till de klasser som använder dem.

- "Refused Bequest"
  Denna lukt uppstår då subklasser ärver data och metoder som inte används eller inte implementeras i subklasserna. Nio gånger av tio är denna lukt inte värd att angripa, men det kan ibland finnas anledning att få bort den genom att ersätta arv med delegering.

- "Comments" Kommentarer kan missbrukas som en "deodorant" för usel kod. Om ett kodavsnitt behöver förtydligas med kommentarer är det ofta ett tecken på att kodavsnittet bör flyttas till en egen metod med beskrivande namn. Då dåligt skriven kod skrivs om, blir kommentarer ofta överflödiga.

Fowler och Beck påstår att inga mjukvarumått finns som kan utmana en tränad människas intuition att avgöra när det är dags för en omkonstruktion i ett mjukvarusystem. De ger därför inga gränsvärden för hur lång en metod ska vara för att klassas som lång eller hur många parametrar som utgör en lång parameterlista. I Refactoring Workbook [40], skriver William Wake om ovan nämnda lukter:

> "The smells in this chapter are similar. They're dead easy to detect. They're objective (once you decide on a way to count and a maximum acceptable score). They're odious. And, they're common. You can think of these smells as being

caught by a software metric. Each metric tends to catch different aspects of why code isn't as good as it could be. Some metrics measure variants of code length; others try to measure the connections between methods or objects; others measure a distance from an ideal."

Omkonstruktion handlar om att göra mjukvara mer förändringsbar och de beskrivna dåliga lukterna eller strukturproblemen är exempel på faktorer som minskar mjukvarans förändringsbarhet och ökar dess komplexitet. Att lista alla faktorer som påverkar förändringsbarheten positivt eller negativt är ingen trivial uppgift. Att genomföra mätningar som tar hänsyn till alla förändringsbarhetsfaktorer är heller inte trivialt, om det överhuvudtaget är möjligt. Finns det faktorer som mer påtagligt än andra påverkar förändringsbarheten? Om så är fallet kan vi göra en avgränsning och endast ta hänsyn till ett urval av faktorer. Kerievsky [27] utökar Fowlers och Becks lista av dåliga lukter. Följande fyra strukturproblem är ett försök att med hjälp av deras beskrivningar peka ut de mest kritiska lukterna.

- Duplicerad kod

  Duplicerad kod anses vara den mest unkna stanken [22]. Borttagande av kodduplicering är också en aktivitet som Beck tycks använda nästan synonymt med omkonstruktion (refactoring). Beck skriver i Test-Driven Development By Example [6]:

  > Remember, the cycle is as follows.
  >
  > 1. Add a little test.
  > 2. Run all tests and fail.
  > 3. Make a little change.
  > 4. Run the tests and succeed.
  > 5. Refactor to remove duplication.

- Svårtolkad villkorslogik

  Svårtolkad villkorslogik är ett strukturproblem av sådan kaliber att det föreslagits att detta enda strukturproblem skulle kunna ligga till grund för mätning av programkomplexitet [28]. Att mätning av villkorslogik i metoder (se cyklomatisk komplexitet i avsnitt 3.4.1) inte ensamt är tillräckligt för att mäta komplexitet, är nu en åsikt som delas av många [19]. Att utbredd och svårtolkad villkorslogik ändå är ett allvarligt strukturproblem som ökar mjukvarans komplexitet står ändå klart [22]. Villkorslogik är en ingrediens i flera andra lukter. “Conditional Complexity” är en dålig lukt som beskrivs av Kerievsky [27]. “Switch Statements” [22] handlar också om villkorslogik för att styra programflödet.

- Långa metoder

  Långa metoder är ett strukturproblem som går hand i hand med kodduplicering och villkorslogik. Utbredd villkorslogik bidrar till långa metoder och långa metoder är också en grogrund för kodduplicering [27]. Ett överflöd av kommentarer i koden är ofta ett tecken på alltför långa metoder [22]. Långa metoder orsakar ofta försämrad läsbarhet [22].

- Bristande inkapsling

  Bristande inkapsling handlar om detaljnivåer och beroenden. Villkorslogik och långa stycken av kod är exempel på strukturproblem som tynger kodläsaren med mycket detaljer. “Primitive Obsession” och långa parameterlistor är andra exempel på strukturproblem som bidrar till mycket detaljer i koden [22]. “Inappropriate Intimacy” [22] är ett strukturproblem som skapar osunda beroenden i ett system. “Shotgun Surgery” [22] är ett strukturproblem som ger uttryck för oönskade beroenden i ett system. I grunden handlar dessa strukturproblem om bristande inkapsling.

2.7 Strukturproblem

2.7.1 Duplicerad kod

> "Kod som förekommer i samma eller nästan samma skepnad på flera ställen tyder på att den mest unkna av alla återanvändningsmetoder har använts: kopiera/klistra in"
>
> Ulf Bilting, *Designmönster för programmerare* [10]

Studier av Ducasse et.al. [16] visar att mellan 8% och 12% av industrimjukvara består av duplicerad kod. Duplicerad kod leder till system som är svårare att förstå och förändra [15]. Om kod finns duplicerad i ett system och ändringar inte görs på samtliga duplicerade kodblock, finns uppenbara risker att fel uppstår. Duplicerad kod är antingen uppenbar eller subtil [27] och kan utgöras av:

- Kodavsnitt som är exakt duplicerade
- Kodavsnitt som är duplicerade sånär som på några parametrar
- Strukturer som till ytan ser olika ut, men som i grunden är duplicerade

"Combinatorial Explosion" [27] är en dålig lukt som utgörs av en situation där flera kodblock utför samma sorts uppgift, men med olika typer eller mängder av data. "Oddball Solution" [27] är en liknande lukt som uppstår när det i ett och samma system existerar två olika sätt att lösa samma typ av problem. Avsnitt 4.2 ger exempel på hur objektorienterade mönster och virtuella metoder kan användas för att undvika duplicerad kod.

2.7.2 Utbredd villkorslogik

> "Conditional logic is innocent in its infancy, when it is simple to understand and contained within a few lines of code. Unfortunately, it rarely ages well."

Joshua Kerievsky, *Refactoring to Patterns* [27]

Villkorslogik är en nödvändig ingrediens i de flesta system där komplexiteten överträffar ett "Hello World"-program. Enstaka villkorssatser i en metod utgör oftast inget strukturproblem, men då antalet villkorssatser ökar kan de bidra till svårläst och svårunderhållen kod. Enligt Fowler [22] är komplex villkorslogik en av de vanligaste typerna av komplexitet i ett program. I takt med att villkorslogik läggs till, växer också den metod i vilken logiken finns och även om det går att följa vad som händer är det lätt att tappa bort sammanhanget. Avsnitt 4.3 ger exempel på hur objektorienterade mönster och virtuella metoder kan användas för att styra programflödet mellan basklasser och subklasser.

2.7.3 Långa metoder

> "Systems that have a majority of small methods tend to be easier to extend and maintain because they're easier to understand and contain less duplication."

Joshua Kerievsky, *Refactoring to Patterns* [27]

Fowler [22] menar att de objektorienterade program som lever längst har korta metoder. Korta metoder ger ökade förutsättningar att dela kod. Korta metoder ökar läsbarheten och gör det lättare att förstå koden. Fowler framhåller att de som är inte är vana vid objektorientering ofta upplever att objektorienterade program utgörs av en oändlig sekvens av delegerande anrop. De korta metoderna och de delegerande anropen ger också upplevelsen att inga beräkningar görs.

Korta metoder kan upplevas besvärande vid felsökning, då programflödet hoppar mellan metoderna. En bra namnsättning på metoder minskar dock behovet av att granska kodinnehållet i metoderna. Om metodnamnen kommunicerar vad metoderna gör, behövs ingen regelbunden granskning av metodernas innehåll.

Långa metoder är enligt Fowler och Beck [22] en grogrund för kodduplicering. Långa metoder är ofta ett direkt resultat av utbredd villkorslogik. Motmedlet mot långa metoder

är oftast att bryta ut kod ur den långa metoden till nya korta metoder. Avsnitt 4.4 ger exempel på objektorienterade mönster som kan användas i vissa situationer där långa metoder existerar.

2.7.4 Bristande inkapsling

> "Find what varies and encapsulate it."
>
> Alan Shalloway och James Trott, *Design Patterns Explained* [31]

Shalloway och Trott förklarar att inkapsling (eng. encapsulation) är mer än att bara gömma klassers interna struktur bakom ett gränssnitt av metoder. De beklagar att många felaktigt likställer inkapsling med informationsdöljande (eng. information hiding). Blaha och Rumbaugh [11] definierar t ex inkapsling på följande sätt:

> "Encapsulation (also information hiding) separates the external aspects of an object, that are accessible to other objects, from the internal implementation details, that are hidden from other objects."

Inkapsling är enligt Shalloway och Trott ett vidare begrepp än informationsdöljande. Inkapsling handlar om att reducera beroenden mellan delsystem, vilket i sin tur förebygger spridningseffekter (eng. ripple effects) mellan delsystemen då förändringar görs på datastrukturer och algoritmer i ett delsystem. Inkapsling eller informationsdöljande av datamedlemmar i en klass är bara en nivå av inkapsling, som görs för att klasser inte ska bli beroende av varandras interna datastruktur. Inkapsling kan göras på olika nivåer. Metoder kapslar in data och datastrukturer som används lokalt i metoderna. Klasser kapslar in både data och metoder. Klasser kapslas ofta in i paket, moduler eller bibliotek. I sin fallstudie om utformningen av en ordbehandlare, ger Gamma et al [24] exempel på ett flertal typer av inkapslingar:

- Inkapsling av formateringsalgoritm

- Inkapsling av implementationsberoenden
- Inkapsling av användaråtgärder (eng. request)
- Inkapsling av åtkomst och traversering (eng. traversal)
- Inkapsling av textanalys

Observera hur begreppet inkapsling används av Gamma et al för att beteckna inkapsling av koncept, snarare än som ett sätt att dölja den interna strukturen i en klass. Dessa koncept kan vara tillämpningsspecifika som t ex formateringsalgoritm eller mer generella som t ex implementationsberoenden. Inkapslingen görs för att kunna tillåta variationer inom det inkapslade konceptet, utan att de delsystem som är beroende av konceptet blir beroende av dess variationer. Ett delsystem som t ex är beroende av att kunna formatera, ska inte vara beroende av implementationen av formateringsalgoritmen. En inkapsling av formateringsalgoritmen gör att algoritmen kan ändras utan att externa delsystem påverkas. Inkapsling är ett centralt begrepp inom objektorienterad programmering och är relaterat med begreppet abstraktion. Blaha och Rumbaugh [11] definierar abstraktion på följande sätt:

> "Abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant."

Inkapsling av ett delsystem reducerar externa delsystems beroende av det inkapslade delsystemets inre struktur. Abstraktion innebär att en datamodell endast innehåller en begränsad mängd egenskaper från det som modelleras. Resterande egenskaper är inte inkapslade av datamodellen. De har helt utelämnats från datamodellen.

När en ändring genomförs är det önskvärt att ändringen kan genomföras i en enda tydlig punkt i systemet [22]. Förändringar försvåras om de strukturer som ska ändras är utspridda

i olika delsystem. "Localize Modifications" [4] är en designstrategi där ansvarsområdena för varje delsystem anpassas till förväntade ändringskrav, i syfte att varje ändringskrav ska påverka så få delsystem som möjligt.

Vidare, när ändringskrav verkställs för berörda delsystem är det önskvärt att ändringen inte indirekt påverkar andra delsystem. Att ett delsystem B indirekt påverkas av en genomförd ändring i delsystem A, innebär att ändringen i A har en spridningseffekt. Spridningseffekten från delsystem A till delsystem B är då ett direkt resultat av bristande inkapsling. Delsystem B är beroende av delsystem A:s inre struktur. Bengtssons [8] erfarenhet är att spridningseffekter är svåra att identifiera. Avsnitt 4.5 ger exempel på hur objektorienterade mönster kan användas för att kapsla in datastrukturer och delsystem på olika nivåer.

2.8 Kategorisering av mönster

Gamma et al [24] delar in alla sina designmönster i två dimensioner (se tabell 2.1). Ett mönsters syfte (eng. purpose) beskriver vad mönstret gör och dess omfång (eng. scope) anger om mönstrets primära tillämpning är knutet till klasser eller objekt. Skapelsemönster (eng. creational patterns) är inriktade på skapandet av objekt. Strukturmönster (eng. structural patterns) är inriktade på hur klasser eller objekt sätts samman och Beteendemönster (eng. behavioral patterns) handlar om samspel och ansvarsfördelning mellan objekt.

Gamma et al menar att det finns många sätt att organisera designmönster och att ytterligare indelningar av mönstren kan fördjupa förståelsen för vad de gör, hur de skiljer sig åt och när de ska tillämpas. Då mönster införs genom omkonstruktion används mönstren för att komma tillrätta med strukturproblem. Kerievsky [27] menar att beskrivningen av ett designmönsters avsikt (eng. intent) är till större hjälp om beskrivningen fokuserar på vilken typ av designproblem mönstret hjälper till att lösa. Beskrivningarna av avsikterna med Template Method och State, lyder enligt Gamma et al [24]:

> "Define the skeleton of an algorithm in an operation, deferring some steps to

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Tabell 2.1: Indelning av designmönster enligt Gamma et al [24]

subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure." (Template Method)

"Allow an object to alter its behavior when its internal state changes. The object will appear to change its class." (State)

Kerievsky menar att många programmerare har svårt att hitta tillämpningar för designmönster pga. att de utgår från de beskrivna avsikterna för mönstren, istället för att fokusera på mönstrens tillämpningsområden. Template Method är ett användbart mönster för att reducera eller ta bort duplicerad kod i subklasser inom samma klasshierarki. State förenklar komplicerad villkorslogik som involverar tillståndsändringar. En indelning av mönster behöver inte innebära att varje mönster ingår i exakt en kategori. Notera att mönstret Adapter av sina upphovsmän placerats i två kategorier (tabell 2.1). De objektorienterade mönster som tas upp i kapitel 4 beskrivs utifrån de strukturproblem de löser. Då ett flertal mönster är tillämpbara för mer än ett strukturproblem kan beskrivningen av ett mönster vara uppdelad i olika avsnitt.

2.9 Mönster och kommunikation

Språk underlättar kommunikation. Språk reducerar behovet av att peka och göra underliga ljud. Språk låter oss förmedla abstrakta tankar till andra människor, förutsatt att de vi kommunicerar med förstår språket. Om en grupp människor som talar olika språk vill samarbeta för att lösa en uppgift är det en god idé om gruppen först enas om ett arbetsspråk och sedan lär sig språket tillräckligt väl för att kunna dela sina tankar. Mönster är som tidigare nämnts "en namngiven pedagogisk lösning på ett i en viss omgivning återkommande problem" [10]. Förutom att mönstren är bärare av lösningar på återkommande problem, bär de också ett namn som underlättar kommunikationen mellan systemutvecklare. En hel del information om ett mjukvarusystem kan förmedlas genom att tala om vilka mönster som finns i systemet. Den som läst Fowlers bok om arkitekturmönster [23] kan t ex delges mycket information genom att få reda på att ett mjukvarusystems dataåtkomstlager är uppbyggt av ett antal Data Mappers [23] och att all kommunikation med databasen är inkapslat bakom ett frågespråk baserat på mönstret Query Object [23]. Den som inte hört talas om de båda mönstren blir inte mycket klokare av informationen. Den som däremot känner till namnen på mönstren och kan redogöra för hur de tillämpas, kan med hjälp av en enda mening skapa sig en ganska god uppfattning om hur ett delsystem är uppbyggt, utan att granska vare sig kod eller diagram.

2.10 Mönster, överdesign och underdesign

> "Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
>
> Martin Fowler, *Refactoring* [22]

Kerievsky [27] säger att om det krävs insatta personer för att förklara vad programkoden gör, är det dags för omkonstruktion. Att förtydliga dåligt skriven kod med hjälp av

kommentarer är som att dölja en redan dålig lukt med deodorant. Syftet med kontinuerlig omkonstruktion är att göra koden lättare att förstå, så att koden blir lättare att utöka, förändra och underhålla.

Kod som inte kommunicerar vad den gör kan vara ett resultat av underdesign, men otydlig kod kan också bero på motsatsen, överdesign. Användningen av mönster är inte liktydigt med bra design. Då mönster ses som eleganta lösningar vars tillämpningar i ett system höjer dess kvalité, finns en risk att mönster införs för mönstrens egen skull eller att mönster införs av programmerare som känner ett behov av att visa sina mönsterkunskaper. Kerievsky [27] menar att sådana programmerare är “patterns happy”. Mönster är till för att reducera systemkomplexitet. Om den komplexitet som ett infört mönster för med sig inte överskuggar den komplexitet det är ämnat att förebygga eller reducera, finns det anledning att genomföra en omkonstruktion som tar bort mönstret.

2.11 Mönster och inlärningstider

Alla ändringar som ska genomföras i ett system är förenade med en ställtid för att sätta sig in i det sammanhang där ändringen ska genomföras. Sammanhangets komplexitet, såsom den upplevs av den som genomför ändringen, påverkar direkt ställtidens längd. Systemutvecklare är väl medvetna om att även om de genomför en lyckad ändring i ett komplext kodavsnitt idag, kommer sammanhangets komplexitet att göra sig påmint då framtida ändringar ska genomföras i samma kodavsnitt. Ställtiden kanske minskar till nästa gång, men den försvinner inte helt, eftersom vi glömmer en del av det vi lär oss. Att införa mönster i ett system underhållet av en grupp utvecklare som inte känner till de mönster som införs, kan innebära att utvecklarna upplever att komplexiteten ökar. Den tid det tar att lära sig ett mönster varierar från person till person. En intressant jämförelse skulle vara att jämföra ställtiden för att genomföra en ändring i ett komplext delsystem med inlärningstiden för att lära sig ett mönster som genom omkonstruktion införts i samma

delsystem.

På Karlstads universitet ges en kurs i objektorienterade designmetoder (OODM) [37] med omfattningen 5 högskolepoäng. 1 poäng av kursen tillägnas diskussion av designmönster i seminarieform och 1 poäng tillägnas laborationer där designmönster tillämpas. Då varje högskolepoäng motsvarar 40 timmars studier är en grov uppskattning att 80 timmar tillägnas inlärning av designmönster. Varje student förväntas lära sig nio av GOF-mönstren [24] tillräckligt väl för att inför en grupp kunna presentera mönstrens struktur och syfte, samt diskutera deras tillämpning. Grundläggande kunskap om de övriga 14 mönstren ingår i kursen. Varje seminarietillfälle då mönster diskuteras pågår i två timmar och omfattar presentation och diskussion av ett eller två mönster. Ett enda seminarium tillägnas åt diskussion av de 14 mindre prioriterade mönstren. De nio mönster som diskuteras mer ingående (än övriga 14) är:

- Abstract Factory
- Factory Method
- Singleton
- Adapter
- Composite
- Decorator
- Iterator
- Observer
- Template Method

Valet av dessa nio mönster inkluderar sju av de åtta mönster som Gamma et al [24] anser tillhör de enklaste och vanligaste designmönstren. De nio mönster som studeras i OODM

inkluderar Singleton och Iterator, vilka inte räknas till de åtta enklaste och vanligaste. Det enkla och vanliga designmönster som inte studeras ingående i OODM är mönstret Strategy. 60 högskolepoäng inom datavetenskap är ett förkunskapskrav till kursen, vilket ger en ungefärlig uppskattning av studenternas kunskapsnivå då mönstren studeras.

2.12 Klassnormalisering

> "Class normalization is a process by which you reorganize the structure of your object schema in such a way as to increase the cohesion of classes while minimizing the coupling between them."
>
> Scott W. Ambler, *Introduction to Class Normalization* [3]

Normalisering av data ett känt begrepp inom databasdesign. Ambler menar att normalisering också är tillämpbart på objektorienterad design i form av klassnormalisering. Klassnormalisering omfattar dock normalisering av både data och beteende. Klassnormalisering är ett begrepp som är relaterat till designmönster och omkonstruktion. Ambler påpekar att tillämpningen av designmönster ofta leder till starkt normaliserad objektdesign, samt att omkonstruktion ofta tillämpas då klassnormalisering utförs. Omkonstruktion, enligt Ambler, är något som utförs på källkod, medan klassnormalisering tillämpas på modeller. Ambler menar också att klassnormalisering som begrepp är ett sätt att överbrygga det gap som ofta finns mellan dem som förstår objektdesign och de som förstår databasdesign. Ambler definierar objektnormalformer (ONF) på följande sätt [3]:

- 1ONF

  En klass är i 1ONF då det beteende som krävs av ett sammansatt attribut är inkapslat i en egen klass. En objektdesign (eng. object schema) är i 1ONF då alla dess klasser är i 1ONF. Ett sammansatt attribut är ett attribut som representerar en samling av attribut, en samling som kan delas upp i separata attribut och flyttas till en egen klass tillsammans med de metoder som använder attributen.

- 2ONF: En klass är i 2ONF när den är i 1ONF och när delat beteende som behövs av fler än en instans av klassen är inkapslat i en egen klass. En objektdesign är i 2ONF då alla dess klasser är i 2ONF. Med den definitionen menar Ambler att värden på attribut inte ska vara duplicerade i klassens instanser, så att en ändring av attributvärden måste göras i flera objekt.

- 3ONF: En klass är i 3ONF när den är i 2ONF och när den bara kapslar in ett ansvarsområde ("one set of cohesive behaviors"). En objektdesign är i 3ONF då alla dess klasser är i 3ONF. Vad som anses representera ett ansvarsområde är inte alltid uppenbart. Ambler menar t ex att hantering av datumintervall är ett eget ansvarsområde som motiverar en egen klass DateRange.

2.13 Mönster, omkonstruktion och prestanda

I inledningen till denna rapport nämndes prestanda som ett kvalitetsmål, vid sidan av förändringsbarhet. Prestanda är ett kvalitetsmål som ibland blir lidande av omkonstruktioner, då fokus för omkonstruktion är förändringsbarhet och enkel design. Prestandaoptimeringar inför ofta komplexitet till ett system, som gör systemet svårare att arbeta med. Fowlers [22] råd är att prioritera enkel design och att göra prestandaoptimeringar i ett senare skede av systemutvecklingen. De flesta mönster som beskrivs av Fowler har som mål att reducera komplexitet, men det finns också mönster som har en direkt koppling till prestanda och hantering av systemresurser. Mönstret Lazy Load [23] är ett exempel på ett sådant mönster, där återskapandet av objekt från lagrad data senareläggs till den tidpunkt då objektets data verkligen behövs. Beskrivning av mönster för effektiv hantering av systemresurser ligger utanför ramarna för detta examensarbete och kommer därför inte att tas upp.

2.14 Användningen av mönster i INCA

Flera kända mönster kan identifieras i INCA, både designmönster beskrivna av Gamma et al [24] och mönster beskrivna av Fowler [23]. INCA använder t ex mönstren Proxy [24], Iterator [24], Template Method [24], (en variant av) Query Object [23], Optimistic Offline Lock [23] och Active Record [23]. Denna rapport beskriver inte hur mönster som dessa har tillämpats i INCA, utan beskriver generellt (i kapitel 4) hur mönster kan införas i mjukvarusystem för att komma tillrätta med strukturproblem i form av kodduplicering, villkorslogik, långa metoder och bristande inkapsling. För att visa hur mönster kan införas genom omkonstruktion har ett delsystem i INCA omkonstruerats. Omkonstruktionen beskrivs steg för steg i kapitel 5. För att mäta vilken effekt omkonstruktionen har på de strukturproblem som angrips, identifieras lämpliga mjukvarumått i kapitel 3, vilka sedan tillämpas på den genomförda omkonstruktionen. Resultatet av mätningen presenteras och kommenteras i kapitel 6.

2.15 Kapitelsammanfattning

Omkonstruktion och tillämpningen av objektorienterade mönster är två olika förhållningssätt till objektorienterad mjukvarudesign. Mönster kan användas för att bedriva en planerad design, där utvecklare försöker förutse kommande ändringar i systemet och investerar i designlösningar som gör systemet mer förändringsbart. Omkonstruktion möjliggör en framväxande design, där utvecklare inte spekulerar om kommande ändringar, utan fokuserar på att hålla mjukvarusystemets inre struktur fri från sådana strukturproblem som gör systemet svårare att förstå och dyrare att förändra. Omkonstruktion kan ses som en avbetalning av designskulder, medan en planerad design med mönster är en designinvestering. Omkonstruktion och tillämpningen av mönster har ett gemensamt mål att öka förändringsbarheten i mjukvarusystem. Istället för att välja mellan omkonstruktion och mönster, kan de båda förhållningssätten förenas genom att mönster införs genom omkonstruktion. In-

förande av mönster genom omkonstruktion är ett förhållningssätt till mönster med fokus på att angripa och stoppa utvecklingen av existerande strukturproblem i systemet, såsom kodduplicering, villkorslogik, långa metoder och bristande inkapsling.

Kapitel 3

Strukturproblem och mjukvarumått

3.1 Inledning

Strukturproblem	**Mjukvarumått**
Duplicerad kod	Antal kodrader (LOC) Antal satser
Villkorslogik	Cyklomatisk komplexitet Kodblocksdjup
Långa metoder	Antal satser / metod Antal metoder med fler än X satser
Bristande inkapsling	Coupling

Tabell 3.1: Mjukvarumått för hur omkonstruktioner påverkar strukturproblem

Avsnitt 2.7 identifierade och beskrev strukturproblemen kodduplicering, villkorslogik, långa metoder och bristande inkapsling. För att mäta vilken effekt omkonstruktioner har på dessa fyra strukturproblem identifieras och beskrivs i detta kapitel lämpliga mjukvarumått (tabell 3.1). Att genomföra mätningar på programkod kan effektiviseras genom att använda mjukvarumätningsverktyg. Tre verktyg har utvärderats i samband med att mjukvarumått bestämts för de strukturproblem som beskrivs i detta examensarbete:

- SourceMonitor Version 2.2

  Kan användas fritt.

- Resource Standard Metrics (RSM) Version 6.92

  Kan användas fritt på upp till 10 filer.

- devMetrics Community Edition 2.0

  Kan användas fritt. Integreras i Visual Studio .NET. Endast .NET 1.0 och 1.1.

Användningen av dessa tre verktyg diskuteras i detta kapitel. Bilaga A innehåller skärmbilder som illustrerar hur dessa verktyg används (NUnits källkod mäts).

3.2 Kända mjukvarumått

Source	**Metric**	**OO Construct**
Traditional	Cyclomatic complexity (CC)	Method
Traditional	Lines of Code (LOC)	Method
Traditional	Comment percentage (CP)	Method
NEW Object-Oriented	Weighted methods per class (WMC)	Class/Method
NEW Object-Oriented	Response for a class (RFC)	Class/Message
NEW Object-Oriented	Lack of cohesion of methods (LCOM)	Class/Cohesion
NEW Object-Oriented	Coupling between objects (CBO)	Coupling
NEW Object-Oriented	Depth of inheritance tree (DIT)	Inheritance
NEW Object-Oriented	Number of children (NOC)	Inheritance

Tabell 3.2: SATC Mjukvarumått för objektorienterade system

Det finns ett flertal kända mjukvarumått för att mäta kvalitetsaspekter av objektorienterad mjukvara. Software Assurance Technology Center (SATC) [30] vid NASA använder nio mjukvarumått för objektorienterade system (tabell 3.2). De sex OO-specifika mått som SATC använder är exakt samma mått som föreslås av Fenton och Pfleeger [19]. Då de nio måtten valdes av SATC var det ett krav att måtten gick att beräkna med ett verktyg. Tabell 3.3 visar översiktligt hur SATC tolkar värdena för de nio måtten. Ett lågt värde för CC är bra och gör det lättare att testa systemet, ökar förståelsen, etc.

Metric	Objective	Testing Effort	Understand-ability	Maintain-ability	Develop Effort	Reuse
CC	Low	Low	High	High		
LOC	Low	Low	High	High		
CP	High	Low	High	High	Low	
WMC	Low			High	Low	
RFC	Low	Low				
LCOM	Low		High	High	Low	High
CBO	Low	Low	High	High		High
DIT	Low	Low	High	High		High
NOC	Low	Low	High	High		High

Tabell 3.3: SATC Mål för mjukvarumått och effekter på kvalitetsaspekter

3.3 Duplicerad kod

Ett sätt att mäta exakt duplicerad kod i ett system är att räkna antalet exakt duplicerade satser. Att mäta duplicering av kod som är nästan identisk, kräver mer analysarbete. Den kodduplicering som är svårast att mäta är den som består av strukturer som till ytan är olika, men i grunden är duplicerade. Antal kodrader (LOC) är ett mått som ibland används för att mäta produktivitet. Att mäta produktivitet med hjälp av enheten LOC kan starkt ifrågasättas [20]. LOC är dock ett användbart mått som indikator för kodduplicering:

> “Any good developer knows that they can code the same stuff with huge variations in lines of code, furthermore code that’s well designed and factored will be shorter because it eliminates the duplication. Copy and paste programming leads to high LOC counts and poor design because it breeds duplication. You can prove this to yourself if you go at a program with a refactoring tool that supports Inline Method. Just using that on common routines should allow you to easy double the LOC count.”

Martin Fowler, *[www] CannotMeasureProductivity* [20]

En reducering av kodduplicering innebär att duplicerad kod tas bort. För att åstadkomma kodduplliceringen tillkommer ny kod för att t ex dela tidigare duplicerad kod i nya metoder. En reducering av enstaka duplicerade kodrader kan innebära att det totala antalet kodrader ökar vid reduceringen, men då duplicerad kod ofta utgörs av kodavsnitt med flera kodrader är det rimligt att anta att det totala antalet kodrader minskar i ett system efter att duplicerad kod tagits bort. En grov uppskattning av hur duplicerad kod reduceras genom en omkonstruktion kan då göras genom att räkna antalet kodrader före och efter en omkonstruktion.

3.3.1 Antal kodrader (LOC)

För att räkna antal kodrader är det viktigt att definiera vilka kodrader som räknas. Ska kommentarer i koden räknas? Blankrader? Rader som endast består av en måsvinge (klammerparentes)? Att inkludera kommentarer är nödvändigt för att beräkna hur stor andel av koden som består av kommentarer. Kommentarer kan ibland utgöra en deodorant för dåligt skriven kod [22]. Det kan finnas en poäng med att inkludera kommentarer, men då kodrader räknas för att indikera en reducering av kodduplicering, kan inkluderandet av kommentarer och blankrader utgöra ett för mätningen störande element. För omkonstruktionerna i detta examensarbete exkluderas kommentarer och blankrader, då kodrader räknas för det delsystem som omkonstrueras. Ett mätverktyg som exkluderar kommentarer och blankrader vid beräkning av antal kodrader är RSM (Resource Standard Metrics). RSM används i detta examensarbete för att beräkna LOC.

3.3.2 Antal satser

Ett alternativt sätt att mäta längd på kod är att räkna antal satser. Antal kodrader kan reduceras genom metodanropskomposition. Koden i metoden GetObject i BusinessObjectMapper (figur 4.2) innehåller fem rader. Antal kodrader i GetObject kan variera mellan en

till sex rader (plus måsvingar och metoddefinition), beroende på graden av metodanropskomposition.

Det är önskvärt att presentera ett mjukvarumått på kodlängd som inte påverkas av graden av metodanropskomposition eller av huruvida måsvingar skrivs på samma rad eller inte. Satser borde vara ett sådant mått. Precis som med kodrader är det viktigt att definiera vad som räknas som en sats. DevMetrics är ett verktyg som räknar satser (eng. statements). SourceMonitor är ett annat verktyg som räknar satser. RSM räknar (utöver LOC) effektiva kodrader (eLOC) och logiska kodrader (lLOC). Dessa tre mätverktyg ger olika resultat för dessa mått. Om metoden GetObject i BusinessObjectMapper (figur 4.2) skrivs om med maximal grad av metodanropskomposition (en enda rad), anger SourceMonitor att metoden har en sats. DevMetrics anger två satser. RSM anger eLOC = 1 och lLOC = 1. SourceMonitor tillhandahåller ett annat mått, nämligen antal anrop i metoden. Antalet anrop mäts korrekt till sex anrop. Att antalet anrop i metoder är känt kan dock inte användas för att bestämma antalet satser i metoder. Då inget fritt verktyg hittats som räknar antalet satser på ett tillfredsställande sätt, har SourceMonitor valts som satsräknare i detta examensarbete. Att räkna satser med SourceMonitor ställer krav på att graden av metodanropskomposition är ungefär lika före och efter omkonstruktionerna, samt att samma kodstandard följs (för måsvingar, etc). Hjälpen till SourceMonitor ger tydlig information om vad som räknas som satser:

> "in C#, computational statements are terminated with a semicolon character. Branches such as if, for and while are also counted as statements. Methods, though not terminated by semicolons, are counted as statements. All attributes are counted as statements as well, though calls inside attributes are ignored. The exception control statements try, catch, and finally are also counted as statements. Preprocessor directives #define and #undef are counted as statements. All other preprocessor directives are ignored. In addition all statements between each #else or #elif statement and its closing #endif statement are

ignored, to eliminate fractured block structures. Goto labels are also counted as statements." (Källa: SourceMonitor)

Det kan diskuteras om namnrymdsdeklarationer i C# bör räknas som satser. Att räkna namnrymdsdeklarationer kan ha betydande påverkan då en omkonstruktion resulterar i många nya klasser. SourceMonitor inkluderar namnrymdsdeklarationer som satser. SourceMonitor har ett separat mått för antal satser/metod i en klass eller grupp av klasser. SourceMonitor räknar också antal klasser och antal metoder/klass. Därigenom kan det totala antalet satser i metoder enkelt beräknas som (# satser/metod) * (# metoder/klass) * # klasser. Det totala antalet satser i metoder är särskilt intressant för att påvisa en reducering av duplicerad kod, då det finns en overhead associerad med införandet av nya metoder. Denna overhead består metoddefinitionen och två måsvingar. Då två duplicerade kodblock ersätts med anrop till en gemensam metod, reduceras kodduplliceringen, med det totala antalet kodrader eller satser reduceras enligt formeln:

(Totalt # kodrader)' =
Totalt # kodrader - kodduplliceringsreducering + overhead för ny metod.

Antal satser/metod påverkas inte av någon overhead för nya metoder, förutom av själva anropen som ersätter de duplicerade kodblocken.

För att indikera en reducering av duplicerad kod presenteras för omkonstruktionerna i kapitel 5 , tre mått som involverar satser:

- # satser inkl namnrymdsdeklarationer
- # satser exkl namnrymdsdeklarationer
- # satser i metoder

3.4 Villkorslogik

För att mäta graden av villkorslogik i ett system kan antalet villkorssatser räknas. Villkorssatser kan innehålla sammansatta villkorsuttryck. För att bestämma graden av villkorslogik är det då rimligt att ta hänsyn till om en villkorssats innehåller sammansatta villkorsuttryck. Villkorslogik kan också vara nästlad. Vid en mätning av villkorslogik kan det därför vara intressant att mäta graden av nästling eller nästlingsdjup.

3.4.1 Cyklomatisk komplexitet

Cyklomatisk komplexitet är ett mått framtaget av McCabe [28] och anger antalet linjärt oberoende vägar genom en programmodul [19]. Cyklomatisk komplexitet är ett tal som beräknas enligt formeln $v(F) = e - n + 2$, där F är ett flödesschema (graf) för en programmodul, e är antalet kanter i grafer och n är antalet noder i grafen. Fenton och Pfleeger [19] menar att det matematiskt går att visa att den cyklomatiska komplexiteten för en modul motsvarar antalet beslutspunkter i modulen, plus ett. De framhåller att cyklomatisk komplexitet är ett objektivt mått för att bestämma antalet beslutspunkter i en modul, men att måttet ensamt inte kan användas för att mäta programkomplexitet.

Då cyklomatisk komplexitet är baserat på antalet beslutspunkter i en modul kan måttet användas för att mäta graden av villkorslogik i en modul. Tabell 3.4 visar gränsvärden för cyklomatisk komplexitet framtagna av Software Engineering Institute [38]. Uttrycket "untestable program" i tabell 3.4 är en påminnelse om att cyklomatisk komplexitet kan användas för att bestämma antalet testfall som krävs för att testa alla linjärt oberoende vägar genom en programmodul.

Flera verktyg existerar för att beräkna cyklomatisk komplexitet. Resultatet av beräkningarna beror på vad som räknas som beslutspunkter. SourceMonitor kan beräkna cyklomatisk komplexitet. Hjälpen till SourceMonitor innehåller följande information:

Cyclomatic Complexity	**Risk Evaluation**
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk program
greater than 50	untestable program (very high risk)

Tabell 3.4: Gränsvärden för cyklomatisk komplexitet. Källa: SEI [38]

"The complexity metric is counted approximately as defined by Steve McConnell in his book Code Complete, Microsoft Press, 1993, p.395. The complexity metric measures the number of execution paths through a function or method. Each function or method has a complexity of one plus one for each branch statement such as if, else, for, or while. Arithmetic if statements (MyBoolean ? ValueIfTrue : ValueIfFalse) each add one count to the complexity total. A complexity count is added for each `'&&'` and `'||'` in the logic within if, for, while or similar logic statements.

Switch statements add complexity counts for each exit from a case (due to a break, goto, return, throw, continue, or similar statement), and one count is added for a default case even if one is not present. Each catch or except statement in a try block (but not the try or finally statements) each add one count to the complexity as well." (Källa: SourceMonitor)

3.4.2 Kodblocksdjup

Att den cyklomatiska komplexiteten för en metod är t ex 27, avslöjar att metoden innehåller 26 beslutspunkter i from av if-satser, for-slingor, etc. Måttet säger dock ingenting om hur de 26 beslutspunkterna är nästlade i metoden. Om cyklomatisk komplexitet kompletteras med ett mått som anger nästlingsdjup eller kodblocksdjup, ges en bättre bild av hur koden är strukturerad i en metod.

SourceMonitor beräknar kodblocksdjup för varje metod, samt det genomsnittliga, re-

spektive maximala kodblocksdjupet i en klass eller grupp av klasser. Hjälpen till SourceMonitor redogör för hur maximalt kodblocksdjup beräknas:

> “Maximum Block Depth is the maximum nested block depth level found. At the start of each file the block level is zero. Depths up to 9 are recorded and all statements at deeper levels are counted as depth 9. This is indicated by the "9+label for the deepest level.” (Källa: SourceMonitor)

Genom att utföra mätningar på klassen FrontController i figur 4.6 ges här ett konkret exempel för hur SourceMonitor beräknar genomsnittligt kodblocksdjup. Klassen består av 10 satser (20 kodrader). 1 sats på nivå 0. 2 satser på nivå 1. 5 satser på nivå 2. 2 satser på nivå 3 Det maximala blockdjupet för klassen är 3 och det genomsnittliga blockdjupet är 1,80 = (1*0 + 2*1 + 5*2 + 2*3) / 10.

Satser på nivå 0 påverkar inte täljaren i bråkräkningen. Däremot påverkas nämnaren. Ett stort antal namnrymdsdeklarationer som t ex “using System;”, kan påverka värdet för genomsnittligt blockdjup. Attribut och metoddefinitioner påverkar också beräkningen av genomsnittligt blockdjup. Då genomsnittligt blockdjup önskas användas som indikator för hur villkorslogiken ändras i en omkonstruktion, vore det bättre att använda ett mått där nivå 1 definierades utifrån första nivån i metoderna. Genomsnittligt blockdjup kan ändå vara intressant och kommer att presenteras i samband med omkonstruktionerna i kapitel 5.

3.5 Långa metoder

För att mäta längden på metoder är kodrader och satser bra kandidater.

3.5.1 Antal satser/metod

SourceMonitor räknar antalet satser i varje metod, samt anger det genomsnittliga antalet satser per metod för en klass eller en samling av klasser. Då SourceMonitor används för

att beräkna satser för varje klass (fil), är det lämpligt att samma verktyg används för att räkna antal satser per metod.

3.5.2 Antal metoder med fler än x satser

Kerievsky [27] menar att tio kodrader eller färre är en bra riktlinje för metoder, samt att majoriteten av alla metoder bör ha en till fem kodrader. Han skriver också att om denna riktlinje följs, kan ett fåtal metoder tillåtas vara längre än tio rader, under förutsättning att de är enkla att förstå och inte innehåller kodduplicering. I samband med en beskrivning av omkonstruktionsmönstret “Compose Method”, skriver Kerievsky att sammansatta metoder (dvs. metoder som endast består av anrop till andra metoder) oftast är ungefär fem kodrader långa.

Om tio kodrader ses som en rekommenderad övre gräns som endast ett fåtal metoder kan tillåtas överstiga, är det intressant att jämföra den gränsen med den övre gränsen för cyklomatisk komplexitet i en enkel metod. Enligt SEI är en metod enkel om den har en cyklomatisk komplexitet mellan 1 och 10. En metod som består av nio if-satser utan sammansatta villkor överstiger vanligen 10 kodrader.

För att komplettera en beräkning av det genomsnittliga antalet satser/metod och antal satser i systemets längsta metod, är det lämpligt att räkna antal metoder vars kodlängder överstiger x antal satser. 10 är en kandidat för värdet på x.

3.6 Bristande Inkapsling

Inkapsling på olika nivåer görs för att minska beroenden mellan delsystem, moduler, klasser och strukturer. Att mäta graden av inkapsling skulle därför kunna uppskattas genom att mäta graden av beroende.

3.6.1 Coupling

Coupling (CBO) Coupling är ett mått som anger graden av beroende mellan moduler. Fenton och Pfleeger [19] föreslår att coupling mäts med hjälp av en skala med stigande beroendegrad, R_i<R_j (x och y är de två moduler mellan vilka coupling ska mätas):

- “No coupling relation R_0”:
  x och y har ingen kommunikation, dvs. de är helt oberoende.
- “Data coupling relation R_1”:
  x och y kommunicerar genom parametrar av primitiva typer.
- “Stamp coupling realtion R_2”:
  x och y kommunicerar genom parametrar av icke primitiva typer.
- “Control coupling relation R_3”:
  x kommunicerar med y i syfte att kontrollera y:s beteende.
- “Common coupling relation R_4”:
  x och y refererar till samma globala data.
- “Content coupling relation R_5”:
  x refererar till y:s inre data.

Modulerna x och y är löst (svagt) kopplade om i är 1 eller 2. De är hårt (starkt) kopplade om i är 4 eller 5 [19]. Coupling kan användas för att visa att en omkonstruktion resulterat i en beroendeminskning mellan mellan klasser i en omkonstruktion. Då en omkonstruktion genomförts för att helt eliminera ett beroende till en annan klass eller klassbibliotek räcker det att konstatera att beroendet inte längre existerar. Så är fallet då omkonstruktioner genomförs för att göra klasser helt oberoende av XML eller SQL.

3.7 Kapitelsammanfattning

Detta kapitel identifierade och beskrev mjukvarumått för att mäta hur omkonstruktioner påverkar de fyra strukturproblemen kodduplicering, villkorslogik, långa metoder och bristande inkapsling. Måtten presenteras i tabell 3.1

Kapitel 4

Strukturproblem och mönsterbaserade lösningar

4.1 Inledning

Avsnitt 2.7 identifierade och beskrev strukturproblemen kodduplicering, villkorslogik, långa metoder och bristande inkapsling. För att angripa dessa fyra strukturproblem identifieras och beskrivs i detta kapitel lämpliga objektorienterade mönster. Många mönster angriper fler än ett av de fyra nämnda strukturproblemen, vilket framgår av beskrivningarna. Mönstren är indelade efter det av de fyra nämnda strukturproblemen författaren anser vara det primära strukturproblem respektive mönster angriper (tabell 4.1). Indelningen är subjektiv och kan diskuteras.

Strukturproblem	**Mönster**
Duplicerad kod	Template Method Layer Supertype Adapter
Villkorslogik	Command Strategy State Missing Object
Långa metoder	Collecting Parameter
Bristande inkapsling	Composite Builder Query Object Facade Iterator Observer Decorator Separated Interface Plugin Bridge

Tabell 4.1: Exempel på mönster och de primära strukturproblem de angriper.

4.2 Duplicerad kod

4.2.1 Icke objektorienterade lösningar

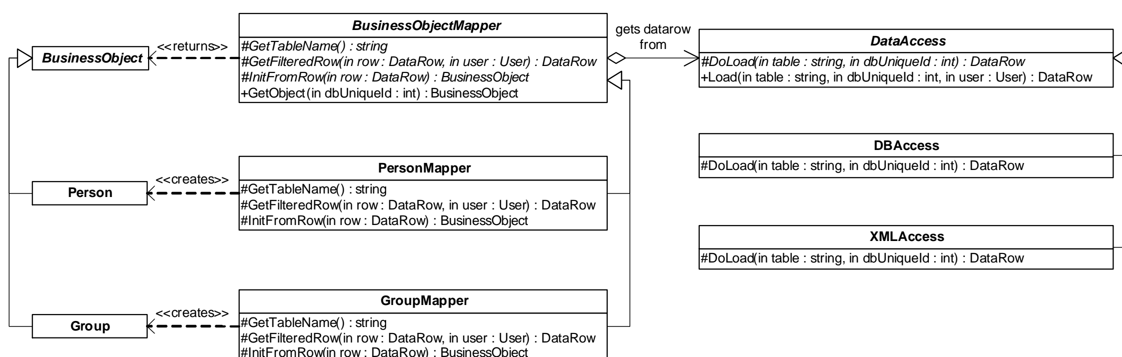
Då två eller flera kodblock innehåller exakt duplicerad kod, kan den duplicerade koden delas i en gemensam metod. Varje exakt duplicerat kodblock ersätts sedan med ett anrop till den gemensamma metoden. Samma teknik kan ofta tillämpas på kodblock som är nästan duplicerade, om metoden förses med extra parametrar och/eller extra villkorstester i metoden. Om parameterlistan blir lång kan längden på parameterlistan utgöra ett illaluktande strukturproblem [22]. Införandet av extra villkorstester ökar alltid den cyklomatiska komplexiteten (se avsnitt 3.4.1) för metoden och gör koden svårare att förstå.

4.2.2 Objektorienterade mönster

Mönster: Template Method

Template Method [24] är ett objektorienterat designmönster som kan användas för att förena kod som är nästan duplicerad. En mallmetod (Template Method) använder arv och polymorfism för att dela kod, utan att kräva extra parametrar eller villkorstester. Istället för att två nästan duplicerade kodblock ersätts med ett anrop till en gemensam metod med x antal parametrar, anropar mallmetoden abstrakta eller virtuella metoder som implementeras av subklasser till den klass i vilken mallmetoden är skriven.

Figur 4.1 illustrerar hur Template Methods kan användas i en mappning av data från en XML-fil eller en databas till ett affärslagerobjekt (mer om lagerindelad arkitektur i avsnitt 4.5.2). Figur 4.2 visar samma exempel i C#-kod. BusinessObjectMapper har en Template Method GetObject som anropar tre metoder som är abstrakt definierade i den egna metoden och implementerade i samtliga subklasser. DataAccess definierar en Template Method Load som anropar en metod som är abstrakt definierad i den egna metoden och implementerad i subklasserna DBAccess och XMLAccess. DataAccess innehåller en villkorstest för att avgöra om objektet redan laddats till minnet eller om det måste laddas in från disk. Observera att BusinessObjectMapper inte har några villkorstester som avgör vilken typ av affärslagerobjekt som ska skapas. Notera även att DataAccess inte har några villkorstester för att avgöra om objekt ska läsas in från en xml-fil eller från databasen. Inga parametrar behöver läsas av för att avgöra vilken specifik implementation av de abstrakta metoderna som ska exekveras. Villkorslogik ersätts med polymorfism genom att basklassreferenser tilldelas subklassobjekt. Mallmetoderna gör det möjligt att dela kod i basklasserna. Alternativet hade varit att skriva koden i subklasserna, ett alternativ som skulle innebära kodduplicering sånär som på några parametrar. Villkorstesterna hade undvikits genom polymorfism även om implementationen för GetObject och Load hade flyttat till subklasserna. Det som utmärker Template Method är att den del av subklassernas kod



Figur 4.1: Exempel på Template Methods: GetObject och Load

som är duplicerad kan delas i basklassen, medan implementationsskillnaderna delegeras till subklasserna - utan extra parametrar eller villkorslogik. BusinessObjectMapper behöver inte informera PersonMapper om vilken typ av affärslagerobjekt PersonMapper ska skapa. Den kunskapen finns redan i PersonMapper. På samma sätt behöver inte DataAccess informera DBAccess om vilken källa DBAccess ska läsa ifrån. Den kunskapen är inkapslad i DBAccess. Valet att läsa från databasen (istället för från xml-filen) tas genom att den DataAccess-objektreferens som används för att anropa Load har tilldelats ett objekt av subklassen DBAccess. Valet görs inte med parametrar och inte heller med villkorstester, utan med en polymorf tilldelning och ett polymorft anrop av en virtuell metod.

Exemplet i figurerna 4.1 och 4.2 inkluderar, förutom Template Method, enkla tillämpningar av tre mönster från Martin Fowlers bok “Patterns of Enterprise Application Architecture” [23]. De tre mönstren är Layer Supertype, Identity Map och Data Mapper.

Mönster: Layer Supertype

Om ett antal klasser har likartade ansvarsområden finns ofta metoder som kan delas i en gemensam basklass (supertyp). Klassen BusinessObject i figur 4.1 är ett exempel på en Layer Supertype [23]. BusinessObject är en basklass för alla klasser i ett affärslager (klasserna Person och Group är affärslagerobjekt). Affärslagerobjekt ges ofta ett id som matchar ett id i databasen. Istället för att duplicera attribut och metoder för id-hantering

```
public abstract class DataAccess {
        private IDictionary _loadedRows = new Hashtable();
        protected abstract DataRow DoLoad(string table, int dbUniqueId);
        public DataRow Load(string table, int dbUniqueId, User user) {
                if (!_loadedRows.Contains(dbUniqueId)) {
                        _loadedRows[dbUniqueId] = DoLoad(table, dbUniqueId);
                }
                return (DataRow)_loadedRows[dbUniqueId];
        }
}
public class DBAccess : DataAccess {
        protected override DataRow DoLoad(string table, int dbUniqueId) {
                //Load and return a DataRow from Database
        }
}
public class XMLAccess : DataAccess {
        protected override DataRow DoLoad(string table, int dbUniqueId) {
                //Load and return a DataRow from XML file
        }
}
public abstract class BusinessObjectMapper {
        private DataAccess _dataAccess = new DBAccess();
        protected abstract string GetTableName();
        protected abstract DataRow GetFilteredRow(DataRow row, User user);
        protected abstract BusinessObject InitFromRow(DataRow row);
        public BusinessObject GetObject(int dbUniqueId) {
                string table = GetTableName();
                User curUser = Session.GetCurrentUser();
                DataRow dataRow = _dataAccess.Load(table, dbUniqueId, curUser);
                DataRow filteredRow = GetFilteredRow(dataRow, curUser);
                return InitFromRow(filteredRow);
        }
}
public class PersonMapper : BusinessObjectMapper {
        protected override string GetTableName() {
                return "Person";
        }
        protected override DataRow GetFilteredRow(DataRow row, User user) {
                //Filter row and return filtered row
        }
        protected override BusinessObject InitFromRow(DataRow row) {
                Person person = new Person();
                person.Id = Convert.ToInt32(row["Id"]);
                person.FirstName = Convert.ToString(row["FirstName"]);
                //Init the rest of the properties and return the person object
        }
}
```

Figur 4.2: Exempel på Template Methods: GetObject och Load

i alla affärslagerklasser, kan sådan hantering skrivas i den gemensamma basklassen. Att introducera en Layer Supertype i ett lager kan ofta utnyttjas av klasser i ett annat lager för att undvika kodduplicering. BusinessObjectMapper i figur 4.1 kan vid mappning av data till affärslagerobjekt utnyttja att affärslagerobjekten har en gemensam basklass med delad funktionalitet, såsom hantering av id.

Mönster: Adapter

Återanvändbarhet (eng. reusability) är ett ord som betonas av Gamma et al [24] och när de beskriver vad som motiverar tillämpningen av mönstret Adapter är återanvändbarhet ett nyckelord. Återanvändning är ett alternativ till kodduplicering. Kod kan skrivas i en metod som anropas av flera klienter, vilket ger uttryck för återanvändning istället för duplicering. Två klasser kan dela gemensam kod i en basklass, vilket ger uttryck för återanvändning istället för duplicering. En Template Method kan införas för att dela en del av implementationen för en algoritm, vilket också handlar om återanvändning istället för att duplicera likartad kod.

Ibland uppstår situationen att det finns en klass med ett *ansvarsområde* som helt eller delvis stämmer med de krav en klientklass har, men klassens *gränssnitt* avspeglar inte de krav och behov klientklassen har. Det finns då två alternativ som båda orsakar problem:

1. Klientklassens algoritm använder den existerande klassen ändå. Eftersom klassens gränssnitt inte helt passar in i klientklassens algoritm, införs komplexitet i klientklassen för att anpassa klientklassens algoritm till den använda klassens gränssnitt.

2. En ny klass skapas med ett ansvarsområde som helt eller delvis är duplicerat i den existerande klassen, men där gränssnittet är anpassat till klientklassens algoritm. Resultatet? Två likartade klasser med olika gränssnitt. Att förena den gamla och nya klassens båda gränssnitt påverkar klientklasserna till de båda klasserna. Om den gamla klassen är en klass i ett tredjepartsbibliotek kanske det inte ens finns någon

möjlighet att förena gränssnitten.

Om alternativ 2 har valts, finns det duplicerad kod i form av "strukturer som till ytan ser olika ut, men som i grunden är duplicerade" (avsnitt 2.7.1). Hur kan denna form av kodduplicering elimineras, samtidigt som komplexiteten i alternativ 1 undviks? Studera gränssnittet för den klass som skapats enligt alternativ 2. Den klassens implementation är duplicerad och ska bort, men klassens gränssnitt ska flyttas till en ny klass, en adapterklass. Adapterklassen duplicerar inte kod som redan finns i en annan klass. Adapterklassen har en referens till en annan klass, vars gränssnitt anpassas eller konverteras till ett annat gränssnitt, nämligen adapterklassens gränssnitt.

En adapterklass kan också skrivas för att sätta samman och återanvända en kombination av flera andra klasser. Adapterklassklienten har inga referenser till de klasser som anpassas, utan är endast beroende av adapterklassens gränssnitt.

Mönster är verktyg för olika ändamål

Mönstret Adapter har här beskrivits som ett mönster vars primära syfte är återanvändning och reducering av duplicerad kod. Den initiala tanken då detta kapitel planerades var att beskriva Adapter som ett mönster vars primära syfte är att angripa bristande inkapsling. Adapterklassen kapslar in den klass som anpassas och bryter klientklassens beroende av den klass som anpassas. De strukturproblem som beskrivs i detta kapitel går ofta hand i hand och de mönster som beskrivs löser ofta mer än ett strukturproblem. Att avgöra vilket strukturproblem som primärt löses av ett mönster är inte alltid trivialt. I många fall är indelningen också subjektiv. Det betyder inte att indelningen inte är meningsfull, utan endast att tillämpningen av mönster kan diskuteras ur olika aspekter. Olika aspekter ger vägledning åt olika typer av omkonstruktioner som motiveras av olika typer av strukturproblem. Samma verktyg kan med framgång användas till olika saker, även om de flesta verktyg har ett primärt användningsområde.

```
if (person.age >= 13 && person.age <= 19) //Dubbla villkor
    //logik
else //alternativ logik

if (IsTeenAger()) //Dubbla villkor inkapslade i metod
    //logik
else //alternativ logik

bool IsTeenAger(){
    return person.age >= 13 && person.age <= 19;
}
```

Figur 4.3: Inkapsling av sammansatta villkor

4.3 Villkorslogik

4.3.1 Icke objektorienterade lösningar

Sammansatta villkor i ett uttryck kan enkelt sammanföras i en metod som returnerar sant eller falskt (figur 4.3). Villkorslogiken kvarstår efter en sådan enkel omkonstruktion, men koden blir lättare att läsa. Om ett sammansatt villkor används i fler än ett kodavsnitt, innebär metodinkapslingen också en reducering av duplicerad kod. En lång metod, vars längd beror på att flera kodavsnitt är inkapslade i ett if-else-block, kan brytas ner genom att separera ut villkorslogiken i en egen beslutsmetod, enligt figur 4.4.

4.3.2 Objektorienterade mönster

Polymorfism är en objektorienterad mekanism som i många fall helt kan ersätta villkorslogik. Command, Strategy och State är tre likartade [10] mönster som beroende på tillämpning helt eliminerar villkorslogik [27] eller åtminstone bryter ner den i mindre och mer läsbara enheter på samma sätt som i figur 4.4. Missing Object (eller Null Object) är ett mönster som kan användas för att ersätta upprepade villkorstester där det undersöks om en objektreferens refererar till ett objekt eller inte.

```
//Lång metod med villkorslogik
int Metod(int villkor, Data inData){
    if (villkor == 1){
        //Många kodrader för att läsa indata och skapa utdata
    }
    else if (villkor == 2){
        //Många kodrader för att läsa indata och skapa utdata
    }
    else if (villkor == 3){
        //Många kodrader för att läsa indata och skapa utdata
    }
    else{
        return FELKOD;
    }
    return utData;
}

//Uppdelning av lång metod med villkorslogik
int Metod(int villkor, Data inData){
    return BeslutsMetod(villkor, inData)
}
int BeslutsMetod(int villkor, Data inData){
    switch (villkor){
        case 1: return Metod1(inData);
        case 2: return Metod2(inData);
        case 3: return Metod3(inData);
        default: return FELKOD;
    }
}
int Metod1(Data inData){
    //Många kodrader för att läsa indata och skapa utdata
    return utData;
}
int Metod2(Data inData){
    //Många kodrader för att läsa indata och skapa utdata
    return utData;
}
int Metod3(Data inData){
    //Många kodrader för att läsa indata och skapa utdata
    return utData;
}
```

Figur 4.4: Uppdelning av villkorslogik med hjälp av beslutsmetod

Mönster: Front Controller, Command och Missing Object

En miljö där villkorslogik lätt kan växa och göra programkod svår att underhålla är webbaserade system vars sidor presenterar innehåll baserat på parametrar som skickas med HTTP-anropen. Både Fowler [23] och Trowbridge et al [33] jämför tre användbara mönster som delar in en webbtillämpning i en presentationsdel (View), en datamodellsdel (Model) och en del som kontrollerar samspelet mellan data och presentation (Controller). De tre mönstren är Model-View-Controller (MVC), Page Controller och Front Controller.

Utan att gå in på detaljer för alla aspekter som skiljer dessa tre mönster åt är MVC det mönster som omnämns först. MVC separerar data från presentation, samt definierar en Controller-modul för att tolka systeminteraktion. En implementation av MVC leder lätt till duplicerad kod för uppgifter som utförs vid varje HTTP-anrop, t ex användarautentisering, validering, läsning av anropsparametrar och presentationsrelaterade databasuppslagningar [33].

Page Controller är en förfinad version av MVC, där varje dynamisk webbsida har en egen Controller-klass. Controller-klasser kan dela kod via en gemensam basklass. I de fall koden är likartad, men inte exakt duplicerad, kan Template Method tillämpas för att dela kod [33]. I takt med att komplexiteten i systemet växer (i form av ett ökat antal Page Controllers), måste arvshierarkin utökas för att undvika att villkorslogiken i basklassen växer och blir ett strukturproblem. Då arvshierarkin växer, utgör längden på arvshierarkin ett annat strukturproblem.

För att undvika långa arvshierarkier som kan uppstå då mönstret Page Controller används för att reducera duplicerad kod och villkorslogik, finns mönstret Front Controller. Istället för en Controller för varje dynamisk webbsida kontrollerar en Front Controller flera sidor. Page Controller och Front Controller kan samexistera i ett system. Page Controller kan tillämpas för webbsidor i ett delsystem, medan en Front Controller används för sidor med mer komplicerad logik. Då en enda Controller-klass läser av anropsparametrar m.m. för att uppdatera datamodellen och styra vilken av flera presentationssidor som visas,

```
string action = Request.Params["action"];
if (action == null)
{
    //Kod för att hantera detta fall
}
else if (action.Equals("showpicture"))
{
    string id = Request.Params["id"];
    string exif = Request.Params["showexif"];
    if (id != null)
    {
        if (exif != null && Convert.ToBoolean(exif))
        {
            //Mer kod, villkorslogik och hantering av specifika parametrar
            //Vidarebefordra anrop till specifik sida
        }
        else
        {
            //Mer kod, villkorslogik och hantering av specifika parametrar
            //Vidarebefordra anrop till specifik sida
        }
    }
    else
    {
        //Kod för att hantera detta fall
    }
}
else if (action.Equals("showgallery"))
{
    //Mer kod, villkorslogik och hantering av specifika parametrar
    //Vidarebefordra anrop till specifik sida
}
else if (action.Equals("deletepictures"))
{
    //Mer kod, villkorslogik och hantering av specifika parametrar
    //Vidarebefordra anrop till specifik sida
}
else if (action.Equals("searchpictures"))
{
    //Mer kod, villkorslogik och hantering av specifika parametrar
    //Vidarebefordra anrop till specifik sida
}
else if (action.Equals("uploadpictures"))
{
    //Mer kod, villkorslogik och hantering av specifika parametrar
    //Vidarebefordra anrop till specifik sida
}
```

Figur 4.5: Exempel på villkorslogik i en Front Controller utan Command

krävs omfattande villkorslogik, om inte villkorslogiken ersätts av en annan mekanism. En sådan mekanism som rekommenderas [23][33] vid användning av en Front Controller är mönstret Command [24]. Figur 4.5 visar schematisk kod för hur en Front Controller utan användandet av Command skulle kunna se ut.

Förhoppningsvis väljer ingen att skriva en Controller som i figur 4.5, utan väljer istället flera Page Controllers (med Template Methods i en gemensam basklass) eller en Front Controller med flera Command-klasser. Figur 4.5 tjänar ändå som ett exempel på villkorslogik där första nivån av villkorstesterna helt kan ersättas med polymorfism, genom införandet av mönstret Command. Beroende på hur av villkorstesterna på nivå 2,3,...n ser ut kan det finnas anledning att ersätta en del av dessa villkorstester med objektorienterade mönster och polymorfism. Hur går det då till att ersätta villkorstesterna på första nivån i figur 4.5 med mönstret Command? Figur 4.6 visar schematiskt den omkonstruerade koden och följande steg beskriver hur omkonstruktionen genomförs:

1. Skapa en abstrakt basklass AbstractCommand.
2. I AbstractCommand, definiera en abstrakt metod ExecuteAction().
3. För varje "action" i figur 4.5, skapa en konkret subklass till AbstractCommand.
4. För varje subklass till AbstractCommand, implementera metoden ExecuteAction().
5. Skapa en metod GetCommand(string action) som returnerar ett konkret Command-objekt baserat på action-sträng. Metoden kan skrivas i samma klass som innehåller villkorstesterna, i en ny klass eller som en statisk metod i AbstractCommand.
6. Ersätt första nivån av villkorstester med ett anrop av GetCommand och ett polymorft anrop av ExecuteAction

Det finns olika sätt att implementera metoden GetCommand. Ett sätt är att använda en teknik som kallas reflection. Det går också att fylla en lista med element {{sträng,

```
public partial class FrontController : Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string action = Request.Params["action"];
        GetCommand(action).ExecuteAction(Request.Params);
    }

    //Exempel på mappning från sträng till objekt via Reflection i .NET
    private AbstractCommand GetCommand(string action)
    {
        string classFullName = nameSpace + "." + action + "Command";
        try {
            return (AbstractCommand)Activator.CreateInstance(GetType(classFullName));
        }
        catch {
            return new UnknownCommand();
        }
    }
}

public class ShowGalleryCommand : AbstractCommand
{
    public override void ExecuteAction(NameValueCollection parameters)
    {
        //Mer kod, villkorslogik och hantering av specifika parametrar
        //Vidarebefordra anrop till specifik sida
    }
}

public class UnknownCommand : AbstractCommand
{
    public override void ExecuteAction(NameValueCollection parameters)
    {
        //Gör ingenting (eller skriv till en eventuell fellogg)
    }
}
```

Figur 4.6: Front Controller med Command och Missing Object

Command-objekt}, {sträng, Command-objekt},...} och sedan göra mappningen från action-sträng till objekt med hjälp av listan. Villkorslogik som endast returnerar Command-objekt är också ett alternativ. Det senare alternativet utgör ingen reducering av antalet villkorstester, utan är endast en omkonstruktion av villkorslogiken som bryter ner den i mindre och mer läsbara enheter. Att skapa objekt med reflection kan utgöra ett prestandaproblem i en applikation med höga prestandakrav eller i en tillämpning där ett stort antal objekt skapas. I en webbtillämpning med täta anrop kan reflection utgöra en flaskhals. Exemplet i figur 4.6 använder ändå reflection för att demonstrera hur tekniken fungerar.

UnknownCommand i figur 4.6 är en tillämpning av ett mönster som kallas Missing Object eller Null Object. UnknownCommand säkerställer att ett objekt alltid returneras av metoden GetCommand. FrontController-klassen behöver inte göra några tester för att se om `Request.Params["action"] == null` eller om `GetCommand(action) == null`. Det kan ifrågasättas om skapandet av en klass är motiverat, då endast två null-tester ersätts. Det kan dock finnas andra tillämpningar där införandet av Missing Object kan eliminera ett större antal null-tester.

Tillämpningar där villkorslogik kan ersättas helt

Exemplet i föregående avsnitt innehåller en mappningsmetod GetCommand (figur 4.6), där en strängparameter mappas via reflection till en klass. Mappningen kan ses som en variant av beslutsmetoden i figur 4.4. Exemplet utgör ett fall där mappningen inte kan tas bort i samband med införandet av mönstret Command (strängparametern skickas i ett HTTP-anrop). I många andra fall innebär införandet av Command, Strategy eller State att mappningen blir överflödig och kan tas bort. Figur 4.7 visar ett exempel där villkorslogik elimineras helt, utan något behov av mappning via reflection, listor eller andra icke-objektorienteringsspecifika tekniker (såsom beslutsmetoden i figur 4.4).

Då programkod delas i en hierarki av Command-klasser, kan kod som är gemensam i dessa klasser delas i den gemensamma basklassen. Mönstren Strategy, State och Command

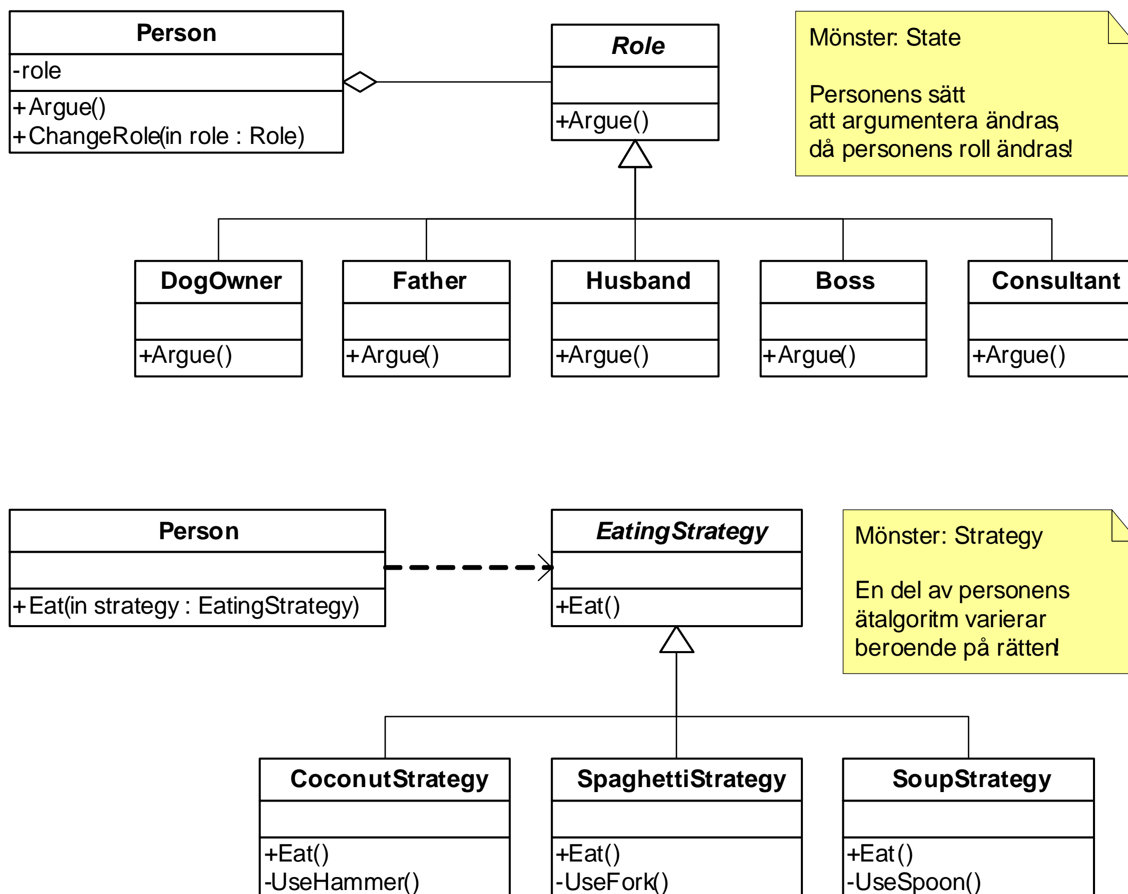
```
/*********************** Kod med villkorslogik *********************/
...
//Klientanrop
serviceManager.PerformTask("Run", parameters);
...
//En klass som hanterar ett antal uppgifter
public class ServiceManager
{
    public void PerformTask(string task, List<string> parameters)
    {
        if (task.Equals("Run"))
        {
            //Många kodrader för att hantera uppgiften "Run"
        }
        else if (task.Equals("Jump"))
        {
            //Många kodrader för att hantera uppgiften "Jump"
        }
        else if (task.Equals("Walk"))
        {
            //Många kodrader för att hantera uppgiften "Walk"
        }
        //Många fler else if
    }
}

/************************ Kod med polymorfism *********************/
...
//Klientanrop
serviceManager.PerformTask(new RunCommand(parameters));
...
public class ServiceManager
{
    //Beroende på vad ServiceManager gör blir kanske hela klassen överflödig!
    public void PerformTask(Command command)
    {
        command.Execute();
    }
}
public class RunCommand : Command
{
    public override void Execute()
    {
        //Många kodrader för att hantera uppgiften "Run"
    }
}
```

Figur 4.7: Exempel där villkorslogik helt ersätts med polymorfism

skapar alla tre möjligheter att dela kod mellan subklasser med hjälp av Template Method.

Mönster: Strategy



Figur 4.8: Strukturen för State och Strategy liknar Command

Strategy är ett mönster, vars struktur liknar Command (figur 4.8), men de båda mönstrens syfte skiljer sig åt. Både Command och Strategy kapslar in beteende. Command representerar ett fullständigt kommando, medan Strategy representerar ett delmoment av en algoritm. Det kan förstås diskuteras om det är en bra strategi att låta personklassen i figur 4.8 vara ovetande om vilket redskap som används i ätalgoritmen. Det finns också anledning att fråga om det är meningsfullt att ha olika namn på mönster som är så lika [10]

som Command och Strategy. Då mönsternamn utöver designstruktur även kommunicerar syfte, kan det ändå vara meningsfullt att skilja på mönster som liknar varandra.

Mönster: State

Ibland är det önskvärt att ändra beteendet hos ett objekt då dess tillstånd ändras (figur 4.8). Istället för att tillståndsvariabler av primitiva typer styr objektets beteende med hjälp av villkorslogik, kan en hierarki av State-klasser skapas, likt den som byggs upp vid införandet av Command och Strategy. En objektreferens av samma typ som State-klassernas basklass eller gränssnitt används som attribut i den klass vars beteende ska kunna ändras dynamiskt. Metoderna kan ändra beteende genom att delegera anrop via State-objektreferensen. Precis som Command och Strategy, använder State polymorfism för att ersätta villkorslogik.

4.4 Långa metoder

4.4.1 Icke objektorienterade lösningar

Motmedlet mot långa metoder är i 99 % av fallen att bryta ut en ny metod ur den långa metoden [22]. Den långa metoden krymps till en kortare sammansatt metod (eng. Composed Method [27]), dvs. en metod som består av anrop till andra metoder. Kommentarer i de långa metoderna ger ofta god vägledning i att avgöra vilken kod som ska brytas ut från den långa metoden och flyttas till den nya metoden. Kerievsky [27] påpekar att en bra sammansatt metod består av kod på samma detaljnivå. Villkorsuttryck och tilldelningssatser har en mer detaljerad nivå än metodanrop. Läsbarheten ökar om villkorslogik och tilldelningssatser kapslas in i metodanrop, istället för att innehållet i en metod består av en blandning av olika detaljnivåer.

4.4.2 Objektorienterade mönster

En metod som ackumulerar information i en lokal variabel kan lätt leda till att metoden blir lång (t ex en metod som bygger upp en sträng i flera steg). För att bryta ner den långa informationsackumulerande metoden i flera korta metoder kan den lokala variabeln ersättas med en insamlande parameter (eng. Collecting Parameter[6]) som skickas mellan metoderna [27].

Långa metoder är ibland besvärliga att bryta ner genom att skapa och anropa flera korta metoder, pga. att de nya korta metoderna behäftas med långa parameterlistor. Om den långa parameterlistan till en metod innehåller parametrar som skulle kunna grupperas i ett objekt, (vars klass har ett beskrivande namn) är det ett sätt att förhindra att läsbarheten blir lidande till följd av långa parameterlistor [22]. En parameterlista {namn, ålder, kön, civilstånd, yrke} kan t ex med fördel grupperas i en klass Person.

Långa metoder är ibland långa för att de har många ansvarsområden och i sig själva utgör ett ämne för en ny klass. En uppdelning av den långa metoden i flera korta metoder lyfter fram dessa ansvarsområden i ljuset. Då den aktuella klassen behäftas med nya metoder som inte logiskt hör ihop med klassens huvudsakliga ansvarsområde, kan det finnas anledning att skapa en ny klass och flytta metoder till den nya klassen.

Långa metoder kan vara ett resultat av omfattande villkorslogik eller bristande inkapsling och i sådana fall kan de mönster som beskrivs i avsnitt 4.3 och 4.5 vara verktyg att minska längden på metoder.

4.5 Bristande inkapsling

4.5.1 Icke objektorienterade lösningar

Inkapsling av data är inte unikt för objektorienterade språk. Ett procedurellt språk (t ex C), kan kapsla in data i funktioner. Ett objektorienterat språk inför ytterligare en nivå av

inkapsling - klasser, samt möjliggör bevarande av data (tillstånd) i objekt. I många fall är det inte nödvändigt att bevara tillstånd mellan anrop av funktioner. Inkapsling av data i en funktion kan då vara fullt tillräckligt.

4.5.2 Objektorienterade mönster

Lagerindelade arkitekturer

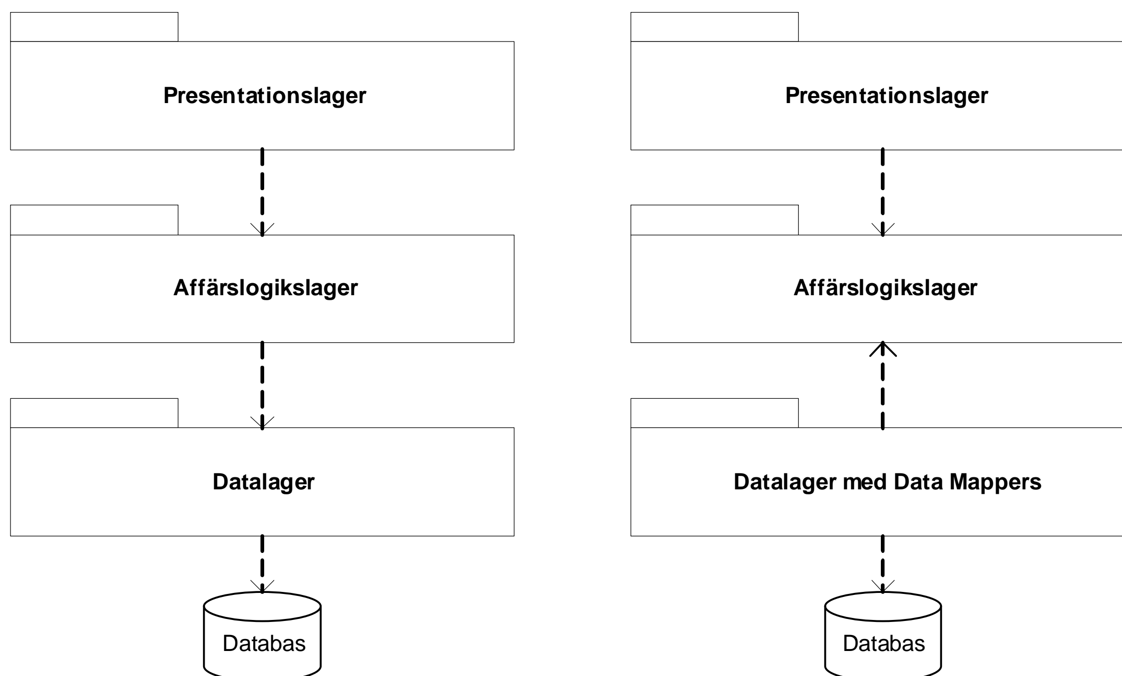
> "Computer Science is the discipline that believes all problems can be solved with one more layer of indirection. –Dennis DeBruler"

Citat av Kent Beck i *Refactoring* [22]

På liknande sätt som datakommunikation sker genom flera lager av protokoll (t ex FTP / TCP / IP / ethernet), delas klasser och moduler i ett informationssystem ofta in i lager, t ex datalager, affärslogikslager och presentationslager [23]. En vanlig regel för en lagerindelad arkitektur är att klasserna i varje lager endast är beroende av klasserna i sitt eget lager eller i det närmast underliggande lagret (se vänsterdelen i figur 4.9). En indelning i presentationslager, affärslogikslager och datalager, där denna regel tillämpas, innebär t ex att presentationslagerklasser inte har någon direkt koppling till klasser i datalagret och att affärslogiksklasser inte är beroende av några presentationslagerklasser. Data Mapper [23] är ett mönster som frångår regeln med endast horisontella eller nedåtriktade beroenden (se figurerna 4.9 och 4.1). Med hjälp av Data Mappers kan affärslagerklasser utvecklas oberoende av datalagerklasserna, vilket enligt Fowler [23] är önskvärt om mappningen mellan relationsdatabastabeller och klasser är komplex.

Inkapsling av SQL och XML

Databasspråket SQL och märkspråket XML används flitigt i informationssystem. Då systemet växer är det önskvärt att begränsa antalet klasser som är beroende av SQL och XML.



Figur 4.9: Exempel på lagerindelade arkitekturer

SQL är inte ett helt standardiserat språk, utan det finns dialektskillnader mellan olika versioner av SQL. För system med krav på databasoberoende är det önskvärt att enkelt kunna byta ut eller skriva om den del av ett system som är databasberoende. En sådan process är svår om SQL-beroendet är utspritt genom hela systemet. I lagerindelade arkitekturer med ett datalager (se figur 4.9) är det önskvärt att begränsa SQL-beroendet till klasserna i datalagret. Vid användning av mönstret Data Mapper [23] begränsas användningen av SQL till de olika mappningsklasserna. För att undvika duplicering av SQL-kod kan mappningsklasserna ha en gemensam abstrakt basklass likt klassen BusinessObjectMapper i figur 4.1. För system med stora krav på databasoberoende, kan SQL-beroende klasser i ett datalager ha subklasser som hanterar SQL-dialektskillnader [7]. Även om stor möda läggs ner på att kapsla in SQL-beroende, blir inkapslingen inte alltid fullständig. Affärslagerklasser behöver ofta kunna ställa frågor till databasen och även om SQL-beroende på något sätt kapslats in i ett datalager, kan databasberoende i form av tabell- och kolumnnamn lysa igenom

till affärslagret. I de fall där sådant databasberoende kvarstår i affärslagret, kan mönstret Metadata Mapping [23] tillämpas, för att undvika ett beroende mellan affärslagerklasserna och databasens tabell- och kolumnnamn.

Att på något sätt kapsla in SQL-anrop i ett mjukvarusystem är önskvärt, men att bygga ett komplext datalager med t ex Data Mappers för att helt eliminera databasberoende i affärslagret, kan innebära att den komplexitet som utgörs av det beroendebrytande mönstret, överskuggar det uppnådda databasoberoendet. Ett enklare mönster kan användas för att kapsla in SQL-anropen, om dessa är enkla och kolumnerna i databastabellerna är nära knutna till attributen i affärslagerklasserna. Active Record är ett sådant enklare mönster, där affärslagerklasser kapslar in både affärslogik och databasåtkomst [23].

På liknande sätt som det är önskvärt att begränsa SQL-beroende i ett system, kan det även finnas anledning att begränsa beroendet av XML och specifika XML-hanteringsklasser. Kerievsky [27] berättar om ett projekt där en smärtsam omkonstruktion blev nödvändig för att uppgradera ett system till att använda en nyare version av Document Object Model (DOM). I systemet fanns XML-hantering (beroende av DOM 1.0) utspridd i systemet. Utöver faran med en nära koppling till XML, är XML-hantering en källa till långa metoder och kodduplicering. Genom att kapsla in XML-hantering med mönster, såsom Composite, Builder eller Collecting Parameter, kan både duplicering och långa metoder undvikas, samtidigt som XML-beroende begränsas [27].

Mönster: Query Object

Ett enkelt sätt att kapsla in SQL-frågor och undvika duplicering av SQL är att skriva SQL-frågor i olika metoder som förses med parametrar. Då antalet metoder med SQL-frågor blir stort, uppstår en dålig lukt som Kerievsky kallar “Combinatorial Explosion” [27]. Interpreter är ett mönster beskrivet av Gamma et al [24] och som kan tillämpas för att bygga upp ett syntaxträd för att tolka någon form av språk med enkel grammatik (en grammatik som kan representeras av några få klasser [27]). Query Object [23] är en

variant av mönstret Interpreter med syftet att representera SQL-frågor. Istället för att skriva hela SQL-frågesträngar, byggs en frågeobjektstruktur upp för att kapsla in och generera SQL-frågor. Utöver inkapslingen av SQL kan en tillämpning av Query Object motiveras med att det blir lättare att skapa variationer av befintliga SQL-frågor, utan att duplicera frågorna och utan att utöka antalet SQL-inkapslande metoder eller förse metoderna med fler parametrar och villkorslogik. Det som talar emot en tillämpning av mönstret är att komplexa SQL-frågor kräver en mer komplex tillämpning av mönstret. Om SQL-frågorna är enkla kan en tillämpning av mönstret skapa en onödigt komplex design. När är det då motiverat att tillämpa mönstret? Fowler menar att mönstret kommer till sin rätt då SQL-frågorna är komplexa, men att det då oftast är bäst att använda en färdig kommersiell tillämpning av mönstret. Det krävs enligt Fowler mycket sofistikerade behov och kunskaper om det ska vara motiverat för en projektgrupp att själva implementera en lösning baserat på Query Object.

Gudomliga klasser och databehållare

Att separera SQL och XML från affärslagerobjekten är ett sätt att förebygga beroenden till implementationsstrukturer. Att separera ut databaskommunikation från affärslagerobjekten är också ett sätt att reducera objektens ansvarsområden och förebygga strukturproblem i form av stora klasser. En hög nivå av inkapsling betyder inte att ett helt system är inkapslat i en enda klass. En sund inkapsling tar hänsyn till vilka ansvarsområden klasser har.

Att tala om gudomliga klasser (eng. God Classes) och databehållare (eng. Data Containers) är ett sätt att beskriva en snedvriden fördelning av ansvarsområden mellan klasser [15]. Databehållarklasser är klasser vars ansvarsområde är begränsat till att lagra data. Om databehållarna har metoder är nästan alla metoder enkla åtkomstmetoder som inte styr beteende. En gudomlig klass utmärker sig som en mycket stor klass som hanterar andra klasser som databehållare. Nästan alla förändringar i systemet påverkar den gudomliga klassen och det är svårt att se vilka effekter ändringarna har. Klassens metoder har

olika ansvarsområden, vilket gör återanvändning och testning av klassen nästan omöjligt. Ett sätt att identifiera gudomliga klasser är att fråga dem som underhåller systemet vilka klasser de helst inte vill röra. Ett annat sätt är att leta efter klasser vars namn innehåller ord som “System”, “Manager”, “Driver” eller “Controller” [15].

Mönster: Facade

Fowler [22] menar att det finns tillfällen då omkonstruktioner inte bör göras. Att genomföra en omkonstruktion nära inpå ett avtalat leveransdatum (deadline) är riskabelt. Det finns tillfällen då det är lättare att skriva om ett delsystem från grunden än att omkonstruera det steg för steg. Dåligt skrivna delsystem (som trots sin uselhet måste användas) kan också kapslas in som komponenter, så att en eventuell omkonstruktion kan skjutas på framtiden. Facade [24] är ett mönster som kan användas för att kapsla in ett delsystem, i syfte att förenkla användandet av delsystemet och samtidigt förhindra att användarna av delsystemet blir beroende av delsystemets inre struktur. En tillämpning av Facade på ett delsystem gör det möjligt att senarelägga ett beslut om omkonstruktion eller total omskrivning av delsystemet, utan att påverka användarna av delsystemet.

Demeyer et al [15] påpekar att det är svårt att direkt dela upp och flytta ansvarsområden för en gudomlig klass till nya klasser. Istället för att direkt flytta metoder till nya klasser, rekommenderar de att den gudomliga klassen bryts ner stegvis till att bli en fasad (eng. Facade), dvs. den gudomliga klassens metoder lämnas kvar för att inte påverka de klasser som är beroende av den gudomliga klassen, samt att metoderna delegerar ansvar till nya klasser. Då den gudomliga klassen blivit en fasad för nya klasser, kan metodanropen från fasadklassens klienter steg för steg styras om till de nya klasserna, så att de istället använder de nya klasserna direkt. När inga metodanrop till fasadklassen finns kvar, kan den tas bort.

Evans [18] talar om införande av “anticorruption layers” som ett sätt att isolera ett delsystem från övriga delar av systemet. Facade och Adapter [24] är två mönster som kan användas för att implementera ett lager i syfte att skydda ett system från ett delsystems

fördärvade designmodell.

Inkapsling av objektskapande med Factories

Omfattande databashantering i affärslagerklasser kan leda till att klassernas ansvarsområde i affärslagret skyms. Då processen att skapa eller initiera ett objekt är omfattande eller komplex, bör koden för den processen inte kapslas in tillsammans med den övriga koden i klassen för det objekt som skapas, eftersom koden för objektskapandet då skymmer objektets huvudsakliga ansvarsområde.

Konstruktorer är ett bra alternativ för enkelt objektskapande, men då konstruktorerna blir stora (alternativt kräver många hjälpmetoder) eller blir behäftade med omfattande villkorslogik, finns det anledning att på ett bättre sätt separera den kod som skapar ett objekt från den kod som implementerar objektets primära ansvarsområde. Ett undantag kan vara om en klass primära ansvarsområde är att skapa objekt av sig själv. Eric Evans skriver i sin bok Domain Driven Design:

> "Creation of an object can be a major operation in itself, but complex assembly operations do not fit the responsibility of the created objects. Combining such responsibilities can produce ungainly designs that are hard to understand. Making the client direct construction muddies the design of the client, breaches encapsulation of the assembled object or aggregate, and overly couples the client to the implementation of the created object." [18]

Om komplex kod för objektskapande separeras från koden i det skapade objektets klass, var ska den komplexa koden ta vägen? Om den komplexa koden för att skapa objektet istället hanteras av den klient som anropar objektet blir situationen ännu värre (precis som Evans framhåller), eftersom klienten då blir beroende av implementationen för det skapade objektet. Frågan kvarstår. Var ska den komplexa koden för objektskapande ta vägen om den inte integreras med det objekt som skapas och inte heller hanteras av den

klient som använder objektet? I ett eget objekt. Ett sådant objekt kallas för en Factory [18].

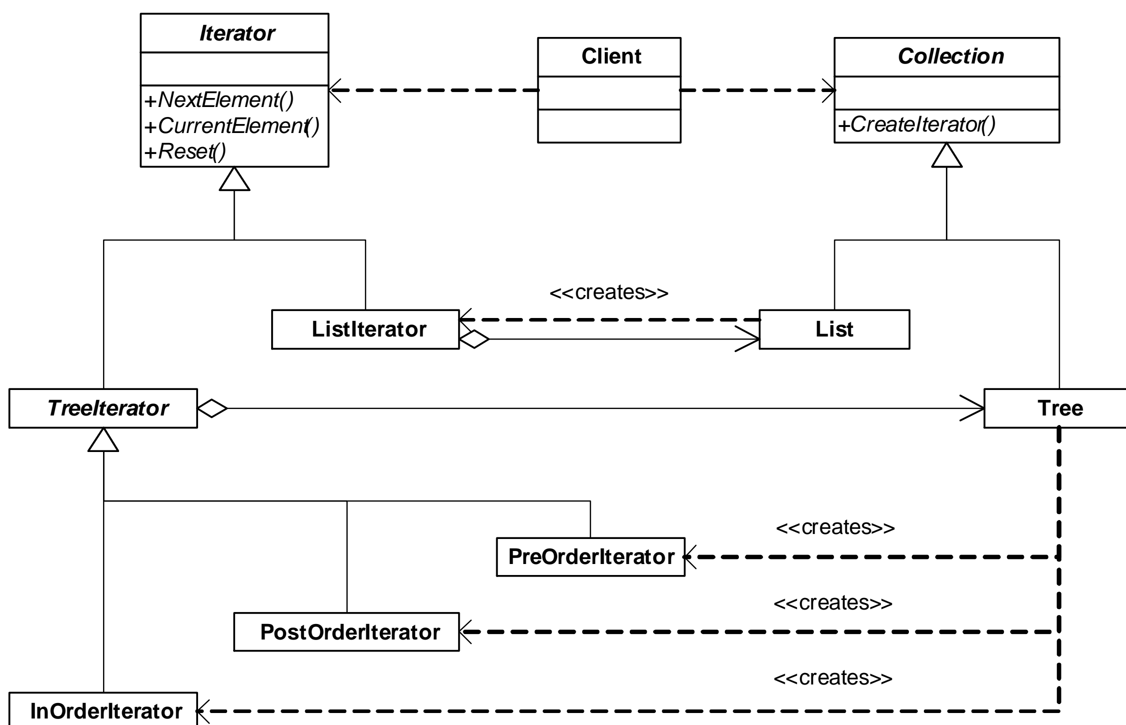
En Factory-klass hanterar komplext objektskapande för en annan klass. En Factory undviker samtidigt IKEA-principen, dvs. att användare av objekt blir beroende av objektens implementation och sammansättning. Blir då inte en Factory-klass starkt kopplad till de objekt de skapar? Naturligtvis, men det primära ansvarsområdet för en Factory är just att känna till hur objekten ska sättas ihop. Då Factories behövs för ett flertal relaterade klasser, finns risk att objektskapande kod blir duplicerad mellan de olika Factory-klasserna. För att undvika sådan kodduplicering, utan att införa ny villkorslogik kan en klasshierarki av relaterade Factories skapas, där Template Method [24] tillämpas. Klasshierarkier med relaterade Factories utgör ett eget mönster, Abstract Factory [24].

I samband med ett införande av Factories för att göra en separat inkapsling av komplext objektskapande, måste en avvägning göras om objektskapandet verkligen är så komplext att det motiverar en Factory. Evans skriver:

> "The introduction of Factories has great advantages, and is generally underused. Yet there are times when the directness of a constructor makes it the best choice. Factories can actually obscure simple objects that don't use polymorphism." [18]

Mönster: Iterator

Iterator [24] är ett mönster som kan införas för att förhindra att en klass blir beroende av datastrukturen i en annan klass. Mönstret är inbyggt i klassbibliotek för flera olika programspråk och plattformar. IEnumerator är ett iteratorgränssnitt som implementeras av många klasser i .NET. En iteratorklass implementerar metoder för att iterera över elementen i en datastruktur (såsom en lista eller ett träd), utan att den klass som använder iteratorklassen blir beroende av *antalet* element i strukturen, *hur* elementen är strukturerade, i vilken *ordning* de besöks eller om vissa element *filtreras* bort då iterering sker över



Figur 4.10: Iteratorer kan kapsla in vad som itereras över och hur

elementen (se figur 4.10). Med hjälp av en iterator kan iteratoranvändaren hållas lyckligt ovetande om den struktur som itereras över. Samma gränssnitt kan presenteras för iterering över en lista och iterering över en trädstruktur. En förändring av datastrukturen påverkar inte iteratorklienten. Är iteratorklasshierarkin komplex? För den som inte förstår virtuella metoder, ja, men det är en komplexitet som främst består i att tolka ett diagram. Komplexiteten syns inte i klientkoden annat än genom en polymorf tilldelning av ett konkret iteratorobjekt till en abstrakt iteratorreferens. Vad skulle hända med koden om det inte fanns en iteratorhierarki eller bara en enda iteratorklass utan subklasser? Villkorslogik måste då införas för att hantera datastrukturtyp (lista eller träd). Ytterligare villkorslogik tillkommer för valet av itereringsalgoritm (framåt, bakåt, preorder, postorder, inorder). Ska elementen filtreras eller inte? Mer villkorslogik. Ska nya datastrukturer införas? Ännu mer villkorslogik. Iteratorklasshierarkin undviker sådan villkorslogik och bäddar för Template

Methods genom vilka kodduplicering kan reduceras i subklasser. Den primära vinningen med mönstret är dock inkapslingen. En iterator är mycket mer än ett substitut för en for-loop!

Kod som läser in XML, validerar, filtrerar och sparar information baserat på ett antal olika villkor, kan lätt leda till långa metoder, svårläst och villkorsrik kod, behäftad med kodduplicering. Sådan XML-hantering kan kapslas in av en eller flera hjälpklasser som sköter inläsning och validering, samt ger åtkomst till filtrerad information via en iterator.

Mönster: Composite

Composite [24] är ett mönster som kan användas för att kapsla in hantering och bearbetning av trädstrukturerad data. Composite-mönstret består av tre "delar": ett gränssnitt, klasser som representerar lövnoder i trädet och klasser som representerar föräldernoder i trädet. Löv och föräldrar har ett gemensamt gränssnitt som gör att klientklasser kan behandla löv och föräldrar på samma sätt. På så sätt kan klientklassen befrias från villkorslogik för att avgöra vilken typ av data som behandlas, samtidigt som navigeringen i trädet kan göras rekursivt. Hanteringen av filer och kataloger i ett filsystem kan kapslas in med hjälp av mönstret Composite [10]. Olika XML-bibliotek hanterar XML-noder med Composite-mönstret [27].

Mönster: Builder

Ibland behöver ett flertal relaterade (men olika) datastrukturer byggas upp i ett system, där den stegvisa byggprocessen för dessa strukturer följer samma mönster. Istället för att den kod som bygger upp datastrukturerna blir beroende av strukturskillnaderna, kan skillnaderna kapslas in av byggarklasser med virtuella metoder. Builder [24] är ett mönster som åstadkommer just den typen av inkapsling. En byggarklass kan även införas för att kapsla in datastrukturuppbyggande kod, där bara en typ av datastruktur byggs upp. Ett exempel på en sådan tillämpning är inkapsling av XML-skapande, där XML-strukturen först kapslas

in med hjälp av Composite-baserade XML-klasser och där Composite-användningen i sin tur kapslas in med Builder [27].

Mönster: Observer

I situationer där ett flertal objekt med samma eller olika klasstillhörighet är beroende av tillståndsändringar i ett annat objekt, är det önskvärt med en mekanism som kapslar in beroendet mellan objekten. Observer [24] är ett mönster där objekt kan göras till observerare av andra objekt (subjekt) som gjorts observerbara. Observerare registrerar sig hos ett subjekt, så att subjektet vid förändring av sitt tillstånd kan meddela en lista av observerare. Mönstret Observer tillämpas av klassbibliotek med stöd för händelsestyrd programmering.

Mönster: Decorator

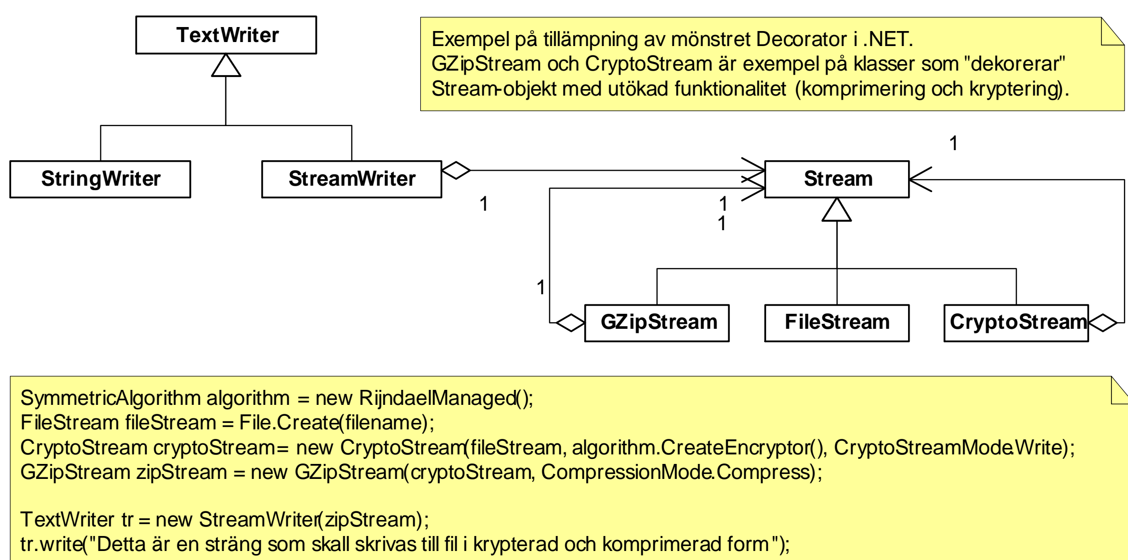
Två fördelar med användningen av arv är att:

- Ansvarsområden sprids ut på flera klasser. Att ha en enda klass som t ex krypterar, komprimerar och skriver data till fil, resulterar i stora klasser, med många metoder och ansvarsområden. Att istället dela upp klasserna i en arvshierarki fördelar ansvar över fler klasser, t ex med en basklass Stream och subklasser FileStream, CryptoStream och GZipStream.

- Arv gör det möjligt för subklasser som FileStream, CryptoStream och GZipStream att återanvända kod i Stream, utan att duplicera koden i sin egen klass.

Hur kan systemet underlätta kodskrivande i en klient som vill skriva komprimerad data till en fil? Ska en ny klass FileGZipStream definieras eller ska klienten skapa objekt av FileStream och GZipStream och därefter anropa dess metoder i tur och ordning, samt styra dataflödet (indata/utdata) mellan de båda objektens metoder?

Decorator [24] är ett mönster som gör det möjligt att utöka ett objekts ansvarsområde dynamiskt, genom att en klient sammanlänkar objekt i en kedja och därefter endast anropar

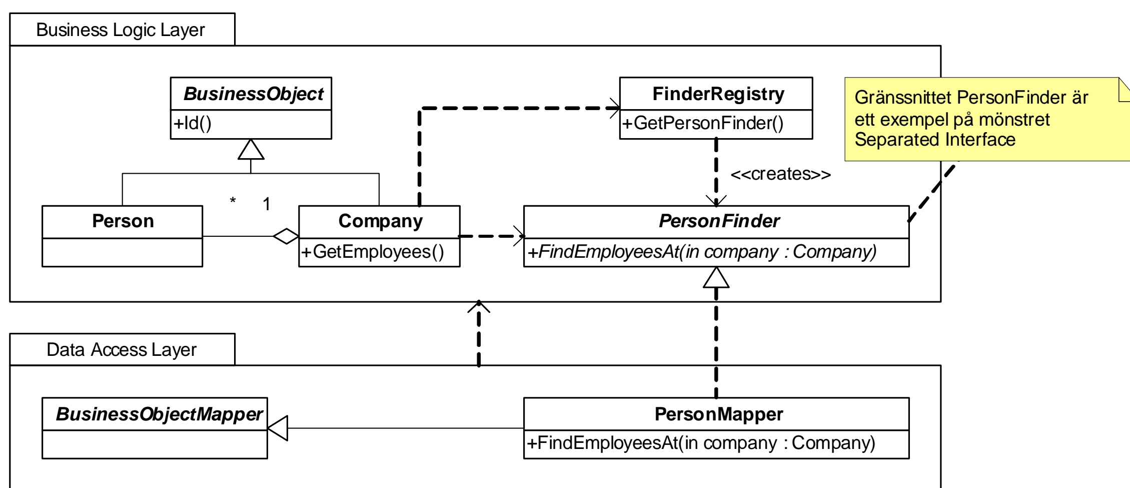


Figur 4.11: Mönstret Decorator tillämpas för strömmar i .NET

det första objektet i kedjan. Decorator har samma förhållande till Composite som en länkad lista har till ett träd. En länkad lista kan ses som ett enkelt träd, där föräldrarna (eller noderna) alltid har exakt ett barn. Composite kapslar in ett träd av sammanlänkade objekt, medan Decorator kapslar in en länkad lista av objekt. Figur 4.11 illustrerar hur .NET tillämpar mönstret Decorator för hantering av strömmar.

Mönster: Separated Interface

Hur kan affärslogiksklasser ställa frågor till eller begära data från datalagret utan att referera till dess klasser? En lösning är att definiera ett gränssnitt för sådana frågor och placera gränssnittet i affärslagret (figur 4.12) eller i ett separat lager. Fowler kallar sådana gränssnitt för “finders” [23]. Att låta DataMapper-klasser implementera ett findergränssnitt är en tillämpning av mönstret Separated Interface [23]. För att affärslagerobjekten ska kunna ställa frågor genom ett findergränssnitt, måste konkreta finderobjekt skapas. Skapandet av dessa finderobjekt kapslas lämpligen in i en separat Registry-klass [23].



Figur 4.12: Separated Interface gör affärslogiklagret oberoende av dataåtkomstlagret

Mönster: Plugin

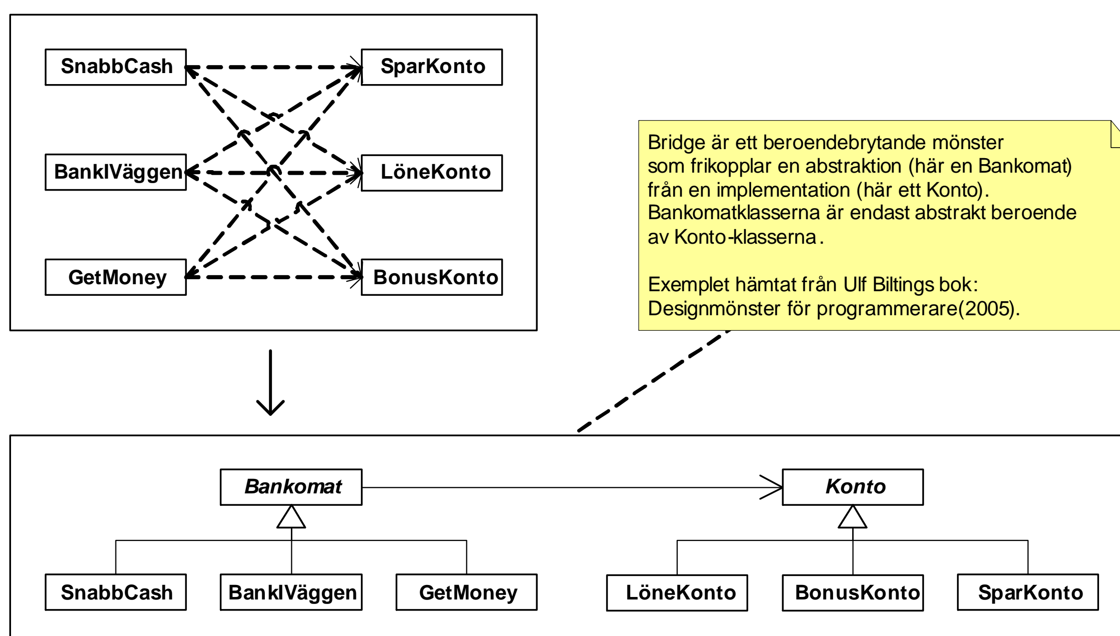
Mönstret Plugin [23] liknar Separated Interface, men istället för att den mängd klasser som implementerar gränssnittet definieras i programkoden, används någon form av textfil för att läsa in klassnamnsinformation och binda en implementation till gränssnittet med hjälp av reflection (kräver att programspråket stödjer det). Fördelen med Plugin är att nya programkomponenter kan läggas till och konfigureras utan att hela systemet måste byggas om.

Mönster: Bridge

Gamma et al introducerar mönstret Bridge med följande avsiktsbeskrivning [24]:

> "Decouple an abstraction from its implementation so that the two can vary independently."

Att frikoppla en abstraktion från sin implementation skulle felaktigt kunna förväxlas med andra mönster, t ex Separated Interface. En tillämpning av Bridge är mer än att bara definiera en abstrakt klass med tillhörande konkreta subklasser. Abstraktionen och



Figur 4.13: En tillämpning av mönstret Bridge. Källa: Ulf Bilting [10].

implementationen består här av två separata klasshierarkier, där den ena klasshierarkin utgör implementationen för den andra hierarkin. Beroendet mellan de två hierarkierna är endast abstrakt, dvs. den abstrakta basklassen för den klasshierarki som utgör *abstraktionen* i mönstret har endast en objektreferens till den abstrakta basklassen i den klasshierarki som utgör *implementationen* i mönstret. Det abstrakta beroendet bildar en brygga mellan de båda klasshierarkierna och därav mönstrets namn, Bridge. Till skillnad från bindningen mellan abstrakta och konkreta klasser, utgör de båda klasshierarkierna i Bridge-tillämpningar ingen bindning som är definierad genom arv. Bilting [10] ger ett exempel på en Bridge-tillämpning där basklasserna i de bryggade klasshierarkierna består av en bankomatklass och en kontoklass, se figur 4.13. En tillämpning av mönstret Bridge tillåter varierande implementationer utan införande av parametrar och villkorslogik. Villkorslogiken ersätts med polymorfism. Bankomatklassen i figur 4.13 kapslar inte bara in sina tre konkreta bankomatsubklasser. Bankomatklassen kapslar in hela kontoklasshierarkin för användaren av bankomaten (en extern klass kan användas för att binda en kontotyp till

bankomaten).

4.6 Kapitelsammanfattning

Detta kapitel identifierade och beskrev objektorienterade mönster som kan användas för att angripa de fyra strukturproblemen kodduplicering, villkorslogik, långa metoder och bristande inkapsling. Mönstren presenteras i tabell 4.1

Kapitel 5

Mönsterbaserade tillämpningar i INCA

5.1 Inledning

I detta kapitel beskrivs en genomförd omkonstruktion av ett delsystem i INCA. Två mönster har en framträdande roll i omkonstruktionen: Template Method och Strategy. Kvantitativa och kvalitativa mätningar har gjorts för att visa hur förändringsbarheten i delsystemet påverkas av omkonstruktionerna. Mätningarna är inte i något avseende representativa för INCA som helhet. Mätningarna omfattar endast det delsystem som berörs av omkonstruktionen. Då INCAs källkod inte är publik kommer de exempel som presenteras inte att innehålla källkod från INCA.

5.2 FormView och TestaFormular

5.2.1 Beskrivning av delsystem och strukturproblem

FormView och TestaFormular är två klasser i INCA som innehåller en hel del nästan duplicerade metoder. Majoriteten av metoderna i TestaFormular återfinns även i FormView, men implementationen av metoderna skiljer sig mellan de båda klasserna. Metoderna i

FormView innehåller bl.a. villkorstester som inte är relevanta för TestaFormular. De metoder som är gemensamma för FormView och TestaFormular ansvarar för att skapa och visa ett av de formular som skapats med INCAs formularverktyg. FormView innehåller även metoder för att skriva och läsa information till och från de grafiska IO-kontrollerna i formulären. Med IO-kontroller avses grafiska komponenter såsom textfält, kryssrutor och listor.

De metoder som är nästan duplicerade mellan FormView och TestaFormular innehåller i sin tur kod som till viss del är nästan duplicerad mellan olika kodblock. För att de båda klasserna ska kunna dela metoder och kod med varandra, måste hänsyn tas till de implementationsskillnader som finns i de båda klasserna. Den kod som är helt duplicerad kan tas bort genom att bryta ut gemensam kod till privata metoder. För att bryta ut kod som är nästan duplicerad kan privata metoder skapas och skillnader kan hanteras med parametrar och/eller utökad villkorslogik. Ett objektorienterat språk innehåller dock en mekanism som kan användas för att dela nästan duplicerad kod utan extra parametrar eller utökad villkorslogik. Mekanismen kallas polymorfism och Template Method är ett av de objektorienterade mönster som använder polymorfism för att dela kod, utan att tillföra extra parametrar och villkorslogik.

Vilka grafiska komponenter som ska ingå i ett visst användardefinierat formulär finns beskrivet i INCAs databas. Då ett formulär ska visas, körs en slinga med villkorslogik som läser av vilka typer av webbkontroller som ska skapas (renderas). För varje typ av webbkontroll som skapas finns likartad kod, men det finns också kod som är helt specifik för varje kontroll. Figur 5.1 visar med schematisk pseudokod hur villkorstester görs i INCA då webbkontroller skapas för ett formulär. MetodX som innehåller dessa villkorstester har en cyklomatisk komplexitet på 37 och innehåller 128 satser.

Villkorslogik som läser av någon form av typkod för att avgöra vilken kod som ska exekveras kan (som visats i avsnitt 4.3) ersättas med polymorfism. Ett av de mönster som kan användas för att ersätta typkodsvillkor med polymorfism är designmönstret Strategy.

```
MetodX(Data formulärdata)
{
        För varje typ av webbkontroll som ingår i formuläret
                Om webbkontrolltypen är ett textfält
                        Skapa textfältet
                        Utför logik knuten till textfältet
                        Lägg till textfältet i formuläret
                Om webbkontrolltypen är en kryssruta
                        Skapa kryssruta
                        Utför logik knuten till kryssrutan
                        Lägg till kryssrutan i formuläret
                Om webbkontrolltypen är en...
                ...
                ...
}
```

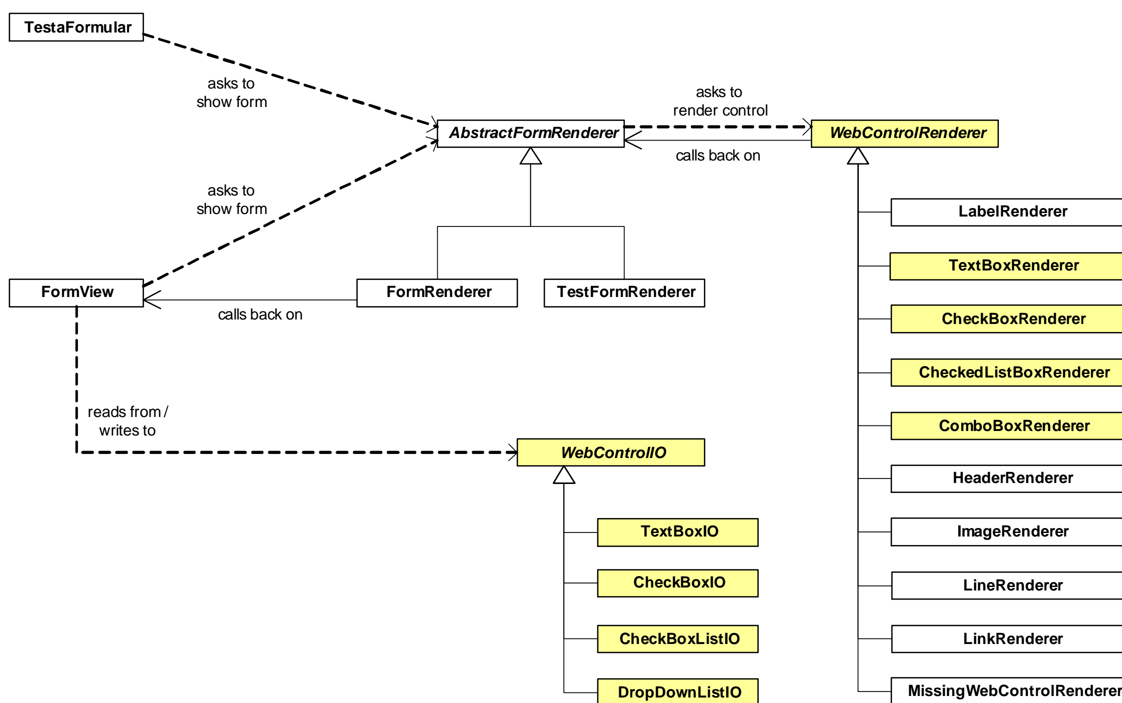
Figur 5.1: Schematisk pseudokod för hur webbkontroller skapas i INCAs formulär

För att införa mönstret Strategy och ersätta den villkorslogik som beskrivs i figur 5.1 behövs en abstrakt basklass, med subklasser för varje typ av webbkontroll. Programlogiken för skrivning och läsning till och från IO-kontroller i ett formulär, har i INCA en villkorslogik med samma struktur som den för att skapa webbkontroller. Figur 5.2 visar hur tre abstrakta basklasser kan brytas ut ur FormView och TestaFormular.

5.2.2 Omkonstruktion: Steg 1

För att genomföra omkonstruktionen och införa Strategy och Template Method, skapades först en abstrakt basklass AbstractFormRenderer med två subklasser FormRenderer och TestFormRenderer, enligt figur 5.2.

Varje metod i TestaFormular, med samma namn och parametrar som någon annan metod i FormView, flyttades till TestFormRenderer. Totalt flyttades 19 metoder från TestaFormular till TestFormRenderer. Motsvarande metoder i FormView flyttades på samma sätt till FormRenderer. De metoder som flyttats till TestFormRenderer var inte beroende av andra metoder som lämnats kvar i TestaFormular. Kompilatorn gav dock till känna att de metoder som flyttats till FormRenderer var beroende av andra metoder i FormView.



Figur 5.2: Omkonstruktion av FormView och TestaFormular

Dessa andra metoder flyttades även de till FormView. Totalt flyttades 27 metoder till FormRenderer, varav 19 delar samma gränssnitt som metoderna i TestFormRenderer.

Då metoder flyttats från FormView och TestaFormular var det också nödvändigt att skicka parametrar till konstruktorerna för TestFormRenderer och FormRenderer. De två parametrar som behövde skickas till TestFormRenderers konstruktor, behövde även skickas till FormRenderers konstruktor. De båda konstruktorerna skickar därför de två gemensamma parametrarna vidare till den gemensamma basklassen AbstractFormRenderer. FormRenderer var utöver dessa två parametrar även beroende av tio andra parametrar. Sex av dessa tio parametrar är publika egenskaper (eng. properties) i FormView. Istället för att skicka dessa sex publika egenskaper som parametrar till FormRenderers konstruktor, tar FormRenderers konstruktor en objektreferens till FormView som parameter (därav referensen från FormRenderer till FormView i figur 5.2). Totalt har FormRenderers konstruktor sex parametrar. Antalet parametrar skulle ytterligare kunna reduceras genom att ersätta

```
class FormView
...
FormRenderer formRenderer = new FormRenderer(...);
formRenderer.ShowForm(...);
...

class TestaFormular
...
TestFormRenderer formRenderer = new TestFormRenderer(...);
formRenderer.ShowForm(...);
...
```

Figur 5.3: Delegerande anrop från FormView och TestaFormular

parametrar med publika properties i FormView, om så önskas.

I FormView och TestaFormular fanns tidigare ett anrop till en lokal metod ShowForm. Detta anrop skrevs om enligt figur 5.3. Kompileringen kunde därefter göras utan fel. En manuell systemtest visade också att formulären renderades korrekt i webbläsaren.

5.2.3 Omkonstruktion: Steg 2

Efter lyckad kompilering och systemtest av de tre nya klasserna AbstractFormRenderer, FormRenderer och TestFormRenderer, kunde omkonstruktionen fortsätta med införandet av Template Methods i AbstractFormRenderer.

För att dela kod mellan FormRenderer och TestFormRenderer följdes följande principer:

- För varje metod i FormRenderer och TestFormRenderer, med samma gränssnitt och exakt duplicerad implementation, flytta metoderna till en gemensam metod i AbstractFormRenderer.
- För varje metod i FormRenderer och TestFormRenderer, med samma gränssnitt och nästan duplicerad implementation, flytta metoderna till en gemensam Template Method i AbstractFormRenderer för att hantera implementationsskillnader i FormRenderer och TestFormRenderer.

```
class FormView
...
AbstractFormRenderer formRenderer = new FormRenderer(...);
formRenderer.ShowForm(...);
...

class TestaFormular
...
AbstractFormRenderer formRenderer = new TestFormRenderer(...);
formRenderer.ShowForm(...);
...
```

Figur 5.4: Delegerande anrop från FormView och TestaFormular

- För varje metod i FormRenderer, som inte har någon matchande metod i TestFormRenderer, om metoden endast behövs av metoder i FormRenderer, låt metoden vara kvar i FormRenderer, annars flytta metoden till AbstractFormRenderer.
- För alla metoder i AbstractFormRenderer, FormRenderer och TestFormRenderer, sätt striktast möjliga åtkomstnivå (private, protected, public).

Då metoden ShowForm flyttats till AbstractFormRenderer, kunde det delegerande anropet i figur 5.3, skrivas om enligt figur 5.4. Efter införandet av mönstret Template Method i AbstractFormRenderer, genomfördes kompilering och manuell systemtest utan fel.

5.2.4 Omkonstruktion: Steg 3

Efter steg 2 fanns inte längre några metoder som var duplicerade mellan FormRenderer och TestFormRenderer. MetodX (se figur 5.1), som i steg 2 flyttades till AbstractFormRenderer, innehöll däremot kod som var nästan duplicerad. Fyra andra metoder i AbstractFormRenderer innehöll också kod som till viss del var duplicerad med koden i MetodX. MetodX som flyttats till AbstractFormRenderer innehöll även villkorslogik som genom mönstret Strategy kan ersättas med polymorfism. Införandet av Strategy ger följande önskade effekter:

- Den cyklomatiska komplexiteten för MetodX minskar från 37 till 2

```
MetodX(Data formulärdata)
{
        För varje typ av webbkontroll som ingår i formuläret
                WebControlRenderer.GetRenderer(...).RenderControl();
}
```

Figur 5.5: Mönstret Strategy ersätter villkorstester med polymorfism

- Längden på MetodX minskar från 128 till 7
- En hierarki av Strategy-klasser skapas där kod kan delas i en gemensam abstrakt basklass

Den cyklomatiska komplexiteten för MetodX minskar delvis p.g.a. att beslutspunkter flyttas från MetodX, men också p.g.a. att villkorstester helt ersätts med polymorfism. Figur 5.5 visar hur den första nivån av villkorstester i MetodX ersätts med ett polymorft anrop till en instans av någon av de konkreta renderarklasserna i figur 5.2. Att längden på MetodX minskar är en direkt följd av att den första nivån av villkorstester ersätts och att kod bryts ut till nya renderarklasser.

WebControlRenderer är en abstrakt basklass för tio konkreta subklasser och ger en abstrakt definition för metoden RenderControl, som implementeras av samtliga tio subklasser. WebControlRenderer tillhandahåller också en publik statisk tillverkarmetod GetRenderer(...), som utför en mappning av en webbkontrolltyp till en konkret instans av någon av subklasserna till WebControlRenderer. Mappningen sker genom Reflection (se avsnitt 4.3 och figur 4.6). För att slippa fem nulltester i AbstractFormRenderer, används mönstret Missing Object (även kallat Null Object) [6][22]. Ett försök till mappning av en webbkontrolltyp som inte används i INCA, resulterar alltid i att metoden GetRenderer(...) returnerar en instans av MissingWebControlRenderer. Metoden RenderControl i klassen MissingWebControlRenderer gör ingenting. Alternativt kan RenderControl definieras om i MissingWebControlRenderer till att skriva till INCAs fellogg eller att direkt informera formulärkonstruktören om ett fel.

```
class WebControlRenderer {
    protected WebControl webControl;
    protected abstract WebControl GetWebControl();
    public WebControlRenderer(...){
        webControl = GetWebControl();
    }
}
class TextBoxRenderer {
    protected override WebControl GetWebControl(){
        return new TextBox();
    }
}
```

Figur 5.6: Factory Method för att tvinga initiering av objektreferens

Från MetodX skapades fem konkreta subklasser till WebControlRenderer. Från fyra andra metoder i AbstractFormRenderer skapades ytterligare fyra konkreta subklasser till WebControlRenderer. Dessa nio klasser delar kod i den gemensamma basklassen WebControlRenderer. Alla nio subklasser har det gemensamt att de skapar och modifierar en subklass till klassen WebControl. En referens till ett WebControl-objekt finns därför som ett attribut i WebControlRenderer och tilldelas i de konkreta renderarklasserna vid skapandet av ett renderarobjekt. Specifik kod som endast berör ett textfält skrivs i TextBoxRenderer. Kod som endast berör kryssrutor skrivs i CheckBoxRenderer, osv. Det finns metoder som är gemensamma för olika typer av webbkontroller, dvs. metoder som är definierade i klassen WebControl. Kod som anropar sådana metoder kan skrivas i WebControlRenderer. Det finns vidare metoder som är gemensamma för olika typer av listkontroller. Kod som anropar listkontrollmetoder kan också skrivas i WebControlRenderer. Varje specifik implementation av RenderControl i renderarklasserna definierar vilka metoder i WebControlRenderer som anropas av respektive renderarklass. Implementationen styrs alltså inte av en mängd villkor som testar vilken typ av webbkontroll som ska renderas i det aktuella formuläret.

En Factory Method [24] definierades i WebControlRenderer för att tvinga subklasserna att initiera en objektreferens i WebControlRenderer (se figur 5.6). Factory Methods

är användbara då basklasser är beroende av att subklasser initierar diverse attribut i basklassen. TextBoxRenderer i figur 5.6 skulle i sin konstruktor kunna göra tilldelningen `webControl = new TextBox()` och därmed göra fabriksmetoden GetWebControl() överflödig. Vitsen med att definiera en abstrakt Factory Method i WebControlRenderer är att påminna programmerare att initiera webControl i subklasserna. Subklasserna måste implementera GetWebControl() för att en lyckad kompilering ska kunna göras. Detta tvång är ett sätt att förebygga att nullreferensundantag kastas (och felsöks) till följd av att en objektreferens glömts bort att initieras.

Efter införandet av mönstret Strategy genomfördes kompilering och manuell systemtest utan fel.

5.2.5 Omkonstruktion: Steg 4

Till skillnad från TestaFormular har FormView metoder för att skriva och läsa till och från IO-kontroller i ett formulär. Strukturen för den metod i FormView som läser från IO-kontroller i aktuellt formulär följer samma mönster som MetodX i figur 5.1. Detsamma gäller den metod i FormView som skriver till IO-kontroller. Då skriv- och läsmetoderna i FormView har samma strukturproblem som MetodX, kan samma mönster (Strategy) tillämpas på dessa två metoder. Här uppstår intressanta frågor:

1. Ska två abstrakta basklasser skapas, t ex WebControlReader och WebControlWriter?

2. Ska en abstrakt basklass skapas, t ex WebControlIO, som hanterar både läsning och skrivning?

3. Ska metoder för läsning och skrivning integreras med renderarklasserna?

I samband med dessa frågeställningar undrade Håkan Cederberg (handledare från Sogeti för detta examensarbete) om frågeställningarna kan jämföras med frågeställningen hur

långt normalisering ska drivas i ett databassystem. En informationssökning kring objektorientering och normalisering gav till känna att denna jämförelse också har gjorts av andra (se avsnitt 2.12).

Om metoder för läsning och skrivning integreras i WebControlRenderer och dess subklasser, uppstår två negativa effekter. För det första blir det svårt att sätta ett informativt namn på dessa klasser då de representerar och gör flera saker (renderar, skriver till och läser från webbkontroller). Hur bra är namn som t ex WebControlHandler eller WebControlManager? Hur mycket information kan kommuniceras endast genom klassnamnet? För att veta vad en WebControlManager gör måste dess metoder inspekteras. IO-kontroller utgör endast en delmängd av alla webbkontroller som kan ingå i ett formulär i INCA. Den andra negativa effekten av att integrera skriv- och läsmetoder i WebControlRenderers klasshierarki är därför att den delmängd renderarklasser som inte renderar IO-kontroller, tvingas ge (tomma) implementationer för de abstrakta metoder som inte är relevanta för dessa klasser. Är inte detta samma problem som uppstår vid otillräcklig normalisering av data i en databas? Fält som inte är relevanta för vissa poster tvingas ge ett värde, t ex NULL. En klasshierarki där subklasser tvingas implementera metoder de inte är intresserade av är en dålig lukt som Fowler kallar “Refused Bequest” [22]. Att hantera skrivning och läsning till och från IO-kontroller i en separat klasshierarki undviker stanken från “Refused Bequest”.

Om separata basklasser skapas för läsning och skrivning, t ex WebControlReader och WebControlWriter, blir de konkreta subklasserna oerhört små och en ökad overhead i form av kodrader tillkommer. Ett subjektivt argument är också att komplexiteten som tillkommer genom det ökade antalet klasser i systemet överskuggar den ökade läsbarheten som ges av separata klasser för läsning och skrivning.

Om metoder för läsning och skrivning kombineras i samma klasshierarki går det ändå att ge informativa namn på klasserna, t ex WebControlIO. IO (Input/Output) är ett suffix som kommunicerar vad klassen representerar och gör. I denna omkonstruktion har valet

gjorts att kombinera läsning och skrivning i en hierarki av Strategy-klasser, där WebControlIO är abstrakt basklass (se figur 5.2). Om det, trots framförda argument, ändå anses att läsning och skrivning bör integreras i WebControlRenderer med subklasser, har de renderarklasser i figur 5.2 som renderar IO-kontroller markerats med samma färg som klasserna för läsning och skrivning.

5.3 Fler omkonstruktioner i INCA

Den initiala tanken var att detta kapitel skulle innehålla beskrivningar av fler omkonstruerade delsystem i INCA. Fler delsystem har identifierats i INCA, där införandet av objektorienterade mönster förväntas bidra till ökad förändringsbarhet. Då ett flertal av dessa omkonstruktioner innebär införande av samma eller liknande mönster som de som införts genom omkonstruktionen av FormView och TestaFormular är det tveksamt om genomförandet, beskrivningen, motiveringen och mätningen av dessa omkonstruktioner utgör värdefulla bidrag till denna examensrapport. Då inkluderandet av fler omkonstruktioner i detta kapitel också innebär mindre tid för övriga delar av rapporten, har valet gjorts att inte genomföra och beskriva fler omkonstruktioner i INCA.

5.4 Kapitelsammanfattning

Detta kapitel har beskrivit en genomförd omkonstruktion av ett delsystem i INCA, där mönstren Template Method och Strategy har använts för att angripa strukturproblemen duplicerad kod, villkorslogik och långa metoder. Template Method infördes primärt för att förena kod som var nästan duplicerad mellan två klasser. Strategy användes primärt för att ersätta villkorslogik med polymorfism. Tillämpningen av mönstret Strategy gjorde det även lämpligt att använda två andra mönster, Factory Method och Missing Object.

Kapitel 6

Resultat och mätningar

6.1 Inledning

Detta kapitel presenterar resultatet av de mätningar som utfördes före och efter den omkonstruktion som beskrevs i kapitel 5. Mätningarna använder de mått som beskrivits i kapitel 3 och visar att omkonstruktionen reducerat duplicerad kod, villkorslogik och längden på metoder i det omkonstruerade delsystemet.

6.2 FormView och TestaFormular

6.2.1 Mjukvarumått för omkonstruktionen

Tabell 6.1 visar mjukvarumått för omkonstruktionen av FormView och TestaFormular. Omkonstruktionen angriper tre typer av strukturproblem:

- Kodduplicering
- Uppsvälld villkorslogik
- Långa metoder

Kategori	**Mjukvarumått**	**Före**	**Efter**	**Reducering**
Delsysteminfo	Totalt # klasser	2	21	
	Totalt # metoder	76	154	
	Medel # metoder/klass	38	7,3	
Kodduplicering	# kodrader* (LOC)	1691	1589	6 %
	# satser* inkl namnrymdsdekl.	1076	904	16 %
	# satser exkl namnrymdsdekl.	1058	817	23 %
	# satser i metoder	946	572	40 %
Långa metoder	Medel # satser/metod	12,5	3,7	70 %
	Max # satser/metod	128	24	80 %
	# metoder längre än 10 satser	27	8	70 %
	# metoder längre än 15 satser	11	1	91 %
	# metoder längre än 20 satser	10	1	90 %
Villkorslogik	Medel cyklomatisk komplexitet	4,2	1,8	57 %
	Max cyklomatisk komplexitet	37	8	78 %
	Medel kodblocksdjup*	3,3	1,8	45 %
	Max kodblocksdjup	9	5	44 %

* Se kapitel 3 för en förklaring av kodrader, satser och kodblocksdjup

Tabell 6.1: Mjukvarumått för delsystemet före och efter omkonstruktion

Reducerad kodduplicering

Koddupliceringen i FormView och TestaFormular har genom omkonstruktionen minskat med 6-40 % beroende på hur beräkning görs (se tabell 6.1). FormView och TestaFormular innehöll före omkonstruktionen 19 metoder som var nästan duplicerade mellan de båda klasserna. För att dela dessa metoder infördes klasserna FormRenderer och TestFormRenderer, samt en abstrakt basklass AbstractFormRenderer. De 19 nästan duplicerade metoderna skulle kunna ha delats genom att flytta metoderna till en enda ny klass. En sådan omkonstruktion skulle dock leda till ett behov av extra villkorslogik och parametrar för att hantera implementationsskillnader i de delade metoderna. Införandet av två subklasser och en gemensam basklass, gjorde det möjligt att hantera implementationsskillnader i de delade metoderna genom att införa Template Methods i basklassen. Template Method gjorde det möjligt att dela kod utan behov av extra villkorslogik och parametrar. Implementationsskillnaderna hanteras av subklasserna, såsom beskrivits i avsnitt 4.2.2.

Tre långa metoder i FormView (varav en var duplicerad i TestaFormular) innehöll villkorslogik som kunde reduceras genom införande av mönstret Strategy. Som ett resultat av införandet av Strategy kunde gemensam (duplicerad) logik för respektive konkret Strategy-klass delas i de två abstrakta Strategy-basklasserna WebControlIO och WebControlRenderer. Den duplicerade koden i de tre långa metoderna i FormView skulle kunna ha reducerats genom att dela den i privata metoder i FormView (alternativt AbstractFormRenderer). Införandet av Strategy gjorde det möjligt att reducera duplicerad kod, samtidigt som villkorslogiken reducerades. Genom införandet av nya klasser minskade också antalet kodrader i FormView. Det stora antalet kodrader i FormView var i sig ett problem, enligt utvecklarna (se avsnitt 6.2.2).

Reducerad villkorslogik

Tabell 6.1 visar hur den cyklomatiska komplexiteten minskat genom omkonstruktionen av FormView och TestaFormular. Införandet av de båda hierarkierna av Strategy-klasser innebar att vissa villkorstester ersattes med polymorfism (se figur 5.1 och 5.5). Totalt ersattes 13 if-satser (18 om villkorstesterna i den duplicerade metoden i TestaFormular räknas) med anrop till Strategy-klasser. De ersatta villkorstesterna var samtidigt orsak till de längsta och mest komplexa (cyklomatisk komplexitet) metoderna i FormView och TestaFormular.

Att den genomsnittliga cyklomatiska komplexiteten minskat genom omkonstruktionen beror först och främst på att beslutspunkterna spridits ut i det ökade antalet metoder (76 metoder innan omkonstruktionen och 154 efter). Den ackumulerade cyklomatiska komplexiteten för alla metoder i delsystemet var innan omkonstruktionen 319 = 76 * 4,2 och efter omkonstruktionen 277 = 154 * 1,8. Då cyklomatisk komplexitet för en metod är detsamma som antalet beslutspunkter i metoden + 1, kan det totala antalet beslutspunkter i delsystemet före omkonstruktionen beräknas till 243 = 319 - 76. Antalet beslutspunkter i delsystemet efter omkonstruktionen beräknas på samma sätt till 123 = 277 - 154. Antalet

beslutspunkter i det omkonstruerade delsystemet har alltså reducerats från 243 till 123, dvs. en reducering med 120 beslutspunkter (ca 50 %). 13 av dessa beslutspunkter beror på att villkorstester har ersatts av anrop till Strategy-klasser. Övriga 107 beslutspunkter hör samman med reducering av antalet duplicerade metoder och duplicerade villkorssatser i metoder.

Kortare metoder

Tabell 6.1 visar hur längden på metoder minskat genom omkonstruktionen av FormView och TestaFormular. Reduceringen av kodduplicering och villkorslogik medför kortare metoder. Kortare metoder gör det lättare att dela kod. Den genomsnittliga längden på en metod innan omkonstruktion var 12,5 satser och efter omkonstruktion 3,7 satser. Det genomsnittliga antalet *kodrader* per metod kan uppskattas genom att använda förhållandet mellan kodrader och satser. Antalet kodrader per metod före omkonstruktion var ungefär 19,6 = (1691 / 1076) * 12,5. Antalet kodrader per metod efter omkonstruktion var ungefär 6,5 = (1589 / 904) * 3,7.

6.2.2 INCA-gruppens tyckande om omkonstruktionen

Den genomförda omkonstruktionen presenterades och förklarades den 16 november 2006 för tre av INCA:s utvecklare. Samtliga tre utvecklare var medvetna om de strukturproblem som fanns i FormView och TestaFormular och ser gärna att den genomförda omkonstruktionen införs i INCA. Samtliga tre utvecklare kunde snabbt ta till sig och se fördelarna med Template Method och Strategy. De båda mönstren förklarades under en dryg timme lång "mönsterkväll" i oktober 2006 och repeterades i samband med presentationen av omkonstruktionen. Presentationen av omkonstruktionen genomfördes på ca en timme där både klassdiagram och kod granskades.

Att två klasser (FormView och TestaFormular) efter omkonstruktionen blivit 21 klasser ses uteslutande som positivt, då storleken på FormView och TestaFormular upplevts som

besvärande. Den enda kritiken mot omkonstruktionen är att felsökning i Visual Studio är enklare om metoderna är långa. Det är lätt att förlora kontrollen över felsökningen då programflödet hoppar fram och tillbaka mellan klasser och metoder. Alla är dock helt överens om att reducering av kodduplicering och villkorslogik överskuggar den försvårade felsökningen.

6.2.3 Likheter med mönstret Bridge

Den del av klassdiagrammet i figur 5.2 som omfattar klasshierarkin med AbstractFormRenderer och klasshierarkin med WebControlRenderer har samma struktur som mönstret Bridge. Till skillnad från en Bridge-tillämpning representerar inte subklasserna till WebControlRenderer kompletta implementationer av subklasserna till AbstractFormRenderer. Det är därför olämpligt att benämna klasstrukturen som en Bridge-tillämpning, även om klassdiagrammet är identiskt med Bridge. Det finns flera mönster vars klassdiagram är snarlika eller identiska, men där mönstren ändå ges olika namn. State och Strategy är exempel på två mönster vars strukturer påminner om varandra, men där mönstrens syften och tillämpningar skiljer sig åt.

6.3 Kapitelsammanfattning

Detta kapitel presenterade resultatet av mätningarna av den omkonstruktion som beskrevs i kapitel 5. Mätningarna visar att kodduplicering i det omkonstruerade delsystemet reducerats med 6-40 %. Den genomsnittliga längden på metoder minskade med 70 %. Den genomsnittliga cyklomatiska komlexiteten per metod minskade med 57 % och det totala antalet beslutspunkter i delsystemet minskade med 50 %, vilket indikerar en minskning av villkorslogik.

Kapitel 7

Slutsats

Denna rapport stödjer hypotesen att objektorienterade mönster kan tillämpas för att öka ett mjukvarusystems förändringsbarhet, men den stora slutsatsen i denna rapport handlar mer om ett *förhållningssätt* till mönster, där mönster blir en del av en annan designstrategi, nämligen kontinuerlig omkonstruktion.

Rapporten utgick från hypotesen att tillämpningen av objektorienterade mönster gör mjukvarusystem mer förändringsbara. Hypotesen gav upphov till ett antal frågeställningar. Hur kan hypotesen prövas? Vad är ett mönster? Vad är förändringsbarhet? Finns det något objektivt sätt att visa att ett mjukvarusystem är mer förändringsbart än ett annat eller är frågan om förändringsbarhet rent subjektiv? Vad kännetecknar mjukvarusystem som är mindre förändringsbara?

Intresset för mönster fanns hos författaren redan innan detta examensarbete påbörjades, men att studera mönster utifrån ett förändringsbarhetsperspektiv var nytt. För att förstå hur mönster kan bidra till ökad förändringsbarhet studerades litteratur om mönster. För att försöka svara på frågan om vad förändringsbarhet är, hur den kan uppnås och mätas i ett mjukvarusystem, gjordes även litteratursökningar och studier om förändringsbarhet. Vilka faktorer påverkar förändringsbarheten positivt och vilka påverkar den negativt? Finns det strukturer som gör mjukvara mindre förändringsbar? Då svar söktes på

dessa frågor, var "refactoring" (omkonstruktion) ett ord som återkom. Då omkonstruktioner görs för att upprätthålla förändringsbarheten i mjukvarusystem, var det naturligt att fråga om det fanns ett samband mellan mönster och omkonstruktion. Då svar söktes på den frågan, upptäcktes boken "Refactoring to Patterns" av Joshua Kerievsky [27]. Kerievsky kontrasterar användningen av mönster i en planerad "upfront" design med införandet av mönster genom omkonstruktion.

Omkonstruktioner angriper strukturproblem. Att införa mönster i ett mjukvarusystem genom omkonstruktion är ett förhållningssätt till mönster, där mönster ses som verktyg för att komma tillrätta med strukturproblem, snarare än att ses som eleganta och flexibla lösningar, vilket de också är. Omkonstruktioner görs inte på grund av spekulationer om framtida ändringskrav. Omkonstruktioner görs för att bli av med strukturproblem, vilka kan identifieras som dåliga lukter. Denna rapport identifierade fyra mätbara strukturproblem vars dåliga lukter förekommer både enskilt och blandade med varandra:

- Duplicerad kod
- Villkorslogik
- Långa metoder
- Bristande inkapsling

Det finns fler mätbara strukturproblem som kan beskrivas i form av dåliga lukter. Stora klasser är ett exempel [22]. De fyra strukturproblem som identifierats och beskrivits i rapporten är dock tillräckliga för att exemplifiera ett arbetssätt vid mjukvaruutveckling:

1. Spekulera inte om framtida ändringskrav. Tillämpa enklast möjliga lösning som fungerar.
2. Då strukturproblem upptäcks, manuellt eller med hjälp av ett verkyg, omkonstruera.

3. I samband med omkonstruktionen, undersök om det finns ett användbart mönster, där en tillämpning av mönstret löser strukturproblemet.

Att tillämpa enkla lösningar är inte liktydigt med att inte tänka. Enkla lösningar utesluter heller inte tillämpningar av mönster, men det är rimligt att ifrågasätta om en lösning är enkel om den förutsätter att problemlösaren först fördjupar sig i olika tänkbara lösningar, analyserar alla fördelar och nackdelar med respektive lösning, samt väljer sin lösning baserat på förväntade ändringar. Vad som är en enkel lösning är subjektivt och beroende av varje enskild utvecklares erfarenhet. Enkla lösningar väljs baserat på problemlösarens erfarenhet. Enkla lösningar innebär att problemlösaren väljer den lösning som verkar enklast för stunden, fullt medveten om att andra lösningar existerar och kan införas i ett senare skede då ett strukturproblem uppstått och då problemlösarens erfarenhet vuxit till en nivå där han eller hon ser en enkel lösning att rätta till problemet, kanske med hjälp av mönster.

Att införa mönster genom omkonstruktion (dvs. genom en framväxande design) innebär att mjukvaruutvecklingen inte behöver stanna upp pga. av långa diskussioner om design. Det finns dock tillfällen då det är motiverat att stanna upp och fundera. Var går gränsen för vad som kan omkonstrueras? Hur mycket designskuld kan ett system bära innan skulden blir så stor och räntan så hög att det blir praktiskt omöjligt att betala av skulden? Oavsett var den teoretiska gränsen går för vad som kan omkonstrueras med framgång, finns det en praktisk gräns som bestäms av varje utvecklares erfarenhet. En designskuld måste övervakas och kontrolleras, så att den inte skenar iväg och blir okontrollerbar. Omkonstruktion måste vara en regelbunden åtgärd.

Att välja den enklaste lösningen som fungerar är fel val om problemlösaren fullt medvetet försätter sig i en så stor skuld att han eller hon inte ser något enkelt sätt att betala av skulden. Ingen klok människa hoppar från ett flygplan utan fallskärm. Den som hoppar har en fallskärm, förväntar sig att den fungerar och utlöser den innan det är för sent. Den som utvecklar mjukvara kan tillåta strukturproblem i små kontrollerbara mängder, men när de överstiger en viss gräns måste de åtgärdas. Den exakta gränsen kan vara svår att fastställa

då strukturproblemet inte helt kan mätas med ett enda objektivt mått. Mjukvarumått kan användas för att definiera en generell gräns för strukturproblem. Då strukturproblemen överstiger gränserna, bör det berörda delsystemet inspekteras manuellt.

Att införa mönster genom omkonstruktion innebär inte att användningen av mönster minskar eller att inlärning av nya mönster inte är viktigt. Mönsterinförande omkonstruktion innebär att användningen av mönster inriktas på att lösa befintliga strukturproblem. En sådan inriktning betyder inte att strukturproblem först måste skapas för att mönster ska kunna tillämpas. En utvecklare med stor erfarenhet ser strukturproblem innan de uppstår. Han eller hon gör då, utifrån sin erfarenhet, en kvalificerad gissning om den omedelbara komplexiteten och tidskostnaden för införandet av ett mönster överskuggar kostnaden för det strukturproblem som uppstår om mönstret inte tillämpas. Det är inte självklart att alla switch-satser ska ersättas med polymorfism, men varje erfaren objektorienterad utvecklare bör slås av tanken att polymorfism är ett alternativ till en switch-sats.

7.1 Författarens reflektioner

Vid sidan av de slutsatser som dras i ett examensrapport kan det vara intressant att även inkludera författarens egna reflektioner. De reflektioner som gjorts under examensarbetets gång och som relaterar till, men är skilda från rapportens "röda tråd" presenteras här i form av frågor med delvis subjektiva svar.

7.1.1 Är polymorfism som alternativ till villkorslogik tillräckligt betonat i undervisningen?

Kurser i objektorienterad programmering talar om OO-språkens egenskaper arv, inkapsling och polymorfism. Rappporten har redan tidigare påpekat att begreppet inkapsling ofta begränsas till inkapsling av attribut i klasser. Rapporten har också beskrivit virtuella metoder som den objektorienterade egenskap som är svårast att ta till sig för en programmerare

med en bakgrund i ett procedurellt språk som C.

Objektorienterade mönster såsom Strategy och State undviker villkorslogik genom användandet av polymorfism och virtuella metoder. För författaren till denna rapport var polymorfism som villkorslogiksersättare en aha-upplevelse under detta examensarbete. Borde inte en sådan aha-upplevelse ha kommit tidigare? Fullt övertygad om att denna aspekt av polymorfism inte betonats i undervisningen bestämde sig författaren för att granska den bok som användes som kurslitteratur då polymorfism introducerades för författaren,"C++ How To Program" [14]. Till stor förvåning upptäcktes då att polymorfism faktiskt nämndes som objektorienterat alternativ till switch-satser då polymorfism introducerades. Det kan dock ifrågasättas om tillräcklig betoning läggs på villkorslogik som ett strukturproblem. Är polymorfism bara ett trevligt alternativ för den som vill programmera på ett annorlunda sätt eller är polymorfism ett sätt att undvika strukturproblem som ökar mjukvaruföretagens kostnader för underhåll?

7.1.2 Är det lättare att lära sig mönster ur ett omkonstruktionsperspektiv?

De designmönster som nämns av Gamma et al [24] beskrivs dels som eleganta lösningar och dels som mål för omkonstruktioner. Kerievsky är involverad i studiegrupper om designmönster och har skrivit en webbartikel med vägledning till att driva studiegrupper om designmönster [26]. Han får ofta höra andra säga att mönster är lättare att lära sig då deras tillämpning diskuteras som en del av omkonstruktioner [27]. Författaren till denna rapport delar den uppfattningen, men är samtidigt medveten om att tidigare studier om designmönster påverkar den inlärning som skett under detta examensarbete.

Gamma et al diskuterar de strukturproblem som angrips då mönstren tillämpas genom omkonstruktion, men som redan nämnts i kapitel 2.8 är den avsikt som nämns för varje mönster ofta långt från strukturproblemsfokuserad. Shalloway och Trott introducerar designmönster genom att först beskriva ett strukturproblem (ignorera figuren som inte är

inkluderad i följande exemplifierande text) [31]:

> "Before showing a solution and deriving the Bridge pattern, I want to mention a few other problems (beyond the combinatorial explosion).
> Looking at Figure 10-3, ask yourself what else is poor about this design.
> Does there appear to be redundancy?
> Would you say things have strong cohesion or weak cohesion?
> Are things tightly or loosely coupled?
> Would you want to have to maintain this code?"

Det finns böcker inom säkerhet som talar om att känna sin fiende. En omkonstruktör är ofta sin egen fiende då en omkonstruktion kan vara framtvingad av omkonstruktören själv. Det är då relevant att vara självkritisk och ställa sig frågor. Om omkonstruktion är en regelbunden aktivitet borde då inte en större medvetenhet utvecklas, så att det ringer en klocka när långa metoder skrivs eller då "kopiera-klistra in" används flitigt för att lägga till "ny" kod?

7.1.3 Finns det utrymme för en kurs i omkonstruktion, mönster och testning?

Vid Karlstads universitets datavetenskapsavdelning finns en kurs som introducerar designmönster. Omkonstruktion som begrepp tas upp i samband med att Extreme Programming lärs ut i kurserna Software Engineering DAVC19 [34] och Projektarbete DAVC20 [35]. Den bok som används som kurslitteratur i dessa båda kurser, dvs Becks "Extreme Programming Explained" [5], ger mycket lite information om omkonstruktion som teknik och ger mest en överblick av Extreme Programming som utvecklingsmetodik. Då XP praktiseras i kursen Projektarbete, finns testning som ett inslag i kursen och ett uttryckligt krav för utvecklingsmodellen i kursen är:

> "Automatisk kontinuerlig test: alla tester (där så är möjligt och rimligt) ska ske med automatiska program." [36]

Vem bedömer om tester är rimliga eller möjliga? Studenterna? Kursledaren? Om studenterna ska göra den bedömningen, har studenterna då fått tillräcklig undervisning i hur enhetstester skrivs? Om ingen grundlig undervisning sker om testning och studenterna uppmanas skriva tester då de själva bedömer att "så är möjligt och rimligt", är då inte risken överhängande att få testfall skrivs, samt att studenternas bild av testning blir att det är något som är svårt och inte helt nödvändigt?

Testning är ett stort ämne i sig, åtminstone tillräckligt stort för att inte behandlas i detta examensarbete. Enhetstestning är starkt knuten till omkonstruktion, där enhetstester utgör ett skyddsnät för omkonstruktioner. I den omkonstruktion som beskrivits i kapitel 5 valdes att inte skriva några enhetstester, då det omkonstruerade delsystemets ansvarsområde i första hand var att rendera ett formulär. Det hade varit möjligt att skriva tester för delsystemet, men i detta fall var en visuell test att formuläret visades på skärmen en test som var tillräcklig. Även om den genomförda omkonstruktionen kunde genomföras på ett tillförlitligt sätt utan enhetstester skulle framtida ändringar i delsystemet på sikt kunna gynnas av automatiserade enhetstester, eftersom sådana kan utföras snabbare än manuella tester (granskningar).

Studenter på Karlstads universitet blir sällan utsatta för större system. Programmeringsuppdrag eller laborationer utgörs nästan uteslutande av små tillämpningar som varje student själv (ensam eller tillsammans med en annan student) utvecklar från grunden. Det finns undantag där lärare låter elever börja utveckla från en existerande "kodbas". Sådana initiativ kan enkelt motiveras med att majoriteten av all utveckling består av underhåll (figur 2.1).

Tid är en begränsande faktor för vad som kan ingå i en ingenjörsutbildning, men kanske finns det ändå utrymme för en D-nivåkurs med inriktning på omkonstruktion, mönster och testning? Ett förslag är att utnyttja möjligheterna med öppen källkod. De mjukvarumått

som presenterats och använts i detta examensarbete är triviala och kan dessutom tillämpas med fritt tillgängliga verktyg. Software Metrics, Software Engineering (eller Projektarbete), samt Objektorienterade Designmetoder skulle kunna vara lämpliga förkunskapskrav till en kurs där strukturproblem i ett större öppenkällkodssystem identifieras med hjälp av ett mätverktyg och angrips genom omkonstruktion, mönster och testning.

Ett annat förslag som kombinerar inlärning av mönster och inlärning av enhetstestning är att studera JUnit (eller NUnit) som ramverk för enhetstestning. Enligt Kerievsky [27] är JUnit ett testramverk där många mönster tillämpats. Han menar också att dess utvecklare, Kent Beck (författare till *Extreme Programming Explained* [5]) och Erich Gamma (författare till *Design Patterns* [24]), drivit utvecklingen av JUnit genom mönsterinförande omkonstruktioner från version 1.0 till dess nuvarande version. En studie av JUnit skulle öka förståelsen både för mönster och testning.

7.2 Framtida arbete

Objektorienterade mönster tar tid att lära sig och tillämpa. Tillämpningen av mönster underlättar förståelsen av ett system, under förutsättning att mönstren är kända av den som granskar systemet. Inlärningstid för mönster, mönsterkunskaper och dess påverkan på den upplevda läsbarheten är aspekter som berörts i denna rapport, men som inte behandlats i detalj. Det är inte självklart att mönster ökar läsbarheten i projekt där mönsterkunskaperna hos utvecklarna varierar. Ett förslag till ett framtida examensarbete kan vara att testa programmerares mönsterkunskaper, använda mjukvarumått för att profilera dessa programmerare efter den kod de skriver och sedan utföra tester på hur samma programmerare upplever läsbarheten för olika typer av kod. Om den kod som skrivs luktar illa, i vilken grad beror det på bristande rutiner och i vilken grad beror det på att de upplever koden som lättare att förstå? Hur mäts en programmerares helhetsbild av ett system?

Under detta examensarbete genomfördes endast en omkonstruktion av ett delsystem

med storleken 1691 kodrader. Omkonstruktionen visade hur mönster kan införas för att reducera kodduplicering, villkorslogik och långa metoder. Den genomförda omkonstruktionen kommer med stor sannolikhet att införas i INCA efter detta examensarbete. Då fler ändringar gjorts av INCAs utvecklare i det berörda delsystemet under hösten 2006, kommer den "experimentellt" genomförda omkonstruktionen som presenterats i denna rapport att göras om på den senaste versionen av INCA-systemet. Det är mindre riskfyllt att göra om omkonstruktionen på den senaste versionen av INCA med den genomförda omkonstruktionen som mall, än att analysera 1500-2000 kodrader för att kartlägga alla ändringar av delsystemet efter version 1.2 av INCA.

Strukturer som motiverar fler mönsterbaserade omkonstruktioner har identifierats i INCA där ökad förändringsbarhet förväntas uppnås genom en bättre design och struktur. Beskrivningar av rekommenderade omkonstruktioner (utan genomförande och jämförande mätningar) kommer att ingå i en separat rapport som presenteras för Sogeti Karlstads INCA-grupp i januari 2006, då författaren till denna examensrapport också fortsätter sitt samarbete med Sogeti Karlstad.

Referenser

[1] Stephen T. Albin. *The Art of Software Architecture: Design Methods and Techniques.* Wiley, 2003. 15

[2] Cristopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, 1977. 10, 11

[3] Scott W. Ambler. *Introduction to Class Normalization.* Ambysoft Inc.
[www] http://www.agiledata.org/essays/classNormalization.html
(Hämtat 2006-11-30). 32

[4] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition.* Addison Wesley, 2003. 1, 27

[5] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 1999. 1, 2, 12, 104, 106

[6] Kent Beck. *Test-Driven Development By Example.* Addison Wesley, 2005. 21, 66, 87

[7] Marco Bellinaso. *ASP.NET 2.0 Website Programming: Problem - Design - Solution.* Wrox Press, 2006. 68

[8] PerOlof Bengtsson. *Architecture-Level Modifiability Analysis.* Blekinge Institute of Technology, 2002. 15, 27

[9] Edward V. Berard. *Abstraction, Encapsulation, and Information Hiding.* The Object Agency.
[www] http://www.toa.com/pub/abstraction.txt
(Hämtat 2006-11-27). 9

[10] Ulf Bilting. *Designmönster för programmerare.* Studentlitteratur, 2005. xiii, 14, 23, 29, 56, 64, 75, 79

[11] Michael Blaha and James Rumbaugh. *Object-Oriented Modeling and Design with UML, Second Edition.* Prentice-Hall, 2005. 25, 26

[12] C. Bohm and G. Jacopini. Flow diagrams, turing machines, and languages with only two formation rules. *Communications of the ACM, Vol. 9, No. 5*, pages 336–371, May 1966. 8

[13] Frank M. Carrano, Paul Helman, and Robert Veroff. *Data Abstraction and Problem Solving with C++*. Addison Wesley, 1998. 9

[14] Harvey M. Deitel and Paul J. Deitel. *C++ How to Program, Third Edition*. Prentice Hall, 2001. 8, 9, 103

[15] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Elsevier Science, 2003. 23, 70, 71

[16] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118, September 1999. 23

[17] Bruce Eckel. *Thinking in C++, 2nd ed. Volume 1*. Prentice Hall, 2000. 8, 9

[18] Eric Evans. *Domain Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 2004. 71, 72, 73

[19] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics - A Rigorous and Practical Approach*. PWS Publishing Company, 1997. 22, 38, 43, 47

[20] Martin Fowler. *CannotMeasureProductivity*. martinfowler.com, 2003.
[www] http://www.martinfowler.com/bliki/CannotMeasureProductivity.html
(Hämtat 2006-12-08). 39

[21] Martin Fowler. *Is Design Dead?* martinfowler.com, 2004.
[www] http://www.martinfowler.com/articles/designDead.html
(Hämtat 2006-11-24). 1, 14

[22] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison Wesley, 1999. 1, 2, 9, 10, 12, 14, 15, 16, 21, 22, 24, 26, 29, 33, 40, 50, 65, 66, 67, 71, 87, 90, 100, 124

[23] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2003. 1, 10, 11, 14, 15, 29, 33, 34, 52, 58, 60, 67, 68, 69, 77, 78, 123, 124

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1995. xv, 1, 7, 10, 11, 12, 14, 15, 19, 25, 27, 28, 31, 34, 51, 54, 60, 69, 71, 73, 75, 76, 78, 88, 103, 106, 123, 124, 125

[25] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison Wesley, 2000. 12

[26] Joshua Kerievsky. *A Learning Guide To Design Patterns*. Industrial Logic, Inc, 2000. [www] http://www.industriallogic.com/papers/learning.html (Hämtat 2006-12-09). 103

[27] Joshua Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2005. 13, 15, 21, 22, 23, 24, 27, 29, 30, 46, 56, 65, 66, 69, 75, 76, 100, 103, 106, 123

[28] Thomas J. McCabe. A complexity measure. *International Conference on Software Engineering, Proceedings of the 2nd international conference on Software engineering, San Francisco, California, United States*, page 407, 1976. 22, 43

[29] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall, 1997. 7

[30] Linda H. Rosenberg. *Applying and Interpreting Object Oriented Metrics*. Software Assurance Technology Center (SATC), NASA Goddard Space Flight Center, 1998. [www] http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply.pdf (Hämtat 2006-12-13). 38

[31] Alan Shalloway and James R. Trott. *Design Patterns Explained A New Perspective on Object-Oriented Design Second Edition*. Addison Wesley, 2004. 9, 25, 104

[32] Sogeti. *INCA – ett nationellt stöd för cancervårdens utveckling*. Sogeti Sverige AB, 2006.
[www] http://www.sogeti.se/refinca
(Hämtat 2006-11-27). 3

[33] David Trowbridge, Dave Mancini, Dave Quick, Gregor Hohpe, James Newkirk, and David Lavigne. *Enterprise Solution Patterns Using Microsoft .Net: Version 2.0 : Patterns & Practices*. Microsoft Press, 2003. 58, 60

[34] Karlstads Universitet. *DAV C19 - Software Engineering*. Karlstads Universitet, 2006. [www] http://www.cs.kau.se/cs/education/courses/davc19/ (Hämtat 2006-12-12). 104

[35] Karlstads Universitet. *DAV C20 - Projektarbete*. Karlstads Universitet, 2006. [www] http://www.cs.kau.se/cs/education/courses/davc20/ (Hämtat 2006-12-12). 104

[36] Karlstads Universitet. *DAV C20 - Projektarbete - Uppdragsspecifikation*. Karlstads Universitet, 2004-10-28.

[www] http://www.cs.kau.se/cs/education/courses/davc20/docs/as04.pdf
(Hämtat 2006-12-12). 105

[37] Karlstads Universitet. *DAV D11 - Objektorienterade designmetoder*. Karlstads Universitet, 2006.
[www] http://www.cs.kau.se/cs/education/courses/davd11/
(Hämtat 2006-12-07). 31

[38] Edmond VanDoren. *Cyclomatic Complexity*. Software Engineering Institute, 2000.
[www] http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html
(Hämtat 2006-11-23). xv, 43, 44

[39] John Vlissides. *C++ Report*. April 1998.
[www] http://www.research.ibm.com/designpatterns/pubs/ph-apr98.pdf
(Hämtat 2006-12-04). 11

[40] William C. Wake. *Refactoring Workbook*. Addison Wesley, 2004. 20

Bilaga A

Mjukvarumätningsverktyg

A.1 SourceMonitor

A.2 DevMetrics

devMetrics Code Measurements Report

Solution	Projects	Statements	Lines
- nunit	25.00	18,131.00	54,808.00

Project	Files	Types	Max Complexity	Statements	Lines
- nunit.core	83.00	101.00	60.00	2,226.00	9,950.00

Type Declaration	Type memebers	Functional Members	Avg. Complexity	Total Complexity	Max Complexity	Statements
+ NUnit.Core.AbstractFixtureBuilder	9.00	8.00	1.22	11.00	3.00	35.00
+ NUnit.Core.AbstractTestCaseBuilder	8.00	7.00	0.88	7.00	4.00	22.00
+ NUnit.Core.AddinManager	18.00	11.00	0.67	12.00	7.00	38.00
+ NUnit.Core.AddinManager.SuiteBuilderWrapper	6.00	3.00	0.67	4.00	2.00	10.00
+ NUnit.Core.Addins	11.00	10.00	0.45	5.00	1.00	25.00
+ NUnit.Core.AddinState	2.00	2.00	0.00	0.00	0.00	4.00
+ NUnit.Core.AssemblyResolver	9.00	6.00	1.89	17.00	15.00	59.00
+ NUnit.Core.AssemblyResolver.AssemblyCache	4.00	3.00	1.00	4.00	3.00	8.00
+ NUnit.Core.Builtins	8.00	6.00	0.50	4.00	1.00	15.00
- NUnit.Core.CategoryFilter	11.00	10.00	1.27	14.00	8.00	32.00

Type Member Declaration	Complexity	Statements
categories	0.00	0.00
CategoryFilter	0.00	1.00
CategoryFilter	0.00	2.00
CategoryFilter	0.00	1.00
CategoryFilter	0.00	3.00
CategoryFilter	0.00	1.00
CategoryFilter	0.00	3.00
AddCategory	0.00	2.00
Pass	8.00	13.00
Pass	5.00	4.00
CheckCategories	1.00	2.00

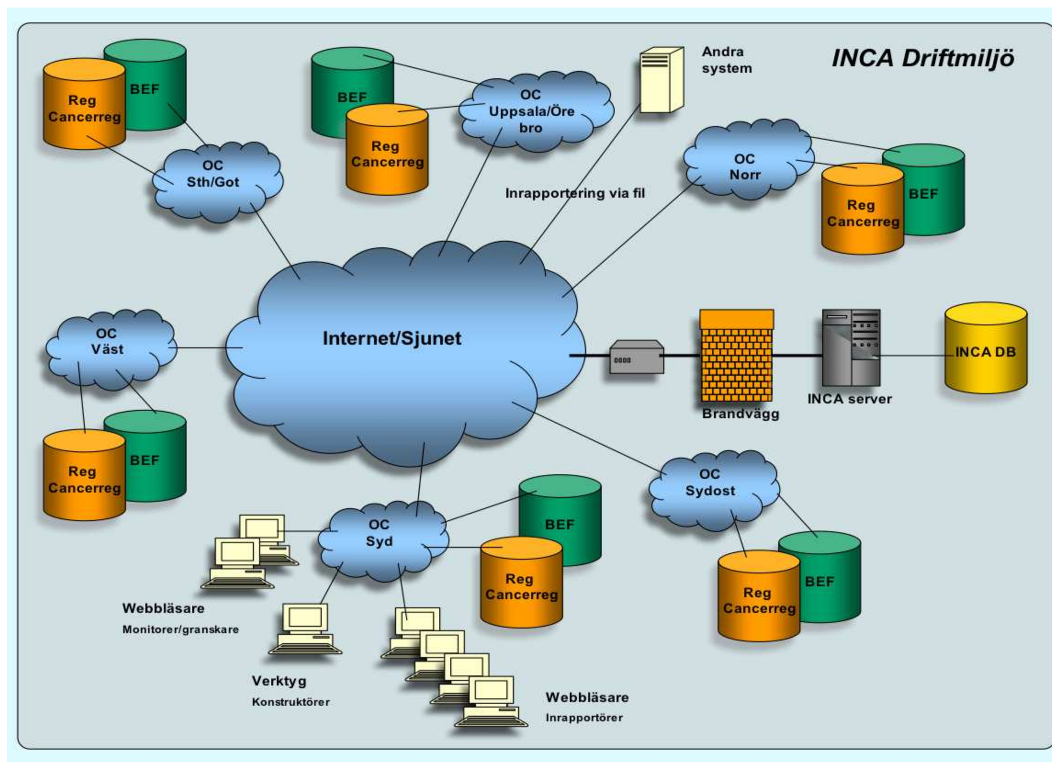
Type Declaration	Type memebers	Functional Members	Avg. Complexity	Total Complexity	Max Complexity	Statements
+ NUnit.Core.CategoryManager	6.00	5.00	0.67	4.00	2.00	19.00
+ NUnit.Core.CSUnitTestCaseBuilder	1.00	1.00	0.00	0.00	0.00	1.00
+ NUnit.Core.Builders.CSUnitTestFixture	4.00	3.00	1.00	4.00	2.00	11.00
+ NUnit.Core.Builders.CSUnitTestFixtureBuilder	3.00	3.00	0.33	1.00	1.00	9.00
+ NUnit.Core.DirectorySwapper	4.00	3.00	0.25	1.00	1.00	7.00
+ NUnit.Core.EmptyFilter	3.00	3.00	1.00	3.00	1.00	6.00

A.3 Resource Standard Metrics (RSM)

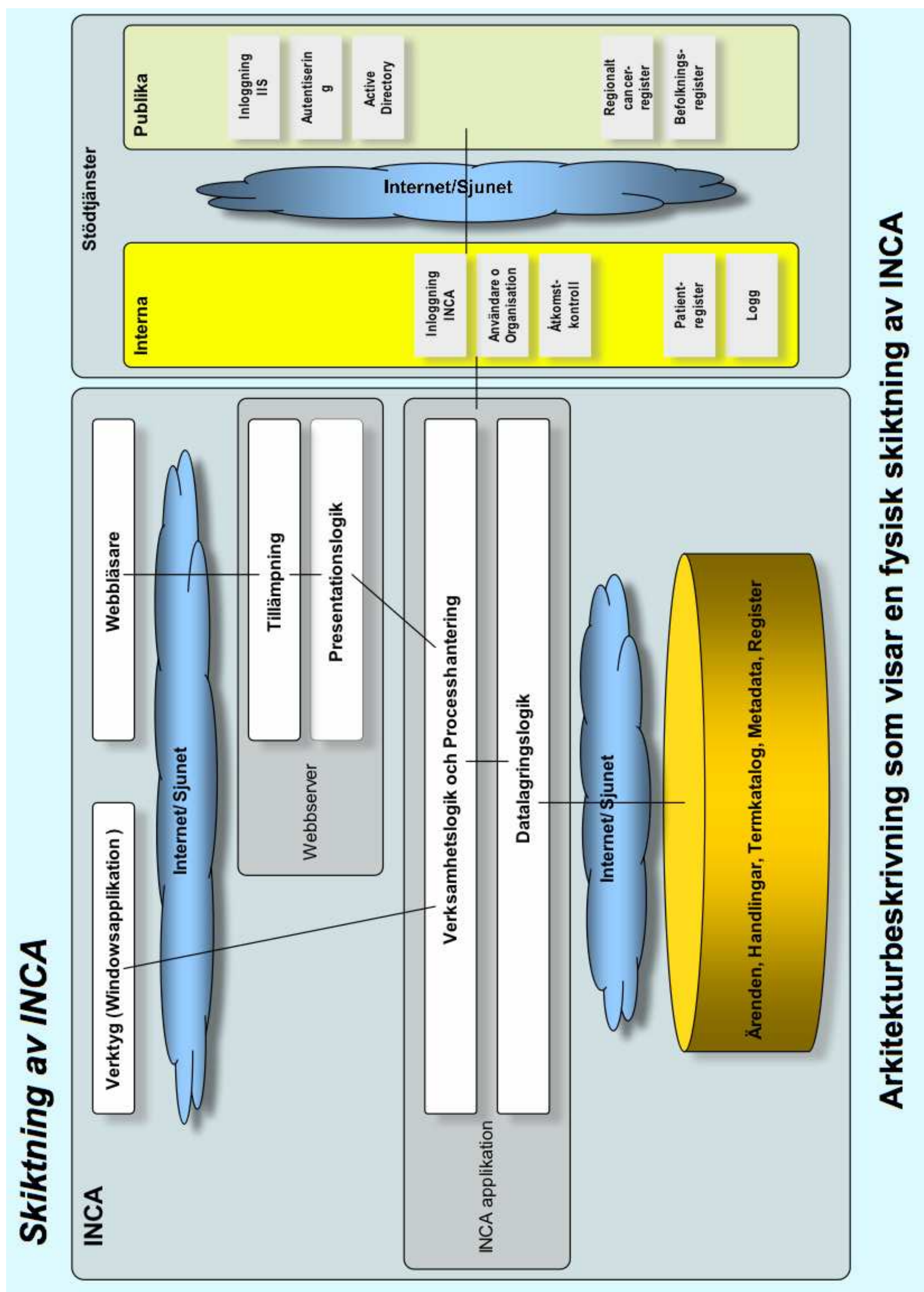
Bilaga B

INCA

B.1 Driftsmiljö



B.2 Skiktning och arkitekturbeskrivning



Ordförklaringar

.NET Microsofts plattform eller ramverk för att köra applikationer byggda med ett .NET-språk, t ex C#.

ASP.NET Active Server Pages. Microsofts plattform eller ramverk för webbapplikationer byggda med ett .NET-språk, t ex C#.

BizTalk Server Microsofts serverprogramvara för integration av data mellan affärssystem.

C#.NET Ett objektorienterat programspråk utvecklat av Microsoft, avsett att kompileras för .NET-plattformen. Uttalas See-Sharp. Språkets syntax liknar C++ och Java.

IIS Internet Information Services. Microsofts webbserverprogramvara.

SQL Server Microsofts relationsdatabashanteringssystem (RDBMS) med stöd för SQL.

Förkortningar

INCA INformationsnätverk för CAncerforskningen. En ny nationell IT-plattform för hantering av nationella register kring cancerpatienter avseende vård och forskning. INCA utvecklas av IT-konsultföretaget Sogeti i Karlstad på uppdrag av Sveriges sex OC.

OC Onkologiskt Centrum. Med onkologiskt centrum avses en funktionell enhet för utveckling och samordning av regionens resurser för diagnostik, vård och förebyggande av cancersjukdomar, baserad på de för ändamålet avsedda resurser som finns inom regionen. Sverige har sex OC, belägna i Stockholm, Göteborg, Umeå, Linköping, Lund och Uppsala. INCA-systemet ägs av Sveriges sex OC.

UML Unified Modeling Language. Ett visuellt språk för modellering av mjukvarusystem, affärsprocesser och datastrukturer.

XP Extreme Programming. En mjukvaruutvecklingsmetodik som bland annat förespråkar en enkel framväxande design understödd av kontinuerlig omkonstruktion.

Objektorienterade Mönster

Abstract Factory Källa: Gamma et.al.[24].

Adapter Källa: Gamma et.al.[24].

Bridge Källa: Gamma et.al.[24].

Builder Källa: Gamma et.al.[24].

Chain of Responsibility Källa: Gamma et.al.[24].

Collecting Parameter Källa: Kerievsky [27].

Command Källa: Gamma et.al.[24].

Composite Källa: Gamma et.al.[24].

Data Mapper Källa: Fowler [23].

Decorator Källa: Gamma et.al.[24].

Facade Källa: Gamma et.al.[24].

Factory Method Källa: Gamma et.al.[24].

Flyweight Källa: Gamma et.al.[24].

Front Controller Källa: Fowler [23].

Identity Map Källa: Fowler [23].

Interpreter Källa: Gamma et.al.[24].

Iterator Källa: Gamma et.al.[24].

Layer Supertype Källa: Fowler [23].

Mediator Källa: Gamma et.al.[24].

Memento Källa: Gamma et.al.[24].

Metadata Mapping Källa: Fowler [23].

Missing Object Ett annat namn för Null Object

MVC Källa: Fowler [23].

Null Object Källa: Fowler[22].

Observer Källa: Gamma et.al.[24].

Page Controller Källa: Fowler [23].

Plugin Källa: Fowler [23].

Prototype Källa: Gamma et.al.[24].

Proxy Källa: Gamma et.al.[24].

Query Object Källa: Fowler [23].

Separated Interface Källa: Fowler [23].

Singleton Källa: Gamma et.al.[24].

State Källa: Gamma et.al.[24].

Strategy Källa: Gamma et.al.[24].

Template Method Källa: Gamma et.al.[24].

Visitor Källa: Gamma et.al.[24].