Avdelning för datavetenskap

Helena Raunegger & Annelie Åslund

Granskning av öppen källkod från Compiere

– En studie av läsbarhet och struktur

Study of an Open Source Code from Compiere

- An investigation of structure and readability

Examensarbete 10 p
Programvaruteknik

Datum:	04-06-03
Handledare:	Eivind Nordby
Examinator:	Martin Blom
löpnummer:	C2007:02

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

Granskning av öppen källkod från Compiere

– En studie av läsbarhet och struktur

Annelie Åslund & Helena Raunegger

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen 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.

Annelie Åslund

Helena Raunegger

Godkänd, Datum

Handledare: Eivind Nordby

Examinator: Martin Blom

Tack

Vi vill rikta ett stort och varmt tack till vår handledare och uppdragsgivare Eivind Nordby, vars tillgänglighet och stöd har varit enormt. Samt även ett stort tack till företaget Redpill, utan vilkas medverkan detta exjobb aldrig hade tillkommit.

Sammanfattning

Denna C-uppsats är en undersökning av ett affärssystem som är en öppen källkodsprodukt som heter Compiere. Arbetet är sammankopplat med ett projekt mellan Redpill AB och Karlstads universitet.

Vårt mål med uppsatsen var att undersöka källkoden med fokus på läsbarhet och struktur. Metoden vi använde oss av var kontraktsprogrammering enligt SEMLA [2], vi upprättade kontrakt utifrån ett antal godtyckligt utvalda funktioner. Funktionerna delades in i olika kategorier för att vi lättare skulle kunna avgränsa potentiella funktioner. Granskningen utfördes på två olika sätt, enledsgranskning och flerledsgranskning.

Vårt resultat sammanfattas som följande: Beroendena mellan moduler är många och svåröverblickbara. Vilket gör koden strukturellt komplex och försvårar läsbarheten markant. Generellt är koden bra dokumenterad, vilket visades när vi upprättade kontrakt vilka sedermera jämfördes med den befintliga dokumentationen. Dock finns det funktioner med svårtolkad dokumentation och kommentarer, och med det menar vi att dokumentationen/kommentarerna är svåra att förstå för någon annan än implementeraren. Det finns även funktioner i källkoden som helt saknar förklarande dokumentation. Resultatet därutav är att läsbarheten försämras överlag. Läsbarheten och strukturen i källkoden från Compiere försvårades genom tvivelaktig feldateringen i koden.

Slutsatsen i vår undersökning är att källkoden från affärssystemet Compiere har låg läsbarheten och en komplex struktur.

Study of an Open Source Code from Compiere

- An investigation of structure and readability

Abstract

This bachelor's degree essay is an investigation of a business system from Compiere, which is an open source product. The investigation is integrated with a project between Redpill AB and Karlstad University.

Our goal with this essay is to investigate the source code with a focus on readability and structure. The method we used is programming by contracts according too SEMLA [2], we establish the contracts on basis of arbitrary chosen functions. The functions where separated in different categories in order to delimit presumptive functions. The review was performed in two different manners, a one-step investigation and a multi-step level investigation.

Our conclusion is following: The dependences between the modules are many and hard to survey. Which make the code structurally complex and make the readability more difficult. In general the code is well documented, which come in to view when the contract was established and afterwards compared with the existing documentation. However there are functions with documentation and comments difficult to interpret, in meaning that the documentation/comments are hard to understand for others than the implementer. Moreover there are functions with no documentation at all. The result is that it makes the general readability more difficult. The readability and the structure from the source code from Compiere increased because there were doubtful exception handlings.

The conclusion in our investigation is that both the readability and the structure in the source code from Compiere are complex.

Innehållsförteckning

1 Inledning 15

1.1 Problembakgrund 15

1.2 Avgränsningar 16

1.3 Begränsningar 16

1.4 Verktyg i uppsatsarbetet 16

1.5 Syfte med uppsatsen 17

1.6 Målgrupp 17

1.7 Disposition 17

2 Centrala begrepp 19

2.1 Öppen källkod 19

2.2 Compiere 19

2.3 Redpill 20

2.4 Compresion 20

3 Teoretisk referensram 21

3.1 Läsbarhet och struktur 21

3.2 Kontraktsprogrammering enligt SEMLA 22

3.2.1 Starka kontrakt

3.2.2 Svaga kontrakt

3.3 Felhantering 26

4 Metod 29

4.1 Metodval 29

4.1.1 Kvantitativ metod

4.1.2 Urvalet

4.1.3 Avgränsning i urvalet

4.2 Reliabilitet 35

5 Resultat 37

5.1 Läsbarhet 37

5.2 Struktur 46

6 Slutsatser **49**

7 Problem och erfarenheter **51**

Referenser **53**

A Granskade Funktioner **55**

A.1 Exempelfunktion 1 56

A.2 Exempelfunktion 2 59

A.3 Exempelfunktion 3 61

Figurförteckning

Figur 1-1: Syntax för ett grepanrop. 16
Figur 3-1: Samverkan mellan klient- och leverantörsfunktion med kontrakt 23
Figur 3-2: Samspel mellan klient- och leverantörsfunktion 23
Figur 3-3: Abstraktionen mellan kontrakt och implementation 24
Figur 3-4: Process vid starkt kontrakt 25
Figur 3-5: För- och eftervillkor för starkt kontrakt 25
Figur 3-6: Process vid svagt kontrakt 26
Figur 3-7: För- och eftervillkor för svagt kontrakt 26
Figur 4-1: Enleds- och flerledsgranskning 31
Figur 4-2: Funktionen getSQL(MTabVO mTabVO) 33
Figur 4-3: Funktionen createFields(`MTabVO mTabVO`) 34
Figur 5-1: Klientfunktion activate() 39
Figur 5-2: Leverantörsfunktion getAD_Tree_ID(String keyColumnName) 40
Figur 5-3: Klientfunktion getWebUser(Properties ctx) 42
Figur 5-4: Leverantörsfunktion getCookieWebUser(HttpServletRequest request) 43
Figur 5-5: Klientfunktion connectRemote() 44
Figur 5-6: Leverantörsfunktion getInitialContext(Hashtable env) 45
Figur 5-7: Exempel på undantagshantering 46

1 Inledning

Detta inledande kapitel ger en introduktion och förutsättningar som bakgrund för vårt arbete. Kapitlet inleds med en problembakgrund till uppsatsens ämne, sedan följer en redogörelse för undersökningens syfte och dess målgrupp. Avslutningsvis redovisas en övergripande disposition av uppsatsen.

1.1 Problembakgrund

Denna studie är ett examensarbete på C-nivå i datavetenskap vid Karlstads universitet. Examensarbetet är en liten del inom ett större forskningsprojekt, Compresion, vars avsikt är att utgöra en avstamp för vidare utveckling och utnyttjande av öppna programvaror.

Allt eftersom företag blir mer datoriserade används IT inte bara som ett hjälpmedel utan även ofta som en central del i ett företags produktion. Användningsområden kan vara allt från ett samlat kundregister till fakturering. De affärssystem som lanseras på marknaden är ofta inte direkt anpassade för ett specifikt företag. Vidare finns det brister i tillgängligheten vad gäller service och support, vilket gör att de licenser som företagen betalat för att utnyttja dessa system ofta blir dyra både när det gäller i tid och i pengar. Företaget Redpill, se avsnitt 2.3, vill anpassa det öppna affärssystemet Compiere, se avsnitt 2.2, till svensk standard och göra den tillgänglig för både offentliga och privata företag.

Ett dataprograms livscykel initieras med en idé och en specifikation av ett dataprogram. Programmet designas sedan efter de krav på funktionalitet som ställs. I designfasen är det viktigt att strukturera koden i moduler för att programmet i framtiden ska vara lätt att modifiera och anpassa efter sitt ändamål. Det är också av betydelse att dokumentera koden väl. Vilket görs med en god beskrivning av funktionernas tillämpning och dess indata och utdata [12].

Kontraktsprogrammering är en metod som används som ett hjälpmedel för att strukturera ett dataprogram. En metod, benämnd Semla [2], är utvecklad av Software Engineering Research Group, SERG [1], en forskningsgrupp på institutionen för datavetenskap vid Karlstads universitet. SERG´s mål var att ta fram en programutvecklingsmetod som fokuserade på den

semantiska beskrivningen, i avsikt att höja programvarukvalitén i komplexa system. Kontrakten garanterar funktionernas korrekthet, vilket underlättar programmeringen avsevärt om det är många utvecklare. Kontrakten beskriver varje funktion exakt, vad som krävs för att anropa den och vad den returnerar. Läsbarhet och dokumentation av koden tillsammans med självständiga moduler ger en god struktur vid programmering och underlättar underhållning och anpassning av system [12].

1.2 Avgränsningar

Det är många komponenter som bidrar till att ett dataprogram utvecklas. Ett programs livscykel delas in i många delar som specifikation, design, riskanalys, verifikation, kodning, testning, anpassningar, produktion och underhåll. Uppsatsen kommer endast att beröra områden kring design och underhåll.

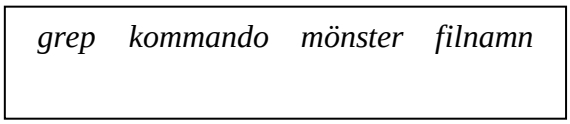
Begreppen läsbarhet och struktur på källkod kan definieras på en mängd sätt. Vi har därmed valt att avgränsa oss till det som vi tolkat som det mest generella i definitionerna, se vidare kapitel 3.1.

1.3 Begränsningar

Tidsramen för c-uppsats om 10 poäng är satt till 20 veckor på halvfart. Författarnas förkunskaper inom området datavetenskap har innefattats av A-, B- samt C-kurser.

1.4 Verktyg i uppsatsarbetet

grep [15] är ett verktyg som används för att söka igenom filer och strömmar efter rader som matchar ett angivet mönster. Det används genom att vid prompten skriva som Figur 1-1 visar.



Figur 1-1: Syntax för ett grepanrop.

grep följs av ett kommando, vilket ger en möjlighet att styra utseendet på utskriften av sökningsresultatet. Möjliga kommandon är t.ex. –b, vilket resulterar i att utskriften sker med en radbrytning mellan alla påträffade matchningar. mönster är det som man önskar söka efter,

t.ex. ett funktionsanrop. Slutligen filnamn vilket anger vart verktyget skall leta, eller * för att leta i alla filer.

1.5 Syfte med uppsatsen

Vi skall granska öppen källkod från Compiere i syfte att ta reda på hur dess läsbarhet och struktur ser ut.

1.6 Målgrupp

Våra resultat kan vara intressant för följande grupper:

- ► Studenter inom Datavetenskap.
- ► Projektet Compresion.
- ► Företaget Redpill.
- ► Övriga studenter och människor som är intresserade av öppen källkod.

1.7 Disposition

C- uppsatsen indelas i sex kapitel:

► Kapitel 2 (Teoretisk referensram)
Kapitlet påbörjas med att beskriva de central begrepp som initierar uppsatsens ämne. Teorin bygger på kontraktsprogrammering enligt SEMLA. Kapitlet fortlöper genom en redogörelse av svaga respektive starka kontrakt. Den avlutande delen består av en förklaring till vad felhantering är och hur det tas om hand i ett system, samt hur det *grep*-verktyg vi använder oss av i undersökningen fungerar.

► Kapitel 3 (Metod)
Metodkapitlet beskriver det tillvägagångssätt som undersökningen har genomförts på. Vidare behandlar metodkapitlet valet till undersökningsmetod och urvalet. Kapitlet avslutas genom en diskussion av reliabilitet i uppsatsen.

► Kapitel 4 (Resultat)

Kapitlet har delats in i två delar. Den första delen behandlar läsbarhet och det andra delen struktur.

► Kapitel 5 (Slutsats)

Kapitlet innehåller en redogörelse av den slutsats som vi dragit från vår granskning av källkoden från Compiere.

► Kapitel 6 (Problem och erfarenheter)

De problem som vi har stött på under vår tid som undersökningen har pågått, samt även de erfarenheter vi erhållit, redovisas.

2 Centrala begrepp

Detta kapitel presenterar de centrala begrepp som vår undersökning har grundats på. Inledningsvis beskriver vi vad öppen källkod är, vilken sorts källkodsprodukt vi granskar, vilket företag som ska modifiera källkodsprodukten och vilket forskningsprojekt som vår uppsats är knuten till.

2.1 Öppen källkod

Öppen källkod [6] är samlingsnamnet på programvara vars källkod finns tillgänglig på Internet. Källkoden kan användas, förändras, förbättras, kopieras och distribueras av den som så önskar. Felaktigt tänker många att öppen källkod, samt dess programvara därmed skall vara gratis, vilket inte alltid är fallet. Normalt är dessutom att programvaran är licensierad, vilket kostar pengar, och där dessutom friheter och rättigheter regleras av olika villkor. Rätten att göra ändringar i koden kan därmed ha begränsats av skaparen/skaparna så att endast ett fåtal personer, som genom kvalificering, har erhållit rättigheter att göra ändringar i koden.

Produktionen av öppen källkod och programvara har under de senaste åren ökat explositionsartat. Resulterat av detta är att marknaden för köpt programvara har fått ge plats för den öppna källkoden och dess programvara.

2.2 Compiere

Compiere [7] är ett öppet källkodsprojekt, grundat av Jorg Janke. Projektarbetet etablerades år 2001 och räknas idag som Internets största affärsapplikation, med mer än 500 000 nedladdningar. Affärsapplikationen tillhandahåller ett färdigt basprogram för små och medelstora företag. Systemet består av 1191 filer som innehåller bland annat hantering av ekonomi, kund- och leverantörsregister samt lagerhållning.

Affärssystemet Compiere finns idag, förutom på engelska, även översatt till tyska, franska, portugisiska, spanska och kinesiska. Utveckling och förbättringar sker kontinuerligt av cirka

50 programmerare runt om i världen, vilka har kvalificerat sig och tilldelats denna rättighet av grundaren. Källkoden är skriven i programspråket Java och kan användas på operativsystem som Linux, Windows, MacOs och Sun Solaris [7].

2.3 Redpill

IT-företaget Redpill [5] arbetar med att integrera och anpassa affärssystemet Compiere till den svenska marknaden. Företagets framtida verksamhet fokuseras på en individuell anpassning av affärssystemet Compiere till enskilda företags behov. Deras affärsidé bygger på att kunden själv kan välja leverantör av support, uppgradering, underhåll samt utbildning av systemet.

2.4 Compresion

Projektet Compresion [16] är finansierat av EU, ägs av Karlstads universitet och leds av företaget Redpill. Inom Compresion finns även delprojektet ROSS, Research on Open Source Software. ROSS skall utvärdera Compiere´s öppna källkod med fokus på kodens inre korrekthet. Projektet Compresion ingår i ett program "Innovativa åtgärder i Norra Mellansverige" som drivs av länsstyrelserna i Värmlands, Dalarnas och Gävleborgs län.

3 Teoretisk referensram

Detta kapitel presenterar den teoretiska referensram som vår undersökning har grundats på. Inledningsvis redovisar vi hur vi har avgränsat läsbarhet och struktur. Vidare kommer vi att ge läsaren teorin bakom kontraktsprogrammering. Kontraktsprogrammering är en abstraktion av programkoden som underlättar läsbarheten och struktur vid granskning av programkod. Vidare redogör vi för skillnaden mellan starka och svaga kontrakt. Kapitlet avslutas med en förklaring gällande felhantering i programkod, då detta påverkar både läsbarhet och struktur.

3.1 Läsbarhet och struktur

Det finns många individuella stilar när det gäller programmering. När ett nytt dataprogram ska utvecklas har ofta utvecklingsföretaget normer och regler för vilken programmeringsstil som skall användas. Det kan gälla allt från hur koden ska dokumenteras och indenteras, till vilket programmeringsspråk som ska användas i implementationen. Genom att standardisera programmeringsstilen i ett system underlättar det för andra än programmeraren/programmerarna att läsa och förstå programmet, koden blir därmed tillgänglig för fler.

En bra strukturerad kod kännetecknas av att koden är uppdelad i små moduler. Modulerna ska vara så generella som möjligt, allt för att de skall kunna anropas av andra moduler och därmed kunna återanvändas. Vidare ska modulerna vara strukturerade med så lösa kopplingar som möjligt mellan modulerna dvs. att modulerna är så oberoende av varandra som möjligt, vilket skapar en struktur som underlättar framtida utveckling och modifiering av systemet [12].

För att läsbarbeten ska vara hög i ett program krävs det att en rad aspekter beaktas [12];

► En bra dokumentation, som beskriver funktionens krav på indata, vad som utförs, samt dess utdata. Dokumentationen placerad över funktionsmodulen vilken den avser.

► Bra kommentation av kod, dvs. korta, tydliga förklaringar inuti funktionsmodulen.

► Funktioner och variabler ska vara namngivna med förklarande namn som speglar dess funktion i källkoden.

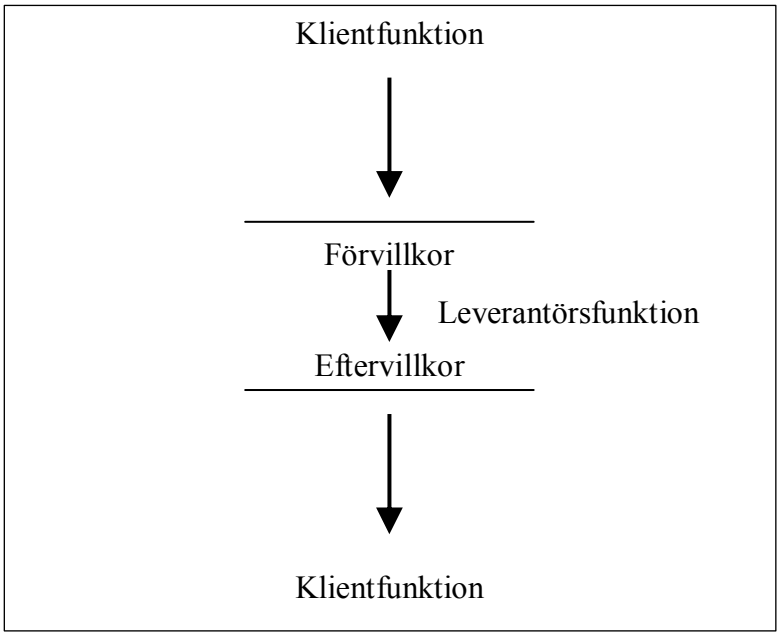
► En välindenterad och luftig kod.

► En tydlig och funktionell felhantering.

3.2 Kontraktsprogrammering enligt SEMLA

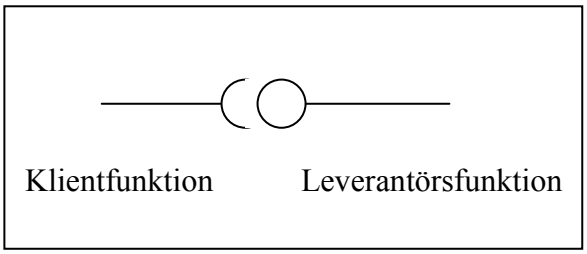
Idén bakom programmering med kontrakt fanns redan i slutet av 1960-talet hos Hoare, men det var Bertrand Meyer som gav kontaktprogrammeringen dess namn. Under åren 1999 till 2001 pågick ett arbete, av forskningsgruppen SERG [1], kring detta på Karlstad universitet. Arbetet resulterade i ett dokument som beskriver en designmetod för semantisk beskrivning kallat SEMLA [2].

Den semantiska beskrivningen, kontraktsprogrammeringen innebär att ett formellt avtal, kontakt, upprättas av implementatören av leverantörsfunktionen, vilket tillhandahåller en abstraktion av koden. Avtalet specificerar varje klients och leverantörs skyldigheter gentemot varandra. Figur 3-1 visar på den samverkan som sker genom kontraktet, där ett förvillkor specificerar vad som krävs för att ett anrop av funktionen skall ske på ett korrekt sätt. Varefter eftervillkoret då garanterar att om förvillkoret är uppfyllt, genererar funktionen utlovat resultat tillbaka till klientfunktionen. Exempel på upprättade kontrakt på källkoden från Compiere, se exempelfunktioner A.1, A.2, A.3.



Figur 3-1: Samverkan mellan klient- och leverantörsfunktion med kontrakt

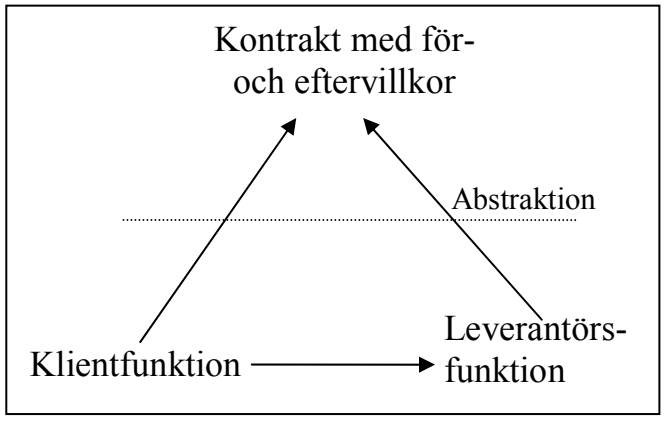
Figur 3-2 exemplifierar ytterligare interaktionen mellan klient- och leverantörsfunktionen. Figuren visar på klientfunktionens exakta utformning efter leverantörsfunktionens form, dvs. att klienten är exakt anpassad till leverantörsfunktionen vid anrop och mottagning av returvärde/värden.



Figur 3-2: Samspel mellan klient- och leverantörsfunktion

Kontraktsprogrammering kan ses som en abstraktion som används för att på ett enkelt sätt upprätthålla konsistensen, se Figur 3-3. Genom att abstrahera en funktion ger man användaren endast de viktiga bitarna på funktionens utförande. Övriga delar lämnas åt implementeraren av funktionen, dvs. det under abstraktionsstrecken i figuren. Det leder till att användaren endast behöver lita till kontraktet som finns mellan användare och implementatören, vilka återfinns över abstraktionsstrecket i figuren. Kontraktet uppfylls genom för- och eftervillkoren som finns för funktionen. Implementationen i respektive funktion är även den avdelad av en

viss abstraktion, klientfunktionen behöver inte se leverantörsfunktionens implementation för att använda den, utan uppfyller endast sin del av kontraktet.



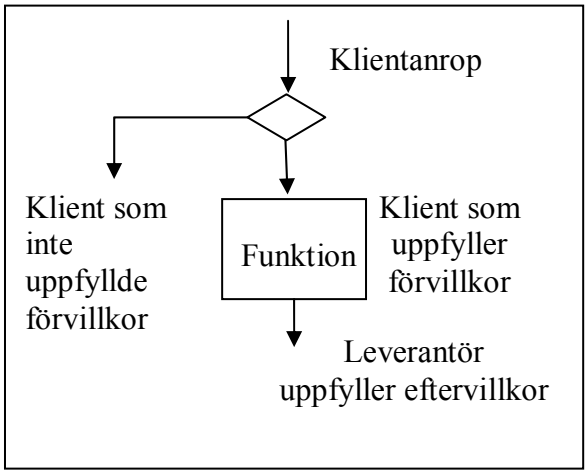
Figur 3-3: Abstraktionen mellan kontrakt och implementation

Figur 3-3 visar därmed hur en funktion har abstraheras från detaljerna som implementationen. Istället har endast de viktiga bitarna såsom funktionens funktionalitet redovisas genom sitt för- och eftervillkor. Abstraktionen hjälper också till att strukturera programmet och öka läsbarheten vid exempelvis modularisering av programmet.

Det finns två olika typer av kontrakt, starka respektive svaga kontrakt. Det som skiljer dem åt är styrkan på förvillkoret.

3.2.1 Starka kontrakt

I programmering med starka kontrakt används ett starkt förvillkor, det vill säga ett som är ställer krav på klienten. För starka kontrakt är förvillkoret sådant att det garanterar ett lyckat genomförande enligt dess egentliga intention. Förvillkoret är det som krävs för att garantera att funktionen går att genomföra korrekt. Funktionen tar inte hand om eventuella felaktiga anrop, utan förväntar sig att klienten uppfyller förvillkoret och garanterar då eftervillkoret. Annars kan inga garantier lämnas för funktionens beteende, se Figur 3-4. Figuren visar anrop från klienten, och om förvillkoret uppfylls exekverar funktionen korrekt och eftervillkoret uppfylls. Men om klienten anropar funktionen på ett felaktigt sätt så finns det inga garantier när det gäller exekvering av funktionen. Det kan leda till att programmet ger felaktiga returvärden eller havererar. Vanligtvis används inte starka kontrakt när funktionen är beroende av mänsklig inmatning, eftersomdet kan vara svårt för den enskilde individen att alltid kontrollera förvillkoret innan han/hon anropar funktionen.



Figur 3-4: Process vid starkt kontrakt

För att ytterligare exemplifiera ett starkt kontrakt används metoden top() på en stack. Figur 3-5 visar på dess för- och eftervillkor. Metoden top() har som förvillkor att stacken inte är tom innan anrop sker. Därmed förbinder sig klienten att kontrollera att stacken inte är tom innan anrop, och leverantörsfunktionen å sin sida förbinder sig att returnera värdet på elementet till klienten.

pre: !isEmpty()

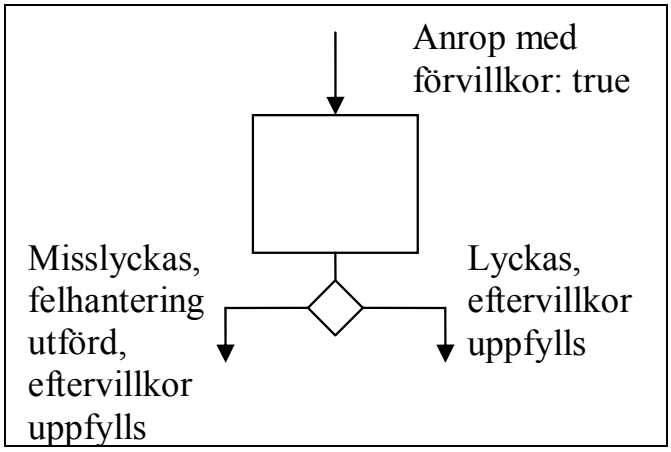
post: Returnerat värdet på översta elementet i stacken.

Figur 3-5: För- och eftervillkor för starkt kontrakt

3.2.2 Svaga kontrakt

Vid svaga kontrakt är förvillkoret för svagt (t.ex. true, dvs. inget alls) för att garantera ett lyckat genomförande av funktionskoden enligt den egentliga intentionen. Men det innebär inte att ett sådant anrop inte är korrekt, utan bara att det inte är klienten som har fått ansvar för ett lyckat genomförande, utan det ansvaret åvilar funktionen själv. Funktionen kan alltså misslyckas i förhållande med sin intention, så eftervillkoret måste beskriva även resultatet vid misslyckade genomföranden. Det är alltså inte frågan om korrekt anrop eller inte, utan om att det är olika krav som ställs på vad som anses korrekt. Det menas att ingen kontroll utförs av klienten för att ta reda på huruvida funktionsanropet är genomförbart eller inte. Figur 3-6 visar på denna process, där funktionen ifråga själv tar hand om testningen av anropen, vilket gör

implementationen mer omfattande än vid starka kontrakt. Leverantören måste därmed även förutse alla tänkbara scenarion som kan inträffa under exekvering och ta hand om dessa. Eftervillkoret blir även det mer omfångsrik, eftersom alla scenarion redovisas i eftervillkoret. Se ytterligare exempel på funktion ur källkoden från Compiere med av oss upprättat svagt kontrakt, se A.3.



Figur 3-6: Process vid svagt kontrakt

Likvärdigt som med starkt kontrakt exemplifierar vi även svagt kontrakt med metoden top() på en stack. Som Figur 3-7 visar på kan metoden anropas utan att klienten behöver veta huruvida det existerar element på stacken eller inte. Alla anrop testas sedan inne i metoden, vilken sedan returnerar det som eftervillkoret utlovat, antingen värdet på översta elementet, om det finns, eller ett felmeddelande, om det inte finns något element på stacken.

pre: True

post: Returnerat värdet på översta elementet i stacken om stacken inte var tom. Om stacken var tom har felmeddelande returnerats.

Figur 3-7: För- och eftervillkor för svagt kontrakt

3.3 Felhantering

Det finns två olika typer av fel som kan inträffa i ett system, externa och interna. Externa fel kan vara fel som inte programmet kan gardera sig mot, som t.ex. felaktig inmatning eller

kommunikationsfel. Interna fel, är de som till viss del är förutsägbara, t.ex. att man inte kan ta bort element ur en redan tom lista dvs. fel som finns i själva programmet.

Felhanteringsmekanismer som kan användas är bland annat kontrakt, implementerad testning eller *exceptions*, en s.k. undantagshantering. Med undantagshantering garderar systemet sig för överraskande händelser i systemet. Om det blir ett fel i exekveringen av leverantörsfunktionen kastar funktionen ett undantag (throw exeption e), klientfunktion fångar då upp det kastade undantaget (catch exeption e) och eventuellt registreras felet i en loggfil.

4 Metod

Detta kapitel presenterar hur undersökningen är genomförd. Till att börja med redovisas de val som gjorts gällande den metod som vi har använt oss av i vår undersökning. Vidare följer en beskrivning om hur urvalet har skett och vilka avgränsningar som gjorts. Sedan följer en redovisning av vår kategorisering av de olika funktionerna, samt innehållet i de olika kategorierna. Kapitlet slutförs genom en diskussion om metodens reliabilitet.

4.1 Metodval

Metod är ett hjälpmedel för att kunna beskriva data som samlas in för undersökning. För att kunna göra en vetenskaplig undersökning krävs det att forskaren har ett problem eller problemområde som hon/han vill undersöka. Ett problem är något som forskaren genom sin undersökning vill lösa eller belysa, samt öka sin kunskap om. Med ett bra metodval och en kritisk granskning av metoden går det att bedöma hur mycket av resultaten som beror på metoden och om resultaten ger en korrekt bild av verkligheten [10]. Det finns två kompletterande vetenskapliga metoder att välja mellan, nämligen kvantitativ eller kvalitativ metod. Det som är viktigt att tänka på när metoden väljs är hur problemet är formulerat och hur det bäst kan besvaras [11].

4.1.1 Kvantitativ metod

Den metod som ansågs passa bäst in för vår undersökning var kvantitativ metod. Den kännetecknas av att representera ett stort urval av undersökningsenheter [8]. Det är ur validitetssynpunkt lätt att extrapolera till hela populationen samt lätt att identifiera prioriteringar. Nackdelarna med denna typ av undersökningar är partiellt bortfall (variabelbortfall) och bearbetningsfel. Partiellt bortfall innebär att vissa frågor inte kan besvaras genom att det urval som ska representera populationen inte är tillförlitligt. Bearbetningsfel är när undersökningsledaren sammanställer svaren och skriver in fel när svaren ska redovisas [9].

Inledningsvis fanns funderingar på hur vi skulle kunna granska koden. För att erhålla en generell överblick över hela systemet, samt erhålla ett första intryck över kodens läsbarhet

och struktur, studerades de 1191 filerna. Vid granskningen fann vi att vi utifrån vår synvinkel på struktur och läsbarhet, kunde dela upp källkoden i fyra olika kategorier.

► **Triviala funktioner - utan dokumentation.**

I denna kategori placerades funktioner som innehöll endast ett fåtal rader kod. Ingen hänsyn togs till om huruvida funktionerna var dokumenterade respektive kommenterade eller inte, då vi med våra kunskaper enkelt kunde avläsa koden och se vad funktionen utförde.

► **Icke triviala (1) – koden var väl dokumenterad och kommenterad**

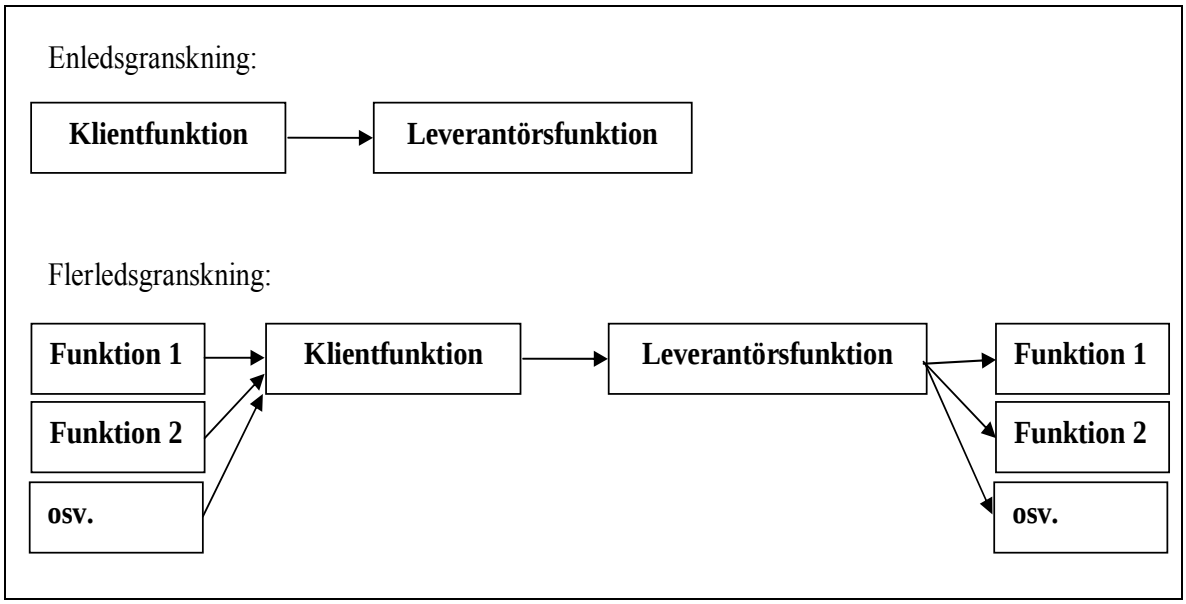
Här placerades funktioner vilka innehöll mer än 3 rader kod. Koden var inte speciellt svår att förstå, men det krävdes lite mer arbete än den första kategorin. Funktionerna var väl dokumenterade och kommenterade, vilket innebar att läsbarheten var hög.

► **Icke triviala (2) – funktionen hade lite dokumentation, och inga kommentarer i koden**

Dessa funktioner hade lite dokumentation, och saknade kommentarer inuti funktionen. Följden blev att granskningen försvårades betydligt vad gäller läsbarheten, då vi var tvungna att utläsa funktionens funktionalitet utav dess kod och kommentarer.

► **Icke triviala (3) – funktionen saknade både dokumentation och kommentarer**

I koden från Compiere fanns det funktioner som varken var dokumenterade eller kommenterade inuti koden. Funktionerna innehöll därtill ett flertal funktionsanrop, vilket försvårade läsbarheten. Även strukturellt var de komplicerade, då det stora systemet gjorde det svårt att hitta leverantörsfunktionerna. För att kunna förstå funktionens uppgift i programmet var vi tvungna att följa funktionsanrop både från klienten till klientfunktionen, dvs. de anrop som skedde till den granskade klientfunktionen, samt leverantörsfunktionens anrop i sin tur till ytterligare leverantör.



Figur 4-1: Enleds- och flerledsgranskning

Utifrån dessa kategorier beslöt vi oss för att klassificera de utvalda funktionerna. Kategorierna kom att utgöras av svårighetsnivån och därmed läsbarheten på de olika funktionerna som vi skulle granska.

Granskningen initierades genom att utläsa den utvalda funktionens uppgift i källkoden. Genom att följa den granskade funktionens funktionsanrop, kunde vi se om dessa anrop ledde oss vidare till nästa funktion. Efter att ha granskat klientfunktionen skapades kontrakt mellan klient- och leverantörsfunktioner. Det gjordes så att klientfunktionens anrop jämfördes med leverantörsfunktionens förvillkor för att se om det uppfylldes. Stämde det visade det på att koden var konsistent och därmed uppfyllde de kriterier vi satt avseende läsbarhet och struktur, se avsnitt 3.1. Uppfylldes däremot inte förvillkoret, medförde detta att läsbarheten och strukturen försämrades avsevärt. Jämförelser gjordes också på vad klientfunktionen hade för förväntningar på leverantörsfunktionen och vad leverantörsfunktionen garanterade för eftervillkor.

Vissa av de granskade funktionerna innehöll flera funktionsanrop. Vi kom då till ett vägskäl, där vi insåg att tidsramen för granskningen och därmed arbetet att gå igenom de utvalda funktionerna inte skulle kunde hållas. Därför beslöt vi oss att dela vårt resultat av granskningen i två olika granskningar, nämligen en enledsgranskning och en flerledsgranskning. Vi har försökt att förtydliga detta resonemang i Figur 4-1.

Till enledsgranskningen valdes 20 funktioner som tillhörde kategori icke triviala (1) och (2), se kapitel 4.1.1. I dessa funktioner följde vi en av klientfunktionens funktionsanrop till nästa funktion. Fokus låg på hanteringen av returvärdet genom att se till att returvärdet togs om hand i koden och att funktionsanropet utfördes på ett korrekt sätt.

Flerledsgranskningen omfattade klient- och leverantörsfunktioner, men också andra funktioner som anropade klientfunktionen och som leverantörsfunktionen i sin tur anropade. Till flerledsgranskningen valdes fem olika funktioner från kategori icke triviala (3) ut. Precis som i enledsgranskningen undersöktes även hanteringen av returvärden och undantag, vilket gjordes för att se påverkan i vidare led.

4.1.2 Urvalet

Källkoden från Compiere omfattas av 1191 javafiler. För att kunna få ett resultat inom tidsrymden om 20 veckor halvfart beslöt vi oss för att göra ett urval ur målpopulationen. Urvalet genomfördes med en slumpfunktion som tillhandahölls av vår handledare.

Det slumpmässiga urvalet som utfördes på filerna från Compiere visade sig inte vara genomförbar. Eftersom de flesta funktioner som valdes ut av slumpfunktionen inte kunde användas i vår undersökning, då de inte anropades av någon annan funktion. Dessa funktioner som uppenbarligen inte användes på något sätt förbryllade oss i arbetet, vilket vi redovisar i kapitel 7. Arbetet övergick istället till att manuellt granska alla filerna och plocka ut filer som vi skulle använda i vår undersökning. Till vår hjälp använde vi *grep*-verktyget, se kapitel 1.4. Genom att använda *grep*-verktyget kunde vi spåra klientfunktionens leverantörsfunktion. De funktioner som anropades av andra funktioner, kopierades, numrerades och skrevs ut för att vidare granskas. Totalt plockade vi ut 25 funktioner som uppfyllde våra kriterier.

Som vi redovisade tidigare i avsnittet fungerade inte slumpningen, därför beslöt vi oss för att ställa ett baskrav på de funktioner som skulle granskas. Baskravet var att funktionerna anropades av någon funktion från en annan klass. Vi avsåg också att endast granska funktioner ur kategori icke triviala (1), (2) och (3). Till en början koncentrerade vi oss på publika metoder. Detta gav tyvärr inget då endast några publika konstruktörer anropades, inga andra funktioner. Även här blev vi förbryllade, och vi frågade oss om systemet vi tagit emot kanske inte var komplett, se vidare kapitel 7. Inriktningen fick därmed ändras så att även

funktioner som definierats protected togs med. Vi fick även övergå till att acceptera get- och setfunktioner som tillhörde kategori icke triviala (1), (2) och (3).

Nedan visas en av de utvalda funktionerna, Figur 4-2. Den är granskad och utifrån koden har vi upprättat dess kontrakt. Funktionen som anropar, Figur 4-3, redovisas även denna med av oss upprättat kontrakt. Funktionen tillhör klassen MFieldVO och är en protectedfunktion som anropas från klassen MTabVO. Detta exempel illustrerar hur kontraktet ökar läsbarheten genom att användaren vet att funktionen kan anropas och att eventuella felanrop tas om hand i funktionen. Kontraktet eftervillkor visar även vad funktionen levererar. Strukturellt underlättar kontraktet vid modularisering eftersom abstraktionen gör att utvecklaren inte behöver gå in i koden för att veta vilken funktionalitet den har. Den befintliga dokumentationen underlättar läsbarheten men ger inte hela sanningen om funktionen dvs. förvillkoret.

```
package org.compiere.model;

/**
 *  Field Model Value Object
 *
 *  @author Jorg Janke
 *  @version  $Id: MFieldVO.java,v 1.5 2003/11/02 07:49:56 jjanke Exp $
 */
public class MFieldVO implements Serializable

Befintlig dokumentation:
            *  Return the SQL statement used for the MFieldVO.create
            *  @param mTabVO tab value object
            *  @return SQL with or w/o translation

Upprättat kontrakt:
pre:       mTabvo != null
post:      returnerar SQL med korrekt språk (Default: basspråk)

protected static String getSQL (MTabVO mTabVO)
{
//         IsActive is part of View
String sql = "SELECT * FROM AD_Field_v WHERE AD_Tab_ID=?" + " ORDER BY
SeqNo";
if (!Env.isBaseLanguage(mTabVO.ctx, "AD_Tab"))
           sql = "SELECT * FROM AD_Field_vt WHERE AD_Tab_ID=?"
           + " AND AD_Language='" + Env.getAD_Language(mTabVO.ctx) + "'"
           + " ORDER BY SeqNo";
 return sql;
}   //  getSQL
```

Figur 4-2: Funktionen getSQL(MTabVO mTabVO)

Funktionen som anropar ovanstående funktion tillhör klassen MTabVO och är en privat funktion.

```
package org.compiere.model;

/**
 *  Model Tab Value Object
 *
 *  @author Jorg Janke
 *  @version  $Id: MTabVO.java,v 1.9 2003/11/02 07:49:56 jjanke Exp $
 */
public class MTabVO implements Serializable

Befintlig documentation:
            *  Create Tab Fields
            *  @param mTabVO tab value object
            *  @return true if fields were created

Upprättat kontrakt:
pre:       mTabVO.Fields.size() != 0
post:      returnerar true om fält skapats, annars false

private static boolean createFields (MTabVO mTabVO)
{
           mTabVO.Fields = new ArrayList();
ANROP:     String sql = MFieldVO.getSQL(mTabVO);
           try
           {
                     PreparedStatement pstmt = DB.prepareStatement(sql);
                     pstmt.setInt(1, mTabVO.AD_Tab_ID);
                     ResultSet rs = pstmt.executeQuery();
                     while (rs.next())
                     {
                               MFieldVO voF = MFieldVO.create (mTabVO.ctx,
                               mTabVO.WindowNo, mTabVO.TabNo,
                               mTabVO.AD_Window_ID, mTabVO.IsReadOnly, rs);
                               if (voF != null)
                     mTabVO.Fields.add(voF);
                     }
                     rs.close();
                     pstmt.close();
           }
           catch (Exception e)
           {
                     Log.error("MTabVO.createFields", e);
                     return false;
           }
           return mTabVO.Fields.size() != 0;
}   //  createFields
```

Figur 4-3: Funktionen createFields(MTabVO mTabVO)

4.1.3 Avgränsning i urvalet

Funktioner som vi valt att inte granska närmare tillhör kategori triviala funktioner. De bedömdes som alltför triviala för att påverka läsbarhet och struktur. Vidare uteslöts även de privata funktionerna då dessa saknar anrop från funktioner inom andra klasser.

4.2 Reliabilitet

Reliabilitet är att kunna visa att insamlade data i en undersökning har en trovärdighet. Med trovärdighet avses att resultatet av insamlingen kan återfås vid till exempel upprepad insamling. Reservationer måste därför göras då vi i uppsatsen inte kunnat använda den vetenskapliga urvalsmetoden, slumpning.

5 Resultat

Resultatet som redovisas i detta kapitel är indelat i två avsnitt. Det första avsnittet innehåller inledningsvis en kort resumé på vad läsbarhet innebär, se kapitel 3.1, samt det resultat vi funnit i vår undersökning med utgångspunkt från läsbarhet. Andra kapitlet innehåller även det en resumé från kapitel 3.1 på vad som avses med struktur i detta sammanhang. Samt det resultat vi fann i vår undersökning med avseende på källkodens struktur.

5.1 Läsbarhet

Utifrån den teoristudie vi genomfört har vi definierat de mest generella aspekterna på läsbarhet i kod. För att läsbarbeten ska vara hög i ett program krävs det att en rad aspekter uppfylls, så som;

► En bra dokumentation, som beskriver funktionens krav på indata, vad som utförs, samt dess utdata. Dokumentationen placerad över funktionsmodulen vilken den avser.

► Bra kommentation av kod, dvs. korta, tydliga förklaringar inuti funktionsmodulen.

► Funktioner och variabler ska vara namngivna med förklarande namn som speglar dess funktion i källkoden.

► En välindenterad och luftig kod.

► En tydlig och funktionell felhantering.

Utifrån dessa aspekter har vi i vår granskning funnit att både dokumentation av funktioner och kommentarer i funktioner varierar i hög grad. Det finns funktioner som saknar både dokumentation och kommentarer, eller har något av det och det finns de funktioner som har både och. Utifrån denna variation blir resultatet ur läsbarhetssynpunkt kopplat till varje specifik funktion. Den totala påverkan som detta ger är att koden generellt försämras då det finns alltför många funktioner som inte kan godtas ur detta avseende, dvs. funktionerna saknar

helt, eller har mycket knapphändig dokumentation och kommentarer. Faktorer som vi menar har påverkat denna variation av dokumentation och kommentarer är att det har varit flera olika utvecklare och därmed olika noggrannhet runt dokumentation och kommentarer. Dessutom finns det tydliga variationer när man ser på datumet för skapandet, vilket kan visa på en förändring i synen på nödvändigheten av dokumentation och kommentarer. Senare konstruerad kod är generellt bättre dokumenterad, än tidigare konstruerad.

Vår granskning har även resulterat i att vi överlag finner att både funktioner och variabler har förklarande namn, vilket höjer läsbarheten. Resultatet dras dock ned av att källkoden är stor och omfattande varför det inledningsvis kan ses som oöverstigligt att få grepp om helheten.

Resultatet vi kom fram till avseende källkodens indentering och luftighet, kopplar vi, precis som med dess dokumentation och kommentarer till de olika utvecklarnas personliga stilar. Med den tidstypiska synen på indentering och luftighet menar vi de förändringar över tid som sker runt omkring oss och påverkar oss i hur vi tänker och vad vi gör. Källkodens utseende varierade stort, från att helt sakna indentering och ha minimalt med mellanrum, till att enligt oss ha allt perfekt indenterat och med lagom stora mellanrum. Kunskapen om programkoders läsbarhet och struktur har förbättrats över tiden eftersom marknaden ställer allt högre krav på en lättmodifierad kod.

Granskningen av felhanteringen med avseende på läsbarhet resulterade i dels en enledsgranskning och dels en flerledsgranskning. I enledsgranskade funktioner följde vi en av klientfunktionens anrop till dess leverantörsfunktion. Utifrån de kontrakt vi upprättat fann vi att åtta stycken av de granskade funktionerna inte hanterade returvärdena på ett korrekt sätt. Det gjorde att läsbarheten försvårades eftersom vi inte kunde se vad som hände vid eventuella fel. Som exempel på detta redovisas nedan funktionen **activate()**, Figur 5-1, som anropar leverantörsfunktionen **getAD_Tree_ID(String keyColumnName)**, Figur 5-2. Vid eventuella fel under exekvering kan värdet noll returneras, och det borde ha dokumenterats att det är giltigt och på vilket sätt det ska tolkas. I minst två fall motsvarar det en felsituation, och då kan det vara svårt att felsituationen hanteras som ett giltigt värde av klienten. Detta försvårar förståelsen och felsökningen och kan vara en eventuell felkälla i sig själv.

```
Public class GridController
Klientfunktion:
public void activate (){
//         Tree to be initiated on second/.. tab
if (m_mTab.isTreeTab() && m_mTab.getTabNo() != 0)
{
           int AD_Tree_ID = Env.getContextAsInt (Env.getCtx(), m_WindowNo,
           "AD_Tree_ID");
           if (AD_Tree_ID == 0)
                      AD_Tree_ID=MTree.getAD_Tree_ID(m_mTab.getKeyColumnName());
           m_tree.initTree (AD_Tree_ID);
}
}          //         activate
```

Figur 5-1: Klientfunktion activate()

```
public class MTree
Leverantörsfunktion:
public static int getAD_Tree_ID (String keyColumnName)
{
          Log.trace(Log.l4_Data, ”MTree.getAD_Tree_ID”, keyColumnName);
          if (keyColumnName == null || keyColumnName.length() == 0)
                    return 0;
          string TreeType = null;
          if (keyColumnName.equals(“AD_Menu_ID”))
                    TreeType = TREETYPE_Menu;
          else if //Bottagen kod, liknade den som ovan
          else {
                    Log.error(”MTree.getAD_Tree_ID – Colud not map ” +
keyColumnName);
                    Return 0;
          }
          int AD_Tree_ID = 0;
          int AD_Client_ID = Env.getContextAsInt(Env.getCtx(),
“#AS_Client_ID”);
          string sql = “SELECT AD_Tree_ID,Name FROM AD_Tree “ + “WHERE
AD_Client_ID=? AND TreeType=? AND IsActive=’Y’”;
          try {
          PreparedStatement pstmt = DB.prepareStatement(sql);
          pstmt.setInt(1, AD_Client_ID);
          pstmt.setString(2, TreeType);
          ResultSet rs = pstmt.executeQuery();
          if (rs.next())
                    AD_Tree_ID = rs.getInt(1);
                    rs.close();
                    pstmt.close();
          }
          catch (SQLException e){
                    Log.error("MTree.getAD_Tree_ID", e);
          }
          return AD_Tree_ID;
}   //  getAD_Tree_ID
```

Figur 5-2: Leverantörsfunktion getAD_Tree_ID(String keyColumnName)

I motsats till ovanstående exempelfunktioner visar vi nedan en funktion som hanterar returvärden korrekt genom testning. Returvärdet som erhålls från **getCookieWebUser(HttpServletRequest request),** Figur 5-4**,** är antingen en sträng eller null. Hanteringen av detta tas om hand i **getWebUser(Properties ctx),** Figur 5-3**,** där kontroller sker av returvärdet för att kontrollera de olika möjligheterna. Detta sätt att hantera returvärdet skapar en bra läsbarhet i koden och ger en strukturellt bra lösning.

```
public class LoginLinkTag
Klientfunktion:
private WebUser getWebUser (Properties ctx)
{
//         Get stored User
      WebUser   wu   =   (WebUser)pageContext.getSession().getAttribute
      (WebUser.NAME);
      if (wu != null)
           log.debug("getWebUser - SessionContext:found " + wu);
      else
      {
           wu = (WebUser)pageContext.getAttribute(WebUser.NAME);
           if (wu != null)
                      log.debug ("getWebUser - Context:found " + wu);
      }
      if (wu != null)
           return wu;
//         Check Coockie
      StringcookieUser=JSPEnv.getCookieWebUser
      ((HttpServletRequest)pageContext.getRequest());
      if (cookieUser == null || cookieUser.trim().length() == 0)
           log.debug ("getWebUser - no cookie");
      else
      {
//         Try to Load
           wu = WebUser.get (ctx, cookieUser);
           log.debug ("getWebUser - got " + wu);
      }
      if (wu != null)
           return wu;
//
      return null;
}          //         getWebUser
```

```
package org.compiere.wstore;
public class JSPEnv
Leverantörsfunktion:
public static String getCookieWebUser (HttpServletRequest request)
{
          Cookie[] cookies = request.getCookies();
          if (cookies == null)
                    return null;
          for (int i = 0; i < cookies.length; i++)
          {
                    if (COOKIE_NAME.equals(cookies[i].getName()))
                              return cookies[i].getValue();
          }
          return null;
}
```

Figur 5-4: Leverantörsfunktion getCookieWebUser(HttpServletRequest request)

Flerledsgranskningen omfattade klient- och leverantörsfunktioner, men också andra funktioner som anropade klientfunktionen. Som exempel från flerledsgranskningen har vi valt att visa nedanstående klientfunktion, se Figur 5-5 och leverantörsfunktion, se Figur 5-6. Felhanteringen i båda funktionerna hanteras med undantag. Ur läsbarhetssynpunkt är det svårt att följa flödet i exekveringen därför att klientfunktionen hanterar eventuella undantagsobjekt som kastats från leverantörsfunktionen med att själv kasta undantag. Hanteringssättet försvårar läsbarheten och skapar en komplicerad struktur i koden. Detta sätt att hantera fel fanns i samtliga flerledsgranskade funktioner.

```
package org.compiere.process;
public class ReplicationLocal
Klientfunktion:
Upprättat kontrakt:
pre:   True
post: En anslutning har skapats om det var möjligt, annars har ett
undantag kastats med information om problemet.
private void connectRemote() throws Exception{
//         Replication Info
     m_replication = new MReplication (getCtx(), getRecord_ID());
//
Borttagen kod
```

Anrop:

```
     InitialContext ic = CConnection.getInitialContext(
     CConnection.getInitialEnvironment(AppsHost, AppsPort,
RMIoverHTTP));
     if (ic == null)
           throw new Exception ("NoInitialContext");
     try{
           ServerHome serverHome = (ServerHome)ic.lookup
           (ServerHome.JNDI_NAME);
//         log.debug("- ServerHome: " + serverHome);
           if (serverHome == null)
                      throw new Exception ("NoServer");
           m_serverRemote = serverHome.create();
//         log.debug("- Server: " + m_serverRemote);
//         log.debug("- Remote Status = " +
m_serverRemote.getStatus());
     }
     catch (Exception ex)
     {
           log.error("connectRemote", ex);
           throw new Exception (ex);
     }
```

Figur 5-5: Klientfunktion connectRemote()

```
package org.compiere.db;
public class CConnection
Leverantörsfunktion:
Befintlig dokumentation:
*          Get Initial Context
*          @param env environment
*          @return Initial Context
Upprättat kontrakt:
pre: true
post: returnerar ett context baserat på 'env' om möjligt, annars
null.

public static InitialContext getInitialContext (Hashtable env){
InitialContext iContext = null;
try{
iContext = new InitialContext (env);
}
catch (Exception ex){
  System.err.println("CConection.getInitialContext-"+
  env.get(Context.PROVIDER_URL)
+ "\n - " + ex.toString ()
+ "\n - " + env);
iContext = null;
if (Log.isTraceLevel(10))
           ex.printStackTrace();
}
return iContext;
}
```

Figur 5-6: Leverantörsfunktion getInitialContext(Hashtable env)

Undantagshantering är en vanlig företeelse i Compiere´s källkod. Figur 5-7 visar ett exempel på en klientfunktion som fångar upp det kastade undantaget från leverantörsfunktionen.

```
Try {
        jbInit();
}
catch(Exception e) {
        e.printStackTrace();
}
```

Figur 5-7: Exempel på undantagshantering

Hanteringen som Figur 5-7 visar på är ingen bra lösning ur läsbarhetssynpunkt av resultat eftersom en spårningslogg tar hand om det kastade undantaget. Användaren kanske inte ens blir medveten om att fel inträffat. Det kräver därför att användaren blir medveten om att fel inträffat och sedan att han/hon öppnar spårningsloggen, den ger dock endast användaren en vägledning om vart felet inträffat [13]. *Exceptions* är vanligtvis implementerade i den tro att de inte ska inträffa. Vid ett fel allokerar *exceptions*-hanteringen upp minnet som kanske är allokerat i det normala programflödet, vilket kan leda till att allokering inte kan ske då minnet är fullt [14]. Strukturellt gör denna typ av process koden än mer komplicerad och ineffektiv, vilket även försämrar den ur läsbarhetssynpunkt.

5.2 Struktur

Vår teoristudie har visat på att en bra strukturerad kod kännetecknas av att koden är uppdelad i små moduler. Modulerna ska vara så generella som möjligt, allt för att de skall kunna anropas av andra moduler och därmed kunna återanvändas. Vidare ska modulerna vara strukturerade med så lösa kopplingar, dvs. vara så oberoende, som möjligt av andra moduler. Sammantaget skapar detta en struktur som underlättar framtida utveckling och modifiering av systemet.

Resultatet av vår granskning avseende källkodens struktur visar på en komplex uppbyggnad. Detta grundar sig till stor del på den stora mängden av kod som systemet är uppbyggt av, vilken idag inryms på 1191 filer. Men även följande inverkar till systemets komplexa struktur:

► Beroenden

Vår teoristudie visar på att oberoende moduler blir lättare att återanvända och modifiera. Ur läsbarhetssynpunkt kunde vi se att beroendena mellan modulerna i källkoden från Compiere

är många och svåröverblickbara. Detta är en följd av de många programmeringsstilar som under åren har används i systemets uppbyggnad. Resultatet påverkar strukturen negativt vid framtida utveckling och modifiering av affärssystemet.

► Moduler

Resultatet av vår granskning avseende källkodens struktur visar på en komplex uppbyggnad. Eftersom flerledsgranskningen visade på en mängd funktionsanrop till andra moduler och även till funktioner som vi inte återfann i källkoden. Modulerna i systemet ska varas självständiga enheter. I Källkoden från Compiere var modulerna överlag stora och innehöll en mängd beroenden sinsemellan.

► Lösa kopplingar

Moduler, och dess funktioner bör vara strukturerade med så lösa kopplingar som möjligt, dvs. vara så oberoende av varandra som möjligt. I systemet från Compiere fanns både lösa och hårda kopplingar mellan moduler. Strukturellt ger denna variation en komplex koduppbyggnad.

► Identifiering

Vår granskning försvårades både av storleken på systemet och av att funktionsnamnen i systemet återkom likvärdigt. Det som skiljde de likvärdiga funktionsnamnen åt var enbart parametrar. Utöver detta hittade vi inte alla leverantörsfunktioner i systemet, vilket gav oss omfattande tankemödor.

► Avskärmning

Strukturellt underlättar kontraktet vid modularisering eftersom abstraktionen gör att utvecklaren inte behöver gå in i koden för att veta vilken funktionalitet den har. Här fann vi att källkoden innehåll både moduler som var dokumenterade, likvärdigt med kontraktsuppbyggnad men även moduler helt utan någon som helst dokumentation. Strukturellt resulterade detta i att moduler inte blev avskärmade, utan krävde av användare/utvecklare att koden granskades innan den nyttjas. Avskärmningen försvåras även av de stora och svåröverblickbara moduler som systemet innefattas av.

► Hantering av returvärden och fel

Resultatet vi fann genom vår granskning visar på både bra hantering av returvärden respektive på fel, och på dålig hantering av desamma. Överlag gör vi dock bedömningen att den dåliga hanteringen överskuggar den bra, då den strukturellt sett är negativ för hela källkoden. Exempel på detta, se Figur 5-5 och Figur 5-6.

► Exekveringsflödet

Som tidigare nämnts, påverkar svårigheten att identifiera funktionerna i exekveringsflödet både i enledsgranskningen och flerledsgranskningen. Dessutom är det svårt att följa flödet i exekveringen då klientfunktionen hanterar eventuella undantagsobjekt som kastats från leverantörsfunktionen med att själv kasta undantag. Hanteringssättet skapar en komplicerad struktur i koden. Sättet att hantera fel fanns i samtliga flerledsgranskade funktioner.

6 Slutsatser

Detta kapitel innehåller de slutsatser som framkommit ur C-uppsatsens syfte. Syftet var att granska den öppna källkoden från Compiere i avsikt att ta reda på hur dess läsbarhet och struktur ser ut.

Källkoden från Compiere innehåller 1191 filer, dvs. är omfattande i storlek, samt innehåller funktionsanrop, vilka funktioner som vi inte återfinner i källkoden. Utifrån detta fungerade inte vår urvalsmetod och vi nödgades att manuellt gå igenom koden och välja ut funktioner för vidare granskning. Vår slutsats är därför att källkoden från Compiere är komplex, både vad gäller läsbarhet och struktur. Komplexiteten vad gäller läsbarheten är den stora variationen vad gäller dokumentation och kommentarer, tillsammans med felhanteringen. Den strukturella komplexiteten ligger i systemets storlek, tillsammans med de många och svåröverblickbara beroendena mellan modulerna.

7 Problem och erfarenheter

Projektet Compresion var under uppstart i samband med att vi påbörjade vår uppsats. Följden blev att mycket av vår tid inledningsvis spenderades på att gå på möten och träffa personer som var involverad i projektet, allt för att få en så god bild av helheten som möjligt. Vi hade många frågor, som var svåra att få svar på. En fråga som kvarstår är huruvida systemet är komplett eller inte? Vi har stött på funktioner som varken anropar eller anropas av någon annan funktion i källkoden.

Källkoden från Compiere är i vårt mått sett, stor och omfattande. I och med att vi saknade tidigare erfarenhet av så omfattande system, var det i början svårt att hitta något utgångsläge inför undersökningen. Vårt arbete gick därför en ”krokig väg” med olika försök, åt olika håll. När vi tillslut hittade fram till en möjlig genomförbar metod inför vår undersökning, visade det sig att denna manuella metod var effektiv, men enormt tidkrävande.

Referenser

[1] Software Engineering Research Group, SERG. Karlstad Universitet

[2] SEMLA, teknisk rapport 2000:25 på Karlstad University studies, ISSN 1403-8099

[3] Slumpfunktion, skapad och tillhandahållen av Eivind Nordby, Karlstad Universitet

[4] Martin Flower, UML Distilled, Third Edition

[5] http://www.redpill.se/index.html?module=nyheter&op=view&id=19, 060509

[6] http://www.free-soft.org/mirrors/www.opensource.org/docs/osd-swedish.php, 060509

[7] http://www.compiere.se, 060509

[8] Backman, J, (1998), Rapporter och uppsatser. Studentlitteratur, Lund.

[9] Hall, H, (2005), Föreläsningsmaterial från kursen Vetenskapsteori och forskning, Informatik, Karlstads universitet.

[10] Jacobsen, D.I, (2002), Vad, hur och varför? – Om metodval i företagsekonomi och andra samhällsvetenskapliga ämnen. Studentlitteratur, Lund.

[11] Patel, R och Davidson, B, (2003), Forskningsmetodikens grunder – att planera, genomföra och rapportera en undersökning. Studentlitteratur, Lund.

[12] Carrano, F. M, Prichard, J. J, (2002) Data Abstraction and problem solving with c ++.Pearson Education, Inc

[13] Jalnert, L-E, Wiberg, T. (2004), Datatyper och Algoritmer, Studentlitteratur, Lund.

[14] Dietel, H.M, Dietel, P.J, (2001), C++ how to program, Prentice-Hall, Inc, Upper Saddle River, New Jersey 07458

[15] http://www.unix.se/Grep, 060601

[16] http://www.cs.kau.se/cs/serg/ROSS/, 060906

A Granskade Funktioner

Nedanstående funktioner är exempel på de vi granskat och upprättat kontrakt för, se vidare kapitel 4.1.

A.1 Exempelfunktion 1

Funktionen är klassificerad icketrivial av kategori 3, där kontrakt har upprättats.

Befintlig dokumentation:

```
package org.compiere.model
public class UOM extends X_C_UOM
/**             Get Minute C_UOM_ID
*  @param ctx context
*               @return C_UOM_ID for Minute*/
```

Upprättat kontrakt:

pre: True

post: Returnerar 'C_UOM_ID' för minut från kontexten ctx, 0 om det inte gick.

Anropad:

```
public static int getMinute_UOM_ID (Properties ctx){
if (Ini.isClient()){
                Iterator it = s_cache.values().iterator();
                while (it.hasNext()){
                                UOM uom = (UOM)it.next();
                                if (uom.isMinute())
                                                return uom.getC_UOM_ID();
                }
}
//              Server
int C_UOM_ID = 0;
String sql = "SELECT C_UOM_ID FROM C_UOM "
+ "WHERE IsActive='Y' AND X12DE355='MJ'";              //              HardCoded
try{
                PreparedStatement pstmt = DB.prepareStatement(sql);
                ResultSet rs = pstmt.executeQuery();
                if (rs.next())
                                C_UOM_ID = rs.getInt(1);
                rs.close();
                pstmt.close();
}
catch (SQLException e){
                Log.error("getMinute_UOM_ID", e);
}
```

Klientfunktion

package org.compiere.model

public class UOMConversion

Befintlig dokumentation:

```
/**
*               Convert qty to target UOM and round.
*  @param ctx context
*  @param C_UOM_ID from UOM
*  @param qty qty
*  @return minutes - 0 if not found
*/
```

Upprättat kontrakt:

pre: True

post: Returnerar kvantiteten qty omgjord från enheten C_UOM_ID till minuter som bestämd av kontexten ctx.

```
static public int convertToMinutes (Properties ctx, int C_UOM_ID, BigDecimal qty){
        if (qty == null)
                return 0;
ANROP:
        int C_UOM_To_ID = UOM.getMinute_UOM_ID(ctx);
        if (C_UOM_ID == C_UOM_To_ID)
                return qty.intValue();
//
        BigDecimal result = convert (ctx, C_UOM_ID, C_UOM_To_ID, qty);
        if (result == null)
                return 0;
        return result.intValue();
}       //      convert
```

A.2 Exempelfunktion 2

Funktionen kan erhålla returvärde null som tilldelas guarantee. Det framgår inte att detta tas om hand inom funktionen på något sätt, samt sökning efter now.after(guarantee) i källkoden har varit resultatlös. Funktionen är klassificerad icketrivial av kategori 2, där kontrakt har upprättats.

Befintlig dokumentation:

```
/**
*		Can we download.
*		Based on guarantee date and availability of download
*		@return true if downloadable
*/
```

Upprättat kontrakt:

pre: **true**

post: **returnerar true om getProductDownloadURL lyckas och 'where != null && where.length() > 0', annars returneras false.**

Klientfunktion:

```
public boolean isDownloadable(){
		if (!isActive())
			return false;
		//
		Timestamp guarantee = getGuaranteeDate();
		if (guarantee == null)
			return false;
		guarantee = TimeUtil.getDay(guarantee);
		Timestamp now = TimeUtil.getDay(System.currentTimeMillis());
		//		valid
		if (!now.after(guarantee))   //not after guarantee date
		{
			String where = getProductDownloadURL();  //do we have a link
			return (where != null && where.length() > 0);
		}
		//
```

```
        return false;
}       //      isDownloadable
```

Befintlig dokumentation:

```
/**
*       Get earliest time of a day (truncate)
*  @param time day and time
*  @return day with 00:00
*/
```

Upprättat kontrakt:

pre: **Variabel av typen Timestamp**

post: **Om dayTime == null, returneras null annars ett Timestamp-objekt**

```
static public Timestamp getDay (Timestamp dayTime)
{
        if (dayTime == null)
                return null;
        return getDay(dayTime.getTime());
}       //      getDay
```

Befintlig dokumentation:

```
        /**
         *      Get earliest time of a day (truncate)
         *      @param day day
         *      @param month month
         *      @param year year
         *      @return timestamp
         */
```

pre: **true**

post: **Timestamp retunerad med år, månad och dag av typen long**

Leveranörsfunktion:

```
static public Timestamp getDay (int day, int month, int year)
{       GregorianCalendar cal = new GregorianCalendar (year, month, day);
                return new Timestamp (cal.getTimeInMillis());
}       //      getDay
```

A.3 Exempelfunktion 3

Funktionen är klassificerad icketrivial av kategori 3, där kontrakt har upprättats.

Befintlig dokumentation:

```
/**
*  Set Subject
*  @param newSubject Subject
*/
```

Upprättat kontrakt:

pre: True

post: Om newSubject är en giltig subject-text är m_subject satt och m_valid är oförändrad, annars är m_valid == false och m_subject oförändrad

Leverantörsfunktion:

```
public void setSubject(String newSubject){
        if (newSubject == null || newSubject.length() == 0)
                m_valid = false;
        else
                m_subject = newSubject;
}   //  setSubject
```

Befintlig dokumentation:

```
/**
*               Send No Guarantee EMail
*               @param A_Asset_ID asset
*               @param R_MailText_ID mail to send
*               @return message - delivery errors start with **
*/
```

Klientfunktion
Upprättat kontrakt:

pre: **true**

post: **Returnerar felmeddelande om fältinnehåll saknas eller om mail inte kan skickas.**

```
private String sendNoGuaranteeMail (int A_Asset_ID, int R_MailText_ID){
                MAsset asset = new MAsset (getCtx(), A_Asset_ID);
```

```
        if (asset.getAD_User_ID() == 0)
                return "** No Asset User";
        MUser user = new MUser (getCtx(), asset.getAD_User_ID());
        if (user.getEmail() == null || user.getEmail().length() == 0)
                return "** No Asset User Email";
        if (m_MailText == null || m_MailText.getR_MailText_ID() != R_MailText_ID)
                m_MailText = new MMailText (getCtx(), R_MailText_ID);
        if (m_MailText.getMailHeader() == null ||
        m_MailText.getMailHeader().length() == 0)
                return "** No Subject";

        //              Create Mail
        EMail email = new EMail(m_client.getSMTPHost(),
        m_client.getRequestEMail(), user.getEmail());
        if (m_client.isSmtpAuthorization())
                email.setEMailUser(m_client.getRequestUser(),
                m_client.getRequestUserPW());
        if (m_MailText.isHtml())
                email.setMessageHTML(m_MailText.getMailHeader(),
                m_MailText.getMailText());
        else{
ANROP:          email.setSubject (m_MailText.getMailHeader());
                email.setMessageText (m_MailText.getMailText());
        }
        String msg = email.send();
        if (!EMail.SENT_OK.equals(msg))
                return "** Not delivered: " + user.getEmail() + " - " + msg;
        //
        return user.getEmail();
}       //              sendNoGuaranteeMail
```