Handelhögskolan vid Göteborg Universitet
Insitutionen för Informatik
Magisteruppsats 20p, vt 1999
Handledare: Birgitta Ahlbom

Återanvändning av kod vid systemutveckling

Författare:
Andreas Fredin
Mattias Malmsköld

Abstrakt

Ämnet för denna magisteruppsats är återanvändning av kod vid systemutveckling i Windows-miljön. Syftet är att belysa i vilka situationer som återanvändning lämpar sig bäst. Vi undersökte också vilka källor det finns för återanvändning vid programutveckling i Windowsmiljö. Studien har utförts genom litteraturstudier och fyra intervjuer utfördes med professionella utvecklare för att ta tillvara deras kunskap och åsikter om återanvändning. Vi utvecklade en modul hos Vinga System och i resultatdelen redogör vi för våra egna erfarenheter av programmering med återanvändning. Våra resultat visade på att systemutvecklingen går mot återanvändning av större moduler som t.ex DLL-filer och COM-objekt. Något som också växer är återanvändning av de standardklasser och funktioner som finns i utvecklingsverktyget och i operativsystemet t.ex. i MFC och ATL.

Förord

Denna magisteruppsats på 20p är skriven under våren 1999 på Systemvetarprogrammet vid Handelshögskolan i Göteborg. Vi båda som skrivit uppsatsen har gått inriktningen mot Programvaruutveckling.

Vi vill först och främst tacka Thomas Olsson och Morgan Åberg för all tid och energi de har lagt på att stödja vårt projekt. Båda arbetar på företaget Vinga System AB där vi utfört vårt arbete med magisteruppsatsen.

Ett stort tack till Birgitta Ahlbom på Institutionen för Informatik som varit vår handledare för uppsatsen. Hon har varit ett bra stöd och styrt oss in på rätt spår.

Vi vill även tacka Kenneth Anttila, Håkan Amaren, Patrik Pettersson och återigen Thomas Olsson för att vi fick komma och intervjua Er.

Göteborg Maj 1999

Andreas Fredin Mattias Malmsköld

Innehållsförteckning

1. Introduktion 9
1.1 Bakgrund 9
1.2 Definition av återanvändning av kod 10
1.3 Syfte 10
1.4 Frågeställning 10
1.5 Avgränsning 11
1.6 Vinga System och vår programmodul 11
1.7 Disposition 12
2. Metod 13
2.1 Våra källor Error! Bookmark not defined.
2.2 Källor 13
2.2.1 Litteraturstudier 13
2.2.2 Egna erfarenheter 14
2.2.3 Intervjuer 14
2.2.4 Uppsatsens tillförlitlighet 14
3. Teori – Återanvändning av kod 16
3.1 Allmänt 16
3.1.1 Fördelar 16
3.1.2 Nackdelar 16
3.1.3 Kostnader 17
3.2 Hur använda återanvändningsbar kod 18
3.2.1 Vilka områden 18
3.2.2 ARV – fördelar och nackdelar 18
3.2.3 Egna komponenter 19
3.2.4 Köpta komponenter 21
3.3 Hur bygga återanvändningsbar kod 27
3.3.1 Kodanpassningar 27
3.3.2 Övriga anpassningar 30
4. Optimeringsmodulen 32
4.1 Övergripande om projektet vi genomfört 32
4.2 Beräkningsmodulens programdelar 33
4.2.1 Matrismodul 33
4.2.2 Capm-programmet 33
4.2.3 Användargränssnitt 34
5. Resultat 35
5.1 Erfarenheter under utvecklandet 35
5.1.1 Återanvändning av företagets tidigare producerade kod 35

5.1.2 RAD-verktyg och MFC 35
5.1.3 Standardkomponenter 37
5.1.4 Design av klass 37
5.1.5 Kodkonventioner 38
5.1.6 Namngivning funktioner 38
5.1.7 Fördelning Header/Source-fil 39
5.1.8 Dokumentation 39
5.1.9 Förslag till hur återanvändningsbar kod skall skrivas 39

5.2 Intervjuresultat 40
5.2.1 Intervjumall 40
5.2.2 Allmänt intervjuföretag 42
5.2.3 Fördelar 43
5.2.4 Nackdelar 43
5.2.5 Hitta återanvändbara komponenter 44
5.2.6 DLL-filer & COM-objekt 45
5.2.7 MFC 46
5.2.8 Andra återvinningsätt 47
5.2.9 Allmänna riktlinjer 49
5.2.10 Testning 50
5.2.11 Dokumentation 51
5.2.12 Andra frågor 52

6. Diskussion och slutsats 54

7. Källförteckning 57

8. Bilagor 59

B.1. Ordlista 59

B.2. MFC Översikt 61

B.3. Hungarian kodkonventioner för namngivning 62

B.4. Källkod 64

1. INTRODUKTION

1.1 BAKGRUND

Vad är syftet med en magisteruppsats egentligen? Det är det största enskilda arbete som utförs under utbildningen. Vi tycker att arbetet med magisteruppsatsen erbjuder en möjlighet att fördjupa sina kunskaper inom ett område som intresserar författarna. Här finns också möjligheter att komplettera utbildningen med bitar som saknats i utbildningen. Ofta får författarna anställning inom det område som behandlas i uppsatsen. Detta är en naturlig följd eftersom författarna är intresserade av ämnet och även skaffat sig omfattande kunskap och förståelse för ämnet. Under höstterminen 1998 diskuterade vi om utbildningen blivit som vi tänkt och hoppats. Naturligtvis hade den i stora delar inte blivit det och till vissa delar är detta kanske bra. Ingen av oss hade t.ex. ens hört ordet systemering innan vi påbörjade vår utbildning, medan vi nu anser att det är mycket viktigt och är tacksamma för att vi fick möjlighet att lära oss innebörden av det.

Inför arbetet med magisteruppsatsen kände vi dock att vi saknade kunskap inom vissa specifika områden. Denna kunskap tyckte vi behövdes för att öka våra chanser att få den anställning och möjlighet att arbeta med arbetsuppgifter som vi önskade. Vi har båda ganska likartade intressen och därför enades vi snabbt om två områden som vi ville lära oss mer om inom Systemvetarprogrammet.

1. **Utvecklingsverktyg med återanvändning av kod.** När vi utfört våra programmeringslaborationer under utbildningen har vi hela tiden endast haft en typ av verktyg till vårt förfogande. Detta har varit en texteditor med en tillhörande kompilator som nästan enbart haft tillgång till standardbiblioteken i det aktuella språket. Vi har lagt ned mycket tid på att utveckla delar som gränssnitt, databasuppkopplingar osv men inom dessa områden i programvaruutvecklingen finns det redan bra och väl fungerande lösningar utvecklade. Därför kommer det alltmer nya programutvecklingsmiljöer som kallas RAD (Rapid Application Development) - verktyg. Dessa verktyg, t.ex. Microsofts Visual Studio, bygger på att standarddelar utvecklas på ett mycket enklare sätt genom att t.ex. rita upp gränssnittet enligt Windows standarden på skärmen. Dessa program som rönt ett mycket stort intresse inom systemutvecklingsbranschen gör att systemutvecklaren inte behöver lägga så mycket tid på rutinuppgifter och kan satsa mer på de svåra uppgifterna i programutvecklingen. Detta gör utvecklarens arbete mera utmanande och förhoppningsvis roligare.

2. **Program för matematiska beräkningar.** Under vår utbildning har vi uteslutande konstruerat en typ av program - administrativa program. Det är troligtvis inom detta område de flesta systemvetare fortsätter att arbeta efter utbildningen. Vi är dock mer roade av matematik och ekonomi. Inom denna genre finns också en utmaning om hur det logiska problemet skall lösas i sig och inte bara de programmeringsmässiga

problemen runt detta. Detta breddar också våra framtida arbetsområden i företag från administration till att omfatta även mer teknisk utveckling. Vi skulle därför vilja få mer kunskap om mer avancerade matematiska beräkningar (än de fyra räknesätten).

Eftersom vi snart slutar denna utbildning ville vi ha en genomgripande överblick över vad RAD-verktyg och annan återanvänding av kod innebär samt vilka områden i programutvecklingen som underlättas av detta. Vi bestämde oss för att i uppsatsen besvara dessa frågor samt att ge svar på hur kod, tänkt att återanvändas, skall konstrueras. Vi hade också för avsikt att lära oss mer matematisk programmering. Vi beslutade oss för att lämna detta område utanför uppsatsen för att inte blanda in för många olika områden.

1.2 DEFINITION AV ÅTERANVÄNDNING AV KOD

Vi definierar återanvändning av kod både som att återanvända sådan kod som företaget tidigare producerat och att använda de hjälpfiler och funktioner som kompilatortillverkaren, operativsystemet och andra programmoduls tillverkare tillhandahåller.

1.3 SYFTE

Syftet med vår uppsats är att lära oss att arbeta mer effektivt genom återanvändning av kod samt att ge läsarna av denna uppsats en inblick i detta område. Vi hoppas att läsaren får en inblick i grundbegrepp och vilka hjälpmedel det finns vid återanvändning av kod vid programutveckling samt att ge tips om hur kod som är tänkt för återanvändning skall byggas.

För att kunna förstå och ta till sig innehållet i uppsatsen krävs det att läsaren har kunskaper motsvarande en systemvetare och kunskap om de grundläggande begreppen inom objektorienterad systemutveckling.

1.4 FRÅGESTÄLLNING

Vi kommer att besvara följande frågor i vår uppsats:

- I vilka situationer i programutvecklingen är det lämpligt att utnyttja återanvändning av kod?
- Vilka källor finns för återanvändning av kod vid programutveckling i Windowsmiljö?
- Vad är viktigt att tänka på när kod skapas för återanvändning?

1.5 AVGRÄNSNING

Sommerville(1995) hävdar att de största produktivitetsvinsterna erhålls genom återanvändning av material som producerats i analys och designfaserna. Vi kommer dock i vår uppsats endast att behandla återanvändning av programvarukod från realisationsfasen.

Exempel på hur återanvändning av kod kan gå till kommer att utgå från programmeringsspråket C++. Naturligtvis är de flesta slutsatser även gångbara för de flesta andra programmeringsspråk och då i synnerhet de objektorienterade språken. Den miljö vi valt att inrikta oss mot i studien är Windows miljön.

1.6 VINGA SYSTEM OCH VÅR PROGRAMMODUL

Vi har under arbetet med magisteruppsatsen skapat en programmodul för företaget Vinga System[1] i Göteborg, som utvecklar finansiell programvara för stora placerare och aktörer på finansmarknaden. Vinga System har två stycken olika inriktningar. Den ena skapar programvara som i realtid distribuerar börskurser och annan ekonomisk information till aktieplacerare i hela Europa företrädesvis i Norden. Den andra inriktningen konstruerar mer skräddarsydda system för specifika kunder. De konstruerar och säljer program som gör att företag får god överblick och bättre kontroll över skulder och placeringsbara (relativt likvida) tillgångar. Det är emot denna del av företaget som vi kommer att vara inriktade.

Den programmodul som vi har skapat i vår magisteruppsats har till uppgift att räkna ut vilka kombinationer av aktier som är bäst att inneha för en viss önskad avkastning. Beräkningarna baserar sig på en ekonomisk modell som kallas CAPM (Capital Asset Pricing Model). Denna utgår från aktiekursernas historiska variation, samvariation med andra aktier och dess historiska avkastning. Modellen finns djupare beskriven i resultatdelen i denna uppsats.

Vi har utvecklat vår programmodul i utvecklingsverktyget Microsoft Visual C++. Vinga System har koncentrerat sin utveckling till att använda just detta utvecklingsverktyg i praktiskt taget all utveckling som görs. Företaget har därför byggt upp en mycket god kompetens inom detta område.

[1] För ytterligare information se http://www.vinga.se

1.7 DISPOSITION

Denna uppsats har följande struktur:

Kapitel 2 som består av metoddelen beskriver hur vi gått tillväga för att genomföra denna magisteruppsats.

Vi kommer sedan i kapitel 3, teoridelen, beskriva delar som hur kod återanvänds i projekt samt hur kod produceras för återanvändning.

I kapitel 4 beskriver vi det program som vi under arbetet med vår magisteruppsats utvecklat åt Vinga System. Syftet med detta var att få en egen djupare förståelse om.problemområdet.

I kapitel 5, resultatdelen, återger vi våra egna åsikter och de erfarenheter av återanvändning av kod som vi erhållit under utvecklandet av programmodulen. Förutom detta innehåller kapitlet också fyra systemutvecklares åsikterna om ämnet.

Kapitel 6, diskussion och slutsats, jämför de olika åsikter och bilder som framkommit. Utifrån dessa för vi en diskussion och drar slutsatser.

Kapitel 7 redovisar de källor vi haft för att genomföra detta arbete.

I bilaga 1 finns en ordlista över de ord som kräver en mer utförlig förklaring för vissa läsare.

Bilaga 2 innehåller en graf över de hjälpklasser som finns inbyggda i den kompilator som vi använt.

Bilaga 3 innehåller de namnkonventioner för variabler och funktioner som ofta används när kod konstrueras för återanvändning.

I Bilaga 4, återfinns en del av den källkod som vi producerat i vårt projekt.

2. METOD

2.1 TILLVÄGA GÅNGSSÄTT

Vi har studerat en uppsats gjord av Duplancic, Lindberg(1998). Dessa kombinerade flera olika metoder för att undersöka sitt problemområde och kallade sitt tillvägagångssätt "Crossroad Metaphor". I vårt arbete använde vi oss av deras metafor och utnyttjade tre av metoderna i deras studie nämligen litteraturstudier, praktisk erfarenhet samt intervjuer. Vi ansåg att genom användandet av dessa tre metoder utnyttjade vi på bästa sätt tidigare nedskrivna åsikter, våra egna lärdomar under arbetet att ta fram programmet samt den kompetens som finns på både på det företag där vi genomförde vår uppsats och de företag där vi utförde våra intervjuer.

Vi skrev om ett ämne som är inriktat mot en smal nisch. Exempelvis i intervjuerna hade vi frågor som gemene man inte har någon större erfarenhet av och vi behövde därför noggrant förklara situationen för de som intervjuades. Vi kunde därför inte i detta arbete skicka ut 100 st frågeformulär till intressanta personer. De frågor som vi ville få svar på krävde också längre svar än bara ett ja och ett nej eller något annat kryssalternativ. Därför har valde vi dessa tre metoder och ett angreppssätt som är av kvalitativ art.

Med hjälp av kvalitativa undersökningsmetoder kan en större mängd information erhållas från ett mindre antal undersökningsenheter. De går in mer på djupet till skillnad mot kvantitiva som är mer breda. Detta för att öka förståelsen för vissa faktorer. Vi tyckte att kvalitativa metoder var lämpliga att använda eftersom det som vi skulle undersöka inte kunde mätas i kvantifierabara mått.

Vi ansåg inte att det finns någon lösning för dessa frågor som är bäst för alla branschområden. Men för att vi skulle kunna bestämma ett bra tillväga gångsätt för den bransch som vi undersökte var det viktigt att komma ut till företaget och se vilken miljö som personerna där arbetade i. Detta ligger i linje med synsättet som trycker hårt på att sätta sig in i människorna i studiens vardag. Vi som utförde undersökningen hade också våra åsikter och dessa stämde överens med det synsätt som tycker det är bra med även mer personliga åsikter och synsätt i forskningen.

2.2 KÄLLOR

2.2.1 LITTERATURSTUDIER

Vi under arbetet med vår magisteruppsats läste vi parallellt en hel del litteratur. Under de första veckorna läste vi främst in oss på problem runt den programmodul som vi byggde. Detta inbegrep både de ekonomiska modellerna och många frågor runt hur

programmeringen syntaxmässigt skulle utformas. När vi hade kommit en bit in i utformandet av programmet tittade vi på vilka problem vi stött på och justerade vårt ämnesval utifrån detta. Därför kunde vi då börja söka efter litteratur. Förutom böcker använde vi oss mycket av artiklar för att få en aktuell bild över hur återanvändningen såg ut i de senaste programutvecklingsmiljöerna.

Vi bedömde att det var mycket viktigt att vi gjorde en noggrann litteraturstudie för att vi inte skulle lägga för mycket fokus på saker som redan hade bra lösningar och istället hitta en nisch på de mer problemfyllda bitarna. I enlighet med det fenomenologiska synsättet är det även här tillåtet att tänka kritiskt när texten läses och försöka sätta sig in i vilken roll och utgångspunkten som författaren har och bedöma vad det har för påverkan i vårt fall.

2.2.2 Egna erfarenheter

Under vår utbildning hade vi lärt oss flera olika programspråk men vi hade ändå ganska begränsade erfarenheter av programutveckling. Magisteruppsatsen innebar dock att vi fick en längre tid med ett visst projekt och i en ny modern programutvecklingsmiljö. Vi fick även möjligheten att sitta "ute i verkligheten" med de intryck som detta medförde. Därför tyckte vi att det var viktigt att personligen reflektera över de problem och tänkbara lösningar på området som vi ansåg fanns. Att vi utvecklade ett eget program gjorde att vi fick en större förståelse för problemen när vi skulle diskutera dem med våra intervjupersoner. Vi uppskattar att vi tillsammans lagt ned runt 300 timmar per person på utvecklingen av modulen. Våra egna erfarenheter gav oss också en bättre bas att stå på när vi skulle försöka tolka vad andra personer tyckte i olika frågor.

2.2.3 Intervjuer

Vi intervjuade fyra personer på lika många företag i Göteborgsregionen. Kravet vi ställde på intervjupersonerna och företagen var att de var professionella systemutvecklare och att de utvecklade program inom C++ eller Java.

Intervjuerna dokumenterades med hjälp av en bärbar bandspelare för att vi skulle få med alla svar och koncentrera oss på svaren så att frågorna inte missuppfattades och att eventuella följdfrågor skulle kunna ställas. Under intervjun med EHPT kunde vi dock tyvärr inte använda oss av bandspelare.

2.2.4 Uppsatsens tillförlitlighet

Två kriterier för bedömning av uppsatsens tillförlitlighet är validitet och reliabilitet.

Validitet
Validitet innebär i vilken utsträckningen uppsatsens resultat stämmer överens med verkligheten. Mäter författaren verkligen det som han eller hon föresatt sig att mäta. Är det rätt frågor som ställs och ställs de verkligen till rätt personer. Yttre validitet står för överförbarhet d.v.s. i vilken mån uppsatsen resultat kan generaliseras.

Vi anser att vår uppsats uppnår såväl inre som yttre validitet. Den inre validiteten erhålls genom att de personer som vi intervjuade har en lång erfarenhet av att arbeta inom branschen. De har en god kännedom om de områden som vi bestämt oss för att undersöka. Den yttre validiteten uppnår vi genom användandet av fler olika metoder på samma problem. Informationen är också inhämtad från flera oberoende källor.

Reliabilitet
Med reliabilitet menas om uppsatsens resultat är tillförlitligt. Detta innebär att om ytterligare en undersökning utförts på samma population skulle denna komma fram till samma resultat. För att uppnå fullständig reliabilitet förutsätts en frånvaro av slumpmässiga mätfel. Vid användandet av kvalitativa intervjuer där intervjupersonernas personliga åsikter och uppfattningar behandlas kan det vara svårt att påvisa reliabilitet. Detta beror på att en persons åsikter kan variera över tiden och från en situation till en annan. Detta problem har vi valt att hantera genom att beskriva den situation som datan är hämtad ifrån. Vi ger en bakgrundsbild av personerna och de företag som de arbetar inom och när detta är klargjort kan läsaren själv bedöma reliabiliteten.

3. TEORI – ÅTERANVÄNDNING AV KOD

3.1 ALLMÄNT

Det ställs allt större krav på systemutvecklare att effektivisera sitt arbete genom att återanvända kod från tidigare projekt och att använda sig av effektiva verktyg. Denna möjlighet har hittills endast utnyttjats i begränsad omfattning. Att inte dra nytta av denna möjlighet är ett stort slöseri av resurser och branschen måste lära sig att se program mer i enskilda moduler som samverkar och som kan återanvändas. En jämförelse kan göras med industrin där Volvo låter underleverantörer produktutveckla och ansvara för komponenter som Airbags och bilstolar. Det enda Volvo specificerar är de yttre egenskaperna på komponenten.

3.1.1 FÖRDELAR

Fördelarna som finns med återanvändning är många. Enligt Kemerer(1997) är den stora fördelen att det förhoppningsvis blir effektivitetsvinster med synsättet och att det därför blir billigare att utveckla programvaran. Eftersom koden är testad och använd i tidigare projekt så ökar chansen att den fungerar på ett pålitligt och mer effektivt sätt. Systemutvecklingstiden kommer också att minska om komponenter byggs ihop istället för att utvecklas i alla delar från grunden. McClure(1997) pekar på att det är lättare för en projektledare att uppskatta tidsåtgången och hur mycket resurser som krävs för att slutföra ett utvecklingsarbete. Hon hävdar även att det oftast är de enklaste och mest förekommande uppgifterna för programmeraren som oftast blir föremål för återanvändning. Detta innebär att systemutvecklaren får mer tid över till de svåra och mer stimulerande uppgifterna.

3.1.2 NACKDELAR

Det finns även nackdelar med konceptet som förklarar varför det ännu inte har fått större genomslag i branschen. Enligt McClure(1997) är det svårt att snabbt hitta sådana komponenter som behövs. Det finns inget register att slå i utan det finns en mängd ställen att leta på. När komponenterna väl hittats är det många gånger svårt att lösa hur kommunikationen mellan de olika delarna i systemet skall fungera. Det krävs ofta anpassningar runt komponenten för att de skall vara tillämpningsbara i det aktuella projektet. Många gånger råder osäkerhet om komponentens tillförlitlighet. Om det skulle finnas brister eller felaktigheter är det mycket svårt att få dem åtgärdade. Eftersom den tekniska utvecklingen sker så snabbt tycker en utvecklare ofta att det finns ett effektivare sätt att lösa problemen på med den nya tekniken. Förutom detta skapas ofta extra kod utöver vad som faktiskt hade behövts i det aktuella projektet vilket motverkar

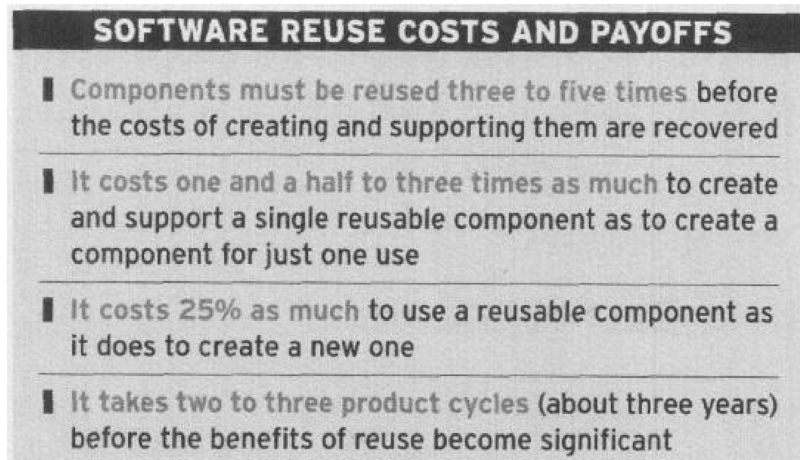
effektivitetsvinsterna. Radding(1998) pekar även på att det krävs extra tid för att testa, kvalitetssäkra och dokumentera kod som skall återanvändas.

Trots de problem som finns är trenden att återanvändningen av komponenter i systemutvecklingen ökar. Det är därför viktigt att som systemutvecklare lära sig använda återanvändning i utvecklingsprocessen. Den som inte gör detta kommer att förlora i produktivitet, oavsett hur duktig personen är på sitt arbete, eftersom det är ett mer effektivt sätt att utveckla på.

3.1.3 KOSTNADER

Att bygga återanvändningsbara komponenter kräver mer utvecklingstid och kostnaden för systemutvecklingen ökar. Enligt Orenstein(1998) så måste en komponent som är utvecklad i återanvändningsbar form generellt sett återanvändas 3 till 5 gånger innan den har betalat tillbaka den ökade investeringen som det innebär att skapa en sådan komponent (se figur 1). Ofta så skapas dock inte sådana komponenter på företag även om återanvändningsgraden är högre än denna. Detta beror enligt författaren på att det oftast i dessa projekt råder en stressad situation och att det inte finns tid att skapa allmängiltiga komponenter.

Det behövs enligt Kemerer(1997) skapas incitament för anställda att skapa användbar kod och dela med sig till andra utvecklare av denna. Han anser att företagen noga måste tänka igenom hur dessa frågor skall hanteras så att komponenter verkligen blir återanvända samt hur de som utvecklat dessa skall bli kompenserade.



SOFTWARE REUSE COSTS AND PAYOFFS

- Components must be reused three to five times before the costs of creating and supporting them are recovered
- It costs one and a half to three times as much to create and support a single reusable component as to create a component for just one use
- It costs 25% as much to use a reusable component as it does to create a new one
- It takes two to three product cycles (about three years) before the benefits of reuse become significant

Figur 1 Code reuse: Reality doesn't match promise

Computerworld; Framingham; Aug 24, 1998; David Orenstein

3.2 HUR ANVÄNDA ÅTERANVÄNDNINGSBAR KOD

3.2.1 VILKA OMRÅDEN

Det kan vara svårt att lokalisera de komponenter som är möjliga att återanvändas. Radding(1997) har gjort en uppdelning i olika kategorier som kan bli föremål för återanvändning. Det är lättast att hitta bra komponenter med sådan funktionalitet som ofta används i andra program. Ett bra exempel är gränssnitt vars utseende är likartat i de flesta program. Ett närbesläktat område är olika systemgränssnitt såsom databasuppkopplingar, Internetuppkopplingar. Det återvinns också klasser och funktioner som utför operationer på ett väl avgränsat område. Ett exempel är den matrisklass med tillhörande matematiska operationer som vi konstruerat under vår uppsats. Idag är det mest vanligt att återanvända endast mindre komponenter liknande de vi beskrivit ovan. Denna typ av återvinning är småskalig och ganska enkel.

Ett område som enligt Radding växer snabbt är att återvinna stora enskilda moduler. Detta tror han blir något av ett paradigmskifte och kommer att medföra en helt annan effektivitet i systemutvecklingen. Det kommer också att krävas att det blir en annan mentalitet och förändring av vanor hos den som programmerar. Exempel på sådana här moduler är att dela upp ett ekonomisystem i en materialhanteringsmodul och en inköpsmodul osv. Dessa har sedan ett standardiserat gränssnitt så de lätt kan sättas samman med andra moduler.

3.2.2 ARV – FÖRDELAR OCH NACKDELAR

C++ är ett objektorienterat språk. Applikationer som är gjorda i C++ är uppdelade i olika klasser. Varje klass innehåller ett antal funktioner inom ett avgränsat område exempelvis vissa matematiska beräkningar. Ett objekt skickar en förfrågan till ett annat objekt genom en funktion och får tillbaka ett svar. En av grundtankarna bakom det objektorienterade synsättet är att återanvända klasserna inom andra applikationer som skall göra liknande saker.

Enligt Lattanzi, Henry(1998) så finns det två sätt att återanvända klasser i programspråket C++. Dels kan tidigare producerad klass kopieras och infogas i projektet och användas som den är. Detta kallas för Black-Box reuse och användaren behöver bara veta hur kommunikationen med klassen går till. Det andra tillvägagångssättet är att en klass återanvänds och byggs ut genom arv och på detta sätt erhålls en utökad funktionalitet. Detta synsätt kallas White-Box reuse eftersom det krävs en större inblick i hur klassen internt är uppbyggd för att utföra detta. Enligt författarna så är det oftast klara produktivitetsvinster om Black-Box reuse utnyttjas medan om White-Box metoden används är det mycket tvivelaktigt om det effektivitetsmässigt lönar sig med återanvändning. I detta fallet krävs det en mycket van programmerare som har förmåga

att snabbt sätta sig in i kod för att ha utbyte av detta. Fördelen med White-Box reuse är att komponenten får ett väl anpassat utseende till den specifika situationen.

3.2.3 EGNA KOMPONENTER

Vi tar nu upp det som traditionellt räknas som återanvändning av kod. Detta inbegriper att tillvarata sådan kod som du själv eller någon annan på företaget tidigare har producerat.

När en modul eller klass har utvecklats finns det många metoder att paketera denna för vidare användning. Det lättaste sättet är naturligtvis att spara klassen som den är och sedan kopiera denna när den skall infogas i framtida programmeringsprojekt. Det finns dock några intressanta alternativ som är användbara, speciellt om det är en större mängd kod som skall återanvändas. Vi har inhämtat kunskap om dessa ifrån Grimes, Stockton, Reilly, Templeman (1998) och MSDN Library Visual Studio 6.0(1998).

3.2.3.1 LIB-moduler

Ett sätt att minska komplexiteten för en utvecklare till en väl avgränsad modul är att kompilera den klass som önskas återanvändas till en s.k. LIB-fil. Den som skall utnyttja denna funktionalitet kopierar sedan denna fil och inkluderar den i sitt eget programmeringsprojekt. Klassen och funktionerna anropas på samma sätt som innan och utvecklaren får tillgång till headerfilen i LIB-modulen och hur den ser ut. LIB-bibliotekets funktioner länkas till programmet och bakas in i EXE-filen under kompileringen.

3.2.3.2 DLL-filer

Det finns också en fil vilken har extensionen DLL(Dynamic Link Libraries)*[2]. Detta är ett bibliotek och precis som övriga inkluderingsbibliotek innehåller det ett antal funktioner. En DLL länkas däremot till programmet först när det exekveras och själva DLL-filen behöver inte ingå bland programmets filer utan bara finnas någonstans på hårddisken. Fördelarna med DLL är att om funktionerna som finns i filen används av flera applikationer behövs endast en DLL då denna blir de åtkomlig för alla andra applikationer som behöver använda den. Själva applikationerna blir därmed mindre. Det går också att förändra och lägga till funktioner som används av programmet utan att EXE-filen behöver förändras. Detta åstadkommes genom att funktionen ändras i DLL-filen. Det som är viktigt att tänka på är att inte interfacet* i DLL-filen får ändras dvs att kommunikationen med filen i ut- och invärden inte ändrar struktur.

[2] Tecknet * anger att ordet finns förklarat i ordlista i bilaga 1

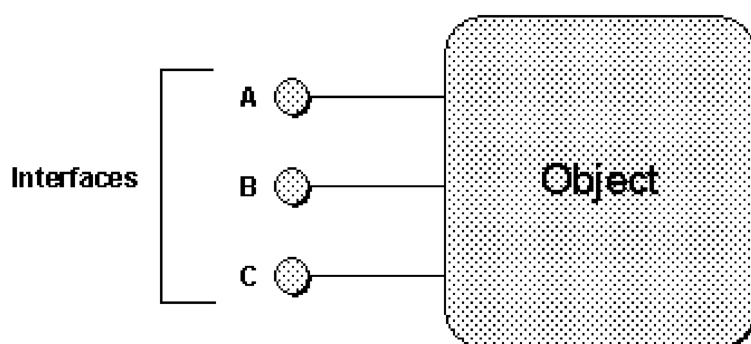
Detta arbetssätt är naturligtvis bra för en programmerare för att kunna skapa en DLL-fil som går att anropa från flera olika program som är skapade för att köras på samma dator.

3.2.3.3 COM-objekt

Ett sätt att paketera den färdiga koden är att lägga den i DLL-filer motsvarande det operativsystemet som vi beskrivit ovan. Både på Vinga System och några av de företag som vi intervjuat har i allt större utsträckning börjat använda sig av denna teknik och detta verkar vara en stark allmän trend. Ett av de stora syftena med denna teknik är just att det skall bli möjligt att dela sina funktioner med andra utvecklare och applikationer.

Ett annat ord för dessa DLL-filer är COM-objekt*. Detta är en standard som är tänkt att användas i plattformsoberoende projekt utvecklat av företagen Microsoft och Digital. I den miljö som vi arbetat i, MS Visual C++, finns det ett hjälpbibliotek som kallas ATL* (Active Template Library) som på ett motsvarande sätt som MFC* är hjälpklasser för att bygga upp COM-objekt. Vad är det då som skapat den popularitet som COM-objektet idag har?

En mycket stor anledning till detta är att när objektet används av andra projekt är det redan kompilerat. Objektet behöver inte vara skrivit i samma progammeringspråk som det projekt som anropar objektet är skrivit i. I ett C++ projekt kan vi därför använda funktioner som är skrivna i exempelvis Visual Basic eller Java. Vad standarden som företagen har byggt upp behandlar är hur COM-objekten skall kommunicera med andra objekt. Standarden bryr sig dock inte om hur objektet internt är programmerat. Dessa objekt kan kommunicera inte bara med en applikation utan alla applikationer på datorn. Ett växande område är också att dessa objekt kan kommunicera över Internet.



Figur 2. Ett COM-objekt som stödjer tre interface A. B och C.
Källa: MSDN Library 6.0(1998)

Kommunikationen med dessa objekten sker genom funktioner som kallas interface (se figur 2). Denna specifikation anger vilka in och utdata som funktionen ger. Om det finns program som använder sig av COM-objektet är det fortfarande möjligt att gå in och ändra i objektet utan att ändra i applikationen. Så länge som inte interfacet har förändrats behövs inga ändringar göras utanför objektet. Detta objekt är även mycket flexibelt för att

lägga till flera interface. Modellen kan därför bra hantera förändringar i krav över tiden. Nackdelen med programmeringen är att den upplevs som ganska svår att komma igång med i början.

3.2.4 KÖPTA KOMPONENTER

Återanvändning av kod kan ta många olika former. Under denna rubrik tar vi upp en aspekt som oftast inte förknippas med återanvändning. Nämligen att utnyttja de resurser som finns inbyggda i kompilatorn och i datorns operativsystem.

3.2.4.1 Programspråkets inkluderingsfiler

Den mest grundläggande typen av återanvändning av kod är användningen av redan färdiga klasser och så kallade inkluderingsbibliotek. I dessa finns färdiga modeller för t ex vektorer, stöd för grafik, matematiska funktioner m.m. Istället för att skriva egen kod för en vektor och olika funktioner som kan utföras på denna vektor, så inkluderas istället en fil i programmet. I dessa filer finns ofta en mängd fördefinierade operatorer och funktioner som kan användas på den vektor som skapats. Vid kompileringen letar kompilatorn rätt på deklarationen av dessa i biblioteket och inkluderar dem när det exekverbara programmet skapas. Dessa bibliotek och klasser finns som standard för ett programmeringsspråk som t ex C++, här finns exempelvis "Iostream.h" som hanterar de olika in- och utmatningar av strömmar med data till programmet.

Enligt Skansholm(1996) finns det en stor poäng i att den kod som byggs så långt som möjligt endast använder sig av syntaxen från standardbiblioteken. Detta har flera anledningar. Dels är det mycket lätt att flytta koden ifrån en viss plattform och utvecklingsmiljö till en annan. Koden behöver därför vanligtvis inte omarbetas om den flyttas ifrån PC till UNIX miljö. En annan stor fördel är att syntaxen är välkänd för alla programmerare och det är därför lätt att sätta sig in i ny kod. Den kanske viktigaste delen att återvinna är just kompetensen från andra systemutvecklare.

3.2.4.2 Windows DLL-filer

Som Simon(1997) skriver glöms det ofta bort att Windows operativsystem är ett program som alla andra. För att kunna återanvända de funktioner som används när Windows körs har Microsoft lagt många av dessa funktioner i just DLL-filer som vi beskrivit ovan. Bland det bästa med detta är att det ligger ungefär samma uppsättning av filer på alla datorer med Windows operativsystem.

Dessa DLL-filer i Windows kan alltså användas i de applikationer som skapas i Windows miljö. I DLL-filerna ligger en mängd olika funktioner som behandlar allt från filhantering, konfigurering av hårdvara till funktioner för att starta om Windows.

Samlingen av DLL-filer i Windows miljön brukar också kallas för API* som betyder Application Program Interface. Utbudet av funktioner som finns här är enormt men det kan vara lite svårt att kommunicera med dessa funktioner. Det finns dock lättare sätt för C++ programmerare genom MFC som beskrivs nedan.

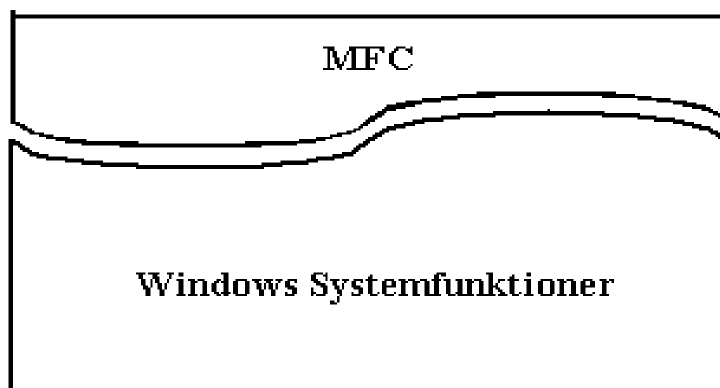
3.2.4.3 Microsoft Foundation Classes

Det brukar också finnas speciella bibliotek och standardklasser för respektive programmeringsverktyg och även för olika operativsystem. Ett exempel på en samling sådana standardklasser är Microsofts MFC.

Vad är då MFC? Vi har hämtat vår kunskap om ämnet ifrån Brain(1998) och Prosise(1997b). Förkortningen står för Microsoft Fundation Classes och innehåller över 200 olika standardklasser, antalet växer för varje version som kommer ut på marknaden. MFC klasserna kan användas av alla program i Windows miljö, såväl Visual Basic, C++ och Java kan inkludera MFC i programmen. Vi skall nu försöka göra en kort överblick vad som finns att tillgå i dessa klasser

- Applikations arkitektur. Klasser som skapar den grundläggande funktionaliteten i ett program. Detta inbegriper bl.a. ramen runt programmet, vad programmet skall göra när det sker inmatning från tangentbord eller musklick osv.
- Grafiska klasser. Dessa specificerar hur grafiken på skärmen såsom knappar, skrivfält, menyer m.m. skall se ut.
- Ritobjekt. Klasser som specificerar upp de vanligaste funktionerna inom ritning som finns i MS Paint.
- Filhantering. Stöd i olika klasser för hur filhanteringskommunikation i olika former skall gå till.
- Felhantering. Klasser för upptäckande och hantering av de vanligaste typerna av fel som kan inträffa t.ex. minnesfel, Internetfel.
- Struktur - listor, vektorer, kartor. Filer med stöd för att spara olika datatyper och för omvandling dem emellan
- Internethantering. Stöd för att koppla upp dig och arbeta mot Internet ifrån ditt program.
- OLE*. Funktionalitet som gör att det går att starta andra program inifrån sitt eget. Detta ger ett enklare utbyte av data mellan klasser.
- Databashantering. Klasser som underlättar uppkoppling och kommunikation med databaser.

- "Allmänna saker". Det finns även en mängd andra klasser bl.a. för hantering av tid, systeminformation



Figur 3 MFC ligger som ett tunt skal ovanpå Windows

I Windows API finns ju också stöd för uppbyggnad av fönster, grafik och filhantering så vad är skillnaden? Skillnaden är att det är lättare att använda MFC för denna lägger sig som ett lager emellan den egna koden och Windows API (se figur 3). I Visual C++ ser dessa klasser ut som vanliga C++ klasser och därför kan en mycket stor del av all den funktionalitet som finns i Windows API nås. Detta gör det mycket enklare och det går snabbare att använda MFC istället. I MFC finns ett objekt som kan kallas basklass - CObject - det är utifrån det objekt som de flesta klasser härstammar. Denna "superklass" innehåller de mest grundläggande datatyperna och medlemsfunktionerna. Klasserna ärver sedan egenskaper av varandra i klasshierarkin[3].

Microsoft Foundation Class Library bygger upp en ram runt programmet som utvecklas. Det förser programmet med gränssnitt, händelsehantering och övriga funktionalitet som är gemensam för de mesta Windows program. Utvecklarens roll är att programmera de delarna som är specifika för sitt eget program.

Fördelar

En av fördelarna med MFC är när ett grafiskt gränssnitt skall skapas. Istället för att skriva all den kod som behövs, så kan grafiska hjälpmedel användas för att rita upp gränssnittet. Sedan är det bara att klistra in de menyer, knappar, hjälp och kortkommandon som önskas, koden för detta genereras automatiskt. Tidsvinsten som här erhålls är markant, det öppnar även möjligheter för mer ovana programmerare att kunna skapa väl fungerande och avancerade gränssnitt utan att besitta den egentliga kunskapen.

Det går också att skapa egna klasser utifrån MFC klasserna som ärver egenskaperna från den MFC klass som är basklass. Detta kan vara till stor nytta om det t ex skall skapas ett eget gränssnitt eller skapas specialvarianter av en vektor.

[3] Se bilaga 2

Nackdelar
Den största nackdelen med att använda sig av MFC är att det automatiskt följer med en hel del funktioner och medlemsvariabler som inte är önskvärda. Det går inte att välja vissa klasser utan hela "paketet" följer med. Prosise(1997b) skriver också att klasserna är allmänt skrivna och därför ofta saknar viss funktionalitet. Författaren uppskattade att de flesta professionella utvecklare som arbetar mot MFC har egna bibliotek härledda från MFC. Att detta går att göra är naturligtvis en styrka men produktivitetsvinsterna som beskrevs ovan minskar ju då drastiskt.

3.2.4.4 RAD-verktyg

Det resonemanget om RAD-verktyg som nu följer är hämtat ifrån boken Rapid Application Development av Martin(1991) där det presenteras en jämförelse mellan olika tekniker att utveckla system samt ifrån Eriksson, Wallström(1998).

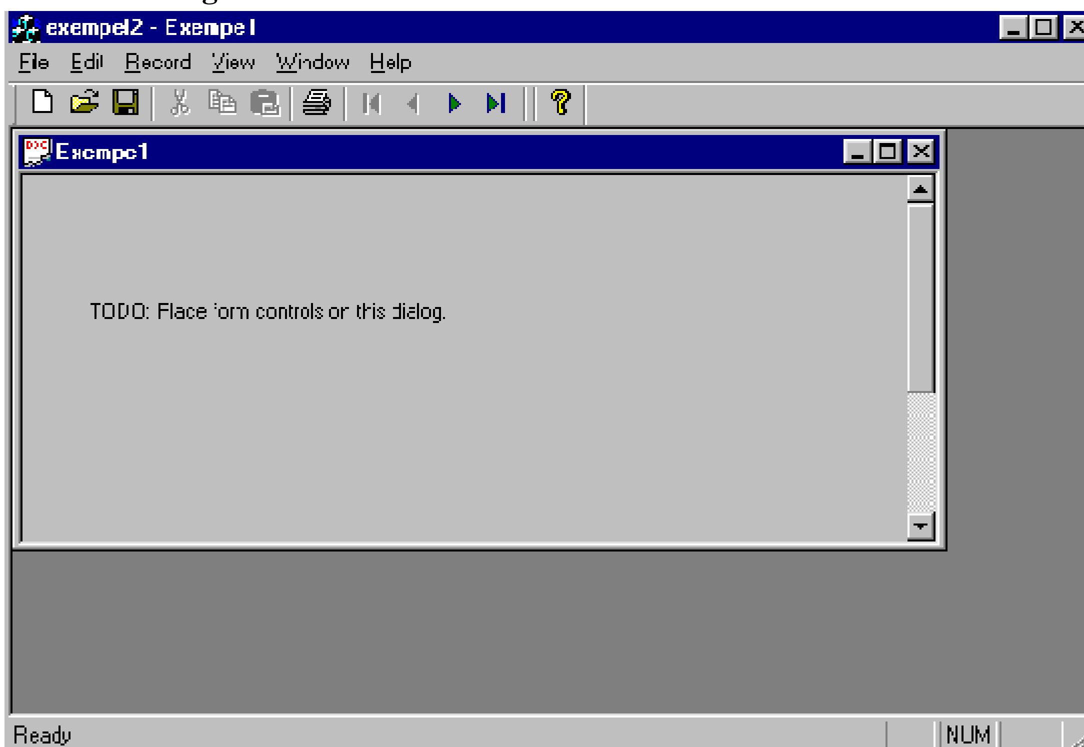
Traditionell utveckling
Innan Radverktygen fanns gjordes programutvecklingen av systemutvecklare som använde sig av papper för att ta fram specifikationerna. Dessa gick sedan vidare till programmerare som satt och kodade dessa i någon form av löpande band princip. Språket som de använde sig var så kallade tredje generationens utvecklingsverktyg bl.a. COBOL och FORTRAN, koden som producerades krävde en mycket omfattande debugging*.

Detta var ett mycket ineffektivt sätt att utveckla program på. Det tog lång tid att färdigställa systemen, kvalitén var ofta bristfällig och det blev dyrt. Datorer och informationssystem började att inta en allt viktigare roll inom såväl affärsvärlden som försvaret, dessa blev den "nya" tidens strategiska vapen. Systemen som behövdes blev allt större och mer komplexa, kraven ökade på kortare utvecklingstid, lägre kostnader och enklare samt billigare underhåll.

Det krävdes ett nytt arbetssätt och nya utvecklingsverktyg för att möta dessa krav. I början på 1980-talet kom så fjärde generationens utvecklingsverktyg, ett verktyg som bestod av en mängd olika integrerade verktyg. Samtidigt så blev SQL en standard. SQL gav möjligheten att på ett enkelt och snabbt sätt hantera data i en relationsdatabas. Det togs också fram verktyg för prototyping, för att snabbt kunna ta fram ett fungerande system och låta slutanvändaren testa detta. Då föddes också den iterativa utvecklingen.

CASE-verktyg blev ett begrepp, dessa verktyg kopplade ihop design fasen med själva programutvecklingen. CASE står för Computer Aided Software Engineering. Det finns över hundra olika CASE-verktyg, som stödjer olika delar av programutvecklingen. Möjligheten öppnades för att grafiskt rita upp modeller och designstrukturer och sedan automatiskt generera kod utifrån analys- och designdokumenten. Nya mer avancerade kodgeneratorer som byggde på det objektorienterade tankesättet togs fram för dessa CASE-verktyg. Det fanns alltså stöd för alla faser i utvecklingsprocessen, det var möjligt att snabbt få fram en prototyp för testning av användarna. Stöd ingår för designfasen, dokumentation och SQL för databashantering.

RAD Utveckling



Figur 4 Automatgenerad programstomme i Visual C++

Ur CASE-verktygen växte ett annat utvecklingspaket, RAD (Rapid Application Development). Dessa verktyg saknar stöd för analys- och designfasen men det finns ett visst stöd för dokumentation. Den stora nyttan med dessa verktyg är Visualfunktionen, som ger användaren möjlighet att bl.a. enkelt rita upp ett användargränssnitt och sedan automatiskt generera koden för utformningen. Det finns också en rad olika ”wizards” som automatiskt skapar en del av den vanligaste gränssnitten bl.a. för menyer och verkygsrader. Programfönstret i figur 4 har skapats automatiskt i en ”wizard”. Inte en rad kod har hittills skrivits ändå finns ett fullt körbart program med möjligheter att spara, skriva ut o.s.v.

Sedan finns det färdiga funktioner och klasser som kan kopplas till gränssnittet. Det gör att det går snabbt att utveckla system, därav kommer namnet Rapid Application Development. RAD innebär till stor del återanvändning av befintlig kod genom att system utvecklas på färdiga klasser och funktioner. Dessa komponenter skall sedan enkelt kunna fogas samman till färdiga applikationer. Det enda som behövs är ”klistret” mellan komponenterna. Detta synsätt kortar både tiden och kostnaden för utvecklingsprojektet. Själva tanken med RAD-verktyg är att snabbt ta fram ett färdigt system till en lägre kostnad och även ha en högre kvalitet. Problemet med synsättet är att det förutsätts finnas god tillgång till de viktigaste komponenterna. De delarna i applikationen som generellt finns i många program ingår ofta i MFC och RAD-verktygen som vi sett ovan. Svårare är

naturligtvis med applikationens speciella delar där det får ske egen utveckling eller återanvändning av sin egna kod som vi kommer att beskriva senare.

Användningen av RAD-verktyg växer mycket snabbt. Inom Windows-miljön ökar användningen av bl.a. Microsoft Access och Visual Basic mycket snabbt och är de mest kända. Även de traditionella programspråken får allt mer RAD-stöd. Exempel på detta är bl.a. Visual Java ++ och Visual C++ som har ett stort hjälpmedel för grafisk utformning. Även företag som Borland har programpaket som bygger på Visual konceptet. Vi har valt att koncentrera oss på Microsoft Visual C++ 6.0. Det är den miljö inom vilken vi arbetat fram modulen åt Vinga System.

Visualprogrammen bygger på händelsestyrd programutveckling vilket innebär att det är händelser som styr vad som händer i programmet. Det är när användaren t ex klickar på musen eller trycker ned en tangent som någonting sker. Den kod som skall exekveras när en viss händelse utförs ligger som ett paket i koden för fönstret och den aktuella händelsen t ex en knapptryckning. Detta fungerar mycket bra ihop med det objektorienterade synsättet med att det anropas en viss funktion i ett objekt när en händelse inträffar.

Det finns också nackdelar med att använda sig av RAD-verktyg. Radding(1998) skriver att ofta genereras mycket kod genom ”Wizards” och det kan därför medföra att koden blir svår att överblicka och att den blir långsam vid exekvering. Ofta är det därför svårt att använda RAD-verktyg till applikationer med många användare. Vi har ovan beskrivit att det är snabbt och enkelt att använda sig av RAD-verktyg för att generera viss kod och detta stämmer. Detta förutsatt att tillvägagångssättet är bekant. Effektivaste sättet att utnyttja verktyget för sitt eget behov kan dock ta lång tid att lära och om verktyget bytts ut mot ett annat på marknaden kan tillvägagångssätten vara mycket annorlunda.

3.3 HUR BYGGA ÅTERANVÄNDNINGSBAR KOD

3.3.1 KODANPASSNINGAR

Det finns en mängd olika förändringar som är lämpliga att utföra med sin vanliga kod om den skall göras återanvändbar. Biddle och Tempero(1998) tycker att de flesta förändringar handlar om att göra kod strukturerad och lätt att förstå och därför bör vara utgångspunkt även när kod produceras som inte skall återanvändas. Det finns dock ej något allmänt accepterat recept över vilka ingredienser som skall ingå i en återanvändningsbar kod. Därför tas nedan upp några punkter som olika författare nämnt. Detta är därmed ingen heltäckande bild men några av de viktigaste punkterna att tänka på.

3.3.1.1 Sommerville's krav på komponenter som skall återanvändas

Sommerville(1995) har lagt fram tre områden som han anser vara viktigast när kod skall anpassas. De tre områdena är namngeneralisering, funktionsgeneralisering och undantagsgeneralisering.

Namngeneralisering

Det finns en mängd konventioner skapade över hur namngivning av variabler och funktioner i C++ skall gå till. Tanken med dessa konventioner är att det skall bli lättare att förstå och att återanvända någon annans kod. Den konvention som idag ser ut att vunnit mest acceptans kallas för Hungarian Coding Conventions. Nedan följer några exempel på konventioner enligt denna metod medan en mer utförlig beskrivning finns i appendix 3.

Använd namnkonventioner för variabler

- "Ch**"** före charactervariabler ex: *chGrade*
- "B" före booleanvariabler ex: bEnabled
- "N" före integervariabler ex: nLenght

Använd speciella funktionsnamn

Set	Sätter objektets egenskaper
Get	Erhåller objektets egenskaper
Is	Frågar om ett objekt har en viss egenskap

En mycket stor del av en programmerares arbete går åt till att finna den rätta syntaxen. Det är ofta en stor tröskeleffekt att lära sig ny hjälpbibliotek men när detta är gjort en gång är de mycket lättare att förstå i fortsättningen. Det är mycket praktiskt med konventioner över uttryckssätt. Naturligtvis kan det vara omständligt att tänka på detta

under programmeringen och ibland är konventionerna konstigt uttryckta. Det är dock viktigt att inte frångå dem för det.

Funktionsgeneralisering

För att kod skall gå att återanvända måste funktionerna täcka de flesta situationer och inte bara de som behövs i det program som klassen/modulen ursprungligen är skapade för. Det är därför viktigt att kartlägga vilka situationer som klassen/modulen kommer att kunna återanvändas inom och utifrån det skapa de nödvändiga funktionerna. Det kommer troligtvis att innebära att programmet kommer att innehålla fler funktioner men det kan också bli aktuellt att minska antalet funktioner. Enligt Gamma(1995) måste kommunikationen med användaren bli så lätt som möjligt. Om det är en modul med flera klasser är det exempelvis bra att göra så att användaren bara behöver kommunicera med en klass.

Undantagsgeneralisering

Precis som att kartlägga vilka funktioner som kan tänkas behövas hos en programmodul måste det också noga utforskas vilka fel som kan tänkas uppstå. Särskild om andra skall använda din kod är det viktigt att fel tas hand på ett strukturerat sätt och inkluderas i gränssnittet mot andra komponenter så att felmeddelande vidarebefordras.

3.3.1.2 McClure's krav på komponenter som skall återanvändas

I McClure(1997) tar kortfattat upp en mängd olika punkter som är värda att ta i beaktning när kod konstrueras för återanvändning.

Generellt uppbyggd

Den skall byggas på ett generellt sätt så att den kan användas på flera ställen i systemet eller i flera olika system. Interfacet som andra använder för att kommunicera med klassen skall vara abstrakt och enkelt. Ytterligare funktioner som kan tänkas behövas i framtiden kan läggas in för att öka komponentens återanvändningsbarhet. Den skall också vara indelad i två olika delar, en fast och en flexibel del. När sedan komponenten återanvänds så är det endast den flexibla delen som får lov att ändras. Om programmeraren fick lov att ändra i hela komponenten så skulle detta kunna leda till oönskade sidoeffekter p.g.a. ändringarna. Detta tankesätt utnyttjar fördelarna både med black-box reuse, den fasta delen och white-box reuse den flexibla delen.

Byggd som enskild modul

Vid återanvändning så är större bättre och desto högre nivå som återanvändning sker på desto bättre är det. Att återanvända en modul har mycket stora fördelar. Oftast behövs det inte göras några ändringar alls i källkoden utan modulen kan användas som den är, detta ger mycket stora tids- och kostnadsvinster. Modulen skall dock vara väl avgränsad inom ett specifikt område så det blir lätt att förstå vad denna kan göra.

Plattformsoberoende
Om komponenten är plattformsoberoende så ökar återanvändbarhetsgraden kraftigt. Den kan då återanvändas oberoende av vilken plattform som används. Det skall vara testat att komponenten verkligen uppfyller detta krav och den skall testas i ett antal olika plattformsbaser.

Applikationsoberoende
Komponenten skall kunna användas oberoende av vilken typ av applikation det är som den återanvänds inom. Detta underlättas mycket genom just Sommervilles(1995) tre punkter som vi beskrivit ovan.

Pålitlig
Chansen ökar att komponenten återanvänds om den är pålitlig. Med detta menas att den skall utföra det som det är tänkt att den skall göra. Det är en fördel om detta görs på snabbt och effektivt sätt. Felhantering skall finnas för de fel och undantag som kan inträffa. Denna skall fånga upp dessa händelser strukturerat och meddela system att det inträffat ett fel och vad som orsakade detta felet.

Lättförståelig/väl dokumenterad
Den skall vara uppbyggd och kodad på ett sådant sätt så att det är enkelt för någon annan att snabbt sätta sig in i koden och förstå vad den gör. Detta uppnås genom användande av namngeneralisering t ex hungarian naming convention. Multipla arv bör undvikas då klasshierarkin blir mycket svår att följa och förstå. Dokumentation bör följa någon av de standarder som finns t ex UML*.

Anpassningsbar/utbyggbar
Det skall vara lätt att anpassa komponenten så att den passar in i det system där den skall återanvändas. Nya funktioner skall kunna läggas till utan att det påverkar övriga funktioner. Komponenten skall vara självständig och löst kopplad till andra komponenter. Detta möjliggör att det är lätt att uppgradera komponenten med en ny utgåva eller byta ut denna mot en annan komponent utan att andra delar av systemet skall behövas byggas om.

Testad och verifierad
När komponenten blivit utvecklad är det viktigt att den testas noga. Det ställs högre krav när det gäller testningen av en komponent som skall vara föremål för återanvändning. Om denna innehåller allvarliga fel kan det få katastrofala följder för de framtida system som använder sig av komponenten.

Det är mycket bra om någon annan än personen som skrivit koden går igenom den. Genom detta arbetsförfarande elimineras de mest grundläggande felen. Sedan testas komponenten i kompilatorns debugger.

Ett annat sätt att testa som är mycket vanligt speciellt när det gäller komponenter som skall återanvändas är att ett testprogram skapas. Med hjälp av detta testas sedan komponenten.

Ytterligare ett sätt att testa på är att släppa ut så kallade demoversioner till vissa utvalda kunder. De testar sedan programmet och rapporterar in felen till företaget som sedan kan åtgärda dessa.

Eftersom de komponenter som återanvänds troligtvis redan har används i minst en applikation medför detta en större säkerhet eftersom de hårdaste testarna –användarna- redan har använt denna.

Lätt underhållen
En stor kostand för ett system är underhållet av det. Därför är det viktigt att det är lätt att underhålla komponenten. Här brister det ofta. För om en annan utvecklares komponent används förloras den djupa inblicken över vad komponenten utför. Det är ofta svårt att få personen som gjort koden att rätta denna i efterhand.

Inkapslad
Information om hur klassen internt fungerar skall vara väl inkapslade i modulen och dold för systemet. Detta kan uppnås med hjälp av objektorientering. Objektet kapslar in data och tillhandahåller endast funktioner för att manipulera datan. Det är viktigt att noga tänka igenom så att användaren av klassen får ett så lätt gränssnitt som möjligt att kommunicera mot. Bara de nödvändiga funktionerna skall vara deklarerade som publika och möjliga att nå utifrån medan de funktioner som endast används internt skall vara privata och ej möjliga att nå utifrån.

3.3.2 ÖVRIGA ANPASSNINGAR

3.3.2.1 Företagsbeslut

För att det skall bli lönsamt att producera återanvändningsbar kod måste denna, som vi tidigare sagt, bli återanvänd ett flertal tillfällen senare. I så gott som samtliga artiklar vi läst betonas att ledningen måste propagera för att det är positivt med återanvändning och bistå med utbildning och lösningar på frågor om hur enskilda resultatenheter inom företag skall bli ersatta för att konstruera och lämna ut kod till andra enheter på samma företag. Scheier(1996) skriver att företaget i denna process skall skapa mått som mäter graden av återanvänder kod och löpande redovisa resultat hur detta utvecklas. Företagsledningen måste besluta om standarder för exempelvis vilka utvecklingsverktyg, operativsystem och databaser som företaget skall ha för att inte få en alltför heterogen miljö. Om antalet miljöer skall begränsas är det mycket viktigt att utgå från var kompetensen finns och i vilken miljö företagets produkter i dagsläget utvecklas. På detta sätt minimeras risken att kompetens går till spillo. Annars kan bristen på beslut bli dyrbara för det är mycket billigare att återanvända redan existerande komponenter än att skapa återanvändbara komponenter från grunden. Ytterligare ett råd från författaren är att det är bra att införa återanvändningen stegvis på ett företag. Börja med de mest besparande områdena. Ofta är

det en lärorik process som skapar bättre förståelse för hur fortsättningen skall se ut. Börja med att skapa mindre komponenter för återanvändning.

3.3.2.2 Dokumentation

Det är nyttigt att förstå att skapa funktioner och klasser som är enkla och att det också bifogas dokumentation som snabbt förklarar de vanligaste frågorna en användare har på klassen. För en klass som är tänkt att användas enligt Black-Box reuse som vi såg ovan skiljer sig behovet av dokumentation mot om White-Box reuse används på komponenten. Vid Black-Box reuse behövs det en övergripande beskrivning om vad klassen/funktionen utför. Det behövs också information om vad det är för parametrar som funktionen kräver och vad den ger ifrån sig. Vid White-Box reuse bör det förutom detta ingå också en mer detaljerad beskrivning över vad funktionen gör för att snabbt kunna sätta sig in i denna.

3.3.2.3 Katalogisering

Ett stort problem med återanvändning av kod som återkommer i flertalet artiklar och litteratur på området är svårigheten att hitta den rätta koden i det aktuella fallet. Enligt Scheier så bör det fattas ett gemensamt beslut inom ett företag hur koden skall katalogiseras. Ett förslag hur detta skulle kunna gå till är att ha ett Intranet där ett bibliotek med återanvändbara komponenter registreras.

Exempel på hur detta skulle kunna vara uppbyggt ges av McClure(1997). Komponenterna kan delas in olika familjer beroende på form de har.

- Kod
- Analys/Design dokument
- Applikationer
- Prototyper
- Template
- Text
- Skelettkod
- Testskript
- Dokumentation

Om denna katalog verkligen skall användas krävs också att den fortlöpande uppdateras och hålls aktuell.

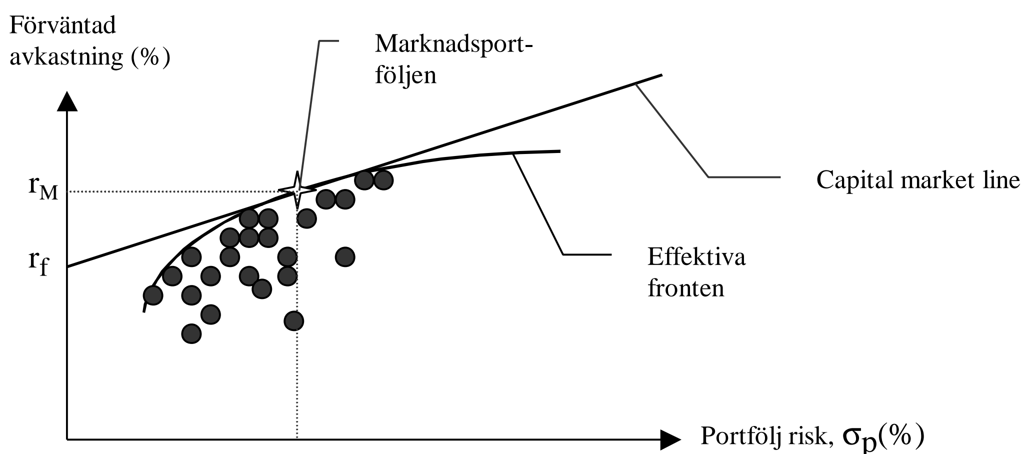
4. OPTIMERINGSMODULEN

4.1 ÖVERGRIPANDE OM PROJEKTET VI GENOMFÖRT

Teorierna till denna ekonomiska diskussionen är hämtade ifrån Copeland(1992). Programmodulen som Vinga System ville få skapad var att formalisera och anpassa den ekonomiska modellen CAPM (Capital Asset Pricing Model) för att användas i ett datorprogam. CAPM går i korthet ut på följande: Ett grundläggande antagande inom den ekonomiska teorin är att en placerare kräver mer genomsnittlig avkastning på en placering ju mer riskfylld denna är. Det är därför aktier, generellt sett, ger bättre avkastning än ett bankkonto som har mycket låg risk. För en investerare är det därför intressant att veta hur stor risk en placering innebär. Att mäta den risk som uppstår vid ett inköp av en aktie är dock inte så lätt. Ett vanligt mått är varians, mätt t.ex. över hur mycket dagsavkastningen varierar mot den genomsnittliga avkastningen. Om en aktieportfölj diversifieras, dvs sprids ut genom köp av aktier i flera olika företag, gärna i olika branscher, sjunker dock variansen beroende på att aktierna inte rör sig fullständigt lika. Det finns ekonomiska modeller uppbyggda för att räkna ut hur hänsyn till samvariationen mellan aktier skall utföras, samt hur den totala risken sedan bedöms. Antag att den historiska risken och avkastningen på aktier gäller, då går det att räkna ut den optimala portföljen för varje risknivå.

Vår uppgift var att ur Vinga Systems ekonomiska databaser hämta indatan om aktiens förväntade avkastning, aktiens varians, dess samvariation och utifrån detta räkna ut den optimala aktiesammansättningen. Den optimala aktiesammansättningen i detta fall är den portfölj som har den minsta variansen för en viss given avkastningsnivå. Om denna process upprepas för en mängd olika förväntade avkastningar skapas den så kallade effektiva fronten som syns i figur 5.

De kunder som Vinga System haft i åtanke när de skapat kravspecifikationen för denna programmodul är i huvudsak professionella fondförvaltare. De har ett antal krav på sig såsom att de inte får utföra blankningsaffärer och måste ha en viss grad av diversifiering på portföljen. För att detta program skulle bli intressant fick även dessa aspekter ingå i den programmodul som vi skapade.



Figur 5 CAPM-modellen

Under vårt projekt var det främst två arbetsuppgifter som tog lång tid att utföra. Dels var det en klass för hantering av matriser och vektorer samt operationer mellan sådana. Vinga System önskade att denna klass skulle vara möjlig att återanvända i framtida liknande projekt. Den andra uppgiften var skapandet av modulen för att beräkna de optimala portföljen. Vinga hade ett äldre men mindre avancerat program för denna uppgift som vi fick återanvända det vi ville ur. Utöver detta utformade vi ett testgränssnitt till programmet i Visual C++ men detta var av begränsad funktionalitet och i den aktuella utvecklingsmiljön tog detta ej lång tid att utveckla.

4.2 BERÄKNINGSMODULENS PROGRAMDELAR

4.2.1 MATRISMODUL

Problemet som vårt program klarar av är i grund och botten ett ekvationssystem. Algoritmen som konstrueras skall naturligtvis vara flexibel och kunna hantera ett ospecificerat antal aktier med restriktioner på sig. Ett lämpligt sätt att hantera ekvationssystem i programmering är att använda sig av matriser eftersom det är svårt att i förväg bestämma hur många variabler som finns. Här skapade vi själva en kodmodul som är tänkt att i framtiden kunna återanvändas. Vi började därför att konstruera en klass som effektivt hanterade matriser och vektorer. De funktioner som ingår är bl.a. att nå enskilda element i matrisen, multiplicera matriser med varandra och invertera matriser.

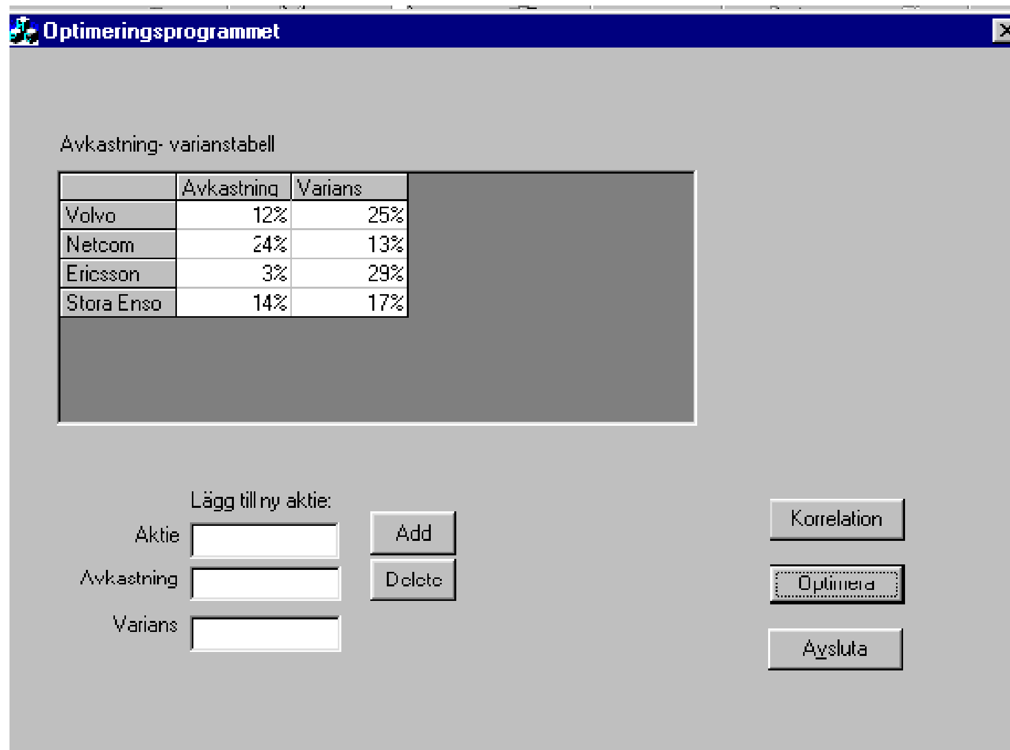
4.2.2 CAPM-PROGRAMMET

När klassen för matrishantering var färdigspecificierad så började arbetet med det riktiga uppdraget, optimeringsmodulen. Vinga System har redan idag en beräkningsmodul för

CAPM i ett av sina program. Vad som skiljer den modul vi konstruerat med deras äldre version är att vår modell har stöd för att på många olika sätt ange begränsningar för portföljvalet som det äldre programmet endast tog hänsyn till i begränsad utsträckning. Ett exempel på en sådan begränsning är att aktierna Ericsson, Volvo och Netcom måste uppgå till minst 10% av portföljen och högst 35%. För detta program skapade vi två klasser. En klass som hanterande restriktionerna och en klass som hanterade de enskilda aktiernas risk och avkastning samt räknade ut den bästa risknivån för varje given avkastning. Den senare klassen har tagit den mesta tiden i anspråk att utforma och innehöll många funktioner för behandling av den data som matats in. En stor del av tiden gick åt till att försöka förutspå vilka fel som kan uppkomma vid en körning och att programmera hanteringen av dessa.

4.2.3 ANVÄNDARGRÄNSSNITT

Efter vi var klara med modulen konstruerade vi ett gränssnitt (se figur 6) till den. I den produkt som Vinga System senare skall släppa där vår modul ingår är det av stor vikt att just gränssnittet med presentationen av resultatet ser inbjudande och pålitligt ut. Det gränssnitt som vi konstruerade kommer därför inte finnas i den slutliga produkten. Vi konstruerade det av två andra anledningar. Dels för att vi på ett enklare och överskådligare sätt skulle kunna testa att vår beräkningsmodul räknade rätt och dels för



Figur 6. Användargränssnitt till optimeringsprototypen.

att vi skulle få inblick i hur ett gränssnitt konstrueras och hur våra klasser kopplas ihop i Visual C++ miljön.

5. RESULTAT

Som vi beskrivit i vår metoddel har vi delat upp vår resultatredovisning i två huvuddelar. Dels beskriver vi de egna erfarenheter av konstruktionsfasen av den beräkningsmodul som vi utförde hos företaget Vinga System. Den andra delen redovisar de intervjuer som vi gjort med professionella systemutvecklare på fyra olika företag i Göteborgsområdet. De ger sin syn på återanvändning av kod i den verksamhet de befinner sig i.

5.1 ERFARENHETER UNDER UTVECKLANDET

5.1.1 ÅTERANVÄNDNING AV FÖRETAGETS TIDIGARE PRODUCERADE KOD

I vårt arbete fick vi tillgång till färdig kod av företaget vid två olika tillfällen. Första gången var när vi skulle programmera funktionen för att invertera en matris. Vinga Systems funktion för detta kom från en klass som använde sig av en annan metod för att hantera enskilda element. Beräkningarna på elementen var annars helt lika och vi hade därför stor nytta av denna funktion. Vi visste i grova drag vad funktionen utförde och det var lätt att återanvända de intressanta delarna.

Det andra tillfället vi återanvände kod från företaget var när vi tittade på deras gamla lösningen av problemet. Detta program var inte alls lika avancerat som vår modul. Vi tänkte återanvända allt som var möjligt från den gamla lösningen men arbetet att sätta sig in i denna klassen var dock mycket mer tidskrävande än vad vi trodde. Till stor del berodde det på att vi inte är så vana programmerare och helt saknade erfarenheter av denna typ av matematisk programmering. Men även en van programmerare hade haft svårigheter att förstå vad programmet gör då det bestod av en sparsamt kommenterad kod med många iterationer.

Vi märkte senare att vår specifikation var såpass bra att vi skulle lagt ned mer tid på att studera denna istället för deras gamla kod. Denna löste ändå inte problemet och vi hade för dålig inblick i hur den fungerade. Att i detta läge avgöra vilken kod som var användbar var naturligtvis svårt. Vi drog lärdomen att det är olämpligt att dra resonemanget om återanvändning av kod för långt. I detta fall hade vi gjort tidsvinster på att bygga koden på egen hand.

5.1.2 RAD-VERKTYG OCH MFC

I teoridelen beskrevs RAD-verktyg. Visual C++ som vi använde för vår programmering räknas till denna kategori. En stor fördel med denna miljö är det enkla sättet att generera gränssnitt som vi kommer att beskriva nedan. Miljön stödde återanvändning av kod på fler sätt än detta exempel. Bl.a. är en finess att när ett visst objekt anropas och en funktion

från detta objekt skall användas kommer det upp en funktionslista i bokstavsordning över de funktioner som finns tillgängliga. Om en funktion valts och knappen F1 trycks in erhålls en beskrivning över funktionen. Dessutom visades en överblick av in och utdatan till funktionen. Det underlättade mycket att få detta så konkret uppritat se figur 7.

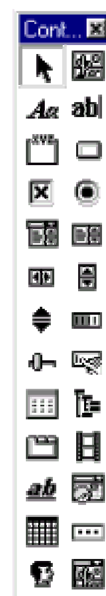
```
        //De två grundrestriktionerna - alltid med
        hinder.Set(0, vOnes, true, 1);
        hinder.Set(1, vMean2, true, avkast);
        hinder.Set(2, restr7, true, 0.5);

        if (!hinder.GaussCheck())
        {
            cout<<endl<<"Senaste restriktionen strider mot de andra"<<endl;
            break;
        }

        //Räkna upp avkastningskravet
        avkast+=0.001;

        //Starta beräkningen
        portfolio.SetConstrList(&hinder);
        if (!portfolio.Optimize(&risc, &x, &z, &y))
        {
            cout<<"Effektiva fronten slut";
            break;
        }

    }

    Matris.
    ////  CholeskyInPlace
          El
          element              size();jj++)
          GetCol
          GetOutput            ,"%f\t",x[jj]);
          GetRow               ,"\n");
       f  GetSize
          GetSum
          GetSumRows
    ////  Invert               ////////////////////////////
```

Figur 7. Lista över tillgängliga funktioner till klassen. Miljön är Microsoft Visual C++ 6.0

Vi har under kursen Objektorienterad systemutveckling skapat ett gränssnitt med hjälp av språket QT. Detta gränssnitt skapade vi genom att först skriva koden för hur fönstren skulle vara uppbyggda sedan kompilerades koden och först när den exekverades syntes resultatet på skärmen. All finjustering av framförallt koordinaterna fick göras genom ändringar i koden som sedan kompileras om. Denna process fick upprepas ett stort antal gånger. Detta jobb upplevde vi som ineffektivt och det tog mycket lång tid. Resultatet blev ej heller vad som var tänkt från början.

När vi nu skulle skapa vårt gränssnitt som skulle kopplas ihop med beräkningsmodulen så valde vi att använda oss av Visual-funktionen. Med hjälp av denna funktion ritade vi först upp själva grunden till gränssnittet. Den funktion som vi använde oss av heter MFC applikations generator och där finns de olika komponenter som deklareras i MFC[4]. De enda komponenterna vi använde var ett antal standardkomponenter som t.ex. knappar, textfält och scrollmenyer se figur 8. Allt som ritas upp bygger på Windows standarden och är välkänt av de flesta användare. Processen att skapa gränssnittet upplevde vi som mycket enkel och den påminde mer om att använda ett ritverktyg än att programmera.

[4] Se bilaga 2



Figur 8. Komponentmeny för konstruktion av gränssnitt

Koden genererades sedan automatiskt när programmet kompileras. Nackdelen var att det genererades mycket kod som var svår att sätta sig in i om vi skulle vilja göra några ändringar. Det fanns även en god hjälp för hantering av olika händelser, t.ex. ett kortkommando eller musklick. När någon händelse inträffar går programmet automatiskt till en viss funktion som exekveras där det enkelt går att specificera vad som skall inträffa. I vårt fall lade vi helt enkelt in våra ovan beskrivna klasser i projektet. Vi hade ritat upp en knapp i vårt gränssnitt. Till denna knapp kopplade vi en funktion. I denna funktion anropades sedan vår optimeringsmodul. Processen att infoga vår modul med detta gränssnitt gick därför mycket lätt.

5.1.3 STANDARDKOMPONENTER

Vinga System hade för tidigare projekt använt sig av en annan matrisklass. Företaget önskade dock att vi för vårt program konstruerade en ny matrisklass som byggde på att dess värden sparades i en STL (Standard Template Library) komponent som heter valarray. När vi började på Vinga System förstod vi inte fördelen med detta men efter ett tag insåg vi de vinster som detta skulle generera. Standardbiblioteket i C++ hade utökats med just STL sedan Vinga System utvecklade sin klass. Bl.a. hade en headerfil för hantering av vektorer, med hjälp av komponenten valarray lagts till. Denna lämpade sig bra för att hantera matriser. Den klass företaget tidigare använt var skriven med hjälp av MFC. Vår klass har därför fördelen att den kan användas och köras på alla C++ kompilatorer som bygger på den nya C++ standarden. Vi har tidigare sett att ett av de största kraven på återanvändningbar kod är att den skall vara användbar på flera olika plattformar och utvecklingsmiljöer. Även om Vinga System eller andra som koden skrivs åt endast använder sig av en miljö i dagsläget är det dumt att låsa in sig.

5.1.4 DESIGN AV KLASS

När vi utformade matrisklassen var det svårt att veta vilka funktioner som skulle ingå. Därför utgick vi ifrån de funktioner som vi behövde för vårt arbete. När vi tyckte att vi fått med alla funktioner som krävdes i vår klass för matrishantering fortsatte vi att programmera de klasser och funktioner som krävdes för att få ut de effektiva portföljerna enligt programspecifikationen. Vi märkte då att vi hade missat viktiga funktioner i vår klass. Vi fick därför gå tillbaka och lägga till fler funktioner allt eftersom. Om andra applikationer skulle använda vår matrisklass måste det troligtvis anslutits fler funktioner som hade varit önskvärda att få med. Det är därför viktigt att noga tänka igenom och testa ut hur en återanvändbar klass kommer att fungera för att få den så generell och användbar som möjligt. Som ovan beskrivits så sjunker ju produktivitetsvinsterna snabbt för användning av återanvändbar kod om en programmerare måste gå in och lägga till

funktioner i en klass genom arv (White-Box reuse) gentemot om det bara är att använda klassen rätt av (Black-Box reuse)

5.1.5 KODKONVENTIONER

Koden som vi återanvände ifrån Vinga System var skriven i enlighet med Hungarian Coding Conventions. Företaget tyckte även att vi skulle använda detta i vår kod. I början så var det mycket förvirrande med nya reglerna för namngivning. Det tog naturligtvis extra tid att själv börja programmera efter denna konvention för att skapa en god återanvändningsbar kod. Vi vande oss emellertid snart och tycker nu istället att de underlättar programmeringen. Det är som att lära sig indentera koden, jobbigt i början men självklart efter ett tag. Vinga System använder hela tiden dessa konventioner i sin programmering oavsett om koden skall återanvändas igen eller inte. Vi tycker att detta är ett helt riktigt sätt att arbeta på. Nu när förkortningarna är välkända ger de fördelen att andra programmerare lättare kan läsa vår kod och tvärtom.

5.1.6 NAMNGIVNING FUNKTIONER

Vi utförde de matematiska beräkningarna på det ekonomiska optimeringsproblemet utifrån en algoritm som vi specificerat på papper tillsammans med Vinga System. Vi märkte då att anropen för beräkningarna blev mycket långa och att vi blev tvungna att använda många temporära variabler. Det var därmed lätt att göra fel för det blev rörigt. Vi omarbetade därför funktionsanropen så att de skulle bli så korta som möjligt och uttrycken så långt som möjligt liknade de som ställdes upp i matematiska uttryck i vanlig framställning. Arbetet med att få uttrycken att likna kravspecifikationen förenklades mycket med hjälp av operatorer för att använda de vanliga matematiska symbolerna som +,*. Dessutom skulle funktionsnamnen vara logiska och lätta att förstå.

Ett exempel från vår kod är utförandet av en matrisinvertering. I kravspecifikationen användes följande uttryck för att visa att en matris skall inverteras:

Kravspecifikation: $\mathbf{B}^{-1}$

Vi använde oss av funktionsnamnet ”Invert” vilket är lätt att förstå vad denna utför samt lägger ett i på variabelnamnet för att visa att denna inverteras.

Källkod: **Bi.Invert();**

Ett annat mindre exempel på en operation är följande matrisberäkning av K

Kravspecifikation: $\mathbf{K} = \mathbf{R}\mathbf{B}^{-1}\mathbf{R}$

När vi översatt problemet till vår källkod blev uttrycket följande vilket är ganska likt och lättförståeligt:

Källkod: **K=R*Bi.Invert()*R**

Följande exempel kan tyckas enkla och att det är onödigt att lägga ned mycket tid på att skapa funktioner som ser ut och uppträder precis som önskas. Vid lite större uttryck som exemplet från vår kravspecifikation nedan underlättar det en hel del med logiska uttryck.

$$\boldsymbol{f}(n+1) = -0.5\mathbf{z}'(n+1)\mathbf{R}\mathbf{B}^{-1}\mathbf{R}'\mathbf{z}(n+1) - \mathbf{z}'(n+1)\mathbf{R}\mathbf{B}^{-1}\mathbf{A}'\mathbf{y}(n+1) - 0.5\mathbf{y}'(n+1)\mathbf{A}\mathbf{B}^{-1}\mathbf{A}'\mathbf{y}(n+1)$$
$$-\mathbf{z}'(n+1)\mathbf{c} - \mathbf{y}'(n+1)\mathbf{d}$$

5.1.7 FÖRDELNING HEADER/SOURCE-FIL

Även om det inte är nödvändigt är det för tydlighetens skull bra att dela upp sina klasser i två filer, Header och Source. Genom Headerfilen går det att snabbt och enkelt att se vilka funktioner som finns tillgängliga och vilka parametrar de kräver samt vad som returneras. De små funktionerna är också lättast att överblicka om de skrivs direkt i headerfilen. Dessutom lärde vi oss att det är mer effektivt om en funktion som anropas ligger i en headerfil.

5.1.8 DOKUMENTATION

När vi började med matrisklassen var det ganska svårt att komma igång med programmeringen. Anledningen till detta var att komponenten valarray och dess funktioner inte var så väl dokumenterade ännu eftersom de var såpass nya. För oss som inte var så vana vid programmering ännu var det därför svårt att förstå hur kommunikationen med valarrayen skulle gå till. Det hade underlättat mycket för oss om det funnits en enkel beskrivning över vad funktionerna utför och hur kommunikationen med dessa skall gå till samt något konkret exempel. Detta är bra att tänka på när det egna komponenter konstrueras som är tänkta att återanvändas men också när det inför ett projekt bestäms vilka externa komponenter som skall ingå att inte ta det senaste utan sådant som är mer dokumenterat.

5.1.9 FÖRSLAG TILL HUR ÅTERANVÄNDNINGSBAR KOD SKALL SKRIVAS

Vi kommer här att beskriva hur vi anser att kod som skall återanvändas bör konstrueras.

När det gäller syntaxen för koden så bör den följa någon slags namnkonvention. Ofta finns en sådan på företaget och om det saknas kan standarder som t.ex. Hungarian kodkonvention användas. Detta gör koden mer lättförståelig och användbar. Källkoden bör kommenteras i headerfilen där det beskrivs vad klassen utför. Korta kommentarer bör också finnas till varje funktion där det förklaras vad funktionen utför.

Att paketera kod som skall återanvändas som hela komponenter eller moduler är klart effektivast. De kan då återanvändas i den form som de är och kräver inte någon anpassning för att fungera i det nya projektet. Det är också lätt att koppla på nya

funktioner. Så länge inte interfacet ändras för de andra funktionerna så påverkas inte deras prestanda eller funktionalitet. Om utvecklingen sker i Windowsmiljö så bör koden paketeras som DLL-filer eller COM-objekt. Fördelen med detta är att om komponenten är skapade i C++ så kan den även användas i andra programmeringsspråk som t.ex. Java eller Visual Basic. Möjligheterna att återanvända koden ökar.

Testning av dessa komponenter är mycket viktigt. Vi rekommenderar att ett särskilt testprogram skrivs till komponenten för att säkerställa att den fungerar på ett tillfredsställande sätt.

När det gäller dokumentation av källkod behöver den inte vara speciellt omfattande om koden följer de riktlinjer som vi angivit ovan. När det gäller komponenter som t.ex. COM-objekt så bör dessa dokumenteras lite mer noggrant. Det finns ingen möjlighet att gå in i koden för att se vad som utförs utan detta bör redovisas i ett separat dokument.

5.2 INTERVJURESULTAT

Intervjumallen nedan användes mera som en diskussionsmall än ett frågeformulär. Vi utförde intervjuerna med öppna frågor, dvs att intervjupersonerna fick en fråga och de fick svara fritt utan några svarsalternativ. Intervjuerna utfördes i följande ordning WM-Data, EHPT, Vinga System och Caesar. En del frågor kunde inte ställas till alla företag. Detta berodde på att företaget inte använde sig av de metoder som frågorna berörde eller att intervjuperson saknade kännedon om dessa.

5.2.1 INTERVJUMALL

Förutsättningar för intervju:
Företaget återanvänder kod enligt vår definition.

Grundläggande frågor:
Lite information om personen och företaget.

Återanvända:

Standardkomponenter
Använder ni i er programmering av:

- RAD-wizards
- MFC, API och DLL
- Kompilatorhjälp
- Företagets tidigare producerade kod
- Köpa in färdig kod

Under varje kategori skall frågas
Vilka områden återanvänds kod från denna kategorin t.ex. gränssnitt?
Hur hittar Ni de funktioner eller klasser som ni vill återanvända?
Ju bättre/mer erfaren du blir med utveckling använder du mer/mindre dessa hjälpbibliotek?
Fördelar/Nackdelar?

Andra områden
Finns det andra källor till återanvändning av kod?, Vad för typ av återanvändning?
Inom vilka områden, som exempelvis viss typ av funktioner återanvänder ni inte kod?

Bygga återanvändbart
Övergripande
Rutiner företaget några övergripande riktlinjer för återanvändning av kod?
Har ni mått för att undersöka graden av återanvändning?
Har ni standardiserat utvecklingsmiljöer osv. För att underlätta återanvändning.

Kodanpassningar
Har ni några riktlinjer inom företaget för hur detta skall gå till som t.ex. Hungarian Coding conventions för Namngeneralisering?
Utför ni mer och annorlunda testning och verifiering av kod tänkt för återanvändning?

Övriga anpassningar
Hur katalogiserar ni koden för att hitta den igen?
Hur dokumenterar ni kod som är tänkt att återanvändas?

Paketering
I vilken form sparas den kod som skall göras återanvändbar?
TYP: DLL Vanliga klassfiler?

Allmänt (om ej framgått):
Vilka stora fördelar ser ni med kodåteranvändning?
Vilka stora nackdelar ser ni med kodåteranvändning?
Beräknar ni lönsamhet innan ni programmerar kod återanvändbar/kostnadsberäkningar?

Användningmönster
Vid konsultjobb. Vem äger koden ni eller hindras återanvändning?

5.2.2 ALLMÄNT INTERVJUFÖRETAG

WM-data
Intervjun gjordes med Kenneth Anntila som har gått på Systemvetarprogrammet i Göteborg där han tog examen 1994. Han arbetar idag som teknisk projektledare.

Han arbetar på WM-data som är ett av Sveriges största IT-företag. WM-data är verksamma inom en rad olika områden. En del av bolagen i koncernen är inriktade mot branschspecifika områden där de inte bara har kompetens om IT utan även kunskap om miljön som deras kunder verkar i. Kenneth arbetar i ett bolag som kallas WM-data Forest som enbart sysslar med applikationer för skogs- och pappersindustrin. Detta kan dock handla om allt från ekonomisystem till produktionssystem och lagersystem. Vid denna avdelning i Göteborg jobbar ca 20-25 personer.

EHPT
På EHPT intervjuades Håkan Amarén som är chef för en av designavdelningarna på företaget. Denna avdelning hade ungefär 20 st anställda. Håkan är utbildad till civilingenjör. Vilket också 70% av alla anställda på avdelningen är. Företaget producerar datorsystem som de säljer till en mängd olika kunder globalt inom telekommunikationsbranschen. Avdelningen designar, utvecklar och testar moduler som efter avslutat testning går vidare till en systemintegrationavdelningen där de olika modulerna sätts samman till ett färdigt system.

Avdelningen arbetar i olika UNIX miljöer samt Windows NT. De programspråk som används är C++ och Java.

Vinga System
Intervjun på Vinga System gjordes med Thomas Olsson. som gått Datateknisk linje på Chalmers 1988-1992. Han började på ett sommarjobb hos Vinga System 1986 och arbetade sedan heltid ett år och fortsatte arbetet under studietiden. Har arbetat med C men i stort sett bara med C++ sedan det kom.

Caesar affärssystem
Intervjun utfördes med Patrik Pettersson som arbetar som programutvecklare på Caesar Affärssystem AB. Patrik har bakom sig Datateknisk utbildning på Chalmers.

Caesar som är ett dotterbolag till Linnédata utvecklar säljstödsystem. Det är en databas med kunduppgifter adresser, telefonnummer vad som sades och gjordes vid senaste kontakten. Detta för att en annan säljare skall kunna ta hand om kunden nästa gång. Det är Windowsmiljö, klient server baserat med en stor databas i mitten och klienter som visar menyer och dialoger. Klienten kan även vara mobil och synkroniseras med databasen när klienterna kommer in.

Caesars målplattformar är Windows 95/98/NT och utvecklingsverktygen är Microsoft Visual C++ och Visual Basic. De har även börjat med Interdev som är Internetbaserat.

5.2.3 FÖRDELAR

WM-data
Tidsvinsten som erhålls vid återanvändning genom att ärva egenskaper från färdiga klasser, jämfört med om all kod skulle skrivas från botten är mycket stor. Oerhört bra, det går mycket snabbt att komma igång med att skriva en applikation jämfört med vanlig C++ kod.

EHPT
I de fall som det går att återanvända kod eller allra helst färdiga komponenter så är det att föredra, det är onödigt att uppfinna hjulet på nytt. Återanvändning minskar kostnaderna vid systemutvecklingen.

Vinga System
Det sparar både tid och pengar. Kod som återanvänds är mer utförligt testad och är genom detta mer säker. Att använda standardkomponenter som är utvecklade av t.ex. Microsoft har flera fördelar. Även dessa är mer utförligt testade och innehåller färre fel. Att utveckla gränssnitt med hjälp av MFC ger stora tidsvinster jämfört med traditionell programmering. Återanvändning tar också tillvara den kompetens som finns i företaget i form av tidigare producerad kod.

Caesar
Den största fördelen enligt Patrik Pettersson är att det blir effektivitetsvinster när kod återanvänds och detta är huvudanledningen till varför detta utnyttjas. De anställda upplever det också som roligare att återanvända vanliga delar istället för att göra samma programdelar om och om igen. Exempel han gav var att rita upp gränssnitt istället för att sitta och justera ”pixlar”.

5.2.4 NACKDELAR

WM-data
Det blir svårt att överblicka när man ärvt i många led. Att veta i vilket av leden ovanför som en viss funktion finns. Därför skulle det behövas mycket mer noggrann dokumentation om hur koden är skriven i de lägre nivåerna.

Det är svårt att få bra prestanda särskilt om man programmerar med dålig struktur. Detta kan ofta inträffa eftersom många klasser och funktioner ärver egenskaper ifrån klasser uppifrån och denna superklass kan då blir initierad flera gånger.

EHPT
Att återanvända kod inom företaget innebär vissa problem. Om en klass återanvänds från en annan avdelning som sedan upptäcker ett fel i klassen bör detta naturligtvis åtgärdas. Den som tillverkade klassen från början kan dock undertiden utvecklat denna mycket åt ett annat håll och har därför inget intresse av att gå in och korrigera i gamla versioner av den klassen.

Vinga System
Ibland kan det vara mycket svårt att sätta sig in i någon annans kod. Det kan i vissa fall krävas så omfattande anpassningar av koden som återanvänds att kostnaden blir högre än att ha skrivit ny kod.

Caesar
Om något annat än kompilatorklasser och operativsystemets kod återanvänds är det problem med att veta hur det ligger till med pålitlighet och dokumentation. Svårigheter att hitta och anpassa en återanvändningsbar komponent kan ofta ta nästan lika lång tid som att bygga en själv.

5.2.5 HITTA ÅTERANVÄNDBARA KOMPONENTER

Vinga System
Vinga System utvecklar i Microsoft Visual C++ och här finns mycket hjälp. Dokumentation finns för alla funktioner och klasser, även operativsystemets funktioner finns också dokumenterade. Thomas tittar mycket i dessa hjälpfiler, det finns också många exempel som är till stor nytta. Det är med hjälp av dessa hjälpfiler som de klasser och funktioner som skall användas hittas.

Thomas använder inte bara kompilatorns hjälpfiler utan även MSDN - Microsoft Developer Network - detta är en produkt som köps in från Microsoft. Genom denna erhålls varje månad ett flertal CD-skivor med program och dokumentation. Eftersom Vinga System i stort sett enbart utvecklar system för Windows miljö så kan dessa hjälpfiler användas, eftersom hänsyn ej behöver tas till att programmen skall vara plattformsoberoende.

Caesar
All källkod ligger i ett Sourcesafearkiv, ett versionshanteringssystem för källkod där koden checkas in och ut. Programmet håller sedan kontroll på de olika versionsnumren. Det ser ut som Explorer i Windows med en trädstruktur och här finns en mapp som heter delad källkod. Här finns den kod som finns för återanvändning. Denna mapp används dock mycket dåligt, istället går det mest på hörsägen. Det finns också ett motstånd till återanvändning av någon annans kod, programmeraren vill skapa sin egen kod.

När det gäller MFC och ATL så är dessa mycket väl dokumenterade och det är därför lättare att förmå programmeraren att använda sig av dessa. Den egna koden är dock nästan inte dokumenterad överhuvudtaget.

För att hitta i MFC används den hjälp som finns i Developer Studio. Markören ställs över en klass eller funktion sedan trycks F1 och då kommer informationen fram om den aktuella klassen fram. Detta är ett fullgott hjälpmedel. Något som är ett önskemål är att detta även skulle fungera för de klasser som företaget själva skapat.

5.2.6 DLL-FILER & COM-OBJEKT

WM-data
Allting som görs paketeras i DLL-filer. Basklassen har en EXE-fil som sedan anropar DLL-filerna. I det projekt som Kenneth deltar i nu ligger 850-900 DLL-filer. Fördelningen är att ca 15% ligger i basklassen, 10% i plattformsklassen, 50% i funktionsklassen och resterande 25% i anpassningsklassen.

Forest har ansvaret för de små delarna i funktionsklassen (runt 10%) och praktiskt taget hela anpassningsklassen. I de delar som Forest ansvarar för består 50% av nyutvecklade och 50% av återanvända funktioner.

Vinga System
Vinga System har på senare tid börjat bygga dessa som COM-objekt komponenter, när dessa sedan återanvänds så är det hela komponenter som används och inte källkoden. Fördelen med detta är att modulerna kan användas i t.ex. Visual Basic- och Delphimiljö. Dessa moduler skapas i C++ och bygger på ATL. Alla nya användarmoduler byggs på detta sätt.

Företaget återanvänder egen kod inom bl a databashantering, kommunikation mellan server och klienter, användarinterface komponenter och hjälp klasser. För att hitta dessa klasser används ett program Sourcesafe för att hålla reda på källkoden. Det saknas en central plats för att samla all dokumentation detta är dock under uppbyggnad. Vinga har mellan 50 och 100 klasser.

Caesar
Källkod sparas som kod och utifrån denna genereras DLL-filer och COM-objekt. Istället för skapa C++ klasser så har Caesar mer och mer gått över till att skapa COM-objekt. Detta beror på att COM-objekten har flera stora fördelar gentemot vanliga C++ klasser. Det kan anropas från andra programmeringsspråk bl.a. Java och Visual Basic. Kommunikationen kan ske via nätet, den är självbeskrivande i form av typbibliotek.

Om det behövs en klass för att hantera loggningsinformation i databasen så utvecklas först en C++ kod. Sedan levereras den i form av ett COM-objekt och den är då färdig för användning.

Själva konceptet med COM-objekt bygger på återanvändning, objekten är till för att användas igen. Det är mycket lätt att lägga in dem i ett nytt projekt. Ett COM-objekt kan ses som en boll med ett antal gränssnitt och utanpå detta kan fler gränssnitt byggas på.

När en klient använder objektet så frågar den efter just det gränssnittet som behövs. När nya gränssnitt läggs in så påverkas inte gamla klienter av detta.

En ytterligare stor fördel med användandet av COM-objekt är att när ytterligare gränssnitt och funktioner läggs till så måste inte alla applikationer kompileras vilket hade varit fallet vid användandet av vanliga C++ klasser.

Mycket av utvecklingen görs i ATL även här finns mycket bra wizards som används. Alla COM-objekt utvecklas med hjälp av ATL. Det är till delar djupt inne i systemen som ATL används och wizards ger även här en mycket stor tidsvinst.

5.2.7 MFC

Vinga System
MFC är också något som Vinga System använder sig mycket av vid utvecklingen av system. MFC är bibliotek som används för att skriva vettiga C++ program mot Windows. En stor fördel med att använda MFC är att Microsoft utvecklat det vilket medför att det finns många användare. Detta innebär att de flesta buggar och fel är upptäckta och de är väl dokumenterade, det finns en också en mängd böcker. Det kanske inte är det bästa klassbiblioteket men det stora användartalet gör det mycket bra.

Det ligger också som ett tunt skal ovanpå Windows, det försöker inte dölja Windows funktionalitet. MFC innehåller inte så mycket utan kopplar istället vidare till Windows funktioner vilket innebär att om utvecklaren kan Windows så förstår han även MFC ganska väl. Att MFC är tunt innebär även att det är lätt att hitta fel.

Nackdelen med MFC är att det inte är plattformsoberoende, det tunna lagret i MFC gör också att man får leva med de brister som finns i Windows. Det kan också vara svårt för personer som inte kan Windows att förstå MFC.

Vinga System har även skapat ett eget gränssnittsbibliotek, som ärver egenskaper från MFC och ATL. I dessa finns ytterligare funktioner och egenskaper, detta för att man vill göra mer avancerade funktioner som inte MFC innehåller.

MFC innehåller även en dokumentvyhantering som Vinga System använder. Denna innebär att man kan få en koppling mellan filer på hårddisken och programmet. Dokumenthantering är den enda funktion som MFC innehåller som är MFCs egna funktion och inte en mappning av Windows funktioner. Vidare innehåller MFC en del mängdklasser eller containerklasser och stränghantering, dessa försöker Vinga System undvika att använda istället används nu det nya STL- Standard Template Library - detta är mer portabelt och generellt. STL är mer kraftfullt än MFC, STL är dock nytt och dokumentationen är inte så väl utbyggd. Microsoft har köpt in ett färdigt källkodsbibliotek och använder denna dokumentationen så den följer inte Microsoft standard på dokumentation så det kan vara svårt att känna igen sig.

Caesar
Klienterna och användargränssnitt byggs upp med hjälp av MFC. Egna klasser har byggts upp som kapslar in MFC:s egenskaper. Bl.a. har en listview skapats med en struktur liknande den som finns i Utforskaren i Windows. Detta för att MFC låg på en allt för låg nivå. Dessa klasser delas mellan produkterna i källkodsform. Det finns dock en fara i att dela klasser i källkodsform. En programmerare kan ändra i en klass och omedvetet förstöra för ett annat projekt.

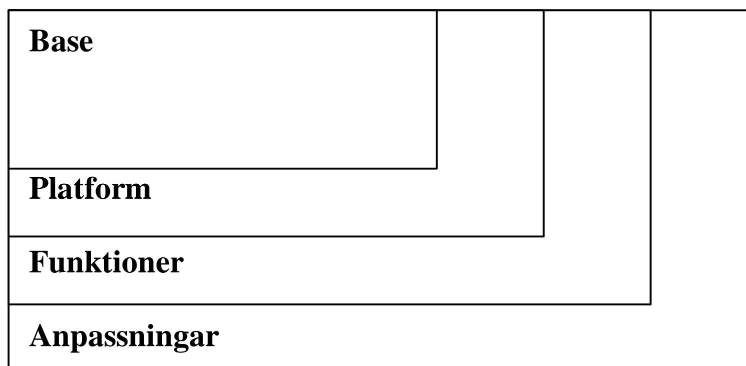
MFC används också eftersom det är speciellt när det gäller användargränssnitt, däremot är databashanteringen allt för dålig. Gränssnittet bygger på Windowsstandard, när Microsoft kommer med något nytt så följer Caesar denna standard. En stor fördel med MFC är att det är väl integrerat med Developer Studio. Här finns verktyg som gör kod som passar för MFC. Denna synergikoppling gör MFC till ett mycket bra hjälpmedel. Om utvecklingen görs i C++ och det är användargränssnnitt som utvecklas så finns det ingen anledning att inte använda MFC.

När MFC används så ärvs egenskaper nästan alltid och en ny modifierad klass skapas utifrån MFC. Ofta finns källkoden i MFC och det enda som saknas är vissa specifika egenskaper. Därför är lämpligt att utgå från denna klass och sedan genom arv lägga till ytterligare funktionalitet.

De nackdelar som finns med MFC är att det börjar bli lite gammalt, det går bl.a. inte att göra några Webapplikationer för då blir programmet allt för stort.

5.2.8 ANDRA ÅTERVINNINGSÄTT

WM-data
Inom detta företag återanvänds kod mycket omfattande. Företaget utvecklar en egen produkt som säljs till en mängd olika kunder och här återanvänds till största del själva stommen. Sedan görs endast mindre förändringar på produkten för den specifika kunden.



Figur 9. WM-data Forest applikationsstruktur

Ett större klassystem har utvecklats av en finsk partner som är dotterbolag till företaget Tieto(WM-data ansvarar för Sverige och Norge medan Tieto ansvarar för övriga världen när det gäller att förse skogsindustrin med lösningar). Hur denna klasshierarki är uppbyggd kan ses i figur 9. I botten ligger basklasser där all grundläggande funktionalitet för en applikation ligger. Ovanpå den ligger plattformsklasser där innehållet i dessa är beroende på vilken av plattformarna, PC eller UNIX, som applikation används mot. I plattformen ligger därför mycket användargränssnitt.

I funktionerna ligger affärslogiken. Vissa av dessa klasser kommer från finska partnern eller andra företag inom WM-data men här ligger också en hel del som WM-data Forest har utvecklat själva. Det kommer ingen affärslogik direkt från externa leverantörer och det beror på att det inte finns några externa komponenter som är användbara. Det har bara hänt vid något enskilt tillfälle att det lagts till några ytterligare komponenter som någon annan leverantör skapat, Crystal Reports. I anpassningar görs mindre förändringar i t.ex. affärslogik eller användargränssnitt. Dessa klasser är till allra största delen utvecklade av Forest.

Det som återanvänds är det mesta som ingår i vanliga program. Det behövs det inte tänkas på konstruktorer/destruktorer, databasaccesser osv. En kodgenerator genererar en basuppsättning för ett nytt objekt sedan läggs de specifika delarna till

Det svåra med objektorientering och återanvändning av funktioner är att veta vilket objekt som skall göra en viss sak och se till så att en viss klass alltid gör samma sak. Dessutom att bryta ned systemet till en lagom nivå så att det inte blir för många klasser och överskådligheten går förlorad.

För de klasser inom funktions- och anpassningsklasser som Forest själva utvecklat finns det en person som är ansvarig för varje klass. De är grupperade så samma person är ansvarig för alla orderklasser osv. Detta gör att det byggs upp kompetens så att det är lätt att få information om vilka klasser som finns inom de olika områdena.

Det finns ett mål i företaget att kunna sälja komponenter med olika innehåll, t.ex. produktionsplanering, produktionsuppföljning och orderhantering med standardiserade gränssnitt så att den sista anpassningen eller koden mellan komponenterna blir så liten som möjligt.

EHPT

På EHPT används dock ej så mycket återanvändning av kod. Detta beror främst på att arten av programmering är annorlunda på detta företag än de tidigare vi intervjuat. EHPT håller på med teknisk programmering inom en speciell nisch där utvecklingen går mycket fort. Kunderna kräver att de för de mesta använder den allra senaste tekniken. Detta gör att det inte finns så mycket att återanvända. Som exempel på detta var EHPT ett av de första företagen att börja utveckla i Java, detta gjordes redan 1995.

Att de också utvecklar mot flera olika plattformar gör att de hela tiden endast vill använda sig av generella standardkomponenter i sitt utveckling. Detta resulterar i att

MFC eller andra programtillägg som låser företaget vid en viss teknisk plattform eller programleverantör är omöjliga att använda.

Dock återanvänds en del kod inom avdelningarna på modulnivå inom applikationerna. Det förekommer även återanvändning av färdiga klasser och komponenter utifrån. Om det t ex finns ett färdigt standardbibliotek eller klass för det som skall byggas som är generell samt plattformsoberoende så köps dessa. Idéer och färdiga produkter är något som återanvänds.

Vinga System
Vinga System köper i stort sätt aldrig in färdiga komponenter eller kod som någon annan har producerat. Det enda är MFC och C++ standardbibliotek. Detta beror på att de gånger detta gjorts har det aldrig passat ihop med deras egna delar. Det har nästan alltid slutat med att de skapat en egen komponent. Vinga System har speciella krav på bl.a. användarinterface komponenter som inte tillfredsställs i de färdiga komponenterna. Bl.a. arbetar de med realtidssystem, ett exempel är en gridlayout där rutor ändras under tiden som användaren arbetar med programmet, detta finns inte någon annanstans.

Två varianter för paketering av egna moduler används. Det ena är källkod och det den andra är som COM-objekt och Active X komponenter som vi beskrivit ovan. När det är större komponenter som t ex användargränssnittmoduler så används alltid COM-objekt.

Caesar
I ett projekt har färdig kod köpts in från ett annat företag. Användargränssnitt klasser köptes in från Stingray. Demoversioner köps även in i form av plugginversioner som dör efter 30 dagar. En del komponenter för hjälphantering har också inhandlats av andra företag.

0Det finns dock en del problem med att köpa in kod eller program. Den inköpta komponenten kan tvinga Caesar till en viss kodstil eller ett sätt att bygga applikationens arkitektur som inte var tänkt från början. Detta måste göras för att integrera den inköpta produkten. De komponenter som köpts in har ej heller varit särskilt stabila vilket skapat en hel del problem.

Caesar utvecklar även system i en mängd olika språk t.ex. svenska, engelska, danska och tyska. De komponenter som köps in är alltid på engelska och det är ett mycket tidskrävande samt svårt arbete att översätta dessa. Ibland är det inte ens tillåtet att översätta komponenterna.

5.2.9 ALLMÄNNA RIKTLINJER

WM-data
Det finns mycket strikta regler för hur man tilldelar namn på klasser, attribut och variabler. Det eftersträvas ett liknande utseende på koden som produceras ända från

basklassen till anpassningsklassen. Det underlättar mycket att bara namnsätta riktigt så det går att särskilja medlemsvariabler och lokala variabler.

EHPT
Riktlinjer för hur kod skall skrivas saknas. Tidigare fanns ett stort dokument med regler men detta efterföljdes inte så det togs bort. De riktlinjer som finns i dagsläget gäller namnsättning vid skapandet av systemgränssnitt mellan olika programdelar och moduler. Detta finns för att undvika namnkonflikter och underlätta sammansättandet av de färdiga delarna. Detta görs också för att öka flexibiliteten i programmen. Då kan hela moduler återanvändas från ett visst program när ett system med liknande funktioner skall skapas.

Vinga System
Vinga System har ett dokument med riktlinjer för hur bl.a. namngivning av funktioner, klasser och variabler. Riktlinjerna är ganska vaga, istället finns en intern jargong där man tittar hur andra har skrivit tidigare. Anledning till att de inte har några mer klara riktlinjer beror bl.a. att det är svårt att hålla dessa aktuella eftersom allting förändras så skulle det behöva uppdateras minst vartannat år.

Caesar
Företaget har inte någon policy för återanvändning av kod. Regler för hur namngiving av klasser, funktioner och variabler skall namnges finns samlat i dokument på 5 sidor. Källkoden skall vara lättläst och parametrar och variabler ha liknande namn. Detta dokument följer Hungarian Namn konvention som tidigare beskrivits.

5.2.10 TESTNING

WM-data
Inget skrivet testförfarande finns. Det har man däremot på Keracom i Finland. När saker upptäcks som skall ändras i basklassen eller i plattformsklassen är det möjligt för Forest att meddela Keracom för att få till en ändring. I praktiken har dock ej detta hänt hittills utan ändringen har utförs på Forest.

Innan dessa basklasser byggdes upp hade program som utvecklades i Göteborgs orderhanteringssystem och i Faluns truckhantering mycket olika utseende. Med hjälp av basklasserna erhålls nu ett mer homogent programpaket. Därför är det mycket lättare att utbyta information och kunskap mellan Forest avdelningarna inom företaget.

Vinga System
Strikta riktlinjer för testning saknas också utan även här. Det finns några enkla riktlinjer bl.a. skall all kod kontrolleras i debuggern. Komponenten skall brytas ner i mindre delar och dessa skall testas isolerat. För alla generella klasser byggs ett testprogram när sedan ändringar görs i klassbiblioteket så kan man testa att även det generella programmet fortfarande fungerar.

Funktioner och moduler som skall återanvändas testas mer noggrant. Kod som skapats av en person lämnas vidare till en annan som går igenom källkoden och letar efter fel som kan vara svåra att upptäcka för personen som konstruerat koden. Testversioner släpps också ut till kunderna. Detta är också ett sätt att testa programmen innan de släpps som färdiga produkter.

Caesar
Klasser som sparas för återanvändning testas inte speciellt noggrant, i en del extremfall där klasserna är väldigt viktiga så har ett speciellt testprogram tagits fram. Programmeraren som har skrivit koden har ansvar för att denna fungerar på riktigt sätt. Patrik anser dock att testning är något som är mycket viktigt men det är en resursfråga hur mycket koden skall testas.

5.2.11 DOKUMENTATION

WM-data
När det gäller dokumentationen av koden hos WM-data är basklassen som används ofta väl dokumenterad. Plattformsklasserna är också ganska väl dokumenterade medan det blir allt sämre i funktions- och anpassningsklasserna. Detta är en resursfråga, det kostar pengar att upprätta hålla en bra dokumentation av all kod som produceras. Mest resurser läggs givetvis på den kod som återanvänds flest gånger. Det lönar sig inte att fullständigt dokumentera alla anpassningsklasserna.

EHPT
Eftersom EHPT är ett stort företag och många anställda kan komma att jobba med samma kod och som kanske också används av flera avdelningar så är det mycket viktigt att koden dokumenteras noggrant. På EHPT finns också ett antal olika avdelningar och det är vanligt att de anställda flyttar till andra avdelningar. Om den kod som den person som lämnat avdelningen inte är väl dokumenterad så försvinner kunskapen om hur koden fungerar och används.

Vinga System
Koden är ganska sparsamt dokumenterad, den dokumentation som finns är kommentarer i källkod och headerfiler. Där finns det kort beskrivet vad de olika funktionerna utför och vilka klasser som används.

När det gäller särskilda komponenter som t.ex. COM-objekt och ActiveX så dokumenteras dessa separat på intranet . Detta görs eftersom vid användning av dessa finns inte källkoden tillgänglig när de används.

Thomas anser att väl skriven kod skall inte behöva någon ytterligare dokumentation. Om koden skrivs tydligt och följer de riktlinjer samt den namngivningsstandard som finns dokumenterad på intranet räcker det.

Det finns även lite dokumentation från designfasen för en del klasser. Denna dokumentation består av klassdiagram mm uppritade efter UML tekniken. Detta ger en överblick hur klasserna hänger ihop, något som det annars kan vara svårt att få en överblick av.

Caesar
Koden skall vara skriven så att den inte kräver någon direkt dokumentation. I funktionshuvudet skrivs kort vad funktionen gör och vilka variabler som används. Mer komplicerade delar av koden kommenteras direkt i själva källkoden.

Test av verktyg som automatiskt generar dokumentation av koden i form av HTML-filer har testas och det har upplevts som ett bra sätt att dokumentera koden. Det tar alltför lång tid att dokumentera koden mer utförligt än på detta sätt. Kostnaden blir alltför hög gentemot den nytta som dokumentationen ger.

Dokumentation för färdiga klasser som används t.ex. DLL-filer finns i form av ett väl dokumenterat gränssnitt som ser ut som en C++ headerfil. I detta finns kommentarer för parametrar och funktioner.

5.2.12 ANDRA FRÅGOR

WM-data
En enskild anställds produktivitet mäts inte utan det är hela projektet som bedöms. Eftersom den som skapat en klass också är ansvarig för den så motiveras de anställda att skriva tydlig och återanvändbar kod. De slipper då göra om samma moment flera gånger.

WM-data äger all kod. Kunden köper en produkt och därmed kan all kod som produceras för en applikation återanvändas i andra applikationer eller i en liknande applikation för ett konkurrerande bolag inom branschen.

Eftersom samma bas används är det också lättare att utnyttja personalen effektivt och när det behövs mer personal i ett projekt kan anställda snabbt sätta sig in i dessa arbetsuppgifter.

EHPT
Kraven på de programutvecklingsmiljöer som företaget använder sig av är att de skall bygga på standardlösningar. Kunderna kräver att systemen är oberoende av någon speciell teknisk lösning. Därför är det viktigt att använda sig av standarder för programspråk, SQL osv och ingen specifik databasdialekt för exempelvis Oracle.

Något som Håkan förordade var att bygga mer flexibla system. Ett sätt att göra detta är exempelvis genom att i vissa delar använda CORBA*. Detta möjliggör att olika programdelar skrivna i olika programspråk kan kommunicera med varandra.

Vinga System
När det gäller ägandet och nyttjanderätt till kod så finns det flera olika varianter hos Vinga System. Det finns fall där företaget utvecklar program och också äger koden. I en del fall så äger Vinga System koden men där även kunden äger en kopia av källkoden. Sedan finns det fall där kunden själv äger koden. Detta innebär vissa begränsningar för vilken kod som kan återanvändas.

Caesar
Caesar utför inga konsultjobb utan säljer färdiga program. Detta innebär att de äger all kod själva och kan därför använda den för återanvändning efter önskemål.

6. DISKUSSION OCH SLUTSATS

Ett av huvudsyftena med uppsatsen var att vi skulle erhålla kunskap inom framförallt två områden – skriva kod med hjälp av utvecklingsverktyg som använder sig av återanvändning av kod s.k. RAD-verktyg och att utveckla matematiskt avancerade program. Denna uppgift tycker vi att vi har lyckats med på ett tillfredsställande sätt. Den erfarenhet och kunskap vi fått under arbetet med denna magisteruppsats kommer vi att ha stor nytta av i vårt framtida arbetsliv. Det har ökat våra chanser att erhålla den anställning hos de företag som vi önskar.

Våra kunskaper inom ämnet objektorienterad systemutveckling har ökat markant. Detta är ett område som breder ut sig alltmer inom IT-branschen. Vi har också fått en inblick i arbetslivet, hur olika företag arbetar, vilka arbetsuppgifter som finns och vilka kunskapskrav som ställs på oss när vi kommer ut i arbetslivet. Vi har lärt oss hur matematiska algoritmer skall formuleras för att på ett enkelt sätt kunna realiseras som programkod.

I vilka situationer i programutvecklingen är det lämpligt att utnyttja återanvändning av kod?

På EHPT ansågs det att återvändandet av kod främst lämpar sig bäst när det gäller administrativa program. Detta beror på att applikationer inom denna genre till stora delar innehåller liknande funktioner. Möjligheterna för återanvändning är mycket mindre inom teknisk programmering. Om företaget har som konkurrensfördel att använda sig av den absolut senaste tekniken kan detta också försvåra återvändandet av kod.

De övriga intervjuerna och litteraturen ger också bilden av att det underlättar mycket om utvecklingen sker i en homogen miljö. Vanliga orsaker till varför inte företag återanvänder kod eller använder sig mer av Wizards är att koden då inte blir plattformsoberoende.

Konstruktion av användargränssnitt är ett av de vanligaste områden där återanvändning av kod förekommer. Systemgränssnitt är också ett område inom vilka de flesta applikationer har ett likartat utseende och standarder har utvecklats. Därför underlättas också återanvändningen på detta område.

Litteraturen och intervjuerna pekar också mot att större enheter i framtiden kommer att paketeras till moduler. Detta skapar möjligheten att sätta ihop komponenter ifrån flera olika miljöer. Detta ökar komponentens återanvändningsgrad avsevärt och det lönar sig bättre att använda återanvändning.

Vilka källor finns för återanvändning av kod vid programutveckling i Windowsmiljö?

Vi har under arbetet lärt oss att om programmeringen skall ske mot en Windows NT-plattform så finns det mycket att vinna på om kunskap finns vad det finns för resurser att tillgå i operativsystemet samt hur dessa nås.

Det finns en mängd olika funktioner nedbäddade i operativsystemet i form av DLL-filer som innehåller allt ifrån att få reda på hur mycket systemklockan är, till hur grafik ritas upp på skärmen. Dessa funktioner kan dock vara svåra att sätta sig in i. Därför finns det hjälpmedel för att underlätta anropet av dessa funktioner. I MS Visual C++ miljön, som vi och tre av våra intervjuföretag använder finns två olika sätt att utnyttja dessa funktioner. Genom ATL och MFC-klasser. Hos dessa tre företag fanns en klar trend, MFC ersattes alltmer av ATL och COM-objekt. Detta hade flera förklaringar, Caesar tyckte att MFC började bli gammalt och Vinga System attraherades av att ATL lade ett mindre skikt kod ovanpå Windows. Den främsta anledningen var ändå att COM-objekt är en större standard och att dessa lätt kan användas även i andra miljöer och mot andra programmeringsspråk.

Vinga System och Caesar använder sig allt mer av operativsystemets hjälpfunktioner och det tycks vara en allmän trend enligt den litteratur vi läst. Detta förklaras bl.a. med att dessa funktioner finns på alla datorer i Windows-miljö samt att dessa är väldokumenterat, pålitligt och standardiserat.

Vad är viktigt att tänka på när kod skapas för återanvändning?

Hur skall klasser och moduler som är tänkta för återanvändning konstrueras. Det är mycket viktigt att programutvecklaren inser att han eller hon inte kan skapa några egna standarder. Bäst är att använda de standarder som de stora mjukvaruföretagen har för att bygga komponenter. Detta underlättar för andra att använda och förstå programmet. Det blir också lättare för en utvecklare att sätta sig in i andras kod då denna är utvecklad på samma notation. Det är därför mycket bra att använda sig av en erkänd standard som t.ex. MFC när det skall konstrueras återanvändningsbar kod. Eller som Prosise(1997a) utrycker det ”good code is never accidental. A programmer who spends a lot of time with MFC picks up tricks and techniques for writing better applications.”

Vi ger i resultatdelen vår syn på hur vi anser att kod skall konstrueras. Det finns dock inte något idealrecept för hur detta skall gå till. Alla de företag som vi intervjuade hade olika synsätt på återanvändning och arbetade på olika sätt. Hur koden skall byggas och i vilken form den sparas är till viss del beroende av vilken miljö utveckling sker i. Om den skall vara plattformsoberoende, vilket programspråk som används och vilken typ av system som utvecklas.

Men de aspekter som vi tar upp i denna uppsats är en bra grund vid utvecklandet av kod för återanvändning. Genom att ta till sig dessa skapas goda möjligheter att lyckas med

uppgiften att utveckla kod som på ett effektivt och bra sätt kan återanvändas i framtida projekt. Vilket i sin tur bidrar till att sänka utvecklingskostnaderna för framtida system.

Slutsats

Vi tillsammans med de företag som vi besökt ser en stark trend att rationalisera systemutvecklingen genom att återanvända mer kod. Ökningen hos dessa företag är dock inte så mycket mer återanvändning av egen producerad kod utan mer användandet av de klasser och funktioner som finns tillgängliga i operativsystemen och utvecklingsverktygen.

Företagens egna kod som återanvänds paketeras allt mera som moduler för att minska komplexiteten vid återanvändning. Företagen och litteraturen hävdar också att trenden går emot att det blir allt större enheter som återanvänds. På sikt går systemutvecklingen mot att lägga lim (d.v.s. endast en mindre mängd kod) mellan stora komponenter.

Ett företag i undersökningen använde nästan ingen återanvändning alls p.g.a. att koden skulle användas på flera olika plattformar, alltid skulle använda sig av den senaste tekniken. Återanvändning av kod innebar också att de fick stora problem med tillförlitligheten. Detta visar några vanliga aspekter varför återanvändning ännu inte är mera utbredd systemutvecklingen.

Det finns en mängd saker att tänka på när komponenter för återanvändning skapas. Nästan samtliga dessa krav är konsistenta med de egenskaper som önskas av kod i vanliga projekt. Därför är vår åsikt att återanvändning av kod till största delen handlar om att ge programmerarna nya vanor. Avslutningsvis har vi märkt att återanvändning av kod ger en mycket bra insikt över hur effektiv och väl strukturerad programkod skall skrivas. Att återanvända kod ger därför dubbla vinster!

7. KÄLLFÖRTECKNING

Böcker:

Backman, J. (1998). *Rapporter och uppsatser.* Lund: Studentlitteratur.

Copeland T., Weston F.(1992). *Financial theory and corporate policy*. New York: Addison-Wesley Publishing Company.

Gamma, E., Helm, R., Johnson, R., Vlissides, J.(1995). Design Patterns – Elements of reusable Object-Oriented Software. New-York: Addison-Wesley.

Grimes, R., Stockton, A., Reilly, G., Templeman, J.,(1998) Beginning Atl Com Programming: Wrox Pr Inc: New York

Martin, J. (1991). *Rapid Application Development.* Mcmillan Publishing Company: New York.

McClure, C. (1997). *Software Reuse Techniques, Adding reuse to the system development process.* London: Prentice-Hall International.

Simon, R.J.(1997). *Windows Nt 4 Win32 Api Bible.* Waite Group Pr UK.

Skansholm, J. (1996). *C++ direkt.* Lund: Studentlitteratur.

Sommerville, I.(1995). *Software Engineering.* Addison-Wesley Pub Co, UK.

Stroustrup, B.(1997). *The C++ Programming Language*. Addison-Wesley Publishing Company, UK.

CD-Rom:

MSDN Library Visual Studio 6.0(1998). Seattle: Microsoft Corporation

Examensarbeten:

Duplancic, A., & Lindberg, K.(1998). *Technologies in Self Provisioning Applications.* (Magisteruppsats på Systemvetarprogrammet). Göteborgs universitet, Institutionen för Informatik.

Eriksson, M.L., & Wallström, L.(1998). Rapid Application Development – framtidens systemutvecklingsmodell? (Examensarbete inom systemvetenskap). Luleå tekniska universitet, Avdelningen för data och systemvetenskap.

Artiklar/Uppsatser:

Biddle, R., & Tempero, E. (1998). Teaching programming by teaching principles of reusability. *Information and software technology*, 40, (4), s. 203-209.

Kemerer, C. (Sep 22, 1997). Reusable asset. *Informationweek*, s. 64-66.

Lattanzi, M., & Henry, S.(1998). Software reuse using C++ classes – The question of inheritance. *The Journal of Systems and Software*, 41,(2), s. 127-132.

Orenstein, D.(1998). Code reuse: Reality doesn't match promise. *Computerworld*, 32, (34), s. 8.

Prosise, J., (1997a). Power MFC programming; *PC Magazine*, 16, (9), s. 237-240.

Prosise, J., (1997b). Building MFC extension DLLs in Visual C++. *PC Magazine*, 16, (15), s. 269-272.

Radding, A. (Mar 31, 1997). Benifits of reuse. *Informationweek*, s 1A-6A.

Radding, A. (Dec28, 1998). Rapid development for complex apps. *Informationweek*, s. 67-70.

Scheier, R, L. (1996). Reuse revealed!!!. *Computerworld*. 30, (42), s. 105-107.

Web-dokument:

Brain, M., (1998) *Introduction to MFC Programming*. http://devcentral.iftech.com/learning/tutorials/mfc-win32/vc6mfc/

FTP:

Microsoft Corporation(1996). *Microsoft Foundation Class Library Development Guidelines.* ftp://ftp.microsoft.com/Softlib/mslfiles/DEVGUID.EXE

8.BILAGOR

B.1. ORDLISTA

ActiveX Control
En programmodul som är baserad på COM. ActiveX kontroller adderar användar komponenter, funktioner och arbete som kallar komponenter som gör det "lilla extra" i program. Du kan baka in dessa i Websidor eller använda dessa som client/server applikationer som kör över ett nätverk.
Se även: COM

API(Application Program Interface)
En mängd rutiner som ett program använder för att nå lågnivå funktioner som datorns operativsystem tillhandahåller. API:erna kan bl.a. sköta om ett programs fönster ikoner, menyer och dialogfönster.

ATL (Active Template Library)
En uppsättning av små, template baserade C++ klasser som förenklar programmerandet av ett COM-objekt. ATL innehåller också mekanismen för att använda och skapa COM-obekt.

COM (Component Object Model)
En öppen arkitektur för objektorienterad utveckling mot många olika plattformar. Standarden är en överenskommelse av Digital Equipment och Microsoft. Varje COM-objekt har en basklass, **Iunknown,** från vilka alla klasser härleds.

CORBA (Common Object Request Broker Architecture)
Ett objektgrupp för att kommunicera mellan distribuerade objekt. Med CORBA kan program skrivna i alla olika språk köras överallt från ett nätverk och från valfri plattform.

Debugger
En funktion i kompilatorn som känner av, lokaliserar och rättar logiska eller syntax fel i programmets kod

DLL (Dynamic Link Library)
Tillåter dig att binda kod till en eller flera exekverbara program under drift i stället för att länka vid kompilering, genom att skapa en miljö som kan ha samma kod i basen.

Interface
Sättet med hur kommunikation med en viss programmodul fungerar.

MFC (Microsoft Foundation Classes)
En uppsättning av C++ klasser som kapslar in mycket av funktionaliteten i applikationer skriven för operativsystemet Windows. Klasserna hämtar mycket av sin funktionalitet ifrån Windows API:er.

OLE control
En custom kontroll med ett synbart interface. Detta kallas nu ActiveX

STL (Standard Template Library)
STL är ett standardbibliotek som innehåller färdiga mallar för hantering av matriser och strängar. Det finns fördefinierade funktioner för matematiska operationer som t.ex. kvadrering.

UML (Unified Modeling Language)
Ett objekt orienterat design språk. UML innehåller populära objekt orienterade metoder som en egen standard

B.2. MFC ÖVERSIKT

Microsoft Foundation Class Library Version 6.0

- **CObject**
  - **Application Architecture**
  - **CCmdTarget**
    - CWinThread
      - CWinApp
        - COleControlModule
          - user application
    - CDocTemplate
      - CSingleDocTemplate
      - CMultiDocTemplate
    - COleObjectFactory
      - COleTemplateServer
    - COleDataSource
    - COleDropSource
    - COleDropTarget
    - COleMessageFilter
    - CConnectionPoint
    - CDocument
      - COleDocument
        - COleLinkingDoc
          - COleServerDoc
            - CRichEditDoc
      - user documents
    - CDocItem
      - COleClientItem
        - COleDocObjectItem
        - CRichEditCntrItem
        - user client items
      - COleServerItem
        - CDocObjectServerItem
        - user server items
    - CDocObjectServer
    - **Window Support**
    - **CWnd**
      - **Frame Windows**
      - CFrameWnd
        - CMDIChildWnd
          - user MDI windows
        - CMDIFrameWnd
          - user MDI workspaces
        - CMiniFrameWnd
        - user SDI windows
        - COleIPFrameWnd
      - CSplitterWnd
      - **Control Bars**
      - CControlBar
        - CDialogBar
        - COleResizeBar
        - CReBar
        - CStatusBar
        - CToolBar
      - **Property Sheets**
      - CPropertySheet
        - CPropertySheetEx
      - **Dialog Boxes**
      - CDialog
        - CCommonDialog
          - CColorDialog
          - CFileDialog
          - CFindReplaceDialog
          - CFontDialog
          - COleDialog
            - COleBusyDialog
            - COleChangeIconDialog
            - COleChangeSourceDialog
            - COleConvertDialog
            - COleInsertDialog
            - COleLinksDialog
              - COleUpdateDialog
            - COlePasteSpecialDialog
            - COlePropertiesDialog
          - CPageSetupDialog
          - CPrintDialog
        - COlePropertyPage
        - CPropertyPage
          - CPropertyPageEX
        - user dialog boxes
      - **Views**
      - CView
        - CCtrlView
          - CEditView
          - CListView
          - CRichEditView
          - CTreeView
        - CScrollView
          - user scroll views
          - CFormView
            - user form views
            - CDaoRecordView
            - CHtmlView
            - COleDBRecordView
            - CRecordView
              - user record views
      - **Controls**
      - CAnimateCtrl
      - CButton
        - CBitmapButton
      - CComboBox
        - CComboBoxEx
      - CDateTimeCtrl
      - CEdit
      - CHeaderCtrl
      - CHotKeyCtrl
      - CIPAddressCtrl
      - CListBox
        - CCheckListBox
        - CDragListBox
      - CListCtrl
      - CMonthCalCtrl
      - COleControl
      - CProgressCtrl
      - CReBarCtrl
      - CRichEditCtrl
      - CScrollBar
      - CSliderCtrl
      - CSpinButtonCtrl
      - CStatic
      - CStatusBarCtrl
      - CTabCtrl
      - CToolBarCtrl
      - CToolTipCtrl
      - CTreeCtrl
  - user objects
  - **Exceptions**
  - CException
    - CArchiveException
    - CDaoException
    - CDBException
    - CFileException
    - CInternetException
    - CMemoryException
    - CNotSupportedException
    - COleException
    - COleDispatchException
    - CResourceException
    - CUserException
  - **File Services**
  - CFile
    - CMemFile
      - CSharedFile
    - COleStreamFile
      - CMonikerFile
        - CAsyncMonikerFile
          - CDataPathProperty
            - CCachedDataPathProperty
    - CSocketFile
    - CStdioFile
      - CInternetFile
        - CGopherFile
        - CHttpFile
  - CRecentFileList
  - **Graphical Drawing**
  - CDC
    - CClientDC
    - CMetaFileDC
    - CPaintDC
    - CWindowDC
  - **Control Support**
  - CDockState
  - CImageList
  - **Graphical Drawing Objects**
  - CGdiObject
    - CBitmap
    - CBrush
    - CFont
    - CPalette
    - CPen
    - CRgn
  - **Menus**
  - CMenu
  - **Command Line**
  - CCommandLineInfo
  - **ODBC Database Support**
  - CDatabase
  - CRecordset
    - user recordsets
  - CLongBinary
  - **DAO Database Support**
  - CDaoDatabase
  - CDaoQueryDef
  - CDaoRecordset
  - CDaoTableDef
  - CDaoWorkspace
  - **Synchronization**
  - CSyncObject
    - CCriticalSection
    - CEvent
    - CMutex
    - CSemaphore
  - **Windows Sockets**
  - CAsyncSocket
    - CSocket
  - **Arrays**
  - CArray (template)
  - CByteArray
  - CDWordArray
  - CObArray
  - CPtrArray
  - CStringArray
  - CUIntArray
  - CWordArray
  - arrays of user types
  - **Lists**
  - CList (template)
  - CPtrList
  - CObList
  - CStringList
  - lists of user types
  - **Maps**
  - CMap (template)
  - CMapWordToPtr
  - CMapPtrToWord
  - CMapPtrToPtr
  - CMapWordToOb
  - CMapStringToPtr
  - CMapStringToOb
  - CMapStringToString
  - maps of user types
  - **Internet Services**
  - CInternetSession
  - CInternetConnection
    - CFtpConnection
    - CGopherConnection
    - CHttpConnection
  - CFileFind
    - CFtpFileFind
    - CGopherFileFind
  - CGopherLocator

Classes Not Derived from CObject

- **Internet Server API**
  - CHtmlStream
  - CHttpFilter
  - CHttpFilterContext
  - CHttpServer
  - CHttpServerContext
- **Run-time Object Model Support**
  - CArchive
  - CDumpContext
  - CRuntimeClass
- **Simple Value Types**
  - CPoint
  - CRect
  - CSize
  - CString
  - CTime
  - CTimeSpan
- **Structures**
  - CCreateContext
  - CMemoryState
  - COleSafeArray
  - CPrintInfo
- **Support Classes**
  - CCmdUI
    - COleCmdUI
  - CDaoFieldExchange
  - CDataExchange
  - CDBVariant
  - CFieldExchange
  - COleDataObject
  - COleDispatchDriver
  - CPropExchange
  - CRectTracker
  - CWaitCursor
- **Typed Template Collections**
  - CTypedPtrArray
  - CTypedPtrList
  - CTypedPtrMap
- **OLE Type Wrappers**
  - CFontHolder
  - CPictureHolder
- **OLE Automation Types**
  - COleCurrency
  - COleDateTime
  - COleDateTimeSpan
  - COleVariant
- **Synchronization**
  - CMultiLock
  - CSingleLock

B.3. HUNGARIAN KODKONVENTIONER FÖR NAMNGIVNING

Table 3 Namnkonventioner Generella Prefix

Prefix	Type	Example
C	Class or structure	**Cdocument, CPrintInfo**
m_	Member variable	**m_pDoc, m_nCustomers**

Table 4 Namnkonventioner Variabel Prefix

Prefix	Type	Description	Example
Ch	**char**	Character	*chGrade*
B	**BOOL**	Boolean value	*bEnabled*
N	**int**	Integer (size dependent on operating system)	*nLength*
N	**UINT**	Unsigned value (size dependent on operating system)	*nLength*
W	**WORD**	16-bit unsigned value	*wPos*
L	**LONG**	32-bit signed integer	*lOffset*
Dw	**DWORD**	32-bit unsigned integer	*dwRange*
P	*****	Pointer	*pDoc*
Lp	**FAR***	Far pointer	*lpDoc*
Lpsz	**LPSTR**	32-bit pointer to character string	*lpszName*
Lpsz	**LPCSTR**	32-bit pointer to constant character string	*lpszName*
Lpsz	**LPCTSTR**	32-bit pointer to constant character string if **_UNICODE** is defined	*lpszName*
H	*handle*	Handle to Windows object	*hWnd*
Lpfn	*callback*	Far pointer to **CALLBACK** function	*lpfnAbort*

Namnkonventioner för Funktions prefix

Prefix	Beskrivning
Set	Sätt värdet av en egenskap hos ett objekt.
Get	Erhåll värdet av en egenskap hos ett objekt. Oftast 'const'.
Is	Returnerar ett booleskt värde om ett objekt har en viss egenskap.
Create	Skapar ett nytt objekt där anroparen svarar för uppresning.
Copy	Skapar en kopia av ett objekt. Anroparen har ansvar för rensning.
Adopt	Funktionen tar över ett objekt och ansvarar för upprensning.
Orphan	Anroparen tar över ansvaret för upprensning

B.4. KÄLLKOD

I denna bilaga bifogar vi endast en mindre del av vår källkod. Här ligger endast koden för den generella matrisklassen. Detta beror på att vi inte kan lägga ut vissa klasser, algoritmer och funktioner då dessa tillhör Vinga Systems affärshemligheter. Mycket arbete har lagts ned både från vår och deras sida för att arbeta fram denna kod och att sedan ge konkurrenterna gratis tillgång till den är ju mycket olämpligt.

Matrisklass - Headerfil

```
#ifndef MATRIX_H
#define MATRIX_H
#include<valarray>
#include <exception>

#include <crtdbg.h>


typedef std::valarray<double> Vector;
typedef std::slice Slice;
typedef unsigned int UINT;

class ExcpSingular : public exception
{

};

class CMatrix {

public:

      //construct a matrix of vector pv and size x and y
      CMatrix(Vector &pv, UINT x, UINT y);

      //operators for scalar to matrix operations
      void operator += (double d) {(m_pv) += d; }
      void operator -= (double d) {(m_pv) -= d; }
      void operator /= (double d) {(m_pv) /= d; }
      void operator *= (double d) {(m_pv) *= d; }

      //operators for matrix to matrix operations
      void operator += (CMatrix const& m) {_ASSERTE(m_Row == m.m_Row &&
m_Col == m.m_Col); (m_pv) += (m.m_pv); }
      void operator -= (CMatrix const& m) {_ASSERTE(m_Row == m.m_Row &&
m_Col == m.m_Col); (m_pv) -= (m.m_pv); }
      void operator /= (CMatrix const& m) {_ASSERTE(m_Row == m.m_Row &&
m_Col == m.m_Col); (m_pv) /= (m.m_pv); }
      void operator *= (CMatrix const& m) {_ASSERTE(m_Row == m.m_Row &&
m_Col == m.m_Col); (m_pv) *= (m.m_pv); }

      void SetSize(UINT x, UINT y); //Set size of matrix
```

```
	void SetZero();//Set all elements of matrix to 0
	void SetIdentity();  //Converte a matrix to a identitymatrix

	//returns a value of an element in matrix
	double El(UINT x, UINT y) const {return m_pv[element(x, y)]; }
	double & El(UINT x, UINT y) {return m_pv[element(x, y)]; }

	double operator() (UINT x, UINT y) const {return El(x, y); }
	double & operator() (UINT x, UINT y) {return El(x, y); }

	CMatrix Multiply(CMatrix& v1);  //Multiply matrix v1 & v2
	//Multiply the transposed matrix v1 with v2
	CMatrix TMultiply(CMatrix& v1);
	//Multiply matrix v1 with the transposed v2
	CMatrix MultiplyT(CMatrix& v1);
	//Quadrate a matrix
	CMatrix Quad1();

	//Multiply a matrix and a vector
	Vector MultV(Vector& v);

	//Multiply a vector and a matrix
	Vector VMult(Vector& v);

	void Invert();  //Invert a matrix

	void GetOutput();       //Print the matrix on the screen
	double GetSum();        //Sums all the elements of a matrix
	Vector GetSumRows();    //Sums the row of a matrix

	//Get the number of elements of a matrix
	UINT GetSize() const {return m_Row*m_Col;}
	//Get the number of rows of a matrix
	UINT GetRow() const {return m_Row;}
	//Get the number of columns of a matrix
	UINT GetCol() const {return m_Col;}

private:
	UINT m_Row, m_Col;
	Vector m_pv;
	UINT element (UINT x, UINT y) const { Slice s(x, GetCol(),
GetRow()); return s.start()+y*s.stride();}
	void CholeskyInPlace ();
};

#endif
```

Matrisklass – Sourcefil
Här finns några av de större funktionerna förklarade mer i detalj. Eftersom detta handlar om matematisk programmering är det ovanligt att läsaren eller användaren förstår med detsamma. Oftast kopieras därför funktionen när den skall återanvändas. Det som är viktigt för användaren är därför en kort beskrivning över vad funktionen utför och vilka in- och utvärden denna kräver.

```
#include "stdafx.h"
#include "Matrix.h"
#include <iostream.h>
#include <math.h>

CMatrix::CMatrix()
{
}

CMatrix::CMatrix(Vector &pv, UINT x, UINT y) : m_pv(pv), m_Row(x),
m_Col(y)
{

}


CMatrix CMatrix::Multiply(CMatrix& m1)
//Multiply the calling matrix with matrix m1
{

	_ASSERTE(GetCol()==m1.GetRow());
	Vector vMult(GetRow()*m1.GetCol());
	CMatrix R(vMult, GetRow(),m1.GetCol());

	for (int i=0;i<R.GetRow();i++)
	{
		for (int j=0;j<R.GetCol();j++)
		{
			R.El(i,j)=0;
			for(int k=0; k<GetCol(); k++)
			{
				R.El(i,j)=El(i,k)*m1.El(k,j)+R.El(i,j);
			}
		}
	}
	return R;
}


CMatrix CMatrix::TMultiply(CMatrix& m1)
//Multiply the tranposed calling matrix with matrix m1
{
	_ASSERTE(GetRow()==m1.GetRow());
	Vector vMult(GetCol()*m1.GetCol());
	CMatrix R(vMult, GetCol(),m1.GetCol());
```

```
	for (int i=0;i<R.GetRow();i++)
	{
		for (int j=0;j<R.GetCol();j++)
		{
			R.El(i,j)=0;
			for(int k=0; k<GetRow(); k++)
			{
				R.El(i,j)=El(k,i)*m1.El(k,j)+R.El(i,j);
			}
		}
	}
	return R;
}



CMatrix CMatrix::MultiplyT(CMatrix& m1)
//Multiply the calling matrix with transposed matrix m1
{

	_ASSERTE(GetCol()==m1.GetCol());
	Vector vMult(GetRow()*m1.GetRow());
	CMatrix R(vMult, GetRow(),m1.GetRow());

	for (int i=0;i<R.GetRow();i++)
		for (int j=0;j<R.GetCol();j++)
	{
		R.El(i,j)=0;
		for(int k=0; k<GetCol(); k++)
		{
			R.El(i,j)=El(i,k)*m1.El(j,k)+R.El(i,j );
		}
	}
	return R;
}


void CMatrix::SetSize(UINT x, UINT y)
//Set the size of matrix to x, y
{
	m_pv.resize(x*y);
	m_Row=x;
	m_Col=y;
}

void CMatrix::SetMatrix(Vector& pv, UINT x, UINT y)	//Give matrix
values for Vector pv and set size x,y
{
	m_pv=pv;
	m_Row=x;
	m_Col=y;
}

double CMatrix::GetSum()
//Get sum of all elements in matrix
{
```

```
	double j=0;
	for (int i=0; i<GetSize(); i++)
	{
		j+=m_pv[i];
	}
	return j;
}


Vector CMatrix::GetSumRows()
//Get sum of all rows in matrix to a vector
{
	Vector vR(GetRow());
	CMatrix R(vR, GetRow(),1);
	for (int i=0; i<GetRow(); i++)
	{
		R.El(i,0)=0;
		for (int j=0; j<GetCol(); j++)
		{
			R.El(i,0)=El(i,j)+R.El(i,0);
		}
	}
	return R.m_pv;
}

void CMatrix::SetZero() // Sets all value in matrix to 0
{
	for (int i=0;i<GetRow();i++)
	{
	  for (int j=0;j<GetCol();j++)
			El(i,j)=0;
  }
}

void CMatrix::GetOutput()     //Prints matrix on screen
{
	cout<<endl;
	for (int i=0;i<GetRow();i++)
	{
	  for (int j=0;j<GetCol();j++)
			cout<<El(i,j)<<"\t";
	  cout<<endl;
  }
}

CMatrix CMatrix::Quad1()  //Quadrate matrix
{
	Vector vR(GetRow()*GetRow());
	CMatrix R(vR, GetRow(),GetRow());
	for (int i=0;i<R.GetRow();i++)
	{
		for (int j=0;j<R.GetCol();j++)
		{
			R.El(i,j)=0;
			for(int k=0; k<GetCol(); k++)
			{
				R.El(i,j)=El(i,k)*El(j,k)+R.El(i,j);
```

```
                  }
            }
      }
      return R;

}
```