SESAM RAPPORT

Ag Metodik / Håkan Edler 15 november 1999

Sammandrag av resultat från mikroprojekt under 1999

Aktiviteten har varit god under året inom arbetsgruppen. De ursprungliga 19 förslagen till mikroprojekt kondenserades till fyra och tre projekt har givit signifikanta resultat till vårt mål, att samla, skapa och sprida information om metodik för systemutveckling. Det första projektet studerade hur man hanterar de stora informationsmängder, som skapas i ett utvecklingsprojekt. Man ställde upp en frågelista och har för varje fråga angivit hur man gör idag och hur man vill göra i framtiden. Det andra projektet studerade effekten av granskningar av dokument i kravhantering och konstruktion. Man har studerat vad som är bra och vad som är dåligt i dagens metodik och hur mycket man vunnit på att hitta fel tidigt. Det tredje projektet studerade metod och praktik för provning av datorbaserade system. Man har sammanfattat och beskrivit tekniker för provning och studerat hur dessa tekniker används. Vidare har man mätt vilken effekt man får av att hitta fel tidigt och studerat hur man kan använda formella metoder vid konstruktion av system.

1. Dynamiska informationsstrukturer

Frågorna kretsar kring "informationstrukturen", som är den information vi bygger upp i ett systemarbete, i en databas eller i ett annat lagringsmedium, med all information om och runt komponenterna i ett system under arbete. Idén är alltså att "informationsstrukturen" skall byggas under utvecklingens gång och vara komplett för varje utgåva (version) som levereras av ett system.

Det "dynamiska" är då att strukturen förändras tillsammans med produkten under utvecklingsarbetets gång. Arbetet har genomförts genom att gruppen ställt samman en frågelista med faktorer att ta hänsyn till under utvecklingen av ett system. Speciellt har man tänkt på hanteringen av krav.

Deltagare var:

- Ingmar Ögren, Romet
- Harriet Borgman, Ericsson Microwave Systems
- Peter Camitz, Ericsson Microwave Systems
- Billy Johansson, CelsiusTech Electronics.

1.1 Representation

1. Hur beskriver vi systemstrukturer (SA, UML klassdiagram, UML komponentdiagram, annat)?
2. Vilka informationsstrukturer har att göra med krav, direkt eller indirekt?
3. Hur bygger vi dessa informationsstrukturer (papper & penna, ordbehandlare, CASE-verktyg, annat)?
4. Hur representerar vi kundkrav?
5. Hur representerar vi övergripande krav på högsta nivå (Uppdrag, förmågor, användningsfall, scenarior, annat)?
6. Arbetar vi med formalisering av krav t. ex. genom att uttrycka kraven i boolesk syntax?
7. Utnyttjar vi formaliserade beteendebeskrivningar (t.ex. tillståndsdiagram, pseudokod) som en del i kravarbetet och i så fall hur?
8. Hur hanterar vi krav med icke-textuella delar, som t. ex. diagram?

1.2 Kravkopplingar

9. Hur fördelar vi krav?
10. Hur relaterar vi olika krav till varandra, t ex kundkrav, härledda krav, funktionella krav, egenskapskrav?
11. Hur relaterar vi krav till provfall?
12. Hur hanterar vi härledda krav, som framkommer under ett utvecklingsarbete?

1.3 Ändringshantering

13. Hur hanterar vi information, som uppkommer under utvecklingsarbetet som t. ex. problem, användarsynpunkter, annat?
14. Hur hanterar vi förändringar i systemstrukturen under en utveckling med säkerställande att krav är korrekta och konsistenta samt hur säkerställer vi att hela strukturen är konsistent?
15. Hur hanterar vi svårigheter att ställa rätt krav, som kan uppstå t.ex. vid långt och nära samarbete med kund, när rollerna användare - kund - leverantör, är oklara och/eller det finns motstridiga uppfattningar?

1.4 Beslutsvägar

16. Hur prioriterar vi mellan krav, t.ex. funktionella krav och icke-funktionella krav? (Konstruktören vet normalt vad systemet ska klara men ibland inte varför)
17. Hur hanterar vi ändringsstyrning av krav, vilka beslutsvägar finns?

1.5 Övrigt

18. Hur stödjer vi återanvändning av krav?
19. Hur hanterar vi krav för att stödja varianter av produkter?
20. Hur hanterar vi krav vid stegvis utveckling med ökad funktionalitet?
21. Hur genererar vi relevanta dokument (krav och andra dokument)?
22. Hur säkerställer vi att programmerare och slutanvändare bägge förstår dokumentationen och varandra?
23. Hur hanterar vi krav vid krav på
    - korta ledtider (specificering och utveckling överlappar varandra)?
    - utveckling av system i flera utgåvor med successivt ökad funktionalitet?
    - flera samtidiga varianter av ett system (utveckling och underhåll), dvs. för kundanpassade system?
24. Hur hanterar vi krav från underleverantörer ?

2. V&V – granskningsmetodik

Granskning av dokumenten i en utvecklingsprocess är en effektiv metod att åstadkomma en given kvalitet i en produkt. Mikroprojektets mål var att samla in erfarenhet om vilka metoder och verktyg man använder vid granskning och data om hur effektiv granskningen är.

Deltagare var:

- Mari Persson, Celsius, Kockum Naval Systems, som gjorde en genomgång av en problemrapportdatabas för ett Ada programutvecklingsprojekt i syfte att finna andelen fel som borde upptäckts vid tidigare granskningstillfällen samt konsekvenserna och slutsatserna av detta.

- Billy Johansson, CelsiusTech Electronics, som samlade in erfarenheter från ett förbättringsprojekt i syfte att ta fram en anpassad och ensad metod för genomförande av formella granskningar samt skaffa erfarenheter från verktygsstöd.
- Mats Forsman, Enator Telub, som har följt upp effekterna av granskning enligt den relativt formella metod man har i företagets handbok för systemutveckling.

2.1 Några resultat

Några motiv man har till att genomföra formella granskningar är:

1. Formella granskningar är den mest kostnadseffektiva metoden att hitta avvikelser och förbättringsförslag. Studier visar på att granskningar har en utväxlingspotential i nytta/kostnad med en faktor >11.
2. Man kan i en enda granskning hitta mer än 80% av felen (granskningseffektivitet)
3. Utvecklingskostnaderna minskade med 50% efter införande av granskningar (IBM studie)
4. Underhållskostnader minskade med en faktor >10 genom införande av granskning (Tom Gilb studie)
5. Granskningar kan minska den totala projekttiden med 50%.
6. Formella granskningar tar tid i början av utvecklingen, men den totala LCC-kostnaden minskar genom att nedlagd tid i prov- och verifieringsfasen samt i underhållsfasen för en produkt reduceras.
7. Formella granskningsmöten är effektivare - man hittar fler fel tillsammans än vid en individuell (informell) granskning.
8. Oklarheter och missförstånd undviks vid formella granskningar genom att det är svårt för t.ex. författaren av ett dokument att avgöra vad som är fel om vederbörande inte har hört granskarens beskrivning av felet.
9. Viktigt att se till nyttan med granskningar, dvs. mäta effekterna av effektiva granskningar inte bara se till kostnaden för granskningen.
10. Största vinsten av genomförande av granskningar erhålls i de tidiga faserna av ett projekt - granskning av kravdokument.

Några resultat av arbetet är:

- Ungefär hälften av avvikelserna borde upptäckts vid något tidigare granskningstillfälle.
- Andelen specifikationsrelaterade problem är över 50%, såväl för "normala" problemrapporter som för de man borde upptäckt tidigare. Det borde löna sig att granska kravdokumenten ännu bättre
- Flertalet avvikelser har passerat två eller flera granskningar utan upptäckt.
- Ju senare felupptäckt desto större tidsåtgång för åtgärdande.
- Korrigeringstiden är längst för kravrelaterade felrapporter.
- Man skall ha en väl etablerad process för datainsamling, sammanställning till mätetal och återkoppling för att höja kvaliteten på produkterna och ge den önskade kompetensuppbyggnaden.

2.2 Sammanfattning

Att genomföra granskningar på ett formellt sätt är inte någon "silver bullet" som löser alla problem med produktutveckling, men det är ett kostnadseffektivt sätt att

hitta fel i produkter och ger andra positiva effekter såsom "teamlärande", dvs. kompetensutveckling. Genom att ta del av andras synpunkter och erfarenheter undviks många problem och ger, på sikt, kortare utvecklingstider.

Med formell granskning avses en procedur som omfattar planering av granskningen, en faktagenomgång inför ett granskningsmöte. Vid granskningsmötet rapporteras alla funna (potentiella) problem som identifierats vid den individuella granskningen. Vid eller efter granskningsmötet beslutas vilka åtgärder som ska vidtas. Därefter sker en uppföljning av vidtagna åtgärder.

Många studier och undersökningar visar på att det existerar brister framförallt vid granskning av kravdokument, bl.a. krav som inte är uppfattade på rätt sätt, förbisedda krav m.fl. Detta för med sig ett antal omtag i projekten som är kostnadsdrivande och ofta medför förseningar. En attitydförändring krävs ofta och att genom styrning se till att mer resurser på granskning läggs i början av produktutvecklingsfasen i allmänhet och i systemspecificeringsfasen i synnerhet.

Saknas någon av de aktiviteter som måste ingå i en effektiv formell granskningsprocess upplevs granskningen som ineffektiv. Mycket tid tas i anspråk för att diskutera t.ex. en konstruktionslösning istället för att ge synpunkter på befintligt underlag. Dessutom förbrukas ett antal timmar i onödan för dem, som kallats som granskare. De sitter overksamma på granskningsmötet och upplever granskningen som onyttig. Så kanske man slutar att ha granskningsmöten, baserat på felaktiga slutsatser. Rätt genomförda granskningar är ett medel att minska kostnader och genomloppstider.

Skilj på att läsa ett dokument och att granska det, dvs. granskning måste ske mot källdokument. Checklistor och "design rules" är ofta ett stöd för främst oerfarna granskare.

Viktigt att ha ett granskningsteam sammansatt av flera olika roller/kompetenser (dock inte nödvändigtvis flera olika personer, en person kan ha flera roller) för att fokusera på vad som är viktigt vid det aktuella granskningstillfället och för det aktuella objektet.

Gör klart målet innan man inleder en granskning. Generella mål för granskningar:

- Säkerställa att man kan påbörja nästa fas.
- Förbättra produktkvaliteten.
- Förbättra processerna (arbetsmetoderna).
- Öka kompetensen, motivationen, utbilda medarbetare etc.

3. V&V, – provningsmetodik

Dynamisk verifiering av programvara innebär att man kör ett programsystem eller en modell av det. Detta till skillnad mot statisk verifiering, då man t ex granskar resultatet av en konstruktion mot förutsättningarna eller kör en kodverifierare. Det finns ett stort antal metoder för såväl dynamisk som statisk provning, men de kan vara svårt att hitta beskrivningar av dem. Målet med mikroprojektet har varit, att kartlägga några metoder för provning, se hur provning tillämpas i praktiken och att relatera detta till utveckling med matematisk formalism.

Deltagare har varit:

- Lena Sporre, FOA
  Val av indata för provning, litteraturstudie
- Håkan Edler, Chalmers
  Black-box-, white-box-, threadtesting, litteraturstudie
- Dag Folkesson. Saab AB, Gripen
  Provning resp övervakning av distribuerade RT-system i farkoster
- Bertil Lundgren, Celsius, Kockums
  Provning / felupptäckt i olika faser
- Per Larsson, Industrilogik L4i AB
  Formell programvaruutveckling med B-metoden

Arbetet har resulterat i fem rapporter enligt följande kapitel.

3.1 Några tekniker för provning med varierande indata samt beskrivningar av debugging-tekniker

3.1.1 Provning och provfall

Provning är en process att exekvera ett program med avsikten att finna fel.

Provfall och provdata är inte samma sak. Provdata är indata som har valts ut för att prova systemet med. Det är ibland möjligt att generera provdata automatiskt men omöjligt att generera provfall.

Ett provfall för provning med varierande indata består av en

- specifikation av indata
- beskrivning av systemfunktionerna provade med dessa indata
- utsaga om det förväntade resultatet

Eftersom man med provning inte kan påvisa frånvaron av felaktigheter utan bara visa på att programvarufel finns, bör provfallen konstrueras så att olika klasser av fel systematiskt avslöjas med ett minimum av ansträngning och tid.

Ett bra provfall bör vara så konstruerat att det

- har en hög sannolikhet att finna ett ännu inte upptäckt fel
- inte är redundant
- varken är för enkelt eller för komplext

Black-box-tekniker fokuserar på provning av funktioner. Två viktiga tekniker finns för att välja provfall:

- Indelning i ekvivalensklasser.
- Val av värden i gränserna mellan ekvivalensklasser.

3.1.2 Debugging

Till skillnad mot provningsprocessen, som består av att exekvera ett program med avseende att finna fel, är debugging en process som utnyttjas när en indikation har erhållits på att det existerar ett fel. Indikationen kan vara resultatet från ett lyckat testfall. Provning och debugging har således en nära koppling till varandra.

Debugging består av aktiviteter i en tvådelad process:

1. bestämning av karaktären på och lokaliseringen av felet
2. rättning av felet

Punkt 1 omfattar här upp mot 95% av det totala arbetet.

Det finns olika typer av metoder för debugging:

- debugging by brute force
- debugging by induction
- debugging by deduction
- debugging by backtracking
- debugging by testing

3.2 Om några metoder att prova programvara

Provning av system har i grunden två olika syften: Att visa att systemet svarar mot kraven och att hitta fel. Det första syftet är dels validering av systemet, när man vill visa, att systemet fungerar så som en användare tänkt sig, dels verifiering, när man antingen granskar en konstruktion och jämför den med dess specifikation eller kör systemet mot provningsföreskrifter och visar att system beter sig som förväntat. Det andra syftet är att finna och ta bort konstruktionsfel och att förutsäga förekomsten av kvarvarande fel.

Statisk provning är när man verifierar eller validerar ett system enbart med hjälp av dess dokumentation. Det är: Granskningar, statisk programanalys och formella metoder.

Dynamisk provning gör man för att finna fel och för att uppskatta systemets tillförlitlighet och då kör man ett programsystem eller en exekverbar modell av det. Stegen i dynamisk provning är: Enhetsprov, modulprov, integrationsprov och acceptansprov.

I integrationsproven kan man använda flera olika tekniker: Top-down, bottom-up, trådad provning och back-to-back.

Gjorda experiment visar att statisk provning är mer effektiv än dynamisk provning. Man fann ett större antal fel och fler fel per tidsenhet vid statisk provning.

Statistisk provning gör man för att bedöma tillförlitligheten i ett system. Man bestämmer användningsprofilen för systemet, väljer provdata slumpmässigt ur användningsprofilen och kör provfall med dessa data och notera tiden varje gång man observerat ett fel. När man samlat tillräckligt många noteringar om fel kan man med hjälp av statistiska metoder bedöma antalet kvarvarande fel.

Det finns två grundläggande principer för felupptäckande provning: strukturstyrd provning - white-box testing - och beteendeprovning - black-box testing. I strukturstyrd provning utnyttjar man kunskap om ett systems interna struktur för att prova alla möjliga exekveringsvägar. Det är en metod att konstruera provfall utifrån kunskapen om hur ett system är uppbyggt. I beteendeprovning utgår man från specifikationerna av ett systems gränssnitt och beteende och provar om transformationerna från indata till utdata är korrekta. Beteendeprovning används för att verifiera ett system både mot funktionskrav och övriga krav. Vid beteendeprovning kör man

systemet utan kunskap om dess interna uppbyggnad, bara indata och utdata är kända. Man väljer en mängd provdata, kör systemet och observerar resultatet.

3.3 Provning stödd av inbyggd realtidövervakning för distribuerade system i farkoster

Effektiv **provning** av realtidssystem erfordrar en kvalificerad detektering/ mätning av feltillstånd, inte endast observation av felyttring/haveri (failure).

Inbyggd övervakning är effektiv och nödvändig för att garantera säkerhet och funktion med varning eller reservinkoppling vid problem. Lång erfarenhet visar även att sådan inbyggd övervakning i distribuerade realtidssystem är en värdefull grund att utnyttja även för utprovning (verifiering och validering) av realtidsexekveringen.

Inbyggd övervakning utför detektering och utvärdering av systemet under ordinarie drift.

Inbyggd test utför reguljärt “provning” av utrustning (mest hårdvara) genom speciell excitering under icke-drift, och även sådan “test”teknik är intressant för såväl ordinarie provning (verifiering/validering) som funktionsövervakning.

Inbyggd test och inbyggd exekveringsövervakning enligt ovan ingår således permanent i det ordinarie systemet.

Den inbyggda övervakningen, feldetekteringen

1. har stora likheter med utvecklingsprovningen,
2. har dessutom stora fördelar av att ha tillgång till tillstånd och data i realtidsexekutiven under exekveringsförloppet och
3. är tillgängligt under betydligt större (drift- och kalender)tid och med betydligt fler och varierande driftfall än "normal" utvecklingsprovning kan vara m a p realtidsförloppet.

Av ovanstående kan då inses att

- utvecklingsprovning/utprovning har stora likheter med den redan inbyggda övervakningen och (konstlast-) testningsfunktionerna.
- och till stor nytta kan utnyttja dessa

3.4 Provning / felupptäckt i olika faser

Hos Kockums delas testverksamheten för programvarusystem, som är utvecklade i Ada, in i följande faser:

- Integrationstest, INT — Test i referenssystemmiljö med vissa teststubbar, men framför allt PC-simulatorer
- Formell test , FQT — Test i referenssystemmiljö med specificerade testplaner och testbeskrivningar, godkända av kund. PC-simulatorer används

- Systemtest i fabrik, FAT — Test i uppbyggd landanläggning, huvudsakligen mot verkliga utrustningar, kompletterade med simulatorer för att skapa scenarier

- Systemtest ombord, SAT — Test hos kund ombord fartygen i verklig miljö. Normal användning ombord.

Det är allmänt känt att ju tidigare man upptäcker ett fel desto enklare och snabbare är det att åtgärda felet. Tiden för att felsöka och åtgärda ett fel ökar med en 'fasfaktor' mellan varje fas.

Med dessa förutsättningar ansätter vi faktorn 4 mellan faserna integrationstest och formell test. Vid övergång till systemtest i fabrik och systemtest ombord är det sämre kontakt mellan den som upptäcker och beskriver felet, och den som skall analysera och åtgärda det. Här ansätter vi den något högre faktorn 5.

Vi gör således följande ansats för en felrättningsinsats under de olika faserna. I tiden ingår arbetet med att identifiera, analysera och korrigera ett programvarufel:

- Under integrationstest, INT 1 h
- Under formell test , FQT 4 h
- Systemtest i fabrik, FAT 20 h
- Systemtest ombord, SAT 100 h

Man kan då göra följande räkneexempel på en trolig besparing vid tidigare felupptäckt:

- Om 8 st SAT-fel i stället hade upptäckts under FAT, blir besparingen: 8 x (100-20) = 640 h
- Om 7 st FAT-fel i stället hade upptäckts under FQT, blir besparingen: 7 x (20-4) = 112 h

Kan man undvika fel i sena faser gör man avsevärda besparingar. Uppflyttningar i de tidiga faserna ger endast marginella besparingar.

3.5 En snabb introduktion till B-metoden

Med formella metoder i systemutveckling menar man i allmänhet användningen av logik och matematik i ett beskrivningsspråk med formell syntax och semantik. Vid användningen av formella metoder utgår man från informella krav och modellerar dem på hög abstraktionsnivå. Vinsten med detta är, dels att man beskriver vad systemet skall göra men inte hur det skall göras, dels att man får en entydig specifikation som underlag för det fortsatta utvecklingsarbetet. Man kan få en precis förståelse för kraven, men gapet till det färdiga programmet, som skall realisera funktionerna, är stort, då detaljeringsgraden i ett programspråk är för stor jämfört med det logiska beskrivningsspråket. Man kan lösa problemet med att använda delmängder av programspråket som pseudokod för att gradvis förfina specifikationerna från krav till färdigt program

B-metoden är en formell metod, som är avsedd att stödja både specifikationer på en hög abstraktionsnivå och utveckling av program på lägre nivåer. Den täcker en stor del av utvecklingskedjan till skillnad från de flesta andra formella metoder, som enbart kan användas för abstrakta beskrivningar av krav. En komplett sy-

stembeskrivning i B består av en hierarki av specifikationer med den abstrakta systemspecifikationen på högsta nivån och en så gott som exekverbar konstruktion på den lägsta. B-metoden är avsedd att stödja utveckling av system som skall skrivas i något av de vanliga imperativa programspråken som Ada eller Pascal.

Metoden har använts i ett antal tillämpningar, där de mest kända är från det franska företaget GEC-AHLSTROM-Transport. Företaget tillverkar utrustning och levererar signalanläggningar för järnvägar. Under det senaste decenniet har man utvecklat flera stora system, där B-metoden använts i hela utvecklingsprocessen, och man rapporterar goda resultat av detta.

I Sverige har B-metoden använts i ett forskningsprojekt om användningen och integrationen av formella metoder i industriell systemutveckling. I projektet deltog både industri och akademi och ett av målen var att prova några formella metoder i praktisk användning, däribland B-metoden. Den tillämpning man valde var dokumenterad med en mängd tillståndsgrafer blandat med C-program och informella beskrivningar. Med hjälp av systemets konstruktör fann man det ganska enkelt att göra om beskrivningarna i enbart B-språket, trots att ingen av projektdeltagarna hade djupare kunskaper i vare sig metod eller språk.

Åtskilliga positiva erfarenheter finns alltså av praktisk användning av B-metoden. Förutom att vara en formell metod har den fördelarna av att ha syntax och semantik, som liknar ett imperativt språk och att det kan användas både på hög abstraktionsnivå och för förfining och detaljering ned till programkonstruktion. En van systemkonstruktör kan relativt lätt börja använda metoden och språket utan att ha lärt sig alla detaljer.

Rapporten från mikroprojektet går igenom B-metodens fundamenta.