NYHETSBLAD FÖR SESAM
Försvarssektorns Adaintressenters Användargrupp för Software Engineering

RENDEZVOUS

Nr 1 mars 2000

Innehåll

GLÖM EJ AIS/SESAM VÅRSEMINARIUM DEN 13 APRIL

Försvarssektorns Adaintressenters Användargrupp för Software Engineering

SESAM

Vad är SESAM?

SESAM har tillkommit för att organisera och stimulera samarbete och samverkan inom programvaruområdet mellan försvarsindustrin, FMV och FOA.

Det avtalsfästa syftet med SESAM är ”att genom organiserat samarbete mellan användargruppens medlemmar främja tillförlitlighet och effektivitet i utveckling och vidmakthållande av programvarusystem i Ada inom försvarssektorn”. Inom ramen härför skall SESAM även anpassa, profilera och förnya sin verksamhet med hänsyn till ändrade tekniska och andra omständigheter av betydelse för intresseområdet.

Följande kommer att ske under den närmaste 2-3-årsperioden.

1. SESAM skall allmänt verka för att sprida information om faktorer som påverkar möjligheterna till tillförlitlig och effektiv utveckling och vidmakthållande av programvarusystem. Särskilt skall härvid Adas betydelse i sammanhanget klargöras.

2. SESAM skall i sin verksamhet fortlöpande bevaka möjligheterna att samla, skapa och sprida information om objektiva mät- och andra resultat och erfarenheter vunna vid användning av ”software engineering”-principer och Ada.

3. SESAM behandlar tillvägagångsättet vid utveckling och vidmakthållande av programsystem. Implicit i detta ligger givetvis att använda processer skall tillförsäkra de resulterande produkterna efterfrågade egenskaper. Produktegenskaper som påverkas av processerna är därför av primärt intresse att bevaka i SESAMs verksamhet.

4. SESAM skall i sin verksamhet fästa stor vikt vid att underlätta samexistens mellan Ada-program och programvara skriven i andra språk. Speciellt skall aspekter vid användning av COTS beaktas.

5. SESAM skall där så är möjligt sätta konkretiserade och mätbara mål för sin verksamhet under avgränsade tidsperioder.

SESAM styrs av ett Råd med representanter för gruppens medlemmar. Rådet har till sin hjälp ett Verkställande Utskott (VU) och ett sekretariat.

Rådets ordförande är Claes Wadsten, AerotechTelub, tel 013-231652 .

VU

Bengtsson Christopher, FMV
chben@tranet.fmv.se

Brandt Roger, FMV
robra@fmv.se
Ekfeldt Sune, AerotechTelub
sune.ekfeldt@aerotechtelub.se
Johansson Billy, CelsiusTech Electronics AB
bijo@celsiustech.se
Källberg Björn, CelsiusTech Systems AB
bjkae@celsiustech.se

Arbetet utförs i ett två arbetsgrupper:

Ag Metodik
Håkan Edler, CTH/Datorteknik
edler@ce.chalmers.se

Ag Teknik
Gunnar Fredriksson, FMV
gufre@fmv.se

Vilka kan vara med i SESAM?

Medlemmarna i SESAM är svenska företag, organisationer och myndigheter (förvaltningar, utbildningsinstitutioner etc) med anknytning till försvarssektorn. Medlemmarna indelas i följande kategorier
- ordinarie medlemmar
- arbetsgruppsmedlemmar
- informationsmedlemmar.

Enskild person kan endast komma ifråga som informationsmedlem.

Inträde i SESAM

För samtliga medlemskategorier gäller att inträde beslutas av Rådet.

För inträde som ordinarie- och arbetsgruppsmedlem krävs status som leverantör till FMV. Dessutom krävs en skriftlig förbindelse att uppfylla åtagande som ordinarie- och arbetsgruppsmedlem.

För inträde som informationsmedlem (erhåller endast informationsbladet) krävs status som leverantör till FMV eller status som myndighet inom totalförsvaret. Rådet kan emellertid anta annan part som informationsmedlem.

För ansökan om medlemskap i SESAM vänd er till sekretariatet.

SESAM-Sekretariatet
AerotechTelub AB
c/o Kåsjös Kontor
Ytterspåret 14
187 54 TÄBY

Ordföranden har ordet

Dagens militära system och produkter innehåller mycket programvara. Med den nyinriktning försvaret gör kommer detta att öka. För att klara kort utvecklingstid och låg utvecklingskostnad krävs nya verktyg och metoder.

Med den snabba utvecklingstakten inom området får vi tillgång till fler och fler verktyg och metoder. Även stora företag har svårt att utvärdera alla till en rimlig kostnad. Ett samarbete mellan företag och institutioner är därför av största vikt.

SESAM kan vara en av de föreningar som hjälper till att få olika företag/institutioner att lösa eller belysa gemensamma problem för att underlätta utvecklingen av den egna produkten.

Vid seminariet hösten 1999 visade medlemmarna i SESAM att det fanns mycket att redovisa från de ”mikroprojekt” som genomförts i samarbete mellan medlemmarna.

För år 2000 har de båda arbetsgrupperna TEKNIK och METODIK ett antal förslag på projekt som passar in på ovan nämnda problem.

De projekt som prioriterats för år 2000 av arbetsgrupperna är:

Ag **METODIK,** ordförande Håkan Edler, CTH

1. Riskhantering
2. Aktuella utvecklingsmetoder
3. Komponentbaserad programutveckling inklusive COTS-relaterade aspekter.
4. Metoder och tekniker för granskning av kod och objektorienterade modeller.

Ag **TEKNIK**, ordförande Gunnar Fredriksson, FMV

5. IT-säkerhet ur tekniskt perspektiv
6. Mönster och arkitekturer med avseende på software engineering
7. Fortsatt verksamhet HLA.

Det är min förhoppning att vi även i år får se medlemmarna samarbeta inom olika frågor och att vi får intressanta redovisningar vid seminariet i höst.

SESAM kan inte behandla alla frågor och problem som finns men belysa några samtidigt som ett kontaktnät skapas. Att stödja institutioner och högskolor samt skapa kontaktnät inom programvara är den viktigaste uppgiften för SESAM. Detta kan vi göra genom att gemensamt samlas kring ovan projektförslag.

Med hälsningar
Claes Wadsten

Status FoTA P12:

Överföring till industrin av programvaruteknik för säkerhetskritiska system

Sedan föregående Rendezvous-blad har P12 hållit två heldagsmöten -ena halvan av dagen reserverad för projektmöte inom P12 och andra halvan presentationer från/för övriga FoTA-projekt inom programvaruteknik.

Vid det senaste mötet, 2000-02-09, gick Lars Asplund (FoTA P10) igenom principerna för den run-time-kärna enligt Ravenscar:s tasking-modell, som är under implementering på Uppsala Universitet (Kristina Lundquist disputerar på arbetet den 14.4). Formell verifiering i verktyget UPPAAL av kärnan + en liten applikation har utförts. Fler Ada-applikationer (på ca 5-8 tasks) söks nu, för att se hur ökad komplexitet påverkar verifierings-och svarstider. (Eventuellt kan en ex-jobbare assistera företag med intresse av att verifiera kärnan på egen strippad applikation). Presentationen samt anvisningar för hur man kan transformera en applikation till verifierbar modell kommer att läggas ut på P12:s hemsida..
Ingmar Ögren från FoTA P11 talade om metoder och verktyg för förståelig formalism och säkerhetsanalys samt hur detta understöds av verktyget TOFS (se www.toolforsystems.com).

Huvudleverantör för P12 är utsedd (Aerotech Telub AB, representerad av Ingemar Johansson). Ingemar har lagt upp en enkel hemsida (www.aerotech.ffv.se/fota_p12) för information mellan FoTA:s projekt P12 och motsvarande huvudprojekt. Hemsidan är tillgänglig för P12-medlemmar samt huvudprojektens kontakpersoner inom industri och FMV.

P12-medlemmarna har genomfört ett antal uppgifter, bla en inventering av det egna företagets teknik för system- och programvarusäkerhet samt granskning av P7:s rapport för etapp 1 (Utredning rörande användning av civila/kommersiella produkter och teknik i militära lednings- och informationssystem). För övriga huvudprojekt, som har material under färdigställande, påminns om denna möjlighet att använda sig av P12 som granskningsinstans.

Nästa möte är inplanerad 2000-05-03. Som vanligt reserveras en halvdag för genomgångar och diskussioner med huvudprojekten. Inriktningen denna gång är mot de huvudprojekt, som ännu inte genomfört en detaljerad presentation (främst P3, P4, P7, P9). Avsikten är att P12-medlemmarna skall få tillräcklig information, för att kunna välja teknik för prov på eget företag under hösten.

Några extra rum har bokats på FMV:s konferensanläggning i B-huset för huvudprojekt som önskar ha egna möten parallellt. De som är intresserade av detta kan kontakta mig.

Inga-Lill Bratteby-Ribbing
AOL FoTA P12
ilbra@fmv.se
tel 018-12 02 63

Tofs AB har beslutat lägga ut programmet Tofs i version 98.2 för gratis nedladdning från webadressen www.toolforsystems.com. Detta innebär att man kan få ett gratis verktyg för systemarbete med stöd för krav, konstruktion, verifiering och dokumentation utan begränsningar vad gäller sådant som tid och systemvolym.

Giraffe styrs med WAP-telefon

Följande artikel har via Torbjörn Andreasson välvilligt ställts till SESAM förfogande av EMW's interna infoblad SENSORN.

Länge var den civila tekniken portförbjuden i försvaret. Höga säkerhetskrav krävde särskilda lösningar. Men nu börjar det hända saker. På divisionen för Sensorer och informationsnät har programvaruingenjörerna satt en webbserver i en Giraffe och styr hela radarstationen från en WAP-telefon. Civil telekomteknik har blivit en grundsten i det framtida försvar som nu tar form.

Giraffe styrs från en WAP-telefon. Johan Andersson, Anders Lange och Torbjörn Andreasson demonstrerar tekniken som utvecklats på divisionen för Sensorer och informationsnät.Foto: Marie Ullnert/Kamerareportage

Tänk dig följande scenario från en framtida väpnad konflikt:

En soldat får order om att genomföra ett uppdrag. Uppdraget kräver detaljerad information om det aktuella området samt understöd från luftvärn. Via sin mobila kommunikationsterminal sätter soldaten samman ett informationssystem, skräddarsytt för det aktuella uppdraget. Han kopplar upp sig mot en radarstation och ett robotförband som han styr under den tid det tar att genomföra uppdraget. Via kommunikationsterminalen väljer han de mål radarn ska låsa på, skickar vidare radarinformationen till artilleriförbandet och ger order om eldgivning med en knapptryckning.

PC i radarstationen

Det kan låta som science fiction, men ett första steg mot det beskrivna scenariot har redan tagits. På Ericsson Microwaves nybildade division för Sensorer och informationsnät har programvaruingenjörerna utvecklat ett system för fjärrstyrning av luftvärnsradarn Giraffe.

En PC placerad i radarstationen fungerar som en webbserver. Servern översätter informationen från radarstationen till öppna, civila protokoll och gör den tillgänglig via en vanlig Internetbrowser eller en Wap-telefon.

- Vi använder civil kommunikationsteknik på sätt som är helt nytt för försvaret, berättar Torbjörn Andreasson, ansvarig för programvarulabbet på divisionen för Sensorer och informationsnät.

Med webbservern är radaroperatören inte längre bunden till Giraffe-stationens hydda. När informationen distribueras på Internet eller trådlöst, kan radarstationen styras var som helst ifrån. Från en centralt placerad stridsledning eller av en soldat i fält.

Tillgången till information är grundstenen i den inriktning svenska försvaret valt för framtiden. Det egna territoriet och den närmaste omgivningen ska detaljbevakas med sensorer av olika slag: radar, tv- och IR-kameror. Informationen från sensorerna distribueras på ett bredbandigt telekomnät, byggt med civil teknik.

Det är här det nya systemet från Sensorer och informationsnät kommer in i bilden.

- Vårt system kan ses som en ryggsäck som kan hängas på vilket försvarssystem som helst, berättar Anders Lange, ansvarig för produkt- och systemledning på divisionen för Sensorer och informationsnät. När alla radarstationer, robotenheter, artilleriförband med mera förses med en webbserver och kopplas till telekomnätet kan man hämta information om och styra insatser till vilken del av territoriet som helst, var man än befinner sig.

Men tekniken från divisionen för Sensorer och informationsnät är mer än bara ett sätt att få olika försvarssystem att prata med varandra. Divisionens programvaruingenjörer introducerar ett helt nytt sätt att bygga militära ledningssystem.

Kan lätt skräddarsys

Dagens system är utvecklade för specifika tillämpningar. I takt med omvärldsförändringar och den tekniska utvecklingen passeras systemens bäst föredatum.

- Vad vi har gjort är att vända på alltihop, berättar Johan Andersson, ansvarig för marknads- och affärsutveckling på Sensorer och informationsnät. Istället för att utveckla ett system för en specifik uppgift ser vi framför oss en plattformsoberoende infrastruktur för en mängd olika tjänster och funktioner, till exempel styrning av en radarstation. Funktionerna kan kombineras till ett system, skräddarsytt för den aktuella tilllämpningen.

Niclas Henningsson

50 år av ”Software Engineering”; vad har betytt mest

Referat ur IEEE Software Jan/Feb numret 2000

Efter 50 år av Software Engineering (begreppet som sådant uppfanns väl dock först 1968) och inför det nya århundradet kan det vara intressant att diskutera vad som har det bästa inflytandet på detta område. Det är vad redaktören Steve McConnel på IEEE Software iscensatt i tidskriftens Jan/Feb nummer. Där redovisar han synpunkter från sina redaktions- och industriuella råd, sammansatt av internationella experter inom området, på en 10-bästa lista enligt följande:

- Reviews and Inspections
- Information Hiding
- Incremental Development
- User Involvement
- Automated Revision Control
- Development Using the Internet
- Programming Languages
- Capability Maturity Model for Software
- Object-Oriented Programming
- Component Based Development
- Metrics and Measurement

Åsikterna skiljer sig en hel del mellan experterna på vissa punkter. Betr *Reviews and Inspection*s och *Information Hiding* innebörd och positiva inverkan verkar de dock vara helt överens och eftersom de är rätt så självklara begrepp så går vi inte närmare in på synpunkterna på dessa.

Betr *Incremental Development* har man litet olika åsikter betr ”Daily Build”, vilket kanske kan sägas vara det ultimativa i inkrementell väg. Någon tycker att det är en ”brute force” metod som inte är någon god förebild, medan någon annan menade att det bara är en följd av att man insett det naiva i att tro att man kan förutse alla implikationer av en komplex kravbild. Ytterligare någon menar att Barry Boehms Spiralmodell är det sätt på vilket de flesta goda programmerare alltid arbetat. T o m Boehm sägs på 70-80-talet att ha noterat att ”Daily Build” låg närmare verkligheten än den tidens fasindelade (vattenfalls)modeller.

User Involvement tycker någon är särskilt viktigt för användargränssnitt för att det handlar om ett fundamentalt ingenjörskoncept, nämligen att bygga modeller så att folk kan se i förväg vad det är dom kommer at få (som ett hus eler en bro). En annan framhäver ”Use Case” som en kraftfull metod för att förstå användarkrav och bygga system som uppfyller dem. Detta får, inte oväntat, visst mothugg från Steve Mellor, en i expertgruppen, som anser att de är usla som underlag för att skapa nödvändiga abstraktioner, men att dom i vissa fall är användbara.

Automated Revision Control ansers vara viktigt i stora projekt, men någon påpekar att de verkligt intressanta programmen tenderar att skrivas i små grupper, där fördelarna av automatiserad ändringskontroll inte är så tydliga, t ex i många ”open-source” verksamheter. Man förutser också att framtida verktyg i denna kategori kommer att bättre utnyttja datorenas inneboende förmåga att komma ihåg, kommunicera och organisera data, snarare än som hittills att försöka automatisera pappersrutiner.

Development Using the Internet föranleder inte så många kommentarer, men antas bli mycket viktigt genom att det tillåter stora distribuerade samverkansprojekt som ”can change the face of software development as we know it”.

Betr *Programming Languages* behandlar man de fyra språk man anser hör till programspråkens ”Hall of Fame”, Fortran, Cobol, Turbo Pascal och Visual Basic. *Fortran* anses om inte annat ha stor betydelse som det första allmänt spridda tredje generationens språk. *Cobols* betydelse vara att göra byggandet av stora IT-system praktiskt genomförbara och möjliga att göra underhåll på. Problemet med Cobol var programmerarna som använde språket. Många av dem använder eller missbrukar i dag C++ och Java och ”legacy” problemen som plågat Cobol kommer säkert att uppträda även med dessa nya språk. *Turbo Pascal*s bidrag var inte så mycket språket självt som att det var den första kommersiella integrerade programmeringsmiljön (editor-kompilator-debugger), vilket banade väg för effektiva kod-fokuserade och inkrementella utvecklingsansatser, som dom som används hos Microsoft och även förordas av företrädare för Extreme Programming.

Visual Basic möjliggjorde på mycket kort tid (endast Java har spritt sig snabbare) att en kommersiell marknad för komponenter uppstod, vilket akademiker och forskare tidigare utan större framgång diskuterat i tiotals år. Visual Basic är begränsat, men inom sina gränser har det haft djupgående betydelse och har gjort meningsfylld programutveckling tillgänglig för en mycket större grupp människor än vad som eljest skulle varit möjligt. Här måste Microsoft få ett erkännande.

Capability Maturity Model for Software d v s SEIs SW-CMM, är en av få ”varumärkta” metoder som har påverkat organisation av programvaruutveckling. Det finns mer än 1000 organisationer och mer än 5000 projekt som har blivt evaluerade mot CMM. Ändå anser en av experterna att CMM är både en ”bästa inflytande” och en ”dead end” för programvaruutvecklingen. De finns mycket spel för gallerierna och missbruk av CMM, som lett till bortkastade och kontraproduktiva verksamheter. En av de oväntade konsekvenserna av CMM har blivit att göra kunskapsmassan inom Software Engineering mycket grundare. I stället för att se till att utbilda berörda i t ex ”risk management”, eller ta hjälp av specialister, har man tagit fyra sidor ur CMM-dokumentationen och stuvat om dem till några sidor i sin lokala processdokumentation. Totalt anser dock samme person att CMM varit till nytta, men att det finns begränsningar i hur mycket den kan hjälpa oss in i det nya århundradet. Det är inte CMM som sådan, utan behovet av standardiserade metoder som är det viktiga, enligt ett par av de andra experterna. Användning av SPI-ramverk (Software Process Improvement), vare sig det är CMM eller ISO 9000, ger struktur och koherens åt SPI-aktiviteter. En annan anser att SPI hör till bästa-listan, inkl både ISO 9000-3 och CMM tre första nivåer, men anser inte att CMM som sådan hör till bästa-listan. Ytterligare en åsikt är att trots sin enkelhet, är det största problemet med CMM den rigida och okunniga tolkning som den ofta ges. Själva konceptet är egentligen en truism. Det viktigaste som CMM åstadkommit är dess framgång som ett marknadsföringskoncept från SEI. Det har verkligen ökat medvetenheten inom tusentals företag om bra Software Engineering principer, liksom hos universiteten. Däremot håller inte denne person med om att ISO 9000-3 skulle vara värd samma plats som CMM på bästa-listan. Med ISO 9000-3 kan man göra alla de felaktiga sakerna och fortfarande uppfylla kraven bara man gör dem systematiskt (”diligent”).

Slutligen framhåller en annan av experterna att CMM definitivt är den standard som ”sets the pace”. F n är det det ramverk som tillåter ”benchmarking” och som bidrar mest till att lösa programvarukrisen, helt enkelt genom att definiera en standardiserad terminolgi och att kondensera industrins ”best practicies”. CMM är denne persons personliga favorit på bästa-listan.

Object-Oriented Programming lovade stora fördelar med ”naturlig” design och programmering, men de som praktiserade detta, lämnades ibland, efter det att den inledande ”hypen” ebbat ut, med progarmvaruteknologier som ökade komplexiteten, gav endast marginella prduktivitetsvinster, gav kod som inte kunde underhållas och endast kunde användas av experter. Det verkliga värdet av OOP är förmodligen inte objekten själva, utan förmågan att aggregera programmeringskoncept till större ”klumpar” än subrutiner eller funktioner. Detta är säkert en av de starkaste influenserna inom Software Engineering, men är det en av de bästa? Åsikten från en av experterna var att vad än de verkliga fördelarna som kom från OO var, så kom de huvudsakligen från användning av den traditionella Parnasstilen för ”information hiding” och inte från OOs all övriga ”utsmyckningar”. OO ansågs vara en ”badly mixed” metafor som försöker spela på den mänskliga hjärnans visuella förmåga att organisera objekt i rymden, men som sedan nästan omedelbart omvandlar detta till en ruggig mängd syntaxkonstruktioner som inte mappar väl med hjärnans lingvistiska förmåga. Resultatet är en ”ugly mix” som vanligen varken är särskilt intiutiv eller så kraftfull som det ursprungligen reklamerades med. En annan av experterna instämmer i att många inte tagit sig tid att sätta sig in i OO ordentligt innan de börjat designa. På kort sikt har OO skapat mer problem än s k dålig Cobol-programmering. En annan av experterna ifrågasätter om både Cobol och OO inte hellre borde stå på ”10 sämsta” i stället för ”10 bästa” listan och anser att sväl Cobol som andra procedurella hög-nivå språk kan beskyllas för att ha skapat de flesta nuvarande programmeringsproblemen. Det är så lätt att programmera i sådana språk att även noviser kan skriva stora program, men resultatet är ofta dåligt strukturerat. Med ”riktiga” OO-språk som Smalltalk, anstränger sig programmerarna att lära sig att programmemra rätt och resultatet är välstrukturerade program. Att ha programmeringsansatser som bara kan användas av experter kanske inte är så dumt. Vår tidigare attityd att vilja ha ”easy-going” programmering är en av huvudorsakerna till den nuvarande programvarukrisen, inkl Y2K-porblemet. En annan av experterna instämmer betr OO-stil enligt Smalltalk och är optimistisk om Java och anser att den

verkliga katastrofen var C++, som blandade metaforer på ett verkligen hemskt sätt och uppmuntrade gamla C-programerare att skapa klasser med några hundra metoder i sig för att göra systemet ”objektorienterat”.

Component Based Development är man överens om är en av de viktigaste företeelserna inför det nya århundradet, men man verkar anse att mycket arbete återstår innan de utlovade fördelarna kan realiseras, om de ens någonsin kan det. Hittills har detta fungerat bättre i labben än i verkligheten. Oförenligheter mellan komponentversioner har t ex lett till massiva uppsättningsproblem, opredikterbar påverkan mellan program, avinstallationsproblem och ett behov av hjälpprogram som återställer alla komponenterna i en dator till senast kända korrekta status. Men man pekar också på andra företeelser som kan hjälpa komponentbaserad utveckling och återanvändning på traven, eller möjligen kringgå dem, som t ex ”patterns” och domängeneralisering. En annan av experterna tycker dock att det, bortsett från kompilatorkonstruktion, varit för få framsteg på komponentvägen och framhåller att den största succen på senare tid att bygga generellt användbar programvara, har varit ERP (Enterprise Resource Planning) system som SAPs, vilka hanterar en hel domän. En annan åsikt är dock att även om komponentvägen gett mycket huvudvärk, så är det ingen vedertagen ”dead-end”. Om man har en i förväg tillräckligt verifierad miljö, så visar ju Visual Basic att den kan fungera. Däremot ligger nog den fullt generiska ”vad som helst från vem som helst” modellen bortom ”state-of-the art”. Dock påpekar en annna av experterna att arbetet på komponenter har visat vägen mot ramverk, återanvändning och patterns och således varit ett nyttigt mellansteg. Att tänka i komponenter är fortfarande en bra designprincip och helt i linje med t ex att praktisera ”information hiding”.

Metrics and Measurement anses av experterna både kunna bli en ”dead-end” och höra till det som kan ge bäst hopp inför det nya århundradet. Det största problemet är att detta begrepp är dåligt förstått och därför används på fel sätt. Att kunna uttrycka förhållanden i siffor är en väsentlig fråga i alla ingenjörsdiscipliner. Hur skall man kunna tala om ”software engineering” utan mätningar. En annan anser att problemet inte ligger i metriken utan hos de som arbetar med metrik, som har ”messed up” när de använt den. En annan kommentar är att felet är att vi försöker tillämpa mätmetoder som fungerat bra för tillverkning, på en så kreativ process som programvaruutveckling. Det liknar att försöka lära folk att skriva som Shakespeare genom att hela tiden kolla stavfel. Sant, håller en anna expert med om, men metrik har gjort stora framsteg de senaste 10 åren och är av stort värde för styrningen av grupper och projekt. Det är lika nödvändigt med en metrikdisciplin i software engineering som på andra ingenjörsområden om man skall kunna undvika utvecklingsansatser som bygger på annat än ”gut feeling”.

I slutsammanfattningen sägs (av Maarten Boasson som nog många SESAM-iter känner till) att det största problemet som programvaruområdet har, är tendensen att tro att en ny idé skall lösa alla problem. Det leder till att tidigare lösta problem glöms bort så snart en ny idé börjar få stöd och att som konsekvens man börjar lösa om problemen i den nya stilen. Detta är ett säkert recept för att hindra framsteg och är enligt Boassons åsikt en följd av att Software Engineering är ett så omoget område. Jämfört med andra områden har vi knappt kommit ur dagis-stadiet. Det som vi dock kollektivt får ta på oss, är den religiösa fanatism med vilken vi försvarar vår nuvarande tro mot upplevda attacker från dem som gör saker annorlunda. Det finns ingen enskild teknik som passar till att lösa alla komplexa problem som programvara behöver användas till. Det Boasson anser fordras, är först och främst att företagsledare inser att software engineering inte är en ingenjörsdisciplin som andra, och att standarder, metoder och verktyg alla förmodligen är felaktiga (när vi väl förstått vad det innebär att utveckla programvara). Detta betyder att programvaruutveckling fortfarande är mycket mera en konst än vad företagsledningarna tror och att därför endast de ”best and brightest” skall tillåtas utveckla system av mer än trivial komplexitet. En bieffekt av detta skulle bli en signifikant minskning av de insatser som behövs för deras utveckling. För det andra borde det göras riktiga experiment med nya och alternativa ansatser att utveckla programvara, möjligen i samarbete mellan industri och akademi. Här skall det nte vara fråga om några förenklade typiska akademiska problem, utan verlikga ”real-life” system. Dessa måste då utföras parallellt med traditionell utveckling av samma system, eftersom ingen projektledare skulle acceptera risken i den alternativa systemutvecklingen (även om Boasson tror att den i verkligheten skulle vara mindre). Resultaten skulle publiceras och analyseras öppet för alla, oavsett vilka de skulle bli. För det tredje skulle man ompröva meritsystemet för akademiker. Den nuvarande ”publish or perish” paradigmen garanterar att stora mängder skräp publiceras. För det fjärde bör IEEE kontinuerligt understryka att

problemen inte ännu har lösts, att innovation är väsentlig och att alla anspråk på att ha uppfunnit LÖSNINGEN är fel.

***

Det blir ofta en strid flod av insändare efter potentiellt kontroversiella artiklar i IEEE Software, så det kan vara anledning att läsa kommande nummer, för att se vilka reaktioner denna artikel väcker.

I Carlsson hade läst IEEE Software

Thought is the basis for action

Human thought is unlimited since there is no limit to what we can imagine. At the same time there are some limits to human thinking since we cannot keep to many things in mind at the same time. Anything we do or build must start with a thought, as must the building process itself - thought is the basis for action. We can transform thoughts into words and pictures to communicate our thought to others. This ability is the key to human cooperation and also to more powerful thinking when a group of humans can be made to cooperate in activities such as brainstorming or reviewing.

What has this quasi-philosophical reasoning to do with systems engineering? Quite a lot, since technical systems and also complex systems, where humans cooperate with technical hardware and software entities are all based on the human thought. Consequently, it is all-important to us as systems engineers to take care of human thinking and to make sure that the thinking process for individuals and groups involved in the process or creating and updating systems is as efficient as possible. This is a real challenge for those of us who try to provide and use systems engineering methods and tools.

Some aspects of support for the all-important human thought are:

KISS = Keep It Simple Stupid
Systems engineers and system end-users are not stupid. But as any human, we have a limited capacity to keep different things in mind at the same time. This limited capacity should be used as far as possible to really investigate and understand ”the system” and not to try remembering the syntactic details of the method and tool used for work with the system. This means that it can be a good idea to look for methods and tools, which are based on a single and simple model for description of system structure and behavior. Also to beware of those methods and tools, which has grown syntactically like an unpruned bush for reasons such as keeping the different method architects friendly to each other.

Support communication
It is good to see a genius at work. However those of us, who have been able to work with a genius have often been able to find flaws in the genial work - if we were able to understand it. This means that anything, created as a result of human thought, must be expressed and communicated in a way that is fully understandable to everyone involved in a systems creation process in order to take advantage of the group’s total mental capacity.

Understand the combination of real and ”canned” thoughts.
Since software and hardware entities are all based on thoughts, they can be seen as ”canned” thoughts. What happens when these entities are included in a system together with operators is that these ”canned” thoughts need to interact with the operators’ real dynamic and original thinking. This interaction can be tricky, since it is impossible for anyone who creates the ”canned” thoughts to foresee what the operator might think when encountering the software or hardware. The conclusion is that anyone who creates systems with person-machine interfaces must not only foresee the possible action space of the operator but also limit the operators’ actions, with respect to the system, to what is meaningful in cooperation with the ”canned thoughts” included in the system. To do this is simpler if you have tools, which allow you to model the technical and human system parts in one context.

Understand the mission and concentrate on it

As engineers we are trained to investigate and put together technical detail. That is good, but we also tend to forget to clarify a system's ultimate mission since "everyone knows the mission". That is of course true although the people involved may have widely differing opinions of what the mission might be. The conclusion is that any system description should include a clear statement of mission with a description of how the system's components contribute to completion of the mission.

Simplification

As humans we can understand a complex system on a simple overview level (helicopter view) OR at detailed level (technicians view). Systems engineering requires both views and also a clear conviction that they are really consistent. Since the ability to keep both views in mind is beyond human capacity (for complex systems), this represent one of the basic challenges for tool-builders.

Finally: Thought is the basis of all human action, consequently also the basis for systems engineering. Thinking can be done more efficiently if you can combine the thinking of several minds through efficient communication. Good systems engineering methods and tools can assist to overcome the problem of managing overview and complex detail at the same time. And make sure you make everything as simple as possible, but no simpler!

Ingmar Ögren
iog@toolforsystems.com

En för redaktionen ny webb-plats för Ada är "AdaPower, Developer Resource and Tools" http://www.adapower.com/

Vi citerar ur deras "about" not:

After years of working with C, C++ and Pascal, finding Ada was a dream come true. Ada posses the ultimate in flexibility (oo and non-oo), real standardization and validation, true cross platform programming, incredible compile time error checking, readable code, and support of all levels of software engineering. As a way of contributing back to the Ada community and to help advocate this powerful language AdaPower.com was formed.

- Examples of Ada source code illustrating various features of the language and programming techniques
- Examples of Ada source code illustrating various interfaces to popular Operating Systems (Thick and Thin Level Bindings)
- Examples of Ada source code illustrating various algorithms
- A collection of packages for re-use in Ada programs
- Articles on implementing software in Ada
- Home to some of Ada's most inovative GPL projects
- The most complete collection of links to Ada resources on the Net

To contribute source code to the Treasury use the article submission form or e-mail AdaSource@AdaPower.Com

Handbok för Programvara i säkerhetskritiska tillämpningar (H ProgSäk)

FMV har sedan 1997 i olika etapper arbetat på en handbok i programvarusäkerhet att användas vid anskaffning av programvara i säkerhetskritiska tillämpningar.

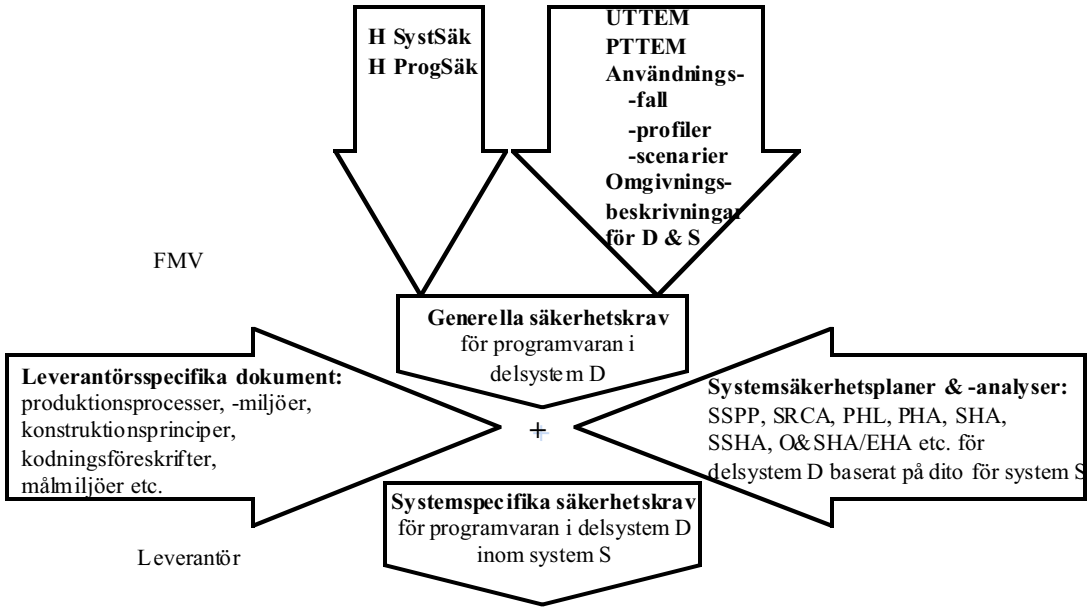
Handboken bygger på den systemsäkerhetsverksamhet som fastläggs i H SystSäk. Samma huvudprinciper för riskreducering gäller, de flesta tekniker för säkerhetsanalys är också gemensamma. H ProgSäk visar hur dessa tekniker kan tillämpas på programvara, vilka övriga, speciellt programvaruinriktade metoder och verktyg som är aktuella, hur koppling mellan traditionell programvaruteknik, systemsäkerhet och tillförlitlighet kan åstadkommas. Den beskriver även möjligheter och problem vid realisering i programvara jämfört med en mer renodlat hårdvarubaserad lösning. En översikt av standarder och handledningar inom system- och programvarusäkerhet ges med korta referat, jämförande tabeller, begreppsskiftningar mellan standarderna osv. Checklistor, begrepp, referenser till övrig information inom området ingår.

En ny utgåva föreligger sedan 1999-10-10 (v 2.1). Arbetet har efter detta datum legat nere fram till årsskiftet, då medel för slutförandet av resterande etapp 3 och 4 tilldelades.

Nuvarande version innehåller i princip samtliga generella säkerhetskrav, som kan ställas på programvara (dvs på programvaruprodukt, använda processer och produktionsmiljöer samt personer verksamma inom dessa) under hela livscykeln.

Kraven är oberoende av applikationsdomän samt numrerade och graderade efter kritikalitet (strängare krav för delar av högre kritikalitet).

Anpassning till enskilt projekt och system är därför nödvändig. Krav relevanta för aktuell tillämpning väljs ut. Dessa kompletteras med systemspecifika säkerhetskrav baserade på resultat från fördjupade säkerhetsanalyser ned på programvarunivå.



Säkerhetskrav för D:s programvara i system S

Under etapp 3, efter prov i några projekt, kompletteras handboken med fler anvisningar och exempel, förslag till innehåll i systemsäkerhetsplan m a p programvara osv. En slutversion planeras till 00-09-20. Projektet avslutas 01-08-20 i o m etapp 4 och översättning, utlägg på web, utbildning m m.

Inga-Lill Bratteby-Ribbing
AOL H ProgSäk, FMV

On principles for model-based systems engineering

Ingmar Ogren, Tofs AB, Fridhem 2, SE-76040 Veddoe Sweden
e-mail iog@toolforsystems.com

1. Abstract

This paper addresses the problem of consolidating technical descriptions of how a system is built with operational descriptions of the missions the system shall complete (how the system is to be used).
It also discusses how a central model constituted from design objects with requirements, test cases, problems and documents as attributes, to these design objects, can support modern principles for "incremental acquisition" and "incremental development".
Modeling principles, based on entity-relationship diagrams and the UML (Unified Modeling Language) component diagram, combined with pseudo code behavioral descriptions, are described as means to build the "central model".
After a "central model" for systems engineering is established ,it is shown how the model can be extended into a "Common Project Model", being common in two ways:

- *Common for "real implementations" and simulators required for the system.*
- *Common for all concerned stakeholders such as acquirers and contractors.*

Application of the "Common Project Modeling" principle, with computer-stored models, holds promises for increased system quality and for more efficient systems engineering.

2. Background with today's solutions

Leonardo daVinci once made some fantastic designs. From these you can understand that technical drawing, in that time, was little different from art. Later technical drawing diverged from art with the introduction of dimensional measurement, different views, and so on. This tradition of technical drawing has developed into qualified CAD drawing systems. Technical drawing is now an excellent means of describing the physical properties of any item, but it gives little understanding of how the item drawn should be used or which missions it can contribute to complete.

When electricity arrived, new drawing techniques were needed, resulting in the electric

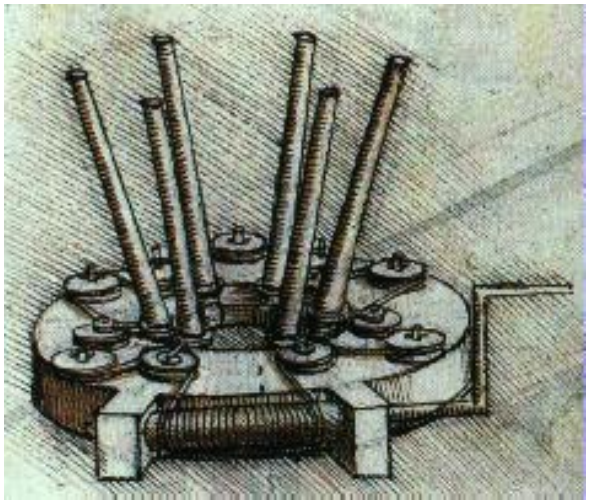


Figure 1 Leonardo DaVinci drawing

schema showing electric components, voltages and currents. Since electric equipment was often complex, the block schema was also introduced to show a higher "structural" level of the electric system. These schemas still concentrate on the system's components, giving little understanding of the system's missions.

With the advent of computer software, it was believed that many problems would be solved through the simplicity of changing the software. Rather soon this simplicity of change proved to be more of a problem than a solution and the need to describe software exactly was understood. One solution was "Structured Analysis" diagrams, derived from the earlier "block schemas". These give a good understanding of the software's structure, but still little information about the software's missions. During the last ten years object-oriented software descriptions techniques have become wide spread, mainly used as a means to support economic software reuse.

Modeling is a well-proven technique for technical research and development. Ships, buildings, airplanes etc. have been modeled for purposes, such as hydro- and aerodynamical research, usability investigation, visualization for end-users, etc. Modeling of software-intensive systems, using the Unified Modeling Language (UML) [1], introduced by Rational Inc, is now also possible. Models represent an excellent way to visualize one

or more aspects of a system, but most models of complex systems still have problems in clarifying the system's missions. Below will be explained how a system can first be modeled in its context, after which the system's missions can be identified and included in the model. First however, a discussion of the development process structure.

3. The three basic processes in Systems Engineering

For software and systems engineering the combination of "waterfall" and "big bang" used to be popular. "Waterfall" then means that system development is visualized and planned as a number of time-separated phases, the main phases being analysis, design and verification. "Big bang" means that development is planned and executed as a single effort going through the phases from requirements' investigation to integration. The combination means that you plan system development as a single large concerted effort, composed from a number of phases, to be gone through, one at a time, separated by reviews. The principle is attractive for several reasons:

- It is simple to explain.
- It is orderly, logical and can be visualized in a single viewgraph.
- It is well suited to traditional acquisition with fixed price.
- It often offers an attractive time schedule, when presented in proposals.

However there is a small problem with the combination of "Waterfall" and "Big bang" for development and evolution of non-trivial systems, since the resulting methodology does not comply with reality. It simply does not work for reasons such as:

- It is not humanly possible to specify a complex system completely and correctly prior to development, since development will always build new knowledge.
- Problems will always surface during development and some of these will cause late changes of requirements.
- In reality, the activities in the "phases" are more concurrent than sequential, making it impossible to put them into a sequential schema.
- It is difficult and often impossible to know the real requirements, with their priorities, until end-users have had an opportunity to acquaint themselves with the final system or at least with a realistic representation (model) of the system.
- It is difficult (impossible) to know the "cost/ contribution to mission" ratio for each system feature before getting rather far into development.

For these reasons several new development models have been defined for software engineering, such as "spiral"[2] and "ball-bearing"[3] models, with a better understanding of the need for concurrency. Several of these introduce new problems concerning acquisition as it becomes painfully obvious that you must understand that the principle of "fixed price contract" is of little use in complex system acquisition, when the requirements are not really known until you are well into development. One principle that gives promises to both manage the concurrency needed in systems/software engineering and to allow fixed price contracts is "Progressive acquisition" with incremental development[4].

A version of incremental development is visualized in Figure 2. The technique is characterized by:

- "Requirements Management", "Development" and "Verification with Test" are established as three concurrent processes.
- Successive releases of the system are produced, giving the developers and end-users "something real" to work with as soon as the first version is released.

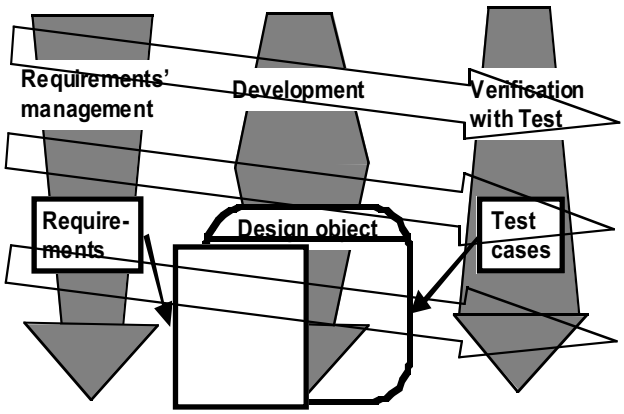


Figure 2 Parallel processes in incremental development

- Requirements Management", includes analysis and this process is more labor intensive in the beginning of the project where most of the requirements' work still needs to be done.
- "Development" includes architectural and detailed design. This process is most labor intensive in the middle of the project.
- "Verification with Test" starts early in the project with verification of initial requirements, but is most labor intensive by the end of the project, with testing in connection with system integration and deployment.
- The three processes are kept together by a central design object structure with requirements and test cases being attributes to the objects.

The principles of incremental acquisition/development are now being introduced in some large system user organizations within the WEAG (The Western Europe Armament Group).

4. The central model

A problem with modern development models with concurrency and incrementality is that they tend to be confusing to Quality Assurance people. They don't find their traditional baselines and "critical reviews" and feel they are getting lost in a multitude of activities and versions.

This is a serious problem, which however can be made less serious through introduction of a central model to base the project on. There are several ways to model and it is essential to decide on modeling technique for a project.

4.1 How do you know what you model?

When you review a software or systems engineering diagram, you often come across simple entities such as for example "aircraft position". You can ask the diagram author what this means: "Is it really the aircraft position or is it the computer's understanding of the position?" The question may cause some confusion and most often the answer will be something like "It is this entry in the data dictionary, represented by that floating point data". If you then put the next question: "How do you know it is the real position?" you may get a clear, crisp and understandable answer. You may also get a confusing discussion of data, communication paths and delays throughout the system, which leaves you with little understanding of how well the data represents its counterpart in the real world.

In these cases it helps to draw a UOD (Universe Of Discourse) diagram. This is a simple way to increase knowledge of how entities in a system represent and connect to entities in the "real world".

To draw a UOD diagram, start with an entity in the real world, such as an aircraft:

Figure 3 A single entity

Next, you can introduce a radar to detect the aircraft together with a couple of relations between the radar and the aircraft. The relations are drawn "both ways" to show that this is not a Data Flow Diagram, but an Entity-Relationship diagram, which simply defines entities and relations. To read and understand the diagram, you simply read the text in one box together with the text along an arrow and the text in box the arrow points at. When you review a UOD diagram, you read these simple texts and check that they are both readable and say something meaningful about the system.

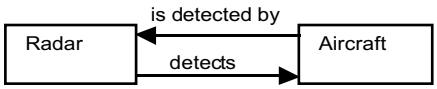


Figure 4 Two entities with relations

If you then want to build an air traffic control system you need to represent the aircraft in the system:

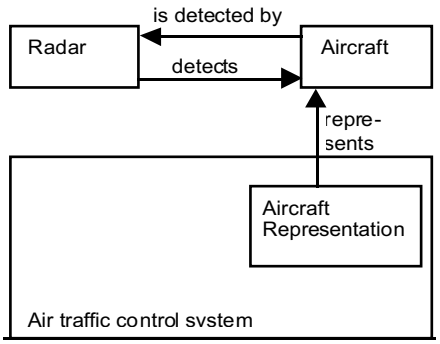


Figure 5 Representation entity added

What you have done so far is simply to add to the original diagram to show that radar is used to detect aircraft and that aircraft must be represented in air

traffic control systems. This may seem completely trivial, but establishment of basic facts like these may well be of crucial importance in other and more complex circumstances.

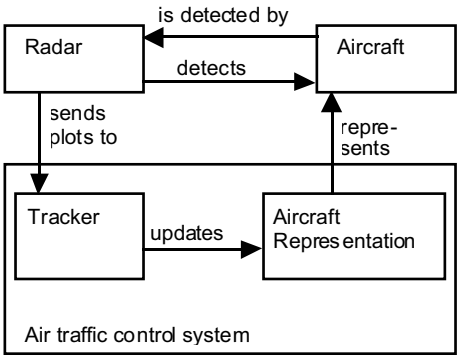


Figure 6 Diagram with "double coupling"

However, the diagram says nothing about how to build the "Aircraft Representation". For the example is presupposed that "Aircraft Representation" is built by a "Tracking" entity, which gets information from the radar. The entity "Tracker" is introduced, with its relations in Figure 6.

You now have a simple UOD diagram with "double coupling". The diagram shows an entity in the environment or "real world" (Aircraft) and its representation in a system (Aircraft representation). The double coupling means that the diagram shows how the environmental entity is represented in the system and how the environmental entity influences its representation.

The diagram also expresses a number of simple facts about the system in its environment if you read the text in each two connected boxes together with the arrow text. If these sentences don't make sense or are not grammatically correct, the diagram probably needs some further work.

What you have done now is basically modeling on two levels. The diagram is a model of a system in its environment and the diagram shows one aspect of how the system's environment is modeled within the system.

Note that "environment" is not necessarily the "real physical" environment. For example an embedded software system may well have other software systems as its environment!

UOD diagrams can be drawn simply with paper and pencil or on a blackboard and this is often an excellent idea, particularly early in system analysis, when you want to build an understanding of an existing or future system. Drawing these diagrams together with an experienced end-user on a blackboard is a very good way to understand and document basic facts.

However remember that what you are drawing is entities and their relations, not a data flow diagram and don't make the diagram too complex. Multiple small simple diagrams are better than one big complex diagram, since it will be difficult to see the errors in a complex structure.

Tooling is an issue for the UOD-graphs. The blackboard is a wonderful tool, but it has its limitations as a means for persistent information storage. Computer storage is better and many simple drawing programs, such as PowerPoint or Visio can be used to draw and store UOD diagrams. You can also use other programs with drawing capacity, such CAD or CASE programs.

However, before you select a program to document and store your models, check that it does not have any awkward syntactical limitations and that it can do useful tricks such "rubberbanding" and "snapping".

4.2 Requirements on modeling and modeling alternatives

From the above discussion it is obvious that modeling is central to achieve quality in complex systems. There are many ways to model a system and you may wonder which one to choose. The answer is very simple: It depends. It depends on which aspect of your system you want to model and who shall read your model. Another answer is that you cannot really choose, since you need to master a palette of modeling techniques to cover the needs during a systems' engineering effort. You must consider what is required for modeling a complex system, to achieve an acceptable quality. Five key requirements are:

1. Determinism with formality
This means that everything expressed in the model must have a single, defined and obvious meaning.

2. Understandability
Since systems engineering should be done in close cooperation with end users, the models used must be readily understood, without extensive education or experience in software or mathematics

3. Inclusion of system missions
The model must elicit the system's missions and also be able to express how different parts of the system contribute to completion of these missions.

4. Modeling of structure and behavior
The modeling technique shall support splitting a system into subsystems, with clarification of interfaces between these systems, and the modeling technique shall also allow definition of behavior within the subsystems defined.

5. Possibility of verification support
It shall be possible to verify a completed model. This verification can be against defined requirements, but it can also concern verification of completeness, consistency, etc. For complex systems, verification will often require computer support, depending on the large amounts of information to be managed.

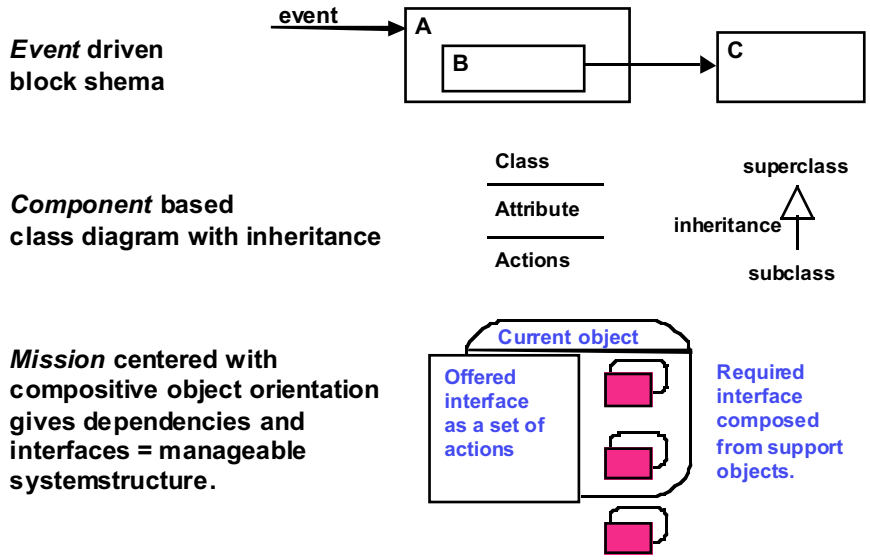


Figure 7 Three principles for modeling

4.3 Some modeling alternatives

Below some useful techniques for modeling are discussed and three of them are also shown in Figure 7.

The Block diagram
The block diagram may be the oldest way to model systems. It is very simple to understand and it can include events as shown in Figure 7. The example shows that A contains B and that B transfers something to C. The block diagram is extensively used for hardware schemata, for organization diagrams and for software structuring (as Data Flow and Context diagrams)

The UML Class Diagram
The UML (Unified Modeling Language) [1] contains a "Class Diagram", which concentrates on a system's components with class inheritance, dependency, association, aggregation and cardinality. The rich syntax and the great power of expression are obvious advantages for the UML Class diagram.

The rich syntax may also be a disadvantage since it is easy to draw complex and confusing diagrams when you use the full syntax. A good idea, particularly when you work with end-users, is to limit each diagram to a subset of the Class diagram syntax.

The UML Component diagram
Another UML diagram is the Component diagram. It was published in Grady Booch's book on Software Engineering from 1983 [5] and it has been developed in the HOOD software development method [6].

The UML component diagram is useful, since it can be used to model compositive object structures. When you work with such structures, you concentrate on each object's interfaces and on dependencies between objects, rather than on "inheritance" between objects.

The diagram in Figure 8 shows that:

- The current object has an offered interface (constituted from a set of actions, which can be invoked from outside the object)
- The current object has a required interface, constituted from parts of the offered interfaces of the support objects
- Two support objects are contained in the same system as the current object, while one support object is outside of that system.

The Component diagram allows you to model not only hardware and software components as objects, but also operator roles and missions. This makes it possible to model complex systems as a set of diagrams on different levels, with clear dependencies among the objects and with a clear understanding of how the different objects contribute to completion of the system's missions.

	Determinism	Understan-dability	Mission inclusion	Structure/ behavior	Verification support
Block diagram	No	Very good	No	Structure	Poor
UML Class diagram	Yes	Difficult	No	Structure	Poor
UML component diagram	Yes	Good	Can be	Structure	Can be good
State charts	Poor unless formalized	Good	Can be	Behavior	Good if formalized
Pseudo code	Yes	Requires explanation	Can be	Behavior	Good if formalized

Table Modeling techniques versus requirements

State Charts
State charts show behavior for a system component as a set of states with transition conditions and transitions between those states. Consequently, they can be used for modeling of behavior.

Pseudo Code
Pseudo Code is a code-like behavior description, constituted from code-like formal control structures, variables (parameters, messages and local variables) of defined types and comments.

Although the modeling techniques discussed here are only a small subset of the available techniques, it is still useful to compare these techniques with the requirements listed in section 4.1. What you need to model a complete complex system is at least one structural modeling technique and one behavioral modeling technique. The requirements and the modeling techniques are listed in Table 1. One choice (highlighted in Table 1), used for the continued discussion, is to start out from the UML Component Diagram and combine it with Pseudo Code.

This yields a central model, which defines system structure as a set of objects, depending on each other and connected through defined interfaces. Within each object its behavior is modeled as pseudo code in a set of actions.

In order to be able to manage the complete development effort attributes are added to the objects and used to manage requirements, test cases, problems and documentation.

4.4 The air traffic control example

In the Air traffic control example used in section 4.1 the initial UOD diagrams result in an understanding that one of the missions in this system is to manage aircraft information. Consequently an object "Manage_Aircraft_Info" is identified with sub-missions to present the aircraft information, to measure course and speed and to calculate collision risks.

Figure 8 Object graph example from the Air traffic control example

Figure 8 shows the example in a modified UML component diagram, drawn with the Tofs toolkit [3].
The behavior description of the single action "Invoke_aircraft_management" in Pseudo Code will then contain one single concur statement to invoke actions in the three sub-mission objects in parallel

```
object Manage_Aircraft_Info is


action Invoke_aircraft_management is
visibility: Offered
purpose: {Invoke concurrent actions in mission
objects to complete the mission of managing the
aircraft information.}

begin
   concur
      # Calculate_collision_risk. Calculate_risks

      # Measure_course_and_speed.
      Course_speed_measure

      # Present_aircraft.
      Present_aircraft_information
   end concur
end

end Manage_Aircraft_Info
```

5. Use the model in the Three Basic Processes

The three basic processes in systems engineering are separate, but still connected through the central model. Below the content of the three processes and how they can be supported by the system model is discussed.

5.1 The Requirements management process

What to do

The requirements management process aims at creation and maintenance of an understanding of the requirements for the current project through the complete project. The requirements must be as complete as possible and they should comply with any constraints concerning, for example, scheduling and cost.

The need to maintain and optimize requirements makes it impossible to have the requirements process "done with" in the beginning of the project.

How to do it

The first thing to do to get the requirements correct is to find, understand and document the mission(s) of the system to be updated or created. After the missions are defined two things can be done in parallel: Define and assign requirements to the missions and create a draft system structure.

After this is done you have a basis to start the design and verification processes while the requirements process continues with addition of new requirements resulting from build-up of knowledge, adjustment of requirements as a result of problem management and distribution of requirements to design objects.

How the model supports requirements management

As soon as you have the missions and a draft top-level design, you can identify a first set of objects in the model. The requirements, problems, etc. can then be assigned as attributes to these objects. The result is that the model supports an orderly management of requirements and other pieces of information, which pertain to requirements management.

5.2 The Development process

What to do

The development process contains architectural and detailed design, expressed as a structure of connected objects. Each object will then contain detailed information to be used as a basis for implementation of that object.

How to do it

After you have the missions and the top-level requirements, you can apply top-down and bottom-up principles for design:

- Top-down through definition of new support objects to the mission objects, with continuation of the process downwards with distribution of requirements to the objects.
- Bottom-up through identification of reusable support objects with insertion of these in the designed structure and distribution of requirements to the objects found.

How the model supports design

During design the model is updated to include the new objects, defined or found, with their dependencies and interfaces. Consequently the model grows during design, and provided that your model is formal and computer-stored, it will support consistency checks of the design.

5.3 The Verification process with test

What to do

Verification includes verification of correctness for requirements, design and also for the completed and integrated system in its application environment. The fact that verification must be applied already on the first set of requirements makes it necessary to start the verification process in parallel with the other processes.

How to do it

Verification is done through reviews and tests on

various levels and concerning various parts of the system under development or update. On the top (mission) level the system must be validated against scenarios, covering the missions defined.

How the model supports verification
When you have a computer-stored object model of a design, this model can support review work through presenting the design line-by-line for inspection and through automatic analysis of the designed structure.

Such a model further supports testing through allowing you to define test cases and test results as attributes to the objects in the design.

6. Managing the issue of criticality

Systems may be critical in different ways, for example safety-critical, mission-critical or environment-critical. For critical systems, you need an extremely low probability of failure. To achieve this it is helpful to have:

- A formal system description to allow for automatic checks on consistency and completeness.
- A system description which is understandable to the system's end-users, since these are the real experts on the system's applications.
- Fault-tolerance, since you must always allow for component failures and human mistakes in the completed system.

A model-based approach, as described above, helps to manage critical systems, since the model will support the necessary analysis activities in several ways:

- The formalized structural and behavioral system description gives the necessary basis for criticality analysis.
- Providing behavior is expressed in simple state charts or "English-like" pseudo code, the behavior should at least be explainable to end-users.
- The model gives an excellent basis for fault-tolerance analysis, since the model includes the dependency structure necessary for application of traditional analysis techniques, such as Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA).

7. Tools for modeling

Complex system are distinguished by the fact that it is not humanly possible to overview the system and at the same time keep an understanding of all the details of the system. When working with such systems, it is obvious that, provided that you have the ambition to find and retrieve the information, the amount of information exceeds what is possible to keep in mind or to manage as "paperwork". This is where a computer-stored model can assist. Stored in a suitable tool, such a model will assist in inputting and retrieving the large amount of information needed for work with complex systems.

One such tool, which supports the combination of component diagrams and formal pseudo code behavioral descriptions, is Tofs (Tool For Systems) [3]. For the discussion below Tofs is used for the example. A Tofs screen, with part of the Air Traffic Control example, is shown in Figure 9.

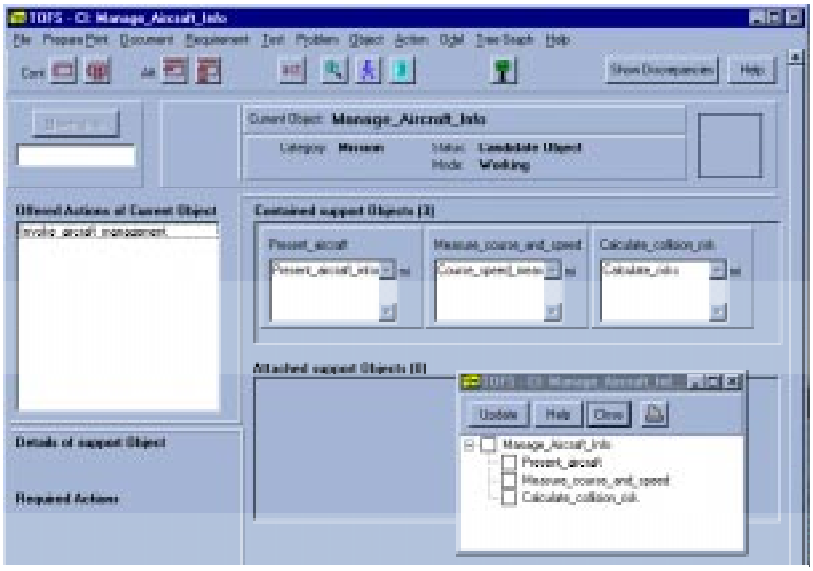


Figure 9 Tofs screen with component diagram for the object "Manage Aircraft info"

8. The Common Project Model (CPM)

8.1 The quality problem

Quality for complex systems concerns compliance between system performance and expectations. These expectations take different form for different stakeholders. For example:

- A simulator user expects the simulator to include a correct representation of the simulated system and its environment
- A system end-user expects the system to comply with his or her original specifications and with its documentation
- A system maintainer expects a system to be delivered with a complete and understandable documentation, which complies with the system.

All these expectations concern the fact that a complex system encompasses not only the system itself, but also a set of models such as simulators for different purposes, models included in the maintenance documentation and mental models maintained in the minds of developers, end-users and maintainers.

It is obvious that whenever one of these models deviates from the real system, a risk is introduced that one of the stakeholders has expectations, which deviate from the system's reality. This results in a quality problem.

8.2 Commonality Acquirer/contractor

Each stakeholder and participant in a project has the right to expect an understandable description of the part of the system he or she is concerned with. This description shall include not only design information, but also valid requirements, test cases, etc. For a partial system description, to be of acceptable quality, for a project participant, it must be possible to show that it is part of a consistent description of the complete system. As a complex system, by its very nature, is not completely understandable for one person at one time it is not an easy task to achieve the desired description quality.

Descriptions of complex systems present a number of problems:

- Modern complex systems are composed from multiple subsystems, which must cooperate in order to complete missions.
- Subsystems are delivered from different vendors and are typically utilized by end users in separate organizations.
- Systems operate in a complex environment, in which external systems may influence mission results.
- Systems will normally exist in multiple releases and may be supported by several simulators, each providing a model of the system.
- Legacy and COTS (GOTS) parts must be integrated into the system with full understanding of how they are interfaced and of how they contribute to completion of the system's mission
- The different vendors and end users, concerned with a complex system will each have their own standard and tradition for system descriptions.

The problem aspects can be summarized as follows: It is crucial to project success and quality that a common understanding covering all system releases and simulators be established between all of the involved parties.

8.3 Why a Common Project Model

As discussed above, a quality problem will surface whenever multiple expectations and models are present in connection with a complex system. It is obvious that it would be possible to diminish these problems if everyone concerned with a complex system could work from a common model to get a common understanding of the system and consequently also common expectations.

8.4 What is a Common Project Model?

A Common Project Model (CPM) is a description of a system's structure and behavior, expressed in a

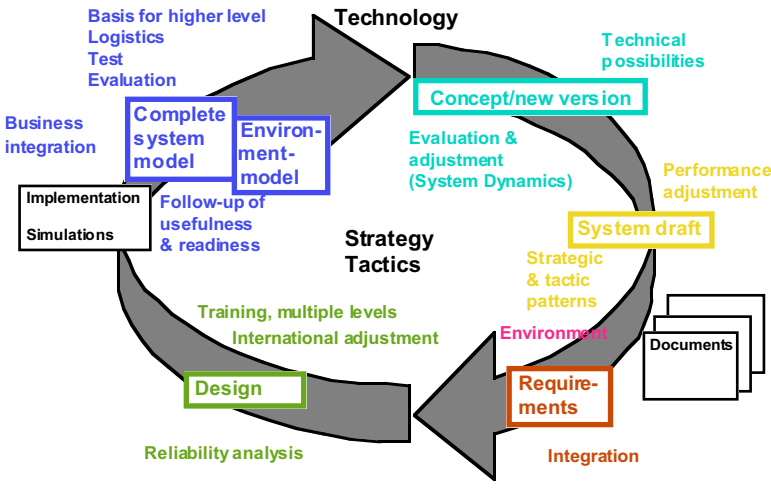


Figure 10 The Common Project Model as a "Project backbone"

way that manages the problems listed in section 8.2. For any non-trivial and complex system it will be necessary to have the model computer-stored to manage and analyze the large amount of information required. Since a complex system will include a lot of documentation on parts of the system, it will be necessary for the model to include references to various pieces of documentation.

8.5 How to build a Common Project Model

To build a CPM, you should start when the project is in its concept stage. The model can then grow together with the project through analysis, design, implementation and commission. It is also possible to build a CPM for an existing project and consequently create the necessary common understanding from existing documentation and from the stakeholders' knowledge and expectations.

To build the model, you can use any qualified systems engineering tool, which includes acceptable modeling principles as discussed above. As an example is shown, in Figure 9 a Tofs screen picture of the initial structure for the "Manage Aircraft info" of the Air Traffic Control example.

Note that a tool to manage Common Project Models must include sub-tools to manage requirements, test cases and documentation as shown in the menu bar in Figure 9.

8.6 How to use a Common Project Model

When a CPM is built through a project it can be seen as "project backbone" as visualized in Figure 10.

The CPM can be used to support a variety of important activities to raise the overall system quality, for example:

- During development, the model helps to establish a common understanding of the system's environment and the environmental requirements on the system.
- During marketing of new system releases the CPM helps to make it possible to model new environments and to study how the system can be tailored to meet the requirements from these.
- In subcontracting, the CPM helps to ensure a common understanding among the contractors involved.
- In system maintenance the CPM helps to provide the necessary understanding of how the different parts of the system contribute to completion of the system's missions.

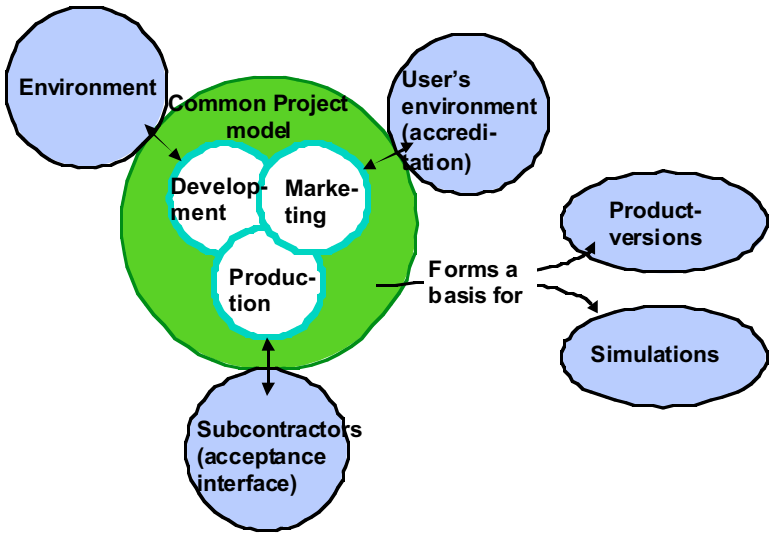


Figure 11 Use of the Common project Model

9. Experiences

The modeling principles, described in this paper, have been applied in multiple projects in Sweden, primarily in defense and industrial applications. Some experiences are:

Application of parallel processes in incremental development.

Particularly early introduction of reviews, for verification of requirements has proven to be an efficient way towards early detection of requirements-related problems.

Inclusion of Mission objects in system structures.

Early identification of system missions and inclusion of these missions in the system structure has proven valuable to build a common understanding of the objectives for a system development among the stakeholders concerned.

Use of Universe Of Discourse diagrams

The Universe Of Discourse diagrams have proven to be valuable to clarify a system's

interfaces with understanding of how the system represents its environment.

Use of UML Component diagrams (Object graphs) to structure models

These diagrams are not as easily understood as traditional "block schemas", why many developers are hesitant to use them. On the other hand the component diagrams have proven their usefulness to build system models to connect a system's misisons with all its components in a single consistent structure.

Use of Pseudo code to formalize behavioral descriptions

End users are hesitant to read pseudo code although it has been proven possible and useful to explain pseudo code to end users. Consequently pseudo code represents a useful compromise between informal natural language an d formal mathematical notations.

Use of Common Project Models

The idea of a Common Project Model (CPM) originated during development of Saab's PMSIM (Presentation and Control simulator) [7] and was to some extent applied in that project. No major CPM has yet been built but the principles have attracted interest form multiple major aerospace industries and the Defense Material Board in Sweden why a project for investigation of CPMs is under way.

10. Conclusions

1. The problem of modeling a system to show, not only its technical structure, but also how the system's components contribute to completion of the system's missions can be solved through extended application of UML component diagrams.
2. The present trend towards incremental acquisition and development, requires parallel processes for requirements management, development and verification. It is possible to achieve the necessary common basis for the three processes with an object based system model.
3. Modeling of system behavior with pseudo code gives a practically useful compromise between informal natural language and mathematical formalism.
4. Introduction of "Common Project Models" is a promising technique to increase system level quality and efficiency in evolution of complex systems.

11. The author

Ingmar Ogren was graduated with an M SC in Electronics from the Royal University of Technology in Stockholm in 1966.

He then worked with the Swedish Defense Material Administration and various consulting companies until 1989 with systems engineering tasks in areas such as Communications, Aircraft and Command & Control.

He now chairs the board and is owning partner in two companies: Tofs which produces and markets the Tofs (Tool For Systems) software and Romet, which consults in the area of systems engineering methods with the method O4S (Objects For Systems) as its main product.

Further information about Ingmar Ogren can be found on the web page http://www.toolforsystems.com

12. References

[1] UML, The Unified Modeling Language, information available from http://www.rational.com

[2] Information about the spiral model, originated by Dr. Professor Barry Boehm can be found at and downloaded from http://sunset.usc.edu/WinWin/winwin.html

[3]Information about the ball-bearing model for system evolution, as well as the Tofs toolkit software, can be downloaded from http://www.toolforsystems.com

[4] Guidance of the Use of Progressive Acquisition, © 1997 WEAG TA-13 & Contributing Companies

[5] Grady Booch: Software Engineering with Ada, Benjamin Cummings 1983

[6] HOOD, An Industrial Approach for Software Design, Jean-Pierre Rosen, Hood Technical Group 1997

[7] Saab PMSIM, Information available from http://www.saab.se

En kort presentation av ”nya” Saab

Nya sammanslagningen av Saab och Celsius satsar på fem affärsområden:

Infomatics,
Aerospace,
Dynamics,
Technical support and services
Space.

Totalt antal anställda ca 17 000. British Aerospace (BaE) äger idag 35 % av Saab koncernen. Koncernchef är Bengt Halse.

Affärsområdet **Infomatics** omfattar lednings- och informationssystem, telekrig, avionik, sensorer och signaturanpsanpassning, simulering och träning samt radarbaserad nivåmätning. I Infomatics återfinns bl.a. de gamla företagen CelsiusTech, Ericsson Saab Avionics, Saab Dynamics (verksamheten i Jönköping och Göteborg), Saab Training Systems, Barracuda Technologies, Combitech Network, Celsius Communication, Saab Celsius TransponderTech, Celsius Automotive m.fl.

Affärsområde **Aerospace** omfattar Gripen, framtida flygande system och delsystem till den civila flygindustrin.

Affärsområde **Dynamics** omfattar missilsystem för flyg, land och marin, bärbara pansarvärnssystem och undervattenssystem. Här återfinns bl.a. företagen Bofors Missiles, Taurus Systems GmbH, Bofors Anti Armour System, Bofors Underwater, Saab Dynamics (verksamheten i Göteborgoch Linköping), Bofors AB.

Affärsområdet **Technical Support and Services** omfattar tekniktjänster, drift och underhåll samt systemleveranser. Vidare underhåll och modifiering av militära och civila flygplan samt helikoptrar och målflyg. Här återfinns bl.a. AerotechTelub.

Affärsområdet **Space** omfattar ombordsdatorer, antenner och mikrovågsteknik samt separationssystem. Här återfinns bl.a. Saab Ericsson Space.

Det sista affärsområdet **Aviation Services** omfattar kommersiellt flygunderhåll samt motor- och komponentunderhåll och utgörs av Celsius Aviation Sevices.

Ett antal företag ingår inte i ovan nämnda affärsområden bl.a. Bofors Weapon System, Saab Aircraft m.fl.

Billy Johansson

AiS/SESAM VÅRSEMINARIUM DEN 13 APRIL

09.30 -10.00	Registrering och kaffe
10.00 -10.05	Välkommen, Örjan Leringe, AiS ordförande
10.05 -11.00	Korpen flyger, Lars Asplund
11.00 -12.00	High-integrity development with SPARK, Rod Chapman
12.00 -13.00	Lunch
13.15 -14.00	Proof more cost-effective than testing?Industrial experience with SPARK..., Rod Chapman
14.00 -14.30	Debate SPARK, Ravencar, Ada and safety...
14.30 -14.50	Kaffe
14.50 -15.50	Ada, Model Railroading, and Software Engineering Education, John McCormick
16.00	Ada i Sverige årsmöte

Anmälan fyll i formulär på www.ada-i-sverige.se . Senast 5 april!

Du besöker väl vår websajt?

http://sesam.tranet.fmv.se

SESAM möten

30 mars 2000	VU
12 april 2000	Rådet
25 maj 2000	VU
25 aug 2000	VU
5 okt 2000	VU
20 okt 2000	Rådet
8 dec 2000	VU

Bidrag till Rendezvous och SESAM hemsida efterlyses

Varken Rendezvous eller hemsidan blir bättre än vad medlemmarna gör den till. Håll ögon och öron öppna för intressanta händelser i eller nära SESAM-världen och skicka in en notis till sekretariatet, alkas@tranet.fmv.se. När det är aktuellt att kunna publicera rapporter eller beskrivningar från projekt som Ni är inblandade i, glöm då inte våra egna medier.

SESAM-Sekretariatet: AerotechTelub AB
c/o Kåsjös Kontor
Ytterspåret 14
187 54 TÄBY

Telefon: 08-510 51866
Telefax: 08-510 51932
GSM: 070-716 9702
E-post: alkas@tranet.fmv.se