SISU analys

Nr. 4

ADA-teknologi

SISU – Svenska Institutet för Systemutveckling

Box 1250
163 13 SPÅNGA

Kistagången 26
KISTA

08-750 75 00

ADA-
TEKNOLOGI

ISSN: 0282-9924

INNEHÅLLSFÖRTECKNING

FÖRORD

SISU-analys är en tidskrift som utges fyra gånger per år. Varje nummer behandlar ett tema - ett aktuellt problemområde och söker då ge en bild av området ur olika aspekter.

Detta nummer av SISU-analys har ett tema som berör de flesta som arbetar med professionell utveckling av system där programvara ingår som väsentlig komponent, nämligen Ada teknologin. Programspråket Ada och den teknologi som utvecklats däromkring har som bakgrund alla de problem och svårigheter som uppstår när man försöker utveckla stora programsystem.

Vi försöker i detta nummer att ge en beskrivning av vad det är som Ada har och som andra språk inte har eller bara har delvist och hur detta påverkar systemutvecklingen. Därutöver redovisar vi erfarenheter av användning av språket och erfarenheter av utbildning i Ada.

Författare till detta nummer har varit:

Kapitel 1-4: Sven Tafvelin Chalmers tekniska högskola

Kapitel 5: Christopher Bengtsson, Försvaret Materielverk

Kapitel 6: Sven Tafvelin CTH, Gunnar Boström TeleLOGIC SUNDSVALL AB, Margaretha Cedercrantz TeleLOGIC AB

Kapitel 7: Jonas Agerberg

Kapitel 8: Jan Skansholm CTH

SISU tackar alla medverkande för deras bidrag.

SAMMANFATTNING

Under det sista decenniet har begreppet "mjukvarukris" myntats. Konventionell ingenjörsverksamhet och utvecklingsprojekt bedrivs rutinmässigt och håller nästan alltid snäva tids- och resursplaner. Det är helt självklart. Inom programvaruområdet är situationen annorlunda. Många programvaruprojekt spräcker såväl tidsramarna som kostnadsramarna flagrant och detta utan att det inom branschen vållar någon som helst förvåning. De programvaruprodukter som erhålls innehåller många gånger felaktigheter och ibland löser de inte ens de problems som beställaren ville ha lösta. Till detta kommer underhållskostnader som ibland är mycket stora och som kommer som en överraskning för många parter.

Mjukvarukrisen har man försökt angripa från flera olika håll. En av dessa metoder är att utforma ett så bra programspråk som möjligt. Där försöker man dra nytta av alla accepterade ideer om hur ett bra programspråk skall se ut och hur effektiv systemutveckling och underhåll skall ske. Ada är ett sådant språk. När ett bra språk har tagits fram är det naturligt att använda det på bästa sätt. Vi får en Ada teknologi.

Detta nummer av SISU-analys beskriver ett antal av de fördelar som Ada har som programspråk. Det är fördelar som andra programspråk saknar eller bara har delar av. Betydelsen av dessa egenskaper beskrivs liksom de effekter som kommer att erhållas. De viktigaste effekterna är att när man arbetar i Ada, slipper man arbeta med flera programspråk, och eftersom Ada är hårt standardiserat med procedurer som genomdriver denna standard så blir det lättare att flytta tillämpningar, programdelar och personal mellan olika system och projekt. Språket har utformats så att de fel som alla programmerare gör skall upptäckas så tidigt som möjligt. Tidig felupptäckt leder till att felrättningskostnaden minskar och att programmerarnas produktivitet ökar och att de levererade systemen innehåller färre fel.

Ett viktigt resultat som man kan förutse är att eftersom språket är hårt standardiserat blir det naturligt för programvaruproducenter att utveckla produkter som stöder utvecklingsprocessen. Språket saknar dialekter, produkterna kan skrivas i Ada och blir då flyttbara till många maskiner och det kommer att finnas en stor och köpkraftig marknad. Av många bedömare anses det här som en av de viktigaste Ada-effekterna.

Försvaret är en stor köpare av programvara och är därför naturligt mycket intresserade av de fördelar som det bedömer att Ada kommer att ge. Resonemangen där redovisas.

Vi försöker dessutom förmedla erfarenheter över hur det är att använda Ada i olika tillämpningar. Där begränsar vi oss inte till enbart tillämpningar där datorer styr system (vilket var Adas officiella utvecklingsmål) utan beskriver ett flertal tillämpningar där Ada har använts i mer administrativt betonade system.

Slutligen redovisas erfarenheterna av att undervisa i Ada i såväl industrimiljö som i högskolemiljö. Erfarenheterna är entydiga. Ada är ett lätt språk att lära ut. Eleverna upplever att det uppfyller de behov som de har och att språket gör det på ett naturligt sätt.

1. BAKGRUND

Det finns många företag och organisationer i Sverige där produktion av programsystem är en väsentlig, ja ibland till och med en dominerande verksamhet. Från att ha varit en aktivitet av marginell betydelse, har programsystemutveckling blivit alltmer betydelsefull. Det är nödvändigt att den sker med bästa industriella metoder. Målsättningen måste även här, liksom i all annan ingenjörsverksamhet, vara att framställa önskat system inom avtalad tid och inom avtalade kostnadsramar. Denna ingenjörsmässiga attityd betonas i engelskspråkiga länder genom att man kallar området för "Software Engineering". Den svenska betckningen "programvaruteknik" saknar helt nyanserna av ingenjörsteknik.

Utveckling av system och utveckling av program och i synnerhet utveckling av programsystem har under årens lopp vållat många bekymmer. Det är självklart att utformning och bygge av exempelvis stora broar för stora belopp sker inom sina tidsramar och inom budget. Bygge av fabriker med inredning och allt som hör till sker inom sin givna tidsram och inom budget. Däremot är det vanligt att utveckling av stora programsystem inte håller sina tidsplaner och sker med stora budgetöverskridanden. Det inträffar så ofta att det inte ens vållar någon förvåning. Till yttermeravisso innehåller programsystemen förvånansvärt många fel eller funktioner, som användarna ville ha annorlunda. Framtiden är bekymmersam. Produktiviteten när det gäller utveckling av avancerade system är låg och ökar inte i paritet med att systembehoven och möjligheterna ökar. Vi kommer i framtiden att behöva allt fler programmerare, och någonstans når vi en gräns, när det inte är möjligt att öka arbetsstyrkan ytterligare.

Allt detta är kända observationer och brukar ofta sammanfattas under benämningen "mjukvarukrisen".

Situationen förvärras ytterligare av att allt fler system hamnar i situationer, där felaktigheter inte tolereras, och att systemens livslängder blir allt större. Livstidskostnader för ett system domineras idag nästan alltid av underhållskostnaderna efter att systemet tagits i drift första gången. För många väsentliga system är den ursprungliga utvecklingskostnaden enbart en mindre del av livstidskostnaden.

Erfarenheterna och forskningsresultaten inom programutvecklingsområdet hade lett till vissa accepterade åsikter om hur programspråken bör se ut och hur arbetet skall ske. Det blev med tiden allt större och större diskrepans mellan kunnandet om hur språken skulle se ut och hur de såg ut i verkligheten.

Allt detta var kända saker för det amerikanska försvarsdepartementet (DoD) i mitten av 1970-talet. Deras situation försvårades ytterligare av att väldigt mycket av deras kvalificerade programvara var skriven i olika assemblerspråk, många olika assemblerspråk. Enbart kravet att hålla kompetens i så många språk, som de hade, drev kostnaderna i höjden. Speciellt såg man att problemen var störst inom realtidssystemområdet. Det var uppenbart nödvändigt att gå från assemblerspråken till något högnivåspråk. Under andra halvan av 1970-talet formulerade man en följd av dokument som allt noggrannare beskrev de krav som man borde ställa på ett modernt högnivåspråk för området. Därefter jämförde man kraven med de språk som redan fanns, som Pascal, Fortran, Jovial osv. Utvärderingen visade dock entydigt, att det inte

fanns något språk som uppfyllde de krav som ställts, och skillnaderna var så stora och kraven så höga, att det fanns skäl att utveckla ett nytt språk. Det bästa förslaget på nytt språk som DoD erhöll kom från Honeywell. Det kan vara intressant att veta att Ada faktiskt till stora delar är en europeisk produkt. Det är en fortsättning på en god tradition. Såväl Algol, Pascal, Prolog, som Modula är utformade i Europa.

Utformningen av Ada gjordes med stor hänsyn tagen till problemen med programtillförlitlighet, enkelhet att underhålla skrivna system, effektivitet och enkelhet för användarna. Man har strävat efter att göra ett enkelt språk, så enkelt som det går att göra med de komplexa och långtgående krav som ställdes på språket. Vid utformningen har man prioriterat läsbarhet framför ett koncist skrivsätt. Det är naturligt eftersom programmen kommer att läsas och ändras i många fler gånger än de skrivs.

Vi skall å andra sidan komma ihåg att Ada är ett programspråk och att det inte har målsättningen att lösa alla problem inom systemutvecklingsområdet. Ada är ett konventionellt, imperativt programspråk. Det är ett ingenjörsmässigt klart steg framåt, ett steg som man kan lita på att det kommer att fungera. Där skiljer det sig från andra försök att lösa systemutvecklingsproblemen med nya språk. Där tar man mycket större kliv. Det gäller exempelvis logikprogrammeringsspråk och funktionella språk. Där har man i gengäld oron kvar över om de verkligen kommer att bli användbara för större volymer av program.

Hur betydelsefulla är språk? Språk är ett hjälpmedel man använder för att representera och uttrycka sina ideer och konstruktioner. Programspråk är bara ett exempel. Det finns också språk för systembeskrivningar (SDL exempelvis), liksom det finns språk som används för att rita ritningar med. Även om språken bara är redskap för att uttrycka ideer och tankar, så är de ändå mycket betydelsefulla. Våra tankar styrs ofta av det språk som representerar dem. Vår begränsade förmåga att hantera komplexitet medför, att med kraftfullare språk kan vi hantera mera komplexa system. På programspråksområdet har det medfört, att det är rimligt att tala om personers produktion i form av antal korrekta programtextrader per månad oavsett språk. Med ett mer avancerat språk får man fler problem lösta på dessa rader.

2. EGENSKAPER AV BETYDELSE HOS SPRÅKET

Ada och Software Engineering (SE) hör samman. Ada utformades så sent att det fanns mycken erfarenhet av systemutveckling, och det fanns goda exempel på hur programspråk inte skulle se ut. Det fanns också exempel på språkdetaljer som fått god och accepterad utformning. All den erfarenhet som fanns utnyttjades och därför är det ingen tillfällighet att Ada passar särskilt bra för utveckling av professionell högkvalitetsprogramvara. Ada har utformats för att passa i krävande sammanhang med utveckling och vidmakthållande av stora, besvärliga system. Det är i detta sammanhang som Ada skall jämföras med andra språk. Det finns enklare, leksaksbetonade, språk som ser fina ut, när de behandlar sina förenklade problem, men de kommer alltid till korta, när kraven stiger.

Det är uppenbart att Ada är resultatet av en 25-årig evolution och att det inte har som målsättning att ställa till med revolution.

Vad har Ada som inte de flesta andra språk har?

- Stark typning
- Modularisering
- Separatkompilering med full säkerhet
- Generiska moduler
- Processbegreppet
- Felhantering
- Lågnivåfaciliteter
- Klar definition med validering av kompilatorer

2.1 Stark typning

En av målsättningarna med programspråk är att det skall vara lätt att instruera kompilatorn hur dataobjekt logiskt skall se ut och hur de skall användas. En annan målsättning är att höja kompilatorernas felupptäckande förmåga. Det skall vara så liten risk som möjligt att ett felaktigt program skall gå genom kompilatorn utan felutskrifter. Det är detsamma som att säga att det skall finnas redundans i språket. Under hela den tid som det har funnits programspråk har de blivit allt starkare inom detta område. Ett sätt som positivt påverkar båda områdena är språkens typning. Allt fler och fler språk förstärks i sin typning och Ada tillhör klassen av starkt typade språk. Det är en orsakerna till att Adakompilatorer har så god felutpptäcktsförmåga. En erfarenhet från användning av Ada är att program som passerat kompilatorn också har hyfsad chans att också vara logiskt felfritt.

Många äldre språk och kompilatorer lät separatkompilering verkligen ske separat utan att göra kontroller mellan modulerna. Därefter användes en normal, "dum" länkare som inte bryr sig om annat än att länka samman efter namnlikhet. Typen för objekten bortsåg den glatt ifrån. Så är det inte när det gäller Ada. Där gäller stark typningen inte enbart inom en kompilering utan hela programsystemet kontrolleras.

2.2 Modularisering

Ada stimulerar till att program delas upp i vettiga moduler. En moduls specifikation är klart frikopplad från dess implementering. Följaktligen kan man enkelt i ett senare skeda förändra modulers realisering om bara deras specifikation förblir oförändrad.

En modul, eller ett paket som det kallas i Ada, kan användas för att hålla samman objekt med olika ambitionsnivåer. Ett paket kan exempelvis innehålla:

- en samling deklarationer

- en samling data och operationer

- en samling skyddade data och operationer på dessa

- en abstrakt datatyp

Det har ofta varit en knäckfråga var pakets egna data skall placeras. I Ada har man insett att det är nödvändigt att tillåta både fallet med att data hålls internt i paketet eller att de placeras externt hos användaren av paketet.

Den moderna utvecklingen inom programutvecklingsmetodiken har betonat betydelsen av abstraktioner. Ada som ett modernt språk uppfyller dessa behov.

Frikopplingen av pakets specifikation från deras realisering medför också att det är möjligt att kompilera dem vid olika tidpunkter. En naturlig användning av detta är att tidigt i systemutvecklingen bestämma moduluppdelning och snitten för dessa. Detta skrivs ner som Ada specifikationer och kompileras. Därefter kan skrivningen av kropparna börja och det kan ske utan att programmerarna behöver bekymra sig om i vilken ordningsföljd modulerna skrivs. Alla specifikationer finns ju redan.

2.3 Separatkompilering med full säkerhet

Ett stort system måste alltid delas upp i flera kompileringsenheter. Det är nödvändigt för att programmeraren skall kunna arbeta med hanterbara enheter. Ada stöder detta fullt ut. Kompilatorn "tittar" i alla relevanta moduler vid kompileringen och kontrollerar specifikationerna mot användningen. Vi får därför full säkerhet även vid separatkompilering.

2.4 Generiska moduler

Moderna programspråk skall vara starkt typade. Det förbättra väsentligt felupptäcktsmöjligheterna vid kompileringen. Å andra sidan vållar stark typning svårigheter i vissa situationer. Det blir omöjligt att skriva generella program som fungerar på flera olika datatyper. Ada löser detta genom **generiska moduler**. Det ger möjligheten att skriva generella programmoduler där vissa aspekter lämnas odefinierade. Senare när modulen skall användas kan den preciseras (anpassas) till det aktuella användningsområdet.

2.5 Processbegreppet

Processbegreppet är ett viktigt redskap, när man skall strukturera problem och utnyttja de parallellismmöjligheter, som datorsystem har. Med tiden har man fått en klar uppfattning om hur processbegreppet i ett modernt programspråk skall se ut, och Ada är utformat därefter. Vi kan definiera enkla processer, ange hur de kan påverkas av andra processer och ge dess algoritm, dvs programtext, som den skall följa. I och med deklarationen skapas en process som upfyller beskrivningen. Detta är

det enkla och naturliga fallet.

När vi har behov av mera dynamik i vårt system, finns möjligheten att deklarera upp en specifikation av en process och ge den programkod som skall följas. Vi har då fått en processtyp. Det kan jämföras med att man kan deklarera datatyper separat. När vi har en processtyp, kan vi deklarera upp ett antal objekt av denna typ, och då får vi processer som exekverar. Vi kan till och med dynamiskt generera processer på samma sätt som vi hämtar dynamiskt minne.

De vanligaste fallen kommer naturligvis att vara de statiskt deklarerade processerna. Det finns naturligtvis ingen orsak att ta till mera dynamiska metoder, än vad problemställningen kräver. Å andra sidan sett är det skönt, att när problemställningen kräver dynamik även bland skapandet av processer, så har språket möjligheterna.

Processer avslutas vanligtvis genom att de har exekverat färdigt och går ut genom sitt slut. Det hela sker på ett konsistent sätt, så att ifall den har några underprocesser så väntas dessa in. Det finns besvärligare situationer också, där flera processer väntar på nya uppdrag, eller också kan de tänka sig att sluta. Vanliga programspråk, som har parallella processer, har inte tänkt på detta fall. I Ada har man tänkt på detta också. Där är det möjligt att ange, att man väntar på nytt uppdrag, men att ifall "alla andra" runt omkring ligger i samma läge eller vill sluta, så slutar vi gemensamt som en grupp.

Processer måste kunna kommunicera sinsemellan. I Ada finns primitiver för detta ändamål. Två processer möts, de gör en rendezvous, och kan då utbyta data i båda riktningarna. När mötet är klart skiljs man åt och arbetar var och en på sitt håll. Det finns goda möjligheter att ange vilka rendezvous-er man är intresserad av att genomföra.

I professionell programvara är det nödvändigt att gardera sig mot problem med andra i systemet. Därför kan någon eller båda processerna gardera sig genom att sätta en övre tidsgräns (timeout), som de är villiga att vänta.

2.6 Felhantering

Felhantering är ett besvärligt område när det gäller programmering och det gäller i synnerhet programmering av professionella system. Där kan man inte hoppa över denna problematik. Det är synd att alltför många programspråk lämnar oss i sticket!

I Ada är det möjligt att skriva kodpartier som skall genomlöpas när olika fel uppträder. Detta kallas för **exception**. Det finns såväl språkdefinierade som användardefinierade exceptions. Det sätt som exceptions har införts i språket medför att de samverkar utmärkt med abstraktionstankarna, som varit grundläggande vid konstruktionen av Ada. Ifall vi har ett kodparti (block eller kropp), som implementerar en abstraktion, så sätter vi relevanta exceptions så att de tillhör partiet. Ifall fel inträffar längre ner får de kontrollen, och vi kan lätt

lämna ett svar utåt, som är relevant med tanke på vår abstraktion.

2.7 Lågnivåfaciliteter

Utveckling av komplicerade, realistiska system leder ibland till att det är nödvändigt att lämna den rena, fina kodningen. Men även "smutsig" kod måste kunna skrivas i ett professionellt programspråk. I Ada är det exempelvis möjligt att ange

- Representation av datatyper
- Placering av vissa objekt som
    + Hårdvaruregister
    + In/utmatningsregister
    + Avbrottsrutiner
- Generering av vissa maskininstruktioner
- Anrop av icke-Ada subprogram
- Otillåten typkonvertering

2.8 Klar definition med validering av kompilatorer

Ada utformades genom en mycket medveten process. Man visste då väl om alla svårigheter beträffande dialekter och delmängdskompilatorer som andra språk drabbats av. Eftersom programmeraren naturligt anpassar sig efter den dialekt och den kompilator som han/hon har tillgång till, får man program som bara begränsat är flyttbara.

Detta skulle inte få hända Ada. Dialekter och delmängdskompilatorer har man förhindrat på flera sätt:

- Den vanligaste orsaken till dialekter är att språket inte är fullständigt. Det finns behov som det inte löser. Så är inte fallet med Ada. Det har gjorts komplett, vilket är en av orsakerna till att det betraktas som stort. Det skall inte finnas behov av utvidgningar. Detta kan kontrasteras mot exempelvis Pascals brist på separatkompilering.
- Språket är väl definierat, även om semantiken saknar formell definition. Därför uppstår sällan frågan hos kompilatorbyggare om hur (kombinationer av) konstruktioner skall tolkas. När det uppstår, finns en organisation som avgör frågan.
- När en Ada-kompilator är klar måste den **valideras** för att få kallas för en Ada-kompilator. Validering innebär att ett stort antal program (idag ca 2500) skall kompileras och i vissa fall exekveras med resultat som är korrekta. På detta sätt kontrolleras att kompilatorn realiserar Ada, hela Ada och inget annat än Ada.

Fördelarna med denna process är att alla kompilatorer tolkar språket likadant och att alla program ger samma resultat. Detta leder i sin tur till att Ada-program blir flyttbara mellan kompilatorer och mellan datorsystem. Adas faciliteter för paketering tillsammans med flyttbarheten leder till att återanvändning av programvara äntligen kommer att vara möjlig i större skala. Detta gör det också möjligt att grunda en komponentindustri inom programvaruområdet.

Självklart leder friheten från dialekter och subsetkompilatorer till att programmerarnas flyttbarhet ökar. Det blir lättare att börja arbete på ett nytt projekt, på en ny avdelning eller på ett nytt företag. På analogt sätt blir en nyanställd programmerare fortare produktiv.

3. EFFEKTER AV ADAS EGENSKAPER

Hur manifesterar sig Adas goda egenskaper i praktiskt arbete? Det sker på många sätt.

Ada har stark typkontroll som sträcker sig över separatkompileringsgränserna. Detta leder till tidig felupptäckt. Tidig felupptäckt leder till billig felrättning och snabbare till ett fungerande system.

Ada är lättläst. Vid utformningen av språket har läsbarheten prioriterats framför koncist skrivsätt. Därför blir systemen lättare att förstå i efterhand. Följaktligen kan underhållsarbete ske snabbare och med större tillförlitlighet. Livscykelkostnaderna sjunker.

Ada klarar av all programmering; även de delar där man förr var tvungen att använda assembler. Det är en stor fördel att slippa kunna flera språk och blanda dessa. Upplärningsbehoven minskar, felkällorna vid språkbytet försvinner, hjälpmedlen blir bättre när man kan koncentrera sig på ett språk i stället för att splittra utvecklingen på flera osv.

Som tidigare förklarats är Ada program mer portabla än program skrivna i andra språk. Detta skapar möjlighet för modulåterbruk i en omfattning som varit omöjlig förut. Återbruk av programvara kommer troligen att vara den kraftigaste produktivitetshöjande faktorn för framtiden.

Portabiliteten av Ada moduler skapar också möjligheter för programvarukomponentindustri. Det kan bli lönsamt att specialisera sig på att utveckla moduler för ett visst område och sälja dessa som komponenter till andra programvaruproducerande företag.

4. UTGÅNGSPUNKT FÖR EFFEKTIVA HJÄLPMEDEL

Alla behöver vi hjälpmedel för att arbeta. Inom systemutvecklingsområdet är det normalt datoriserade hjälpmedel för informationsbehandling som åsyftas. Avsikten med hjälpmedel är att öka vår produktivitet, minska felfrekvensen, hjälpa oss att lagra och återvinna information eller att kontrollera att vi följer de metoder som skall användas. Samlingen av hjälpmedel brukar ofta kallas för programutvecklingssystem eller ibland för programutvecklingsmiljö.

Utveckling av system är fortfarande i många avseenden oftast en manuell process. Det är märkligt med tanke på den stora volym utvecklingsverksamheten har och den lönenivå som personalen har. Investeringarna per anställd i "normal" produktion är mycket större än motsvarande vid utveckling av system. De investeringar, som systemutvecklarna kan dra nytta av, är framförallt på hjälpmedelssidan. Även om systemutveckling även i framtiden kommer att vara ett hantverk, är det väsentligt, att "hantverkarna" har effektiva hjälpmedel som gör dem produktiva.

Förutom produktiviteten påverkar även hjälpmedlen sättet att arbeta. De får en styrande effekt. Om det är bekvämast och effektivast att följa företagsstandard, kommer den att följas, utan att piska behöver tillgripas.

Ett exempel här kan vara en Ada-kunnig editor. Om den är det bekvämaste vektyget, när det gäller att arbeta på Ada text, så kommer alla program att se lika ut eftersom utseendet i detaljer bestäms av editorn. Därför kommer de som arbetar med programmet i efterhand att känna igen sig och kunna arbeta snabbare och säkrare. Att en sådan editor dessutom höjer produktiviteten och minskar felfrekvensen är naturligtvis gynnsamt det också.

Ada som programspråk ger av många orsaker en bra grund att bygga redskap på.

Det har definierats ett mellanspråk, att använda när Ada kompileras, som kallas för Diana. Det är en lämplig nivå för många redskap att arbeta från. Kompileringen från Ada till Diana medför fullständig syntaxanalys. Programmet i Dianaform är syntaktiskt korrekt och är analyserat men kan ändå rekonstrueras textmässigt.

Ada är väl definierat även om det inte har en formellt definierad semantik. Det uppfyller en av grundförutsättningarna för att det skall vara möjligt att utforma redskap som analyserar Ada-program och Ada-programsystem. Det kommer att i viss utsträckning vara möjligt att skriva program som studerar ett system och ger utsagor om hur det fungerar. Typiska frågor som kan besvaras i viss omfattning är:

- Hur är systemets huvudstruktur?

- Vilka processer och processtyper finns i systemet?

- Hur kommunicerar processerna med varandra?

- Finns det risk för låsning (deadlock)?

- ...

Självklart kan inte alla frågor besvaras helt och fullt ut. Programbevisning är inte lättare för Ada-program än för program i andra språk. Ibland är det rimligt att redskapen ger partialsvar och det är naturligtvis bättre än inga svar alls, vilket är dagens läge för de flesta språken.

Analysverktygen kommer att vara nyttiga när man skriver program naturligtvis. De kommer att vara ännu nyttigare när det är skrivet. Då kan man se ifall det följer de krav som ursprungligen ställdes i specifikationen. Den största nyttan kommer de att ha i efterhand, när man förändrar systemet (underhåll). Verktyg, som beskriver hur ett system ser ut, är då mycket nyttiga. Man kan aldrig vara helt säker på att eventuell dokumentation överensstämmer med verkligheten. Dessutom är det oftast onödigt att dokumentera sådant som verktyg kan extrahera ur textmassan.

Ada standardiserades på en gång och valideringsproceduren för kompilatorerna medför att de alla följer denna standard. Vi kommer att slippa problemen med dialekter och delmängdskompilatorer. Eftersom alla Ada-program därför följer samma språkdefinition, får redskap som arbetar på programtexterna stort användningsområde. Det blir ekonomiskt rimligt och lockande att utveckla redskap för Ada.

En följd av acceptansen av Ada är att man även börjar använda det som bas för språk att beskriva program och deras algoritmer. Den typen av språk brukar kallas för "Program Description Languages (PDL)". Ada har varit utgångspunkten för ett antal sådana språk.

Många tillämpningar där Ada används är mycket kritiska beträffande felfunktioner. Stora ansträngningar måste göras för att undvika att programmen innehåller felaktigheter. Eftersom programmerare bara är människor, är det lätt hänt vid manuella processer som progrmmering att av misstag föra in fel. Verktyg, som hjälper till med att minska felfrekvensen, är därför viktiga och nödvändiga.

Ada kommer att användas i professionella programutvecklingsmiljöer i företag, som utvecklar system professsionellt. Dessa företag vet, att produktivitet är av mycket stor betydelse för deras resultat, och de är beredda att investera i hjälpmedel så fort effektiva sådana finns. Produktivitetshöjande verktyg kommer att användas.

Allt detta leder till att det kring Ada kommer att finnas ett stort utbud av goda redskap. Det är en följd av språkets egna kvaliteter och den omgivning som det kommer att arbeta inom.

5. VARFÖR GÅR FÖRSVARET IN FÖR ADA?

(av Christopher Bengtsson, FMV)

Sommaren 1985 utfärdade Försvarets materielverk (FMV) interna anvisningar för dels Software engineering och dels användning av programspråket Ada i tillämpningar främst för inneslutna system och realtidssystem. Anvisningarna gäller för all anskaffning av system med programvara, som utförs av FMV, och innebär i stort

- fr o m **1 oktober 1985** ställs särskilda krav på systemering och programmering, tillämpning av god **Software engineering**, (gäller alla typer av system)
- fr o m **1 januari 1986** krävs att **Ada-baserade PDL** används vid konstruktion av inneslutna system och vissa andra system
- fr o m **1 januari 1987** krävs att **Ada** används vid nyutveckling av inneslutna system och vissa andra system.

Vidare innebär anvisningarna en strävan att begränsa antalet olika programutvecklingsmiljöer för Ada.

För att sprida detta budskap om Software engineering och Ada har FMV under hösten 1985 informerat försvarsindustrin och sina samarbetspartners dels översiktligt vid FMV och dels mer i detalj under förberedda besök vid de berörda företagen. Vidare har FMV under våren 1986 (planerad fortsättning under hösten) arrangerat ett antal seminarier som belyser olika aspekter av modern Software engineering och Ada-teknologin. Ett viktigt informellt samarbete har i dessa sammanhang skett med Televerket och FOA.

Aldrig tidigare har försvaret så hårt gått in för någon viss metod eller teknologi i programvarusammanhang - möjligen med undantag för UNIX i Struktur 90 på den administrativa sidan. **Varför denna satsning? Vad är den grundad på? Vilka är våra förväntningar och förhoppningar?**

Satsningen på Ada gör FMV inför utsikten, att under de närmaste åren endast behöva arbeta med **"ett nytt"** programspråk i stället för flera nya språk. Samtidigt ser man en möjlighet att **"reducera arvet"** av olika språk.

Det finns en liknande orsak till satsningen på Software engineering, som för övrigt rimmal väl med Ada-teknologin. Vi tror nämligen, att god Software engineering är en förutsättning för att fördelarna med Ada skall kunna utnyttjas till fullo, och att man därmed kan konstruera system som är lättare att modifiera och vidareutveckla än eljest.

Satsningen grundar sig på bistra erfarenheter - både egna och andras. Nästan alla känner till begreppet "software crisis", som inte upptäckts bara i USA utan av alla dem som utnyttjar programvara och speciellt av

dem som betalar notan. I svenska försvaret lägger man för närvarande ner omkring 1.2 miljarder kronor om året på utveckling, anskaffning och vidmakthållande av system för informationsteknologi. Av dessa pengar går nästan 600 miljoner kronor åt till programvarudelen. FMV som anskaffare och såväl ÖB som försvarsgrenscheferna som betalare är väl medvetna därom. Orsakerna till de höga kostnaderna kan till stor del tillskrivas den flora av (undermåliga) utvecklingsmetoder och olika programspråk som används. En undersökning som gjordes 1982 visade, att inom FMV ansvarsområde användes mer än elva högnivåspråk (plus ett antal dialekter), mer än 24 assembleringsspråk och ett stort antal rena maskinspråk vid programmering. Härefter har tillkommit minst tre högnivåspråk och förmodligen några nya språk på lägre nivåer.

Detta resultat är kanske inte så drastiskt som motsvarande undersökning i USA visade, men tillräckligt anmärkningsvärt för att man skall inse att antalet olika språk är alltför stort för att vara ekonomiskt med tanke på utbildning, på underhåll och vidareutveckling av system i drift. I försvaret finns speciella förhållanden som ytterligare understryker detta: Försvarsspecifika krav fordrar ofta nyutveckling. Under den tid, som löper från specificering av ett system tills det kommer i drift, har utvecklingen tagit ett stort steg framåt. (Det kan ibland vara så, att utrustning är omodärn innan den tas i bruk!) Materielen utnyttjas i medeltal under femton år, varför den snabba utvecklingen idag dessutom medför att materiel och system som skall samverka ofta är av helt olika ålder och därför är utförda på skilda sätt.

Under denna långa tid förändras också användarkraven. (Vem skulle för femton år sedan drömma om att vid en liten arbetsplats kunna utföra alla de konster som idag är möjliga med en förhållandevis enkel persondator - och vem skulle då våga kräva det som idag är självklart?). Ibland sker organisationsförändringar, som kan resultera i stora utbyggnader och modifieringar av befintliga system. För att möta alla dessa förändringar vill FMV tillgripa en taktik som - åtminstone i fortsättningen - anpassar arbet och helst gör det mindra. Detta är speciellt viktigt vid samverkande system eller grupper av system, som utnyttjas i krig, även om en autonom funktion också i många fall måste vara möjlig. Även mellan datasystem för fredsrationell verksamhet är samverkan ofta ett måste.

Vi vet inte idag vilka krav som kommer att ställas om några år - än mindre om femton år (möjligheter skapar krav!). Vad vi däremot vet är att kravbilden kommer att drastiskt förändras, och att vi då står med ett arv som måste kunna anpassas till förändringarna. Detta kräver att vi nu tillämpar en taktik som tar hänsyn härtill. En väsentlig del av en sådan taktik är en väl avvägd **standardisering**, som utformas så att den främjar utvecklingen och samtidigt kan utnyttjas som styrmedel för att nå rationella och ekonomiska lösningar.

I standardiseringsarbetet bör vi därför ta vederbörlig hänsyn till arvet och vara medvetna om, att nya standardprodukter innebär en utökning därav. För att inte arvet skall få ett innehåll, vars komplexitet bara ökar, måste man gå tillväga på ett sätt, som medger att arvet anpassas till den tekniska utvecklingen och att rationella övergångslösningar skapas.

Syftet mad standardisering bör därför vara att uppnå **typbegränsning** och **kompatibilitet** samt medge **"planerad ersättningsanskaffning"**.

- **Typbegränsning** främjar i första hand arbetet med system i drift. Färre typer ger lägre kostnader vid utbildning av såväl användare som tekniker, lägre kostnader för utbytesenheter och reservdelar samt vid underhåll och service.

- **Kompatibilitet** främjar i första hand ny- och vidareutveckling av systemutbyggnad med nya funktioner och tjänster. Utnyttjande av modernare komponenenter underlättas i hög grad. Flyttbarheten av programvara mellan system respektive systemintegrering av olika komponenter förenklas.

- **"Planerad ersättningsanskaffning"** minskar arvets komplexitet. När ny utrustning anskaffas sker detta genom anskaffning av en typ som redan används eller genom att välja en ny version och därvid även passa på att ersätta föråldrad del av arvet med denna nya version.

Standardiseringen skall naturligtvis inte gälla bara maskinvaran. Programvaran - ja, hela systemkonstruktionen och därmed systemutvecklingen - skall bli föremål för "mjuka" standardiseringsåtgärder, mjuka för att medge vederbörlig frihet inom ramen för typbegränsning och kompatibilitet. Det är här modern Software engineering och Ada på allvar kommmer in i bilden. De förväntade förändringarna i kraven innebär att de system som produceras måste **struktureras** på ett sätt som medger att de kan modifieras och byggas ut. System- och programutvecklingsmetoder som härvid används skall därför vara av sådan kvalitet att de medger denna strukturering av system och en därtill anpassad dokumentation för framtida bruk vid förändringar. Delar av ett programsystem skall kunna bytas ut mot nya, samverkan med andra system skall senare kunna implementeras etc.

Ada-teknologin innehåller ju inte endast programspråket Ada utan även en väl definierad programutvecklingsmiljö. För FMV, som är en stor anskaffare av programvara, fordras stora resursinsatser för utvärdering och kontroll av programvarutunga system - insatser som blir större desto flera programspråk man skall bemästra. Detta förhållande gäller även system- och programutvecklingsmetoder. Införande av Ada-teknologin lindrar därför båda dessa problem. För att en strukturerad system- och programkonstruktion skall komma till sin rätt krävs en fortsättning i processen som passar därtill. Ada-teknologin är en sådan fortsättning - förmodligen den enda nu existerande som i sig förenar kraven på modularisering, realtidsprogrammering, stark typkontroll etc med kraven på en enhetlig, men ej begränsad, utvecklingsmiljö.

Dessa fördelar passar FMV, därför satsar vi på Ada.

Det finns emellertid andra grunder, som har bidragit till valet av Ada i försvaret: I USA är Ada standard både civilt och militärt sedan 1983. Det amerikanska försvaret har gjort en enastående satsning på detta programspråk och kräver programmering i Ada av sina leverantörer. NATO, Storbritannien, Västtyskland och andra länder har följt efter. Varför skall då Sverige välja något annat? Om försvaret i Sverige går in för Ada, tror vi att den svenska (försvars)industrin tidigare än annars själva satsar på Ada och att den därigenom skapar sig bättre konkurrenskraft utomlands.

FMV förväntar sig, att den svenska industrin och de svenska

programvaruföretagen svarar på Ada-satsningen och själva bidrar till Ada-teknologins införande i Sverige. För att uppmuntra en sådan utveckling har FMV på olika sätt stimulerat industrin, dels med ett stort samverkansprojekt, SDS 80/A, som bl a innebär utveckling av en programutvecklingsmiljö för Ada, dels med medverkan i mindre pilotprojekt. Hittills har gensvaret varit stort men det verkliga införandet försiktigt trevande! Vår förhoppning är att Ada-teknologin kommer att tränga igenom - inte bara för försvars- och processtillämpningar utan även för mer konventionella administrativa tillämpningar. Ada är nämligen ett språk, som genom de krav som tillkommit under dess utvecklingsskede, blivit generellt användbart.

Ada medger programmering av moduler som kan kompileras separat. I en Ada-värld kan därför program köras i en mängd olika datorer - programvara kan flyttas på ett helt annat sätt än tidigare. Vår förhoppning är att detta utnyttjas, att det uppstår en marknad för Ada-program, där det blir billigare att köpa ett visst program än att själv utveckla det. Detta skulle kunna drastiskt sänka kostnaderna för programutveckling och därigenom ge ekonomiskt utrymme för ambitionshöjningar mot än mer sofistikerade och användarvänliga system.

6. ERFARENHETER AV ADA

Det börjar finnas en del erfarenhet av användning av Ada i olika system. Det som redovisas idag har naturligtvis "igår", när arbetet gjordes, haft problem med tillgång på kompilatorer av produktionskvalitet. De första kompilatorerna av den typen kom under andra halvåret 1985 och senare. Undervisning av Ada har pågått en tid och det finns erfarenheter att redovisa därifrån också.

6.1 Medicinskt interaktivt grafiksystem för sjukvårdsmiljö

Det finns artiklar publicerade som berättar om erfarenheterna av användning av Ada i utceklingsprojekt. Ofta visar det sig att tillämpningen inte är ett inbäddat system. En typisk sådan erfarenhet presenterar Y S Eisa och J S Page i sin artikel "Designing an interactive, adaptive graphics system using Ada" (Proceedings of Ada-Europe International Conference, Edingburgh, 6-8 maj 1986, Cambridge University Press, sid 174). Det system vars utveckling de presenterar erfarenheterna från är ett grafiskt system för inmatning och lagring av data. Systemet har utvecklats för ett sjukhus för dess egen databehandling. Som inmatningsmedia används i första hand en grafisk tablett och i andra hand tangentbord när det gäller text. Systemet används av såväl läkare, sjuksköterskor som sekreterare. Utvecklingen skedde med en mycket begränsad budget. Erfarenheterna av att använda Ada är:

> "Overall, the use of Ada was very benificial. It is believed that a design might not have been realised within the constraints given, if it were not for the ability of Ada to support abstractions so readily and clearly. Furthermore, the use of Ada to provide continuous documentation of the design from the outset proved to be of considerable value in terms of project

management and productivity.

The success achieved in following the dictates of software engineering principles (Ross et al: Software Engineering: Process, Principles, and Goals. Computer, May 1975, p 66) was not only due to Ada's ability to enforce good coding discipline and consistency, but also to its capacity to support the design process. Ada has helped to create a much better understanding of the underlying problems, has allowed an object-based approach to be followed and, most importantly, has allowed the design to be documented as it evolved."

6.2 ADA i administrativt system

(Gunnar Boström, TeleLOGIC SUNDSVALL AB)

6.2.1 TeleLOGIC byter FORTRAN mot Ada

TeleLOGIC SUNDSVALL AB har sedan november 85 arbetat i ett Ada-projekt kallat TIR2. Det hela startades av TeleLOGIC i Sköndal hösten 83. Då en FORTRAN-tillämpning. Beställare är Pi, Televerkets instrumentsektion. Pi är ansvarig för inköp, service och uthyrning av instrument inom Televerket.

6.2.2 Instrumentregister nu i drift

TIR, Televerkets Instrument Register är i drift sedan dec -84. Instrumenten är klassificerade i användningsområden. Dessa områden är hierarkiskt uppbyggda med en fyrställig kod. Vid sökning av instrument kan man antingen ange hela instrumentklassen eller stegvis söka sig fram. När man funnit rätt klass kan man söka upp de instrumenttyper som finns. De finns registrerade med fabrikat, typnummer samt en kort beskrivning av tekniska data. Ur denna lista kan man se på varje typ och få fram detaljerade data och tillbehör. När man har hittat den instrumenttyp man behöver kan man ta fram en lista på alla individer av den typen. I listan visas aktuellt lånestatus. Om man är behörig kan man sedan boka eller direkt hyra instrumentet.

6.2.3 Baserat på MIMER relationsbatabas

TIR är skrivet i FORTRAN med hjälp av MIMER/DB och MIMER/SH som är en relationsdatabas respektive skärmhanterare från MIS, Mimer Information Systems AB i Uppsala. Koden innehåller många GOTO och Computed GOTO och anses svår att underhålla. Programmet förutsätter en Tandberg 2230 bildskärm. Efter det att programmet var klart har MIS kommit ut med förbättrade versioner av SH och DB. Speciellt har säkerhetsfunktionerna byggts ut väsentligt. Pi vill ha vissa funktionella förändringar. Blanda annat vill man kunna ha varianter av typer. Dessutom vill man lägga till flera nya funktioner.

6.2.4 Modifiera eller skriva nytt?

Vi i Sundsvall fick ansvaret att göra modiferingarna i grundversionen.

Efter en tidsuppskattning kom vi fram till att modifiering av FORTRAN-koden eller en nyskrivning i Ada skulle vara av samma storleksordning, ca 2000 timmar. Samtidigt skulle TeleLOGIC Sköndal implementera helt nya funktioner i TIR. Ada bedömdes här som klart överlägset FORTRAN.

Att vi övervägde Ada som programmeringsspråk var naturligt eftersom TeleLOGIC satsar stora resurser på Ada. TeleLOGIC samarbetar med TeleSoft i USA om en Adakompilator. Bibliotekshanterare och kodgenerator för VAX/VMS och UNIX datorer med MC68000 görs av TeleLOGIC. Trots att vi tar fram en egen kompilator valde vi här att använda Digital's. Det viktigaste skälet var att den kompilatorn är anpassad till parameteröverföringsmetoderna i den flerspråksmiljö som används i VAX/VMS. MIMER är skrivet i FORTRAN. Därför måste vårt Ada-program kunna anropa och bli anropade av FORTRANprocedurer.

6.2.5 Anpassning Ada/MIMER

Den första kod vi skrev var anpassningar mellan MIMERs nivå 2 rutiner och Ada. Dessa delades upp i 5 paket. Ett paket innehåller gemensamma deklarationer. Anpassningarna delades upp i 2 nivåer. På den lägsta finns specifikationerna för anrop av FORTRAN-rutinerna. Om koden skall kompileras med en annan Ada-kompilator måste denna nivå skrivas om. Det beror på att vi har använt ett av Digital definierat PRAGMA som specifierar hur parameteröverföringen skall gå till. Den övre nivån skall ej behöva ändras.

Den översta nivån är till för att göra anropen till MIMER klarare och säkrare. Eftersom MIMER är skrivet i FORTRAN och är portabelt måste procedurnamn vara högst 6 tecken långa. Dessa blir då i en del fall ganska oläsbara. Vi har gjort procedurnamnen längre och mer förklarande. I vissa fall har parametrar till MIMER ett begränsant antal värden. Parametrarna är av sträng- eller heltalstyp. Om man använder ett värde som inte finns får man fel vid exekveringen av programmet. Vi har använt två olika metoder för att ta hand om dessa fel vid kompileringen i stället. Ett sätt är att göra uppräkningstyper. Det andra är att göra olika subrutinanrop och i anpassningen göra om det som MIMER vill ha det. Valet mellan dessa två metoder har gjorts beroende på vad som har gett bäst läsbarhet.

6.2.6 Ada i tillämpningen TIR

TIR2 systemet är uppbyggda av ett antal paket. Det finns ett paket som sköter inloggningen till programmet och initieringen av MIMER. Därefter lämnas kontrollen till ett paket som sköter hela menyhanteringen. Varje meny innehåller maximalt 9 stycken alternativ som väljs med en siffra. En användare kan söka sig ner i menyträdet till dess att han kommer till den bild han vill använda. En van användare som vet den sekvens som måste gås igenom kan slå samtliga siffror och slipper då bläddra sig fram. Varje bild i systemet har ett namn. Om man känner det namnet kan man i stället ange detta vid menyvalet.

Det finns tre typer av bilder i systemet: meny-, list- och registervårdsbilder. Endast behöriga användare får använda den sista typen av bilder. Varje list- och registervårdsbild hanteras i separata paket. Samtliga bilder är i princip helt oberoende av varandra och en användare kan alltid välja vilken bild som helst utan behöva gå via någon annan. Den enda koppling som finns är några globala variabler.

Dessa är av typen fabrikat och individnummer. Det vill säga sådan information som man ofta anger som nyckelbegrepp i bilderna. Vi har lagt dessa globala, för att man skall kunna ta med sig dessa överallt i systemet. Till exempel kan man i en listbild över fabrikanter leta fram en speciell fabrikant. Om man därefter går in i registervårdsbilden för fabrikanter är fabrikantnamnet förifyllt. Systemet kommer ihåg senaste bild så man kan lätt växla mellan två bilder.

Varje bild utom menybilder hanteras i ett eget paket. Denna uppbyggnad gör att man lätt kan modifiera och testa bilder utan att vara beroende av några andra. Vi har använt Ada's möjligheter till att dölja subprogram och variabler i paket. Detta gör att vi kan använda samma procedurnamn för likartade funktioner i de olika paketen utan namnkonflikter. Det är lätt att förstå ett paket som någon annan har skrivit.

Till systemet hör också ett antal generiska paket som förenklar programmeringen av de olika bilderna. Vi har en bild för att kunna lista och göra utval på fabrikantnamn. Bilden tillåter att man söker på fabrikanter med "wild cards". Dvs man kan söka på fabrikanter som börjar på, innehåller, är större än eller mindre än ett antal inslagna tecken. När man får fler träffar än det som ryms på en sida, kan man bläddra framåt och bakåt. Man kan göra urval av en viss fabrikant och lägga i den globala variabeln. Man kan komma ut i VMS hjälpfunktion. Om man får bilden förstörd av ett meddelande eller något annat kan man direkt återskapa den med en funktionstangent. Funktionen utförs i en paket på 270 rader med bara 61 Adainstruktioner. I proceduranrop använder vi namnanrop på parametrar, och varje parameter står på en egen rad. Vi använder strukturerade konstanter (arrayer av records) där varje komponent står på en egen rad. Vi har inte utnyttjat Ada's möjlighet att göra USE på paket. I stället gör vi rename på paketet till ett kort namn och kvalificerar alla proceduranrop med detta. Det gör det enkelt att se i vilket paket ett subprogram finns. Allt detta gör att vi anser att vi har uppnått målet att få en strukturerad, lättläst och lättunderhållen kod. Multitaskmöjligheterna i Ada har vi inte utnyttjat alls.

6.2.7 Ada är lätt att använda

Vi är fem personer som har varit med i projektet i Sundsvall. Tre oerfarna programmerare och två erfarna. De oerfarna har ett halvt års erfarenhet av COBOL och Ada. De anser att det har varit lättare att använda Ada än COBOL. De problem de haft har till största delen berört användningen av MIMER och kopplingarna till applikationen. En del svårigheter har det också varit att förstå hur de generella paketen skall utnyttjas. Av oss två som är erfarna är det endast jag som hade kunskap om MIMER tidigare. Vi har båda Pascal-bakgrund och har haft lätt att lära oss Ada. Vi har skrivit de generella generiska paketen och mindre delar av applikationen.

Projektet pågår fortfarande så vi har inte gjort någon efterstudie. Vi har fortfarande alla kontroller påslagna som Ada kan göra. Kompileringar och länkningar sker med debugger. Därför kan vi inte säga så mycket om kodstorlekar och exekveringstider.

6.2.8 Vad har Ada betytt

Följande fördelar har vi fått genom att använda Ada:

- Bra läsbarhet

- Strukturerad underhållsvänlig kod.

- Återanvändbara paket.

- Bra säkerhet

- Färre instruktioner att underhålla.

- Portabel kod.

Biblioteksshanteringen i Ada gör att man aldrig kan länka ihop ett system där de ingående delarna har felaktiga beroenden.

Sammanfattningsvis kan sägas att MIMER och Ada utgör en mycket bra kombination för utveckling av administrativa program, som vi kommer att fortsätta med.

6.3 Användning av Ada i ett tekniskt applikationsprojekt.

(Margaretha Cedercrantz, TeleLOGIC AB)

MARC är ett system för övervakning av stora kundväxlar. Det är utvecklat av TeleLOGIC på uppdrag av Televerkets Företagsservice. Utvecklingsarbetet har skett i två etapper. I den första utvecklades funktioner för övervakning av A345-växlar. Denna utveckling skedde mha en subset Ada-kompilator för en 68000 maskin, APN 167. Etapp två, med början hösten 1984,innebar stora utvidningar, övervakning av tre andra växeltyper, ny målmaskinvara, etc.

En utredning gjord av Devenator förordade Ada som fortsatt utvecklingsspråk trots att det begränsade valet av målmaskin. De faktorer som talade för Ada enligt Devenators utredning var:

- Ada är ett i sann mening portabelt språk. På grund av Ada's långtgående standarisering bör MARC-systemet kunna köras på en annan målmaskin med ett minimum av modifiering. Detta gör att ett framtida byte av målsystem enkelt kan göras när den tekniska utvecklingen har gått ifrån dagens maskiner.

- Ada underlättar skrivandet av högkvalitativ programvara, dvs programvara som är välstrukturerad, felfri och flexibel. Ada ger framförallt goda möjligheter att bygga sitt system genom att implementera generella byggbitar (i form av packages eller procedurer) som sedan kan återanvändas.

- Ett system skrivet i Ada är lätt att underhålla.

- Introduktionen av Ada har gått ganska långsamt i början. Vi bedömer ändå att Ada är ett språk för framtiden. Nu börjar också utvecklingen ta fart i och med att en mängd nya validerade kompilatorer finns på marknaden.

Inom Marc projektet har positiva erfarenheter erhållits på just dessa punkter.

6.3.1 MARC systemets funktioner

MARC övervakar fyra typer av växlar, A345, A335, Hotellsystem 900 samt Triton 600. Övervakningen sker dygnet runt och är rikstäckande. Det finns fyra system i landet. Övervakningen går i korthet till så att A345-växlar rings upp (scannas) av systemet och växlarnas felbuffertar läses in och analyseras varpå larm skrivs ut på en larmskrivare. Övriga växeltyper ringer själva via en larmsändarutrustning upp MARC-systemet. MARC hämtar relevant information från systemets register och skriver ut ett larmmeddelande på en eller flera larmskrivare. Larmskrivare finns på varje teleområde och på Televerkets nätdriftcentraler. Nattetid styrs också larmen till nätdriftscentraler där det finns bemanning dygnet runt.

MARC är utvecklat under VAX/VMS för MicroVAX/VMS.

Projektet har omfattat ca 12 000 timmar och som mest har 10 personer varit involverade.

MARC projektet har gått helt efter tidplanen utan några komplikationer. Under våren har det varit provdrift av MARC. Erfarenheterna från denna period talar för att det är ett kvalitativt bra system.

6.3.2 Portabilitet

En konvertering gjordes våren 85 av första etappens MARC från APN 167 till MicroVAX. Det totala konverteringsarbetet för applikationen var ca en halv dags arbete. Några få anrop till drivrutiner ändrades och sedan räckte det att kompilera om med den kompilator som genererade VAX/VMS-kod. Några drivrutiner fick också skrivas för MicroVAXen vilket tog 6 manveckor.

6.3.3 Högkvalitativ programvara

Att använda Ada som programspråk har underlättat väsentligt i både design-arbetet och implementeringsfasen. MARC har troligen fått en klarare och bättre struktur tack vare att stor del av specifikationen gjordes i Ada-termer. Ada's paket understödjer både användandet av "svarta lådor" och objektorienterad design. I och med att varje modul i systemet entydigt beskrivs av specifikationen kan bodyn ses som en "svart låda". När paketspecifikationerna var klara kunde arbetet därför fortskrida mycket självständigt med varje modul. Och man var ändå garanterad att gränssnitten och samarbetet mellan modulerna stämde under hela utvecklingen.

Ett paket kan definieras att hantera en typ av objekt och ingångarna i paketet är de operationer som kan utföras på objektet. För objekt beskrivna av poster måste då t ex operationen uppdatera finnas för varje element i posten. Vid stora poster blir detta otympligt även om det är den ideala designen. I MARC har vissa kompromisser gjorts och vissa objekttyper har gjorts synliga.

I MARC var Ada's begrepp generics till stor nytta. All överföring av data skedde via buffertar för att undvika synkroniseringsproblem. Dessa var alla realiserade via ett och samma generiska paket med typen av buffrad data som parameter. Utvecklingen av de registerfunktioner för de fyra typer av register som finns i MARC gjordes också i stor utsträckning mha generics.

Ada begreppet tasking har också varit mycket användbart för att lösa olika realtids och resursfördelningsproblem. Men man bör nog sträva efter att minimera antalet task-byten eftersom de inverkar negativt på prestanda.

6.3.4 Underhåll

Det har t ex varit enkelt att införa nya/ändrade önskemål från kunden på ett sent stadium. Under provdriftsperioden infördes tex möjligheten att ha ett godtyckligt antal (Ett maximum finnes men begränsningen ligger utanför applikationen) scanners parallellt arbetande. Ändringen tog en vecka.

6.3.5 Övriga erfarenheter

Ada understödjer god programmeringsstil men det är fortfarande viktigt att inom ett projekt ha god disciplin. Det är också viktigt att "verktygen" runt omkring passar väl ihop med Ada. I MARC projektet användes Strix till att hålla samman alla filer i en trädstruktur samt till att generera dokument. VMS-miljön utvidgades med verktyg för produkthantering, nattliga konsistenskontroller/kompileringar, etc.

Adas bibliotekshanterare var också ett viktigt verktyg. All uttestad programvara kompilerades in i ett bibliotek på en övre nivå. Program under utveckling låg på nivån under i olika modulbibliotek. Varje natt kompilerades automatiskt eventuellt nya program in på den övre nivån. Detta gjorde att projektmedlemmarna delade en stabil miljö.

Det är mycket viktigt under designfasen att minimera antalet beroenden mellan modulerna. Hellre många små och konsistenta moduler än några stora hybrider. Det är också en fördel att samla "globala" konstanter/typer i ett eget paket som alla moduler är beroende av. Om sådan istället är deklarerade i ett paket som refererar till ett antal andra så kan varje kompilering medföra att hela systemet kontrolleras och kan resultera i "onödiga" omkompileringsbehov.

Detta är en följd av att det redan vid kompileringstillfället sker konsistenskontroller. Vilket medför att mycket integrationsarbete görs under utvecklingen. Om man som i MARC-projeket kompilerar om system varje natt så finns en logg på morgonen att gränssnitten fortfarande stämmer. För många andra programspråk kan denna kontroll göras först vid länkningen och kontrollen måste göras av programmeraren själv.

En viss användning av Ada komponenter kom till ständ. T ex användes paket för lågnivå io, och hantering av länkade listor. Dessutom användes formulärhanteraren Logiform, som är skriven i Ada.

MARC utvecklades i två steg. Först skrevs samtliga specifikationer som beskrev gränsnittet i systemet och samtliga bodies innehöll enklast

tänkbara kod. Därefter integrerades hela systemet och testkördes. Denna integration tog en knapp vecka. Därefter frystes samtliga specifikationer och riktig kod skrevs för resp. body. Integrationen som följde tog en knapp dag där den mesta tiden gick åt att få hårdvaran att fungera.

Sammanfattningsvis så är erfarenheterna av Ada i MARC projektet mycket goda. Ada har givetvis också brister, men jämfört med andra programspråk så ger Ada ett överlägset stöd under projektets hela livslängd, från designfas till underhållsfas.

7. ERFARENHETER AV ADAUTBILDNING INOM INDUSTRIN

(av Jonas Agerberg)

7.1 Ingress

Det jag här skall redovisa är erfarenheter från mer än fyra års verksamhet med utbildning i Ada. Sammanlagt rör det sig om ett tjugotal kurser med totalt mer än 200 elever, de flesta från industrin.

I samtliga kurser har ungefär halva kurstiden ägnats åt praktiska tillämpningsövningar på dator.

Formen är en intensivkurs med begränsat deltagarantal (8 - 12 elever) som körs under fyra konsekutiva heldagar. Övningar görs i arbetsgrupper om två eller tre elever. Varje grupp disponerar en egen arbetsstation med komplett Adasystem.

I kursen behandlas och övas allt väsentligt i språket, inklusive processhantering, generiska programmoduler m m.

Den kursmodell jag kommit fram till har varit mycket framgångsrik att döma av de värderingar som samtliga elever lämnat in efter avslutad kurs. I enkäten har de också fått ge uttryck för sin syn på språkets komplexitet. Mer om detta nedan.

7.2 Synen på Ada och Adautbildning

Den gängse uppfattningen om Ada är att det är ett stort, komplicerat och svårbemästrar språk, någonting i stil med PL/1 fast värre.

Jag delar inte den uppfattningen, och det gör uppenbarligen inte heller eleverna efter genomgången kurs.

Ada är förvisso inte något litet språk. Det är en bra bit "större" än t ex Pascal. Men samtidigt är det betydligt mer anpassat för den praktiska verkligheten. Ada är dessutom strikt logiskt uppbyggt kring ett fåtal grundprinciper. Dessa är anpassade efter behoven vid praktisk programutveckling - det är inte fråga om en rikhaltig repertoar av disparata "finesser".

Detta gör att man kan lära sig hur språket fungerar på logiska och behovsmässiga grunder snarare än genom inlärning av en mängd minnesregler. Erfarenheterna från kurserna styrker denna uppfattning.

Programvarutekniska begrepp som modularisering, gränssnittskontroll, program- och datastrukturer, parametrisering etc finns ingen anledning att presentera som abstrakta principer inom "software engineering". I Ada finns motsvarande begrepp inbyggda på ett sätt som gör det naturligt att använda dem. Det är analogt med att vi inom mekanisk konstruktion hellre säger "skruva samman" än "foga ihop olika med hål försedda föremål med hjälp av delvis cylindriska metalldelar vilkas cylindriska del försetts med spiralformiga spår.....".

De här egenskaperna hos Ada underlättar undervisningen väsentligt - än mer när eleverna direkt kan öva upp sitt Adatänkande i praktiska övningar.

7.3 Utbildningsmål

Eleverna skall efter genomgången kurs ha fått en grundlig inblick i hur Ada kan användas som:

- Uttrycksmedel för beskrivning och design
- Kommunikationsmedel mellan medarbetare inom ett projekt (horisonellt och vertikalt)
- Programmeringsverktyg för projektets genomförande

Detta innebär att det inte är fråga om en ren programmeringskurs. Kursen är upplagd så att olika kategorier, från programmerare till projektledare, skall kunna tillgodogöra sig den fast på skilda sätt.

7.4 Förkunskaper

Kursen är helt baserad på Ada och kräver inga speciella programmeringsförkunskaper. Det innebär att elever med olika förkunskaper kan tillgodogöra sig kursen på olika sätt. Den som jobbar med projektledning kan se de strategiska egenskaperna i språket medan den som jobbar med programutveckling kanske mera ser till de taktiska.

För vissa individer kan förkunskaperna leda till förutfattade meningar. I sådana fall kan och måste kursen också innebära en form av hjärntvätt. Det är ytterligare en anledning till att låta kursen stå på egna ben och inte repliera på vare sig något speciellt språk eller någon speciell metodik.

7.5 Kursuppläggning

Kursens längd är en viktig faktor när eleverna kommer från industrin. De som jobbar på programvarusidan står ofta under stark tidspress i arbetet och kan inte avvara mer än några få dagar. En kurslängd på fyra dagar ger en måndag eller fredag för arbete, sammanträden, MBL-

aktiviteter m m.

Tempot är viktigt. Eleverna får inte ges tillfälle att glömma det som behövs för att lösa nästa övningsuppgift. Därför varvas föreläsningar och övningar så att minst två övningar hinns med per dag.

Den första övningen startar efter c:a en halvtimmes introduktion och den andra skall vara klar före lunch den första dagen. I dessa övningar introduceras paketmekanismen i Ada genom att eleverna får lära sig att använda paket med färdiglagade datatyper och operationer på dessa typer.

Med dessa erfarenheter i bagaget har vi nästan gratis fått följande:

- Grunderna i den viktiga modulariserings- och kapslingstekniken i Ada (paketering, privata typer).

- Grunderna för hur separatkompileringsmekaniken fungerar i Ada.

- Principerna för gränssnittskontroll vid hopkoppling av programmoduler.

- Underlag för att förstå standardtyperna och tillåtna operationer för dessa (heltal, flyttal, tecken, strängar m m).

- Underlag för att förstå viktiga delar av in- och utmatningspaketen.

Ada är gjort så att man skall kunna bygga upp en repertoar av effektiva och återanvändbara verktyg. Vi har alltså börjat med att lära oss utnyttja en sådan verktygslåda.

Först därefter är det dags att gå ner och gå igenom detaljer, dvs de byggbitar som man utnyttjar när man tillverkar moduler på en högre nivå. Det är främst fråga om procedurer och funktioner och deras innanmäten i form av datatyper, variabler, konstanter, satser och felhantering. Detta tar ungefär en dag inklusive övningar.

Därefter är det dags att höja nivån igen och visa hur man paketerar objekt och datatyper tillsammans med deras operationer. Vidare hur man genom generisk parametrisering kan bredda användningsspektrum för paket och subprogram. Separatkompileringsmekaniken gås igenom systematiskt.
Parametriseringstekniken är viktig vid byggande av generella programmoduler. Ett avsnitt ägnas åt parametriserade datatyper och ett åt dynamiska datatyper.

Därmed har vi avverkat den sekventiella delen av Ada och två och en halv av de fyra kursdagarna.

Drygt en dag ägnas åt processhanteringen i Ada. Här genomförs den största övningsuppgiften där varje grupp skall göra en nod i ett enkelt "token passing network". Programmet testas genom att varje grupp kopplar ihop sig med sig själv i en "enstationsring". När alla grupper är klara kopplas de ihop i en enda ring och varje grupp kan sända adresserade meddelanden till godtycklig perifer enhet på nätet. Varje meddelande kan ha flera adressater. Det brukar bli en minst sagt livfull tillställning som också markerar avslutningen på den egentliga

kursen.

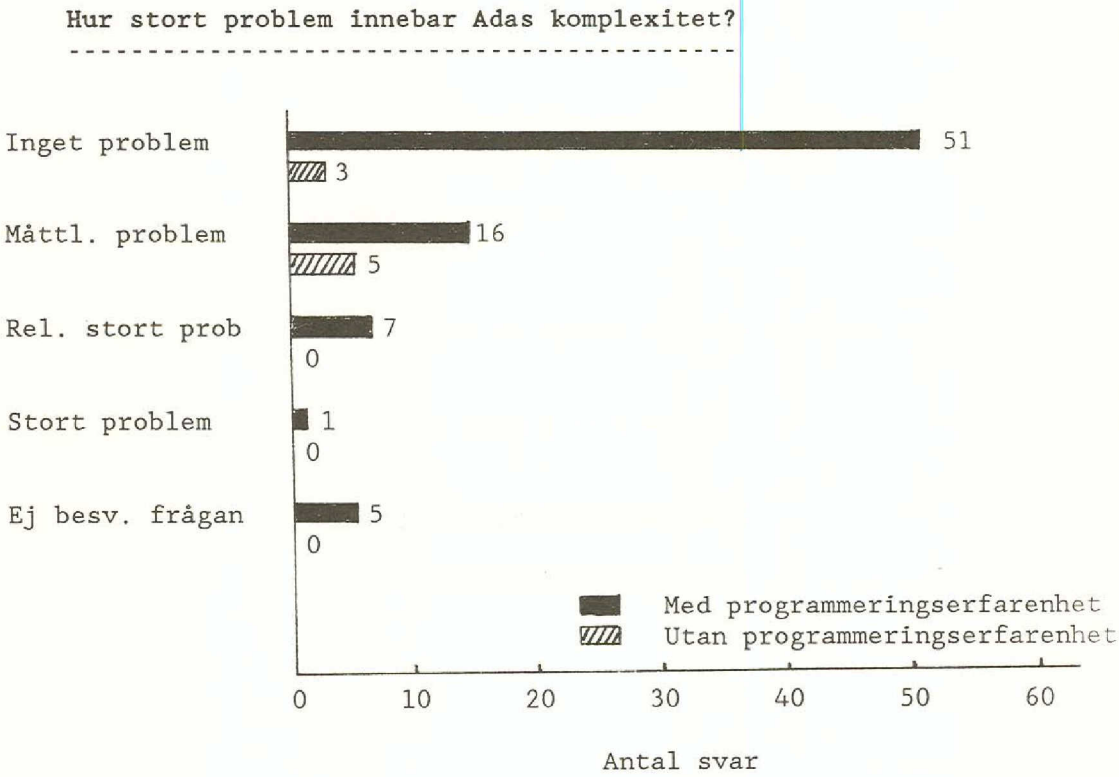
Kursen avslutas som regel med en fråge- och diskussionsstund.

Korta avsnitt om maskinnära programmering, APSE-problematiken m m sprängs in på lämpliga ställen. Längd och djup på dessa brukar anpassas efter elevernas bakgrund och intresse.

7.6 Kursenkäterna

I en kurs av detta slag där kunskapsförmedlingen inte kan kontrolleras med objektiva metoder som tentamina o d är man hänvisad till subjektiva metoder. Kursenkäten ger elevernas subjektiva uppfattning om utbytet av olika avsnitt och kursen som helhet. Dessutom om balansen mellan föreläsningar och övningar, kursdokumentation m m. Här skall redovisas några fragment.

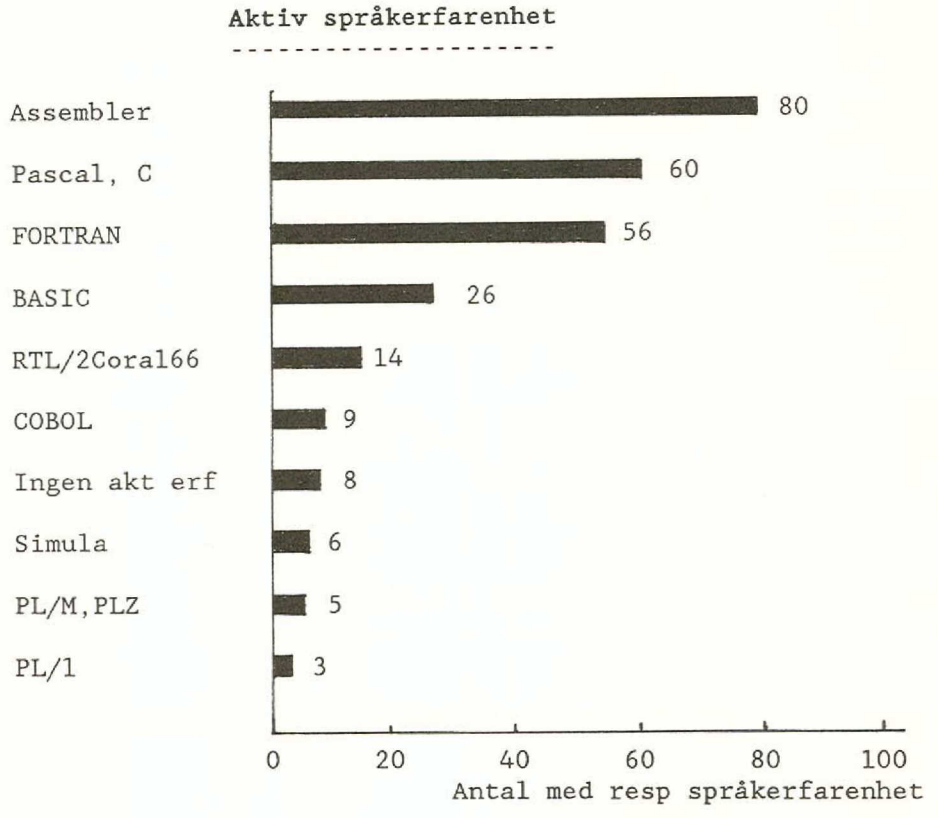
Den intressantaste frågan i enkäten är den om hur eleverna uppfattat Adas komplexitet och hur det påverkat deras möjlighet att sätta sig in i språket. Svaret fördelade sig sålunda:



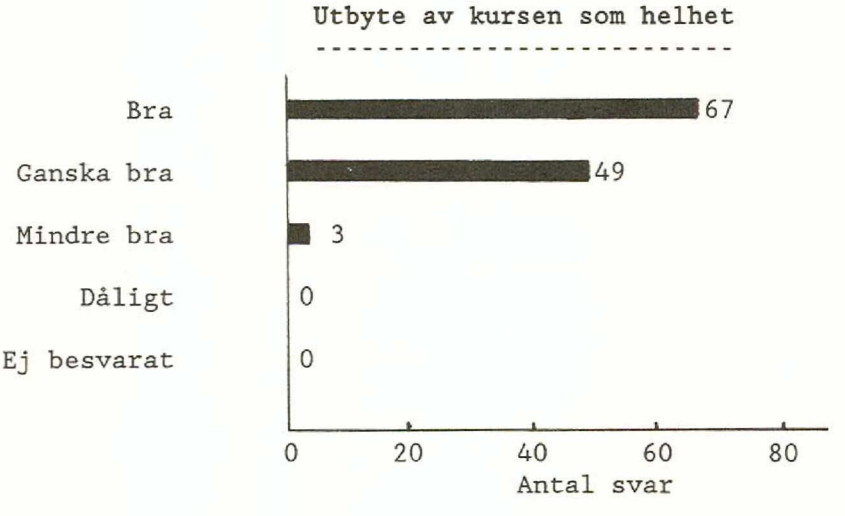
Det övervägande flertalet har alltså inte upplevt Ada som ett speciellt komplext och svårlärt språk. Ett mindre antal (8 st) hade ingen tidigare programmeringserfarenhet. Det sade sig ha något större problem,

dock inte så att någon hamnade på den undre halvan.

Fördelningen av elevernas tidigare aktiva programspråkserfarenhet fördelade sig så här:



Elevernas värdering av sitt utbyte av kursen utföll på följande sätt:



En närmare analys visar att varken uppfattningen om Adas komplexitet eller värderingen av kursen är speciellt starkt korrelerade till elevernas förkunskaper eller vilken språkkategori vederbörande jobbat med.

Man skulle tro att till exempel de som huvudsakligen arbetat på assemblernivå skulle ha större problem än dem som har Pascalbakgrund. Så är inte fallet.

7.7 Erfarenheter

Ada har visat sig vara en utmärkt partner i undervisningen. De goda resultaten är till övervägande del språkets förtjänst. Min uppgift som lärare och handledare har mest varit att visa vägen. Sedan har eleverna själva med Adas hjälp och litet stöttning här och där klarat av sina uppgifter snabbt och ofta elegant.

Det beror naturligtvis ytterst på att språket är konstruerat så att det skall kunna fungera som ett funktionellt, ingenjörsmässigt effektivt verktyg.

Några övriga erfarenheter och iakttagelser sammanfattas punktvis nedan.

7.8 Praktik först - teori sedan

Jag försöker i möjligaste mån inskränka teoridelen före en övning till vad som behövs just för det momentet. Övningen ger eleverna referensramar att hänga teoribitarna i.

Teorin innebär som regel en förklaring av de logiska sammanhangen, dvs varför konstruktionerna måste se ut si eller så. Det är lättare att komma ihåg en regel om den uppfattas som naturlig och självklar.

7.9 Ada - ett naturligt uttrycksmedel

När man följer elevernas arbete med övningar är det påfallande hur relativt sällan de konsulterar referensmanual, lärobok eller referenskort. De lär sig mycket snabbt att skriva formellt korrekta program.

Därav kan man dra den slutsatsen att enbart en syntaxanalysator är av diskutabelt värde för utbildning i Ada. Jag är snarare av den uppfattningen att den kan vara skadlig eftersom risken finns att eleverna koncentrerar sig på för programbyggnadstekniken oväsentliga ting.

7.10 Subsetkompilatorer

Det finns fortfarande på marknaden kompilatorer som inte är validerade och som utger sig för att klara "större delen" av språket.

Min erfarenhet i detta avseende är att det krävs mycket grundliga kunskaper i Ada om man skall lotsa fram eleverna via en sådan kompilator. Det är nämligen näst intill omöjligt att definiera en delmängd av Ada utan att bryta mot väsentliga logiska sammanhang. Risken är att mycket tid går åt att förklara hur det "egentligen skulle ha sett ut". Det kan också bli en Pascalutbildning i Adaförklädnad. Ada är definitivt inte Pascal.

Det kan också vara på sin plats att påminna om att valideringen av en kompilator endast innebär att den överensstämmer med ANSI-standarden.

Det är i inget avseende något kvalitetsprov. Än mindre ett test på lämpligheten för utbildningsändamål. Själv använder jag vid kurserna fortfarande en av "urkompilatorerna", den som Gensoft validerade 1983 och som är den enda som går på en maskin med endast 128 kb direktminne.

7.11 Kurslitteratur

Det finns åtskilliga Adaböcker på marknaden, de allra flesta på engelska. När jag började min kursverksamhet fanns det inte någon som motsvarade mina krav. Därför fick jag utarbeta ett eget kurskompendium som successivt byggdes ut. När Studentlitteratur ville ha en lärobok i Ada på sin repertoar fick jag hjälp av min dotter (som är arkitekt) att utvidga och omarbeta materialet. Resultatet blev "Ada-boken" (Studentlitteratur 1984). Boken är avsedd som stöd vid den typ av lärarledd hands-on-utbildning jag har beskrivit. För torrläsning kräver den en viss språkvana. Det är fortfarande den enda svenskspråkiga läroboken i Ada.

7.12 Arbete i grupp eller enskilt?

Jag har funnit att arbetsgrupper på två eller tre elever ligger nära idealet. En stor del av utbildningseffekten kommer från den intensiva diskussion som spontant uppstår inom grupperna.

Programutveckling i större skala är ju dessutom en i hög grad kollektiv aktivitet. Att sätta en elev vid var sin terminal vore direkt felaktigt, åtminstone för den här sortens kurser. Dessutom krävs i så fall en högre handledartäthet.

7.13 Vad hinner man med på fyra dagar?

Givetvis är det inte möjligt att utbilda folk till fullfjädrade programkonstruktörer på fyra dagar. Däremot kan man lägga en sådan grund att de som redan har en bakgrund i tillämpad "software engineering" skall kunna använda och utveckla sitt Adakunnande tämligen direkt.

De som inte har denna bakgrund får en begreppsapparat och en nomenklatur som är en utmärkt bas för vidare utbildning (kursmässigt eller i arbetet) inom tillämpad programvaruteknik.

De som inte alls själva sysslar med programmering (projektledare m m) får bättre möjligheter att kommunicera med sina medarbetare i frågor som rör programprodukternas uppbyggnad och funktion.

7.14 Slutord

Ytterst är det fråga om ekonomi.

Ada och utbildning i och kring språket är inget självändamål. Inte förrän man kommit dithän att man fått en väsentligt rationellare programvaruteknologi än vi har i dag har de tankar som legat bakom språkets tillkomst blivit förverkligade.

Då krävs att inte bara några programmerare gått Adakurs. Det krävs förändringar på många plan: organisation, arbetsformer, kompetensprofil. Med Adateknologin i botten.

De företag som kommer igenom den rationalisringsprocessen har pengar att hämta. På sikt sannolikt ganska stora pengar.

8. ERFARENHETER AV ADA I HÖGSKOLEUNDERVISNINGEN

(av Jan Skansholm, CTH)

Är Ada ett svårt språk som bara kan utnyttjas av professionella programmerare i komplicerade tillämpningar? Erfarenheter från undervisning av Ada vid Chalmers Tekniska Högskola och Göteborgs Universitet visar att att svaret på denna fråga är nej. Det har tvärtom visat sig att Ada fungerar utmärkt som språk också för ovana programmerare och att Ada med fördel kan användas som nybörjarspråk.

Pascal är idag vid de flesta universitet och högskolor det dominerande språket inom programmeringsundervisningen. Ada har hittills använts främst inom kurser i realtidssystem och operativsystem. Pascal har valts för att språket är litet och enkelt. Efter ett antal års erfarenhet av Pascal som undervisningsspråk börjar man dock nu ställa sig frågan om inte Pascal är lite *för* enkelt. Det finns många vanliga programmeringssituationer som på grund av begränsningar i Pascal måste lösas på ett krångligt sätt. Som exempel kan nämnas inläsning av textsträngar.

Ännu allvarligare är att man i Pascal alltför fort "slår huvudet i taket". Det är t.ex. svårt att lära ut den viktiga tekniken med modulär programutveckling när Pascal i sin standardform inte ens tillåter separatkompilering. För att kunna förklara begreppet modul måste man studera utvidgningar eller varianter av språket (t.ex. Pascal Plus). Detta har av eleverna upplevts som mycket rörigt och man har haft svårt att förstå vad det hela går ut på.

Detta är bakgrunden till att man velat pröva ett annat undervisningsspråk. och det finns flera skäl till att Ada då är ett intressant alternativ.

Ada är ett modernt språk som grundar sig på erfarenheter från tidigare språk. Ada är ändå ett konventionellt språk som till sin struktur är likt flera andra vanliga programspråk. Detta gör att den som lärt sig Ada också på kort tid kan lära sig programmera i något annat vanligt språk, t.ex. Pascal.

Ada är ett standardiserat språk. Man slipper problem med olika dialekter och varianter. Det man lär sig vid en programmeringskurs i Ada kan alltså användas i alla Ada-installationer man kan komma i kontakt med.

Det viktigaste i en programmeringskurs är egentligen inte att lära ut

ett visst språk utan att ge principer för hur man skall göra när man konstruerar program. Detta betyder inte att valet av språk är oviktigt. Nybörjare har oftast svår att skilja på språket och principerna och det är därför viktigt att de principer man vill lära ut också har stöd i språket. Vill man t.ex. lära ut modulär programutveckling måste ett modul- eller paketbegrepp finnas i språket. Detta gör att Ada är ett lämpligt språk eftersom det har många möjligheter och till skillnad från Pascal ger stöd åt flera viktiga programmeringsprinciper.

Ada är ett språk man kan växa med. Det kan användas som genomgående språk i programmeringsundervisningen från rena nybörjarkurser till kurser i realtidsprogrammering och software engineering. Alla de programkonstruktioner man behöver finns i språket. Detta gör att man som elev slipper förvirras av olika språkliga varianter och tillägg.

En sak man skall observera är att de flesta av dagens studenter har en viss programmeringserfarenhet (oftast av BASIC) då de börjar vid universitet eller högskola. Så var det inte för några år sedan. Det borde då vara mer stimulerande och intresseväckande för studenterna att få arbeta med ett språk som Ada med alla de möjligheter detta erbjuder.

Några ord måste sägas om kurslitteratur. De flesta Ada-böcker är skrivna för läsare som har ganska goda tidigare programmeringskunskaper. Dessa böcker går bra att använda om man introducerar Ada först i en kurs i realtidsprogrammering. Skall man använda Ada i en nybörjarkurs eller i en annan kurs där eleverna har begränsad tidigare programmeringserfarenhet är det viktigt att man har kurslitteratur som vänder sig till nybörjare. Kurslitteraturen måste behandla *programmeringsteknik* och får inte bara vara en beskrivning av språket Ada.

Hur har det då fungerat i praktiken med Ada i undervisningen? Vid Chalmers har Ada under några år haft sin givna plats i realtids- och operativsystemskurser. Där har språket varit ett utmärkt hjälpmedel för att beskriva klassiska begrepp som processer och synkronisering av processer. I dessa kurser ges först eleverna en snabb genomgång av grunderna i Ada. Därefter koncentrerar man sig på de delar av språket som rör parallell programmering. Risken med denna kursuppläggning är att studenterna visserligen får bra kunskaper om parallell programmering i Ada men att det vilar på en bräcklig grund. Kunskaperna om de grundläggande delarna av Ada blir alltför ytliga.

Under vårterminen -86 har de första försöken gjorts att införa Ada i ett tidigare skede i programmeringsundervisningen. Vid Göteborgs Univiersitet har Ada använts som språk på en ren nybörjarkurs omfattande 10 poäng och vid F-linjen på Chalmers har Ada använts i den första fortsättningskursen i programmeringsteknik (här har eleverna tidigare läst en mindre introduktionskurs i Pascal). Eftersom kurserna när detta skrivs ännu inte är helt avslutade är det kanske för tidigt att göra någon definitiv utvärdering men något kan ändå sägas.

Det påstås ibland att Ada är ett "svårt" språk eftersom språket är så stort och att det därför är bättre med ett "enklare" språk som Pascal. Hur har eleverna upplevt det? När man skall lära sig de mest grundläggande programkonstruktionerna (olika satser, deklarationer och underprogram) verkar det faktiskt vara så att Ada för en nybörjare är lättare att förstå än Pascal. Om man jämför de båda språken visar det sig att Adas satser har en renare struktur och att mekanismen för

överföring av parametrar och resultat till underprogram är mer generell och konsekevent. Också när det gäller in- och utmatning verkar det som Ada är enklare. Det är visserligen lite klumpigt att man alltid måste ha med with- och use-satser för att få tillgång till in- och utmatningspaket men detta vållar inte eleverna några praktiska problem.

En verkligt stor pedagogisk fördel med Ada jämfört med Pascal får man när man kommer till området modulär programutveckling. Det har varit möjligt att på ett konkret sätt tidigt i programmeringsundervisningen införa de viktiga begreppen moduler och abstrakta datatyper. Adas paketbegrepp har av eleverna upplevts som mycket naturligt och man har kommit med frågor som: "Varför skulle detta vara svårt?".

Som exempel kan nämnas att det finns en programmeringslaboration som vi under flera år använt inom undervisningen. Avsikten med laborationen är att ge övning i att utveckla större program. Laborationen har tidigare utförts i Pascal och har alltid vållat studenterna stora problem. Ofta har man (trots att laborationen till slut blivit godkänd) inte riktigt förstått vad man har gjort och vad det hela gått ut på. Denna laboration har nu, tack vara Ada, för första gången blivit bra och lättbegriplig och studenterna har förstått tekniken med att dela upp större program i väl avgränsade moduler.

Vilka problem har uppstått? Det har naturligtvis funnits problem med att byta undervisningsspråk. Men dessa problem har inte berott på Ada som språk utan har varit av teknisk natur. Det främsta problemet har varit att kompileringstiderna blivit för långa. Vi gjorde en alltför optimistisk bedömning av laborationsdatorns förmåga att klara av den ökade belastningen och utnyttjandet av datorn blev felaktigt planerat. Dessutom visade det sig att den kompilator vi använde var ovanligt slösaktig när det gällde att utnyttja sekundärminne och att den innehöll en del buggar.

Utbudet av Ada-kompilatorer är ännu inte så stort och dagens kompilatorer är i första hand konstruerade för att generera effektiv kod. I undervisningssammanhang är man sällan intresserad av de färdiga programmens effektivitet. Det som är viktigast är att man har en bra programutvecklingsmiljö och att kompileringar sker någorlunda snabbt. En Ada-interpretator i stället för en kompilator skulle vara välkommen. Dessutom skulle man (trots att det strider mot den nuvarande Ada-standarden) i de första programmeringskurserna kunnat klara sig bra med en kompilator som bara hanterade en väl avgränsad delmängd av Ada. De delar som rör parallella program skulle t.ex. kunna utgå.

På den kurs där eleverna tidigare hade läst en introduktionskurs i Pascal gjordes en enkät vid kursens slut. Det är intressant att notera att samtliga studenter utom en (som tyckte att valet av programspråk inte spelade någon roll) ansåg att det var en fördel att Ada användes i stället för Pascal på kursen (trots problemet med kompileringstiderna).

Sammanfattningsvis kan alltså sägas att Ada har sin givna plats i kurser i realtidssystem och i operativsystem men att det dessutom har klara fördelar jämfört med Pascal som nybörjarspråk. I flera fall upplevs Ada av eleverna som enklare än Pascal. En verkligt stor pedagogisk fördel erbjuder Ada i samband med modulär programutveckling som kan läras ut på ett naturligt sätt tidigt i undervisningen. Valet av kurslitteratur är viktigt i en grundläggande kurs. Man måste ha litteratur som är avsedd

för nybörjare. En noggrann planering av utnyttjandet av datorresurser för laborationer måste göras så att en alltför hård belastning undviks.

9. LITTERATUR

Det finns idag en omfattande litteratur om Ada och Ada-teknologin. Det är inte målsättningen att här ge en i någon mening fullständig förteckning. Det är inte möjligt eftersom den är alltför omfattande redan. Några förslag kan dock vara:

Svenska böcker:

- Jonas Agerberg, Anna-Bie Agergerg: *Ada-boken*, Studentlitteratur, 1984.

- Jan Skansholm: *Ada från början*, Studentlitteratur, 1986.

Båda böckerna försöker på ett lättfattligt sätt lära ut såväl språket som en god programmeringsteknik.

Utländska läroböcker:

- John G.P. Barnes: *Programming in Ada*, (2nd ed), Addison-Wesley Publishing Co, 1984.

- Phillip Caverly, Philip Goldstein: *Introduction to Ada, A Top-Down Approach for Programmers*, Brooks/Cole Publishing Co, Monterey 1986.

- Putnam P. Texel: *Introductory ADA, Packages for Programming*, Wadsworth Publishing Co, Belmont Calif, 1986.

Cambridge University Press publicerar i samarbete med Europeiska Gemenskapen en serie böcker om Ada och Ada teknologi: *The Ada Companion Series*. I denna serie finns titlar som

- J. McDermid, K. Ripken (Ed): *Life cycle support in the Ada environment.*

- J.C.D. Nissen, P.J.L. Wallis (Ed): *Portability and style in Ada.*

- M.W. Rogers: *Language, compilers and bibliography.*

- M. Tedd, S. Crespi-Reghizzi, A. Natali (Ed): *Ada for multi-microprocessors.*

- S.J. Goldsack: *Ada for specification: possibilities and limitations.*

- A Burns: *Concurrent programming in Ada.*

- T.G.L. Lyons, J.C.D. Nissen (Ed): *Selecting an Ada Environment.*