Software metrics: struktur samt några nya forskningsresultat

Wohlin, C., Lennselius, B. and Vrana, C.

Proceedings Milinf, pp. 10B2:27-35, Enköping, Sweden, 1986.

"SOFTWARE METRICS" - STRUKTUR SAMT NÅGRA NYA FORSKNINGSRESULTAT

Claes Wohlin*, Bo Lennselius* och Ctirad Vrana**
* Inst. f. Teletrafiksystem, Lunds Tekniska Högskola och ** Telelogic AB

1. INLEDNING

Datorer med tillhörande programvara utgör idag ett allt väsentligare inslag vad det gäller realiseringen av system. Detta gäller samtliga applikationer, såväl administrativa som tekniska. Speciellt på den tekniska sidan växer applikationerna och kraven explosionsartat; de typiska tillämpningarna för både industriella och militära system är styrning i olika sammanhang och realisering av kommunikationssystem.

Samtidigt som den så kallade hårdvaran i datorsystem har blivit billigare med tiden, utgör kostnaderna för programvara ett allt större inslag; både i relativa och absoluta tal.

Den så kallade "programvarukrisen" är ett känt fenomen. Denna kris löses inte utan att

- det tillkommer nya och effektivare programvaruhanteringsmiljöer
- programvaruprodukternas kvalitet förbättras
- kompetensen breddas och förhöjs hos de som hanterar dessa produkter.

2. SOFTWARE METRICS

En förutsättning för all industriell verksamhet är att de uppnådda resultaten kan mätas. Faktum är att det man inte kan mäta kan man heller inte styra. Den så kallade "management by objectives", se figur 1, kräver att följande tre huvudvillkor blir uppfyllda:

- Det finns (direkt mätbara) mätetal på egenskaper.
- Sambandet mellan dessa och andra storheter (ekonomiska, planeringstekniska) är kända.
- Det finns en återkoppling (organisatorisk) som kan nyttja dessa resultat i styrnings-, planerings- och andra syften.

Det råder ibland en språklig förvirring kring ordet "qualities". I den engelskspråkiga litteraturen avses oftast egenskaper, det vill säga de olika attribut som kan sammankopplas med produkterna. Det är dessa egenskaper som man försöker mäta och beskriva med "metrics". Det är först då man talar om kvalitetssäkran ("quality assurance") som det värdeladdade svenska ordet kvalitet avses. Kvalitet i, i detta fall, meningen bra eller dålig, alltså egenskaper relaterade till de ställda kraven.

"Metrics" är både av kvalitativ och kvantitativ karaktär. De kan relateras till

- egenskaper (hos programegenskaperna)
- hanteringen (av produkterna)

och indirekt också till "aktörerna" (det vill säga organisationer och individer som deltager i hanteringsprocessen).

"Metrics" utgör den teoretiska grunden till hela området "Software Engineering". Listan över de områden och sammanhang där olika "metrics" behövs kan göras mycket lång. Några typiska exempel följer nedan:

- avtal/ansvar
- kravspecifikation (av produkter)
- planering/kalkylering (av projekt)
- styrning (av projekt)
- val av metoder
- val av grundteknik (språk, hårdvara, ...)
- "quality assurance"
- test
- garantier

- återvinning av erfarenheter
- med mera

3. STRUKTUREN

Det grundläggande steget är att hitta en struktur bland olika egenskaper och se till att denna struktur är harmoniserad med andra modeller och betraktelsesätt knutna till begreppet "programvaruprodukter".

Den struktur som presenteras nedan är harmoniserad med produktbegrepp som tillämpas på programvaruprodukter inom Televerkskoncernen (och även i andra företag) och som egentligen efterliknar det gängse sättet att hantera mekaniska produkter.

I enlighet med system- och livscykelmodellerna för ett system, ref (1) och (2), definieras ett system genom dokumentationen (definitionsmässigt existerar inte systemet om inte en adekvat dokumentation finns). All hantering av ett system definieras som en transformation av dessa dokument eller utgår från dessa. Utgående från det abstrakta systemet (källsystemet) härledes (kopieras, konstrueras, tillverkas) anläggningar som är avsedda att användas i den operativa miljön, se figur 2.

De olika ekonomiska och egenskapsaspekterna kan (och bör) struktureras på ett likartat sätt. Även om strukturen inte blir helt ortogonal underlättar den avsevärt framtagning och studier av mätetal och dess tillämpning, se figur 3; observera den engelska terminologin används på grund av avsaknad av motsvarande svensk.

En studie, presenterad i ref (3), visar hur egenskaper enligt ovan påverkas i ett system som är uppbyggt med så kallad hierarkisk modulär produkt- och funktionsstruktur.

I system uppbyggda på detta sätt förbättras nästan samtliga egenskaper.

4. NÅGRA SPECIFIKA MÅTT

Som det framgår är det många egenskaper som kan vara av intresse. Vilka som anses viktiga beror på olika applikationer och de krav dessa ställer. Området i sin helhet är ej helt utrett.

Dock har vissa av måtten traditionellt och på grund av sin vikt blivit använda under längre tid och de har också utretts mera teoretiskt.

Till dessa hör de mått som enligt ovan betecknas som

- "correctness" respektive "reliability"
- "complexity" (och ur dessa härledda andra egenskaper)

samt samband mellan dessa.

Det är dessa denna artikel koncentreras på.

4.1 KOMPLEXITETSMÅTT

Introduktion

Med hjälp av komplexitetsmått kan vi mäta hur komplicerad en programvaruprodukt är. Historiskt sett har komplexitetsmåtten baserats på källkoden. Men skall vi kunna utnyttja komplexitetsmätningar som ett hjälpmedel för projektplanering och -kontroll krävs det att mätningar skall kunna göras innan kodningsfasen.

En programvaruprodukt är genom hela livscykeln definierad av dess dokumentation. Om produkten är hierarkiskt uppbyggd definierar olika dokument olika (abstraktions-)nivåer i produktens struktur. En hantering transformerar dokumenten från en nivå till en lägre nivå. Under antagandet att varje nivå dokumenteras med en väl definierad och enhetlig beskrivningsteknik, är det möjligt att kunna mäta beskrivningskomplexiteten på varje nivå.

"SOFTWARE METRICS" - STRUKTUR SAMT NÅGRA NYA FORSKNINGSRESULTAT

Beskrivningskomplexiteten (C_d) är ett mått på hur komplicerad en beskrivning (dokument) är och komplexiteten kan delas upp i olika delar; Komplexitet härrörande från storleken (C_s), från kontrollstrukturen (C_{cs}), från dataflödet (C_{da}), från beroendet mellan olika beskrivningar (programmoduler) (C_{co}) och så vidare. Detta ger oss följande relation:

$$C_d = f(C_s, C_{cs}\ C_{da}, C_{co}, \ldots.)$$

I dag finns det inget mått som täcker alla dessa aspekter på en programvaruprodukts beskrivningskomplexitet (benämns nedan enbart som komplexitet). Det är därför mycket viktigt att man vid komplexitetsmätningar utnyttjar olika typer av mått, vilka mäter olika delar av komplexiteten.

Tillsammans med andra faktorer, såsom exempelvis projektmedlemmarnas kompetens, påverkar komplexiteten den mänskliga hanteringen av dokumenten under både utvecklings- och underhållsfasen. Under antagandet att inverkan från de övriga faktorerna är lika för alla moduler i samma produkt (projekt) erhåller vi följande samband mellan produktens felinnehåll (FEL), respektive krävd arbetsinsats (ARB) och komplexiteten:

$$FEL \simeq k_1\ C_s^{z1} + k_2\ C_{cs}^{z2} + k_3\ C_{da}^{z3} + k_4\ C_{co}^{z4} + \ldots\ldots$$

$$ARB \simeq k_5\ C_s^{z5} + k_6\ C_{cs}^{z6} + k_7\ C_{da}^{z7} + k_8\ C_{co}^{z8} + \ldots\ldots$$

där k_i och z_i är konstanter som baseras på beskrivningsspråk, programmeringsspråk, metodik och hjälpmedel.

Tillämpningar

En mycket viktig tillämpning av komplexitetsmätningar är att identifiera felintensiva moduler i produkten, för att på så sätt kunna göra en kostnadseffektiv tilldelning av granskningsresurser, testresurser etc.

Via erfarenhet från liknande projekt (produkter) kan de mest signifikanta felgenererande respektive arbetsintensiva delarna av komplexiteten identifieras och tillhörande konstanter kan bestämmas. På så sätt kan antalet fel i produkten och krävd arbetsinsats för efterföljande hanteringar predikteras. Genom att studera sambandet mellan komplexiteten och programvaruhanteringarna erhålles riktlinjer för hur produkten skall struktureras för att minimera livscykelkostnaden.

Genom att mäta komplexitetens inverkan på programvaruhanteringarna kan vi identifiera de svaga delarna i utvecklingsmetodiken. Dessutom ger komplexitetsmätningar en objektiv grund för jämförelse mellan olika typer av utvecklingsmetoder (-miljöer).

Några resultat

Nedan presenteras några resultat från en undersökning av komplexiteten hos 15 moduler (block) i ett nytt delsystem i telefonisystemet AXE (projekt A). Undersökningen, som gjordes 1985, har jämförts med liknande undersökningar som tidigare gjorts på 20 (projekt B) respektive 28 moduler (projekt C) i AXE-systemet, se vidare ref (4), (5), (6) och (7).

Dessa moduler är beskrivna med en SDL-liknande beskrivning. SDL (Specification and Description Language, ref (8)) är ett standardiserat beskrivningsspråk som väl uppfyller kravet på en väl definierad och enhetlig beskrivningsteknik. Det primära målet med dessa undersökningar var att studera huruvida komplexitetsmätningar i SDL-beskrivningar kan utnyttjas för att till exempel identifiera felintensiva moduler.

I studien utnyttjades bland annat följande komplexitetsmått. De två sista måtten är kodbaserade, medan de övriga erhölls ur de SDL-liknande beskrivningarna.

SIG: Ett mått som mäter beroendet mellan en modul och dess "omvärld", genom beräkning av antalet unika insignaler.

SYMB: Som ett mått på en moduls storlek beräknades antalet SDL-symboler.

C(G): Ett mått som mäter en moduls kontrollstruktur. C(G) är en modifiering, ref (9), av "McCabe's Cyclomatic Complexity", ref (10).

MSYMB, MC(G): För projekt A modifierades måtten SYMB och C(G) med hänsyn taget till likheter i beskrivningen för olika delar av ett block, ref (4).

V: Som ett mått på en moduls storlek beräknades från koden "Halstead's Programvolym, ref (11), för projekt A.
KOD: Antalet rader kod.

Genom att utnyttja nedanstående teknik kan vi få svar på följande frågor:
- Är moduler med hög komplexitet (som den uttrycks via våra mått) goda indikatorer på felintensiva block?
- Är de SDL-baserade måtten bättre eller sämre på att identifiera de felintensiva modulerna än vad antalet rader kod är?

Fördelningen över antalet fel per modul beräknas för respektive projekt. Beräkna medelvärdet och standardavvikelsen. Definiera en modul som felintensiv om dess felinnehåll är större än medelvärdet adderat med standardavvikelsen. Definiera på liknande sätt "komplexitetsintensiva" moduler för respektive mått.

Av våra 63 moduler var 11 stycken så kallade "felintensiva". Dessa representeras av kolumnerna i tabell 1, medan X markerar vilka av dessa moduler som identifierades på grund av att de var "komplexitetsintensiva" för respektive mått. I kolumnen "Icke felintensiva" anges antalet moduler som felaktigt identifierades som "felintensiva", det vill säga moduler som inte var "felintensiva" trots att de var "komplexitetsintensiva".

Mått	"Felintensiva" Moduler 1 2 3 4 5 6 7 8 9 10 11	Summa "Felintensiva" moduler	Summa "icke felintensiva" moduler
KOD	x x x x x x	6	3
SYMB	x x x x x x x	7	3
C(G)	x x x x x x x	7	4
SIG	x x x x x x x x x x	10	2

Tabell 1 "Felintensiva" moduler mot "komplexitetsintensiva" moduler.

Från tabell 1 kan vi konstatera att de felintensiva modulerna kan identifieras relativt bra med hjälp av komplexitetstalen. Dessutom ser vi att de SDL-baserade måtten är minst lika bra indikatorer som det ofta använda måttet "antalet rader kod". Detta ger oss en möjlighet till att identifiera felintensiva moduler tidigt i programvarans livscykel (till exempel kan SIG erhållas så tidigt som ur den statiska SDL-beskrivningen).

Genom regressionsanalys har förhållandet mellan antalet fel och våra komplexitetstal studerats. Beroendet mellan antal fel och de SDL-baserade måtten, som det uttrycks via korrelationen i kvadrat (r^2_A), är minst lika stort som beroendet mellan antalet fel och de kodbaserade måtten (I tabell 2 visas dessa resultat för projekt A, när linjärt beroende antages). Detta indikerar att antalet fel i produkten kan skattas tidigt i livscykeln med hjälp av de SDL-baserade måtten.

Mått	V	KOD	SIG	MSYMB	MC(G)
r^2_A	0.87	0.75	0.89	0.85	0.84

Tabell 2 Korrelationen i kvadrat (r^2_A) mellan antalet fel och respektive komplexitetstal för projekt A.

Närmare studier av förhållandet mellan antalet fel och komplexitetstalen visar att detta förhållande är olinjärt, se exempelvis figur 4. En studie av sambandet mellan tiden att hitta medvetet införda fel i SDL-beskrivningar och beskrivningarnas komplexitetstal visade att även detta förhållande är olinjärt, ref (9). Detta ger oss en indikation på att allt för komplexa moduler kan bli mycket felintensiva och svåra att hantera under underhållsfasen, vilket visar vikten av att tidigt kunna detektera allt för komplexa moduler och eventuellt dela upp dessa i moduler med lägre komplexitet, för att på så sätt minska det totala felinnehållet i produkten och minska den totala livscykelkostnaden för produkten.

I projekt A studerades också sambandet mellan kodningstid och de olika komplexitetstalen. Även i detta fall visade det sig att korrelationen mellan kodningstid och de SDL-baserade måtten är lika hög som för de kodbaserade måtten.

4.2 TILLFÖRLITLIGHET

Inledning

En av de viktigaste kvalitetsaspekterna är tillförlitligheten hos programvaran. Tillförlitligheten hos ett program är sannolikheten att programmet utför vad det är meningen att det skall utföra under en specificerad tid. Ett programs tillförlitlighet är i hög grad beroende av antalet fel man har gjort när man skrev programmet, eftersom man inte har samma utslitningsfenomen som för hårdvaran. Detta är dock en sanning med modifikation, då man ofta ändrar, förbättrar och gör tillägg i programvaran under dess livslängd.

För att få en uppfattning om tillförlitligheten vill vi kunna bestämma faktorer som; återstående antal fel, tid mellan fel, det vill säga överhuvudtaget prediktera den fortsatta felutvecklingen hos programvaran. Vi kan redan på ett tidigt stadium skaffa oss en uppfattning om denna utveckling, då vi ur kompexitetsmått kan skatta antalet fel i vår programvaruprodukt, se under Komplexitetsmått. Denna skattning kan sedan användas som indata till våra tillförlitlighetsmodeller. Resultaten från modellerna kan förbättras efterhand som projektet framskrider genom insamling av felstatistik och med hjälp av denna förfina skattningarna av modellernas parametrar. Genom att skatta felutvecklingen är det möjligt att prediktera exempelvis en lämplig tidpunkt att släppa programvaruprodukten på och prediktera hur allokeringen av testresurser bör ske, för att klara av att hålla utlovade leveranstider och utlovad kvalitet.

Modellernas giltighetsområden

En realistisk prediktering av den framtida felutvecklingen är naturligtvis ett måste för att kunna dra några slutsatser från den. Detta är i hög grad beroende av att man använder tillförlitlighetsmodeller som är väl anpassade till den situation som råder. Modellerna måste vara rimliga med avseende på en rad faktorer som till exempel utvecklingsmiljö, hjälpmedel, applikation, testmiljö etc. För att finna de modeller som är möjliga att använda för våra produkter krävs en noggrann undersökning, klassificering och jämförelse av existerande modeller, samt en studie av vår egen miljö. Ett första steg mot en jämförelse av existerande modeller presenteras i ref (12).

Resultaten av undersökningarna kan antingen bli att vi finner några lämpliga modeller, eller att vi kommer fram till att det inte finns någon som väl modellerar vår miljö. Det är av stor vikt att man inte bara tar en modell och använder den, utan att noggrannt studera dess antaganden och tillämpningsområden. I de fall vi ej finner någon modell alls blir vi tvungna att utveckla egna modeller eller acceptera att vi inte kan uttala oss om den framtida felutvecklingen. Ett accepterande av det sistnämnda innebär ett stort steg i riktning mot att tappa kontrollen över vår programvaruprodukt.

I ref (13) presenteras ett exempel på hur en undersökning av en specifik testmiljö kan leda fram till att en tillförlitlighetsmodell, som är anpassad till rådande förhållanden, utvecklas. Denna undersökning ledde också fram till att vi kan konstatera att vi inte kan förvänta oss att finna en global modell, utan att vi behöver olika modeller under olika faser i programvarans livscykel. Modellerna måste anpassas till respektive fas och när en övergång till en ny aktivitet sker måste vi acceptera att vi behöver en ny tillförlitlighetsmodell. Flertalet av de "klassiska" programvarutillförlitlighetsmodellerna är anpassade till driftsliknande förhållanden, vilket för det mesta inte råder under test av ett system.

Det är mycket viktigt att utveckla modeller som är tillämpbara under test, då test är ett av våra bästa "vapen" mot programvarufel. Det vill säga att testa vår produkt ger oss en tillförlitligare produkt. Om vi har tillförlitlighetsmodeller för test har vi en möjlighet att bestämma programvarans tillförlitlighet vid olika tidpunkter, till exempel beräknad release-tidpunkt. Därmed har vi kommit in på en mycket viktig tillämpning av våra modeller, nämligen som hjälpmedel för ekonomisk optimering av vårt programvaruprojekt.

Expempel på tillämpning

Det är allmänt känt (och accepterat) att kostnaden för rättning av fel ökar ju längre vi har hunnit i vårt projekt, samtidigt som det (naturligtvis) inte lönar sig att testa hur länge som helst. Detta enkla resonemang leder fram till att det är uppenbart att det finns ett ekonomiskt

optimum, då vi skall avsluta testen och släppa produkten. Antag att vi undersöker hur kostnaderna fördelar sig och på så sätt kan komma fram till en ekonomisk modell för våra programvaruprojekt Det är ju sedan helt naturligt att vi vill testa tills vi uppnått detta optimum.

Det finns då två stycken principiellt olika angreppssätt när det gäller testfilosofin:

- resurserna bestämda, det vill säga vissa resurser har tilldelats projektet och det kan inte ändras.
- release-tidpunkten bestämd, det vill säga vi måste vara klara till en viss tidpunkt oavsett vilka resurser som krävs.

Den normala situationen är oftast en hybrid av dessa principiellt olika tillvägagångssätt. Eftersom vi nu antar att vi funnit eller utvecklat realistiska tillförlitlighetsmodeller för vår miljö kan vi nu beräkna

- förväntad tidpunkt för att uppnå optimum och hur den varierar, det vill säga med bestämda resurser.

eller

- bestämma resurserna som krävs för att med en viss sannolikhet uppnå vår utlovade release-tidpunkt.

Vi kan naturligtvis genomföra ovanstående beräkningar med alla tänkbara kombinationer av resurser och release-tidpunkter. Utgående ifrån resultaten kan vi dra slutsatser om vilka åtgärder vi behöver sätta in för att klara av de krav som ställs på vår programvaruprodukt, både vad det gäller tillförlitlighet, tid och ekonomi.

En oerhört viktig aspekt angående beräkningar ovan, är att vara medveten om och planera för de stokastiska variationer som vi har i uppträdandet hos programvarufel. Det är inte realistiskt att bara räkna med medelvärden utan vi måste också ta hänsyn till varianserna. I figur 5 visas det principiella resultatet som är en direkt följd av den stokastiska process som ligger bakom programvarufelen och dess upptäckt. Följande definitioner har använts:

* N_t - antal fel vid en godtycklig tidpunkt t
* N_0 - initialt antal fel
* ΔN_0 - osäkerheten i N
* N_{opt} - antal återstående fel då vi har uppnått det ekonomiska optimum
* ΔT_e - osäkerheten i tidpunkten då vi uppnår det ekonomiska optimum
* ΔT - den totala osäkerheten i tidpunkten då vi uppnår det ekonomiska optimum

I figur 5 ser vi hur antalet fel initialt varierar inom intervallet (N_0-ΔN_0, N_0+ΔN_0) och hur de sedan avtar med tiden. Vi ser hur tidpunkten då vi når N_{opt} varierar för de tre fallen då antalet fel startar på N_0-ΔN_0, N_0 och N_0+ΔN_0. Detta ger upphov till en total osäkerhet i tidpunkten som motsvarar ΔT.

ΔT är med stor säkerhet alldeles för stort och det krävs en rad åtgärder för att komma tillrätta med problemet. En noggrannare undersökning av konsekvenserna av olika fördelningar för tid mellan fel presenteras i ref (14), där även problemet, beräkningar och åtgärder diskuteras närmre. Vi kan dock konstatera att vi har ett antal tänkbara åtgärder, för att med större säkerhet kunna uttala oss om när vi uppnår det ekonomiska optimum, och de är

- minska initialt antal fel
- bättre skattning av initialt antal fel
- bättre testmetoder

Om vi inte lyckas med detta finns två möjligheter, båda oekonomiska

1. Vi testar för länge, för att vara på den "säkra" sidan.
2. Vi avslutar testen för tidigt, vilket kan leda till en ineffektiv, otillförlitlig, okontrollerbar och ej underhållsbar programvaruprodukt.

5. AVSLUTNING

Inom de ovan beskrivna delområdena har man vid LTH i samarbete med Telelogic, under tre års forskning byggt upp en kompetens. De framtagna nya resultaten börjar bli av intresse även internationellt.

Utöver de två områden som ingående studerats kommer det fortsatta arbetet även att inrikta sig på studier av andra egenskaper. Närmast ligger problemområdet kring effektivitet under "run-time" och andra kapacitetsfrågor.

Samtidigt med de teoretiska studierna har modellerna testats (som framgår) inom ett antal projekt; detta dock främst i efterhand, med projektfacit i handen.

För att nu använda "metrics" och modellerna i den operativa verksamheten, förmedlas resultaten till de som ansvarar för metodikfrågor, konstruktion av hjälpmedel och för QA.

I det närmaste kommer de olika sambanden vad det gäller strukturering och komplexitet att användas inom de programutvecklingsmiljöer som konstrueras vid Telelogic kring SDL (CCITT:s specifikationsspråk, se ref (8)) och kring Ada.

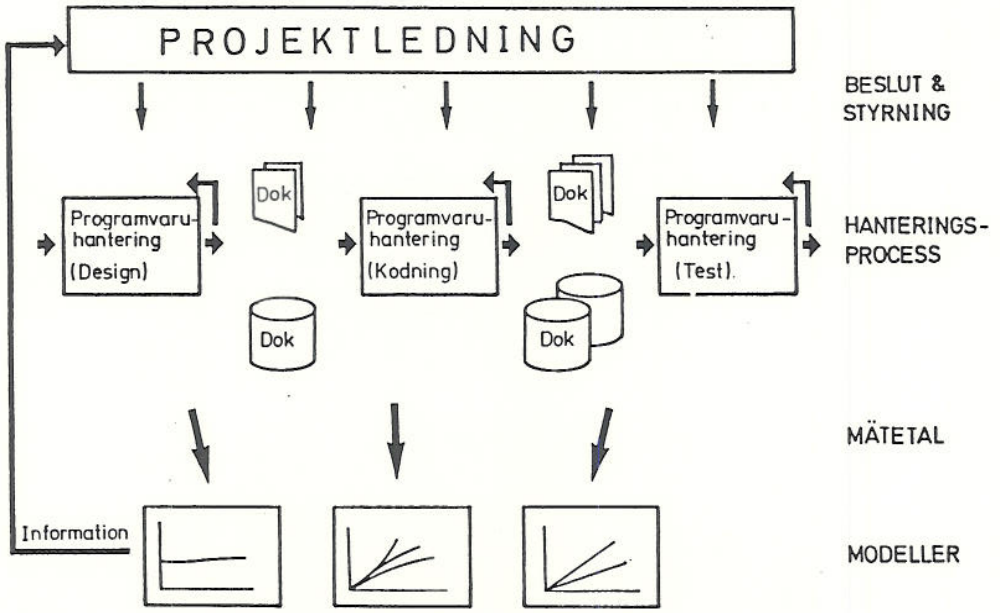
Det finns också prototyp till verktyg för uppföljning av "reliability" hos produkterna; den operativa tillämpningen studeras vidare.

På sikt kommer även andra egenskaper att kunna mätas och kontrolleras i den integrerade hanteringsmiljön.

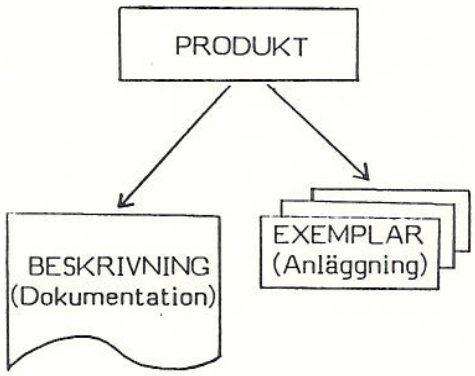
På det sättet, tror vi, att man på ett väsentligt sätt förbättrar möjligheterna till att bemästra problemen kring programvaruhanteringen.

6. REFERENSER

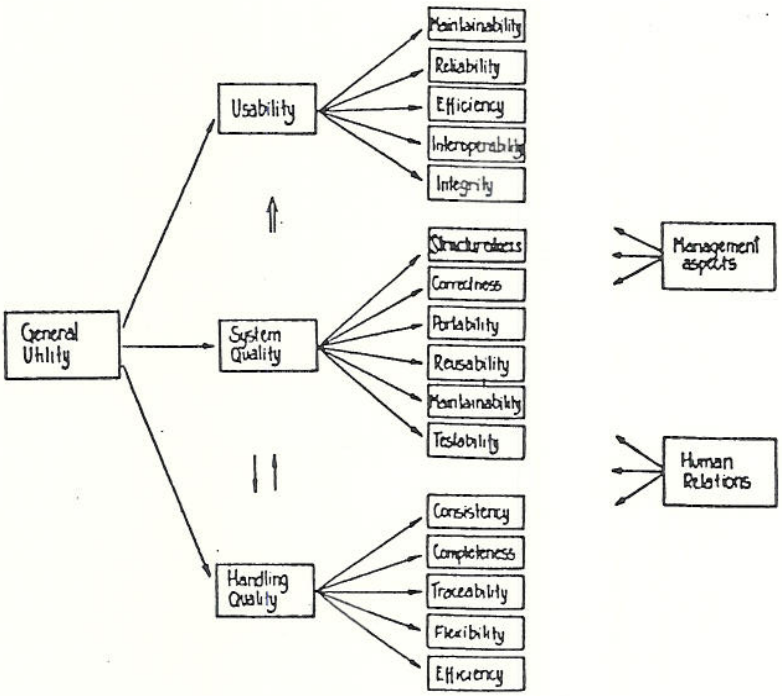
1. Rapp, D., and Vrana, C., 1984, "Systems and System Management Environment", NT-symposium, Turku, Finland.
2. Vrana, C., 1983, "Livscykelmodell för tekniska system", Intern rapport, Telelogic.
3. Vrana, C., 1984, "S/W Engineering Economics - Models for Systems with a Hierarchical Modular Structure", NT-symposium, Turku, Finland.
4. Lennselius, B., 1986, "Software Compexity and its Impact on Different Software Handling Processes", Proc. 6th Int. Conf. on Software Engineering for Telecommunication Switching Systems, Eindhoven, The Netherlands.
5. Vrana, C., and Wallander, A., 1983, "S/W Quality and Complexity - Different Aspects and Measurement Results", Proc. 5th Int. Conf. on Software Engineering for Telecommunication Switching Systems, Lund, Sweden.
6. Wallander, A., 1982, "Programvarukvalitet och -komplexitet", Examensarbete, Lunds Tekniska Högskola.
7. Wohlin, C., 1983, "En studie av programvarukomplexitet", Examensarbete, Lunds Tekniska Högskola.
8. CCITT, 1981, "Recommendations Z101-104, Yellow Book", Volume VI, Fascicle VI.6, Geneva, Switzerland.
9. Lennselius, B., and Vrana, C., 1985, "Complexity in the SDL-Graph-Description and its Impact on Different Handling Activities", Proc. 2nd SDL-Users and Implementors Forum, Helsinki, Finland.
10. McCabe, T., 1976, IEEE Trans. on Software Engineering, Vol SE-2, No. 4, 308-320.
11. Halstead, M., 1977, "Elements of Software Science", Elsevier North-Holland Pub. Co., New York, USA.
12. Wohlin, C., 1986, "A Comparison Between Two Software Reliability Models: Jelinski-Moranda's De-Eutrophication Model and Goel-Okumoto's Non-Homogenous Poisson Process Model", Technical Report, Lund Institute of Technology, Lund, Sweden.
13. Wohlin, C., 1985, "Software Testing and Reliability for Telecommunication Systems", Technical Report, Lund Institute of Technology, Lund, Sweden.
14. Wohlin, C., and Vrana, C., 1986, "A Quality Constraint Model to be Used During the Test Phase of the Software Lifecycle", Proc. 6th Int. Conf. on Software Engineering for Telecommunication Switching Systems, Eindhoven, The Netherlands.



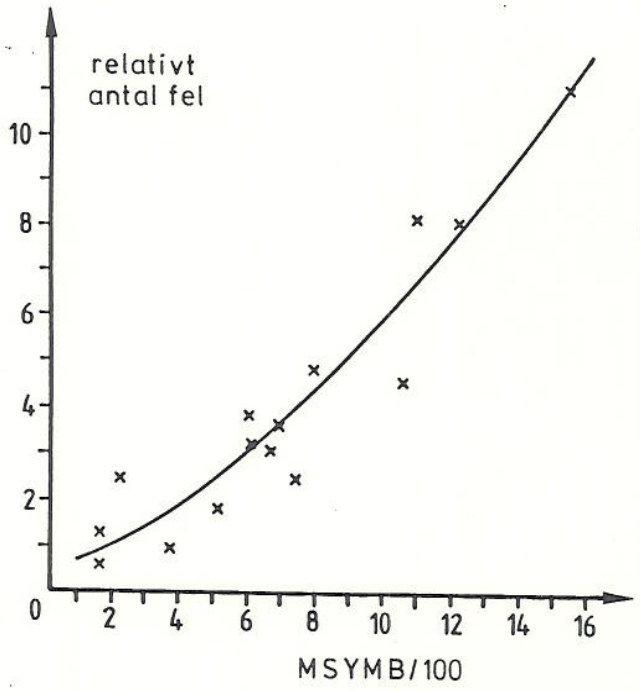
Figur 1 Projektledning för ett programvaruprojekt.



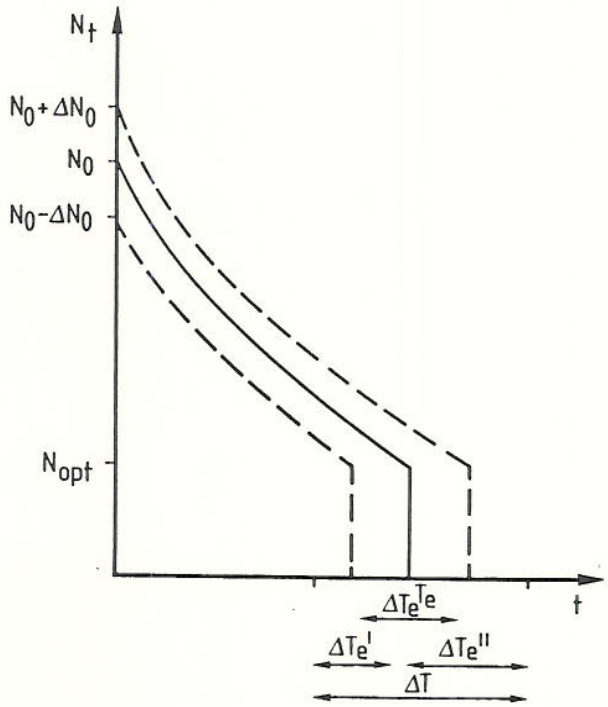
Figur 2 Analys av produktbegreppet.



Figur 3 A Criteria Structure.



Figur 4 Förhållande mellan antal fel och MSYMB för projekt A.



Figur 5 Antalet återstående fel i programvaruprodukten och variationen hos tidpunkten då vi uppnår vårt ekonomiska optimum.

BIBLIOGRAFIER

Wohlin, Claes:
Civilingenjörsexamen (Elektroteknik) 1983 från Lunds Tekniska Högskola. Anställd sedan 1983 vid Institutionen för Teletrafiksystem, Lunds Tekniska Högskola som forskningsassistent, och arbetar för närvarande i ett forskningsprojekt inom området "Software Engineering". Doktorandstudierna inriktas speciellt på programvarutillförlitlighet.

Lennselius, Bo:
Civilingenjörsexamen (Elektroteknik) 1983 från Lunds Tekniska Högskola. Anställd sedan 1983 vid Institutionen för Teletrafiksystem, Lunds Tekniska Högskola som forskningsassistent, och arbetar för närvarande i ett forskningsprojekt inom området "Software Engineering". Doktorandstudierna inriktas speciellt på programvarumått och -modeller.

Vrana, Ctirad:
Civilingenjörsexamen (Elektroteknik) 1976 från Lunds Tekniska Högskola. Forskningsassistent (område Datorstyrning) vid Institutionen för Teletrafiksystem, Lunds Tekniska Högskola, 1976-82. 1982-83 anställd vid tekniska utvecklingsavdelningen på Televerkets Huvudkontor, metodik och kvalitetsfrågor för programvara. Teknologie Licentiat vid Institutionen för Teletrafiksystem, LTH, 1984, inriktning "Software Engineering". För närvarande ansvarig för utbildningsverksamheten vid Telelogic AB.