Handelshögskolan vid Göteborgs Universitet
Institutionen för Informatik

VT-2001

Att designa grafiska användargränssnitt för testning av inbäddade system

Magisteruppsats i Informatik

Författare: Jona Bolin
Per Jacobsson
Torbjörn Håkansson

Handledare: Kjell Engberg

Abstrakt

Uppsatsens syfte är tvådelat. Dels att presentera ett teoretiskt möjligt förslag till hur ett grafiskt användargränssnitt för testning av inbäddade system optimerat för expertanvändare bör se ut, dels att undersöka hur detta förslag påverkas av praktiska erfarenheter. Med utgångspunkt i en fallstudie gjord i samarbete med Ericsson Mobile Data Design AB samt i teori om testning och gränssnittsdesign kommer vi att resonera oss fram till en hypotes - 12 teoretiska riktlinjer - vilka beskriver hur ett grafiskt användargränssnitt bör utformas så att användbarheten förbättras. Utifrån dessa riktlinjer utvecklar vi sedan en gränssnittsprototyp vilken fungerar som underlag för en, genom användbarhetstester och intervjuer, prövning av gränsnittets kvalitet och riktlinjernas relevans. Resultatet visade sig vara framgångsrikt och gränssnittsprototypen hade en hög kvalitet, men vi fann ändå anledning att revidera hypotesen.

I vår slutsats finner vi att riktlinjerna är välgrundade och adekvata med undantag för tre vilka ansågs vara för generella och därmed ströks, samt att ytterligare några justerades om. Den största skillnaden mellan det teoretiska förslaget och de praktiska erfarenheterna handlar dock om den strukturella synen på riktlinjerna. Vi frångår en enkel lista där varje riktlinje väger lika tungt. Det nya förslaget utgörs istället av två huvudpunkter vilka har större vikt än de resterande sju vilka i sammanhanget betecknas som underordnade.

Vi vill rikta ett stort tack till våra handledare - Kjell Engberg vid Institutionen för Informatik samt guru Björn Lindberg på Ericsson Mobile Data Design AB.

Del 1 Introduktion **7**
Testning *7*
Användargränssnitt *8*
Problemformulering *9*
Del 2 Metod **11**
Hypotesprövning *11*
Intervjuer *12*
Prototyping *12*
Användbarhetstester *14*
Del 3 Teoretiskt Ramverk **17**
3.1 Grafiskt användargränssnitt **18**
Målinriktad design *18*
Användarens mål 18
Mjukvarudesign 19
Tre modeller 19
Visuellt användarinterface 20
Form *22*
Tre gränssnittsparadigm 22
Fönsterhantering 23
Dokumenthantering 23
Beteende *25*
Flöde 26
Stämning och tillstånd 27
Ett programs tillstånd 28
Interaktion *29*
Musen 29
Tangentbordet 30
Taktiska verktyg *30*
Menyer 31
Dialogfönster 32
Verktygsfält 35
Gizmos 35
3.2 Testning **37**
Testning i mjukvaruutveckling *37*
Statisk testning 38
Dynamisk testning 38
Testning i livscykelmodellen *38*
Livscykelmodellen och dess faser 38
Testning som ”a continous improvement process” 39

Acceptance testing 40
System testing 40
Integration testing 41
Unit Testing 41

Unittestning och Debugging 41
Testfall 42
Debugging kontra testning 43
Blackbox och whitebox 44

Gränsnitt för testverktyg 44
Software Monitoring 44
Gränssnitt – för utvärdering av testresultat 46

3.3 Kvalitet 48

ISO-modellen 48
ISO 9126 48

Del 4 Förberedande analys 51

Användarna 51
Vem är användaren? 51

Miljö 52
Testningsprocessen 53
Testmiljön 55

Del 5 Hypotesformulering 61

5.1 Användarna 62

Kontrollbehov 62

Utrymmesbehov 63

Behov av stöd - som hjälp och feedback 63

Installationsprocessen 64

5.2 Miljö 65

Frågor relaterade till avgränsning 65
Vad finns tillgängligt och vad behövs? 65
Hur kan informationen i loggfilerna användas? 66
Hur skall loggfilerna redovisas? 66

Frågor relaterade till stöd för testning 67
Hur presenteras noden? 67
Hur konfigureras noden? 68
Hur hanteras processerna? 68

5.3 Avbildning 70

Användarens mål 70

Form 71
Att tänka idiomatiskt 71

Fönsterhantering ... 72
Dokumenthantering ... 72
Meny ... 72
Avbildning av objekt och processer ... 73

Beteende och tillstånd ... 75
Flöde ... 75

5.4 Härledda förutsägelser ... 79

Del 6 Hypotesprövning ... 83

Intervjuresultat ... 83

Resultat av användbarhetstest ... 88
Testscenario 1 ... 88
Testscenario 2 ... 89
Sammanfattning av användbarhetstestets resultat ... 90
Diskussion kring användbarhetstestet ... 91

Del 7 Diskussion och slutsats ... 93

Utvärderingens tillförlitlighet ... 93
Intervjuer ... 93
Användbarhetstest ... 93

Reflexioner kring användbarhetstesterna ... 94

Diskussion - Riktlinjer ... 95

Slutsats - Riktlinjer ... 99
Reviderade riktlinjer ... 99
Sammanfattning ... 102

Litteraturförteckning ... 103

Appendix 1 BUNNY DOCUMENTATION ... 105

Appendix 2 BUNNY MANUAL ... 117

Del 1 Introduktion

Denna uppsats har utförts i samarbete med Ericsson Mobile Data Design AB (ERV) som är ett företag i Ericsson-koncernen. En avdelning inom ERV utvecklar en mjukvarumiljö som kallas Distributed Processing Environment (DPE). DPE är en del av ett betydligt större system, Wireless Packet Plattform (WPP), som hanterar mobil datapaketstrafik även kallad *det trådlösa Internet*. Dessa system kan till exempel användas som paketdataväxlar för datakommunikation inom mobilsystem som GPRS[1] och UMTS[2] och faller under samlingsnamnet nod. En nod består av en mängd rackmonterad hårdvara i form av olika typer av kretskort och processorer. Noden är öppen för olika hårdvarukonfigureringar och kan köra flera operativsystem samtidigt.

Konkret så erbjuder DPE en miljö där applikationer på högre nivåer kan exekvera. DPE håller reda på vilka hårdvaruenheter som finns tillgängliga, distribuerar och kontrollerar processer över noden, samt erbjuder stöd för kontinuerlig, tillförlitlig och robust exekvering.

Vid vidareutveckling av DPE finns det idag två möjligheter för utvecklarna att testköra det som ändrats i sitt naturliga sammanhang. Det första är att boka en tid och köra mot en riktig nod. Ett förfarande som är tidsödande och krångligt. Det andra är att använda ett program som simulerar en nod och gör att DPE går att testköra på en vanlig arbetsstation. Detta program kallas för arbetsstationslösningen (ASL) och innebär, i förhållande till att köra mot en riktig nod, betydande effektivisering av testningsprocessen. Problemet med ASL är att den är att den är svår att använda. Gränssnittet är kommandobaserat och för att DPE ska fungera som användaren tänkt sig krävs omfattande och fortlöpande justeringar i ett antal olika filer. Vid varje ny uppstart måste samma tidsödande förfarande upprepas och risken för att användaren råkar ställa in fel inställningar är hela tiden betydande.

Det som ERV söker är en lösning som förbättrar användbarheten av ASL. Detta genom att utveckla ett grafiskt användargränssnitt för att samordna och styra all splittrad funktionalitet. Testningsprocessen ska därigenom ytterligare effektiviseras.

Målgruppen för denna uppsats är först och främst ERV-personal, dvs de utvecklare och testare vi varit i kontakt med under vårt arbete, men i viss mån även andra, utomstående som finner ett intresse i området .

Testning

Datorer används i en mängd olika sammanhang, från att kontrollera enkla hushållsmaskiner till att styra automatiserade tillverkningsindustrier. Mjukvaran som styr sådana datorer interagerar med andra system och annan hårdvara. Den är inbäddad i ett större hårdvarusystem, så kallade *Embedded Systems*[3], och reagerar på händelser i systemets miljö. Eftersom sådana system ”sköter sig själva” och så begränsas gränssnittet mot eventuella användare till möjligheten att starta och stänga av. Det finns även möjlighet att ändra programmets inställningar men detta sköts oftare genom att manuellt gå in och ändra i

[1] General packet radio service
[2] Universal Mobile Telecommunications System
[3] Sommerville, Ian, *Software engineering*, Pearson Education, 2001, sid 286

konfigureringsfiler än med grafiska menyval. Saknaden av ett mer omfattande användargränssnitt är motiverat av det faktum att det inte krävs någon avancerad interaktion med användaren, förutom i vissa fall, till exempel vid testning.

Testning av mjukvarusystem sköts ofta av mjukvaruutvecklarna själva eller professionella testningsavdelningar. Inledningsvis är det programmeraren själv som testkör den senaste koden och utför den övergripande testningen av den modul han utvecklar. Detta kan vara svårt att realisera om det mjukvarusystem som skall utvecklas eller vidareutvecklas är ett inbäddat system. Mjukvarusystemet testas inte som helhet utan del för del. Då är det inte alltid meningsfullt att köra det på den riktiga hårdvaran, ”*the target hardware system”*. Dessutom saknar hårdvaran kanske presentationsmedel (ex. bildskärmar, utskriftsfaciliteter etc.) varpå man antingen måste koppla den mot en extern dator med tillgång till presentationsmedel eller installera mjukvaran på en sådan dator. För att testa mjukvaran krävs det stöd för att inhämta relevant och adekvat information om systemet, vilket sker via ett testverktyg. Testverktyget bör erbjuda en möjlighet att starta, stoppa, övervaka och styra systemet. Om mjukvaran dessutom installeras på främmande hårdvara behöver verktyget imitera den ursprungliga hårdvarumiljön så långt som möjligt.

Testverktyg för ett inbäddat system är ofta utvecklade av samma team som utvecklar systemet i syfte att underlätta implementerings- och testningsarbetet. Testverktygen är snabba lösningar på ett akut problem och fokuserar på att erbjuda användaren den nödvändiga funktionaliteten. På grund av tidsbrist hinner man ofta inte att lägga samma kraft på användbarheten. Ur ett kort perspektiv är detta acceptabelt, men om utvecklingsarbetet tar lång tid eller systemet kontinuerligt vidareutvecklas ökar behovet av ett bättre gränssnitt. Med förbättrad användbarhet, förenklas hanteringen och testningsprocessen effektiviseras. Vilket i sin tur sparar tid, pengar och förbättrar slutprodukten.

Användargränssnitt

Att utveckla ett grafiskt användargränssnitt följer till stor del en allmän standard som oberoende av användningsområde uppvisar stora likheter program emellan. Men det finns fundamentala skillnader vilka påverkas av typen användare programmet kommer att ha och i vilket sammanhang programmet ska verka. De verktyg med vars hjälp utvecklingen av ett GUI (Graphical User Interface) kan underlättas är enligt Alan Cooper i *About face the essential of user interface design*[4] av två typer, taktiska och strategiska. Taktiska verktyg består av praktiska anvisningar i hur man skapar gränssnittsidiom och strategiska verktyg lägger grunden för hur idiomen och användarna interagerar.

Mycket av systemets kraftfullhet, flexibilitet och användbarhet är relaterat till dess användargränssnitt. Det är den här delen av systemet som användaren ser, vilket medför att gränssnittets egenskaper i hög grad påverkar hur användaren uppfattar systemet. Men det finns ingen universallösning som exakt beskriver hur ett GUI ska se ut i alla situationer och för alla typer av användare. Visst kan man skapa något som är visuellt praktfullt genom god användning av taktiska verktyg men om kunskapen om hur ett GUI används saknas blir den visuella prakten betydelselös.

En nyckelutgångspunkt vid GUI-utveckling är att kunna formulera vilka mål användaren har med programmet. Målen består fundamentalt sett oftast av att någon vill få något utfört på

[4] Alan Cooper, *About face the essential of user interface design*, IDG Books Worldwide, 1995

enklast och snabbast möjliga vis och utan att hindras och ifrågasättas. Problemet för utvecklare är att dessa mål inte alltid är enkla att urskilja och hålla fast vid då en mängd olika intressenter och användare är inblandade i programdesignen. Det gäller att uppmärksamma och förstå att de kvalitetskriterier som bör uppfyllas vid utveckling av ett GUI är av användarcentrerad natur inte av teknologicentrerad. Detta innebär att de modeller som designen baseras på ska reflektera användarnas konceptuella syn inte den bakomliggande teknologin.

Problemformulering

Genom att studera en organisation som arbetar med mjukvaruutveckling och testning av inbäddade system kommer vi att illustrera vilka särskilda förutsättningar som råder i denna miljö. Detta för att kunna utveckla ett grafiskt användargränssnitt som ersätter och kompletterar ett kommandobaserat. Med utgångspunkt i en fallstudie gjord i samarbete med ERV samt i teori om testning och gränssnittsdesign kommer vi att resonera oss fram till ett antal teoretiska riktlinjer vilka beskriver hur ett gränssnitt bör utformas så att användbarheten förbättras. Vi utgår från följande generella frågeställning för att komma fram till riktlinjerna:

Hur utformas ett grafiskt användargränssnitt för testning av inbäddade system med syfte att optimera användbarheten för expertanvändare?

Riktlinjerna och den förutsagda effekt dessa får för användbarheten utgör vårt ställningstagande, vår hypotes. Vi påstår:

Genom att följa riktlinjerna för utformning av grafiska användargränssnitt kommer användbarheten förbättras.

Ställningstagandet prövas genom att vi utvecklar en prototyp utifrån riktlinjerna. Vi undersöker sedan riktigheten i hypotesen genom att utföra användbarhetstester på prototypen. Slutligen diskuterar vi resultatet av undersökningen och korrigerar eventuella brister genom att revidera våra riktlinjer och ställer därmed frågan:

Fungerar riktlinjerna i praktiken?

Syftet med uppsatsen är därmed tvådelat. Dels att presentera ett teoretiskt möjligt förslag på hur ett gränssnitt i den beskrivna miljön ska se ut, dels att undersöka hur detta förslag påverkas av praktiska erfarenheter.

Del 2 Metod

Ett delmål i vårt arbete har varit att samla in tillräckligt mycket primärdata för att kunna förstå de behov och krav på funktionalitet som finns från användarna och därigenom kunna se lösningar som användarna själva för tillfället inte kan se. Detta ställer stora krav på att rätt användare intervjuas och att rätt frågor behandlas.

Ett ytterligare delmål har varit att skapa ett grafiskt användargränssnitt som uppfyller ett antal kvalitetskriterier och som avsevärt underlättar användarnas arbetsvardag. Även här handlar det om att finna lösningar som användarna kanske inte själva är medvetna om. För att lyckas med detta ställs hårda krav på såväl primärdata som sekundärdata.

Syftet med uppsatsen är att beskriva ett teoretiskt möjligt förslag på en gränssnittsdesign vilket kommer att mynna ut i ett antal hypotespunkter. Som stöd i framtagandet av dessa använder vi litteratur om gränssnittsdesign och testning samt en omfattande användar- och miljöanalys. Punkterna prövas sedan genom användbarhetstester och uppföljningsintervjuer för att slutligen revideras och anpassas. Den systemutvecklingsmetod som används är prototyping då vi hela tiden arbetat i nära samvaro med de framtida användarna.

Hypotesprövning

I modern vetenskapsfilosofi är en hypotes ett påstående som antas utan att vara verifierat. Ett påstående upphör att vara en hypotes i samma stund som det blir bekräftat, dvs. verifierat, eller förkastat, dvs. falsifierat, av erfarenheten.

Metoden har varit kritiserad inom vetenskapsfilosofin då det påpekats att det endast sällan finns stränga logiska relationer mellan vetenskapliga teorier och observationsutsagor. Vidare är det omstritt, huruvida användandet av metoden rent faktiskt kännetecknar vetenskapliga undersökningar.

Vår ambition är att illustrera nyttan av att formulera riktlinjer för gränssnittsdesign vid speciella förutsättningar, undersöka vilka faktorer som påverkar användbarheten, och inte med exakthet bestämma hur användbarheten optimeras.

Vi har valt att använda oss av en Hypotetisk-deduktiv metod[5] vilken i vårt fall innebär att:

1. Göra antaganden
   - formulera en hypotetisk lösning på ett problem

2. Diskutera och härleda den kvalitativa effekt som antagandena får
   - formulera ett antal härledda förutsägelser som ökar användbarheten för ett GUI

3. Pröva lösningen och diskutera dess riktighet
   - *utveckla en GUI-protyp och kontrollera dess kvalitet genom användbarhetstest*

---

[5] Arne Grøn m.fl. Redaktör: Paul Lübcke, *Filosofilexikonet*, Bokförlaget Forum AB, 1988.

Intervjuer

Styrkan i den kvalitativa intervjuformen ligger i att undersökningssituationen liknar en vardaglig situation och ett samtal. Det innebär att detta är den intervjuform där forskaren utövar den minsta styrningen vad gäller undersökningspersonerna. Det finns tvärtom en strävan mot att låta dem få påverka samtalets utveckling. Man måste samtidigt försäkra sig om att få svar på de frågor man vill belysa. Man kan se det som att man ”vaskar fram” den information man kan få om de frågor man är intresserade av.[6]

Vi har genomfört ett antal kvalitativa intervjuer med utvalda användare som en del av analysarbetet. Urvalet av undersökningspersoner är en avgörande del av undersökningen. Väljer man fel personer i urvalet kan det leda till att hela undersökningen blir värdelös i relation till den utgångspunkt man har när man börjar. Vi har i valet av intervjuobjekt tagit hjälp av vår handledare på Ericsson. Han besitter kunskap om vilka användare som är intressanta, vilka som representerar skilda delar av organisationen och som kan tänkas ha olika syn på vad applikationen bör kunna utföra.

Vi har tagit fram en intervjumanual snarare än ett utförligt utformat frågeformulär. Manualen kan sägas fungera som en minneslista som bildar utgångspunkt för intervjun. Det innebär inte att vi ska vara slaviskt bundna till manualen - intervjupersonen ska i största möjliga mån själv få utforma sina tankar och åsikter på ett naturligt sätt. Intervjuerna skall helst spelas in på band.[7] Det visade sig dock att de flesta intervjupersoner helst ville slippa inspelningsmomentet - vi valde därför att genomföra intervjuerna på så sätt att en person förde själva samtalet och två personer antecknade allt som yttrades.

Prototyping

Det är ofta svårt för slutanvändarna att i förväg veta exakt hur deras nya mjukvarusystem kommer att påverka deras dagliga situation. Om systemen är stora och komplexa är det förmodligen omöjligt att bedöma detta innan systemet är färdigt och sjösatt. Det är viktigt att användarna får vara delaktiga i att utforma systemet. Ett sätt att tackla detta problem är att använda sig av prototyper.

Prototyping har en speciell innebörd i systemutvecklingssammanhang, och det finns ingen riktigt bra svensk översättning av ordet. Prototyping innebär byggandet av systemprototyper, där varje prototyp utformas efter de senast erhållna önskemålen från användarna. Denna prototyp testas sedan för att få feedback om dess prestanda och möjligheter till förbättringar. Användarnas synpunkter tillsammans med den nuvarande prototypen bildar underlag för nästa prototyp.

Prototyping fokuserar på osäkerhet i samband med kravspecifikation och föreslår en experimentell strategi för problemlösning. Konceptuellt sett kan prototyping ses som en serie av mycket snabba upprepningar av de aktiviteter som återfinns inom det traditionella sättet för systemutveckling - analys, design och implementering. Varje cykel i denna process resulterar i en prototyp som testas. Fördelen med detta sätt att arbete är bland annat att man hela tiden rör sig mot målet - det är lättare att undvika att ”tappa bort sig”. Man vet hela tiden vad man ändrat på och varför och man får en direkt inblick i om ändringen varit till det bättre eller till

---

[6] Idar Magne Holme, Bernt Krohn Sovang, *Forskningsmetodik*, Studentlitteratur, Lund, 1991
[7] Holme, Solvang, 1991

det sämre. Det finns två allmänt vedertagna metoder för prototyputveckling, evolutionär och throw-away ptototyping.

Den evolutionära prototypingen utgår från att man implementerar ett relativt enkelt system som tillgodoser användarnas viktigaste krav. Systemet byggs ut och ändras i takt med att fler och fler önskemål upptäcks. Om allt fortlöper på ett optimalt sätt så får användarna till slut det önskade systemet. En alternativ metod är att utveckla en s.k. throw-away-prototyp för att synliggöra kraven. Denna prototyp kastas bort då man fångat vad användarna vill ha - man börjar om från grunden. Dock leder denna prototypsteknik till att man kan ta fram en systemspecifikation utifrån vilken man sedan bygger systemet.[8] Bild 2.1 ger en översiktlig bild av de två olika teknikerna.

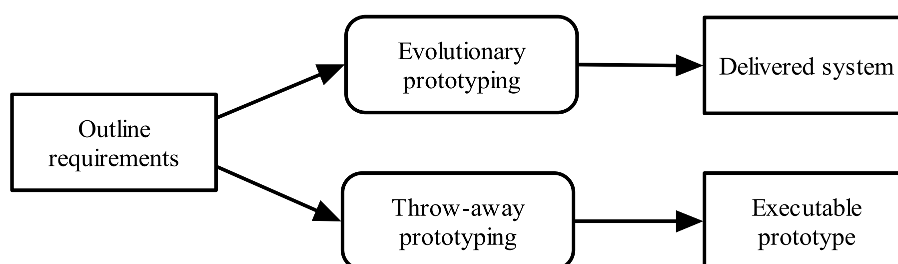


Bild 2.1
En översiktlig bild över de två ledande prototypingtekninkerna.

Vid utveckling av vår applikation har vi jobbat utifrån en evolutionär prototypingmodell. Detta eftersom användarna till en början inte kunde definiera alla krav och önskemål - dessa blev synliga efterhand. Det har varit en dynamisk process där nya önskemål kommit fram efter att applikationen visats upp. Även när det gäller gränssnittets utseende visade det sig mer passande att tillämpa evolutionär prototyping - användarna fick tycka till om olika presenterade förslag och till slut kunde man enas om ett lämpligt utseende. Med denna teknik har man ej tillgång till någon detaljerad systemspecifikation, och i många fall finns det ej någon formell kravspecifikation.[9] Denna teknik användes till en början för de system (ex. AI-system) som var svåra eller omöjliga att specificera. Nu har den evolutionära metoden växt till att bli den gängse. Bild 2.2 ger en översiktlig bild av evolutionär prototyping.

[8] Ian Sommerville, *Software Engineering 6th Edition*, Pearson Education Ltd, 2001.
[9] Sommerville, 2001

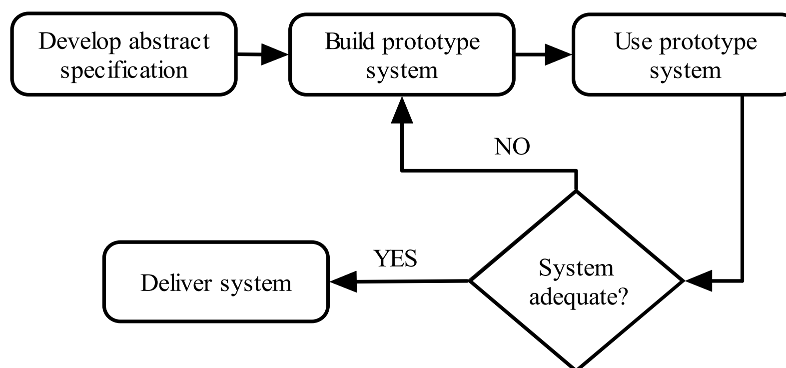


Bild 2.2
En översiktlig bild over evolutionär prototyping

Det finns ett par stora fördelar med att använda sig av evolutionär prototyping:

- *Snabbare leverans av systemet.* I vissa fall är snabb leverans och sjösättning viktigare kvalitetskriterier än detaljer i funktionalitet eller långsiktig hållbarhet.
- *Engagerade användare.* Att blanda in användarna i utvecklingsprocessen innebär inte bara att systemet lättare uppfyller deras önskemål. De känner sig dessutom delaktiga och tack vare detta är det mer troligt att de vill få systemet att fungera.

Användbarhetstester

Användbarhetstestning är ett utmärkt sätt att observera användaren när han eller hon testar en applikation för att på så sätt ta reda på om den är lätt eller svår att använda[10]. Då man började använda sig av dessa tester visade det sig att man fann fler fel i applikationerna än vad som kunde åtgärdas - detta om något visar att testerna är bra. Testningen ger bäst resultat om den genomförs tidigt och ofta snarare än i slutfasen då det är för sent för att genomföra ändringar. Vi kommer att redovisa resultatet användbarhetstesterna i Del 6 - Hypotesprövning. Vi använder användbarhetstestet som ett komplement till våra användarintervjuer för att synliggöra eventuella fel och för att ge ERV en typ av karta över vad man kan ändra och komplettera i framtiden. Alla användbarhetstester har fem ingående egenskaper:

- Det primära målet är att förbättra produktens användbarhet.
- Deltagarna representerar verkliga användare som innehar en lagom kunskapsnivå.
- Testet skall innehålla verklighetstrogna uppgifter. Man bör välja uppgifter som med hög sannolikhet leder till att användbarhetsproblem upptäcks.
- Observation och insamling
- Analyserande av insamlad data, diagnostisera problemen och rekommendera ändringar

[10] J S. Dumas, J C Redish, *A practical guide to usability testing*, Ablex publishing corporation, Norwood, New Jersey, 1994

Resultatet används för att förändra produkten och processen. Ett användbarhetstest är framgångsrikt endast om det hjälper till att förbättra produkten som testades och processen under vilken produkten utvecklas.

I ett användbarhetstest mäts följande:

- Vad och hur deltagarna gör när de använder produkten. Dessa mått är kvantitativa - man kan t.ex. mäta hur lång tid något tar att utföra, hur många fel som görs, hur många gånger samma fel upprepas m.m. Dessa mått kräver noggrann observation.
- Deltagarnas perception, åsikter och omdömen. Dessa kan vara antingen kvantitativa eller kvalitativa. Man kan exempelvis be deltagaren att sätta ett betyg på en 7-gradig skala på hur lätt eller svår en produkt är att använda. På så sätt får man en kvantitativ respons. Man kan också registrera deltagarnas spontana kommentarer genom att be dem tänka högt när de arbetar med produkten. Dessa kommentarer är både subjektiva och kvalitativa och det går även att se hur många som kommenterade samma problem.

Vi har valt att endast utföra ett användbarhetstest - detta som ett komplement till våra kvalitativa intervjuer. Vi har valt den deltagaren som vi tror i störst utsträckning kommer att använda applikationen i framtiden. Vi har också valt att fokusera mer på deltagarens subjektiva åsikter och omdömen än de rent kvantitativa aspekterna. De kvantitativa moment som finns med i vår undersökning finns i användbarhetstestet. Vi har inte följt användbarhetstestets regler dogmatiskt utan snarare modifierat det lite för att använda det som ett komplement.

Planeringen är viktig vid design av ett lyckat användbarhetstest. Tiden man lägger ner på testet kan vara helt bortkastad om man inte tänker på några viktiga saker:

- Vilka aspekter på produkten är kanske inte så användbara som de borde vara?
- Hur väl representerar deltagarna de verkliga användarna av produkten?
- Vilka uppgifter skall deltagarna utföra under den korta tid de har till förfogande?
- Vilken information skall samlas in när deltagarna observeras?
- Hur skall den insamlade informationen analyseras?
- Vad skall man göra med informationen när den har analyserats?

Del 3 Teoretiskt Ramverk

Uppsatsens teoretiska bas utgörs främst av avsnitt om GUI-design respektive testning men även aspekter på kvalitet och ISO-modellen behandlas.

Den litteratur som väglett och inspirerat oss i gränssnittssammanhang är huvudsakligen boken *About face the essential of user interface design* av Alan Cooper. Det finns en enorm flora av litteratur som vill berätta hur en god gränssnittsdesign bör se ut. Mycket av denna litteratur är mer eller mindre likformig. Anledningen till att vi valt att grunda vår design på Coopers idéer är att de är lite annorlunda. Han ställer hela tiden saker på sin spets, och ifrågasätter den traditionella synen på vad som kan kallas god gränssnittsdesign. Exempel: Om en applikation inte innefattar någon filhantering behövs ingen ”file-meny”. Det har kommit att bli en närmast oreflekterad standard att i alla gränssnitt döpa menyvalet längst till vänster till ”file” - vare sig det är befogat eller inte. Cooper menar bland annat att mjukvaruutveckling bör vara mer målorienterad och fokusera på att programmet ska utföra sin uppgift. I gränssnittssammanhang blir denna uppgift liktydig med att spegla vad han kallar användarnas mentala modell[11]. Vid sidan av Cooper har vi även använt oss av Lars Mathiassens et al *Objektorienterad analys och design*[12] och Ian Sommervilles *Software Engineering*[13].

Teorin om testning redovisar övergripande en av de vanligaste testningsmodellerna och de begrepp som är relaterade till testningsprocessen. Fokus ligger på den testning som utförs av programutvecklarna själva., så kallad unittestning. Unittestning tillsammans med debugging utgör den första och grundläggande fasen i testningsprocessen och är den typ av testning som är mest relevant för fallstudien. Den litteratur som vi stöder oss på består av två verk som båda ger en samlad och övergripande beskrivning om testning, dess teorier, metoder och syften: *Software Testing and Continuous Quality Improvement*[14] av William E. Lewis samt *Analysis and Testing of Distributed Software Applications*[15] av Henryk Krawczyk & Bogdan Wiszniewski.

[11] Cooper, 1995 (sid. 29)
[12] Mathiassen, Lars, et al, *Objektorienterad analys och design*, Studentlitteratur, 1998

[14] William E. Lewis, *Software Testing and Continuous Quality Improvement,* CRC Press LLC, 2000
[15] Henryk Krawczyk & Bogdan Wiszniewski, *Analysis and Testing of Distributed Software Applications,* Research Studies Press Ltd, 1998.

3.1 Grafiskt användargränssnitt

I denna del presenterar vi de teoretiska grundfundament för de verktyg vi använt oss av för att komma fram till hur ett GUI bör se ut. Verktygen kan delas upp i två varianter: taktiska och strategiska. De taktiska utgörs av praktiska tips och anvisningar för hur man använder och skapar gränssnittsidiom, som dialogboxar och knappar. Strategiska verktyg lägger grunden för ett sätt att tänka om gränssnittsidiom – med andra ord, det sätt som användaren och idiomen interagerar. Nyckeln till att skapa ett framgångsrikt GUI ligger i sammanflätningen av dessa verktyg. Det finns till exempel inte något sådant som en objektivt sett bra designad dialogruta – kvalitén beror helt på situationen: vem användaren är, vilken bakgrund och vilka mål hon har. Vi börjar med en genomgång av de strategiska verktygen vilken vi lägger störts vikt vid att beskriva för att sedan successivt presentera de taktiska.

Målinriktad design

Alan Cooper menar i boken *About face the essential of user interface design* att stor del av den programvara som har producerats och idag produceras inte är designad. Även om det från början har funnits en grundläggande och medveten tanke vilken dokumenterats och specificerat så omkullkastas mycket av detta arbete vid implementeringen av ett program. Program tillkommer ofta genom att successivt träda fram ur ett mjukvaruteams samlade ansträngningar. Under denna fas görs mycket av designarbetet om för att stämma överens med de tekniska, tidsmässiga och kompetensbaserade kriterier som utvecklingsarbetet lyder under. Om ett projekt inte har ett tydlig mål får detta till följd att program vanligtvis designas ur ett programmerarperspektiv, ibland ur marknadsföringsavdelningens perspektiv och då och då ur ett användarperspektiv. Inget av dessa tre perspektiv reflekterar dock vad Cooper kallar *användarens mål*[16]. Programmerare tenderar att favorisera teknologiska och programmeringsmetodologiska imperativ. Marknadsföringsavdelningen är fokuserade kring vad som väcker mest uppmärksamhet på markanden och användarna är så upptagna av sina vardagliga uppgifter att de kan ha svårt att formulera och vara medvetna om vilka deras mål är. Det ligger på dem som designar programmet att utvinna och formulera programmets målsättning, dess syfte, ur alla intressenters och användares enskilda behov och uttryck.

Användarens mål

Det finns ingen universallösning för hur ett bra GUI ska se ut, gränssnittets kvalité är helt beroende av dess kontext. Hur ska programmet användas? Vem ska använda det? Hur ofta? Under hur lång tid i taget? Hur viktig är dataintegriteten? Inlärningsfrekvensen? Portabiliteten? Svaren på dessa frågor skiftar från applikation till applikation och det första en mjukvarudesigner måste göra är att försöka besvara dessa och andra användarcentrerade frågor. Att bestämma vilka så kallade ”features” som ska ingå i programmet och hur grafiskt avancerat ett gränssnitt skall vara bör inte bestämmas av vad som teknologiskt går att genomföra. Istället bör resultatet spegla de mål som användarna har. Det är meningslöst och i många fall irriterande att skapa något som ändå inte kommer att användas eller som genom sin grafiska prakt enbart belastar minnet om användarens mål inte överensstämmer med detta. Cooper åskådliggör detta genom att skilja mellan feature- och målcentrerad design:

[16] Cooper, 1995, (sid. 11)

- **Featurecentrerad design** – Programmeringstypen som tänker i termer av funktioner och features. Ett fullständigt naturligt tillvägagångssätt då det är så program byggs - funktion efter funktion. Problemet är att användare inte använder ett program på det sättet, vilket skapar en destruktiv diskrepans mellan hur programmet är tänkt att användas och hur det sedan används.
- **Målcentrerad design** – Att hela tiden fokusera designarbetet på vilken uppgift som programmet ska lösa och hur detta uppnås på bästa sätt. Fokuseringen måste utgå från de framtida användarnas verklighetsbild och omgivning.

Ett bra program gör användarna mer effektiva och det är upp till mjukvarudesignen att formulera hur denna effektivitet manifesteras under de omständigheter som råder i ett speciellt fall.

Mjukvarudesign

Det föreligger ofta en intressekonflikt vid utvecklingen av mjukvara då de som ska implementera programmet ofta även designar det. Detta får till resultat att kod som en gång skrivits tenderar att bli kvar även om koden tillkommit under prototypingen och därmed borde kastats. De mjukvaruverktyg som ska stödja designprocessen är dessutom ofta av universaltyp och fungerar även som programmeringsverktyg. Designprocessen borde enligt Cooper i så hög utsträckning som möjligt skiljas från implementeringsprocessen. Mötet mellan design och implementering sker vid designverifikationen – prototypingen – vilken är ett sätt att testa olika tekniker och designlösningar. När designen är fastlagd bör den kod som uppkommit under prototypingen kastas för att sedan skrivas om i ett bättre och mer konsistent skick under implementeringen. Man kan även tänka sig att, som i vårt fall, design- och implementeringsfasen på ett naturligt sätt växer samman. Men detta kräver en naturlig och ständig närhet till användarna. Om designen förändras under implementeringsfasen innebär denna förändring en förbättrad formulering av användarnas mål och inte en konsekvens av programmeringsteamets lättja eller ointresse. Cooper återger följande fundamentala definition av mjukvarudesign[17]:

Definition av mjukvarudesign:

1. Vad ska programmet göra?
2. Hur ska det se ut?
3. Hur ska det kommunicera med användarna?

Användargränssnittsdesign är en delkomponent vid mjukvaruutvecklingen och innefattar främst punkt 2 och 3, men det är dock ofta svårt att helt separera dessa två punkter från punkt 1.

Tre modeller

En maskin använder sig alltid av en metod för att genomföra en uppgift. Den specifika metod som beskriver hur maskinen fungerar kallar Cooper för maskinens implementeringsmodell. Cooper skiljer denna metod från hur en användare uppfattar det som maskinen åstadkommer - användarens mentala modell, eller dennes konceptuella modell. En grundläggande

[17] Cooper, 1995 (sid. 24)

målsättning vid mjukvaruutveckling är enligt Cooper att i så hög grad som möjligt överföra användarens mentala modell i det utvecklade programmet. Hur väl denna överföring stämmer överens med verkligheten kan beskrivas genom programmets manifestmodell[18]. Det är genom användargränssnittet som användaren kommunicerar med programmet och manifestmodellen beskriver hur intuitiv och välmatchad denna kommunikation blivit.

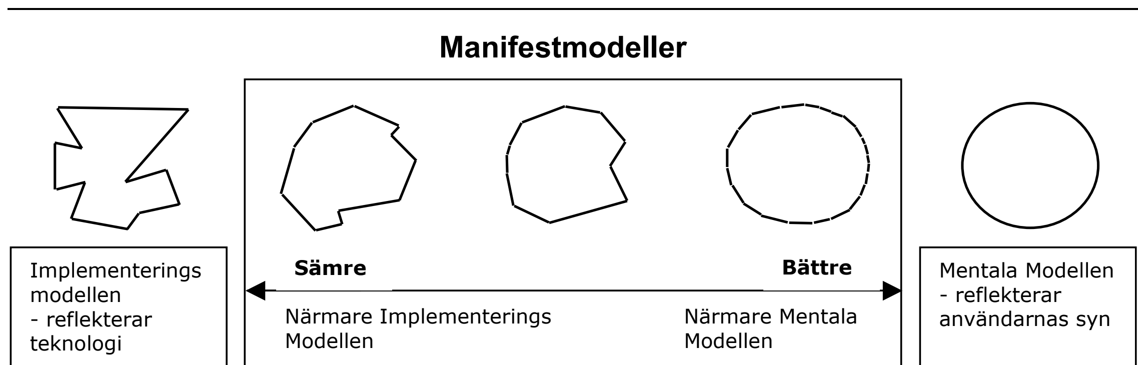


Bild 3-1
Ett programs manifestmodell reflekterar designens tolkning av användarnas verklighet. Ett bra användarinterface ska enligt Alan Cooper vara så likt användarnas mentala modell som möjligt.

Vid systemutveckling är det viktigt att kommunicera med de framtida användarna under alla faser av utvecklingscykeln. Användarna bryr sig oftast inte om hur programmet är implementerat och egentligen fungerar. Det enda som spelar roll är att programmet på enklast och mest intuitiva sätt utför sin uppgift. Ett grafiskt användarinterface som ligger nära användarens mentala modell blir enklare att lära sig och att använda då denna utnyttjar bilder och uttryck som är familjära.

Visuellt användarinterface

Vanligtvis anses ett grafiskt användarinterface vara att föredra framför ett teckenbaserat. Men ett dåligt designat GUI som överbelastar datorn eller inte nämnvärt effektiviserar programmets uppgifter kan upplevas som irriterande och onödigt och därmed försvåra arbetet istället för att förenkla det. De kvalitetskriterier som bör uppfyllas för att ett GUI ska bli uppskattat och framgångsrikt är av användarcentrerad natur inte teknologicentrerad. De två viktigaste kriterierna enligt Cooper är mjukvarans *visualitet*[19] och programmets *vokabulär*[20].
De flesta människor bearbetar information bättre visuellt än via text. Visst lär vi oss mycket genom att läsa men vi lär oss mycket mer och snabbare genom att se hur saker fungerar i verkligheten och i sitt samanhang. Det är inte grafiken i sig som möjliggör ett förenklat kommunicerande, grafik är en teknologisk term utan egentligt innehåll. Det är istället visualiteten av interaktionen, ett visuellt användarinterface – ett VUI – som är det väsentliga. Ett väldesignat VUI förmedlar en känsla av ledighet och frihet, och möjliggör för användaren att utföra sina uppgifter i vilken ordning han själv finner bäst och utan att hindra och förvirra detta arbete. Interaktionen mellan användare och program ska ske så friktionsfritt som möjligt och vägen mot målet ska gå fort och utan distraktioner.

[18] Cooper, 1995 (sid. 29)
[19] Cooper, 1995, (sid. 42)
[20] Cooper, 1995, (sid. 47)

Ett effektivt visuellt användargränssnitt borde byggas utifrån vissa användarcentrerade visuella mönster. Dessa mönster, vilka klarläggs och utvecklas i samverkan med användarna, ska representera och efterlikna användarnas undermedvetna uttryck och verklighet. Genom att bygga programmets VUI med stöd av vedertagna bilder och mönster blir programmet enklare all lära sig och att använda. Att läsa innebär att hjärnan medvetet måste arbeta för att förstå, en bild eller ett mönster kan däremot förmedla en innerbörd omedvetet och mycket snabbt. En lista med olika objekt blir exempelvis enklare att lösa om objekt av samma typ på något vis är kopplade till varandra genom bilder eller färger. Bilder och symboler är dessutom lätta att lära sig så länge de inte är allt för många och för ologiska..

Ett programs vokabulär kännetecknas av den kunskap och de element som behövs för att någon ska kunna kommunicera med programmet. I ett GUI kan en användare peka på bilder eller ord på skärmen med musen. Genom att använda musknapparna kan användaren exempelvis dubbelklicka eller klicka-och-dra på något. Ett GUI är givetvis mycket enklare att lära sig än ett kommandobaserat interface därför att de element brukaren behöver lära sig för att använda och förstå ett program är färre. Men å andra sidan kan det vara svårt att visualisera något som är språkligt komplicerat och differentierat. Om man som expertanvändare behärskar språket kan man genom ett kommandobaserat interface uttrycka sig snabbare och effektivare. Men oavsett vilket typ av interface man kommunicerar med hjälp av så ska programmets vokabulär fungera så intuitivt och följsamt som möjligt. En bra designad vokabulär har formen av en inverterad pyramid. Alla kommunikationssystem som är enkla att lära sig följer detta mönster som Cooper kallar *the canonical vocabulary*[21].

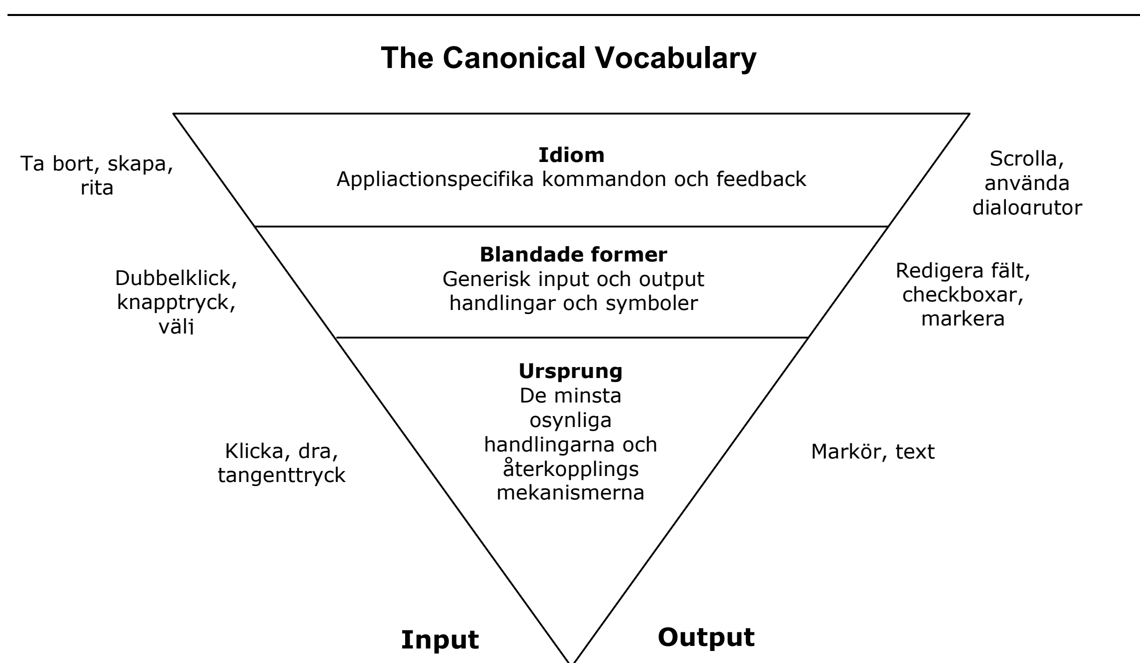


Bild 3-2
Huvudanledningen till att GUI:s är enklare än andra interface att använda är dess vokabulär är uppbyggt på detta inverterade pyramidlika sätt. Den lägsta nivån består av ett antal ursprungskomponenter som kontrollerar och skapar allt annat. Generellt borde antalet komponenter inte överskrida fyra. Mittennivån består av mer komplexa konstruktioner vilka utgörs av olika kombinationer av ursprungskomponenter. Den högsta nivån består av den kunskap som tillförs under en speciell situation och i ett speciellt program.

[21] Cooper, 1995, (sid. 47)

Bottensegmentet i pyramiden består av de ursprungskomponenter språket är uppbyggt kring och kontrolleras genom. Dessa komponenter ska vara så små och få som möjligt. I ett GUI består dessa av pekning, klickning, dragning och tangenttryck. Mittensegmentet består av kombinationer av ursprungskomponenter vilket i ett GUI exempelvis gestaltas genom dubbelklick, klick-och-drag och checkbox val. Det översta segmentet utgörs av mer programspecifika handlingsmönster som i ett GUI exempelvis utgörs av kända ikoner som ”spara-ikonen”, OK-knappen, listboxar etc.
För att skapa ett effektivt användarinterface måste designen skapa en interaktion mellan program och användare som utgår från *the canonical vocabulary* och uttrycka denna visuellt. Vokabulären ska följa användarens mentala modell, även om denna modell skiljer sig från den ”fysiskt” korrekta.

Form

Ett programs form gestaltas av den helhetslösning som renodlas och fastslås under utvecklingsarbetets designfas. Formen återger, beroende på vilka de tänkta användarna är, någons verklighetsbild. En lyckad design återger denna bild korrekt och har som färdig produkt stor chans att upplevas som enkel och effektiv att använda. Att återge något i enlighet med användarnas mentala bilder är inte liktydigt med att finna intuitiva mönster som man av någon orsak upplever som nedärvda. Vår omgivning är full av bilder som vi i ett kort tidsperspektiv och i ett visst kulturellt sammanhang upplever som intuitivt associerade med vissa betydelser. Att använda sig av sådana bilder och förlita sig på att deras intuitiva innebörd garanterar att ett GUI blir enkelt att förstå kan få negativa konsekvenser. Detta då den intuitiva innebörden förändras och fördunklas över tid och att individer på grund av kulturella och sociala orsaker inte upplever saker på samma sätt. Istället för att låsa upp formuttrycken kring metaforer är det bättre att fokusera på att det GUI man skapar är enkelt att lära sig. De symboler och former som är tänkta att återge en känsla av vilken funktion de utför bör designas med intentionen att de är enkla att lära sig. Detta utesluter inte att vissa intuitiva sammanhang används, absolut inte. Det är bara en inställningsmässig skillnad. Generellt sett brukar det löna sig att använda sig av formmässiga standarder, så också i gränssnittssammanhang. Det finns olika typer av fönsterhanteringssystem vilka beroende på vilken uppgift programmet har, fungerar som standardlösningar. Och många av de delkomponenter som ett gränssnitt består av, som menyer och verktygsfält, har med tiden upphöjts till standard. Allt detta ska man använda sig av, men samtidigt är det viktigt att tänka på att kombinera rätt komponenter till en fungerande helhet.

Tre gränssnittsparadigm

Det finns tre paradigm som har dominerat gränssnittsdesignen genom åren. Dessa är teknologiparadigmet, metaforparadigmet och det idiomatiska paradigmet[22].

Teknologiparadigmet baseras på förståelse för hur saker och ting fungerar vilket innebär att ett användarinterface speglar implementeringsmodellen. Genom teknologiparadigmet förmedlas teknikernas syn på hur programmet är konstruerat. Detta är en bild som ofta inte är speciellt generell och intuitiv utan bara påvisar en specifik grupp individers tankar kring sin egen konstruktion. Oavsett hur tekniskt kunniga användarna är så borde ett programs främsta uppgift vara att lösa deras problem, inte att förevisa hur programmet är konstruerat.

[22] Cooper, 1995, (sid. 54)

Metaforparadigmet baseras på en intuition om hur saker och ting fungerar, vilket innebär att alla som använder programmet förväntas ha en likvärdig verklighetsuppfattning. De bilder och mönster som ett GUI är uppbyggt kring är tänkta att representera vissa intuitiva företeelser och därmed förmedla en självklar betydelse. Problemet med metaforer är att de associationer de ger upphov till kan skilja sig från individ till individ. Metaforer är kulturbetingade vilket gör att lätt kan missförstås och om symboliken som förmedlas misstolkas försvåras givetvis inlärningsfrekvensen.

Det idiomatiska paradigmet baseras på lärande, om hur man åstadkommer något. De flesta elementen i ett GUI består av idiom. Fönster, drop-down-menyer och musklick är något som vi lär oss att använda idiomatiskt och inte intuitivt. Alla idiom måste läras in och ju bättre de är designade desto enklare är de att lära sig.

Många av de GUI-element som idag upplevs som metaforiska är i själva verket idiomatiska, de är väldesignade och lättlärda och har därmed upphöjts till någon slags standard. Att arbeta utifrån ett metaforiskt perspektiv kan verka naturligt vid avbildning av fysiska objekt som dokument och printrar men blir svårt eller till och med omöjligt vid avbildning av processer, relationer, tjänster och transformationer – vilka alla är mycket vanliga företeelser i mjukvara. Ett annat problem med metaforer är att de tenderar att åldras, att exempelvis hålla fast vid att symbolen *diskett* betyder *spara* i ett samhälle där disketter inte längre existerar kan över tid verka förvirrande. För att ett GUI ska bli enkelt att använda bör designen inte baseras på någon slags godtycklig metaforisk standard. Och om man använder sig av metaforer ska dessa vara väl förankrade och tydligt återknutna till programmets idéer och meningar. Dessutom bör en bilds symbolik även finnas tydligt beskriven inom programmet på något vis.

Fönsterhantering

Ett program är ofta konstruerat av två typer av fönster; huvudfönster och underordnade fönster. Varje fönster bör ha en tydlig mening och funktioner bör placeras i det fönster där de används. Program lider ofta av ”fönsternedsmutsning” vilket innebär att ett övermått av dialogrutor och underordnade fönster används utan att detta är nödvändigt. Designen bör därför utmynna i att minimera antalet fönster och placera all funktionalitet i det fönster där uppgiften ska lösas.

Dokumenthantering

Hur man går till väga för att spara något i ett GUI har blivit en etablerad standard. Genom att gå till ”File”-menyn och där välja ”Save” sparar man sitt dokument. Detta är ett tydligt exempel på hur vokabulären i ett program följer implementeringsmodellen. En fil och ett filsystem är implementeringstermer som beskriver hur programmet fungerar utifrån en programmerares perspektiv. Om man istället försöker anpassa vokabulären i ett program till användarens mentala modell borde namnet på ”File”-menyn istället beskriva vilken typ av dokument som man för tillfället arbetar med. Att döpa om ”File”-menyn till exempelvis ”Document”-menyn om det är dokument man arbetar med kränker en väl inarbetad standard och kan vid en första anblick förvirra vana användare. Men för nya användare, och även för vana användare, borde en sådan åtgärd uppfattas som mer anpassad till användarens egen bild av vad programmet ska åstadkomma. Om man arbetar med exempelvis en bild är det denna bild som ska sparas, visst är bilden också en fil men detta är främst en implementeringsdetalj. För användaren är objektet primärt en bild och i enlighet med den mentala modellen bör detta avspeglas i det grafiska användargränssnittet.

Filhanteringen är den del av ett GUI som mest av allt är standardiserat. De flesta program behandlar någon typ av filer som går att öppna, stänga och spara. Vid sidan av dessa generella funktioner finns det ett antal mindre målorienterade funktioner som användaren skulle kunna vilja ha. Dessa är exempelvis[23]:

- Skapa en kopia av dokumentet.
- Namnge och döpa om dokumentet
- Placera och förvara dokumentet
- Specificera dokumentets lagringsformat
- Reversera vissa förändringar
- Överge alla förändringar

En generell riktlinje vid utveckling av GUI är enligt Cooper att ett program bör utföra sin uppgift tyst, effektivt och känsligt, utan att störa användaren med onödiga frågor. Genom en bra design ska så mycket som möjligt av programmets funktonalitet utföras utan att användaren får frågor av typen om de verkligen vill utföra det de precis har beordrat.

Ett program kan ha mer eller mindre raffinerade system för lagring och återhämtning av filer. Vid sidan av de generella sökfunktionerna som erbjuds genom operativsystemet kan ett program utvecklas så att det exempelvis känner till vilka filer programmet använt senast. Cooper gör skillnad mellan tre fundamentala sätt på vilket något kan återfinnas. Dessa sätt är:

- **Positionsåterfinnande** – Man hittar något genom att komma ihåg var man lagt det.
- **Identitetsåterfinnande** – Man hittar något genom att komma ihåg objektets identifierande namn.
- **Associativt återfinnande** – Baseras på att ett objekt går att söka upp genom vetskap om någon inneboende kvalitet eller egenskap hos objektet.

Positions- och identitetsåterfinnande brukar vanligtvis implementeras genom att ett antal av de filer som senast använts listas med både sökväg och identifierande namn. Användaren väljer vilken fil som ska hämtas in i programmet och får samtidigt vetskap om var i filsystemet filen är placerad. Vid positions- och identitetsåterfinnande är det upp till programmet att registrera var relevanta filer är placerade. Filen behöver bara registreras med ett namn. Vid associativt återfinnande måste filen däremot vidhäftas med mer information som till exempel:

- Vilket program som skapade filen.
- Vilken typ av fil handlade det om: ord, nummer, tabeller, grafik etc.
- Vilket program som senast öppnade filen.
- Om filen är exceptionellt stor eller liten.
- Om filen inte har öppnats under lång tid.
- Hur lång tidsperiod som en fil senast har varit öppen.
- Storleken på den information som lades till eller togs bort i senaste versionen.
- Om filen har blivit redigerad i fler än en typ av program.
- Om filen innehåller inbäddade objekt från andra program.
- Om dokumentet redigeras ofta.

[23] Cooper, 1995 (sid. 107)

Alla dessa faktorer är relativt enkla för ett program att registrera om en fil. Ju fler filegenskaper ett program registrerar desto fler möjligheter finns det att associativt återfinna filen. Dialogfönstret genom vilket man hämtar in en fil till ett program ser ofta likadan ut oberoende av program. Detta beror till stor del på att det finns standardklasser som erbjuder färdigdesignade öppna- och spara-fönster och att programmerare väljer att inte anpassa dessa efter användarnas mentala modell, utan bara använder den färdigskrivna koden rakt av. Standardklasser är givetvis värdefulla men de reflekterar ofta någon slags implementeringsmodell och behöver, om man vill återge ett specifikt dokuments karaktär, anpassas från fall till fall. Bild 3-3 visar ett bra exempel på hur man kan bygga vidare på en standardklass och tillföra dokumentspecifika karaktäristika.

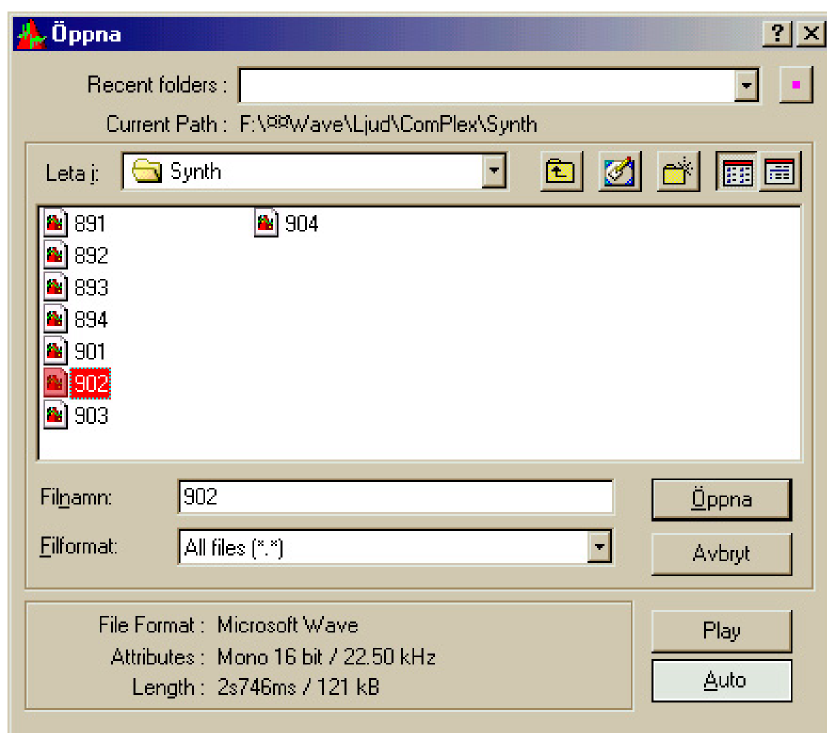


Bild 3-3
Bilden visar hur programmet Wavelab:s öppna-fönster ser ut. Vid sidan av standardiserad funktionalitet som bläddring i filträd, byta namn på fil, ta bort en fil och val av filformat så har utvecklarna byggt på programspecifik funktionalitet. Överst i dialogrutan kan man att öppna en fil ur de kataloger som man senast använt i programmet och nederst får man information om filen och om en ljudfil markeras möjlighet att lyssna till denna.

Beteende

För att ett program ska bli mer produktivt, måste de som använder programmet också göras mer produktiva. Ett sätt att öka produktiviteten är att få användarna att uppleva att de är i harmoni med sitt arbete. Ett gränssnitt fyller här en grundläggande och viktig funktion då det är gränssnittet som till stor del dikterar användarens sätt att använda programmet. Ett GUI bör som vi tidigare nämnt reflektera användarnas mentala modell eftersom det ökar programmets användarbarhet. Men det är också viktigt att användarna litar på att programmet verkligen

utför det som det ska göra. Att programmet är enkelt att styra och förstå blir meningslöst om användarna inte övertygas om riktigheten i det som åstadkoms. Ett väl designat GUI låter användaren lösa sina uppgifter utan att blanda sig i eller förhindra detta arbete. Idealfallet vore ett osynligt GUI som inte stör men som hela tiden läser av användarens beteende och agerar exakt vid rätt tillfälle, ett GUI som motiverar och stödjer användarens arbetsflöde och som dessutom utstrålar och genererar trovärdighet.

Flöde

För att skapa ett flöde bör vår interaktion med mjukvaran bli transparent. Cooper nämner fyra punkter som ett lyckat "osynligt" GUI bör uppfylla. Dessa är[24]:

1. Följ mentala modeller
2. Dirigera istället för att diskutera
3. Ge hela tiden användare tillgång till relevanta verktyg
4. Ge feedback, men i neutral och saklig ton och utan att störa

Olika användare skapar sig olika mentala modeller över hur en process ska utföra sin uppgift och dessa bilder formuleras sällan som detaljerade beskrivningar över hur dataprocessen ser ut. Istället tydliggör de en individuell tolkning som uttrycks i en mental bild av dataprocessen. Ett GUI bör återspegla en användares mentala bild istället för den struktur som programmet är byggt kring. De flesta användare är målorienterade, de vill att programmet ska utföra en uppgift åt dem. Detta ska ske så enkelt och friktionsfritt som möjligt, den ideala interaktionen är inte en dialogruta utan snarare användandet av ett verktyg. Om användarens mentala modell är uppfylld finns relevanta verktyg som servar användarens aktuella behov hela tiden till hands. De verktyg som inte används frekvent bör även de vara lättåkommliga men inte i samma omfattning som de som akut behövs för att lösa uppgiften. Användaren förväntar sig förmodligen att programmet säger ifrån om något går fel, men inte genom en tidsödande tvåvägskommunikation. Utan snarare genom en exakt och korrekt dirigering som erbjuder en snabb lösning. Dirigeringen bör ske på ett sätt som Cooper benämner *modeless feedback*[25] vilket innebär att feedbacken byggs in i det normala interfacet och inte poppar upp som dialogrutor. Genom att minimera förekomsten av dialogrutor och fönster störs inte det normala flödet av systemaktiviteter och interaktion.

Det är vitalt att alla element i ett GUI arbetar tillsammans mot ett gemensamt mål. Programmets funktionalitet måste koordineras och styras utan att programmet upplevs som rörigt och svårt att förstå. Ju mindre grafisk utsmyckning och krånglighet som byggs in i ett GUI desto enklare och smidigare borde det fungera i interaktionen med användaren. För om det primära för användarna är att få jobbet gjort borde ett GUI distrahera så lite som möjligt och bara agera mellanhand. Att bygga in för mycket funktionalitet i ett program som ingen egentligen behöver är givetvis meningslöst, i ett kommandobaserat gränssnitt spelar detta dock inte så stor roll. Men när funktionaliteten ska gestaltas genom ett GUI kan effekten upplevas som förvirrande och direkt destruktiv. En alltför rik flora av detaljer och funktonalitet gör att användaren blir så frustrerad och förvirrad att syftet med programmet, det användaren vill uppnå, kraftigt försämras.

Överhuvudtaget bör alla designmässiga lösningar i ett GUI konstrueras så att användarens kreativa flöde inte förhindras. Detta kräver givetvis att programmet designas med en

[24] Cooper, 1995 (sid. 128)
[25] Cooper, 1995, (sid. 131)

medvetenhet om vad som kan gå fel och när. Men alla fel som uppstår och handlingar som användaren utför behöver nödvändigtvis inte visas upp i gränssnittet. Mängden och vilken typ av information beror helt på vilka mentala modeller man valt att bygga programmet kring. Den generella riktlinjen vid konstruktion av ett GUI är att användarens arbetsflöde ska uppmuntras och förenklas.

Stämning och tillstånd

Ett programs GUI kan se ut precis hur som helst bara det finns en målorienterad orsak till utseendet. De färger, uttryck och visuella former som programmet presenteras genom påverkar programmets användarbarhet, dessa måste därför ha en välgrundad anledning till att finnas till. Ett programs beteende bör reflektera sättet det används på, inte tvärt om, och därmed vara anpassat till rätt grupp av användare. Enligt Cooper finns det, beroende på vilken attityd som programmet ska förmedla, fyra applikationskategorier[26]: sovereign, transient, daemonic och parasitic. Dessa kategorier gestaltar olika typer av beteendemässiga attribut såväl som olika former av användarinteraktion. Om ett program har designats efter ett visst beteendemönster och sedan inte används i enlighet med detta kommer användarnas uppfattning av programmet att bli missvisande. Det finns givetvis program som uppvisar andra egenskaper eller blandade former av de egenskaper som de fyra applikationskategorierna ger uttryck för. Men dessa fyra är grundformer som får tjäna som exempel för en indelning av de uttryck som olika applikationer förmedlar.

Sovereign Posture

Ett program är sovereign om dess beteende upptar hela skärmen, och monopoliserar användarens uppmärksamhet under en längre period. Detta innebär att användaren förmodligen kräver att programmet ska prioritera kraft och snabbhet. Sorvereign-program är inte primärt designade för förstagångsanvändare utan vänder sig till någon form av erfarna användare. För även om de, som exempelvis Microsoft Word eller Adobe Photoshop, är relativt enkla att komma igång med, så finns det ett stort utrymme för användare att lära sig ännu mer om programmets funktonalitet. Detta öppnar upp stora möjligheter för en utvecklare då man kan ”slösa” med både skärmutrymme och funktonalitet. Om programmet till exempel kräver fyra verktygsfält för att manövreras så kan det konstrueras så. Ett övermått av detaljer är givetvis aldrig bra, men detta kan lösas genom att användaren själv får bestämma vilken funktonalitet som exempelvis ska kunna styras genom ett verktygsfält. De knappar som verktygsfältet består av bör ta lite plats och med hjälp av en illustrering med vidhäftande verktygstips enkelt påvisa sin funktionalitet.

Eftersom sorvereign-program är tänkta att arbetas med under en längre tidsperiod är det viktigt att den visuella presentationen är relativt diskret. Detta innebär att färgpalletter och former ska hållas tillbaka till förmån för ett mer konservativt och stramt uttryck. Programmet bör vidare ge användaren fortlöpande information om beteende-, tillstånd- och stämningsmässiga förändringar. Men inte på ett sätt som hindrar flödet, utan istället exempelvis som uppdelade informationsbitar i fönstrets nederkant.

Transient Posture

Om ett program manipulerar ett dokument men endast utför en relativt enkel funktion som till exempel att scanna in ett dokument är det av typen *transient*. Ett sådant program upptar endast skärmutrymme under tiden det utför sin uppgift. Ofta opererar det som ett

[26] Cooper, 1995, (sid. 153)

instickningsprogram i ett annat program och kallas fram vid behov för att sedan försvinna. Eftersom ett *transient-posture* program är designat för en speciell och begränsad uppgift, som användaren förmodligen vill utföra så snabbt som möjligt, bör programmet vara mycket lätt att använda. Eftersom användarna inte exponeras för programmet under längre tidsperioder bör allt vara enkelt att förstå och i det närmaste självinstruerande. Knappar får gärna vara stora och tydliga och alla instruktioner som kan behövas för att utföra uppgiften bör vara inbyggda i programmet.

Daemonic Posture

Program som normalt inte interagerar med användaren utan bara utför sina uppgifter kallar Cooper för *daemonic posture*-program. Dessa program arbetar tyst och osynligt i bakgrunden medan de utför sina uppgifter som exempelvis en skrivarrutin. Det viktiga ur ett användarperspektiv är att bli informerad om vad programmet utför, hur man kan ändra dess beteende, hur man byter ut det etc. Det också viktigt att *daemonic posture*-program ger ifrån sig någon form av signal när det inte lyckas utföra sin uppgift.

Parasitic Posture

Program som blandar *sovereign-* och *transient*-egenskaper kallar Cooper för *parasitic posture*-program. Vilket innebär att programmet upptar användarens uppmärksamhet under en längre tidsperiod med samtidigt interagerar med andra program och har därmed inte tillgång till hela skärmytan. Programmet samsas därmed en stödjande funktion i en större helhet och fungerar ofta som en rapportör av pågående processer vilka även på olika sätt möjligen kan manipuleras genom programmet. *Parasitic posture*-program ska inte form- och färgmässigt designas så att de drar uppmärksamheten från moderprogrammet. Om dess främsta uppgift är att stödja ett annat program men samtidig under längre tidsperioder finnas närvarande på skärmen, bör det inte utstråla för vidlyftiga uttryck. Man kan dock tänka sig att dessa program kan förses med någon form av uttryckstillval typ byte av ”skinn” i programmet Winamp, så att användaren kan förända programmets utseende efter tycke och smak. Eftersom programmet ska befinna sig på skärmen under en längre tidsperiod kan knappar och ikoner etc. designas under samma premisser som ett *sovereign posture*-program.

Ett programs tillstånd

Ett programs tillstånd kan betyda ett antal olika saker. Det kan innebära att fönstrets storlek är minimerat, maximerat eller anpassat. Tillståndet kan också utgöra en beskrivning av en viss punkt i ett programs beteende, om beteendet beskriver helheten så är tillståndet delarna. Programdesignen bör sträva efter att minimera tillstånd som inte direkt hjälper till att lösa användarnas problem. Genom att korta kedjan av tillstånd som tillsammans får något att hända kommer användarnas arbete bli mer effektivt. Ju mer komplext ett program är desto större är chanserna att det uppstår tillstånd som egentligen inte behövs. Designen bör sträva efter att skapa ett så målorienterat och därigenom avskalat program som möjligt. Exempel på tillstånd som innebär att användarens intentioner att utföra sin uppgift hindras är:

- Tvinga inte användaren att gå till ett annat fönster för att utföra en uppgift som påverkar det fönstret han arbetar i.
- Tvinga inte användaren att komma ihåg var han sparat saker i ett hierarkiskt filsystem.
- Tvinga inte användaren att förstora fönstret. Anpassa fönstrets storlek efter innehållet och gör det varken för stort eller för litet.

- Tvinga inte användaren att flytta fönster. Om det finns ledig plats på arbetsytan placera då programmet där och inte över andra program. Dialogrutor bör komma upp i mitten av det program man arbetar med.
- Tvinga inte användaren att återinställa sina personliga inställningar. Bevara och spara så många som möjligt av dessa, även fönsterstorlek.
- Be inte användaren att konfirmera sina handlingar, erbjud istället en ångra-funktion.
- Designa programmet så att användarens handlingar inte genererar felaktigheter.

Dessa exempel har alla det gemensamt att om de inte löses kommer de att försämra användarens arbetsflöde. Genom att eliminera de tillstånd i ett GUI som hindrar användaren från att uppfylla sitt mål blir programmet förmodligen bättre och mer uppskattat.
Felmeddelanden är annan källa till irritation som noga bör analyseras och verifieras. Genom att i samarbete med användarna, med den allmänna standarden i beaktning, utforma en vokabulär som är begriplig och meningsfull kan detta problem försvinna. Språkligt sett bör all information som programmet ger ifrån sig vara välformulerad och korrekt. Men användarna bör även besparas systemspecifika meddelanden som de inte har någon nytta av. Endast information som beskriver fel och tillstånd som har sitt ursprung i användarnas mentala modell bör existera. Väldigt få av dessa bör visas genom dialogrutor.

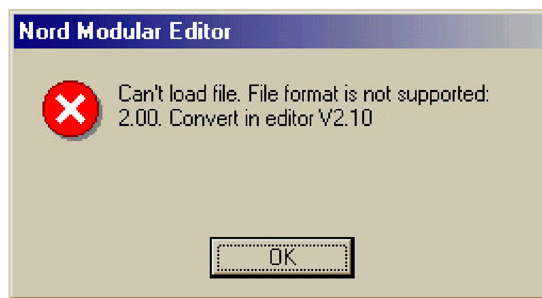


Bild 3-4
Ett exempel på hur felmeddelanden ofta används. Programmet i det här exemplet styr en extern maskin som ger ifrån sig ljud. Varje ljud har en egen fil varav vissa filer är skapade i en äldre version av programmet. Denna dialogruta visar sig varje gång man försöker öppna en gammal fil. Till saken hör att man byter ljud ofta och inte ser skillnad på vilka filer som är skapade i den äldre versionen förrän detta meddelande dyker upp.
En bättre lösning vore att, utan att stoppa arbetsflödet genom att behöva stänga en dialogruta, i huvudfönstrets nederkant påvisa att filen inte går att öppna och varför. Detta kombinerat med att skapa filikoner som ser annorlunda ut beroende på vilken programversion som skapat filen.

Interaktion

För att kunna föra in data eller på något vis påverka ett programs beteende använder vi oss idag av i huvudsak två externa verktyg – musen och tangentbordet. Dessa kommer med stor sannolikhet att över tiden förändras och till slut ersättas av andra hjälpmedel, men idag är de oumbärliga för att kunna interagera med ett dataprogram.

Musen

Fysiskt sett är antalet interaktionsmöjligheter som går att utföra med musen få. Man kan röra den över skärmen, peka på olika saker och trycka på knapparna. Ytterligare former av interaktion sker genom en kombination av dessa grundformer. Mushandlingar kan även

kombineras med tangentbordskommandon och på så vis göras än mer kraftfulla. De mushandlingar som normalt sett kan finnas implementerade i olika program utan att blanda in tangentbordskombinationer är följande:

- Peka (Peka)
- Peka, klicka, släppa (Klicka)
- Peka, klicka, dra (Markera)
- Peka, klicka, dra, släppa (Klicka-och-dra)
- Peka, klicka, släppa, klicka, släppa (Dubbel-klicka)
- Peka, klicka, klicka på en annan knapp, släppa, släppa (Ackord-klicka)
- Peka, klicka, släppa, klicka, släppa, klicka, släppa (Trippel-klicka)
- Peka, klicka, släppa, klicka, dra, släppa (Dubbel-dra)

Givetvis kan alla dessa handlingar utföras med vilken musknapp som helst. Och om musen har fler knappar blir det teoretiskt möjligt att utföra kvadrupel-klickningar. Men detta är sällsynt då musens styrka ligger i dess enkelhet. Om man bryter ett så enkelt och invant beteendemönster som hur musens handlingar utförs, måste det finnas mycket starka skäl som legitimerar detta. Frågan är om sådana skäl ens existerar.

Med musens hjälp kan användaren utföra två grundbeteenden: Välja någonting, och välja att göra något med det som valts. Det finns många nyanser av dessa båda beteenden och en designer har stora möjligheter att påverka hur enkelt ett program blir att använda genom att implementera mer eller mindre av dessa generellt sett standardiserade beteendemönster.
Att koppla en viss tangentbordsknapp med en mushandling innebär att handlingsmöjligheterna blir vidare och kraftfullare. De knappar som används är de så kallade meta-tangenterna vilka är operativsystemspecifika. Vid användning av metantangenter är det viktigt att följa den etablerade standarden samt om ytterligare funktionalitet läggs till, dokumentera denna väl.

När man rör markören över skärmen kan denna förändras beroende på vad som är möjligt att genomföra. Denna åtgärd fungerar som hjälpfunktion och visar exempelvis på att en tabells storlek går att justera. Ett sådant förfarande har även det upphöjts till standard och när man idag använder sig av standardklasser vid gränssnittskonstruktion får man dessa funktioner på köpet. När det gäller hur musen fungerar och hur markören förändras i ett fönster förväntar sig användaren inga överraskningar. Det är viktigt att följa den etablerade standard som finns och stödja alla de funktioner som användaren kan förvänta sig.

Tangentbordet

Tangentbordet används främst vid införandet av tecken till ett program men fungerar även som styrmekanism och funktionsaktiverare. Användaren kan manövrera sig genom exempelvis ett dokument eller en meny med hjälp av tangentbordet och hon eller han kan även utlösa funktionalitet genom att använda kortkommandon.

Taktiska verktyg

Fönster, menyer, dialogrutor och knappar är några av de mest synliga tillbehören i ett GUI. Dessa element är effekter snarare än orsaker av en god design. Varje komponent fyller ett syfte och detta syfte bör medvetandegöras under designprocessen för att elementen tillsammans ska kunna skapa en konstruktiv helhet.

Menyer

Menyer är ett av de element i ett GUI som successivt har fått en standardiserad form. De flesta GUI har menyrubriker som kallas för ”File” och ”Edit” längst åt vänster och ”Help” längst åt höger. Detta är så vanligt förekommande, även över olika plattformar, att man kan tala om en standard. Normalt sett bör utvecklare följa etablerade standarder för att inte förvirra användare. Men om vissa element i ett GUI framtagits och upphöjts till standard på felaktiga grunder bör man enligt Cooper frångå denna standard och bidra till att skapa en ny och mer korrekt sådan. Om vi utgår från att interaktionen mellan användare och program ska återge och spegla användarnas mentala modell och dessutom vara organiserad på ett målorienterat vis blir konsekvensen att den standardiserade namnsättningen på menyerna ”File” och ”Edit” är missvisande. ”File”-menyn är namngiven efter hur ett operativsystem fungerar och ”Edit”-menyn baseras på det utklippningssystem (clipboard) som operativsystem använder sig av. Det gemensamma för dessa båda exempel är att de speglar hur en dator fungerar och inte bidrar till att på ett intuitivt vis reflektera de programspecifika uttrycken som grundar sig på användares mentala bilder. Men förfarandet är komplicerat, att byta ut något som alla vant sig vid är riskfyllt och för att menyernas innehåll ska spegla användarnas mål med programmet och samtidigt inte irrtera eller förvirra måste designen grundas på samverkan och samförstånd. Överhuvudtaget är det viktigt att finna en logiskt menystruktur som gör att funktionalitet är enkel att finna och utnyttja. Det finns nog inget mer irriterande än att leta efter något man vet ska finnas inom räckhåll men ändå inte finna det.

En förnuftig utgångspunkt vid design av menyer är enligt Alan Cooper att gruppera dem efter verksamhetsområde och vikt. Detta innebär att de mest globala företeelserna styrs under menyalternativ till vänster i programmet och handlingarna blir mer och mer specifika ju längre åt höger vi rör oss. Om detta synsätt appliceras på exempelvis ett dokumentcentrerat program kan en stiliserad meny se ut som följer:

Program	Document	Pieces	Help
Properties Views Functions Access	Properties Views Functions Access	Properties Views Functions Access	

Bild 3-5

En grovt stiliserad bild över ett förslag på en menystruktur. Längst åt vänster placeras handlingar av bredast omfattning; under ”Program”-menyn. Handlingar av mindre omfattning placeras åt höger i fallande dignitetsskala, först representerat av ”Document”-menyn och sedan av de mindre delar som inkluderas i ett dokument genom ”Pieces”-menyn. Varje meny representeras genom de fyra aspekterna: Properties, views, funktions och access, genom vilka användaren kommer åt programmets handlingar.

En mer realistisk syn

Den ovan presenterade menyindelningen skiljer sig kraftigt från den etablerade, vilken representeras av huvudgrupperingar som ”File”, ”Edit”, Windows” etc. Att i ett steg förändra något som successivt växt sig starkt kommer förmodligen att väcka reaktioner. Även om man kan motivera en lösning utifrån teoretiska och logiska argument, och till och med bevisa att denna lösning är bättre, är det inte säkert att den går att genomföra. Detta beror antingen på att

omgivningen helt enkelt ännu inte mogen, eller på att standarder vanligtvis sätts av mäktiga aktörer som genom finansiell tyngd kan påverka vad andra ska tycka och köpa.

Om vi istället för att automatiskt designa ett GUI utifrån ett fil-centrerat synsätt och istället väljer ett dokument-centrerat blir konsekvensen att ”File”-menyn byter namn till ”Document”-menyn. Detta mindre ingrepp borde inte nämnvärt störa användarnas förmåga att finna sig till rätta i programmet. I ett kort tidsperspektiv kan förändringar av standarder innebära viss förvirring men om förändringarna är logiska och användarcentrerade borde de vara enkla att lära sig. En menys uppbyggnad är viktig, speciellt om programmet innehåller mycket funktionalitet, och om vi utgår från den etablerade menystrukturen så finns det en rad designtips att beakta. Några av dessa är:

- Använd inte ”Edit”-menyn som soptipp för funktioner som inte verkar passa någon annanstans. Samla istället ihop dessa i exempelvis en ”Options”- eller ”Preferences”-dialogruta som nås genom ”Tools”-menyn.
- ”Insert”-menyn är egentligen bara en förlängning av ”Edit”-menyn och om få saker styrs härifrån kan de lika gärna flyttas till ”Edit”-menyn.
- Om programmet har en ”Setting”-meny, placera då alla programmets inställningar här.
- Lås de menyalternativ som för tillfället inte ska gå att välja.
- Ha inte för många undermenyer.
- Använd likadana symboler i menyer som i verktygsfält för att visualisera samma funktionalitet.

Kortkommandon

Ett kortkommando innebär att en funktion aktiveras genom att en kombination av tangenter trycks ned. Vanligtvis sker detta genom att användaren först trycker ned en så kallad META-tangent, som ”CTRL”, ”ALT” eller ”SHIFT”, för att sedan kompletteringstrycka med en annan tangent vilken är kopplad till önskad funktion. Det finns två grundtyper av kortkommandon, de som direkt utför en funktion – så kallade ”Accelerators”, och de som man med tangentbordstryck kan manövrera bland menyalternativen – så kallade ”Mnemonics”. Vid design av kortkommandon bör följande beaktas:

- Följ standarder.
- Designa dem för ett dagligt användande.
- Visa tydligt hur man kommer åt dem.

Dialogfönster

Vi har tidigare utmålat dialogfönster som ett flödeshämmande moment som bör undvikas. Givetvis går inte allt som ett program ska utföra att samla i ett fönster. Ibland måste användaren tvingas att konversera med programmet genom andra fönster än de som betraktas som huvudfönster. Denna process bör göras så smärtfri som möjligt, bland annat genom att presentera informationen i fönstret på ett stilsäkert och självinstruerande vis och att språkligt sett formulera textmeddelanden så att de är begripliga och därmed användbara.

Det finns två typer av dialogfönster: modala och ickemodala. Den modala varianten är vanligast och innebär att hela systemet fryses när dialogfönstret visas, när de ickemodala fönstren visas är dock allt aktivt samtidigt. Det gemensamma för de båda typerna är att de har minst en knapp som stänger fönstret och vanligtvis brukar de ha två knappar som utför kommandona ”OK” och ”CANCEL”. Men för övrigt beror deras innehåll helt på vilken situation de är designade att lösa.

Modala dialogfönster

Generellt sett är den modala formen av dialogfönster enklast att förstå för både användare och utvecklare. När fönstret frammanas blir resten av systemet omöjligt att interagera med, det är endast med dialogrutan som det går att kommunicera. Detta innebär att den information som presenteras i dialogrutan alltid är korrekt sett ur ett tidsperspektiv, resten av systemet är ju fryst så inga ändringar går att utföra.

Ickemodala dialogfönster

När ett ickemodalt dialogfönster är uppe fungerar resten av systemet som normalt. Det uppenbara problemet med ickemodala dialogfönster är att om systemet forsätter att operera och uppdateras kan denna information hamna i konflikt med dialogfönstrets. Dessutom ökar risken för fönsternedsmutsning och att fönster gömmer sig bakom varandra.

Innehåll och form

För att kunna designa ett fungerande och effektivt dialogfönster bör designen vara väl förankrad i de målorienterande användarmönster som föreligger. Dialogfönstret bör utföra de uppgifter som det instiftats för på enklast möjliga vis. Ur ett målorienterat perspektiv finns det fyra olika variationer av dialogfönster[27]: egenskaps-, funktions-, periodiska- och process-relaterade.

Egenskapsdialogfönster presenterar inställningar och karakteristika för specifikt valda objekt och tillåter användaren att förändra dessa. Objekten kan vara generella, som hela programmet eller ett dokument, eller mer specifika, som att förändra ett aktuellt typsnitts egenskaper. Egenskapsdialogfönster fungerar som en kontrollpanel med exponerade konfigurations-möjligheter för det aktuella objektet. Egenskapsdialogfönster är oftast modala men kan även vara ickemodala.

Funktionsdialogfönster framkallas oftast genom anrop från menyn. Genom dessa fönster styrs en specifik funktion som utskrifter, infoga objekt etc. Vid sidan av att användaren kan utföra en handling kan denna handling även ofta modifieras. Funktionsdialogfönster är nästan uteslutande modala.

Periodiska dialogfönster är en av de mest avskydda delarna i ett traditionellt GUI då de ofta är helt meningslösa och därigenom förargelseväckande. De utgörs av ett fönster som förmedlar ett medelande, ofta av typen ”error”, och minst en knapp som kan tryckas på för att godkänna att man deltagit i informationsutbytet. Problemet med dessa dialogfönster är inte dess form utan att de används slentrianmässigt och utan att vara speciellt genomtänkta. De frammanas inte av användaren utan genom att något oförutsett händer i programmet. För att legitimera periodiska dialogfönsters existensberättigande måste dessa förmedla information av sådan vikt att man inte kan vara utan denna för att kunna fortsätta med sina aktiviteter. De kan även förmedla kritisk systeminformation innan ett allvarligt fel inträffar men då bör denna information vara utförlig och tydlig. Oftast används dock dessa modala fönster för att förmedla smådetaljer som lika väl skulle kunna förmedlas genom huvudfönstret.

Processdialogfönster framkallas i likhet med periodiska dialogfönster av systemet. De indikerar att programmet är upptaget och ber användaren ha överseende med detta. Det finns en viktig informationsförmedlande aspekt i designen av processdialogfönster, att systemet talar om vad som försiggår och hur lång tid detta kommer att ta. Dessutom bör användaren

[27] Cooper, 1995, (sid. 317)

uppmärksammans på att operationen är fullständigt normal och ha möjlighet att avbryta operationen. Istället för att förändra markören till ett timglas kan exempelvis ett självstängande processdialogfönster användas.

Bild 3-6
Ett exempel på ett väldesignat processdialogfönster är kopieringsfönstret i Windows. Användaren får information om att systemet utför en kopieringsprocess, hur lång tid detta tar och har möjlighet att avbryta. Användargränssnittet förmedlar även en känsla av att allt är som det ska, systemet utför bara en process som tillfälligt försämrar systemets kapacitet.

Etikett

Syftet med ett dialogfönster är att erbjuda användaren en tjänst som inte går att förmedla genom det ursprungliga fönstret. Som vi tidigare sett kan denna tjänst exempelvis bestå i att ändra vissa inställningar, utföra en inställningsbar funktion eller att meddela att systemet är upptaget. Dialogrutor kan vara designade på ett mer eller mindre väluppfostrat vis och även om den situation de används i är legitim kan beteendet och vokabulären hos dialogfönstret irritera. Generellt sett bör designen av dialogrutor inte resultera i att de upplevs som oförskämda och oförutsedda. Istället bör de upplevas som artiga och hjälpfulla följeslagare.

Två av de mest fundamentala och uppenbara regler som dialogfönsters beteende bör uppfylla är att de kommer upp i mitten av programmet och att de går att flytta. Vidare bör all text som ska vara till för att hjälpa och stödja vara formulerad på ett sätt som just hjälper och stödjer. Detta genom att exempelvis undvika användning av avslutande ord då dessa direkt kommer i konflikt med orden ”OK” och ”CANCEL”, vilka används som aktiverande knappar. All text bör formuleras så att användaren inte behöver fundera över vilken betydelse som förmedlas och hur han eller hon ska agera. Om det är mycket information som ska presenteras kan man använda sig av flikförsedda dialoger men för att underlätta användbarheten bör flikarna inte lagras på varandra det räcker ofta med ett lager. Om all information inte får plats är det bättre att skapa större eller fler fönster.

Felmeddellanden

Felmeddellanden är något negativt, de signalerar att något är fel i programmet och att användaren har utfört en förbjuden åtgärd. Oftast visar de sig som dialogrutor vilket stoppar flödet, och den information som förmedlas genom dem är ofta av uppläxande och oförskämd karaktär. Cooper menar att designen ska sträva efter att eliminera felmeddellanden. Detta går bara att genomföra om det inte finns någon möjlighet för användaren att inte göra några fel, vilket kan låta som en utopisk tanke om man tänker i implementeringstermer. Men om vi bortser från om programmets funktionalitet ska vara hundra procent felsäkert eller inte och istället tänker på hur användarflödet i ett GUI ser ut, kan man genom att kartlägga detta flöde

stoppa alla möjligheter för användaren att utföra fel saker. Om ingångsvägarna till felen täpps till och designen utgår från hur framtida användare kommer att använda ett program borde antalet felmeddelande kunna decimeras till ett minimum.

Verktygsfält

Ett verktygsfält är en samling knappar, vanligtvis med bildmotiv som brukar vara placerade under menyn. Ett verktygsfälts status kan vara fast eller flytande. Ett flytande verktygsfält går att flytta över skärmen och kan placeras var som helst. Fördelen med verktygsfält är att användaren kommer åt funktioner snabbt och enkelt, nackdelen är att om funktionskvantiteten är betydande kan verktygsfältet bli rörigt och uppta stor plats på skärmen. Dessutom finns det en inlärningströskel på aldrig använda symboler. Men detta kan lösas genom att låta användaren konfigurera sitt eget verktygsfält och välja vilken funktionalitet som ska gestaltas genom knappar. Inlärningsfrekvensen minskar dessutom betydligt om knapparna förses med förklarande verktygstips som visas när musmarkören förs över en knapp.

Gizmos

Gizmos är ett samlingsnamn för direktmanipulerbara, självförsörjande och på skärmen visuellt synliga idiom. Exempel på traditionella gizmos är knappar, textfält, listboxar, och ”comboboxar”. Antalet gizmos som vi lärt oss att använda har de senaste tio åren ständigt ökat. Olika programmeringsspråks standardklasser för GUI följer denna trend och erbjuder successivt fler och fler färdigdesignade komponenter. Enligt Alan Cooper kan man med utgångspunkt ur användarnas mål urskilja fyra typer av Gizmos: imperativa, selektiva, förevisande och ingångsgizmos[28].

Imperativa gizmos används för att initiera en funktion. Knappar är en av de delarna i ett GUI som nästan uteslutande är imperativa. Textknappar finns som färdigdesignade komponenter i ett klassbibliotek men knappar med ikoner på, så kallade buttcons[29], måste skapas. Att designa buttcons är inte helt trivialt. En imperativ knapp sätter vanligtvis igång en process och denna process är ett verb. Ett substantiv kan vara enkelt att fånga och gestalta visuellt men ett verb är svårare. Kvaliteten på de buttcons som utvecklas beror på hur bra den visuella metaforen som gestaltar verbet har blivit och detta kan påverkas genom att finna en bättre grafisk design. Bilder är enkla att komma ihåg och de tar mindre plats än text. Väl designade buttcons med vidhäftande funktionalitetsförklarande verktygstips sparar plats i ett fönster, är självinstruerande och förmedlar en känsla av professionalism och skicklighet.

Selektiva gizmos används för att välja något bland olika alternativ eller data. Exempel på selektiva gizmos är checkboxar, radioknappar och listboxar. Vad man bör tänka på vid design av dessa är att när antalet valbara objekt ökar bör objekten grafiskt på något vis särskiljas från varandra. Detta genom att exempelvis använda sig av illustrerande bilder eller färger.

Förevisande gizmos används för att direkt förändra programmets utseende visuellt. Dessa gizmos används för att hantera den visuella presentationen av information på skärmen. Exempel är ”scrollbars” och ”screensplitters”.

Ingångsgizmos används för att föra in data i systemet. Det mest uppenbara exemplet är ett text-redigera-fält. Huvudfönstret i ett program består förmodligen av ett ingångsgizmo och många egenskaps- och funktionsdialogfönster innehåller fler eller färre av dessa. Om

---

[28] Cooper, 1995, (sid. 372)
[29] Cooper, 1995, (sid. 375)

designen kräver att flera ingångsgizmos ska samsas i samma fönster kan man för att avskilja dem använda sig av ramar och på så vis binda samman data som hör ihop.

3.2 Testning

Med utgångspunkt i en generell systemutvecklingsmodell, vattenfallsmodellen eller *lifecycle development methodology* som den också kan kallas, beskriver vi den roll testning har för att säkerställa användarnas och systemets krav på kvalitet och funktionalitet. Här följer en översiktlig beskrivning av de olika testfaserna som sedan fördjupar sig i den situation som råder under unittestningen.

Testning i mjukvaruutveckling

> *Testing, together with verification and validation, constitutes a rigorous framework for for assuring quality of software products. This framework requires a well planned application of specific standards and practises implemented by all participants throughout the the software development and maintenance lifecycle. Application of these standards and practises implies the ability to evaluate the software product in order to assess its correctness, reliability and level of confidence required to accept it.*[30]

I vanliga fall uppfattar man testning av mjukvara och system som det led i systemutvecklings-processen som följer på implementeringen av systemet och säkerställer att det tillfredsställer kraven från användare och design. Man analyserar, designar, programmerar, testar och sen är systemet klart. På detta sätt fungerar det nog också i väldigt många fall även om andra strömningar börjar få fotfäste. I de flesta systemutvecklingsmodeller som används idag och i majoriteten av litteraturen om testning (tex. Beizer, Lewis, Krawczyk & Wiszniewski), påpekas dock vikten av att man samtidigt under analys och utvecklingsarbetet, verifierar samt validerar systemet på varje nivå och under varje fas av arbetet.

- Testning innebär oftast att man experimenterar med ett systems programkod med olika indata, tillstånd och scenarion för att jämföra det verkliga resultatet med det förväntade resultatet på ett kontrollerat och systematiskt sätt. Syftet är att påvisa önskad funktionalitet och saknaden av oväntat eller felaktigt exekveringsbeteende. Det ger ett naturligt verktyg att mäta och bedöma systemets kvaliteter.
- Verifiering av ett system stöds lämpligtvis av testningsexperiment för att kontrollera om produkten av en given utvecklingsfas (tex. fysisk design) tillfredsställer de krav som ställts på den vid inledningen av samma fas. Verifiering syftar till att säkerställa att *produkten är gjord på rätt sätt*.
- Validering innebär att man värderar en produkt under eller vid slutet av systemutvecklings-processen för att se om den tillfredställer användarnas och kundens krav. Valideringen syftar till att säkerställa att man *gjort rätt produkt*.

Testning, tillsammans med verifiering och validering bör ses som en process som pågår under hela systemutvecklingen. Såsom beskrivs nedan i kapitlet om *Testning i livscykelmodellen*. De två huvudsakliga typerna av testning är statisk och dynamisk.

[30] Krawczyk & Wiszniewski, 1998, sid 133

Statisk testning

Statisk testning är ett samlings begrepp för olika test-tekniker som inte inbegriper exekvering av binärkod utan analyserar programmeringskoden utifrån design och syntax. Kompilering med dess syntaxkontroll kan således betraktas som statisk testning. Ytterligare metoder är till exempel *program inspection*, där man kontrollerar program koden rad för rad mot den relaterade designen; och *structured walkthroughs*, där man följer logiska scenarion steg för steg och jämför design och kod deras beteende.

Dynamisk testning

Är således ett samlings begrepp för den testning där man exekverar binär programkod eller åtminstone simulerar den genom att beskriva dess logik steg för steg. Black-box testing, white-box testing, boundary testing är alla exempel på dynamisk testning.

I de följande kapitlen kommer vi främst beskriva de faktorer och den roll som den dynamiska testningen spelar för de olika testfaserna. Orsaken är att denna är relevant för fallstudien och den statiska testningen är inte det.

Testning i livscykelmodellen

Orsaken till att vi tar upp livscykelmodellen är för att det är en välkänd och representativ modell för systemutvecklingsarbete. Dess faser och ingridienser finner man hos de flesta andra utvecklingsmetoder, även om man fokuserar på olika saker och använder sig av en annan struktur och förlopp återfinns samma moment och begrepp hos de flesta. Analys, design, implementering, testning och arkitektur. Framförallt används den i den organisation som står i fokus i fallstudien.

Livscykelmodellen och dess faser

I vattenfall eller livscykelmodellen bryter man ned utvecklingsprocessen i klart avgränsade faser, där varje fas har en tydlig sekventiell följd med en väl definierad början och slut. Varje fas skall vara helt avslutad innan man påbörjar nästa. I teorin är tanken den att när man väl avslutat en fas skall man inte återgå för att göra förändringar.

Den första fasen är användaranalysen, *user requirements*, där användarna är intervjuade och deras önskemål och krav är analyserade och resulterar i ett dokument som specificerar användarnas krav på systemet. Vid vidareutveckling av existerande system så införlivas omdesignen i denna fas, där hänsyn tas till de nya krav som ställs på systemet.

I nästa fas, logisk design, skapar man klassdiagram över relationer mellan objekt, beskriver strukturerade processer och dataflödesdiagram för att strukturera upp en detaljerad bild av systemet ur ett perspektiv som påvisar vilken data och funktionalitet som ska finnas.
Resultatet från den logiska designen används sedan för att definiera och utveckla systemets fysiska design. Systemet delas upp i olika moduler och komponenter, till exempel en datalagringsenhet, olika funktionella komponenter och gränssnitt. Varje modul beskrivs var för sig, men koncentrerar sig på strukturen dem emellan och den kommunikation som skickas mellan dem. Slutligen specificeras varje enskild komponents indata samt utdata och systemets strukturella arkitektur definieras.

Under den sista fasen, moduldesignen eller *unitdesign*, definierar man varje komponent i detalj och specificerar dess datatyper, algoritmer och logik via abstraktioner som sekvens, iteration och selektion. När detta är klart övergår man till kodningsfasen, där man implementerar varje enhet för sig på valt programmeringsspråk för att sedan testas under testfaserna.

Testning som "a continous improvement process"

Enligt W.E.Lewis så ingår mjukvarutestning som en väsentlig del av kvalitetssäkring av mjukvarusystem. Dess syfte är verifiera och validera utvecklingsarbetets olika aktiviteter så man garanterar att mjukvarans design, programkod och dokumentation motsvarar de krav som ställts på dem. Testningen fokuserar på planering, design, utveckling och genomförande av tester. [31]

Testningen skall till stor del helst genomföras av en annan organisation en den som utvecklar mjukvaran. Orsaken till detta är bland annat uppfattningen att fel uppstår på grund av att programmerare inte förstår designen och systemkraven och att det är svårt för utvecklare att ställa om från det kreativa utvecklingstänkandet till det "destruktiva" felsökandet som testning innebär. Även om kan tycka att sådana idéer är rena missuppfattningar kan man inte argumentera mot värdet av att en neutral part objektivt granskar arbetsresultatet.

Oavsett om man har tillgång till en separat testningsorganisation eller inte bör man förkasta den gammalmodiga uppfattningen om att testning sker efter implementation, utan snarare se det som en kontinuerlig process som följer parallellt med utvecklingsprocessen och integrerad med denna.

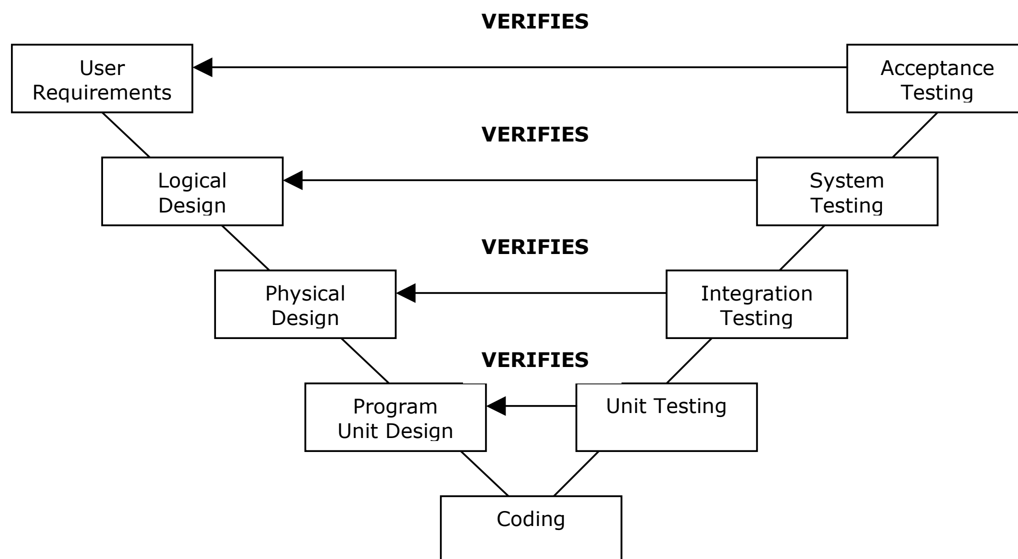


Bild 3-7 Development phases vs. testing types

[31] Lewis, 2000, sid 42

För varje fas i utvecklingsarbetet finns det en motsvarande fas i testningsprocessen så som vi kan se i bild 3.7.[32] Under varje enskild fas skall det definieras vilka målen är med testningen, dvs. vilket resultat man förväntar sig uppnå med testningen. Man skall fastslå vilka krav som skall valideras och verifieras av testen. Med detta som utgångspunkt skall det definieras en plan för att på lämpligt sätt tillfredställa kraven och fastställa de medel som krävs för att genomföra detta.

Testplan - steg för steg:[33]

1. Define test objectives.
2. Develop the Test Approach.
3. Define the Test Environment.
4. Develop the Test Specifications.
5. Schedule the Test
6. Review and Approve the Test Plan.

Utifrån planen skall man designa, utveckla och genomföra testfall. Dessa skall slutligen dokumenteras, analyseras och resultera i feedback till systemutvecklarna för eventuella justeringar av design respektive programkod. Varje nivå ses ur sitt eget perspektiv med dess specifika krav och resulterar i en egen testplan.

Acceptance testing

Vid analysfasen definierar man en kravspecifikation på systemet. När man gjort ett första utkast till en sådan låter man testteamet granska denna genom statisk testning, dvs genom inspektion, checklistor och *walkthroughs*. Om man inte finner några brister i analysdokumenten är man redo för nästa fas, om inte så ger man utvecklingsteamet feedback för justeringar och sedan upprepas processen.

Den färdiga kravspecifikationen används för att definiera en testplan som skall validera systemet gentemot kravspecifikationen. Detta är den sista fasen av testningen som ser till att det färdiga systemet motsvarar förväntningarna. Man testar på en högre abstraktionsnivå och kontrollerar att den generella funktionalitet och kvalitet som krävs, verkligen realiserats.[34] Testningen skall besvara frågor som: Är systemet tillräckligt säkert, användarvänligt eller exakt? Är svarstiderna snabba nog? När denna testfas är avklarad betecknas systemet som flygfärdigt.

System testing

I den logiska designfasen utgår man från kravspecifikationen och definierar en datamodell bestående bland annat av klassdiagram och objektrelationer, en processmodell samt kopplingen mellan data och processer. Det sammanfattas i vad man kallar systemspecifikationen. På samma sätt som under acceptance testningen, granskar man nu (via statiska testmetoder) systemspecifikationen så att den inte innehåller brister eller att det saknas detaljer för att realisera det kravspecifikationen fastställt. [35]

När den logiska designen är klar utgår testteamet från denna för att utveckla en testplan. System testningen ägnar sig åt att validera denna design. Man kontrollerar systemet som en

[32] Lewis, 2000, sid 38
[33] Lewis, 2000, sid 48f
[34] Lewis, 2000, sid 61ff
[35] Lewis, 2000, sid 67ff

helhet. Man bryr sig inte om hur systemets olika delar är kopplade eller hur väl dessa delar fungerar var för sig. Man kontrollerar att de processer som definierats i processmodellen beter sig som önskat. Man följer dataobjekt, dess tillstånd, beteende och relationer, för att kontrollera att detta stämmer med datamodellen.[36]

Integration testing

Den fysiska designen utgår från systemspecifikationen och skapar systemarkitekturen. Hur systemet skall delas upp i fysiska och logiska delar och hur dessa skall implementeras. Man definierar relationerna mellan de olika delarna och hur dessas gemensamma kommunikation skall se ut. Man fastställer varje enhets in respektive utdata. Testteamet granskar systemarkitekturen via statisk testning för att kontrollera att den stämmer överens med de riktlinjer och krav som ställts av systemspecifikationen. Brister rapporteras för att justering och omdesign, detta upprepas tills systemkraven är tillfredsställda.[37]

När den fysiska designen är klar utgår testteamet från denna för att utveckla en testplan. Integrations testningen syftar till att verifiera systemarkitekturen och den fysiska designen. Testningen involverar sammanslagningen av två eller flera delar till en större enhet. Man kontrollerar att den sammanslagna enheten fungerar som planerat och att de överenskomna kopplingarna mellan de olika delarna stämmer med designen. Att gemensam data är riktig, att meddelanden följer överenskomna normer och att funktioner tar emot och returnerar rätt parametrar.[38]

Unit Testing

Under unitdesignen, även kallat den detaljerade designen, bestämmer man varje enhets specifika data och algoritm struktur utifrån den anvisningar som givits i systemarkitektur och systemspecifikation. Man specificerar det detaljerade flödet av kontroll, så att det på ett enkelt sätt ska gå att översätta designen till programkod. Testteamet kontrollerar att den designade strukturen är i samklang med kraven från systemarkitekturen, så att det skall gå att införliva enheten i helheten. Kontroller görs för att se om det finns brister i flödesstrukturen och algoritmerna. Detta kontrolleras genom s.k. *structured walkthroughs*, då det gås genom strukturen hos det designade programmet för att eliminera logiska fel. Proceduren upprepas tills alla krav är mötta.[39]

När den detaljerade designen av enheten är klar skapar man en testplan utifrån denna. Unittestningen syftar till att verifiera den detaljerade designen och se till att programenheten är redo att införlivas i resten av systemet. I kapitlet nedan skall vi ge en mer utförlig beskrivning av den problematik och de faktorer som man möts av under unittestningen.

Unittestning och Debugging

Som för alla andra faser av testningen behöver man en testplan för att kunna genomföra testningen på ett systematiskt och tillförlitligt sätt, skillnaden är att det oftast är programmeraren själv som genomför och planerar dessa test. Varje enhet är oftast så omfattande att en eller ett fåtal programmerare kan hantera programmeringen själva. Eftersom

---

[36] Krawczyk & Wiszniewski, 1998, sid 193
[37] Lewis, 2000, sid 73ff
[38] Krawczyk & Wiszniewski, 1998, sid 132
[39] Lewis, 2000, sid 77ff

man under unittesten skall kontrollera varje logisk konstruktion i enheten och det är programmeraren som skapat dessa samt ofta även designat dem, är han också den mest lämpade att utföra testningen. Testning av en logisk konstruktion innebär på denna nivå till exempel: Kontroll att en sekvens, iteration, selektion och instanser av objekt beter sig som väntat.

Man kan argumentera för att detta borde gälla på de andra testnivåerna också eftersom det även då är fråga om testning av exekverbar kod. Skillnaden är dock den att man då testar systemet på högre abstraktionsnivåer snarare än som under unittester då man tittar på hur algoritmer i en mindre enhet beter sig. Här handlar det istället om att se hur samma enhet kommunicerar med resten av systemet, om programmet håller sig inom ramen för nyttjat närminne eller om det exekverar snabbt nog.

Testfall

Unittestningen består oftast av ett begränsat antal testfall, utvalda utifrån ett oändligt antal möjliga fall. Testfallen bör täcka och kontrollera alla realistiska situationer som programmets beteende ger upphov till. Varje fall utförs i form av scenarion där specifika testdata testar förutbestämda beteendeområden och gränsvärden. Exekveringen av programmet under ett testscenario kräver att man kan inhämta information om programmets data och processer samt att man kan manipulera dessa data och processer under testkörningens gång. Informationen lagras oftast i speciella loggfiler och programmet utsätts för förändringar via script som definieras i förväg av testaren. Den registrerade datan utgör resultatet från testningen som sedan analyseras och jämförs mot det förväntade resultatet. Bild 3.8 visar ett testfalls livscykel.[40]

Logfilen med den registrerade informationen innehåller vanligtvis följande:

- Tidpunkten för varje registrerad data med utgångspunkt från tex. programmets start eller stopp.
- Programmet exekveringskontext när datan registreras. Dvs. värdet hos relevanta programräknare, specifika input och output förållanden m.m.
- Värdet på lokala och globala variabler som är relaterade till den registrerade händelsen.

[40] Krawczyk & Wiszniewski, 1998, sid 189

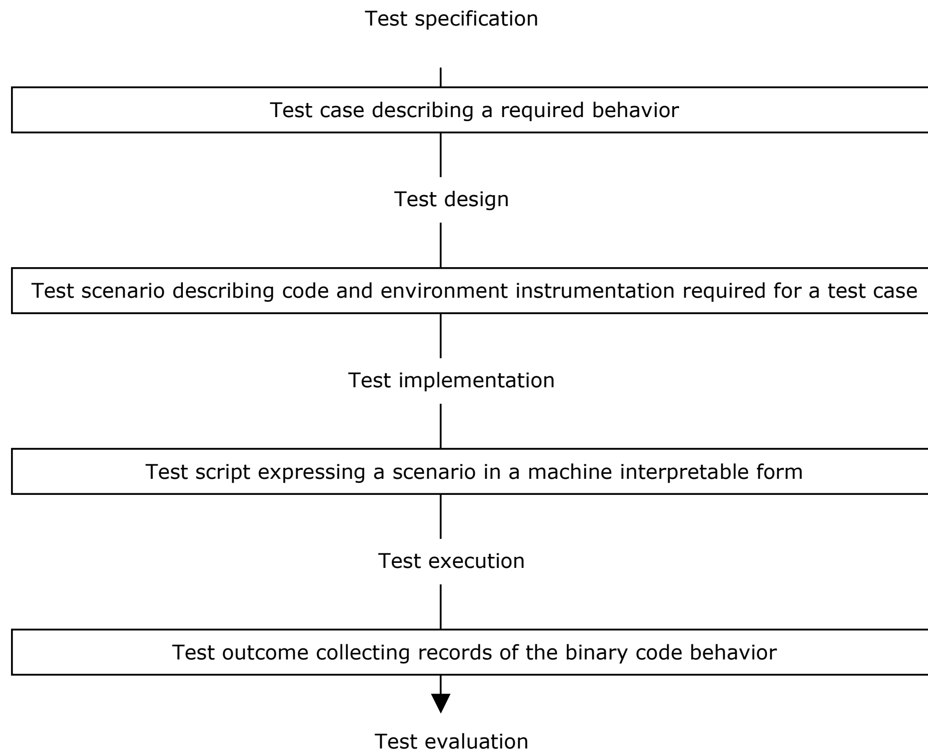


Bild 3-8 ”Life-cycle” of a test case.

Debugging kontra testning

När man genomfört ett testfall och resultatet inte överensstämmer med det förväntade resultatet måste man finna orsaken till avvikelsen i programmet. Då debuggar man programmet för att identifiera och rätta programmets felaktigheter. Bild 3.9 visar hur testning och debuggning samverkar med varandra.[41] När ett fel påträffas designar men ett nytt testfall för att ”zooma in” på defekten. Man kan få justera det nya testfallet flera gånger för att komma till botten av problemet som uppstått. När man väl funnit orsaken till felet så rättas det till och sedan återupptar man testningen där man avbröt senast.

[41] Krawczyk & Wiszniewski, 1998, sid 191

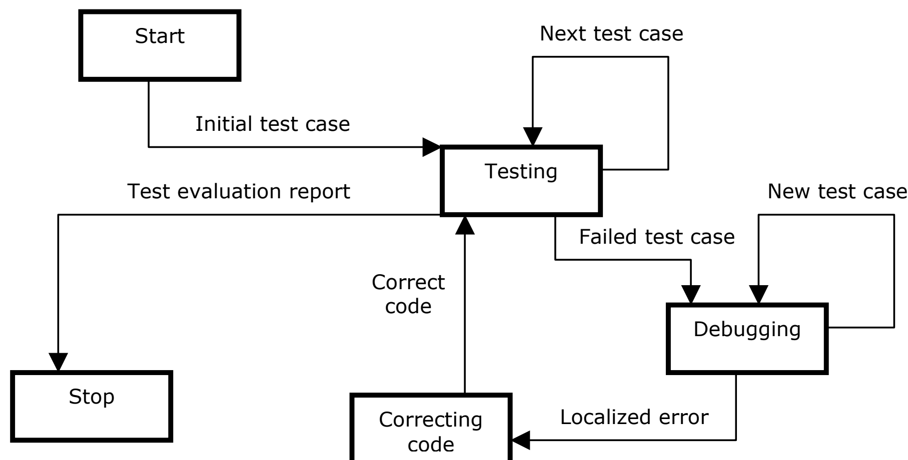


Bild 3-9 The testing-debugging cycle.

Blackbox och whitebox

Det finns i huvudsak två grundläggande synsätt på dynamisk testning. Det ena är Black Box testing som beskrivs i detalj av Boris Beizer i *Black-Box Testing: Techniques for Functional Yesting forTesting of Software and Systems*. Black box testning kallas även för funktionell testning där testförhållandena baseras på programmets eller systemets funktionalitet. Med utgångspunkt från systemspecifikationen så inhämtar man information om den input och output som systemet kräver och genererar. Tekniken bryr sig inte om hur systemet eller enheten fungerar internt utan kontrollerar bara om det tillfredställer de yttre specifikations-kraven. Typer på sådan testning är : testning av gränsvärden, undantag och databasintegritet, random testing, limit testing och range testing.

White-box testning, eller strukturell testning, bygger å andra sidan på att testningen har sin utgångspunkt i utvärderingen av logiska följder. Fokuset ligger på systemets interna logiska struktur. Testfallen designas så att man manipulerar samt följer testdata så att programmets logik och struktur undersöks. För att kunna utföra denna typ av tester krävs det att testaren känner till programkoden.

Gränsnitt för testverktyg

Om man studerar grafiska användargränssnitt för olika testverktyg kan man urskilja vissa gemensamma grunddrag. Dessa reflekterar generella egenskaper hos testverktygen som man måste ta hänsyn till vid utformning av GUI:s för testverktyg. Vi skall redovisa de mest elementära beståndsdelarna i sådana verktyg och beskriva dess grundläggande funktionalitet.

Software Monitoring

För att genomföra meningsfull testning måsta man kunna inhämta olika data från det testade systemet. Sådan ”övervakning” (eng. monitoring) är en förutsättning för att kunna analysera testresultatet och hitta orsakerna till eventuella felaktigheter. När man övervakar programvara följer man händelser och datavärden under exekveringen genom att införliva speciella

instruktioner i programkoden. Dessa kallas ”breakpoint traps”, vilket är villkor som uppfylls i vissa situationer och då förflyttas kontrollen av programmet till en separat programenhet som registrerar information om programmets tillstånd. Dvs. värden hos olika lokala och globala variabler. Den registrerade informationen lagras vanligtvis i en eller flera loggfiler eller redovisas ”online” i ett grafiskt gränssnitt.

Innehåll i loggfiler

Övervakning används för att observera beteendet hos ett program som kör i en given datamiljö. Data om programmets beteende och tillstånd lagras och ger möjlighet till felsökning. I loggfilen registreras fakta om specifika händelser. Vilken typ av data och händelser som är intressant att registrera skiljer sig givetvis från fall till fall. Nedan följer några exempel på intressanta händelser:[42]

> *Starting process, ending process, sending message, receiving message, beginning transaction, ending transaction, opening file, closing file, accessing file, exception thrown, exception caught, security authorization, failures etc.*

Detaljnivå

Det kan också vara relevant att ha möjligheten att välja till vilken grad man skall övervaka systemet. Möjlighet att välja bort viss typ av information och bestämma vilken detaljnivå den skall registreras på. Sådan valmöjlighet måste programmeras in i övervakningsinstuktionerna i koden. Villkorsförhållanden som kontrollerar önskad ”nivå” och ger följaktliga resultat i loggfilerna.

Oavsett om man har valmöjligheter gällande logginnehållet är det viktigt att nivån på eller mängden av information är tillräcklig för att ge testaren skall kunna analysera resultatet på ett meningsfullt sätt. En måttstock på miniminivån som krävs är att loggresultatet bör skilja sig mellan ett testfall och ett annat, hur liten skillnaden än är mellan testdatan i de olika fallen.

Avläsningsmetoder

Det finns i huvudsak tre olika tekniker för övervakning av mjukvarusystem:

- *Hårdvarubaserad teknik* – man avläser hårdvarusystemet externt med hjälp av emulatorer och *logical analyzers*, för att observera mjukvaran utan att påverka den. Begränsningen med den typen av observation är att den insamlade datan är på en mycket låg abstraktionsnivå, det kostar tid och pengar att analysera ända ner till maskinkodsnivå.
- *Mjukvarubaserad teknik* – den vanligaste metoden, som kortfattat innebär att man placerar instruktioner direkt i programkoden. Instruktioner som exempelvis skriver valda data till fil eller skal. Detta erbjuder information till en hög abstraktionsnivå som är lätt fär testaren att tillägna sig. Nackdelen med metoden är tidsförlusten som uppstår när man avbryter programmets naturliga sekvens för att registrera datan, samt att detta kan påverka det totala systemets beteende.
- *Hybridteknik* – en kombination av de ovanstående metoderna vars implementation kräver system med speciell hårdvaruarkitektur. Kortfattat innebär det att instruktioner finns i koden men avläses externt via hårdvaruteknik. Resultatet ger en hög abstraktionsnivå till en liten åtkostnad i tid och påverkan.

---

[42] Krawczyk & Wiszniewski, 1998, Table 5.1

Distribuerade System

I distribuerade system där man använder flera processorer och/eller exekverar parallella processer uppstår situationer då det kan vara svårt att observera systemets beteende som helhet. Problematiken beror oftast på tidsförhållanden mellan olika realaterade processer. När data om en process inhämtas kan det orsaka fördröjningar för resten av systemet. Exempelvis kan en process vara tvungen att vänta in att en annan skickar ett meddelande eller har avslutat en uppgift. I förlängningen kan det störa systemet som helhet och inte bara försämra tidsprestandan.

Främsta problemet är dock inte att man stör systemet, vilket givetvis bör undvikas, utan istället svårigheten att inhämta data från olika processer samtidigt och ge ett meningsfullt resultat till testaren. För att skilja på processerna måste man ge dem var sin unik identifierare, data om var processen exekverar och exakta tidsangivelser för all registrerad data. Huruvida man väljer att registrera all data i samma loggfil eller skilja processer åt genom att tilldela separata loggfiler för var och en, skiljer sig från fall till fall.

Genom att låta varje process i systemet registrera data om sig själv vid fasta tidspunkter kan överblick fås över ett distribuerat system. Ett sådant så kallat distribuerat *snapshot* innebär att alla processer synkroniseras så att datainhämtningen sker samtidigt. På så sätt minimeras extra tidsfördröjningar vid kommunikation mellan processer. Data registreras om processernas lokala tillstånd samt tillstånd hos kanaler för utdata och indata.

Gränssnitt – för utvärdering av testresultat

Ett grafiskt gränssnitt (GUI) kan stödja experimentering och testning av mjukvarusystem. Ett sådant stöd implementeras antingen som en del av mjukvarusystemet eller utgörs av ett externt testverktyg som startar och övervakar programvaran. Oavsett implementation innefattar det så gott som alltid de följande operationerna:

a) Initiering av den övervakande miljön. (testverktyget eller den del av gränssnittet som visar testresultaten)
b) Uppstart av programmet eller del av programmet som övervakas.
c) Interaktiv konfigurering av testdata.
d) Visualisering och utvärdering av den registrerade datan.

Informationen från testfallen visas antingen *online,* direkt under exekveringsperioden genom att grafiskt redovisa händelser och data när de sker, eller *offline,* genom att till exempel visa relationer och beroenden mellan olika poster i den loggade informationen.

Online visualiering

Under testkörningen kan det vara intressant att följa förändringar i data, tillstånd och händelser när de sker. Dels ger detta en mer övergripande bild av händelseförlopp än om man analyserar loggdata, men framförallt är det en fördel om man vill interagera med det testade programmet under körning.

Det finns många sätt att visualisera data online, men ändå inte lätt då informationen som visas bör vara relevant, lättavläst och ge användaren en omedelbar överblick. Enklaste metoden är förmodligen att redovisa den loggade informationen direkt i ett fönster. Om man har flera processer med olika logginformationen är det meningsfullt att använda multipla fönster. Problemet är oftast då det gäller stora mängder information, så är det inte meningsfullt att visa

hela loggfilsresultatet online. Dessutom sker exekveringen och utskriften så snabbt att ögat inte förmår att följa den. Därför måste man sålla bland informationen och endast välja ut övergripande data för de viktigaste entiteterna. Vanliga sätt att redovisa testresultat online är i tabellform, grafer, animerade symboler etc. Den lämpligaste formen avgörs beroende på typen av mjukvara, testdata samt testmiljö och får anpassas efter dessa förutsättningar.

Stöd för offline analys

Efter att testfallet genomförts påbörjas analysen av testresultatet. Analysen skall besvara frågor som: Beter sig programmet som förväntat? Om inte: Har fel uppstått? Var, hur och varför uppstod dessa fel?

Resultaten från testningen har i detta läge redan registrerats i tex loggfiler. För att underlätta analysen av informationen behöver man verktyg som stödjer tracing, utsökning och visualisering av relationer mellan testdatan.

Det vanligaste tillvägagångssättet är att viktiga händelser registreras i loggfiler och testaren måste leta igenom dessa för att finna eventuella fel och orsaken till dessa. Testaren känner oftast till vad han behöver söka efter och använder sökverktyg som kommer med operativsystem eller enkla texteditorer. Sökningen sker på nyckelord och testaren prövar sig fram. Tillvägagångsättet är ganska primitivt och skulle kunna förbättras. Exempel på stöd för analysen är bland annat:

- Sökning på nyckelord.
- Möjlighet för testaren att definiera vilka data som skall visas. Genom att definiera olika satslogiska villkor för typen av information som skall visas. Tex. Visa poster som innehåller information av typen A OCH B men INTE C.
- Istället för dynamiska och föränderliga objekt och data som används online, bör man kunna redovisa testresultatet i ”snapshot” liknande bilder. Tillståndet hos relaterade data visas beroende på den tidpunkt som är intressant. Moment för moment stegar man igenom datan i tidsföljd.
- Redovisa tillstånd hos entiteter och relationer genom listor, tabeller, grafer, strukturella scheman och symboliska figurer.

Testverktyg

Testverktyg kan skilja sig på många sätt från fall till fall, men det finns generella egenskaper som är gemensamma för de flesta kommersiella verktyg. De verktyg på marknaden som är anpassade för distribuerade system hanterar i allmänhet följande:[43]

- Processer – antingen lokala eller distribuerade, för att bestämma deras status och manipulera och påverka dess exekveringsbeteende.
- Data – för att redovisa, manipulera och behålla specifika variabelvärden.
- Avläsning – för att dynamiskt kunna föra in, ta bort och förändra avläsningsvillkor i programkoden, dvs. vad som loggas.
- Meddelanden – för att registrera innehållet i meddelanden, mäta datamängden, vasändare, mottagare, länkning och andra egenskaper hos den skickade datan.
- Loggfiler – registrering av programmets exekveringshistoria.
- Testskript – för att kunna köra specifika fördefinierade testscenarion.

[43] Krawczyk & Wiszniewski, 1998, sid 243f

3.3 Kvalitet

ISO-modellen

Vikten av att fokusera på kvalitet vid utveckling av mjukvara kan ej nog poängteras, och det har inte bara att göra med att uppnå långsiktig vinst. På grund av komplexiteten i mjukvaruprodukter, och de ofta förekommande förändringar som måste bearbetas under utvecklingen av mjukvara, är det rent nödvändigt att bibehålla uppmärksamheten på kvalitetsfaktorerna om vi skall kunna förverkliga tillfredsställande produkter.[44] Detta förstärks ytterligare med tiden då mjukvara mer och mer blir en del av vårt vardagliga liv. Produkter med låg kvalitet gör dels kunderna missnöjda, men leder också till att användarna försummar systemen som var ämnade att stödja deras arbete.

ISO - the International Standards Organisation har etablerat flera standarder som är ämnade att underlätta kontrollen över kvalitet. Den mest tillämpliga i vårt område, utveckling och underhåll av mjukvara, är ISO-9126. I denna nyligen utgivna version har ytterligare ansträngningar gjorts för att definiera en uppsättning kvalitetsegenskaper. Där kvalitetsfaktorer och kriterier definierade av McCall och andra är närbesläktade med varandra är ISO-modellen hierarkisk - varje subkriterie är relaterat till exakt ett kriterie. ISO-modellens kriterier är strängt relaterade till en mjukvaru*produkt*. Deras definitioner inbegriper inte *process*kvalitet. Vidare har subkriterierna att göra med kvalitetsaspekter som är *synliga* för användaren. Reusability, till exempel, är inte inkluderat i ISO-modellen. På ett sätt är det så att de tidigare taxonomierna reflekterar en produktsyn, medan ISO 9126 reflekterar användarnas syn på kvalitet.[45]

För att kunna kontrollera mjukvarukvalitet krävs att man vet vad kvalitet är. Inom varje given organisation kan man definiera de kvalitetskriterier som är viktigast. Dessa kan skilja sig från organisation till organisation. Oftast är det så att man inte kan eller behöver fokusera på ISO-modellens samtliga kriterier samtidigt. Vissa, ofta mindre omfattande applikationer, kräver endast att särskild fokus läggs på ett visst kriterie. Vid utveckling av ett grafiskt användargränssnitt kommer vi främst att inrikta oss på att uppskatta användbarheten - **Usability**.

ISO 9126

Functionality - *attribut som är relaterade hur väl mjukvaran utnyttjar de resurser som ställs till dess förfogande*

- *suitability* - till vilken grad mjukvaran har de rätta funktionerna för de uppgifter den skall utföra.
- *accuracy* - till vilken grad mjukvaran producerar rätt/överenskomna resultat.
- *interoperability* - till vilken grad mjukvaran existerar i enlighet med rådande lagar och regler.

---

[44] Hans Van Vliet, *Software Engineering – principles and practice*, John Wiley&Sons, Ltd, Chichester, England, 2000

[45] Vliet, 2000

- *security* - till vilken grad mjukvaran är kapabel att förhindra access till oauktoriserade användare eller program.

Reliability - *attribut som är relaterade till hur mjukvaran klarar av att behålla sin prestandanivå under förbestämda förutsättningar under en specifik tidsperiod*

- *maturity* - antalet fel mjukvaran visar upp då körningsfel uppstår.
- *fault tolerance* - mjukvarans förmåga att upprätthålla en viss förutbestämd prestandanivå då körningsfel uppstår.
- *recoverability* - mjukvarans förmåga att återfå sin prestanda efter ett strömavbrott eller hårdvarufel samt dess förmåga att reparera eventuella dataförluster.

Usability - attribut som är relaterade till hur lätt det är att tillägna sig och använda mjukvaran för de tänkta användarna.

- *understandability* - den grad av lätthet med vilken användaren kan känna igen och ta till sig de logiska koncept som mjukvaran presenterar.
- *learnability* - graden av ansträngning som krävs av användaren för att lära sig hantera mjukvaran.
- *operability* - graden av ansträngning som krävs för att hantera operationella kontrollaspekter som mjukvaran kräver. Ex. backup, filhantering etc.
- *attractiveness* – Mjukvaruproduktens kapacitet att bli omtyckt av användaren.
- *helpfulness* – graden av stöd för att lära sig mjukvaran.
- *customisability* – graden av stöd för valmöjligheter och anpassningar.

Efficiency - *attribut som är relaterade hur väl mjukvarans utnyttjar de reurser som ställs till dess förfogande.*

- *time behavior* - mjukvarans förmåga att tillgodose krav på svarstid, processtid, överföringshastigheter etc.
- *resource utilization* - hur väl utnyttjas den kringutrustning som mjukvaran använder (i/o-enheter, lagringsenheter etc). Stor mängd kringutrustning kan resultera i sämre prestanda.

Maintainability - *attribut som är relaterade till hur lätt det går att underhålla samt modifiera mjukvaran.*

- *analyzability* - graden av ansträngning som krävs för att hitta anledningen till körningsfel i mjukvaran.
- *changeability* - graden av ansträngning som krävs för att modifiera eller ”återställa” fel.
- *stability* - risken för oväntade fel vid modifieringar.
- *testability* - graden av ansträngning som krävs för att validera/testa den modifierade mjukvaran.

Portability - *attribut som är relaterade till hur lätt det är att förflytta mjukvaran från en miljö till en annan.*

- *adaptability* - mjukvarans förmåga att anpassas till olika miljöer.

- *installability* - graden av ansträngning som krävs för att installera mjukvaran i en specificerad miljö.
- *co-existence* - mjukvarans förmåga att uppfylla portabilitetsstandarder.
- *replaceability* - mjukvarans förmåga att ersätta annan specificerad mjukvara i samma miljö.

Del 4 Förberedande analys

För att urskilja den problemsituation som vi ställts inför vid utformningen av gränssnittet utförde vi en förstudie av användarna och verksamheten. Uppgiften från ERV var att endast utforma ett grafiskt gränssnitt till ett befintligt system (arbetsstationslösningen, ASL) och inte att utveckla en mjukvara från början till slut. Det fick följden att vi fokuserade mer på användarna och mindre på problemområdet. Områdesanalysen blev begränsad till att studera det system som redan var i drift med utgångspunkten att dess egenskaper och funktionalitet skulle stå fast. Vi intervjuade användarna, studerade manualer och annat dokumentmaterial samt installerade och nyttjade systemet. Det fanns två syften med förstudien:

1) Ta första steget i prototypingprocessen. Genom att snabbt göra en analys av användare och verksamhet, kunna producera ett första utkast till prototyp.
2) Urskilja de problem som är unika för den här typen av uppgift. Problematiken kring design av grafiska gränssnitt för en viss typ av användare i en viss typ av miljö.

Resultatet av studien kommer redovisas med utgångspunkt från den andra punkten. Användarna och miljön presenteras, men vi kommer inte använda någon standardiserad modelleringsteknik (tex. UML) eller redovisa information som bara är intressant för design och implementation av GUI-applikationen. Vi vill urskilja vilka speciella egenskaper hos användarna, och vilken speciella problem som uppstår med förutsättningarna i miljön, som kan spela roll vid utveckling av ett grafiskt gränssnitt.

Användarna

Det är speciellt viktigt att ha kunskap om användarna, vilka de är och hur de karaktäriseras, när man utvecklar ett användargränssnitt. Stor vikt bör fästas vid att bestämma användargruppens karaktäristik och kunskapsnivå, men givetvis också dess uttalade krav och önskemål, för att så långt det är möjligt anpassa produkten till användaren. Gränssnittet skall utformas med syftet att det skall användas för testning, men vi kommer också att resonera om den specifika typen av användare vi har att göra med. Om de har speciella egenskaper och behov som skiljer sig från normalanvändare av IT-teknik, hur deras användningsmönster ser ut, och om hänsyn behöver tas till detta.

Avdelningen som vi har studerat vidareutvecklar det befintliga systemet DPE vilket är en del av Ericssons nuvarande (GSM) och framtida mobilnätsystem (3G). Vi har samarbetat och intervjuat i huvudsak personal från denna avdelningen även om man kan tänka sig en vidare grupp av användare av vår framtida GUI-applikation. Utgångspunkten har varit att utveckla en preliminär produkt som kan vidareutvecklas och inledningsvis endast kommer att användas av den nämnda avdelningen. Därför begränsade vi arbetet och valde att inte intervjua personal på andra avdelningar och företag. Hänsyn har dock tagits till dessa, men i huvudsak så har användaranalysen skett med underlag enbart från den nämnda avdelningen.

Vem är användaren?

De berörda användarna kan beskrivas som så kallade *power users*, De har i allmänhet en gedigen datateknisk högskoleutbildning. De är dataingenjörer eller systemutvecklare, har

omfattande programmeringskunskaper och är väl förtrogna med Unixmiljön, den förhärskande utvecklingsmiljön på avdelningen.

Man arbetar med att vidareutveckla DPE-plattformen. Det rör sig om kontinuerlig mjukvaruutveckling där design, programmering och testning hela tiden avlöser varandra. Man jobbar i projektform där det ofta ingår flera personer.

Samtliga användare har således stora kunskaper om DPE-systemet och erfarenhet av ASL. Utvecklingen sker utan avancerade programmeringsverktyg (tex. 4GL), man använder istället enkla texteditorer, kompilatorer och debuggers med kommandobaserade gränssnitt och hög funktionalitet. Det dominerande programmeringsspråket är C och i viss mån Erlang.

Varje användare har flera ansvarsområden på avdelningen och är oftast involverade i flera projekt samtidigt. Deras scheman är ständigt fyllda med möten och deadlines. Man upplever en viss mån av stress - och om inte stress så åtminstone en klar tidsbrist. ASL som används vid testning av vidareutvecklade enheter är i bruk ett par, kanske fler, gånger per år per användare. Dessa tillfällen sammanfaller med slutfaserna av programmeringsprojekt då man testar koden intensivt i sammanhängande perioder. Eftersom testfasen gärna sker i slutet av utvecklingsprocessen närmar man sig deadline och bristen på tid är ännu mer akut. Då är det essentiellt att alla verktyg fungerar och att miljön är stabil. ASL är ett sådant verktyg och måste fungera. Problemet är att det är svårt att hantera och komplicerat till sin natur. Det tar lång tid att starta upp testmiljön och upprepa sina testscenarion.

Samtidigt ger ASL i sin nuvarande form användaren stor kontroll och möjlighet att manipulera systemet in i minsta detalj. Däremot saknas enkelhet och överblick. Om man introducerar ett nytt gränssnitt mot ASL funktionaliteten måste man övertyga användarna att börja använda det. Nyckeln till användarnas acceptans är att lägga till enkelhet och överblick men samtidigt bevara valmöjligheten och kontrollen.

Miljö

För att kunna utveckla ett adekvat mjukvarugränssnitt måste man också ta hänsyn till den miljö programmet verkar i, samt vilket syfte det har. Informationen och funktionaliteten bör visualiseras på ett sätt som är anpassat till den roll programmet spelar för användarna och måste vara möjligt att implementera på den plattform programmet används på. En av principerna för design av användargränsnitt, *user familarity*[46], påpekar betydelsen av att gränsnittet bör använda sig av uttryck och koncept som användarna känner igen från sin egen arbetsmiljö. Gränsnittet bör spegla den miljön så långt det är möjligt.

I detta fallet handlar det om testning och att utveckla ett gränsnitt för att manipulera, konfigurera och hantera Arbetsstationslösningen (ASL). ASL är en avskalad version av DPE-mjukvaran och går att köra på ett vanligt Unixsystem. Kortfattat ger ASL användaren möjlighet att starta DPE på en arbetsstation och samtidigt simulera att det finns fler processorer tillgängliga. Dessa kan utgöras av andra processorer i nätverket men också simuleras så att DPE ”tror” att en processor egentligen är flera. Under exekveringen registreras olika data från DPE i logfiler så att man kan följa olika händelser och variabler i efterhand.

[46] Sommerville, 2001, sid. 330

ASL är inte ett testverktyg i egentlig mening utan ger snarare utvecklarna av WPP-applikationer och DPE en möjlighet att testköra sin programkod. De kopierar över sina kompilerade programfiler och exekverar dessa med resten av systemet. Via logfilerna kan de följa testresultaten och genom att ändra vissa konfigureringsfiler kan de också definiera utseendet på noden som DPE skall köra på.

Testningsprocessen

På ERV använder man sig av samma systemeringsmodell som de flesta andra mjukvaruutvecklande avdelningar inom företaget. Den bygger på vattenfallsmodellen och följer i stort sett samma struktur som V-modellen som beskrivs utförligare i Teori-testningskapitlet (se sid 33). Skillnaderna är marginella och behöver inte beskrivas närmare.

Man följer dock inte modellen slaviskt utan justerar denna och definierar arbetsgången från projekt till projekt. Denna flexibilitet passar bra i synnerhet på ERV då uppgifterna ofta skiljer sig och nästan aldrig innebär att man utvecklar nya mjukvaruprodukter. Istället handlar det oftast om vidareutveckling av DPE eller delar av detta, det kan vara en uppdateringsmodul eller en utbyggnad. I sådana fall är livscykelmodellen inte alltid lämplig eller utförlig. På sätt och vis utgörs avdelningens arbete allt som oftast av underhåll av ett existerande system.

Inom företaget finns det en kvalitets- och testnings- avdelning som utförligt testar alla produkter som framställs. Deras testning utgörs av de senare delarna av V-modellen Acceptance testing och Systemtestning. Deras tester utförs alltid i de riktiga systemen och under realistiska förhållanden och kommer aldrig ha behov av ASL och vår gränssnittsapplikation. Däremot har de grundkrav på att de tidigare testningsfaserna genomförts framgångsrikt. Kraven definieras utifrån varje projekt och produkt men kräver i samtliga testfaser att programvaran slutgiltigt testats på en riktig nod med tillfredställande resultat. Den mesta testningen kräver dock att koden integrerats med DPE, inte nödvändigtvis den senaste versionen, och en fungerande testapplikation. Detta gäller körning både på riktig nod samt på arbetsstation. Systemet behöver inte kopplas ihop med resterande delar av WPP-plattformen, men kräver ändå en minimikonfiguration för att kunna köras då en enskild del av systemet inte kan köras ensam.

Utvecklarnas testningsarbete

ASL är det huvudsakliga testningsverktyget för programmerarna, det används under Unittestning och Integrationstestning samt som ett debuggnings-verktyg. Under det fortlöpande programmeringsarbetet kör och testar man koden på sin egen arbetsstation under relativt realistiska former. Viss mängd av hårdvara är inte tillgänglig men man kan nyttja andra arbetsstationers processorer eller simulera extra processorer på en och samma dator.

Slutligen tvingas man dock boka tid på en riktig nod för att testa på ”riktigt”. Fördelarna är dock väsentliga då man inte behöver administrera bokningar, vänta på sin tur eller för den del hantera testningen på nod som är än mer komplicerad än ASL. Det är framförallt tillgängligheten som är den viktigaste faktorn hos ASL. Programmeraren kanske gör en mindre justering och kan omedelbart testa denna utan att behöva boka tid. Han kan arbeta i sin egen takt och kontinuerligt kontrollera koden samt bli uppdaterad om sina framsteg eller felsteg.

Testningen som utförs på ASL är nästan uteslutande unittestning men även använd i integrationstester. Kraven på modulerna som testas definieras av projektet den ingår i men den testning som utförs på ASL bestäms oftast av programmeraren själv. Det är småtester som

utgör steg på vägen mot en fungerande och färdig modul. Tester för att kontrollera enskilda funktioner och delar av modulen, sökning efter buggar, kontroll av uppdateringar. Ett medel som ger utvecklaren möjliget att se att han är på rätt väg. Här följer ett par utdrag ur intervjuerna med utvecklarna för att ge en bild av testningsprocessen:

Utvecklare A:

- *Testning och kodning sker integrerat genom att först skriva kod, sedan testa denna kod med hjälp av egendefinierade testfall för att sedan förändra och sedan koda igen i en loop. Detta är renodlad unit-testning. Testprocessen består av tre steg för en enskild programmerare:*
    1. *Testa hela tiden under kodning med hjälp av egenformade testfall, renodlad unit-testning.*
    2. *Bygger in koden i DPE med hjälp av arbetsstationslösningen och testar mer verkliga scenarion i en integrerad miljö.*
    3. *Köra koden på en riktig nod nere i källaren.*

Utvecklare C:

- *Installerar en test-driver (ASL) helt lokalt på arbetsstationen applicerat med den modul som skall testas. Denna modul körs sedan med hjälp av en exempelapplikation på asl-installationen. Generellt följer jag inget specifikt mönster i testningen. Tester skall vara så lite formaliserade som möjligt och så enkla som möjligt att återupprepa.*

Utvecklare D:

- *Vidareutvecklar det som redan finns, bygger inte så mycket nytt. Skriver ihop det jag skall göra, och kör detta sedan på en riktig nod. När man kör mot noden används ett webb-gränssnitt. Om det är större paket som utvecklas skriver jag egna modultestfall. Dessa modultester följer ingen specifik norm. Använder bland annat c- testverktyget purify (rational) som bland annat hittar minnesluckor.*

Gemensamt för alla de intervjuade utvecklarna var att de inte följde några fördefinierade mönster eller typer av checklistor vid testningen. Sunt förnuft och anpassning efter situation får råda. Testning utförs inte heller i någon bestämd mängd, utan utförs i mån av tid och tills dess ”det funkar”.

Testningen består till stor del av typen ”White-Box Testing” (se sid 39) med utgångspunkt från programstrukturen för att kontrollera del för del av koden, samt debuggning. Detta utgör den grundläggande testningen av funktioner, algoritmer, anrop, input, output och variabelvärden.

Nästa fas i testningen syftar till att kontrollera hur modulen fungerar i sin ”normala” miljö. Modulen integreras i en version av DPE med hjälp av ASL och man testar dess funktionalitet. ASL loggar information som kan avläsas för att se hur modulen beter sig i samband med resten av systemet och hur systemets tillstånd förändras i sin helhet. Även om modulen integreras med resten av systemet så innebär det inte att det nödvändigtvis handlar om integrationstestning. Eftersom det oftast handlar om uppdatering och vidareutveckling av existerande moduler så finns det redan en fungerande ram för dessa moduler så de enkelt kan integreras med de andra delarna av systemet. Till exempel kanske man förändrar funktionaliteten i en modul för att möta nya krav men för den delen behöver man inte ändra kopplingen och kommunikationen till resten av DPE.

Förutom funktionaliteten har man också höga krav på pålitligheten (eng. Reliability) hos DPE. Eftersom processer och funktionalitet distribueras över flera processorer och applikationer nyttjas redundans (flera processkopior av samma typ) för att öka återhämtningsförmågan (eng. Recoverability) och feltåligheten (eng. Fault Tolerance) hos systemet. Detta testar man genom att döda olika processer för att se hur systemet reagerar och om det kan återhämta sig. Dessa kvalitetskriterier är visserligen högt prioriterade men ”processdödandet” är också en metod att starta upp stora delar av DPE:s inbyggda funktionalitet.

Testmiljön

Under det inledande programmeringsarbetet skriver programmerarna ofta egna små testprogram så att koden kan testas skiljt från resten av DPE. Man skriver en enkel main funktion och undviker (avmarkerar) kodkommandon som kräver DPE:s närvaro. Man använder kompilatorer och debuggerprogram för att hitta fel och rätta dessa. Arbetet ser ut på samma sätt som för de flesta programmerare:

a) Programmera begränsat avsnitt, tex. en mindre algoritm eller en for-slinga.
b) Kompilera koden.
c) Rätta eventuella fel.
d) Återupprepa b och c tills nytillkommen kod fungerar.
e) Återgå till a för nästa avsnitt och fortsätt tills programmet är klart.

Den här typen av kontinuerlig testning eller debugging under tiden man programmerar, löser dock inte alla problem och felaktigheter. Logiska fel och andra oegentligheter kan uppstå i en vidare miljö och sammanhang. Det enda som man vet är att koden går att exekvera i enskildhet utan större problem. Som hjälpmedel och verktyg för testningen använder man kompilatorer, debuggers, bland annat c-testverktyget purify (rational) som bland annat hittar minnesluckor, och egna printsatser i koden. Gränssnittet är således huvudsakligen text och kommandobaserat

När en enhet börjar få en tydlig ram och struktur och är möjligt att integrera med resten av systemet, vill man börja testa och köra på riktigt. Då får man kopiera över enheten i binärfil till rätt katalog i DPE-installationen. Man kör antingen på riktig nod eller så använder man ASL.

Översiktlig beskrivning av en nod

Hårdvaran som simuleras på arbetsstationslösningen består av rackmonterade kretskort av olika typer som tillsammans bildar en s.k. nod. Det är dessa kort man distribuerar ut sina processer på. Dessa kort är placerade enligt en viss hierarki:

- En nod innehåller ett eller flera s.k.magasin. Ett magasin har en eller flera portar (slots). Magasinen monteras i noden på fabriken, och kan inte tas bort av användaren.
- Ett craneboard är en korttyp som kan placeras i magasinets portar. Dessa kan sättas i och tas bort av användaren.
- Varje craneboard har en eller flera sub-portar(sub-slots) vari s.k. subboards monteras på fabrik. Dessa subboards kan inte tas bort av användaren.
- På ett subboard finns en eller flera processorer. Dessa processorer med operativsystem kallas också PM (Processing modules) och kan vara av typen PowerPc eller Sparc.

Varje hårdvaruelement i en nod identifieras av ett namn – s.k. Equipment Identifier. Detta unika namn visar placeringen av hårdvaruelementet. Enkelt förklarat är det så att magasin, craneboards, subboards och PM är numrerade. Således blir det en 4-siffrig unik identifierare för varje PM kallat positionsnummer.

Exempel: PM nr 1 på subboard nr 2 på craneboard nr 1 i magasin nr 2 får följdaktligen det unika namnet 2.1.2.1. se bild 4.1.

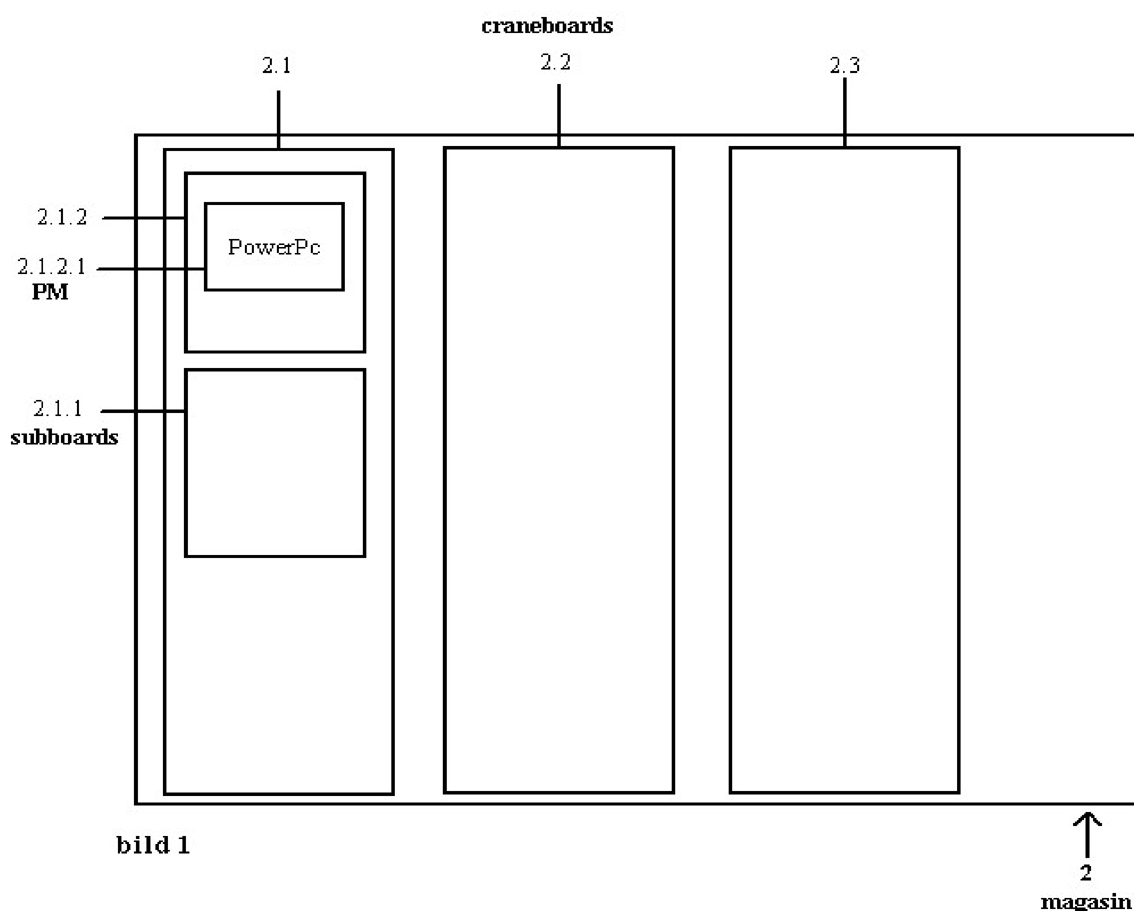


Bild 4-1 Översiktlig bild av en nod.

Arbetsstationslösningen (ASL)

ASL är en anpassad installation av DPE på en vanlig UNIX-station. När man packar upp och installerar mjukvaran krävs det flera beståndsdelar. Bland annat ett paket som innehåller den styrande mjukvarudelen av DPE, Node Delivery Package (NDP), samt ett fritt antal applikationspaket, Application Delivery packages (ADP). NDP innehåller mjukvara som hanterar och kör flera olika program, nämligen ADP-applikationerna, och fungerar ungefär som ett operativsystem.

Innan ASL kan användas på en arbetsstation måste en installation göras. Detta är ett moment som ofta ger användaren stora problem. Installationsprocessen är relativt komplicerad och kräver både kunskap om DPE/WPP i allmänhet, men också kännedom om var alla

beståndsdelar finns lokaliserade samt vilka moment som måste utföras. Manualerna till detta är inte särskilt utförliga och lättförståeliga utan installationen kräver oftast assistans från andra kollegor. Installationen behöver dock göras väldigt sällan, men i och med att det är så pass krångliga moment kan det finnas en stor önskan bland användarna om att få ett bättre stöd för detta i gränssnittsapplikationen.

Hur fungerar det?

Systemet är väldigt komplicerat och det är inte lätt att ge en kortfattad sammanfattning av hur det fungerar. Det följande kommer dock förhoppningsvis ge läsaren en fingervisning om hur ASL funkar och hur man hanterar det.

Mjukvaran innehåller en mängd olika shell-script och konfigureringsfiler som startar och kontrollerar de binärkodsfiler som innehåller den verkliga funktionaliteten hos DPE med applikationer.

För att kunna testa sin kod och köra ASL behöver man inte känna till alla detaljer. Man måste dock kunna manipulera skriptet som startar systemet (S20dpe_start.sh), tre stycken konfigureringsfiler (teie.dat, cbd-filen och *CoreNCLPort*-filen) samt var alla dessa filer befinner sig. Man behöver också veta var den nya och kompilerade enheten skall placeras, dvs. kopieras in i binärfilsform.

- Binärfilen som kopieras in utgör den testade programenheten, resten av systemet fungerar.
- Teie.dat (*Table of Expected Installed Equipment*) innehåller en lista över alla tillgängliga processorenheter (PM) med tillhörande positioner.
- Cbd-filen (*Crane Board Dictionary*) innehåller anvisningar för vilka hårdvaruenheter som systemet kan distribuera processer till. Vilken hårdvara som tillåts fungera som styrenhet (*Node Control Board*) samt vilka hårdvaruenheter som är tillgängliga för olika applikationer.
- *CoreNCLPort*-filen innehåller numret på den kanal som distribuerade processer på processorer inom nätverket kommunicerar på.
- S20dpe_start.sh är det skript som startar upp systemet. Man konfigurerar vart man startar upp kärnan i DPE samt vilka PM som finns tillgängliga inledningsvis.

När systemet startas upp så startas kärnan i DPE, *Node Control Logic* (NCL), på den PM som valts till att fungera som styrenhet. Samtidigt initieras en process på varje tillgänglig PM som hanterar kommunikationen mellan processorn och resten av DPE. Dessa kallas *Equipment Management Agents* (EQMA), och fungerar som agenter för DPE på respektive PM.

När en EQMA-process initieras definieras också dess position så att DPE kan skilja olika PM åt. Eftersom DPE använder positionen för att differentiera mellan processorerna går det att starta upp flera EQMA-processer på samma processor. DPE bryr sig bara om EQMA-processerna och ”tror” att det finns flera PM tillgängliga. På så sätt går det att simulera flera PM på en och samma processor.

Medan DPE kör så registrerar NCL data och händelser, som är centrala för sytemet, i en loggfil (ncl.log). På samma sätt registrerar också varje enskild EQMA-process data, som är specifika för den PM den representerar, i var sin separat loggfil (eqma*.log). Typen av information som återfinns i loggfilerna överenstämmer med de allmänna anvisningar för loggad data som beskrevs tidigare i kapitlet (se sid 40).

Exempel: Initieringsprocedurer, distribution av processer, statuskontroll av systemets delar och meddelanden från distribuerade systemenheter till centralenheten NCL.

Brister och fördelar

De flesta användare är överens om vilka som är bristerna och fördelarna med ASL som det ser ut idag. Bristerna har i stor utsträckning att göra med att det är svårhanterligt och svårt att överblicka. Fördelarna anses framförallt vara dess tillgänglighet och att den är billig. En annan viktig fördel är att det mesta av det som går att utföra på riktig nod faktiskt också går att göra med asl.

Utvecklare A:

> *Fördelarna har att göra med tillgängligheten och att det är en kontrollerad miljö. Nackdelar är att det krävs en massa scriptfippel och kopierande av en massa filer, dessutom är den komplicerad i sin natur.*

Utvecklare B:

> *Fördelen är att det är enkelt på så sätt att ASL hela tiden är tillgänglig. Som nackdelar kan jag nämna det att asl inte är någon riktig nod. Den körs inte på den riktiga hårdvaran vilket får tillföljd att den inte fungerar riktigt på exakt samma sätt. Detta ger utslag främst vid tidstester. Nätverket är dessutom av en annan typ.*

Utvecklare C:

> *Fördelen är att det går att simulera noden på en arbetsstation över huvud taget. Att det går att testa utan riktig nod. Nackdelar är att det är bökigt att komma igång, dels med verktyget men också med installationen. Eftersom verktyget används så pass sällan hinner man glömma det relativt komplicerade moment som installationen innebär.*

Utvecklare D:

> *Fördelen är att den är billig. Den är lätt att komma igång med på så sätt att den alltid är tillgänglig. Det mesta av den nya utvecklingen går att utföra med ASL. Nackdelen är att den är besvärlig att sätta upp och att hacka runt i scripten.*

Testverktyg utan gränssnitt?

I början av kapitlet ”Gränssnitt – för utvärdering av testresultat” (sid 41) beskrivs fyra elementära beståndsdelar hos ett testverktygs gränssnitt. Hanteringen av ASL sker visserligen utan hjälp av ett grafiskt gränssnitt, man använder texteditorer för att manipulera skript och skalkommandon för att initiera och avsluta skript och processer, men innefattar ändå de fyra omnämnda operationsområdena.

a) *Initiering av den övervakande miljön.*

Den främsta övervakningen av DPE sker genom att systemet loggar viktig information. Detta är inget som behöver initieras i förväg utan sköts automatiskt när man startar upp och kör ASL. Man kan dock ställa in vilken nivå man önskar logga, på en fem-gradig skala. Enligt de flesta utvecklarna är det främst en nivå som är intressant, då en lägre eller högre grad av loggning innebär extremt stora eller små mängder av information. Så valmöjligheten är ändå begränsad. Dessutom kan man föra in egna printsatser i koden som skrivs till loggfilen.

Någon egentlig övervakning av ASL kan man inte tala om. Men genom att bevaka slutet på loggfilerna via skalkommandon (UNIX-exempel: tail –f loggfil) kan man följa uppdateringen av loggen. Denna sker givetvis ohyggligt snabbt så att det knappt går att uppfatta vad som sker, men en van användare får ändå en viss överblick och kan utläsa när essentiella moment stegas igenom. Genom att använda utsökningskommandon som ”grep” kan man också leta upp specifik information under tiden det kör.

En ytterligare metod som används är att anropa operativsystemet och kontrollera vilka processer som kör, exempelvis återupprepade ”ps-kommandon”. Detta är en användbar metod eftersom distributionen av processer är en viktig del av DPE. Man kontrollerar att processer initieras som förväntat och om de fortsätter att exekvera utan problem.
Initieringen av övervakningsmiljön innebär således att man öppnar ett flertal kommandofönster. Ett för att startskriptet, ett för processer och ytterligare några för loggfilerna. Sen gäller det bara att snabbt skifta mellan fönstren och ge kommandon i tid. Sammanfattningsvis ett ganska omständigt arbete som kräver både kännedom om loggdatan, flinkhet och koordination, samtidigt som överblicken ändå blir begränsad.
Fördelen är att användarna oftast vet vad de gör, litar på övervakningsmetoden och är vana vid kommandobaserad datoranvändning. Nackdelarna är att det är omständigt, ger dålig överblick samt kräver mycket av användarna.

b) Uppstart av programmet eller del av programmet som övervakas.
Genom att kopiera in binärfilen som skall testas och sedan initiera startskriptet är tillräckligt för att starta upp DPE och påbörja testningen. Före uppstarten kan det dock vara intressant att manipulera konfigureringsfilerna för att köra andra testscenarion. Man kan bestämma antalet PM som skall finnas tillgängliga samt vilka positioner de skall ha. Vilka PM som är tillgängliga som styrenheter och vilka som tillåter att applikationer kör blockprocesser på dem. Förändringar i startskriptet bestämmer vilka PM som initieras vid uppstarten och vilken PM som tilldelas att köra NCL(Node Control Logic).

Det går att konfigurera på många sätt men man måste vara noga med syntax och att de olika filerna stämmer överens. Ett PM som initieras via startskriptet men inte är definierat i cbd-filen kanske inte tilldelas några applikationsprocesser eller orsakar andra problem. Det finns mängder av kombinationer som kan få systemet att inte fungera som önskat. Även om man vill testa sådana annorlunda konfigureringar så måste användaren känna systemet väl och veta varför det beter sig som det gör. Utan sådan kunskap kan en felaktig konfigurering lätt orsaka onödig huvudbry. Testaren kan misstolka avvikelser i systemets beteende och inbilla sig att det är den nya koden som är orsaken.

För att öka realismen vid testningstillfället kan man starta upp EQMA-processer på andra arbetsstationer via nätverket, men framförallt är det relevant att starta upp en ytterligare NCL-process externt. När det finns två NCL i DPE-systemet fungerar den ena som aktiv NCL och den andra som en backup. På så sätt har man möjligheten att simulera att den aktiva NCL går ner och observera om backup:en tar över som aktiv.
Främsta fördelar med att manipulera skript och konfigureringsfilerna är att testaren skaffar sig kontroll. Han manipulerar filer själv och vet vilka förändringar som gjorts. Nackdelen med detta är att det är mycket att hålla i huvudet samtidigt, samt att det tar tid och det är lätt att misstag uppstår.

c) Interaktiv konfigurering av testdata.

Efter uppstarten och initieringen av NCL och EQMA-processer finns det inte utrymme för så mycket ytterligare konfigurering. Filerna som konfigurerats används bara vid uppstarten, de läses in och datan lagras i minnet. Eftersom DPE inte läser av filerna igen under exekveringens gång är det meningslöst att ändra i dessa.

Det går ändå att påverka systemet externt genom att ”döda” processer genom operativsystemet. Detta simulerar händelser som om en PM hängt sig eller gått ner. Man kan också starta upp nya EQMA-processer, som simulerar nya tillgängliga PM, via skript eller skalkommandon. Det är viktigt att tänka på definitionen av position för nya PM, så att man inte skapar kopior av samma PM, dvs. samma position.

På grund av den extensiva felhanteringen samt distributionen av processer hos DPE, testar man gärna systemets stabilitet och återhämtningsförmåga. Då är det relevant att ”döda” och starta upp processer för att observera systemets beteende.

Fördelen är att man överhuvudtaget kan interagera mot systemet, men detta sker ju med extern påverkan och valmöjligheterna är begränsade. Någon kommunikation med systemet erbjuds inte.

d) Visualisering och utvärdering av den registrerade datan.
Som redan omnämnt under a) följer man data om systemet online genom att övervaka processer genom ps-kommandon och skriva ut loggfilsdatan medan den registreras. Men när testkörningen avslutats har man bara tillgång till informationen i loggfilerna, som nu skall analyseras.

Mängden information är mycket omfattande och tar tid att skumma igenom. Testarna använder oftast utsökningsverktyg via skalkommandon eller texteditorer. Man söker på nyckelord och visar information som innehåller sökorden. Det kräver att testaren vet vad han skall söka efter, även om det är samma person som testar, som utvecklar DPE, så känner man oftast inte till all loggdata.

Sammanfattningsvis innebär utvärderingsfasen att man öppnar och läser loggfilerna samt försöker finna fel och avvikelser.

Det är komplicerat och bökigt att hantera ASL som förutom loggfilerna inte erbjuder testaren några verktyg att arbeta med. De verktyg som nyttjas följer istället med operativsystemet och enkla texteditorer. Det är dock värt att anmärka att flera av användarna framhöll det enkla faktumet att ASL finns tillgängligt, som en av de stora fördelarna. Det påvisar problematiken med att testa på riktig nod och avsaknaden av en rimlig testmiljö.

Del 5 Hypotesformulering

Vår uppgift är att skapa ett så målcentrerat och därigenom avskalat program som möjligt. Med utgångspunkt i de framtida användarnas verklighetsbild och omgivning är intentionen med utvecklingsarbetet att hela tiden fokusera kring frågeställningen:

- *Hur utformas ett grafiskt användargränssnitt för testning av inbäddade system med syfte att optimera användbarheten för expertanvändare?*

Användarnas idéer och önskemål kommer i detta avsnitt att kopplas samman med våra egna idéer och den litteratur vi stödjer oss på. Tillsammans leder detta oss fram mot en definition av användarnas mål. Då frågeställningen - så här ska ett GUI för dessa användare och i den här miljön se ut - har sin utgångspunkt i vad Alan Cooper kallar användarnas mål blir hypotesframställningen liktydig med att definiera och komma fram till dessa mål. I detta avsnitt kommer vi att beskriva hur vi anser att ett GUI i den beskrivna kontexten bör se ut. Detta leder oss fram till att vi definierar av ett antal riktlinjer som om de följs skall förbättra kvaliteten hos det GUI vi sedan ska konstruera. Dessa riktlinjer utgör tillsammans grunden i vår hypotes. Vi delar upp diskussionen i tre områden, i var sitt kapitel: om användarna i kapitel 5.1, miljö i kapitel 5.2 samt avbildning i kapitel 5.3.

Riktlinjerna formuleras och deras innebörd sammanfattas i del 5.4 men de återfinns även i del 5.1-5.3 i parantesform efter varje stycke som på något vis relaterar till respektive riktlinje. Detta för att mer konkret ge läsaren ett stöd i hur vi har argumenterat oss fram till varje enskild riktlinje.

5.1 Användarna

Eftersom unittestning oftast utförs av programmeraren själv kan man utgå från att användaren av ASL, dvs. testaren, är mjukvaruutvecklare av något slag och att detta gäller för användarna av de flesta testverktyg. Man kan göra ett generellt antagande om att testare av mjukvara alltid har avancerade kunskaper om IT och de system de testar, samt att de ofta är systemutvecklare av något slag.

IT-utvecklare och programmerare är en användargrupp som skiljer sig från andra typer av IT-användare. Inte så mycket på det sätt människor tar till sig information och bearbetar den, utan på vilken nivå de gör det och i sitt beteendemönster. Den här typen av expertanvändare (*power-users*) av informationsteknologi har kunskap om de system de använder, de önskar information av detaljerat och tekniskt slag samt använder ofta andra och mer avancerade verktyg än en normal användare. Verktyg som inte nödvändigtvis behöver vara alltför användarvänliga utan snarare innehåller en hög funktionalitet. De har en vana vid IT-hantering och de generella begrepp, symboler samt användningsmönster som är gemensamma för gränssnitt i de flesta IT-system. Detta ger möjlighet att snabbt tillägna sig ny mjukvara och börja använda den.

Kontrollbehov

Den historiska utvecklingen av gränssnitt har gått mot att alltmer skilja användaren från den bakomliggande tekniken. De skall slippa att ta hänsyn till filhantering, felhantering och inställningar. Många operationer automatiseras och användaren skyddas från ”farliga moment”. Denna utveckling har fått till följd att en del av kontrollen över tekniken har flyttats från användaren till gränssnittet.

Den viktigaste skillnaden mellan expertanvändare och normala användare är just behovet av kontroll. Utvecklare behöver ha direkt kontroll över de system de utvecklar och den miljö de verkar i. Exakt information och kontroll över verktygen är absolut nödvändigt för att kunna analysera och evaluera utvecklingsarbetet.

Det här kontrollbehovet tar sig oftast uttryck i att man föredrar kommandobaserade gränssnitt, använder enkla textbaserade program med fokus på funktionalitet, samt manipulerar filer och inställningar manuellt via texteditorer och kommandoprompt. Det finns en klassisk motvilja mot operativsystem och programvara som flyttar kontrollen från användaren till gränssnittet, tex Microsoft Windows. Dessa typer av gränssnitt döljer funktionaliteten och tar bort valmöjligheter för användaren som kan vara farliga/kritiska för systemet. Information som är för teknisk eller abstrakt för normalanvändaren reduceras bort.

IT-utvecklaren vill dock ha kvar sådana möjligheter att påverka och övervaka systemet och om man skall utveckla grafiska gränssnitt för den typen av användare bör man ta hänsyn till detta.

(Riktlinje: Ge användaren kontroll över systemet)

Utrymmesbehov

En förlängning på resonemanget om att användarna behöver kontroll, är att de behöver överblick över alla parallella arbetsuppgifter. Testningen sker samtidigt med andra faser i mjukvaruutvecklingen som programmering och i detta arbete används en mängd olika applikationer samtidigt. Samtidigt behöver användarna utrymme för andra aspekter av det dagliga arbetet. Kommunikation via email, dokumenthantering och schemaläggning är exempel på detta.

Vår applikation skall passa in i denna miljö och den skall kunna köras parallellt med övriga program – utan att stjäla vare sig för mycket arbetsyta eller för mycket prestandaresurser. Därför bör man utveckla en ”parasitic-posture”-applikation. Med detta menas att den skall kunna exekvera ”i bakgrunden” utan att låsa upp arbetsytan och därmed göra det omöjligt att samtidigt jobba med övriga program. Således bör användargränssnittet byggas så utrymmessnålt som möjligt.

(Riktlinje: Tillmötesgå användarens utrymmesbehov)

Behov av stöd - som hjälp och feedback

På grund av den mycket höga kunskapsnivån hos användarna finns inget stort behov av en stor hjälpsektion. Den tid det tar att utveckla en uttömmande och korrekt hjälpsektion står inte i paritet med behovet av den. Den behöver endast innehålla det mest nödvändiga. Fundamental kunskap om DPE som användarna redan besitter och är en förutsättning för att utveckla och testa DPE behöver man överhuvudtaget inte ta upp. Endast en enkel beskrivning av de olika beståndsdelarna och hur man hanterar dem. Ambitionen är att göra gränssnittet tillräckligt enkelt och intuitivt så att de flesta operationerna är självinstruerande och självklara.

Av samma anledning kan man rationalisera bort andra hjälpverktyg helt och hållet, som tutorial och walkthrough. Ett sådant stöd är endast irriterande och tidsödande för användarna, programmet är så begränsat och skall vara så lätt att handha att om sådant stöd ändå behövs, är designen misslyckad överlag.

Däremot urskiljer vi ett behov av information som är teknisk och ger insyn i det bakomliggande systemet, ASL. Användarna har använt det tidigare och kan tänkas vilja göra manuella justeringar vid sidan om. De är vana vid ASL och litar både på systemet och sin egen kapacitet. Den feedback som strömmar mot användaren under hanteringen bör därför informera honom vad gränssnittet manipulerar i ASL. Förtrogenheten med tekniken och systemen innebär också att felhantering vid tex. undantag kan presenteras med de meddelanden som produceras av exekveringsmiljön, istället för att formuleras om och anpassas till mer ”användarvänlig” feedback. Det är nästan fallet att användarna redan är anpassade efter systemet och inte tvärtom.

(Riktlinje: Anpassa stöd efter användarnas intresse och kunskapsnivå)

Installationsprocessen

Om man införlivade ett installationsstöd i gränssnittsapplikationen skulle det innebära att användarna slapp ett ganska ansträngande arbetsmoment, som både tar tid att lära sig och utföra. Men det finns i huvudsak tre argument mot att göra detta.

1. Det är ett moment som utförs ytterst sällan.
2. Det finns en nytta med att användaren lär sig momentet. Man får en inblick i filstrukturen, vilka beståndsdelar som skall vara med. En kunskap som underlättar för användaren att göra manuella justeringar i framtiden.
3. Att bygga in stöd för installationsmomentet skulle ta mycket tid.

Framförallt är det ett oprioriterat moment som inte nödvändigtvis behöver finnas med. Om man vill behålla applikationen så enkel och avskalad som möjligt bör alla sådana typer av oprioriterad verktyg rationaliseras bort. Enkelhet underlättar hanteringen av och förståelsen för programmet.

(Riktlinje:Använd enbart befintlig funktionalitet och rationalisera bort oprioriterade moment)

5.2 Miljö

En av de främsta vinsterna med att utveckla ett GUI till ASL är att all funktionalitet på så vis samlas på ett ställe. Moment som konfigurering av noden, skript-initiering och övervakning etc. kan genom ett GUI på ett enkelt vis styras och hanteras. Samtliga intervjuade användare var överens om att gränssnittet borde samla och förenkla konfigureringen samt ge en möjlighet att starta och stoppa systemet. I övrigt skilde sig uppfattningen om hur detta skulle genomföras och vilka valmöjligheter som exakt skulle ingå. Men de gemensamma huvuddragen var en önskan om en ny gränssnittsmiljö som hanterade det mesta som berörde ASL samt tog liten plats på skärmen. De frågor som i detta stycke visas som rubriker har sitt ursprung i användaranalysen och fungerar här som en retorisk ingång mot att diskutera hur användarnas mål kan förbättras genom ett GUI.

Frågor relaterade till avgränsning

Eftersom det handlar om design av ett GUI till ett redan existerande ”testverktyg” finns det begränsningar för vilken typ av funktionalitet som kan införlivas. Den funktionalitet som redan finns i ASL är i stort sett den enda som kan bli representerad i gränssnittet. Det är visserligen möjligt att lägga till vissa *features* men det kräver en större insats och hör inte till gränssnittdesignerns jobb. Däremot kan man rekommendera förändringar i systemdesignen så att systemutvecklarna eventuellt skapar möjligheter att genomföra ytterligare gränssnittsidéer.

Vad finns tillgängligt och vad behövs?

Generella testverktyg (se sid 42) brukar kunna utföra ett mer varierat antal operationer än vad som erbjuds av ASL. Ambitionen är givetvis att låta testverktyget innefatta så många av dessa operationstyper som möjligt, i den mån det är genomförbart och meningsfullt. Hur dessa operationer är representerade i ASL och möjliga att införliva i ett grafiskt gränssnitt följer kortfattat:

Processer	– Processdata tillgänglig via systemanrop, kan redovisas i GUI.
Data	– Redovisas i begränsad form i loggfilerna. Enstaka data kan automatiskt läsas ur loggen och redovisas i GUI.
Avläsning	– Ej via GUI, utan görs av utvecklarna själva då de programmerar DPE.
Meddelanden	– Redovisas i begränsad form i loggfilerna, ej i GUI.
Loggfiler	– Registrerar all tillgänglig information om DPE och kan redovisas i GUI.
Testskript	– Kan genereras via GUI.

När det gäller data om systemprestanda, som kan vara intressant ur ett testningsperspektiv, så skulle det gå att hämta denna från operativsystemet, i begränsad form. Men det är inte någon relevant information i detta fall då ASL körs på lokala arbetsstationer och inte på riktig nod. Mätning av kvalitetskriterier som effektivitet samt i viss mån funktionalitet, är enbart intressant då mjukvaran befinner sig i sin riktiga och normala miljö.

Loggfilerna är det som användarna finner viktigast för testningen. Den biten är dock redan relativt tillfredställd. Förutom det urskiljer man en önskan att kunna förenkla konfigureringen genom att automatisera testskripten via GUI. Övervaka processer och manipulera dessa samt

förenkla uppstart och stopp av systemet. Ytterligare features, som inte på ett eller annat sätt redan erbjuds, är inte efterfrågade eller nödvändiga att lägga till. Det är snarare sättet existerande features representeras i GUI och vilka valmöjligheter dessa har, som är intressant.

(Riktlinje:Använd enbart befintlig funktionalitet och rationalisera bort oprioriterade moment)

Hur kan informationen i loggfilerna användas?

Den enda information som tillhandahålls av DPE-systemet lagras i loggfilerna och utöver det går det bara att se om processer kör eller ej. Om man programmerade om DPE skulle det visserligen vara möjligt att skicka den data som går till loggen, till GUI. Det skulle dock kräva orimliga resurser i tid och pengar. Dessutom skulle den nya koden kräva en extensiv testning, så att den nytillkommna koden på intet sätt påverkar DPE:s stabilitet och funktionalitet.

Däremot borde man kunna hämta datan direkt från loggen. Detta är en tekniskt genomförbar idé, men skulle kräva att man vore väl insatt i DPE-kodens utskriftsfunktionalitet. För att använda datan i loggfilerna måste man känna till de flesta möjliga scenarion (läs: känna till alla typer av utskrifter) för att ta hand om informationen på rätt sätt. Det kräver också en extensiv parsingfunktionalitet - läsa in och plocka ut specifika data ur ett enormt textmaterial. ERV-personalen skulle tvingas skriva de parsefunktionerna själva då det är omöjligt även för en väl insatt GUI-utvecklare att samla den nödvändiga informationen via analys och intervjuer. Det skulle ändå bara gå att hämta in en marginell del av datan i loggen, annars blir utvecklingsarbetet för att skriva parsefunktionerna för omfattande. En annan negativ faktor är faktumet att DPE hela tiden förändras och således förändras också logginformationen, vilket ytterligare försvårar parsingarbetet.

Det finns dock enstaka information i loggfilerna som är tillräckligt intressant för användarna så att den borde redovisas direkt i GUI. Information som användarna ständigt söker ut från loggen. Det mest uppenbara exemplet är när DPE meddelar att noden är startad till fullo, då alla delar fungerar och finns tillgängliga. Den typen av information är begränsad och utskriftsutseendet förändras nästan aldrig, vilket ger möjligheten att enkelt implementera detta i GUI:t. Information som på detta sätt rör generella tillstånd hos DPE, enkelt kan inhämtas och redovisas i GUI samt är av väsentlig vikt för testaren, borde inbegripas i gränssnitts-applikationen.

Hur skall loggfilerna redovisas?

Gränssnittsapplikationen skulle kunna erbjuda möjligheter att öppna loggfilerna i multipla fönster så att det går att följa uppdateringen online. Eftersom det är möjligt för testaren att läsa filerna med andra verktyg (skalkommandon eller texteditorer) borde GUI:t erbjuda samma typ av tracing och sökningstöd som dessa, och av samma kvalitet. En fördel med sådan lösning vore att tillgängligheten till loggfilerna skulle öka, möjligheten att spara ”sökprofiler” och att utsökningarna automatiseras. Profiler som specificerar olika sökbegrepp, tex. i form av satslogiska begrepp. Man listar sökta begrepp som på något sätt synliggörs eller extraheras ur loggtexten. Utsökningarna definieras då av användarna och blir inte låsta vid GUI-implementationen, ett problem som togs upp i avsnittet innan. Det skulle bli lättare att återupprepa utsökningar av samma slag och effektivisera arbetet.

Förbättringarna vore dock marginella i jämförelse med ansträngningen och kostnaden att implementera dem. Det skulle kräva en betydlig insats att imitera de verktyg som redan används av utvecklarna idag. Den enda fördelen vore sparade sökprofiler, men skulle testaren

behöva genomföra samma utsökningar återupprepade gånger kan man med enkla handgrepp skriva ett skript som utför samma operationer. Utseendet på loggpresentationen skulle inte skilja sig nämnvärt från de verktyg som redan används, dvs. enkla fönster med text.

Genom att lämna loggfilerna utanför gränssnittsapplikationen skulle ingen avsevärd användbarhet gå förlorad. Den hantering som användarna utför manuellt med hjälp av andra verktyg fungerar bra och erbjuder all funktionalitet som användarna kräver och behöver. Man skulle inte behöva lära sig nya sökrutiner i GUI utan kan använda UNIX-skalet och editorerna som man redan är familiär med.

(Riktlinje: Inkludera endast features som ökar användbarheten hos systemet)

Frågor relaterade till stöd för testning

För att anpassa gränssnittet efter de förhållanden som råder under testning måste man veta vilken information som skall visas samt vilka operationer som skall kunna utföras. För att verktyget skall vara adekvat krävs det att så gott som ”alla” tänkbara och möjliga testscenarion går att genomföra, via interaktivitet både *offline* och *online*.

Den mest uppenbara delen av gränssnittet är uppstartsmomentet, start av den testade mjukvaran. Det är en obligatorisk del av GUI att kunna starta ASL och stoppa det. Man bör kunna definiera var mjukvaran är installerad och kontrollera alla processer som startas upp så att man senare kan terminera samtliga av dessa och avsluta testet. Hur detta lämpligast implementeras i GUI diskuteras i kapitel 5.3.

Hur presenteras noden?

Tidigare beskrevs de olika skript och filer som kan manipuleras för att konfigurera ASL. Med undantag från en sak, definitionen av kommunikationsport, berör de utseendet på den nod som DPE skall köra på. En riktig fysisk nod består av olika delar av hårdvara som är monterade på varandra efter ett hierarkisk mönster (se sid 50). De hierarkiska nivåerna och placeringen av en hårdvaruenhet bestämmer dess position, vilket är den variabel som DPE använder för att hålla reda på hårdvaran. Konfigureringsfilerna i ASL definierar endast processorenheterna (PM), den yttersta nivån i nodhierarkin, och den position dessa finns på. De andra nivåerna är egentligen ointressanta.

För att visualisera noden och ge möjlighet att manipulera denna krävs det alltså bara en enkel lista på dessa PM och dess position. Användarna skulle enkelt kunna skapa en ny PM och definiera dess position. Vilket innebär en enkel och överskådlig lösning, givetvis med kontroller så att användaren håller sig till rätt syntax och undviker dubbla positionsnummer. Skillnaden mellan en enkel PM-lista och konfigurationsfilerna blir då liten, men man slipper att tänka på syntax och stavningsfel samt att man enbart behöver ändra en gång i ett GUI, istället för flera i olika filer.

En annan lösning som skulle ge samma positiva effekter fast med en pedagogisk dimension, vore att schematiskt visualisera noden med alla dess hårdvarunivåer. Antingen genom att imitera utseendet på en verklig nod eller använda de avbildningar som finns i Ericssons eget informationsmaterial. Sådan presentation av noden kräver dock en avancerad grafisk implementering, man får designa egna grafiska komponenter vilket kan vara tidskrävande. En enklare variant vore att använda färdiga standardklasser som följer med de flesta

programmeringsspråk. En abstrakt strukturell översikt genom exempelvis en filträdstruktur, vore tillräcklig. Användarna är vana vid de flesta grafiska standardkomponenter som erbjuds i programspråkspaketen och är vana vid abstrakta avbildningar av verkligheten.

All information som är tillgänglig om en komponent skall visas om det finns utrymme, annars bara det mest väsentliga. Om informationsmängden är ansenlig och oöversiktlig kan man gömma viss information som endast presenteras om användaren uttrycligen önskar det. Genom menyval, inställningar eller genom ”dubbelklick”: Man klickar på ett objekt och ytterligare information visas om detta.

(Riktlinje: Spegla verkligheten men premiera enkelhet och snabbhet före exakt avbildning)

Hur konfigureras noden?

Tester skall vara så enkla att upprepa som möjligt. Därför är det viktigt att det är lätt att konfigurera noden, som är den viktigaste och mest komplicerade operationen innan exekvering. Bra översikt över noden för snabba justeringar. När noden manipuleras skall man med enkla handgrepp, tex knapptryck, kunna ta bort eller lägga till komponenter till de olika hårdvaruenheterna. Korttyp skall väljas på varje specifik hårdvarunivå. När detta är klart och det är dags att starta sitt test uppdateras skript och konfigfiler.

(Riktlinje: Underlätta återupprepning av tester.)

Det finns även önskemål hos användarna att kunna definiera nya typer av kort, förutom de befintliga korttyper som används idag. Genom att använda implementationen av nodkonfigureringen som mall, går det att enkelt lägga till en sådan kortdefinierande funktionalitet, med några mindre justeringar.

För att göra testningen mer ”realistisk” och adekvat borde man kunna välja att placera olika delar av noden på andra processorer i nätverket. Det kan genomföras via inställningar där man definierar de arbetstationer i nätverket som man önskar fjärrlogga in på. Sedan kan man välja vilken dator (eng. *host*) som ett kort skall köra på (ex: PM vars motsvarande EQMA-process exekverar på vald processor). En orsak till att man vill köra på flera datorer samtidigt är möjligheten att starta upp DPE:s kontrollprocess, NCL, på ytterligare datorer. Då initieras funktionalitet för val av aktiv NCL samt backup NCL.

(Anmärkning: Tyvärr fanns det inte tid till att implementera ”nätverksalternativet vilket vi insåg redan i analys och designfasen, så vi tar inte hänsyn till detta i resten av arbetet.)

(Riktlinje: Se till att alla möjliga testfall är genomförbara)

Samtidigt som man vill visa hela nodstrukturen kan det vara viktigt att göra en avgränsning mellan det mest väsentliga för ASL, nämligen PM:n och resten av noden.

Hur hanteras processerna?

Varje PM som startas upp representeras av en EQMA-process, vilket skiljer PM:n från den andra hårdvaran i noden som bara finns logiskt representerat via positionsnummer. Det är viktigt att distingera mellan den logiska strukturen hos noden och de egentliga processerna som exekverar i systemet. Användarna känner till båda världarna, både verkligheten för ASL

och hur en ”verklig” nod fungerar. En tydlig skillnad underlättar för användaren, så att denne får en god överblick av de distribuerade processerna.

DPE startar upp applikationer som kör applikationsblock-processer på de olika PM:n. Förutom den vanliga datan om ett oaktiverat PM vill man känna till vilken processor den kör på, processid och andra processrelaterade data som operativsystemet erbjuder. När det gäller blockprocesserna så vill man veta vilken applikation de tillhör, var den är utdistribuerad (PM-position) samt processid m.m. PM och applikationsprocesser bör vara skiljda åt för att inte blandas ihop, och blockprocesserna bör sorteras efter applikation. Det finns ytterligare processer som initieras av DPE, framförallt NCL-processen. Den bör presenteras skild från resten av komponenterna i GUI på ett sätt som understryker dess vikt. Information om host, aktivitetsstatus, PM-position samt andra processrelaterade data bör redovisas.

Som tidigare nämnts är testning av pålitlighet av stor vikt. Genom att simulera fel, testar man hur systemet hanterar återhämtning och omdistribution av processer. Felen simuleras genom att ”döda” eller terminera processer. Detta bör således kunna utföras med enkelhet i GUI. Både enskilda processer samt flera processer tillsammans skall vara möjliga att terminera. Man bör också kunna terminera NCL, för att testa om NCL-backupen tar över. När en PM är inaktiv skall den kunna startas, även när resten av systemet är igång. Vilket innebär att man kan lägga till nya kort/PM online och återstarta terminerade PM/EQMA-processer.

Önskvärt vore också att kunna använda den funktionalitet som är inbyggd i den riktiga hårdvarunoden, som blockering av ett kort via *repair handling button*. Det är en knapp som finns på Craneboards och meddelar NCL att kortet skall blockeras, alla applikationer som kör block på detta kort omdistribuerar dem och till slut kan man ta bort kortet. D.v.s. en kontrollerad avstängning. Problemet med att implementera den typen av funktionalitet är att man måste kommunicera med DPE, vilket kräver att man utvecklar för DPE. Funktionaliteten borde dock finnas men kommer inte implementeras i denna fallstudie. Nu nöjer vi oss med att endast terminera processer.

(Riktlinje: Se till att alla möjliga testfall är genomförbara)

När konfigureringen är klar och testet startat, uppdateras och initieras olika skript. Dessa initierar i sin tur olika processer som NCL och EQMA. För att användarna skall vara medvetna om att uppdateringar utförts och mjukvaran startat bör gränssnittet påvisa skillnaden i tillstånd. Det måste vara skillnad på start och stopp. Processerna bör ge en bild av dynamik, faktumet att de exekverar skall vara uppenbart. Information om processernas *state* skall vara tillgänglig. Syftet med att skilja på tillstånd är att det gör det lättare för användaren veta vad den gör och förstå mekanismen och flödet hos gränssnittet.

(Riktlinje: Förtydliga olika ”tillstånd” hos systemet)

Feedback om uppdateringarna och andra förändringar i det bakomliggande systemet (ASL) bör redovisas utförligt för varje specifik operation som genomförs. Information om den bakomliggande funktionaliteten är väsentlig för användarna då de känner till ASL. De kan ha gjort specifika och av gränssnittet oförutsedda förändringar manuellt. Om användaren vet vad som påverkas av gränssnittet kan de undvika eller hitta fel och buggar som annars inte gått att förutsäga eller förstå.

(Riktlinje: Ge användaren kontroll över systemet)

5.3 Avbildning

I detta avsnitt kommer vi mer konkret koppla Alan Coopers idéer om gränssnittsdesign till vårt aktuella fall. Vi diskuterar de taktiska och strategiska verktyg (se teori del 3.1) som vi har till hands. För att på ett generellt plan bestämma vilka gränssnittsidiom vi ska använda och hur dessa interagerar med användaren.

Användarens mål

Generellt sett har programmets syfte varit relativt enkelt att urskilja. Minimikravet är att man genom ett grafiskt gränssnitt ska kunna utföra följande två fundamentala handlingar:

1. Konfigurera en nod
2. Starta DPE

Förutom dessa punkter har vi även valt att tillgodose följande användarcentrerade önskemål:

3. Visa applikationsdistributionen när DPE kör
4. Att inaktivera och aktivera PM:s
5. Visa var NCL körs
6. Inaktivera NCL
7. Skapa nya kort

Målen har identifierats och konkretiserats genom en löpande dialog med användarna. Och dessa mål är som vi påvisat tidigare inte de enda önskvärda utan endast de som vi valt att genomföra. Punkterna beskriver vad programmet ska göra, de säger dock ingenting om hur programmet bör se ut eller hur det ska kommunicera med användarna. För att komma fram till detta har vi som designers eftersträvat att agera som ”användarnas advokater.” Detta innebär att vi för deras talan men inte bara företräder dem utan även försöker upplysa dem om vad som är bäst. Vi skapar något som grundar sig på vad användarna vill ha men tillför också något eget. Det är häri användarnas mål går att finna. I konkretiseringen av alla de bilder som användarna förmedlar blandat med en strukturell och logisk uppbyggnad som vi i ett ständigt utbyte förmedlar tillbaka. Denna process pågår tills dess att användarna känner att deras mål är uppfyllda och att kommunikationen med programmet sker på deras villkor. Följande punkter är exempel på hur användarna själva kan uttrycka hur deras mål med programmets GUI ser ut:

- *Modulärt, lättanvänt, funktionellt och utbyggbart.*
- *Enkelhet och snabbhet och översiktlighet bör prioriteras.*

De bilder som användarna har förmedlat som beskriver hur de anser att användargränssnittet ska se ut har givetvis inte varit entydiga. Vår design är som de flesta andra en kompromisslösning där allas önskemål inte kan bli till hundra procent uppfyllda. Det är denna osäkerhet som är kärnan i vår hypotesprövning. Att finna någon slags verifikation på att det vi designar uppfyller vissa kvalitetskrav och därmed kan betecknas som framgångsrikt.

Form

Formen är den helhetslösning som ska spegla användarnas verklighetsbild och genom ett GUI uppfylla och gestalta denna bild. Vår utgångspunkt är att skapa ett så enkelt och lättanvänt GUI som möjligt. För att åstadkomma detta måste vi utgå från användarnas verkliga miljö och arbetssätt - användarnas mentala modell - och renodla denna för att nå fram till användarnas mål.

Som vi tidigare sett prioriterar användarna kontroll och snabbhet. Kontroll i gränssnitts-sammanhang handlar om översiktlighet och att kunna utföra rätt saker vid rätt tillfälle. Men samtidigt som översiktlighet ska prioriteras ska programmet också vara så litet som möjligt. Båda dessa motsägande faktorer måste uppfyllas jämte en rad andra för att gränssnittet ska bli uppskattat. Ett generellt exempel på hur användarna resonerar påvisas i följande citat:

- *Gränssnittet ska vara flexibelt, vara både enkelt och avancerat. Viktigast av allt är att programmeraren har kontroll över vilka processer som är igång. Vill kunna göra snabba övergripande konfigurationsändringar.*

Vi tror att dessa mål kan uppfyllas genom att skapa en relativt abstrakt och enkel bild av användarnas miljö. Vi behöver inte konstruera några nya avancerade grafiska avbildningar utan kan använda oss av de gizmos som normalt finns i programbiblioteken. Det väsentliga är dock vilka delar vi väljer att sätta samman till en helhet och hur väl denna helhet uppfyller användarnas mål.

Följande konkreta bilder är exempel på vad det grafiska användargränssnittet på ett mer eller mindre abstrakt vis ska visualisera.

- En nods struktur/konfigurering
- Applikationers distribuering under körning
- Att DPE kör respektive inte kör
- Att NCL är aktiv och på vilken position
- Vilken status ett PM/applikationsblock har

Att tänka idiomatiskt

Det idiomatiska paradigmet baseras på lärande. Det handlar om hur något kan designas för att korta inlärningsfrekvensen och få program att verka mer intuitiva. De flesta elementen i ett GUI består av idiom. De har från början lärts in och med tiden upphöjts till standard för att idag uppfattas som fullständigt naturliga, närmast metaforiska. Användarnas mål uppfylls inte nödvändigtvis genom att den miljö de arbetar i avbildas exakt. Vid visualisering av kända processer och objekt premierar vi i första hand inlärningsfrekvens och snabbhet, i andra hand en exakt avbildning av användarnas verklighet. Vi strävar efter att skapa ett ”osynligt” GUI som utför sitt jobb så snabbt och enkelt som möjligt. Konkret får detta till följd att vi beaktar den flora av bilder som användarna omges av och om dessa bilder är förenliga med de krav vi ställer på programmet använder vi oss av dem. Om inte väljer vi en lösning som i högre grad överensstämmer med användarnas huvudmål - snabbhet, enkelhet och översiktlighet. Användarna har under analysfasen uttryckt sig på följande sätt om hur de anser att deras verklighet bör avbildas:

- *Bra om ni följer de bilder som redan är etablerade internt på Ericsson vid visualisering av exempelvis magasin, kort, PM etc.*

- *Allt ska vara så enkelt som möjligt. Vill bara se det nödvändiga, till exempel endast de PM som är definierade, vilken position dessa har och då i formen av 1.2.1.3. Hellre en abstrakt bild än en bild av hur hårdvaran ser ut.*

Dessa båda citat kan tyckas vara varandras motsatser men i ett idiomatiskt perspektiv är de egentligen uttryck för samma sak. Båda användarna anser att programmet ska vara lätt att lära sig. För att komma fram till vilka idiom som ska användas har vi provat oss fram. Genom att grafiskt rita upp strukturer, symboler och begrepp på papper och föra en diskussion med användarna har dessa successivt verifierats och accepterats. Det centrala vid val av idiom är att dessa är väldesignade och enkla att lära sig. Om de även är väl vedertagna är detta givetvis en bonus. Det faktum att användarna är expertanvändare och vana vid att arbeta med ett kommandobaserat gränssnitt innebär dessutom att de förmodligen accepterar en högre grad av visuell abstraktvitet. Det innebär att de därigenom kanske inte vid en första anblick värdesätter de kvaliteter som ett väldesignat idiomatisk uppbyggt GUI kan förmedla. Men denna tveksamhet måste överbryggas genom att man finner den bästa lösningen och inte nöjer sig med att avbilda det självklara. Vår utgångspunkt är att även tillföra ett idiomatiskt flöde som ytterligare förbättrar användarnas mål.

(Riktlinje: Spegla verkligheten men premiera enkelhet och snabbhet före exakt avbildning)

Fönsterhantering

Vår design kommer att sträva efter att minimera antalet fönster då allt för många av dessa stör flödet och därmed försämrar arbetsprocessen. Användarna är samstämmiga i att programmet ska vara litet och lättanvänt och hela deras arbetsmiljö signalerar om att vi inte kan slösa med skärmutrymme. För att lösa detta men samtidigt trygga informationsbehovet väljer vi att skapa ett litet *parasitic-posture*-program bestående av ett fönster med flikar. Programmet ska befinna sig på skärmen under en längre tidsperiod och ta så lite arbetsyta i anspråk som möjligt. För att inte störa användarens pågående arbetsprocess med andra program bör hela designen kännetecknas av ett minimalistiskt uttryck både i fönsterstorlek och formuttryck

(Riktlinje: Tillmötesgå användarens utrymmesbehov)

Dokumenthantering

Användaren ska kunna spara och öppna en konfiguration av noden. Denna process ska kopplas till användarnas mentala bild av vad det är de arbetar med. Primärt är detta en konfiguration, sekundärt en fil. Spara- och öppna-förfarandena följer en väl etablerad standard vilken vi inte ser någon orsak till att avvika ifrån. Vi kompletterar dock med att erbjuda en ”recentfiles-funktion” varigenom användaren snabbt kan komma åt sparade konfigurationer. Genom denna funktion införlivas positions - och identitetsåterfinnandet i programmet.

Meny

Menyns innehåll ska trygga användarnas funktionsbehov, allt som programmet kan dirigeras till att utföra ska kunna styras härifrån. De ord som används för att beskriva denna funktionalitet ska vara väl förankrade i användarens mentala modell och därigenom kännas familjära och användbara. Detta får bland annat till följd att vi inte per automatik använder oss av den etablerade menystrukturen utan strävar efter att så långt det är möjligt gestalta användarnas bild av vad de ska utföra. Vi har redan nu beslutat att inte använda oss av en ”File”-meny utan istället kalla denna meny ”Configuration” då det är konfigurationer som användarna arbetar med. Detta helt i linje med att gestalta användarnas mentala modell och stäva efter att ignorera implementeringsmodellen. Kopplingen mellan menyalternativen och

den verklighet som beskrivs genom dessa ska överensstämma och menystrukturen ska dessutom byggas kring tesen att allt ska gå att nå så enkelt som möjligt. Åtkomligheten kan lösas genom att menyerna grupperas efter verksamhetsområde och vikt. För att åstadkomma en intuitiv känsla bör undermenyalternativen återanknyta till huvudmenyerna genom formen ”handling-objekt” vilket i ”Configuration”-menyn skulle bli ”save-configuration”.

Kortkommandon bör implementeras i så hög omfattning som möjligt och om standarder finns ska dessa följas. Användarna är vana vid att använda kortkommandon och dessa ska överensstämma med UNIX-miljön. I de fall där vår funktionalitet är unik skapas nya kortkommandon i samråd med användarna.

(Riktlinje: Ge användaren kontroll över systemet)

Avbildning av objekt och processer

Ett grafiskt användargränssnitt ska designas så att användarnas mål uppfylls på bästa sätt. Detta kan utföras genom att grafiskt exakt efterlikna den verklighet man vill beskriva eller genom att skapa en abstraktion. Gizmos är ett samlingsnamn för alla de objekt som ett GUI normalt består av. De kan vara standardkomponenter hämtade ur standardbibliotek eller mer eller mindre egenkonstruerade grafiska objekt.

Avbildning av processer

Imperativa gizmos kommer i vårt program främst att gestaltas genom knappar. Vi ämnar följa den väl etablerade standarden att ett program ska ha en verktygsfält med ett antal knappar genom vilka funktionalitet kan utlösas. Knapparnas innerbörd ska visualiseras genom bilder och förtydligas med vidhäftande verktygstips. De knappar med ikoner så kallade *buttcons* som vi skapar ska främst vara enkla att lära sig. Även om det är en fördel att bilderna överensstämmer med användarens metaforiska bilder och därmed känns intuitiva, är det lärandet, det idiomatiska paradigmet, som vi kommer att arbeta utifrån.

Alla *buttcons* i programmets verktygsfält styr processer och är därmed verb. Ett verb är svårare att gestalta än ett substantiv om man gör anspråk på att exakt avbilda verkligheten. Vi har dock inga sådana anspråk, vårt arbete strävar efter att användarnas mål ska uppnås. Därmed beror kvaliteten på de *buttcons* vi skapar på hur enkla de är att lära sig. Vi har valt symboler som känns fundamentala och grundläggande, närmast matematiska. Detta sammankopplat med de minimalistiska anspråk som hela det formmässiga uttrycket ska lyda under gör att de symboler som ska gestalta programmets handlingar kommer att var enkla och vedertagna. De symboler vi valt att använda är följande:

- Öppna en konfiguration - Standardsymbol
- Spara en konfiguration - Standardsymbol
- Lägga till något i konfigurationen - Plus
- Ta bort något ur konfigurationen - Minus
- Starta DPE - Play
- Pausa DPE - Pause
- Stoppa DPE - Stop
- Inaktivera PM eller applikationsblock - Pil ned
- Aktivera PM eller applikationsblock - Pil upp

Alla dessa symboler är väl vedertagna och är därmed enkla att motivera, motiveringen är som följer:

- Öppna- och spararsymbolerna används som standard i de flesta program
- Plus och minus är fundamentala matematiska begrepp
- Play, pause och stop återspeglar den välkända bandspelarmetaforen
- Pil upp och ned fångar en vedertagen betydelse och innebörd som går att överföra på att ned = ta bort (sänka) och upp = återinföra (höja).

(Riktlinje: Spegla verkligheten men premiera enkelhet och snabbhet före exakt avbildning)

Avbildning av objekt

En nods struktur är hierarkiskt uppbyggd. De översta komponenterna består av magasin, följt av craneboards, subboards och slutligen PM:s. Om detta ska gestaltas i ett GUI är det mest naturliga att använda sig av en trädstruktur. Ett träd tar lite plats i förhållande till hur mycket information som kan visas och lämpar sig därmed väl för våra ändamål. För att användarna ska känna sig hemma i trädstukturen kan denna förtydligas med ikoner som avbildar de verkliga objekten. En sådan konstruktion är idiomatiskt förankrad då trädstrukturen är enkel att läras sig och väl vedertagen. Den uppfyller också kraven på att minimera programmets arbetsyta. En trädstruktur kan kompletteras med listor vars innehåll uppdateras beroende på vad som väljs i trädet. Komponenterna magasin, craneboard och subboard fyller ingen egentlig funktion i arbetsstationslösningen mer är att hysa PM:s. Att ändå ha med dessa i gränssnittet tjänar bara syftet att avbilda en verklig nod mer korrekt. Genom att avbilda en nods hierarkiska struktur blir dels konfigureringen enklare att lära sig, dels blir varje PM positionsbestämd på ett naturligt och enkelt sätt. Egentligen skulle syftet med programmet uppnås genom att bara erbjuda en lista med positionsbestämda PM:s. Dessa skulle kunna tas bort och läggas till precis som i ett träd. Fördelen med en abstrakt bild är att man slipper ta hänsyn till avbildningsparadigmet och kan skapa något som är så effektivt och snabbt som möjligt. Men vid avbildningen av en nod går det förmodligen snabbare att genomföra en konfiguration i en trädstruktur än i en lista. Ett PM hamnar då på en naturlig trädposition som överensstämmer med verkligheten och användaren slipper mata in denna position för hand vilket förmodligen skulle bli fallet i en lista.

Att avbilda noden genom ett träd är korrekt utifrån ett logiskt hierarkiskt synsätt men inte om man önskar se hur noden visuellt ser ut i verkligheten. Trädstrukturen i sig är en abstraktion av den verkliga noden och vill man avbilda denna mer verklighetsnära krävs en mer avancerad grafisk lösning. Detta skulle innebära att programmets prestanda försämrades då mer grafik innebär ett långsammare, större, mer skärmutrymmeskrävande och rörigt program. Vilket motsäger alla de grundpremisser som vårt GUI ska bygga på. Ett GUI som exakt avbildar ett av fem nivåer uppbyggt hierarkiskt system skulle förmodligen bli ohanterbart, vi väljer därmed att inte skapa några nya grafiska lösningar för att hantera informationsbehovet. Vi använder oss istället av en kombination av traditionella gizmos som träd, listor och tabeller.

Vid konfigurering av noden vill man kunna lägga till och ta bort magasin, craneboards, subboards och PM:s. - Detta tryggas bäst genom att skapa en trädstruktur för de tre högsta nivåerna kopplad till en lista som visar alla PM. När noden körs ska även de applikationer som är kopplade till alla PM:s visas. Ett PM har dessutom egenskaper och attribut som ska visas under körning - detta tryggas genom att listan med PM byts ut mot en tabell som visar körningsattributen och ytterligare en tabell för applikationer med tillhörande attribut kopplade till rätt PM.

(Riktlinje: Spegla verkligheten men premiera enkelhet och snabbhet före exakt avbildning)
(Riktlinje: Tillmötesgå användarens utrymmesbehov)

Beteende och tillstånd

Programdesign bör sträva efter att minimera tillstånd som inte direkt hjälper till att lösa användarnas problem. Vid utvecklandet av ett GUI blir detta extra tydligt då det är gränssnittet som till stor del dikterar hur programmet ska användas. En grundläggande utgångspunkt är att designa ett GUI som låter användaren lösa sina uppgifter utan att blanda sig i och förhindra detta arbete. Idealfallet vore ett osynligt GUI som inte stör men som hela tiden läser av användarens beteende och agerar exakt vid rätt tillfälle, ett GUI som motiverar och stödjer användarens arbetsflöde och som dessutom utstrålar och genererar trovärdighet.

Att designa program för expertanvändare ställer höga krav på programmets beteende och tillstånd. Expertanvändare ställer andra krav på programmets feedback än normalanvändare och om vi även lägger till det faktum att programmets beteende är av typen *parasitic-posture* så bör beteendet inte störa den pågående arbetsprocessen hos andra program. Hela utvecklingsprocessen ska sträva åt att skapa ett GUI som gör användarna mer produktiva och detta utan att de själva egentligen är medvetna om det.

Flöde

Ett programs flöde är ett mått på hur nöjda användare är med hur programmet utför sina uppgifter. Ett positivt flöde innebär att uppgifterna utförs på ett naturligt och följsamt sätt utan att användaren egentligen reflekterar över hur det gick till. Ett negativt flöde däremot innebär förekomster av mer eller mindre störande moment. Det är svårt att på förhand se vilka moment i ett programs beteende som efter ett repetitivt nyttjande kommer att uppfattas som störande. Men chansen för att sådana moment överhuvudtaget ska förekomma kan minimeras genom en god design. I teoriavsnittet gick vi igenom Alan Coopers fyra punkter som ett lyckat ”osynligt” GUI bör uppfylla. Dessa var:

1. Följ mentala modeller
2. Dirigera istället för att diskutera
3. Ge hela tiden användaren tillgång till relevanta verktyg
4. Ge feedback, men i neutral och saklig ton och utan att störa

Dessa fyra punkter är centrala och kommer under utvecklingsarbetet att fungera som övergripande riktlinjer för hur vi anser att ett GUI bör fungera. Nedan följer en konkretisering av varje punkt.

Följ mentala modeller

Ett program blir enklare att arbeta med om användaren känner igen sig i miljön. Ett GUI bör återspegla användarens individuella tolkning som uttrycks i en mental bild av dataprocessen. Detta gäller för alla programmets delar. Såväl för delar som buttcons, typsnitt och färgval etc. som för helheten vilken ska vara väl sammansatt och koordineras så att alla element arbetar mot samma mål. I vårt fall handlar det, som vi tidigare diskuterat, om att tänka idiomatiskt och utifrån den specifika miljön konstruera ett GUI som användarna känner sig hemma i. Utgångspunkten för hela utvecklingsarbetet är att öka användarnas produktivitet. Att skapa en ny miljö som utför vissa arbetsmoment snabbare och enklare men utan att förlora möjligheten

att arbeta som man gjorde förut. Den nya miljön ska inte ersätta den gamla utan fungera i symbios med denna och bara ta över vissa arbetsmoment. Det grafiska gränssnittet måste spegla detta faktum - att programmet är löst kopplat till befintlig data och funktionalitet - för att användarna ska lita på det. Vilket förutom att bygga en miljö som är intuitiv och harmonisk även ställer krav på vilken och hur information presenteras i gränssnittet. Vid sidan av att avbilda processer och objekt utifrån användarnas konceptuella modeller måste allt även sättas samman så att arbetsprocessen upplevs som naturlig. Användarna ska känna att det är de som styr programmet, inte tvärtom. Detta kan åstadkommas genom att arbeta utifrån ett minimalistiskt perspektiv och successivt plocka bort detaljer som inte direkt används för att lösa uppgiften. Ju mindre grafisk utsmyckning och krånglighet som byggs in i ett GUI desto enklare och smidigare kommer interaktionen med användarna fungera. Målet är att skapa ett GUI av typen *parasitic-posture* som egentligen inte syns utan bara utför sin uppgift på enklast möjliga sätt.

(Riktlinje: Betona det naturliga flödet)

Dirigera istället för att diskutera

Normalförfarandet i många program är att när något går fel påvisa detta genom att låsa systemet med en dialogruta. Dialogrutan ska ge användarna kännedom om vad som gått fel och för att bli av med den måste de trycka på ”OK”-knappen. All användning av dialogrutor hämmar programmets flöde. Det förmedlar en uppfattning om att programmet vill diskutera något som antingen är självklart och redan beslutat eller omöjligt att göra något åt. De användare vi vänder oss till vill diskutera så lite som möjligt med programmet men samtidigt ha mycket information om vad som händer. Programmet ska säga till när något går fel men inte genom en tidsödande tvåvägskommunikation utan snarare genom en exakt och korrekt dirigering som erbjuder en snabb lösning.

Ett sätt att hindra användarna från att utföra felaktiga moment är att låsa de interaktions-möjligheter som för tillfället inte ska gå att nyttja. Detta kan göras i menyer och i verktygsfält och innebär att vissa alternativ ”mörkas” för att symbolisera att de för tillfället inte är valbara. Effekten blir att användarna besparas en mängd valbara alternativ som om de inte ”mörkades” skulle generera onödiga felmeddelanden eller direkta fel.

För att trygga det informationsbehov som användarna har men samtidigt inte hämma flödet kommer vi att minimera användandet av dialogrutor och istället bygga in feedbacken direkt i det normala interfacet. Att upplåta utrymme i huvudfönstrets nederkant för att förmedla programspecifik information har nuförtiden närmast blivit en standard. Men vi tänker inte bara använda detta utrymme till att enbart visa löpande information utan för all typ av information. Inom UNIX-världen är detta inget märkvärdigt, programmet emacs använder exempelvis aldrig dialogrutor för att förmedla information. För att ytterligare förstärka känslan av att veta vad som försiggår i programmet kan systeminformationen loggas så att möjlighet finns att gå tillbaka och se vad som hände vid ett specifikt tillfälle. En möjlig gränssnittslösning vore att skapa en lista med ”scrollfunktion” placerad i programmets nedkant vilken kan förstoras eller förminskas efter behov. En sådan lösning skulle utan att hindra flödet visa systeminformation, vara förenlig med att arbetsytan ska vara så liten som möjligt och även bevara en historia över vad som har hänt i programmet.

Användarna ska bara behöva hantera andra fönster än huvudfönstret vid två tillfällen:

1. Lägga till ett nytt kort

2. Ställa in programmets inställningar

Situationer som dessa är väl avgränsade och kräver mer skärmutrymme än vad som ryms i huvudfönstret. De är genom sina specifika uppgifter utmärkta att placera i dialogfönster som nås genom exempelvis menyn. Det utmärkande för dessa fönster är att de ska vara självförsörjande och inte vara beroende av huvudfönstret. All information som behövs för att utföra fönstrets uppgifter och all feedback ska placeras inom fönstret, vilket får till följd att även dessa fönster förses med någon form av informationsfält.

(Riktlinje: Passivt stöd, dirigera istället för att diskutera)

Ge hela tiden användaren tillgång till relevanta verktyg

Den ideala interaktionen mellan användare och program är användandet av ett verktyg. Men en alltför rik flora av detaljer och funktionalitet kommer att upplevas som ohanterbart om alternativen presenteras samtidigt och utan rangordning. Vår intention är att hela tiden ge användaren tillgång till relevanta verktyg. Detta kräver kännedom om alla de interaktionsmöjligheter som användaren kräver i alla specifika situationer. Vi ämnar implementera kortkommandon som överensstämmer med UNIX-standarden och ge åtkomst till relevant funktionalitet i ett verktygsfält. Programmets samlade funktonalitetsutbud finns givetvis samlat i huvudmenyn men dessutom ska popupmenyer designas för varje manipulerbart objekt. Dessa menyer uppkommer vid höger-musklick och ska innehålla alla handlingar som det markerade objektet kan utsättas för.

De interaktionsmöjligheter som bjuds användarna i form av muskaraktäristika, kortkommandon och markeringsmöjligheter ska så långt det är möjligt följa UNIX-standarden. Expertanvändare vill ha valmöjligheter i interaktionen med programmet, de är sedan tidigare väl förtrogna med ett kommandobaserat gränssnitt där man hela tiden har tillgång till allt. Den enda begränsande faktorn är användarens kunskap. Denna känsla av att kunna styra och utföra vad som helst, när som helst ska i möjligaste mån överföras i det GUI som konstrueras.

(Riktlinje: Ge hela tiden användaren tillgång till relevanta verktyg)

Ge feedback, men i neutral och saklig ton och utan att störa

Expertanvändare vill ha kontroll över vad som försiggår vilket ställer höga krav på systemets feedback. Denna bör vara utförlig och exakt och tala om vad som försiggår eller vad som har inträffat. Om det finns en direkt lösning på det inträffade bör även detta presenteras. I vårt fall resulterar detta i att feedback visas i en scrollbar lista i programmets nederkant. Intentionen är att ge så mycket feedback som möjligt men att formulera denna sakligt och neutralt.

Programmets tillstånd kan förtydligas genom att grafiska detaljer i gränssnittet förändras. Helheten ska kännetecknas av ett minimalistiskt uttryck och de detaljer som byter skepnad för att påvisa ett tillstånd bör även de vara grafiskt återhållsamma. De tillstånd som ett GUI i vårt fall kan förtydliga är följande:

- När DPE sätts igång byts fönster automatiskt och alla processer startas unisont. PLAY-knappen hålls nedtryckt tills dessa att DPE stoppas.
- När PAUSE aktiveras hålls dess knapp nedtryckt till dess att stop eller pause aktiveras precis som på en bandspelare.

- Visa vilken status NCL har genom att ändra färg på dess symbol och förtydliga denna information med vidhäftande text.
- Visa tillstånd hos PM och applikationer genom att ändra värden i respektive listor.

Att förtydliga systemets tillstånd genom gränssnittet förenklar användbarheten avsevärt. Med relativt små grafiska medel kan användaren informeras om viktiga händelser och tillstånd och på så vis utföra sitt arbete mer effektivt. Vår ansats är att förtydliga centrala tillstånd hos systemet, som hur NCL mår och om DPE kör eller inte, och utföra detta så att det märks men inte irriterar. Den grafiska visionen ska utmärkas av ett ordnat och enkelt uttryck.

(Riktlinje: Förtydliga olika ”tillstånd” hos systemet)

5.4 Härledda förutsägelser

Sammanfattningvis kan man dra vissa slutsatser utifrån diskussionen så här långt. Vi urskiljer att användaren enkelt vill kunna återupprepa tester på ett diversifierat sätt. Det är avgörande att gränssnittet inledningsvis är så övertygande och attraktivt, så att användarna överger (åtminstone delvis) sina tidigare rutiner och börjar nyttja gränssnittsapplikationen. Samtidigt förutsätter detta att funktionaliteten, det vill säga verktyg för testning, är adekvat.

Utformningen av det grafiska gränssnittet bör således vara så enkel och avskalad som möjligt, samtidigt som den måste behålla tillräckligt med valmöjligheter. Detta är ett påstående som är motsägelsefullt, men ändå inte oförenligt utan snarare innebär att fördelarna med enkelhet och mångfald måste vägas mot varandra. Vidare måste det grafiska användargränssnittet vara attraktivt, tillgängligt och intuitivt så att användarna tar till sig applikationen så snabbt som möjligt. Resultatet skall förbättra användbarheten (usability) hos testningsverktyget på ett flertal sätt.

Användarnas roll i formandet av vår hypotes har mest handlat om att ge oss insikt i och förståelse för deras kunskapsnivå och arbetssituation. Detta snarare än att ge oss konkreta förslag om utseende och funktionalitet.

Nedan följer de riktlinjer som vi urskiljer som avgörande för att utforma gränsnitt anpassade för testning. De är i några fall specifika för fallstudien, men de flesta går också att generalisera så de blir relevanta för gränssnittsdesign i ett bredare perspektiv. Vi formulerar dessa punktvis med utgångspunkten. - Hur man utformar ett grafiskt gränssnitt för testning av inbäddade system och hur detta förbättrar användbarheten. Dessa förutsägelser utgör våra hypoteser och det är dessa vi sedan kommer att testa och eventuellt revidera.

- ***Ge användaren kontroll över systemet***
  Teknisk information om den bakomliggande funktionaliteten (ASL) bör presenteras för användarna om de är familjära med denna och behöver kontroll över systemet. Användare som gör ovanliga manuella konfigureringar, genom att direkt manipulera information utanför det grafiska gränssnittet, behöver få information om vilken data som ändras och hur och när detta sker i GUI. Om detta uppfylls ökar attraktiviteten att använda gränssnittet för de mest avancerade användarna. Om användarna inte förlorar kontroll kommer de att acceptera gränssnittet snabbare förutsatt att det ger en förbättrad användbarhet i övrigt. Interaktionsmöjligheterna bör dessutom vara intuitivt designade så att användarna enkelt och snabbt kan genomföra det de tänkt sig.
  \+ *attractiveness*

- ***Tillmötesgå användarens utrymmesbehov***
  Genom att minimera arbetsytan för GUI eller åtminstone ge möjlighet att anpassa arbetsytan, kommer användaren kunna arbeta med parallella arbetsuppgifter samtidigt som ett snabbt ögonkast ger en överblick över testfallet. Testningen tar ofta lång tid och kräver inte användarens kontinuerliga uppmärksamhet. Interaktionen med programmet under testningen begränsas också ofta till uppstart och avbrott. En sådan

lösning ger en ökad attraktivitet och underlättar arbetet för användaren om man ser till arbetet som helhet och inte bara till testningen.
+ *attractiveness*
+ *operability*

- ***Anpassa stöd efter användarnas intresse och kunskapsnivå***
Eftersom användarna förväntas vara kunniga och erfarna bör man minimera hjälp och stödverktyg som tutorials och walkthroughs, begränsa stödet till feedback i det aktuella arbetsfönstret och dirigera istället för diskutera. Man bör även fokusera på den feedback som ges så att den håller tillräckligt hög kvalitet och därmed kompenserar avsaknaden av hjälpmanualer etc.

Förutsättningen är ändå att programmet är tillräckligt enkelt och intuitivt så att användarna kan lära sig och förstå programmet utan onödig hjälp. Detta resulterar i att onödiga programenheter skalas bort och gör GUI enkelt och attraktivt, då man slipper irriterande stödfunktioner, men samtidigt bibehåller eller förbättrar användarstödet.
+ *attractiveness*
+ *helpfulness*

- ***Använd enbart befintlig funktionalitet och rationalisera bort oprioriterade moment***
Den funktionalitet som fanns innan GUI-utformningen införlivas i gränssnittet så att detta kan nyttjas på bästa sätt. Även om man finner att det saknas viss typ av funktionalitet och operationer för att optimera övervakningen av mjukvara under test så är det oftast inte meningsfullt att utveckla detta. Det är gränssnittet som skall utformas och inte funktionaliteten som skall designas om. Den ekonomiska och tidsmässiga kostnaden skulle vara för stor och det är absolut nödvändigt att snabbt producera ett billigt och lättillgängligt GUI. Alla detaljer som inte är absolut nödvändiga och som kan förlänga inlärningsperioden bör rationaliseras bort. Bevarad enkelhet underlättar GUI-hanteringen, på bekostnad av en större valmöjlighet.
+ *operability*
\- *customisability*

- ***Inkludera endast features som ökar användbarheten hos systemet.***
De verktyg i det existerande gränssnittet som användarna redan hanterar på ett tillfredställande sätt och som inte går att förbättra nämnvärt, behöver inte inkluderas i GUI. Det stöd för testningen som användarna får från dessa verktyg riskerar snarare att försämras än förbättras. Dessutom tvingar man inte användarna att lära sig nya funktioner som de redan behärskar med ett annat verktyg. Detta bevarar enkelheten hos GUI och ger ingen negativ effekt på hanteringen av testningen.

- ***Spegla verkligheten men premiera enkelhet och snabbhet före exakt avbildning***
Genom att spegla förhållanden i testmiljön och den bakomliggande funktionaliteten, blir det lättare att tillägna sig och förstå de abstraktioner som presenteras i GUI. Genom att använda användarnas egna mentala modeller över testningsmiljön blir gränssnittet mer attraktivt att använda och enklare att förstå. Vi premierar i första hand inlärningsfrekvens och snabbhet, i andra hand en exakt avbildning av användarnas verklighet detta i linje med att tänka idiomatiskt.
+ *attractiveness*
+ *understandability*

- ***Underlätta återupprepning av tester***
  Den data som kan manipuleras och förändras, så att ett test skiljer sig från ett annat, skall finnas tillgänglig för manipulation i GUI. Dessa data ska också presenteras på ett sätt som underlättar konfigurering så att den sker snabbt och felsäkert. Snabb konfigurering innebär att det blir lätt att genomföra ett test. Användarna bör känna igen sig och intuitivt förstå gränssnittsobjekten och hanteringen av dessa. Felsäkerhet innebär att användaren inte behöver göra om sina procedurer. Man undviker syntaxfel genom att automatisera uppdatering av bakomliggande data, samt varnar vid undantag som kan få negativa konsekvenser för testkörningen.
  \+ *operability*

- ***Se till att alla möjliga testfall är genomförbara***
  Liksom i punkten ovan gäller att den data som kan manipuleras och förändras skall finnas tillgänglig för manipulation i GUI.

  Man måste ta hänsyn till alla faktorer och åtgärder som kan påverka mjukvara under test. När testaren definierar sitt testscenario måste alla dessa valmöjligheter finnas tillgängliga. Det gäller förberedande konfigurering av mjukvaran och interaktiv manipulering online, under testets gång. Sådan konfigurering och interaktivitet måste inbegripas i GUI. En viss avgränsning är ändå nödvändig, det gäller framförallt operationer som på grund av ekonomiska och tidsmässiga kostnader inte går att implementera. Dessutom behöver gränssnittet inte ta hänsyn till justeringar som lika enkelt går att utföra manuellt utan GUI.
  \+ *customisability*

- ***Förtydliga olika "tillstånd" hos systemet***
  Skapa en tydlig skillnad mellan offline och online under testningsproceduren. Övervakning av processer och andra dynamiska entiteter skall redovisas dynamiskt, genom dess tillstånd och status. Gör det lättare att förstå gränssnittet, vad som sker och vad som kan göras genom att grafiskt påvisa vad som händer. Om dynamiken speglas så förbättras det övergripande intrycket av gränssnittet.
  \+ *understandability*
  \+ *attractiveness*

- ***Ge hela tiden användaren tillgång till relevanta verktyg***
  Genom att ha kännedom om alla de interaktionsmöjligheter som användaren kräver i alla specifika situationer kan man hela tiden se till att användaren får tillgång till relevanta verktyg. Expertanvändare vill ha valmöjligheter i interaktionen med programmet, de är sedan tidigare förtrogna med ett kommandobaserat gränssnitt där man hela tiden har tillgång till allt. Överför känslan av att kunna utföra vad som helst, när som helst.
  \+ *operability*
  \+ *customisability*

- ***Betona det naturliga flödet***
  Öka produktiviteten genom att få användarna att uppleva att de är i harmoni med sitt arbete. Sträva efter att skapa ett "osynligt" GUI som utför sitt jobb så snabbt och enkelt som möjligt. Användarna ska känna att det är de som styr programmet inte

tvärtom. Designa gränssnittet så att det på ett naturligt sätt speglar de olika arbetsmomentens turordning. All användning av dialogrutor hämmar flödet. Genom att bygga in feedbacken i det normala interfacet blir programmet smidigare att använda och mindre irritationsskapande. Den feedback som ett program genererar när användaren på något sätt gör ett misstag eller inte förstår hur något fungerar, borde utan att raljera reflektera programmets status och om så behövs erbjuda ett botemedel. Informationen bör presenteras i gränssnittet på ett sådant sätt att det inte stör användarens arbetsflöde.
\+ *operability*

- ***Passivt stöd, dirigera istället för att diskutera***
  Meningen är att användaren ska få tillgång till relevant information utan att behöva föra en dialog med programmet. Dirigera användaren till att inte göra fel genom att stänga möjligheten till att interagera med programmet på ett felaktigt sätt. Detta genom att exempelvis låsa de meny- och knappalternativ som för tillfället inte är valbara. Därmed minimeras uppkomsten av felmeddelanden. Överhuvudtaget bör alla designmässiga lösningar i ett GUI konstrueras så att användarens kreativa flöde inte förhindras. Detta kräver att programmet designas med en medvetenhet om vad som kan gå fel och när.
  \+ *operability*
  \+ *attractiveness*

Del 6 Hypotesprövning

Hypotesen prövades genom att vi utvecklade en applikation och ett grafiskt användargränssnitt utifrån de riktlinjer som definierades i kapitel 5.4. Vi använde oss av prototyping när vi tog fram applikationen och förde en kontinuerlig dialog med användarna. Dokumentation om den färdiga prototypen finns att tillgå i Appendix 1: BUNNY DOCUMENTATION och Appendix 2: BUNNY MANUAL.

Vårt program utvärderades sedan genom intervjuer med användarna samt användbarhetstest för att mäta kvaliteten på gränssnittet, användbarheten. Denna utvärdering och mätning utgör grunden för att undersöka validiteten hos de påståenden vi gjort i hypotesen. Det vill säga, stämmer det att användbarheten förbättras i enlighet med riktlinjerna.

Det följande kapitlet redovisar de resultat vi samlat. Inledningsvis sammanställer vi intervjuresultatet. Intervjuerna redovisas inte genom citat utan som en sammanfattning av de åsikter som användarna uttryckt. Avslutningsvis redovisas resultatet från användbarhetstesterna med tillhörande kommentarer.

Intervjuresultat

Fyra personer intervjuades av en prognosticerad användargrupp på max 30 personer det närmsta året. Vilket innebär en andel på över 10 procent. Individerna som har intervjuats är också nyckelpersoner på den avdelning som främst kommer att nyttja applikationen, samt de har olika roller vid utvecklingen av DPE. Detta borde utgöra ett adekvat urval både vad gäller antalet intervjuade personer och hur lämpliga dessa är.

Intervjuerna med användarna genomfördes som en öppen diskussion där vi utgick från ett antal punkter som relaterades till våra hypotetiska antaganden. Vi bad dock aldrig användarna konkret värdera kvaliteten på gränssnittet då de inte alltid var familjära med kvalitetsterminologin och innebörden hos dessa begrepp. Vi undvek också att ställa enkla ja och nej frågor för att pröva antagandena i hypotesen, på grund av att sådana frågor skulle kunna vara alltför ledande. Vi har istället styrt diskussionen så att den berör de ämnen och områden som är relevanta för hypotesen, men ändå låtit användarna med egna ord kommentera sina intryck och vad som är bra eller dåligt med gränssnittet.

Ge användaren kontroll över systemet

Som läsaren kanske minns har vi i vår hypotes antagit att expertanvändare har speciella önskemål angående behovet av att ha kontroll. Vi tror att vår typ av användare inte vill tappa möjligheten till kontroll över systemet, och trots att vi bygger in ett grafiskt stöd för de flesta funktioner så tror vi oss behöva behålla möjligheten att fortsätta jobba manuellt. Vi bygger således en applikation med låg koppling och hoppas genom detta tillfredsställa användarna.

Vid vår uppföljningsintervju visade det sig också att vi haft rätt idéer. Kontrollbehovet ansågs vara mycket viktigt - man vill kunna komma in under skalet, som någon uttryckte det. Det händer ofta att man vill utföra någon form av ovanlig och avancerad konfiguration, och man ser helst att man då utför dessa manuellt utanför gränssnittet. Trots att vi jobbat hårt med att bibehålla en låg koppling har det visat sig att användarna skulle vilja ha en ännu lägre

koppling. Till exempel uppdateras start-scriptet (S20_dpeStart.sh) genom vår applikation på ett sätt som upplevs som lite dumt. Vi använder scriptet som det är, men ändrar några få kommandon. Användarna hade hellre sett att vi sparar undan det uppdaterade scriptet som en ny kopia och låter det ursprungliga scriptet vara som det är. Vidare tycker användarna att gränssnittet skall spegla det underliggande systemet på ett bra sätt, och man gör en skillnad mellan att spegla detta och att spegla hur koden är skriven. Man vill inte att man genom att använda applikationen skall kunna se exakt hur koden är skriven, och detta är något vi lyckats med.

Information om ASL presenteras i den inbyggda konsolen. Denna är man relativt nöjd med. Dock hade man önskat sig att all loggning gjordes till fil, så att man kunde öppna denna för att se allt som utförts – som det är nu raderas loggningen då applikationen stängs ner. Eventuellt hade man kunnat nöja sig med statusbaren om all loggning gjordes till fil. Då hade man kunnat spara ytterligare lite arbetsyta.

Tillmötesgå testarens/programmerarens utrymmesbehov

I våra inledande användarintervjuer poängterades ett relativt ostrukturerat men bestämt önskemål om att applikationen skulle vara en ”liten hanterbar sak”. Med detta menades att den skulle vara enkel och inte ta upp en massa arbetsyta. Den skulle kunna ”ligga uppe i hörnet och tuffa på medan man använder en massa andra program”. Detta ställer stora krav bl.a. på valet av metod för att visualisera en nodkonfiguration. Vi valde till slut att använda oss av ett filträd för representation av noden. Detta tar lite plats i anspråk, är enkelt att bygga ut och förminska samt kan på ett enkelt sätt visa hierarkiska nivåer. Vi hade inledningsvis idéer om att använda oss av någon form av manipulerbara objekt, men valde bort detta alternativ av utrymmesskäl samt av skäl som har att göra med implementeringssvårigheter.

Användarna tycker att valet av filträd är bra eftersom det sparar mycket utrymme. Den generella uppfattningen hos användarna är att vi lyckats uppfylla kravet på en ”liten hanterbar sak”. Programmet behöver sällan ta mer utrymme i anspråk än en sjättedel av den totala arbetsytan på skärmen, och det finns gott om plats för andra program – editorer, kompilatorer m.m.

Anpassa stöd efter användarnas intresse och kunskapsnivå

Som vi tidigare redovisat har vi tagit fasta på det faktum att typen av information är viktig att anpassa till användaren. Den feedback som systemet ger bör också vara i saklig och neutral ton. Man skall undvika att i onödan göra användaren tveksam eller orolig över eventuella valmöjligheter eller påpekanden. Vi har valt att använda korta, formella påpekanden om vad som sker i systemet. Detta manifesteras i vår applikation genom att vi använder oss av teknisk information som användarna förstår. Vid olika typer av ”exceptions” använder vi oss i så stor utsträckning som möjligt av information i form av systemmeddelanden tagna direkt från ASL för att undvika förvirring. Genom detta tror vi oss ha lyckats få användarna nöjda. Då vi anser att vi lyckats göra en hanterbar och enkel applikation har vi också valt att undvika avancerat hjälpstöd såsom walkthrough och tutorial.

Från användarhåll är man relativt nöjd med typen av information som systemet ger. Man tycker att den är ”på rätt nivå” – dock hävdar man att det är svårt att avgöra vad som är bra eller dåligt och att det skulle vara lättare för en nybörjare att känna om det ligger på rätt nivå. Användarna poängterar vikten av att få feedback . Att vi som feedback använder oss av systemmeddelanden upplever man som bra. Det finns, tycker man, en stor risk att vi som utvecklare inte vet tillräckligt om vilken typ av information man som användare vill ha. Om

vi då skulle designa egna felmeddelanden är risken stor att vi försämrar förståelsen för systemet, däremot kan vi genom att så långt det är möjligt använda oss av systemmeddelanden undvika missförstånd.

Avsaknaden av tutorial och walkthrough är enligt användarna berättigad då man å ena sidan hävdar att en hanterbar applikation är misslyckad om det finns ett behov av den typen av stöd. Det man inte har kunskap om är hur vår applikation handhas rent konkret, det vill säga hur gränssnittets funktionalitet är beskaffad. Man är således tillfreds med att vi i hjälpsektionen endast kortfattat tar upp det konkreta handhavandet.

Använd enbart befintlig funktionalitet/rationalisera bort oprioriterade moment

Detta avsnitt är delvis kopplat till föregående när det gäller att åstadkomma en låg koppling. Att endast bygga in stöd för den redan existerande funktionaliteten innebär att man tvingas gå utanför gränssnittet för att utföra vissa moment. Användarna upplever i viss utsträckning att de i vår applikation saknar stöd för vissa moment, men man anser också att vi fokuserat på det viktigaste. Som vi tidigare nämnt var vår ambition att implementera ett stöd för att kunna köra processer på flera arbetsstationer samtidigt. Detta hade dock varit ett synnerligen tidsödande arbete, och vi valde att lägga det åt sidan. Detta är något som användarna också har accepterat och därmed har det inte varit svårt att begränsa utbyggandet av funktionaliteten. Vi har därmed också relativt enkelt kunnat motivera och få användarstöd för att inte bygga in stöd för installations- och uppackningsprocessen. Se tidigare resonemang, sid. 62.

Vid våra inledande intervjuer framkom även att man mycket gärna fått ett inbyggt stöd för den s.k. ”repair handling button” som finns på en fysisk nod. Detta skulle dock kräva att delar av DPE-koden måste skrivas om. Därmed ligger detta utanför vad vi kunnat prestera på den tid vi haft till förfogande.

Inkludera endast features som ökar användbarheten hos systemet.

Vi hade kunnat välja att ägna tid åt att exempelvis bygga in stöd för att presentera loggfilernas utskrifter i gränssnittet. Det gjorde vi inte. Användarna tycker att detta fungerar tillräckligt bra ändå. Man tycker att det går enkelt och snabbt att använda sig av de metoder för loggfilssökningar som man har tillgång till idag. Man känner sig trygg och effektiv då man använder dessa. Man hade inledningsvis vissa idéer om att vi skulle kunna ge ett bättre grafiskt stöd för att göra utsökningar i loggfilerna, men detta är något som vi inte ansett oss kunna förbättra – snarare tvärtom. Vid våra uppföljningsintervjuer har användarna också förstått att det vore mycket svårt och framför allt tidsödande att implementera ett bättre stöd för detta än vad man har tillgång till idag.

Spegla verkligheten men premiera enkelhet och snabbhet före exakt avbildning

Genom användandet av filträd som en representation av noden har vi hoppats kunna spegla verkligheten och därmed underlätta förståelsen för de abstraktioner som presenteras i GUI. Användarna har uttryckt mestadels positiva omdömen om filträdet. Man tycker sig få en god förståelse för de abstraktioner som detta innebär. De hierarkiska nivåerna ger en direkt överblick över de olika nivåer som finns i en nodkonfiguration, och man förstår intuitivt vad man gör. En ännu tydligare differentiering mellan de olika nivåerna hade dock varit önskvärd. Man önskar att man fick tillgång till olika symboler för olika nivåer i trädet istället för de

textförkortningar vi valt att använda. Således kan vi konstatera att vi abstraherat lite för mycket – vi borde eventuellt haft en ännu klarare spegling av verkligheten.

Eventuellt skulle vi kunnat visa endast PM och undvikit att visa alla de olika nivåerna. Det är trots allt endast PM som man utför någon typ av operation på. Att visa endast PM hade också tagit upp mindre plats på skärmen i och med att trädet i så fall kunnat rationaliseras bort. Dock skulle förmodligen överblicken bli sämre, i synnerhet om man i sin konfiguration har ett stort antal PM. Dessutom tyckte inte användarna att någon ytterligare förenkling behövdes.

Införandet av filträdet visade sig inte endast ge positiva effekter. Samtidigt som det ger möjlighet till snabba konfigurationer kan man som användare uppleva att det är svårt att veta på vilken hierarkisk nivå man för tillfället befinner sig. Detta blir som mest tydligt då man vill använda sig av combo-boxen för att lägga till ett nytt kortkluster då man måste markera en rätt nivå för att få tillgång till rätt element.

Underlätta återupprepning av tester

Snabbheten i vår applikation är enligt användarna en av de allra största fördelarna. Själva tanken med att bygga ett grafiskt användargränssnitt var också just att snabba upp konfigurationsmomentet, och detta har vi lyckats med. Användarna upplever att det går mycket fort och enkelt att återupprepa tester genom att använda vårt filträd. Man kan enkelt lägga till och ta bort element tills man är nöjd, och sedan spara undan konfigurationen. Man kan också snabbt och lätt komma åt de senast sparade konfigurationerna i en speciell meny – detta för att spara ytterligare tid och underlätta återupprepningar.

Se till att alla möjliga testfall är genomförbara

Alla testfall är möjliga utanför vår applikation, och vår ambition har varit att så långt det är möjligt implementera ett stöd för de flesta testfall. Vi har dock inte alls hunnit bygga in stöd för alla möjliga testfall. Det saknas exempelvis ett nätverksalternativ för stöd av multipla arbetsstationer samtidigt men av orsaker vi tidigare redovisat har denna typ av extra funktionalitet rationaliserats bort. Nätverkskopplingen går att utföra ändå – vid sidan av vår gränssnittsapplikation.

Förtydliga olika tillstånd hos systemet

Om dynamiken i systemet speglas så förbättras det övergripande intrycket av gränssnittet och för att uppnå detta krävs att processer och andra dynamiska entiteters tillstånd och status redovisas. Tydliga skillnader mellan offline och online under testningsproceduren bör ge en ökad intuitivitet och förståelse för gränssnittets koppling till systemets processer.

Användarna känner sig relativt nöjda med hur vi visar skillnaden mellan ”av” och ”på” vid testningen. Användandet av bandspelarsymboliken känns lämplig, likaså det faktum att man enkelt ser om knapparna är intryckta eller inte. Man är också tillfreds med att samtliga utdistribuerade processers tillstånd redovisas. Man skulle dock kunna tänka sig en ytterligare utökad visualisering av olika tillstånd i systemet, exempelvis om applikationen har låst sig. Man skulle tycka att det vore synnerligen bra om man i gränssnittet fick information om orsaken till *varför* ASL frusit, och exakt hur långt programmet kommit vid uppstart. Dock inser man att det skulle vara mycket svårt för oss att implementera detta stöd eftersom vår applikation startar upp andra processer. Vår applikation har ingen större kontroll över vad andra processer gör, och vi kan heller inte ta reda på allt.

Ge hela tiden användaren tillgång till relevanta verktyg

Vi har försökt ge användaren möjlighet att utföra allt som kan utföras i varje situation utan att han/hon ska tappa fokus genom att behöva byta arbetsfönster. Detta innebär att vi strävat efter att samla funktionalitet för ett visst arbetsmoment endast i den arbetsyta, eller del av arbetsyta som berörs. Användarna upplever att man för det mesta har god tillgång till de verktyg som krävs utan att behöva utföra speciella operationer. Någon hävdar att det visserligen krävs att man använder applikationen några gånger innan man känner sig hemma med verktygen, men att man relativt snabbt kommer underfund med funktionaliteten.

Betona det naturliga flödet

Som läsaren kanske minns har vi tidigare argumenterat för varför vi valt att undvika dialogrutor. Vi bygger istället in eventuella felmeddelanden i själva arbetsytan, vilket gör att man aldrig behöver flytta fokuseringen från denna.

Användarna hävdar också bestämt att de gärna slipper dialogrutor för att dessa stör arbetsgången. Vidare är det en åsikt att det inte behövs dialogrutor om gränssnittet är genomtänkt från början med en intuitiv placering av knappar och menyer. Dock menar man att det oftast bör finnas medel för att reparera misstag genom en ”undo-funktion.” Då vår applikation inte innehåller några riskfyllda arbetsmoment har vi kunnat undvika även ”undo-funktionen”. Detta upplevs inte som en nackdel av användarna då de aldrig känt ett behov av en sådan i vår applikation.

En av användarna hittade något som han upplevde som en nackdel med avsaknaden av dialogrutor. Han konfigurerade en nod, och valde sedan ”New node” utan att spara den förra. Då skrivs den gamla över med ett nytt, tomt konfigureringsträd. En dialogruta innehållandes meddelandet: ”Vill du spara konfigurationen?” hade användaren i fråga önskat sig i denna situation. Då denna synpunkt kom fram insåg vi att det finns fler situationer då ett mer offensivt stöd skulle kunna vara lämpligt. Om man exempelvis försöker spara en fil som man för tillfället inte har åtkomst till kan det troligtvis vara lätt att tro att man ändå lyckats spara filen. Meddelandet ”Couldn´t save file x” skrivs visserligen i konsolfönstret men missförstånd skulle eventuellt lättare undvikas om en dialogruta användes. Användarna är dock i stort sett eniga om att man hellre slipper dialogrutor, åtminstone i de allra flesta fall.

Andra metoder för att betona det naturliga flödet är att se till att exempelvis menyer, fönster och verktygsfält är logiskt uppbyggda där knappar och menyval har lämplig struktur och ordningsföljd. På denna punkt har vi i stort sett lyckats uppfylla användarnas önskemål. En användare uttrycker sig på följande sätt:
”Flödet börjar från vänster – går sedan ner i de stora fönstren för att konfigurera, sedan upp igen då man skall exekvera. Man tänker inte så mycket på det.”

Att välja rätt standarder vid val av ex. kortkommandon, musbeteenden etc. är också något som är viktigt för att åstadkomma en applikation med naturligt flöde. Användarna hade vissa invändningar mot våra val av kortkommandon vilka man ansåg vara lite onödigt omständiga. Man hade exempelvis önskat sig funktionsknapparna F1 och F2 för start och stopp. Detta tyckte vi på ett tidigt stadium hade varit det naturliga valet eftersom det är gängse standard i denna typ av applikationer – problemet har dock varit att vi inte lyckats implementera detta. Av någon anledning har det varit svårt att, i Java swing, hitta en lösning på problemet. De musbeteenden som finns tillgängliga är däremot tillfredsställande för användarna – det handlar främst om att vi implementerat högerklicksalternativ för de flesta standardoperationer. Man hävdar från användarhåll att det inte är särskilt troligt att högerklicksalternativen

kommer att användas då dessa upplevs som ett omständigare och långsammare sätt att utföra annars snabba operationer. Dock hävdas att det ändå är trevligt med många alternativ.

Passivt stöd – dirigera istället för att diskutera

Genom att låsa de menyer, knappar, fält m.m. som inte kan eller bör användas för tillfället har vi förhoppningsvis kunnat generera ett naturligt passivt stöd som innebär att man får svårt att utföra annat än rätt sak. Detta upplevs av användarna som bra förutom på en punkt. Man måste börja konfigureringen med att markera roten i trädet innan man kan börja bygga på med magasin. Man tycker att roten borde markeras automatiskt. Detta kan vi bara instämma i. Det är en detalj som av någon anledning inte kommit fram vid vår prototyping.

Så kallade tool-tips används också som ett passivt stöd, något som uppfattas som en bra form av idiom. Med idiom menas att man redan efter att ha använt programmet ett par gånger lärt sig knapparnas funktion. Någon tycker att det upplevs som störande med tool-tips och att det hade varit bättre om det krävdes lite mer tid för att komma underfund med knapparnas betydelser. Vi är beredda att till viss del instämma i kritiken eftersom vår applikation endast innehåller 11 knappar. Det skulle inte dröja särskilt lång tid innan användaren var bekant med samtliga knappar.

Resultat av användbarhetstest

I vårt fall är inte applikationen mer omfattande än att vi med lätthet kan fokusera på de delar av användningen som är centrala. Det handlar om att försöka se hur pass enkelt det är att konfigurera upp en nod, köra igång den, stoppa den, rekonfigurera den och i övrigt interagera med den. Som försöksobjekt har vi valt en central framtida användare, som är van vid att använda ASL och dessutom vidarutvecklar DPE. Testet består av att mäta antalet upplevda irriterande moment vid användning av den nya applikationen jämfört med antalet irriterande moment vid användning av den gamla applikationen. Dessutom mäter vi tiden det tar att använda den nya applikationen kontra den gamla.

Då testet utfördes hade vi låtit deltagaren bekanta sig med den nya applikationen från och till under en veckas tid för att i så stor utsträckning som möjligt eliminera effekten av att deltagaren är mer bekant med den gamla applikationen.

Testscenario 1

Vi bad först deltagaren att i normal arbetstakt konfigurera upp en nod med ett förutbestämt utseende med hjälp av den gamla applikationen. Deltagaren skulle sedan köra igång den, vänta tills alla processer var utdistribuerade, pausa applikationen, tillföra ett visst antal element och köra igång den igen. Vi bad deltagaren göra oss uppmärksamma varje gång ett moment upplevdes som irriterande. Deltagaren skulle under körningen inte utveckla resonemanget varför ett visst moment upplevdes som irriterande utan endast signalera. Vi övervakade arbetet och räknade samman samtliga irriterande moment. Vi mätte också tiden det tog att utföra hela scenariot. Efter körningen gick vi igenom mer noggrant vad som upplevdes som irriterande och varför.

Efter detta var det dags att utföra exakt samma scenario med vår nya applikation för att synliggöra eventuella skillnader i hur deltagaren uppfattar handhavandet i de två olika applikationerna. Återigen övervakade vi tid och antal irriterande moment.

Resultat 1

Med den gamla applikationen tog arbetet **9 minuter och 35 sekunder** att utföra. Antalet irriterande moment uppgick till **14**.

Med den nya applikationen utfördes arbetsmomenten på **5 minuter och 10 sekunder**. Antalet irriterande arbetsmoment var nu **5**.

Då testscenario 1 var utfört bad vi deltagaren att upprepa detta ytterligare två gånger för att till viss del eliminera slumpfaktorer. Det sammanlagda antalet utförda testscenarion av typen 1 uppgår alltså till 3 för varje applikation.

Med den första gamla applikationen tog det sammanlagda arbetet **27 minuter och 40 sekunder** att utföra. Antalet irriterande moment uppgick till **41**.

Med den nya applikationen utfördes arbetsmomenten på **14 minuter och 50 sekunder**. Antalet irriterande arbetsmoment var nu **15**.

Testscenario 2

Vi bad användaren öppna en sparad nodkonfiguration från fil, köra igång den, vänta tills samtliga processer var utdistribuerade, pausa, ta bort ett visst antal element och köra igång den igen. Användaren skulle även avsluta med att stoppa distributionen samt spara över den gamla konfigurationen med den nya. Dessa moment utfördes först med den gamla applikationen och vi mätte återigen samma variabler – tiden och antalet upplevda irritationsmoment.

Exakt samma kedja av moment utfördes sedan med den nya applikationen.

Resultat 2

Med den första gamla applikationen tog arbetet **8 minuter och 42 sekunder** att utföra. Antalet irriterande moment uppgick till **19**.

Med den nya applikationen utfördes arbetsmomenten på **4 minuter och 0 sekunder**. Antalet irriterande arbetsmoment var nu **2**.

Vi bad även deltagaren att upprepa scenario 2 ytterligare två gånger för att till viss del eliminera slumpfaktorer. Det sammanlagda antalet utförda scenarion av typen 2 uppgår alltså till 3 för varje applikation.

Med den första gamla applikationen tog det sammanlagda arbetet **25 minuter och 20 sekunder** att utföra. Antalet irriterande moment uppgick till **59**.

Med den nya applikationen utfördes arbetsmomenten på **11 minuter och 30 sekunder**. Antalet irriterande arbetsmoment var nu **8**.

Sammanfattning av användbarhetstestets resultat

Nedan används en tabell för att åskådliggöra hur dessa irritationsmoment fördelade sig enligt de tidigare diskuterade kvalitetsfaktorerna. Vi har valt att markera de anmärkningsvärda resultaten med fetstil.

Gamla applikationen

Genomsnittlig tid för testscenario 1: 9min.20sek.
Genomsnittlig tid för testscenario 2: 8min.27sek.
Genomsnittligt antal irritationsmoment för testscenario 1: 13.67 stycken
Genomsnittligt antal irritationsmoment för testscenario 2: 19.67 stycken

Nya applikationen

Genomsnittlig tid för testscenario 1: 4min.53sek.
Genomsnittlig tid för testscenario 2: 3min.50sek.
Genomsnittligt antal irritationsmoment för testscenario 1: 5.00 stycken
Genomsnittligt antal irritationsmoment för testscenario 2: 2.67 stycken

Den nya applikationen utför båda testscenarion ungefär dubbelt så snabbt jämfört med den gamla applikationen. Antalet upplevda irritationsmoment är för testscenario 1 i genomsnitt 2.73 ggr fler med den gamla applikationen och hela 7.37 ggr fler för testscenario 2.

KVALITETS- FAKTOR	APPLIKATION	ANTAL IRRITATIONS-MOMENT I SCENARIO 1	ANTAL IRRITATIONS-*MOMENT I SCENARIO 2*
Understandability	NYA	3	2
	GAMLA	2	2
Learnability	NYA	4	0
	GAMLA	3	1
Operability	**NYA**	**6**	**2**
	GAMLA	**31**	**52**
Attractiveness	NYA	2	2
	GAMLA	5	4
Helpfulness	NYA	0	1
	GAMLA	0	0
Customisability	NYA	0	1
	GAMLA	0	0
Totalt	**NYA**	**15**	**8**
	GAMLA	**41**	**59**

Diskussion kring användbarhetstestet

För båda båda användbarhetstesterna gäller att vår applikation reducerar antalet irritationsmoment – i det ena fallet är det en dramatisk minskning. Det är svårt att avgöra med säkerhet varför skillnaden blir så pass mycket större i testscenario 2 än i scenario 1. Vi finner två troliga anledningar. Dels tror vi att testscenario 2 innehåller fler moment där den gamla applikationen synliggör sina största svagheter. Dessa moment är t.ex. att öppna konfiguration från fil och spara till fil då man tvingas navigera i katalogstrukturer genom att använda sig av ett terminalfönster. Dessa moment underlättas avsevärt i det nya gränssnittet. En annan anledning till att antalet irritationsmoment i testscenario 2 blir så pass lågt med vår applikation tror vi beror på att det upplevs som svårare att bygga upp ett träd från grunden än att utgå från ett redan färdigt träd. Vi tror att de mest tankekrävande momenten i vår applikation är just att få rätt element på rätt hierarkiska nivå i trädet. I testscenario 2 handlade det ju endast om att *ta bort* element från trädet, och då slapp man det allra jobbigaste momentet. Det är helt enkelt en viss inlärningsperiod innan man lärt sig att hitta rätt i trädstrukturen, och om detta är fallet så tror vi också att även testscenario 1 skulle visa upp en större skillnad mellan den gamla och den nya applikationen.

Värt att notera är att deltagaren har arbetat under ett par års tid med den gamla applikationen och under en knapp veckas tid med den nya applikationen. Trots avsaknaden av vana hos användaren visar sig den nya applikationen vara så pass mycket effektivare. Vi tycker det är synd att vi inte kan jämföra våra resultat med någon annan applikation. Nu kan vi endast jämföra med den gamla applikationen – detta är en relativt orättvis jämförelse då den inte har något grafiskt användargränssnitt alls. Det är inte särskilt svårt att göra ett GUI som är bättre än ”inget alls”.

Värt att notera är också att vi i våra testscenarion valde nodkonfigurationer bestående av relativt få element. Eftersom det moment som är väsentligt snabbare med det nya gränssnittet är att just lägga till och ta bort element är det rimligt att anta att vinsterna bör bli än större vid mer omfattande konfigurationer.

Del 7 Diskussion och slutsats

I det följande kapitlet diskuterar vi resultatet från hypotesprövningen. Vi utvärderar resultatet och drar slutsatser i form av en reviderad hypotes. Hypotesprövningen skedde genom att utvärdera en GUI-prototyp via användbarhetstest och intervjuer med användarna. Vi tog också hänsyn till synpunkter som kom fram under prototypingprocessen.

Vår slutsats har som ambition att besvara om vår hypotes stämde eller inte. Förbättrades användbarheten på det sätt som vi förutsåg? Vi anknyter till frågeställningen i introduktionskapitlet:

- *Fungerade riktlinjerna i praktiken?*

Om detta inte är fallet, så försöker vi besvara hur man bör formulera om riktlinjerna. Det reviderade förslaget på riktlinjer för GUI-design bör sedan utgöra en bättre mall som någorlunda adekvat besvarar den fråga vi inledningsvis utgick ifrån:

- *Hur utformas ett grafiskt användargränssnitt för testning av inbäddade system med syfte att optimera användbarheten för expertanvändare?*

Kapitlet inleds med ett avsnitt om utvärderingens tillförlitlighet. Sedan diskuteras resultatet från använbarhetstesterna och intervjuerna. Slutligen presenterar vi ett förslag på hur de reviderade riktlinjerna borde se ut.

Utvärderingens tillförlitlighet

Kvalitetsutvärderingar är ofta svåra att genomföra, i synnerhet när det handlar om kvalitetskriterier som användbarhet. Problemet är att hitta metoder för att objektivt mäta graden av kvalitet. Klassisk empirisk metod för att säkerställa tillförlitligheten på vetenskapliga undersökningar är inte alltid tillämpbara då det är svårt att kvantifiera vissa kriterier för kvalitet. Alternativet blir att förlita sig på kvalitativa intervjuer och de subjektiva värderingar sådana intervjuer medför.

Intervjuer

Intervjuresultatet är adekvat både vad gäller antalet utfrågade personer och urvalet av dessa. Möjligen kunde en mer strukturerad intervjuform med konkreta fördefinierade frågor gett ett annorlunda resultat, men vi är ändå nöjda med den feedback som vi utvann ur våra öppna diskussioner.

Den uppenbara begränsningen med att använda kvalitativa intervjuer som utvärderingsmetod är att den är subjektiv. Användarnas subjektiva åsikter och intryck tolkas i sin tur av oss vid en ytterligare subjektiv utvärdering och slutsats. Den subjektiva faktorn går dock inte att undvika utan endast att minimera. Genom att vara medveten om sin egen roll vid intervjuerna och analysen av dessa har vi försökt bibehålla den objektivitet som finns.

Användbarhetstest

Det är givetvis att betrakta som en svaghet att endast använda sig av en enda deltagare. Det är fullt möjligt, rent av troligt att andra deltagare hade haft helt andra upplevelser. Tyvärr hade vi

inte tillgång till fler testobjekt på grund av tidsbrist och en ovilja att delta hos de berörda användarna. En användare utgör trots allt en ganska stor del av den tänkta framtida användargruppen, men det ändrar inte faktum och problemet kvarstår. Testen borde utförts med åtminstone 3-5 användare för att få ett mer tillförlitligt resultat. Vi har därför sett testresultatet som ett komplement till intervjuerna och använt de tendenser som kan urskiljas där, som ett stöd för de slutsatser vi dragit från intervjuresultatet.

Vidare går det att ifrågasätta lämpligheten att jämföra gränssnittet hos den nya applikationen med ASL:s. Istället borde man jämföra applikationen med ett annat program med liknande användningsområde, syfte och egenskaper. Det programmet bör dessutom ha ett grafiskt gränssnitt utvecklat efter konventionella designprinciper för gränssnitt så att det går att urskilja skillnader mellan vårt designförslag och ett mer gängse. Eftersom vi inte hade tillgång till någon sådan mjukvara skulle vi varit tvungna att ta fram en alternativ gränssnittsprototyp. Vi gjorde inte detta på grund av tidsbrist.

Reflexioner kring användbarhetstesterna

Vinsten i tid är det mest uppenbara resultatet från användbarhetstesterna. Det tar i snitt hälften så lång tid att använda den nya applikationen jämfört med ASL. Viss osäkerhet kan urskiljas beträffande tidsåtgången då stor del av tiden går åt till att vänta in DPE. Det tar olika lång tid för DPE att distribuera ut och initiera nya processer beroende på en rad faktorer som vi inte kunnat påverka. Exempelvis hur trafikerat nätverket är, tillgänglighet på minne och processorkraft. Vår uppfattning var dock att DPE betedde sig på ett likartat sätt under samtliga test och att en stor del av den totala tidsåtgången utgjordes av just väntan på DPE.

Detta faktum innebär att den reella tidsvinsten vid själva *hanteringen* av gränssnittet är ännu större. En ungefärlig uppskattning av väntetiden vid det första scenariot är 3-4 minuter, vilket skulle innebära att hanteringen av ASL tog ungefärligen:

9min 20 sek – 3min 30sek = **5 min 50 sek.**

Samma beräkning utförd mot den nya applikationen ger:

4min 53sek – 3min 30sek = **1 min 23 sek**.

Själva hanteringsförfarandet blir i detta fallet **4.2** gånger snabbare med den nya applikationen. Denna beräkning är inte exakt och visar inte att användarna gör ytterligare tidsvinster utan påvisar att den nya applikationen underlättar och förenklar hanteringen mer än vad de officiella tidsresultaten från testerna uppvisar. Eftersom användaren fortfarande inte var helt intränad på att använda den nya applikationen men var mycket van vid att hantera ASL innebär att man kan förvänta sig ytterligare tidsvinster vid kontinuerligt nyttjande av vårt program.

Vinsten i tid implicerar att de kvalitativa förbättringarna hos gränssnittet görs vid handhavandet av programmet (*operability*) samt i viss mån inlärning (*learnability*) och förståelse (*understandability*) av programmet. Faktumet att irritationsmomenten nästan uteslutande relaterades till operability understryker att det är just hanteringen som förbättrats och att det är där de största kvalitativa vinsterna gjorts.

Vid en första anblick uppfattar man det även som att det gjorts stora vinster vid antalet upplevda irritationsmoment. Eftersom dessa som sagt till störst del utgörs av operability

relaterade moment så speglar detta främst tidsresultatet. Man kan resonera sig fram till att genom att slippa en sån mängd irriterande detaljer så bör också den nya applikationen vara betydligt mer attraktiv (*attractiveness*). Det går också att urskilja en sådan tendens i tabellen, där man finner dubbelt så många irritationsmoment relaterade till attraktivitet i ASL än hos den nya applikationen.

Vad gäller de andra resultaten från antalet upplevda irritationsmoment så ser man att understandability och learnability verkar förbättras från det första till det andra scenariot, men att det inte är större skillnader mellan gammalt och nytt. Snarare har saker försämrats med den nya applikationen beträffande inlärning och förståelse. Detta lite annorlunda resultat kan förklaras med att det andra scenariot utfördes efter det första samt att användaren hade vant sig vid hanteringen efter en runda och inte kommenterade samma moment en gång till. Beträffande inlärningen och förståelsen så måste man vara medveten om att användaren redan kan ASL men samtidigt inte är lika van vid den nya applikationen. Att det ändå är så få irritationsmoment förknippade med inlärning och förståelse antyder att det nya GUI:t är enkelt och intuitivt. Det kräver inga större intellektuella ansträngningar att tillägna sig så länge man har den grundläggande kunskapen om DPE och ASL.

Sammanfattningsvis kan man egentligen bara dra en slutsats från användbarhetstesterna. Det har skett en avsevärd förbättring av användbarheten hos gränssnittet och att denna förbättring utgörs i huvudsak av ett enklare och snabbare handhavande (*operability*). Gränssnittet hos den nya applikationen är rimligtvis också betydligt mer attraktivt, medan det är osäkert om det gjorts förbättringar för inlärning och förståelse. Däremot gavs det inga indikationer på hur kvalitetsfaktorer som helpfulness och customisability påverkats.

Diskussion - Riktlinjer

Det grafiska gränssnittet utformades i enlighet med de riktlinjer vi utformat i hypotesen. Vi påstod att genom att följa riktlinjerna skulle användbarheten förbättras. Varje riktlinje skulle förbättra användbarheten på sitt enskilda vis, både allmänt och i samklang med de andra. Riktlinjerna skulle även specifikt påverka underordnade kvalitetskriterier i enlighet med de förutsägelser vi härledde under varje punkt i hypotesavsnittet.

Ge användaren kontroll över systemet

Vi hade rätt i att urskilja ett behov, hos användarna, att behålla kontrollen över systemet. Varje gång vi sett någon användare testa applikationen så kontrollerar de vad som händer i ASL. De litar inte på programmet och behöver övertygas om att allt fungerar som det ska. De vill manipulera ASL manuellt och inte begränsas av att GUI automatiserar script och filer på ett oöverskådligt och av användarna okontrollerbart sätt. Vi löste detta genom att behålla en låg koppling mellan GUI och ASL samt genom att ge kontinuerlig information till användaren om vad och hur ASL manipulerades av programmet. Genom att vi inte lyckades implementera en tillräckligt låg koppling accentuerades det faktum att detta var av största vikt. Vi uppfattade en tydlig skepsis till att använda programmet då det framkom att vi uppdaterade vissa filer utan att återställa ASL i ursprungsskick efter varje testkörning.

Feedback om vad som sker med ASL fångade däremot inte användarnas intresse. Informationen kom för snabbt och var inte tillräckligt detaljerad för att ge användarna kontroll över situationen. Användarna var egentligen bara intresserade av sådan information om den påverkade manuella justeringar av något slag och först då den manuella hanteringen strulade.

Slutsatsen blev att feedbacken som rörde ASL inte var nödvändig i den form som vi erbjöd utan att den istället borde loggas till fil för senare bruk. Informationen som vi presenterade i konsolfönstret kunde således framställas betydligt mer kortfattat i en statusbar, medan den mer omfattande informationen skrevs till fil.

Sammanfattningsvis urskiljer vi att det är av största vikt att låta användarna behålla kontrollen över det bakomliggande systemet genom att främst bibehålla en låg koppling. Detta medför att användarna är villiga acceptera och använda systemet. Om kopplingen mellan GUI och ASL påverkar den manuella hanteringen kommer applikationen att ratas och inte användas. Låg koppling påverkar i detta fall mjukvarans attraktivitet (*Attractiveness*) positivt, på ett mycket avgörande sätt.

Tillmötesgå användarens utrymmesbehov

Efter att ha både intervjuat och studerat användarna framgår det med all tydlighet att applikationen skall ta upp så liten plats som möjligt. Användaren behöver ha överblick över programmet samtidigt som det kör även om han inte behöver interagera mot det kontinuerligt. Användarna har uppe flera konsolfönster och söker i loggfiler samtigt som man kör. Samtliga användare verkar nöjda över lösningen gällande utrymme.

Däremot tycker de inte att hanteringen av testningen som helhet blir enklare bara för att de slipper byta fönster utan det är överblicken som gör en ”liten” lösning attraktiv. Eftersom alla features (ex. visa loggfiler) finns tillgängliga i applikationen är den lilla lösningen att föredra. Däremot kunde all hantering (programmering, kompilering och testning) av ASL införlivats i applikationen skulle en ”helfönster”-lösning varit att föredra. Att tillmötesgå användarens utrymmesbehov innebär således inte att applikationen alltid skall vara så liten som möjligt, utan att den anpassas för de operationer och program som krävs för att lösa uppgiften.
Attractiveness ökar men däremot inte *operability*.

Anpassa stöd efter användarnas intresse och kunskapsnivå

Användarna saknar inte vare sig walkthrough eller tutorial. Hjälpmanualen anses tillräcklig för att sköta programmet och lära sig vad som är vad. Men sammantaget uppfattar man det inte som om hjälpen är bättre eller sämre än vad det går att förvänta sig. Att standardisera feedback i enlighet med systemmeddelanden ses positivt men användarna verkar egentligen inte bry sig.

Vår slutsats blir att det är rätt att rationalisera bort onödiga hjälpavsnitt, men att den hjälp som återstår inte blir förbättrad av det andra saknas. Däremot verkar användarna uppskatta att slippa onödiga stödfunktioner som de i vanliga fall aldrig behöver. Applikationen blir lite mer attraktiv (*attractiveness*) men ger ingen förbättrad hjälp (*helpfulness*). I stora drag verkar denna punkt inte behöva prioriteras så länge stödet inte håller undermålig klass.

Använd enbart befintlig funktionalitet och rationalisera bort oprioriterade moment

Användarna har inte utryckt någon saknad av stöd för uppacknings och installationsprocessen. De förväntar sig enbart kunna utföra samma moment som de kan utföra manuellt fast enklare, säkrare och lika diversifierat. Användare uppfattar programmet så enkelt och avskalat som möjligt och att det infriar de förväntningar de haft. Eftersom applikationen inte innehåller extra funktionalitet eller installationsstöd innebär det att den hålls så enkel som möjligt och sålunda blir lättare att hantera (*operability*). De röster som höjts gällande ytterligare

funktionalitet har varit få och lågmälda vilket ger slutsatsen att applikationens mångsidighet (*customisability*) inte verkar alltför stukad.

Inkludera endast features som ökar användbarheten hos systemet.

Användarna är i stort sett eniga på denna punkten också. Det finns ingen anledning att bygga in presentation och tracing av loggfilerna som användarna redan hanterar med lätthet. Eftersom applikationen tar så litet utrymme samverkar de olika verktygen väl under testningsförfarandet. Vi undviker att bygga in features som skulle kunna göra programmet klumpigt och skrymmande. Detta innebär inga direkta förbättringar beträffande operability eller attractiveness. Loggfilerna måste ändå hanteras och det vore givetvis lämpligt ur ett attraktivitetsperspektiv att samla alla operationer och features under ett tak. Men programmet blir enklare och går fortare att implementera, samtidigt som det redan finns ett adekvat stöd för loggfilerna.

Denna punkt innebär inga förbättringar för användbarheten men ser till att undvika försämringar. Det har dock framkommit uppgifter som talar för att den här punkten inte är aktuell i samtliga fall. Med andra förutsättningar och krav vore det kanske bättre att samla alla features under samma tak. Med större resurser och mer tid vore det kanske att föredra, men i detta fallet önskade användarna ha tillgång till applikationen så snabbt och billigt som möjligt. Avgränsningen bör kanske istället relateras till de krav som användaren har och de förutsättningar som ges till utvecklaren. Vilket skulle innebära att man följer generella riktlinjer för GUI-design som inte är specifika för testning.

Spegla verkligheten men premiera enkelhet och snabbhet före exakt avbildning

Under användbarhetstesterna framgick att det inte fanns några direkta problem med att förstå filträdstrukturen, hur den relaterades till en nodstruktur eller hur den hanterades. De påvisade att det finns både en inlärd och intuitiv förståelse av ett sådant grafiskt objekt som lätt översätts till användarnas bild av en nod. Intervjuerna bekräftade denna uppfattning liksom hela prototypingprocessen. All feedback under prototypingen har understrukit fördelen med att använda filträd. Det är endast när vi påpekat andra lösningar med exempelvis manipulerbara objekt som användarna reflekterat över eventuella brister och möjligheter. Ingen har dock funnit anledning att föreslå en annan lösning.

Lätt att förstå (*understandability*) och lätt att tillägna sig (*learnability*).

Underlätta återupprepning av tester

Genom att göra programmet så enkelt och lättillgängligt som möjligt, samtidigt som det lätt går att göra de konfigureringar som är nödvändiga, gör det väldigt enkelt att genomföra och återupprepa ett testscenario. Intervjuerna och användbarhetstesterna påvisar att detta både är det mest efterfrågade samt framgångsrikt implementerade delen av applikationen.

Det är väldigt enkelt (*operability*) att konfigurera noden, starta upp, manipulera processer och avsluta.

Se till att alla möjliga testfall är genomförbara

De flesta testfall är genomförbara men implementeringen har ändå blivit begränsad på ett flertal sätt. Framförallt har vi inte implementerat nätverksalternativet vilket varit väldigt efterfrågat bland vissa användare. Det intryck som har getts under intervjuerna är att alla

operationer som går att utföra i ASL fortfarande går att genomföra samtidigt som GUI-applikationen kör, men att det hade varit lämpligt om dessa operationer funnits tillgängliga i GUI:t. Det känns som ett ganska självklart krav som rimligen skulle tillfredställas. Men om ”alla” operationer implementerats skulle det samtidigt vara svårt att bibehålla den enkelhet flera andra riktlinjer förordar.

Vår slutsats blir att denna riktlinje inte är helt kompatibel med flera andra riktlinjer och vi konstaterar att vår implementering inte lyckats göra alla testfall genomförbara (*customisability*) utan att denna riktlinje möjligen skulle omdefinieras till ”så många som möjligt” istället för ”alla möjliga”.

Förtydliga olika ”tillstånd” hos systemet

Under hela prototyping processen har användarna påpekat att det är oklart om processer kör eller inte och vilken status som DPE har. Vi har ändå från första början försökt förtydliga olika tillstånd hos systemet och slutligen ändå varit någorlunda överens med användarna om att designen är okej. Trots att vi dels förtydligat tillstånden och användarna lärt sig att urskilja nyanserna, menar vi att detta inte är nog. Vi har delvis misslyckats med att förtydliga tillstånden och användarna uppvisar fortfarande en viss osäkerhet även om de inte vill erkänna det. Osäkerheten har spridit sig så att användarnas uppvisat ett allmänt osäkert beteende och tvekat under operationer som de utfört utan problem bara en stund tidigare. Tillstånden hos entiteter utgör väsentlig information för användarna under exekvering och sådan information bör vara exakt och tydlig.

Slutsatsen från detta kan bara vara att vi inte kan förtydliga tillstånd nog. Om man kan undvika osäkerhet blir GUI:t lättare att förstå (*understandability*) och framförallt skrämmer man inte bort användaren (*attractiveness*).

Ge hela tiden användaren tillgång till relevanta verktyg

Användarna uppfattar det som de har tillgång till rätt verktyg vid varje enskild situation men att det ibland varit svårt att hitta det. Detta är visserligen en inlärningsprocess, som i stort sett handlar om navigering. Då vill man att liknande beteenden får liknande resultat, att GUI:t uppvisar ett konsekvent beteende. Under prototypingprocessen och vid användbarhetstesterna visade sig detta fungera med all tydlighet. De flesta irritationsmomenten som rörde *operability* registrerades i början av testerna innan användaren blivit varm i kläderna. Vid flera tillfällen när användaren tvekade kring ett moment så prövade han ”liknande” handlingar som resulterade i ett ”Ja, just det, vad bra”.

Hanteringen av GUI:t underlättades (*operability*) av detta sätt att utforma gränssnittet men däremot så uppfattade inte användarna det som man hade en större valmöjlighet (*customisability*) än normalt. Detta beror möjligen på att GUI:t är så enkelt och avskalat. Det finns inga extra ”gömda” alternativ utan alla verktyg finns oftast redan framme. Sammantaget skulle man kunna påstå att denna riktlinjen är en självklarhet för vilken typ av gränssnitt som helst, men samtidigt uppfattar vi den som viktig och adekvat. Men den enda tydliga positiva effekten på användbarheten utgörs av förbättrad *operability*.

Betona det naturliga flödet

Både under prototypingen, användbarhetstesterna och intervjuerna visar det sig att flödet i programmet fungerar väl. Handlingsföljden är intuitiv och smidig. De kommentarer som gjorts angående brister i flödet och oklarheter hos GUI är samtliga resultatet av designmisstag

eller implementeringsproblem. Applikationen var inte felfri men ambitionen med ett skapa ett naturligt flöde var ändå rätt. Däremot fanns det fall där vi gått för långt, till exempel rationaliserade vi bort alla typer av felmeddelanden i dialogrutor varpå användarna inte uppfattade viktiga händelser som behövde vara mer ”högljudda”.

Det naturliga flödet bör betonas (utan att överdriva) och det underlättar hanteringen av programmet (*operability*).

Passivt stöd, dirigera istället för att diskutera

Användarna tyckte att i huvudsak att det var en bra lösning att låsa menyer för att undvika fel och förhindra att användarna gjorde otillåtna operationer i GUI:t. Låsta menyer och knappar skuggades. Samtidigt visade sig detta vara ganska irriterande innan man lärt sig vad som går och inte går att utföra vid en given situation. Man kunde inte utföra vad man tänkt sig då detta var låst och dessutom fick man ingen eller väldigt subtil information om varför detta inte gick att utföra. Efter inlärningsperioden minskade denna irritation men upphörde aldrig helt. Användarna tyckte dock att detta var oväsentligt på grund av att man undviker fel och slipper offensiva felmeddelanden. Användarna tycktes också lära sig snabbare vad som kunde göras då deras valmöjligheter begränsades via låsningen.

Passivt stöd innebar att det blev lättare att hantera (*operability*) samt lära sig (learnability) GUI:t. Däremot visade det sig inte vara lika populärt och attraktivt (*attractiveness*) som vi förutsett.

Slutsats - Riktlinjer

Under arbetet med att utveckla prototypen och sedan utvärdera denna har det framkommit att vissa av riktlinjerna går in i eller påverkar varandra. Några framstår som överordnade andra och tvärtom. Ett fåtal riktlinjer har även framstått som relativt generella för gränssnittsdesign överlag och är inte specifika för den typ av GUI som vi studerat.

Visserligen anser användarna att applikationen fungerar väl och generellt sett har en hög användbarhet. Men den uppvisar ändå begränsningar och vissa av riktlinjerna har i högre grad producerat de egenskaper och features som uppfattats positivt.

Reviderade riktlinjer

I efterhand uppfattar vi några faktorer som särskilt viktiga vid design av ett grafiskt gränssnitt för testning. Vi urskiljer att användarna snabbt vill kunna komma igång och återupprepa tester men samtidigt inte bli begränsade vid den ”manuella” testningen, det vill säga hanteringen av ASL. Användarna behöver också övertygas om att acceptera den nya applikationen samtidigt som de vill ha tillgång till ett testverktyg så snabbt som möjligt. De vill att applikationen skall vara okomplicerad och lätthanterlig. Uppstår det problem med programmet eller för den delen ASL, blir de misstänksamma och skeptiska. De signaler vi fått från utvärderingen säger oss att man löser detta genom en enkel och felsäker design. Ju enklare, desto lättare att utveckla och implementera, samt använda och lära sig applikationen.

Vi urskiljer att användarens behov av kontroll, enkelhet och snabbhet bör premieras och att alla riktlinjerna är underordnade dessa behov samt har till syfte att tillfredställa dessa.

Enkelt och snabbt

Den viktigaste riktlinjen är

(1) *Underlätta återupprepning av tester.*

Det åstadkommer man genom ett antal underordnade riktlinjer som är relaterade till enkelhet och snabbhet.

(2) *Betona det naturliga flödet.*
(3) *Passivt stöd, dirigera istället för att diskutera.*

Dessa riktlinjer bibehålls också intakta och ser till att det testningen genomförs på ett intuitivt och snabbt vis, utan störande och komplicerade moment. För att göra GUI:t och testningen så enkel som möjligt genomförs vissa prioriteringar. De följande riktlinjerna slås ihop:

Anpassa stöd efter användarnas intresse och kunskapsnivå.
Använd enbart befintlig funktionalitet och rationalisera bort oprioriterade moment.

De förenklas till:

(4) *Rationalisera bort oprioriterade moment.*

Eftersom det egentligen är en självklarhet att enbart den befintliga funktionaliteten används vid designen av ett GUI känns detta onödigt att påpeka. Allt faller under rationaliseringsprincipen, om det så är hjälp/stöd funktioner eller extra features, så skall de inte införlivas i gränssnittet för att bevara dess enkelhet. Även nästa riktlinje handlar om prioritering:

Inkludera endast features som ökar användbarheten hos systemet.

Denna bör dock formuleras om för att mer exakt beskriva vad dess syfte är. Det är självklart att enbart features som förbättrar användbarheten inkluderas i ett gränssnitt, vilket som helst. Det handlar snarare om:

(5) *Inkludera inte operationer och verktyg som redan har ett adekvat gränssnitt.*

Detta måste givetvis göras med varsamhet så att det totala användargränssnittet (alla verktyg som är nödvändiga för testningen) inte blir alltför splittrat. Vilket skulle försämra användbarheten.

Kontroll och överblick

Den näst viktigaste riktlinjen handlar om att användarna skall acceptera applikationen och ge dem kontroll och överblick över sitt arbete:

(6) *Ge användaren kontroll över systemet.*

Detta åstadkoms främst genom att behålla en låg koppling mellan GUI och resten av systemet. Det bör finnas tillgång till exakt information om vad som utförts gentemot andra entiteter i systemet. Men samtidigt erbjuda information som ger användaren en omedelbar överblick och förståelse:

(7) *Spegla verkligheten men premiera enkelhet och snabbhet före exakt avbildning.*
(8) *Förtydliga olika ”tillstånd” hos systemet.*

Dessa båda riktlinjer behålls i ursprungsskick men är underordnade behovet av kontroll. Det följer från vikten av att underlätta testningen, att man skall premiera enkelhet och snabbhet, men är fortfarande befogat att påpeka igen. Slutligen bör den sista riktlinjen som vi behåller formuleras om.

Se till att alla möjliga testfall är genomförbara.

På grund av att vi ser ett behov av att förenkla och prioritera till hög grad, är det stor risk att vissa testfall inte kommer att kunna genomföras. Men genom att följa (1) och bibehålla låg koppling mellan GUI och system så kommer det fortfarande vara möjligt att genomföra dessa testfall genom manuell hantering parallellt med att GUI:t används. Vi omformulerar:

(9) *Erbjud möjlighet att genomföra diversifierade tester*

De senaste tre riktlinjerna syftar alla till att ge användaren kontroll och överblick över testningen.

Strukna riktlinjer

De återstående två riktlinjerna väljer vi att styka från listan då de inte är tillräckligt specifika för GUI:s för testning utan är allmängiltiga för alla grafiska användargränssnitt. De uppvisade inte heller en nämnvärd positiv effekt på användbarheten, utan förbättringarna kunde tillskrivas andra riktlinjer.

Ge hela tiden användaren tillgång till relevanta verktyg.

Denna riktlinje är för generell och visade sig inte vara avgörande för att användbarheten förbättrades. Eftersom designen skall vara så enkel som möjligt är det svårt att inte erbjuda de rätta verktygen. Om man misslyckas med denna enkla uppgift kommer gränssnittet i och för sig bli helt värdelöst och oanvändbart. Men vi tycker inte det behöver understrykas mer än vad normal teori om GUI-design gör.

Tillmötesgå användarens utrymmesbehov.

Denna riktlinje visade sig vara beroende på mängden funktionalitet som införlivades i GUI:t och hur många verktyg som behövde användas parallellt med applikationen. Riktlinjen menar att GUI:t skall ta en så liten plats som möjligt, men detta är inte nödvändigt i alla fall. Om alla verktyg finns tillgängliga i gränssnittet kan hela fönstret användas. Det borde snarare formuleras som en anpassning till användarens utrymmesbegränsningar. Detta är en allmängiltig regel som vi inte specifikt behöver ta upp för våran typ av GUI.

Sammanfattning

Sammanfattningsvis kan vi konstatera att riktlinjerna i allmänhet fungerat väl, även om inte alltid med exakt förväntat resultat. Den inledande tolv riktlinjerna reducerades till nio, om än något justerade. Den största förändringen vi föreslår handlar dock om den strukturella synen på riktlinjerna. Vi frångår en enkel lista av riktlinjer där varje punkt väger lika tungt. Det nya förslaget utgörs istället av två huvudpunkter som utvecklas i ett antal underordnade riktlinjer på det sätt som redovisats nedan. Huvudpunkterna har även vidareutvecklas och konkretiserats något så att det budskap de förmedlar mer överensstämmer med undersökningen.

(1)
Underlätta återupprepning av tester genom att premiera enkelhet och snabbhet.

(2)
Betona det naturliga flödet.

(3)
Passivt stöd, dirigera istället för att diskutera.

(4)
Rationalisera bort oprioriterade moment.

(5)
Inkludera inte operationer och verktyg som redan har ett adekvat gränssnitt.

(6)
Ge användaren kontroll och överblick över systemet.

(7)
Spegla verkligheten men premiera enkelhet och snabbhet före exakt avbildning.

(8)
Förtydliga olika ”tillstånd” hos systemet.

(9)
Erbjud möjlighet att genomföra diversifierade tester

Vår undersökning visar att de riktlinjer vi teoretiskt formulerat, om än i reviderad form, även håller för en praktisk prövning. Om dessa riktlinjer följs vid utveckling av grafiska användargränssnitt för testning av inbäddade system för expertanvändare kommer applikationers användbarhet att avsevärt förbättras.

Litteraturförteckning

Ben-Menachem Mordechai & Marliss Gary S., *Software Quality – Producing Practical, Consistent Software*, International Thomson Computer Press, 1997.

Brooks Frederick P., *The Mythical Man Month – Essays on Software Engineering,* Addison Wesley Longnam Inc, 1995.

Cooper Alan, *About face the essential of user interface design*, IDG Books Worldwide, 1995

Dumas J S., Redish J C, *A practical guide to usability testing*, Ablex publishing corporation, Norwood, New Jersey, 1994

Easterby, Smith, et al, *Management Research*, 1991

Ekeroth Lars & Hedstöm Per-Martin, *GPRS Support Nodes* (Ericsson Review No.3, 2000, Vol. 77), Telefonaktiebolaget LM Ericsson, 2000.

Grøn Arne m.fl. Redaktör: Paul Lübcke, *Filosofilexikonet*, Bokförlaget Forum AB, 1988.

Harris Errol E., *Hypothesis and Perception*, George Allen & Unwin Ltd, 1970.

Krawczyk Henryk & Wiszniewski Bogdan, *Analysis and Testing of Distributed Software Applications,* Research Studies Press Ltd, 1998.

Hellevik Ottar, *Forskningsmetoder i sociologi och statsvetenskap*, Natur & Kultur, 1980.

Holme Idar Magne, Krohn Sovang Bernt, *Forskningsmetodik*, Studentlitteratur, Lund, 1991

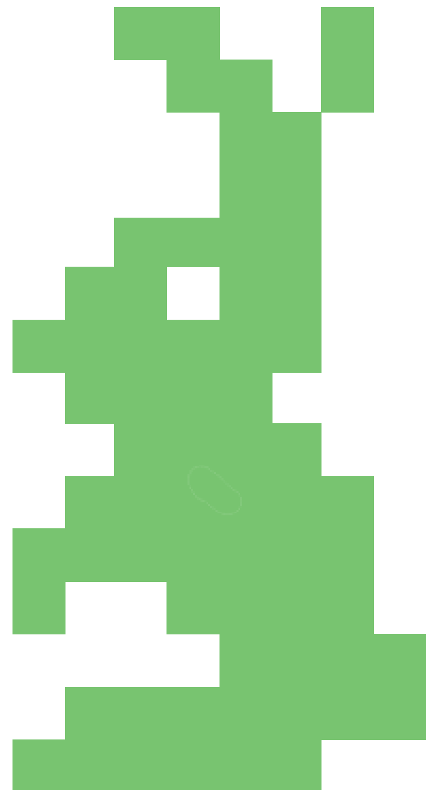
Leung Hareton K.N., *Quality Metrics for Intranet Applications* (Information & Management No.38), Elsevier Science B.V, 2001.

Lewis William E., *Software Testing and Continuous Quality Improvement,* CRC Press LLC, 2000

Sommerville Ian, *Software Engineering 6th Edition*, Pearson Education Ltd, 2001.

Van Vliet Hans, *Software Engineering – principles and practice*, John Wiley&Sons, Ltd, Chichester, England, 2000

Appendix 1 BUNNY DOCUMENTATION



BUNNY APPLICATION
CREATED BY JONA BOLIN - TORBJÖRN HÅKANSSON - PER JACOBSSON 2001
ARBETARKLUBBEN@TELIA.COM

Task

System definition

The *bunny application* provides a user-friendly interface towards the workstation solution (WSS). It constitutes a testing tool through which WSS easily and intuitively can be started, stopped and configured. For continuous development of DPE and WPP-applications it provides the right prerequisites for the user/programmer to test his/her applications or changes in a diverse and graspable way. The program runs on a UNIX-workstation and will be used by DPE-developers for debugging and unit testing. It contains the following functionality:

- Start and stop of WSS
- Node configuration
- Process monitoring (NCL, Eqma, Block)
- Termination of PM:s (Eqma)
- Termination of Block Instances
- Termination of NCL
- Activation of PM during runtime

Purpose

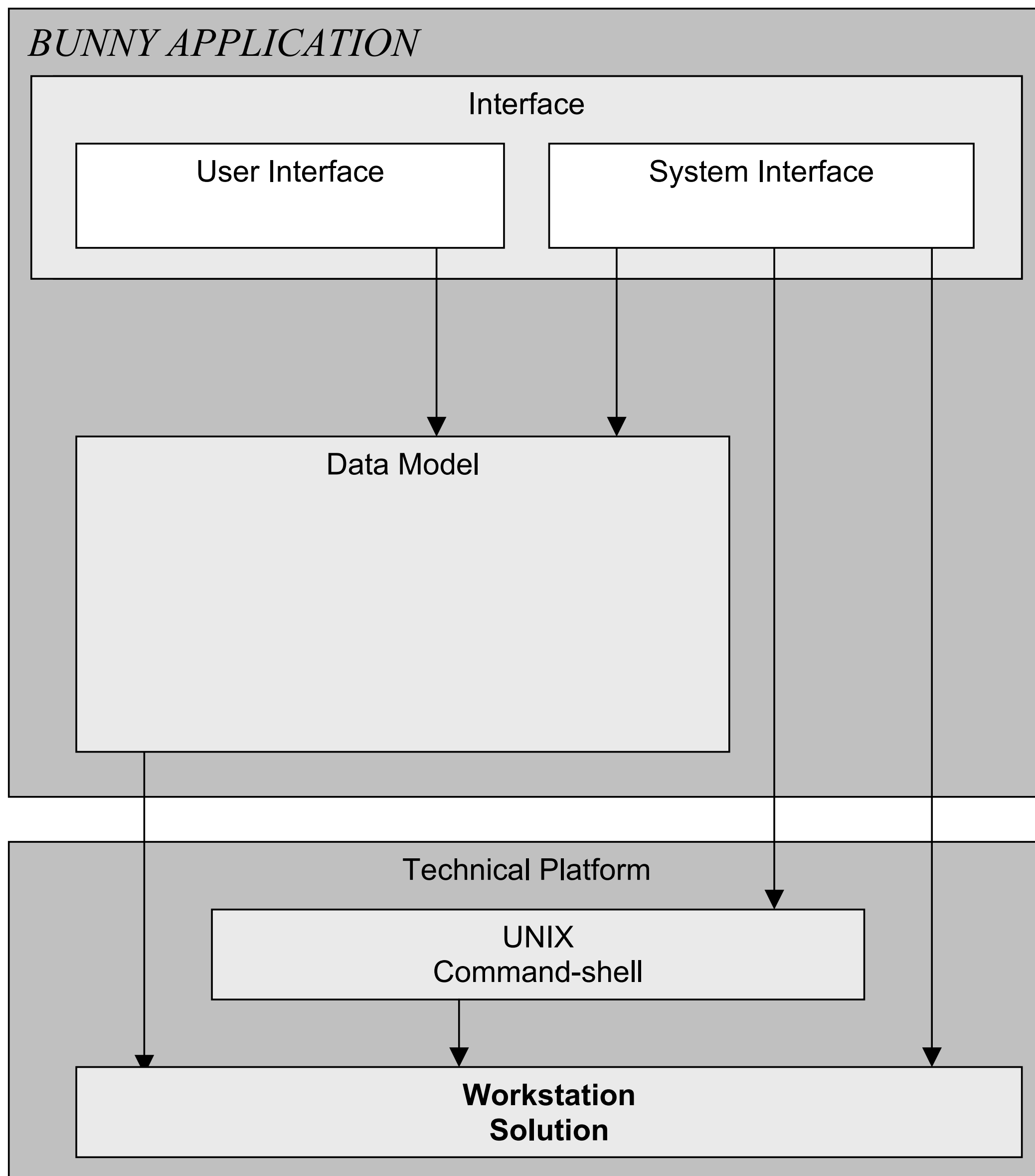
To make it possible for programmers to relatively easily test his or her applications or changes in the DPE. This is done on a UNIX-workstation in order to avoid the need of performing the testing on a real physical node, which demands time and costs.

Technical Platform

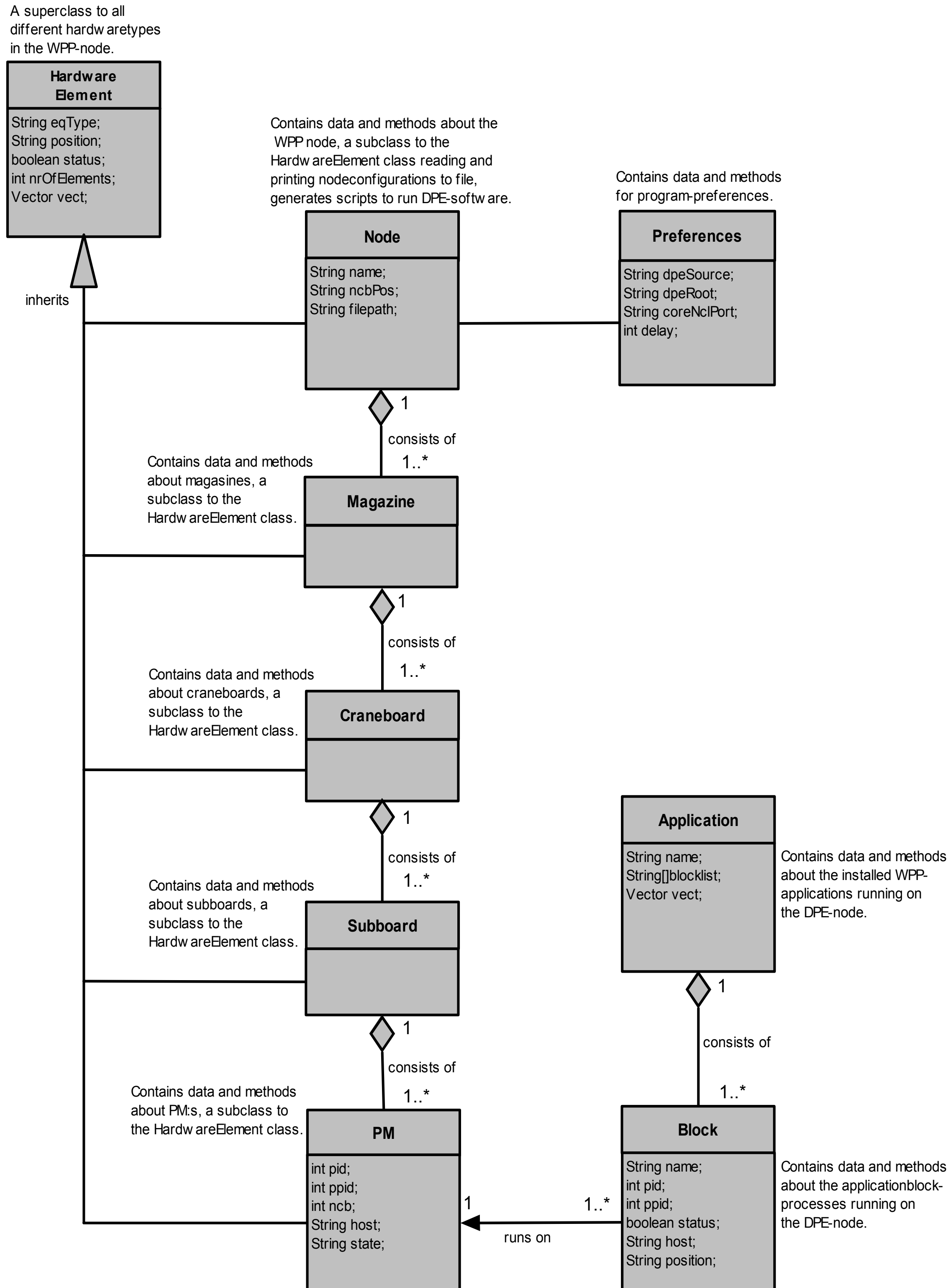
The program is developed on the Java 2 Platform using graphical components from the Javax.Swing package. The program application should be installed and run on a UNIX-workstation with a Solaris Sparc processing unit with JDK.1.1.x installed. In order to run the program successfully the user needs to install the Workstation Solution in a folder accessible to the application.

Architecture

The software design uses a layered architecture, where the interface components use the operations and data from the Data Model component. The System Interface includes functions that manipulate different files in the Workstation Solution and makes system calls to the Unix shell. The system calls executes commands in the shell that operates the scripts and processes related to the WSS. Operations in the Data Model component reads and prints data to files in WSS.



Data Model



Interface between GUI and the WSS

Description of how the GUI-application is connected to the Workstation Solution through processmanipulation and filehandling. The following document describes how files are manipulated and generated in the installation package of the Workstation Solution. Furthermore what type of system calls that are made to monitor and manipulate the processes related to DPE.

Preferences

When the application is started, the preferences are changed, or the PLAY-button is pressed and DPE is about to be initiated, the program checks if the preferences are correct.

Pref.checkDpeRoot( )

This function checks if the chosen filedirectory of the root of DPE contains the S20dpe_start.sh file.

Pref.checkDpeSource( )

This function checks if the s10-script has been executed and has created links in the /tmp-directory. It checks if the /tmp/DPE_ROOT directory exists and if not - the function returns a false boolean. If the directory is missing this signals that the s10-script should be executed

Pref.getCoreNclPortFromFile( )

This function extracts the CORE_NCL_PORT number from the Core.def file in /SiteSpecificData.

Pref.setCoreNclPortInFile( )

This function manipulates the Core.def file in /SiteSpecificData directory and updates the CORE_NCL_PORT number.

Get names of applications and blocks

When the preferences are checked and there is a correcttly installed DPE in the rootdirectory the application collects the names of the installed ADP-applications and their applicationblocks.

App.getAllApplications( )

This function checks every directory in /StoredLoadUnits. If there is a corresponding .blu-file in /StoredNCLData it creates a App-object and reads the revision name and the name of all the applicationblocks in the .blu-file. The application name is stored in the *name* variable and the blocknames are stored in an array of Strings, *blocklist*.

Start of DPE

When the START-button is pressed the application checks if the Node contains a PM with a position: x.x.2.1. If not the start is aborted and a error message is displayed for the user. Otherwise the nodeconfiguration is used to update the teie.dat and s20dpe_start.sh files.

Node.applyNodeConfig( )

This function controls that teie.dat and s20dpe_start.sh exists and saves them in temporary files. Then it initiates the update functions and if these are successful the temporary files are deleted, otherwise those temporary files are used to reset any changes made to the updated files.

Node.teieUpdate( )

This function updates teie.dat with rows, consisting of the positionnumbers and the name of the PM type, corresponding to every PM in the nodeconfiguration.

Node.s20Update( )

This function copies every row in the existing (source) s20dpe_start.sh file and prints it to a new (target) file until it finds a row starting with ”$EQMA” or ”POSITION”. When it finds the position it prints the selected position of the NCB. When it finds a row with eqma, the control-statements that initiates EQMA-wrappers are replaced with new statements in order to start a different EQMA-wrapper for every corresponding PM in the nodeconfiguration. When all the rows in the file has been read and copied, the source S20dpe_start.sh file is replaced by the new target file.
In short, the S20dpe_start.sh file is copied the way it is with changes made only to the initiation of EQMA-wrappers and the NCB-position.

DpeProcess.startDpe( )

This function begins to initiate functions to check the preferences and if these are ok, executes the S20dpe_start.sh script in the Unix shell in order to initiate DPE. It uses the Java.Runtime class and executes a command in the operative system shell, reads all the process output from the shell and waits for the command to finish. Finally it terminates the connection to the shell and passes the control over to the program.

Monitoring of processes

When DPE is initiated and running the status of all related processes are displayed for the user in the GUI. The processstatus is updated repeatedly every few seconds, as defined by the user in the preferences.

DpeProcess.checkProcess( )

This function is executed everytime the processes shall be updated. Initially all the blockinstances are deleted from the *App* objects and all the PM:s process id:s will reset and a PM that currently was running has its state set to “killed”. The reason for this is if the PM does not have any corresponding EQMA-process it has been killed and the rest of the PM:s will have its state changed later in the algorithm, i.e. as long as they haven’t been killed. Next, a system command is executed in the shell, “ps –o fname, pid, ppid, args”.
The function collects the process output from the ps-command, where every line shows data about one process, and checks the process against the PM:s and ncl. The position for a process can be found in the “args” data and is checked against the PM:s. When a hit is found the corresponding dataobject in the program is updated according to the output from the ps-command.
All processes other than “eqma_cp1”, “eqma_wra”, “ncl_wra” and “ncl” is checked against the blocknames for every application in the *App.blocklist*-array. If there is a corresponding

blockname for a process a *Block* is created and appended to the end of the *Vector* in the *App* object.
Finally when all the outputrows have been checked the applicationblocks are set to the PM it runs on.

App.setAppBlocktoPm( )

This function iterates through every *App*-object and all of their *Blocks* and checks the *Block.ppid* against every *PM.pid*, when there is a match the position of the *Block* is updated accordingly. A blockprocess usually has a an eqma-process as its parent process.

DpeTree.updateListsAndTables( )

When the data has been updated the table of PM:s and applicationblocks are updated. This is also true about the ncl status line.

Activating processes

When the user selects a PM that isn't currently running he can activate it.

NodeRuntime.unkill()

The selected PM:s are used to generate and execute a script that will initiate a corresponding eqma-wrapper for every selected PM. Execution is made through a system call to the unixshell.

Node.makeAddEqmaScript( )

This function generates a script that will initiate eqma-wrappers for every corresponding PM in the node. IF that PM is selected by the user AND its state is"killed" or "unknown".

Terminating processes

When the user selects a PM, *Block* or NCL that is currently running he can terminate it. All termination is executed through a system call to the unixshell, "kill –9 pid".

NodeRuntime.kill( )

This function makes a system call to the shell for every selected PM or *Block* if it is currently running and uses the process id to terminate it.

DpeProcess.stopDpe( )

This function generates a script that terminates all processes related to DPE. It uses the command statement "kill –9" for every process with a name like "candidat", "ncl", "eqma", "epmd" and whatever applicationblock that could be running.

GUI – classes

Xframe

Contains data and methods that defines and operates the GUI for the main window. When the program starts the X-class makes a call for it is this class. The following things in the GUI are defined in the Xframe class:

- The main menu
- The toolbar
- The tabbed pane
- The consol pane

All behaviors that those four objects are able to perform are trigged and handled in this class. The tabbed pane contains two sub ordered windows that also are classes – node configuration (NodeConfig) and node runtime (NodeRuntime) – those classes initiate by function calls from the Xframe class at startup.

NodeConfig

Contains data and methods that define and operate the GUI for the node configuration window. This window is sub ordered, placed in the tabbed pane and called by at startup by the Xframe class. Its main function is to display the tree structure containing magazines, crane boards and sub boards and the list with PM's. The following things in the GUI are defined in the NodeConfig class:

- The configuration tree structure that will contain magazines, crane boards and sub boards by creating an object of the DpeTree-class
- The configuration list that contains PM's
- The two popup menus displayed by a right button click on the mouse in the tree and the PM table.

NodeRuntime

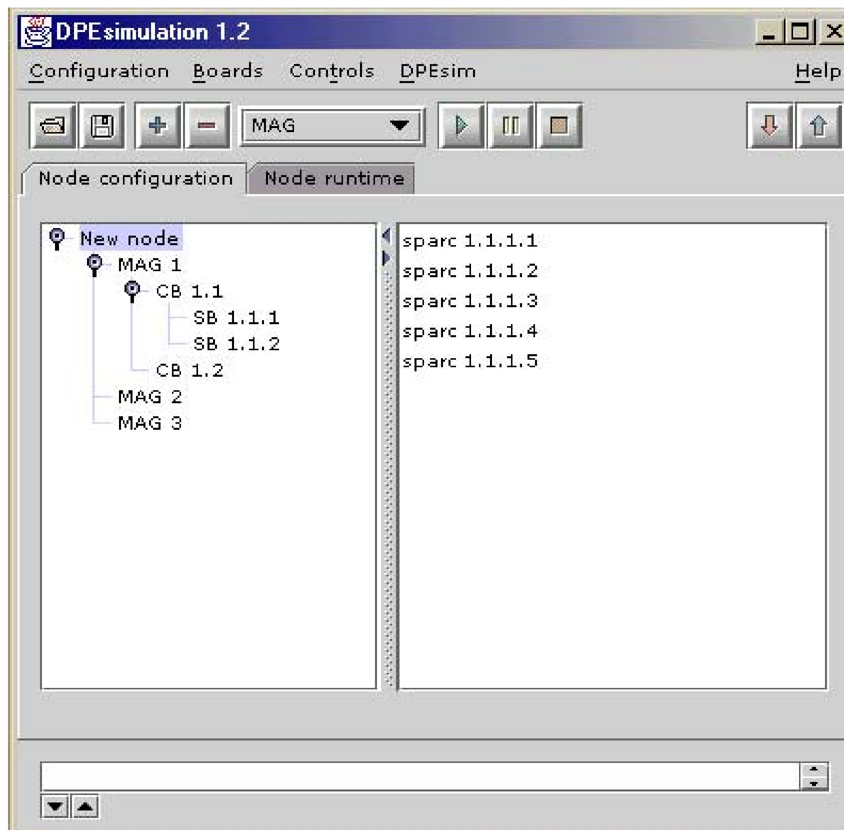
Contains data and methods that define and operate the GUI for the node runtime window. This window is sub ordered, placed in the tabbed pane and called by at startup by the Xframe class. Its main function is to display the tree structure containing magazines, crane boards and sub boards, a table with PM's and a table with applications. The following things in the GUI are defined in the NodeRuntime class:

- The configuration tree structure that will contain magazines, crane boards and sub boards by creating an object of the DpeTree-class
- The runtime table that contains PM's
- The application distribution table that contains the application blocks
- The three popup menus displayed by a right button click on the mouse in the tree, the runtime PM table and the application distribution table.

DpeTree

Contains data and methods that define and operate the tree in GUI for the NodeConfig- and the NodeRuntime-class. Whenever a tree is to be constructed an object of this class is created. The class contains methods that operates a tree, functions like:

- Add an object to the tree structure
- Remove an object from the tree structure
- Clear the tree structure from all objects
- A tree listener that calls a update function when selection has changed in tree



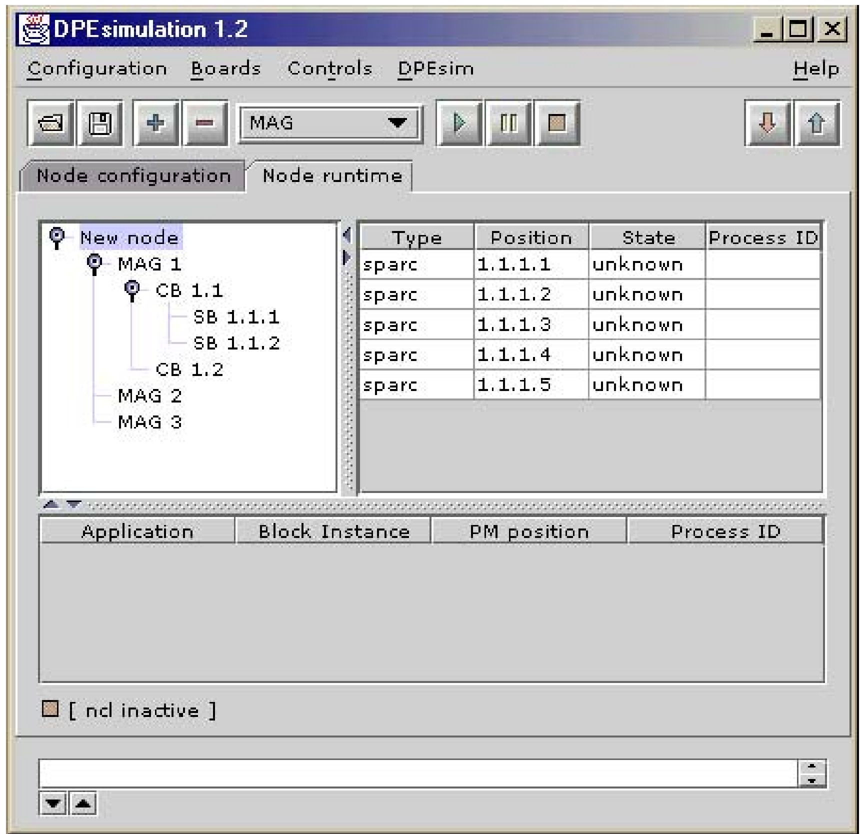
BUNNY APPLICATION

Class:
Xframe + NodeConfig + DpeTree

The node configuration view is an object of the NodeConfig class. Instanciated in the Xframe class.

The tree is an object of the DpeTree class instanciated in the NodeConfig class.

Other GUI-components created in the Xframe class



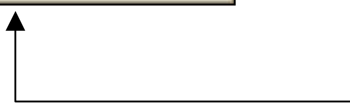
BUNNY APPLICATION

Class:
Xframe + NodeRuntime + DpeTree

The node runtime view is an object of the NodeRuntime class. Instanciated in the Xframe class.

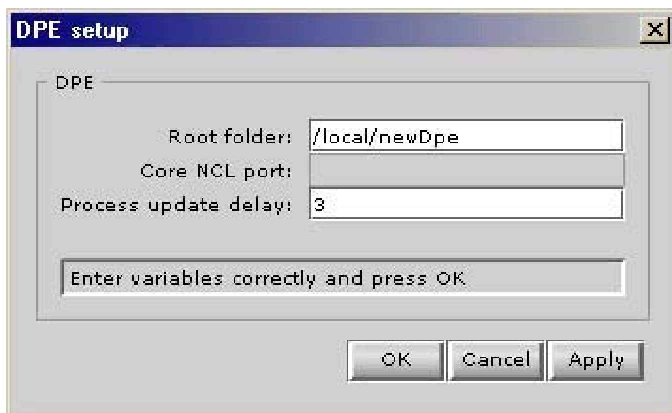
The tree is an object of the DpeTree class instanciated in the NodeRuntime *class.*

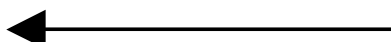
Other GUI-components created in the Xframe class



SetupDialog

Contains data and methods that defines and operates the GUI for the setup dialog window.





BUNNY APPLICATION

Class:
SetupDialog

All GUI-components created within class

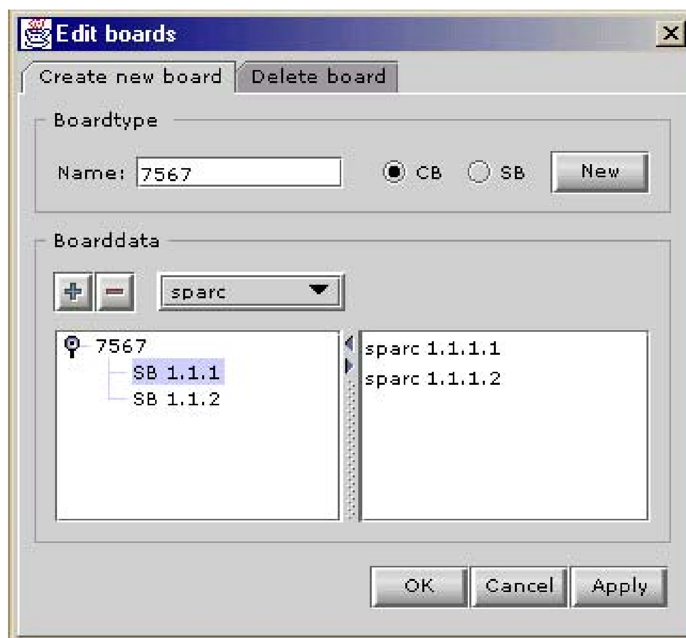
BoardDialog

Contains data and methods that define and operate the GUI for the board dialog window.

BoardTree

Contains data and methods that define and operate the tree in the BoardDialog. Whenever a tree is to be constructed an object of this class is created. The class contains methods that operates a tree, functions like:

- Add an object to the tree structure
- Remove an object from the tree structure
- Clear the tree structure from all objects
- A tree listener that calls update function when selection has changed in tree



BUNNY APPLICATION

Class:
BoardDialog + BoardTree

The tree is an object of the BoardTree class instanciated in the BoardDialog class.

Other GUI-components created in the BoardDialog class

Appendix 2 BUNNY MANUAL

Table of contents

1. BEFORE WE START
2. GETTING STARTED
3. OVERVIEW
    3.1 the menu
    3.2 the toolbar
    3.3 the node configuration window
    3.4 the node runtime window
    3.5 the console window
4. EDIT BOARDS
5. SHORT COMMANDS

1. Before we start

In order to understand Bunny and its help section the user must have knowledge of DPE and its environment. This help section does not give answers to questions like:

What is a craneboard?
How do GPB:s differ from IBE:s?
What´s the use of the S10dpe_links.sh-script?

These questions are all part of the DPE-fundamentals. Bunny will for example NOT assist the user with the installation of NDP and ADP, updating the S10-script or changing the cbd-file. These are all matters that must be taken care of before using Bunny. X-men do not take responsibility for any frustrated actions taken by users who lack necessary knowledge of the DPE-world.

2. Getting started

SETUP

The first time you start Bunny you must perform a proper setup. This is done by choosing DpeSim ->Setup in the menu. Here you will find 3 fields. One of these is empty. In the empty Root folder-field you must type the search path to the folder you have chosen to be root folder. In the Core NCL port-field you will find the same number as defined in the file /Sitespecificdata/Core.def. NOTE! This is only true if the Root folder is correct. The user can change this number at any time. Finally in the Process update delay-field you you can change the amount of seconds to elapse between every process-list update (1 is minimum, 10 is rather

a lot, 3 is nice.) When this is done - click OK. This procedure is only required once for every installation, so next time you start Bunny the setup is OK.

CONFIGURATION...

Now you have reached the point where you can start building your preferred node. This can be done in more than one way: You can either build a new one from scratch, or you can choose to open an already existing node configuration from file. Which ever method you might choose, you can modify your configuration by adding elements (magasines, craneboards, subboards and PM:s) until you are pleased.

...FROM SCRATCH

All configuration is performed in the Node configuration-window.

Magasines are added by marking the top level in the tree and clicking the "add-button" marked with a plus-symbol. You can also use the right-click function to add a magasine.

Craneboards are added by marking a MAG and clicking the "add-button". You can also add a craneboard by using the right-click function.

Subboards are added by marking a CR and clicking the "add-button".

PM:s are added by marking an SB and clicking the "add-button". The PM:s, as you will see, appear in the right window.

When you have reached the point of being satisfied with your configuration - click the PLAY-button. Now the start-script (S20start_dpe.sh) updates automatically and the distribution starts. In case you change your mind about your configuration you can use the Revert-function in the menu to go back to your latest saved configuration.

Deleting of elements is done by marking the element that is to be deleted, and either click the minus-button, use right-click or choose remove component from the menu.

...FROM FILE

If you want to open a saved configuration from file you can either use the OPEN-button, or you can choose Configuration->Open from the menu. Pick your preferred file and click OK. This file will now transform into a tree-structure. This structure can of course also be further modified as described above. The five most recent files can also be reached via Configuration -> Recent... in the menu.

START DPE

When PLAY is being pressed the distribution starts and DPEsim automatically switches over to the Node configuration-window. In this window the user can have full view of the distribution as well as the ability to interact with the configuration by adding or deleting elements in runtime.

STOP DPE

The STOP-button kills all processes.

3. Overview

This chapter explains the basic concepts of Bunny.

3.1 the menu...

Although using the menu is not the most efficient way to use the program, most of the functions can execute this way. This is often considered a nice way of learning the program before the user gets familiar with all the alternative ways such as right-clicks and short-commands.

Configuration

New
Creates a new empty node configuration.
Open
Opens an existing node configuration from file.
Save/Save As
Saves the current node configuration.
Revert
Switches to the latest saved configuration.
Recent files
Contains short-cuts to the 5 most recent node configurations.
Exit
Closes the program.

Boards

Edit
Creates a new type of board. (See chapter 4)

Controls

Add component
Adds a component to the node configuration. Magasines are added by marking the symbol N (N for Node) and choosing "Add component". You can also use the button marked with plus-symbol, use the right-click function or press Alt 1 to add a magasine. Craneboards are added by marking the M-symbol (M for magasine) and choosing "Add component". Other ways of adding craneboards are the same as for adding magasines. Subboards are added by marking the C-symbol(C for Craneboard) and choosing "Add component".Other ways of adding subboards are the same as for adding magasines. PM:s are added by marking the S-symbol(S for Subboard) and choosing "Add component". The

PM:s, as you will see, appear in the right window. Other ways of adding craneboards are the same as for adding magasines.

Remove component
Removes a component from the existing node configuration. You can remove any kind of element - magasine, craneboard, subboard or pm. Delete an element by marking it and choose "Remove component". You can also use the button marked with a minus-symbol, use the right-click function or press Alt 2 to remove an element.

Start DPE
With Start DPE the distribution starts and DPEsim automatically switches over to the Node configuration-window.

Stop DPE
The Stop DPE-button kills all processes.

Kill PM or block
Kills the PM or block that is marked.

Activate PM
Activates the PM that is marked.

Activate all PM:s
If the user has added many PM:s and wants to get all of them running - this is the function to use.

Kill NCL
This action terminates the NCL-process.

DPEsim

Setup
The first time you start Bunny you must perform a proper setup. This is done by choosing DpeSim ->Setup in the menu. Here you will find 3 fields. One of these is empty. In the empty Root folder-field you must type the search path to the folder you have chosen to be root folder. In the Core NCL port-field you will find the same number as defined in the file /Sitespecificdata/Core.def. NOTE! This is only true if the Root folder is correct. The user can change this number at any time. Finally in the Process update delay-field you you can change the amount of seconds to elapse between every process-list update (1 is minimum, 10 is rather a lot, 3 is nice.) When this is done - click OK. This procedure is only required once for every installation, so next time you start Bunny the setup is OK.

Help

Help contents
This is where you are right now.

About DPEsim

Contains information about the authors of DPEsim.

3.2 the toolbar...

The toolbar contains a set of buttons for the most common actions. The buttons are from left to right:

OPEN - This button is for opening an existing node configuration from file.
SAVE - This button is for saving an opened node configuration to file.
ADD ELEMENT - This button is for adding different types of elements to the node configuration tree. Magasines are added by marking the symbol N (N for Node) and clicking the "add-button". You can also mark the N-symbol, then use the right-click function to add a magasine. Craneboards are added by marking the M-symbol (M for magasine) and clicking the "add-button". You can also mark the symbol M (M for Magasine),then use the right-click function to add a craneboard. Subboards are added by marking the C-symbol(C for Craneboard) and clicking the "add-button". The right-click function works as above. PM:s are added by marking the S-symbol(S for Subboard) and clicking the "add-button". The right-click function works as above.
REMOVE ELEMENT - This button is for deleting elements. Just mark the element you wish to delete and click the "remove-button". You can delete elements on any level in the tree, which means that if a magasine is deleted - all of its craneboards, subboards and PM:s are also deleted.
BOARD COMBO-BOX - This combo-box contains all sets of elements that the user has predefined with the Create board-function (see chapter 4). Just select the desired element-set from the combo-box, and the click the same "add-button" as used for adding other types of elements.
PLAY - This button starts the DPE services.
PAUSE - This button automatically activates as soon as the user marks any field in the node runtime window. The reason for this is to prevent constant updating of tree and lists when elements are added or removed in runtime. The user must do the desired change, and press PAUSE again to update tree and lists. The user can also manually press down the PAUSE-key before doing the changes.
STOP - This button is for stopping the DPE services and killing all processes.

3.3 the node configuration window...

The node configuration window is the area in which all configuration activities prior to runtime take place, and it has two main windows. The left of these contains the node configuration tree model that the user creates, and the right window contains a list of all PM:s in the node configuration. For walkthrough see section about configuration.

3.4 the node runtime window...

As soon as the user presses down the start-button, the program automatically switches over to the node runtime window. Here the user can observe how the PM:s behave during runtime. The user can also see the process distribution, and interfere by killing/unkilling PM:s/processes and see the effects of these changes. This window has a different look. In the right hand window, the PM:s are listed with various additional information. PM distribution window:
Type - Can be either "sparc" or "ppc".

Position - Displays the PM:s position in the node.
State - Can have these different values:

"Unknown"(PM has not been added to the S20-script).
"Ready"(PM has been added to S20-script or an individual EQMA-script and is ready to get started).
"Starting"(EQMA-wrapper for PM has been initialized but is yet to start the "real" EQMA- process).
"Running"(The real EQMA-process for the PM has been started and is running).
"Killed"(The PM has been started and finally killed by terminating the EQMA-process).

Process ID - Contains the process ID (pid) for the PM:s.

3.5 the console window...

The user will find the console window in the bottom area of Bunny, and it produces output as soon as any changes has been done. It can be resized using the small arrow-buttons in the bottom left corner of the program - it can show 1, 2, 4 or 8 rows of information at the same time. Of course it also has a scrollbar so that the user can trace any specific change that has been made.

4 Edit boards

The user can store a variety of predefined boards which easily can be reached in a combo-box. Choose Boards -> Edit in the menu. In the window that appears the user can create different types of boards that can act as a kind of time-saving shortcuts instead of creating all the different elements one by one.

The first thing that has to be done is to give the board a name and decide whether it should be a Craneboard or a Subboard. When this is done and the user clicks "New" the board is created. After it is created the user can add components to any liking with the plus- and minus-buttons, and if it is a craneboard that is created all the different subboards that is already created are listed in the combo-box. This makes it even more easy to configure a craneboard in a fast way.

When you are pleased and want to add the created board to the main program´s combo-box - click Apply. Or click OK if you are finished with the board creation activities.

The Edit boards-window is also the place to be for deleting the predefined boards. Mark the board that you want to delete and click the minus-button. When enough boards are deleted - click Apply or OK. Since these boards often contain a massive amount of elements - there is also a regret-button in the form of an arrow-button.

5. Short commands

FUNCTION	COMMAND
Start	Ctrl-F1
Stop	Ctrl-F2
Add element	Ctrl-A
Remove element	Ctrl-R
Kill PM or block	Ctrl-K
Activate PM	Ctrl-U
Kill NCL	Ctrl-I
New configuration	Ctrl-N
Openconfiguration	Ctrl-O
Save configuration	Ctrl-S
Help	Ctrl-H