Introduktion till
PROGRAMVARUUTVECKLING

CLAES WOHLIN

Studentlitteratur

Claes Wohlin

Introduktion till programvaruutveckling

Studentlitteratur

Denna trycksak är miljöanpassad, både när det gäller papper och tryckprocess.

Art.nr 31291
eISBN 978-91-44-06583-0
Print ISBN 978-91-44-02861-3

Omslagsbild: Comstock
Omslagslayout: Pernilla Eriksson

Printed in Sweden
Studentlitteratur, Lund
Webbadress: www.studentlitteratur.se

Tryckning/år 1 2 3 4 5 6 7 8 9 10 | 2009 08 07 06 05

Till Margaretha, Carl och Eric

Förord

Denna bok är tänkt att fungera som en allmän introduktion till programvaruutveckling, utan någon egentlig hänsyn om du är intresserad av utveckling av tekniska system eller administrativa system. Tanken är att boken skall kunna fungera som en kartbok, det vill säga den skall presentera en översikt och ett sammanhang, där detaljkunskap om olika områden kan placeras in. Det innebär till exempel att mycket god kunskap inom programmering kan sättas in ett större sammanhang, det vill säga programvaruutveckling som helhet.

Vidare är boken planerad att kunna fungera antingen som en allmän introduktion till programvaruutveckling för den intresserade eller som kursbok. Boken är inte avsedd för en specifik utbildning, utan avsikten är att den skall gå att använda både inom tekniska utbildningar, till exempel i programvaruteknik och datavetenskap, eller i systemutbildningar, till exempel i systemvetenskap eller informationssystem.

Boken skall kunna användas på två olika sätt:

1 Boken kan användas som komplement till annan kurslitteratur. Den kan lämpligen kompletteras med olika böcker som ger ett djup inom ett visst område. Det betyder att boken kan användas i en programmeringskurs där programmeringsspråket hanteras i en bok och denna bok ger ett sammanhang för programmering, det vill säga studenten får en möjlighet att se hur kunskapen i programmering passar in i ett helhetsperspektiv för utveckling av programvarusystem. Boken kan även användas vid en fördjupade introduktion av något annat ämne till exempel design eller test.

2 Det är även möjligt att använda boken separat som en översiktsbok till programvaruutveckling. I detta fall kan boken eventuellt kompletteras med utvalda artiklar. Huvudmålsättningen är att

läsaren skall få en god översikt och förståelse för programvaruutveckling som en helhet.

Boken är indelad i fem delar. Den första delen ger en inledning och bakgrund till området. Den andra delen behandlar projektbegreppet och tar upp organisationen i vilken programvarutveckling sker. I den tredje delen presenteras de olika delar som brukar ingå i utvecklingen från kravhantering till underhåll av programvara, samt verktygsstöd i utvecklingen. Den fjärde delen fokuserar på det ingenjörsmässiga perspektivet genom en diskussion kring ingenjörens roll, kvalitet, mätningar och olika programvarutillämpningar. Femte delen är en avslutning. Den innehåller därmed en kort sammanfattning och något om trender inför framtiden.

Avslutningsvis är det värt att betona att innehållet i denna bok är baserad på 20 års erfarenhet av arbete, undervisning och forskning inom programvaruområdet och en genomgång av litteraturen på området. Förhoppningen är att erfarenheterna av vad som är viktigt skall underlätta för läsaren att få kunskap och förståelse för programvaruutveckling.

Jag vill rikta ett tack till alla personer som genom åren som har inspirerat mig att hela tiden lära mig ämnesområdet. Det innefattar speciellt alla arbetskamrater och kolleger vid Blekinge tekniska högskola, Lunds tekniska högskola, Linköpings universitet och de företag som jag arbetat vid/åt sedan 1983, då jag startade min bana inom storskalig programvaruutveckling.

Jag vill även tacka Viktor Jonsson, Inger Jänchen och Ronnie Andersson vid Studentlitteratur för samarbetet och hjälpen samt ett stort förstående när jag inte har lyckats med alla målsättningar avseende tidplanen. Slutligen vill jag tacka min familj för den förståelse ni har haft när jag vid olika tidpunkter har haft ett behov av att skriva. Tack!

3 juli 2005

Claes Wohlin

Innehåll

Del 1: Introduktion

Målsättningen med denna del är att ge en introduktion till området. Detta sker i två kapitel. Det första försöker fånga områdets betydelse och därmed motivera varför kunskap och kompetens inom programvaruområdet kommer att vara viktiga framöver. Vidare innehåller delen en historisk tillbakablick. Det är ofta viktigt att veta bakgrunden för att förstå nuet och kunna bilda sig en uppfattning om framtiden. Med anledning av detta finns ett kapitel som försöker beskriva utvecklingen inom datorområdet i allmänhet och inom programvaruområdet i synnerhet.

1 Inledning

Datorn uppfanns i slutet på 1940–talet och har sedan dess utvecklats oerhört snabbt. Det finns ingen utveckling som kan jämföras med de förbättringar som genomförts inom hårdvaruutvecklingen för datorer. Styrningen av hårdvaran sker av programvara eller mjukvara som det också kallas. Den senare har också utvecklats under åren från att inledningsvis bestå av stansade hål i kort till dagens webbprogrammering med objektorientering i form av till exempel Java.

Den oerhörda utvecklingen har inneburit inte bara utveckling av den dator som vi idag har på skrivborden utan även av mikrodatorer som idag finns i alltfler produkter. Detta har ökat betydelsen och vikten av kunskap om programvaruutveckling, vilket naturligtvis även har ökat behovet av utbildning och forskning inom området.

Inledningsvis var det oftast små program som skrevs för att göra en specifik funktion. Idag utgörs programvaran oftast av programvarusystem, det vill säga stora och komplexa programvaror som antingen säljs till specifika kunder eller till en marknad. Det förra kan exemplifieras med programvaror till mobiltelefoni, där operatörer kan köpa kundanpassad programvara till sina telefonisystem. Det senare kan exemplifieras med affärssystem för företag eller de spel som vi idag kan köpa i ett stort antal affärer. Dessutom finns det hybridformer till exempel generella affärssystem som anpassas till en stor kund som vill ha en speciell lösning. Det är dock även vanligt att företag idag anpassar sin verksamhet till affärssystemen.

Denna storskalighet innebär också att samspelet och samverkan mellan produkten (till exempel dess struktur), utvecklingsprocessen (arbetsmetoderna) och deltagande personer har blivit allt viktigare. Stora programvarusystem måste struktureras dels så att utveckling kan ske parallellt i olika delar och dels så att programvaran går att

vidareutveckla. Arbetsmetodiken är ett viktigt redskap för utvecklingen. Den beskriver hur utvecklingsarbetet skall genomföras och den skall fungera som ett stöd till utvecklarna. De skall inte behöva fundera på sin arbetsmetodik utan den innovativa kraften skall läggas på att göra en bra produkt, det vill säga med ett bra funktionellt innehåll och med rätt kvalitet. Slutligen är också personerna som arbetar med utvecklingen mycket väsentliga. Det är viktigt att de har rätt kunskap och kompetens. Det innebär inte att alla skall ha samma kunskap och kompetens, utan det är väsentligt att det finns rätt blandning. Experter kan behövas för vissa uppgifter. Ty det behövs personer med olika sorters kompetens för att ett utvecklingsprojekt som helhet skall fungera bra.

Principerna, metoderna och teknikerna för utveckling av programvara är oftast generella det vill säga oberoende av tillämpningsområde. Det finns många gånger en olycklig uppdelning i utveckling av tekniska system, till exempel inom telekommunikationsområdet, och administrativa system, till exempel affärssystem. Det finns med andra ord ingen större anledning att dela upp området som görs idag i Sverige mellan till exempel programvaruteknik och informationssystem åtminstone inte sett ur ett programvaruutvecklingsperspektiv. Skillnader finns naturligtvis, till exempel måste speciell hänsyn tas vid utveckling av ett säkerhetskritiskt system, men skillnaden ligger oftast inte i de generella principerna. Målsättningen med denna bok är att den skall introducera programvaruutveckling i generella termer med ett speciellt fokus på storskalig programvaruutveckling oavsett tillämpningsområde. Detta innebär att fokus är på programvaruprojekt med ett (stort) antal deltagare, det vill säga projekt där programmering utgör en del om än väsentlig.

Detta kapitel ger en kortfattad bakgrund till området. Nedan följer ytterligare motivering för behovet av den kunskap som förmedlas via denna bok. Vidare lyfts de aspekter fram som gör programvara unik som produkt. Det finns mycket att lära från annan produktutveckling, men samtidigt finns det ett antal aspekter som gör programvaran unik. En väsentlig aspekt som kommer att genomsyra boken är ingenjörsmässighet. Detta betyder dock inte att boken är riktad till ingenjörer utan ingenjörsmässig används som ett begrepp för ett strukturerat och systematiskt angreppssätt till programvaruutveckling. Ingenjörsmässighet är följaktligen ett viktigt förhåll-

ningssätt till programvaruutveckling. Detta är viktigt för alla som arbetar med programvaruutveckling oavsett bakgrund och utbildning. Det viktigaste är arbetsrollen. Kapitlet avslutas med en översikt av resten av boken.

1.1 Motivering

Programvara finns idag i allt fler produkter från leksaker till kärnkraftverk. Den finns i form av ”rena” programvaruprodukter som till exempel ordbehandlare och spel, men programvaran har även blivit en allt viktigare del av värdet i andra produkter. Det kan röra sig om programvara som implementerar olika typer av tjänster som till exempel de tjänster som tillhandahålls via Internetbanker och mobiltelefoner. Vidare finns det programvara där användaren oftast upplever att en annan produkt köps, men stora delar av värdet är i form av programvara. Detta brukar refereras till som varande inbäddad programvara, det vill säga programvaran är en del av en produkt och köparen upplever inte att hon eller han köper programvara. I vissa bilar finns idag programvara som tillhandahåller en hel del tjänster, till exempel kartfunktioner och hjälp för att hitta. Vidare finns det programvara som även bidrar vid styrning av grundläggande funktionalitet som till exempel broms- och styrsystem.

Programvaran har kommit att bli en allt viktigare del i många produkter tack vare den möjlighet det innebär att kunna lägga till, ändra och anpassa programvaran utan att ändra andra produktdelar. I många fall tillhandahåller företag olika produkter där skillnaden i huvudsak utgörs av olika versioner av programvaran. I vissa fall levereas till och med enbart en programvara, men genom inställningar i programvaran görs en viss funktionalitet tillgänglig. Detta är ett sätt för företagen att göra produktionsprocessen av till exempel hårdvara billigare.

Ovanstående betyder i korthet att utveckling av programvara har blivit en allt viktigare del av produktutveckling som helhet. Vårt fokus kommer dock enbart att vara på utvecklingen av programvaran. Programvaruutveckling bedrivs oftast i projektform. En anled-

ning är att storleken på programvaran genom åren blivit allt större. Det är inte ovanligt med programvaruprodukter som är flera miljoner rader kod. Programvaruprodukter används här genomgående som ett begrepp för all typ av programvara oavsett om det är en produkt i sig eller om den är inbäddad i en annan produkt som i bilar eller mobiltelefoner.

Det faktum att programvaran blivit allt större gör det omöjligt för enskilda individer att utveckla en programvaruprodukt. Vidare innebär det att det inte räcker med programmering. Det krävs mycket mer. Det är viktigt att klargöra exakt vad som skall göras och hur det skall göras samt vem som skall göra vad. Vidare måste det säkerställa att de krav, önskemål och förväntningar som finns antingen från en kund eller från en allmän marknad verkligen har uppfyllts. För att säkerställa en korrekt hantering av en stor programvaruprodukt krävs ett strukturerat och systematiskt sätt för utvecklingen av programvaran.

Storleken på dagens programvarusystem har gjort att det helt enkelt blivit för dyrt att utveckla programvara från början. Ofta byggs nya utgåvor baserat på existerande programvara, det vill säga programvaran vidareutvecklas. Livslängden på systemen innebär också att merparten av dem som var med om att utveckla den första versionen har gått vidare till nya arbetsuppgifter eller till andra arbetsgivare när produkten har funnits tillgänglig ett antal år. Det betyder att det finns ett stort behov av att andra kan förstå och lära sig programvaran. För att ge en känsla för storleken på programvarusystem idag kan vi göra en jämförelse.

Exempel:

Antag att en programvaruprodukt består av tre miljoner rader kod. Om detta jämförs med böcker. Låt oss anta att en bok (beroende av layout) består av 40 rader text per sida och att en normal bok har 300 sidor. Om vi nu har tre miljoner rader, hur många böcker bli det? I detta exempel innehåller en bok 40 * 300 rader text, vilket är 12 000 rader text. Antalet böcker blir följaktligen 250 böcker (3 000 000 / 12 000). 250 böcker innebär ett antal hyllmeter böcker. Om vi då tänker oss att dessa 250 böcker måste vara fullständigt konsistenta och gärna korrekta inser vi att detta inte är en lätt uppgift. Konsistent innebär till exempel att alla personer som beskrivs i

böckerna måste beskrivas på samma sätt i samtliga böcker. En person får inte ha blåa ögon i en bok och bruna ögon i en annan. Det är inte enkelt för författare att vara fullständigt konsistenta och det blir inte enklare av att de tre miljonerna raderna skrivs av ett stort antal personer. Å andra sidan fungerar det inte att en person skriver hela programvarusystemet, ty även om det vore möjligt att skriva en rad kod per minut (i snitt) skulle det innebära (med en årsarbetstid på 1600 timmar) att det tar mer än 30 år att utveckla programvaran och då är det stor risk att produkten får anses inaktuell. Det måste med andra ord till bra metoder och tekniker för att klara den utmaning som utveckling av programvarusystem innebär. Tilläggas skall att det är högt räknat med att skriva en rad kod i snitt i minuten i stora programvaruprojekt. Detta beror dels på tiden som räknas inte enbart är för programmeringen utan all projekttid räknas in och dels att det är en stor utmaning att utveckla stora system.

1.2 Unikt med programvara

Programvara är en produkt och det betyder att det även har mycket gemensamt med annan produktutveckling, till exempel vad avser att tillgodose kundernas krav. Å andra sidan skiljer sig programvara också väsentligt från andra typer av produkter. Här kommer vi i första hand att fokusera på det unika, men det får inte glömmas bort att det även finns mycket att lära från allmän produktutveckling.

En väsentlig skillnad är att utveckling av programvara innebär design. Vid utveckling av en ny bilmodell är den innovativa delen framtagning en prototyp och sedan följer produktionen. Den senare existerar i det närmaste inte för programvara. Mycket arbete har lagts på att förbättra produktionsprocessen vid produktutveckling. Det är dock inget som är direkt tillämpbart för programvara. Detta innebär också att problemen som ofta finns inom programvaruutveckling inte är överraskade om än irriterande. I programvaruutveckling är det alltför vanligt med missade tidplaner och budget, vilket kanske inte är konstigt med tanke på att hela utvecklingen är en innovativ process. Det är dock inte acceptabelt, utan det finns all

anledning att förbättra detta. Det är med andra ord en förklaring till problemen men ingen bra ursäkt.

En annan sak som gör programvaruutvecklingen svår är att produkten inte är direkt synlig. Det är klart att vi kan skriva ut dokument och kod, men vi ser inte produkten på samma sätt som om vi byggde en bro. Det är vid brobygge enkelt att bedöma hur långt vi har kommit, vilket definitivt inte är fallet vid programvaruutveckling. Detta betyder inte att andra projekt inte kan misslyckas och råka ut för problem. Ett tydligt exempel på detta är de problem som inträffat i samband med tågtunneln genom Hallandsåsen i västsverige. Utmaningen med programvara är att svårigheter verkar inträffa i alla projekt och en grundläggande orsak är att vi inte kan se och ta på produkten. Programvara exekveras och därmed observerar vi resultatet av exekveringen och inte programvaran själv. Det betyder också om att något fel inträffar är det konsekvensen av felet som observeras och inte felet självt.

Andra aspekter som är unika för programvaran relaterar till storleken, komplexiteten och livslängden som redan har berörts. Programvarusystem blir ofta komplexa. Det är oftast omöjligt att undvika komplexitet med tanke på hur mycket programvaran skall klara av och hur flexibel den skall vara. Det viktiga är att inte göra programvaran (onödigt) komplicerad. Komplexiteten kommer från storleken på programvaran och problemen som programvaran skall lösa. En aspekt i anslutning till detta är att programvaran måste vara konsistent. Även om programvaran har utvecklats av ett stort antal människor under en lång tid krävs det att det är en programvara och att alla delar fungerar tillsammans.

Den långa livslängden och kostnaden för att ändra på till exempel hårdvaran bidrar till programvarans sista unika egenskap, nämligen möjligheten att förändra den. Möjligheten att ändra programvaran är både en styrka och svaghet. Det är en styrka då det möjliggör lösning av många problem med programvara, speciellt utan att ändra på hårdvaran, det vill säga samma dator används men den beter sig annorlunda baserat på att programvaran är ändrad. Svagheten ligger i att det är frestande att uppdatera programvaran för mycket och för ofta.

Sammanfattningsvis är programvara unik jämfört med andra produkter. Det betyder inte att andra produkter inte också kan vara till exempel komplexa men utmaningen med programvaran är att den har följande fem egenskaper i alla programvaror som kräver mer än en utvecklare:

- Design
- Synlighet (eller brist på det)
- Komplexitet
- Konsistens
- Förändringsmöjlighet

De unika egenskaperna med programvara ställer stora krav på alla inblandade parter i programvaruutveckling. Speciellt ställs det stora krav på utvecklarna och projektledarna. För att hantera utmaningarna med programvaruutvecklingen krävs ett ingenjörsmässigt förhållningssätt.

1.3 Ingenjörsmässighet

Med ett ingenjörsmässigt förhållningssätt avses att ett antal aktiviter genomförs vid utvecklingen, speciellt inom ramen för projektet men även för organisationen som helhet. Det senare kan exemplifieras med att varje projekt kan hjälpa till med att förbättra utvecklingen i framtiden inom organisationen. I princip avses med ingenjörsmässighet att arbete sker med aktiviteter som till exempel:

- Förståelse
- Skattning
- Planering
- Uppföljning
- Utvärdering
- Förbättring

Med förståelse innebär att varje person som arbetar med programvaruutveckling har ett ansvar att förstå målsättningen med det arbete som utförs och relationen till övriga delar i systemet. Detta betyder inte att alla måste förstå alla delar av ett system i detalj utan att varje person måste förstå hur ens eget arbete relaterar dels till relevanta delar och dels till helheten. Om utvecklare inte förstår

detta är det stor risk att en enskild utvecklare kan göra något som bidrar till problem med programvaran längre fram.

Skattning är en viktig aktivitet. Det är mycket viktigt att kunna skatta arbetstid, både i form av antal arbetstimmar och i form av kalendertid. Det senare syftar på när det är klart. Detta är inte bara en aktivitet för projektledaren utan det är viktigt att skattning görs för varje del i utvecklingen och att varje utvecklare tar ansvar för skattning avseende sina ansvarsområden. Vid skattning av både arbetstid men framförallt kalendertid är det nödvändigt att det finns en förståelse för det egna arbetets del i en större helhet. Om en utvecklare inte förstår sin egen del finns det uppenbara problem, men det är också viktigt att förstå relationer och beroende till andra delar för att klara av att göra korrekta skattningar. Skattning är svårt och det är lätt att göra fel. Det är dock ingen ursäkt för att inte göra den bästa möjliga skattningen.

Om en utvecklare skall skriva ett program, hur lång tid tar det i arbetstid och när blir det klart givit övriga åtagande som vederbörande har? Det är inte tillåtet för utvecklaren att inte veta, utan det är varje utvecklares skyldighet att kunna ge ett svar även om svaret är ett intervall. Det är till och med så att ett intervall är väl så bra som en punktskattning då ett intervall även visar på den osäkerhet som finns. Tyvärr är det inte troligt att chefen uppskattar intervallsskattningar, men trots detta är det viktigt att visa på den osäkerhet som råder i en skattning.

En förutsättning för planering är skattning. Det finns ingen möjlighet att planera ett projekt eller annan verksamhet utan att ha skattningar. En projektledare måste ha skattningar för att kunna planera ett projekt och för att kunna bemanna ett projekt på bästa möjliga sätt för att nå de ofta högt ställda målen med ett projekt. Det är dock inte enbart projektledaren som behöver planera, utan det gäller även personer som har ansvar för vissa delar av systemet och enskilda utvecklare. Hur planerar jag mitt arbete så att jag bidrar på ett bra sätt till det gemensamma projektet?

Om olika typer av arbete eller projekt planeras innebär det också att det finns möjlighet att följa upp arbetet. Utan planering är det inte möjligt att göra någon vettig uppföljning. Det finns ingen möjlighet att veta hur långt utvecklingen har kommit utan planering och

uppföljning. Uppföljning innebär inte bara att det är möjligt att avgöra hur arbetet eller projektet ligger till i förhållande till den initiala planen, utan den utgör också en bas för att bedöma om, till exempel, projektet behöver planeras om. Den utgör grunden för beslut om mer resurser skall tillsättas, oftast i form av fler personer, eller om planeringen måste göras om. Det senare kan innebära att något datum flyttas eller rentav att målet med projektet förändras till exmpel i form av programvarans funktionalitet.

Uppföljning innebär också att olika saker bör utvärderas. I många projekt har något nytt introducerats, till exempel ett nytt utvecklingsverktyg eller en ny designmetod. För att kunna avgöra om introduktionen av det nya var positivt krävs det att utfallet utvärderas. Detta är viktigt då det måste finnas underlag för framtida förändringar. Det ingenjörsmässiga förhållningssättet innebär att det finns ett kontinuerligt fokus på att utvärdera det som görs och därmed även identifiera svaga punkter.

Identifikationen av svaga punkter leder sedan vidare till förbättringsmöjligheter. Om vi upplever att vi hittar alldeles för få fel när varje utvecklare testar sin del av programvaran och att dessa fel ställer till stora problem när vi kopplar samman delarna och testar dem, då har vi hittat en svag punkt. Denna svaga punkt kan adresseras genom att vi ser över metoderna och verktygsstödet för test som genomförs av den enskilda utvecklaren. Identifikation av nya metoder kan göras genom erfarenheter från andra projekt, tillgängliga metoder och verktyg på den kommersiella marknaden eller genom kontakt med forskningsorganisationer (interna eller externa).

Efter förbättring är det sedan viktigt att gå tillbaka till förståelse igen, det vill säga vi måste kunna förstå konsekvensen av varje förbättring och dess samverkan med andra delar i utvecklingen. Antag att vi förbättrar testsidan som beskrevs ovan. Vad innebär det för andra aktiviteter? Är det lämpligt att även ändra andra testaktiviteter? De kortfattat beskrivna punkterna ovan är att betrakta som en kontinuerlig process. Det finns alltid något som kan göras annorlunda, speciellt som nya metoder, tekniker och verktyg utvecklas kontinuerligt för att stödja programvaruutvecklarna.

Sammantaget innebär aktiviteterna ovan att vi har ett ingenjörsmässigt förhållningssätt till utveckling av programvara. Det är möjligt att lägga till andra aktiviteter, men ovanstående ger ett bra exempel på vad som genomgående kommer att avses med ingenjörsmässighet.

1.4 Översikt av boken

Boken består av 16 kapitel uppdelade på fem delar. Den första delen ger en introduktion och en kortfattad bakgrund till utvecklingen på området. I Del 2 presenteras de två huvudperspektiv som finns i en organisation i vardera ett kapitel. Organisationer är oftast uppbyggda i form av en matrisstruktur med en projektorganisation och en linjeorganisation. Inledningsvis diskuteras aktiviteter vid genomförande av ett enskilt programvaruutvecklingsprojekt. Därefter beskrivs organisationen och dess utvecklingsprocess samt kvalitetsarbete på organisationsnivå. Det senare kapitlet innehåller bland annat beskrivningar av olika typer av utvecklingsprocesser.

Den tredje delen tar upp de olika aktiviteterna i utvecklingsprocessen. Det finns många olika benämningar på de olika aktiviteterna. Här kommer följande uppdelning att användas: krav, design, implementation, granskning, test och underhåll. Det är viktigt att notera att i designkapitlet diskuteras all design från övergripande arkitektur till mer detaljerad design. Utöver aktivitetskapiteln innehåller delen också ett kapitel om utvecklingsverktyg. Här är det viktigt att notera att programmering är enbart en liten del av programvaruutveckling. Programmeringen görs som en del i implementationen. Det är viktigt att skilja på programvaruutveckling och programmering. Det är lätt att tro att det är möjligt att utveckla programvarusystem baserat på kunskap i programmering. Programmeringskunskap är en nödvändig, men inte tillräcklig kunskap för att bli en god programvaruutvecklare.

Del 4 fokuserar på ingenjörsmässighet och de kompetenser, egenskaper och aktiviteter som behövs för att kunna hävda att arbetet bedrivs på ett strukturerat och systematiskt sätt. Detta görs först i ett kapitel som benämns ingenjörsmässighet. Här syftas på att arbetet

bedrivs på ett systematiskt och strukturerat sätt, det vill säga det betyder inte att personen måste ha en ingenjörsutbildning. Namnet på kapitlet är valt för att betona det ingenjörsmässiga förhållningssättet och avsikten är inte att fokusera på utveckling av tekniska system eller att fokus skall ligga på ingenjörsutbildningar. Delen avslutas med två kapitel som lyfter fram två viktiga aspekter för det ingenjörsmässiga förhållningssättet, nämligen kvalitet och mätningar. I kapitlet om kvalitet beskrivs olika metoder för kvalitetsförbättring, där kvalitet syftar på tre kvalitetsperspektiv: kunden eller användaren, management och utvecklaren. Dessa tre perspektiv måste balanseras på ett bra sätt för produkten skall vara konkurrenskraftig. För att få grepp om alla kvalitetsaspekter från programvarufel till arbetstid behövs mätningar. Med anledning av detta följs kapitlet om kvalitet av ett kapitel om mått och modeller för mätningar av programvara. Delen avslutas med ett kapitel som diskuterar några olika tillämpningsområden, där programvaran spelar en väsentlig roll.

Boken avslutas med en femte del som kortfattat sammanfattar och pekar på framtida utvecklingen inom området.

2 Bakgrund

2.1 Datorns funktion

Dagens datorer bygger på matematik som utvecklades under 1800-talet av den engelske matematikern George Boole. Matematiken han utvecklade var en form av logisk algebra. Idag benämns den ofta Boolesk algebra. Logiken representeras av ettor och nollor som ligger till grund för digital- och datatekniken. Boole utvecklade en matematik där alla positiva heltal kunde representeras av ettor och nollor. Han började med att notera att noll och ett kan representera av sig själva, men för att representera talet två behövs två tecken. Han valde att skriva detta som 10, där ettan skall tolkas som att talet innehåller en tvåa och nollan representerar sig själv. På samma sätt skrevs talet tre som 11, där den första ettan representerar en tvåa och den andra ettan en etta, vilket tillsammans blir tre. Alltså kan alla tal från noll till tre representeras av två positioner som kan anta talen noll respektive ett. Med tre positioner kan alla tal från noll till sju representeras som följer: 0 = 000; 1 = 001; 2 = 010; 3 = 011; 4 = 100; 5 = 101; 6 = 110; och 7 = 111.

Dessa tal kallas även binära tal, då de enbart innehåller nollor och ettor. Boole noterade också att varje position representerar talet 2^n där *n* går från 0 och uppåt, det vill säga första positionen representerar huruvida talet ett ingår i det tal som skriva binärt. De följande positionerna representerar talen två, fyra, åtta, sexton och så vidare. Rent principiellt innebär detta att ett godtyckligt stort heltal kan representeras av enbart ettor och nollor. Boole definierade vidare hur det är möjligt att räkna med denna typ av representation, det vill säga hur binära tal adderas respektive subtraheras. Vid addition läggs varje position samman från höger. Noll plus noll blir noll, ett plus noll blir ett och ett plus ett blir noll med ett i minne till närmaste position till vänster. Ett enkelt exempel, antag att vi vill räkna

ut sju plus sju. Detta representeras av 111 + 111. Om vi börjat längst till höger så blir ett plus ett noll och vi får en etta i minne till nästa position. Det betyder att för den andra positionen skall tre ettor läggas samman vilket ger en etta och en etta i minne. Resultatet blir i binär form 1110, det vill säga en åtta, en fyra och en två, vilket adderat blir 14. Avsikten här är inte är lära ut Booelsk algebra, utan att visa hur denna algebra ligger till grunden för uppbyggnaden av dagens datorer.

Inom datortekniken används benämningen bit för att beskriva en position enligt den Booleska algebran. En bit får inte blandas samman med en byte som består av åtta bitar. Även om den Booleska algebran tillåter oändligt stora tal finns det fysiska begränsningar i datorer. En dator kan representera ett visst antal bitar och därmed sätts det begränsningar på hur stora tal som kan representeras i en dator. Olika datorer har olika begränsningar. Dessa begränsningar är viktiga att känna till då en dator skall programmeras, åtminstone om programmet som skall skrivas skall kunna representera stora tal. Antag att du skall skriva ett program för beräkning av $\binom{n}{k}$, då vet vi att:

$$\binom{n}{k} = \frac{n!}{(n-k)! \times k!}$$

men om *n* blir stort kommer *n!* att bli mycket stort mycket snabbt. Det finns med andra ord mycket stor risk att datorn inte kan representera så stora tal. Om vi förenklar problemet och antar att datorn enbart kan representera fyra positioner, det vill säga tal upp till 15. Antag vidare att vi vill beräkna $\binom{4}{2}$, hur gör vi? Svaret är sex, vilket är klart under 15, men samtidigt är *4!* större än 15. Hur löses detta fall och vad är därmed den generella lösningen på hur detta skall lösas för att datorn skall klara av att representera alla tal i beräkningen? Detta triviala exempel visar på hur viktigt det är att använda datorns representation på ett smart sätt och att ha en förståelse för datorns uppbyggnad och hur den representerar information.

En nackdel med att enbart skriva tal med nollor och ettor är att talen snabbt blir mycket långa. För att minska detta problem har olika typer av kodning introducerats. Ett exempel är hexadecimal kodning. Många som har köpt programvara har sett hexadecimal

kodning utan att kanske notera det. För många programvaruprodukter skall anges en kod för att kunna genomföra installationen. Denna kod kan bestå av till exempel fem fält med vardera fyra tecken. Det betyder att 20 tecken skall skrivas in. Tecknen som skall skrivas in är ofta siffror (0–9) och bokstäver (A–F). Dessa sexton (varav namnet hexadecimal) tecken representerar vardera fyra positioner i det binära talsystemet. Det betyder att 0–9 representerar sig själv, medan A–F representerar talen 10–15. Det betyder att när den som skall installera programmet skriver *D* tolkas detta av datorn som 1101. Kodningen är vald så att alla tal upp till 15 kan representeras av ett tecken, det vill talen 0–9 eller A–F. Det viktiga är att den som installerar programmet behöver inte veta om kodningen, utan enbart skriva in det som förväntas och sedan tolkar datorn informationen på ett korrekt sätt då de som utvecklat installationsrutinen har tagit hand om översättningen från hexadecimalt till binärt. Datorn känner alltså inte till något annat än ettor och nollor och därmed måste det till en översättning.

På liknande sätt representeras alla tecken som finns på tangentborden. Datorn vet inte något om olika bokstäver, utan den är programmerad att tolka en nedtryckning av till exempel ett *A* som en sträng av ettor och nollor. Den kodning som används är så kallad *ascii*-kodning. Det finns både sju och åtta bitars *ascii*-kodning och beroende på vilket som används kan en mängd olika tecken representeras i form av ettor och nollor.

Det finns en mängd olika sätt att koda information. Kodningen används sedan både i datorn och mellan datorer, till exempel när e-mail skickas mellan datorer. Vid överföring mellan datorer, till exempel i ett lokal datanät (LAN) eller via Internet, införs ofta säkerhetsbitar, det vill säga bitar som används för att kontrollera att den överförda informationen är korrekt. Detta är gjort på ett liknande sätt som svenska personnummer, där den sista siffran är en kontrollsiffra som kan beräknas från övriga siffror. Om den sista siffran inte stämmer vid beräkningen är det uppenbarligen något fel med det angivna personnumret. På liknande sätt kan felaktigheter vid elektronisk överföring identifieras och beroende på hur avancerade koder som används kan olika mycket slutsatser dras. Vid enkel kodning som med personnumret kan enbart slutsatsen dras att något är fel, medan vid mer avancerade koder kan även vissa typer av fel kor-

rigeras. Val av kodningsmetod görs baserat på risken för fel, normalt värde på informationen och kostnaden för en mer avancerad kod. Mer avancerade koder tar mer plats i form av fler bitar, vilket i sin tur minskar överföringshastigheten för den riktiga informationen då en del av kapaciteten tas i anspråk av koden.

Ovanstående är i första hand en fråga om binära tal och representation av data med olika typer av kodning. Nästa steg i förståelsen är att förstå hur detta kan överföras till elektronik. Det mest grundläggande elementet är en grind som kan byggs upp av till exempel transistorer. En grind har ett antal ingångar och en utgång. En grind bygger på att det kan finnas spänning eller inte på ingångarna och beroende på grindtypen finns det spänning på utgången. Spänning eller inte spänning används vidare för att representera ettor och nollor elektroniskt. Det finns fem typer av grundläggande (med två ingångar) grindar:

- AND-grind: Spänning ut om det finns spänning på båda ingångarna.
- OR-grind: Spänning ut om det finns spänning på minst en av ingångarna.
- XOR-grind: Spänning ut om det finns spänning på antingen den ena eller den andra ingången, dock inte båda.
- NAND-grind: Detta är en negerad eller inverterad AND-grind, det vill säga den ger spänning ut i alla fall utom när det är spänning in på båda ingångarna.
- NOR-grind: Detta är en negerad eller inverterad OR-grind, det vill säga den ger enbart spänning ut när ingen av ingångarna har någon spänning.

Det finns grindar med fler ingångar, men principen är densamma. Grindarna i sin tur används för att bygga upp kretsar. Ett exempel på en krets kan vara för att ett antal grindar sätts samman för att styra ett trafikljus. Kretsen styrs sedan av instruktioner för hur trafikljus fungerar, det vill säga vilka kombinationer av lampor som är tillåtna och i vilken ordning de skall tändas och släckas. Detta lilla exempel kan utökas till att inkludera en fyrvägskorsning med ljussignaler för både trafik och gående. Det är också möjligt att utöka

exemplet till att inkludera separata ljussignaler för svängande trafik. I detta exempel ser vi hur ettor och nollor representeras elektroniskt och hur detta i sin tur kan användas för att styra något. En viktig del i detta är naturligtvis instruktionerna.

Instruktionerna görs i form av ett program, vilket skrivs i ett programmeringsspråk. I grund och botten är ett programmeringsspråk bara ett för människan enklare sätt att representera de instruktioner till elektroniken i form av spänning (ettor) eller inte spänning (nollor) i grindar. Datorn förstår enbart binär information. Å andra sidan är detta en dålig form ur ett mänskligt perspektiv. För att hantera detta har människan utvecklat ett antal representationsformer i form av programmeringsspråk och översättare. De senare har utformats för att kunna översätta från ett programmeringsspråk till den form som datorn förstår. Den lägsta formen av programmeringsspråk är olika typer av assemblerspråk eller maskinnära språk. I dessa språk måste programmeraren hantera olika delar av maskinvaran explicit till exempel i form av att det måste anges var information skall lagras och så vidare. Denna nivå har mer eller mindre övergivits idag, även om det kan finnas fall när assemblerprogrammering behövs på grund av till exempel extrema prestandakrav. Idag används i första hand det som kallas högnivåspråk och sedan sköts översättning till binär information av datorn med hjälp av kompilatorer (översättare från högnivåspråk till lägre nivåer) och andra verktyg. Kompilatorn identifierar också fel, det vill säga fel som inte stämmer med definitionen av språket för vilket kompilatorn är byggd. En kompilator kan dock inte identifiera logiska fel.

Programmen som skrivs för att instruera datorn behöver köras eller exekveras av datorn, vilket görs i en processor (ofta kallad CPU för Central Processing Unit). Hastigheten med vilken datorn exekverar instruktionerna beror av klockfrekvensen, det vill säga hur snabbt saker och ting kan ske. Denna hastighet på processorn anges i Hertz. Idag pratar vi om hastigheter på gigahertz, det vill säga mer än 10^9 instruktioner per sekund. Det räcker dock inte att kunna exekvera instruktionerna snabbt utan det behövs även minne för att spara information.

Det finns två huvudtyper av minne: primär- respektive sekundärminne. Primärminnet eller arbetsminnet används för att spara

information kortsiktigt, det vill säga information som behövs för att till exempel köra ett program. Det sekundära minnet är för långsiktigt sparande och representeras av datorns hårddisk. Storleken på respektive minne avgör dels hur snabbt exekveringen kan genomföras och dels hur mycket som kan lagras. Det är med andra ord inte enbart processorns hastighet som avgör hur snabbt det går utan även storleken på primärminnet, då det tar väsentligt längre tid att hämta information från sekundärminnet (disken).

För att underlätta användandet av datorns hårdvara finns ett operativsystem som tar hand om den interna kommunikationen och funktionerna i datorn. Tanken är att användaren i första hand skall arbeta med den tillämpning som är av intresse utan att behöva bry sig om datorns interna uppbyggnad. Operativsystemet blir därmed länken mellan instruktionerna i form av ett program och maskinvaran. En dator idag sätts ofta samman av delar från olika tillverkare, till exempel någon utvecklar ljudkort och någon annan grafikkort. Som klister mellan dessa olika delar och operativsystemet finns drivrutiner, det vill säga mindre program som skall underlätta användandet av de olika delarna i datorn.

Datorns huvudkomponent är processorn och runt den finns ett antal hårdvarukomponenter i form av till exempel grafikkort och skärm samt kommunikationsmedia mellan komponenterna. För att hantera dessa hårdvarukomponenter finns operativsystem och drivrutiner. Användaren utvecklar eller använder sedan program (instruktioner) till datorn som körs på processorn och utnyttjar minnena som finns. Samtidigt skall noteras att på den mest primitiva nivån är det ”bara” frågan om spänning eller inte spänning, det vill säga ettor och nollor.

2.2 Historik

2.2.1 Datorns barndom

Datorns historia är svår att beskriva, då det i stor utsträckning blir en fråga om vad vi menar med begreppet dator. Ett antal vetenskapsmän och -kvinnor har arbetat med räknemaskiner. Dessa kan ses som föregångare till dagens datorer. Pascal utvecklade under 1600-talet en additionsmaskin. Som framgår av namnet kunde denna maskin kunde enbart hantera addition. Leibniz vidareutvecklade idén och byggde en maskin där operatören kunde välja mellan olika algoritmer. I likhet med Pascals maskin var logiken (algoritmen) inbyggd i arkitekturen av maskinen, det vill säga vi skulle idag säga att det var en hårdvarulösning.

Den första programmerbara räknemaskinen utvecklades av Charles Babbage under 1800-talet. Hans maskin styrdes av kort med hål. Hålen i korten bestämde vilka funktioner som maskinen skulle utföra. Babbage lösning hade således separerat hårdvaran (maskinen) från logiken (hålkorten). Han var dock inte först med att dela upp maskin från logik på detta sätt. Jacquard hade 1801 använt ett liknande angreppssätt för att styra vävstolar. Babbage var dock först med att utveckla en mer generell räknemaskin, då Jacquards maskin var avsedd för ett enda ändamål, det vill säga att styra vävstolar. Skillnaden mellan Jacquards och Babbages lösningar visar tydligt på skillnaden mellan lösningen av ett problem och en mer generell maskin. Vi ser samma uppdelning idag med generella datorer och datorbaserade lösningar för ett specifikt ändamål. I det senare fallet kan det finnas specialanpassad hårdvara eller programvara. Specialanpassning är vanlig inom många industrier, till exempel inom bilindustrin när en specifik lösning tas fram för en funktion hos en specifik modell eller för ett bilmärke. Generella datorer och programvara ser vi idag överallt.

Babbage hade en kvinnlig assistent vid namn Augusta Ada Byron som var dotter till den stora poeten Lord Byron. Hon räknas idag som världens första programmerare. För att hedra några av dessa pionjärer har några programmeringsspråk erhållit sina namn efter

pionjärerna. Ett exempel är Pascal som är ett programmeringsspråk som lanserades under 1970-talet. Språket utvecklades av en av de stora inom datavetenskapen nämligen Niklaus Wirth. Den första programmeraren har också givit namn åt ett språk nämligen Ada. Ada är ett språk utvecklat baserat på ett initiativ av det amerikanska försvarsdepartementet. Målsättningen var att utveckla ett bra språk för realtidssystem. Språket var avsett att bli en standard för alla programvarusystem som utvecklades för det amerikanska försvaret.

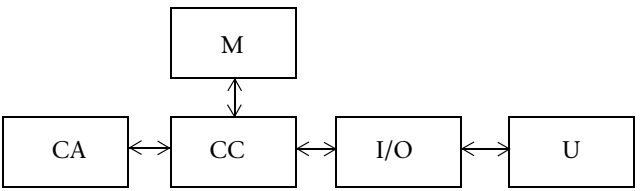
Det fanns många andra som bidrog under dessa århundraden och lade grunden för att utvecklingen kunde ta fart under 1900-talet och leda fram till de datorer vi ser idag. En förutsättning för den vidare utvecklingen var upptäckten av elektriciteten under 1800-talet och sedermera framväxten av elektroniken. Utvecklingen av glödlampan under senare delen av 1800-talet med Thomas Alva Edison i spetsen lade grunden till den moderna elektroniken. Ungefär samtidigt som utvecklingen av glödlampan uppfanns också telefonen. Den senare utvecklades parallellt av Alexander Graham Bell i USA och Lars Magnus Ericsson i Sverige. De allra första användbara telefonerna i USA installerades 1877 av Bell. I Sverige startade Ericsson 1876 utveckling och produktion av telefonkonstruktioner. Telefonen får anses vara grunden för dagens moderna analoga och digitala kommunikation.

Baserat på räknemaskiner som Babbage, upptäckten av elektricitet och utvecklingen av telefonen har under 1900-talet datorn kunnat utvecklas. Det är intressant att notera att utvecklingen av datorn bygger på ett antal saker, det vill säga en kombination av olika idéer och tidigare framsteg. Ett exempel på detta är Herman Hollerith som 1890 använde idén att representera information som hål (hålkort) för att snabba upp en folkräkning i USA. Detta arbete av Hollerith ledde sedermera till skapandet av IBM (International Business Machines).

Den moderna datortekniken får anses ha sin början på 1940-talet. Detta trots att tysken Konrad Zuse under 1930-talet konstruerade en programmerbar räknemaskin. Maskinen fungerade dock aldrig som avsett. Zuse fortsatte sitt arbete och bytte ut de mekaniska komponenterna mot elektroniska reläer. Denna konstruktion betraktas av många som den första datorn och den var klar 1941. IBM blev inte

klara med sin motsvarighet förrän 1944. IBM:s första dator döptes till Mark I. Andra är av uppfattningen att både Zuses konstruktion och Mark I inte är datorer utan i första hand räknemaskiner.

På 1940-talet presenterade von Neumann en logisk beskrivning av en dator. Hans beskrivning var den första beskrivningen av den logiska uppbyggnaden av en dator. Idag kan den verka enkel, men på sin tid var den genial i sin enkelhet och än idag fungerar den som en schematisk bild över hur en dator är uppbyggd. von Neumanns logiska beskrivning av en dator återfinns i Figur 2.1. Beskrivningen innefattar en Central Arithmetic Unit (CA), en Central Control Unit (CU), minne (M), en enhet för in- och utdata (I/O) och användaren (U). von Neumanns beskrivning låg till grund för utvecklingen av en dator vid University of Pennsylvania. En alternativ kandidat till Mark I eller Zuses konstruktion till att benämnas den första datorn är ENIAC (Electronic Numerical Integrator And Computer) som utvecklades parallellt med det arbete som von Neumann gjorde för att åstadkomma en logisk beskrivning av en datorarkitektur.



Figur 2.1 von Neumanns logiska uppbyggnad av datorn.

Även ENIAC utvecklades vid University of Pennsylvania i USA. Huvudmännen bakom ENIAC var Eckert och Mauchly. ENIAC var stor som ett småhus. Den bestod av mer än 100 000 elektriska komponenter varav 18 000 radiorör och flera kilometer av kabel. Beräkningskapaciteten hos ENIAC är idag möjlig att genomföra med ett microchip som ryms på en nagel. Det kan vara intressant att notera att när Eckert och Mauchly sökte pengar för att kommersialisera sin konstruktion sade en av personerna bakom Mark I att det inte fanns någon anledning att ge Eckert och Mauchly något stöd. Skälet som angavs var att det inte fanns något behov i USA för mer än fem eller sex maskiner av den typen. Idag vet vi hur fel det uttalandet var!

Eckert och Mauchly fortsatte att utveckla sin konstruktion och lanserade under 50-talet en dator under benämningen UNIVAC (Universal Automatic Computer). Dessa datorer var det som idag kallas stordatorer, det vill säga en stor central dator. Inledningsvis styrdes datorn av hålkort och senare kopplades terminaler till dem. Terminal innebär att det inte finns någon egentlig processorkapacitet i enheten utan behandlingen sker i den centrala datorn och informationen visas sedan på den terminal som skickade informationen till stordatorn.

Som en anekdot kan nämnas att så sent som 1979 genomfördes den första programmeringskursen vid Lunds tekniska högskola (LTH) utan direkt tillgång till en dator. Programmen som skrevs rättades av lärare med ett undantag. Det fanns en uppgift där studenten fick stansa hålkort och lämna in dem. Programmet kördes sedan under natten och följande dag var det möjligt att hämta resultatet. I bästa fall hade programmet fungerat, men i många fall hade kompilatorn upptäckt ett fel. Detta innebar att ett nytt kort fick stansas och ett nytt försök fick göras. Efter godkänd tentamen fick studenten genomföra en terminalkurs där programmeringen gjordes på en terminal som hade direkt kontakt med stordatorn. 1980 fick studenterna för första gången genomföra den första programmeringskursen utan hålkort. En anledning till ovanstående var att även om utvecklingen gått fort så var inte datorer något som alla hade, ty dels var inte persondatorn lanserad ännu och dels var kostnaden för en stordator mycket hög. Som exempel kan nämnas att en stordator lämplig för en institution vid LTH i mitten på 80-talet kostade runt en miljon kronor (i dåtidens penningvärde). Kapaciteten i den datorn är med dagens mått mycket ringa, det vill säga de bärbara datorer som vi har idag har större kapacitet än dåtidens dator och det till en bråkdel av priset.

Utvecklingen av datorerna fortsatte under 50-talet och 1958 byttes de stora radiorören ut mot transistorer. Denna utveckling av elektroniken ledde fram till en väsentlig minskning av storleken på datorerna. IBM hade initialt fokus på räknemaskiner men insåg snart hotet från datorer som UNIVAC. IBM:s första dator lanserades 1952. Utvecklingen under 1950-talet och 1960-talet karakteriseras av förbättring av många delar i datorn, till exempel övergången till användning av transistorer som nämnts ovan. Vidare utvecklades

minnena till datorerna och nya datorarkitekturer utvecklades. Fler företag växte upp inom området, även om marknaden dominerades fullständigt av IBM i början av 1960-talet.

I samband med utvecklingen av maskinerna insågs också vikten av att utveckla det sätt som datorerna styrdes. De första datorerna styrdes med hålkort som nämnts ovan. Detta betyder att inga program fanns i hårdvaran. En tanke som idag känns mycket främmande med det stora antalet program som finns på datorer idag. Ganska tidigt insågs att det inte var optimalt att tillföra logiken varje gång, vilket ledde till utvecklingen av kompilatorer och operativsystem. Kompilatorerna översatte instruktioner som gavs till datorn i en form som kunde tolkas av hårdvaran, det vill säga ettor och nollor. Operativsystemen utvecklades för att hantera datorns interna resurser som minne och olika typer av register. Utvecklingen av kompilatorer och operativsystem innebar också att utvecklingen av språk för att kommunicera med datorn tog fart.

2.2.2 Programvarans barndom

Utvecklingen av det som vi idag kallar programvara påbörjades i början av 1950-talet, även om arbetet med att skriva koder för att styra datorerna påbörjades redan på 1940-talet. Forskarna insåg att för att kunna generalisera användningen av datorer måste de styras av logik som var frikopplad från hårdvaran. Detta ledde till utvecklingen av de första kompilatorerna som först mest fungerade som väljare av rätt kort att använda. Kompilatorerna vidareutvecklades sedermera till att bli översättare mellan ett språk som programmeraren/operatören använde och den representation som maskinen hade. Den första kompilatorn vars funktion liknar dagens utvecklades 1954 av Lanning och Zierler vid Massachusetts Institute of Technology (MIT). Denna utveckling mötte motstånd, då kompilatorn innebar att exekveringen av programmen gick långsammare än om programmen skrevs direkt i maskinkod. Motståndet mot att skriva i mer abstrakta språk än maskinkod fortsatte långt in på 1980-talet. Det var inte förrän persondatorn slog igenom och processorkapaciteten blev sådan att skillnaden mellan att skriva direkt i maskinkod eller i ett språk på en högre nivå blev försumbar

(åtminstone för användaren). Det finns dock ännu idag vissa tillfällen då programmerare måste skriva mer maskinnära kod på grund av extrema tidskrav.

Utvecklingen av kompilatorn till en översättare gjorde att språkutvecklingen tog fart och begreppet programmeringsspråk introducerades. Anledningen till benämningen språk är att programmeringsspråken precis som det naturliga språket har en syntax. Det kan vara värt att notera att ett programmeringsspråk har en mycket begränsad vokabulär och utmaningen med programmering är många gånger att kunna lösa ett problem med denna begränsade vokabulär.

De första språken som utvecklades var assemblerspråk. De var med dagens mått mätt mycket maskinnära och det fanns ett språk för varje typ av dator, det vill säga det fanns assembler för UNIVAC och på motsvarande sätt för andra datormärken. Språken karakteriseras av att de inte skiljer på operativsystem och användarprogram som sker idag. Programmeraren arbetade med koder som talade om vad som skulle läggas i olika register på datorn och även vilka operationer som skulle genomföras på det som fanns i de olika registren.

Nästa steg i utvecklingen var att ta fram språk som inte direkt var ämnade för en specifik typ av dator. I slutet av 1950-talet kom första generationen av de första generella programmeringsspråken. De generella programmeringsspråken innebar också startpunkten för utveckling av de första operativsystemen, det vill säga programmeraren behövde inte bry sig om den interna strukturen på datorn i samma omfattning. Det första språket var FORTRAN (Formula Translation) som utvecklades av IBM och som lanserades 1957. Inledningsvis var inte FORTRAN ett riktigt generellt programmeringsspråk, utan programmeraren skulle fortfarande i viss utsträckning hantera datorns register. Detta ändrades dock efterhand som (andra) generella språk utvecklades. Som framgår av namnet var FORTRAN primärt tänkt för att hantera beräkningar och var därmed primärt ett språk för tekniska tillämpningar. För att kunna klara informationshantering bättre och för användning för mer kommersiella tillämpningar lanserades 1959 COBOL (Common Business Oriented Language). COBOL betraktas som det första generella pro-

grammeringsspråket då samma program (med mycket små modifikationer) kunde köras på flera olika datormärken.

Både FORTRAN och COBOL har överlevt länge och så sent som i slutet på 1970-talet var fortfarande FORTRAN det första programmeringsspråket som det undervisades i vid LTH. Språket hade dock utvecklats sedan lanseringen 1957. Även idag programmeras det i både FORTRAN och COBOL och inte minst finns det många system i dessa språk som fortfarande underhålls. Det märktes inte minst i samband med de uppdateringar som behövde göras för att kunna hantera datorsystemen i samband med 2000-problemet. Få hade tänkt sig att programvarusystemen skulle ha så lång livslängd och därmed representerades årtal i många system endast av två siffror, vilket innebar problem vid övergång från 1999 till 2000.

I början av 1960-talet lanserades ALGOL som det första generella programmeringsspråket för matematiska beräkningar. Förhoppningen var att det skulle konkurrera ut FORTRAN och därmed också bryta IBM:s dominans inom området. FORTRAN var utvecklat av IBM och för deras egna datorer. Så blev dock inte fallet. Istället vidareutvecklade IBM FORTRAN och gjorde det till ett generellt programmeringsspråk. Detta var en viktig faktor för att IBM skulle kunna behålla sin dominerande ställning genom hela 60-talet.

Under 1960-talet fortsatte utvecklingen av operativsystem och även andra nya programmeringsspråk utvecklades. De första erfarenheterna av storskalig programvaruutveckling kom fram. Fred Brooks har i sin klassiska bok ”The Mythical Man-Month” dokumenterat många av de problem som IBM fick under utvecklingen av operativsystemet OS/360. Brooks bok tar upp många problem och utmaningar som är lika aktuella idag som de var under 60-talet. Det kan synas konstigt att vi inte kommit längre, men det är värt att betänka att systemen idag är ännu mer komplexa och många fler människor utvecklar programvara idag än under 60-talet.

I slutet av 1960-talet lyftes vikten av strukturerad programmering fram. Vid denna tid användes ett språkkoncept som kallas ”go to” flitigt. Det innebar att det var möjligt att var som helst i ett program hoppa till ett annat namngivit ställe. Diskussionen som uppstod var huruvida det innebar att programmens kvalitet påverkades. Dijkstra som var den som initierade diskussionen ville får program-

mering att gå från att vara en konstart till att hanteras vetenskapligt.

Brooks bok och diskussionen kring ”go to” var två saker som bidrog till definitionen av ”software engineering”, det vill säga behovet av ingenjörsmässig utveckling av programvara. 1968 hölls en konferens med det vid denna tidpunkt provokativa namnet ”software engineering”. Konferensen hölls i Garmisch i Tyskland och får anses vara startpunkten för etableringen av storskalig programvaruutveckling som en vetenskaplig disciplin. Nästan 30 år senare vid en konferens i Berlin drogs slutsatsen att målsättningen som fanns vid konferensen 1968 inte hade uppnåtts. Programmering och programvaruutveckling är ännu inte betraktad och praktiseras inte som en vetenskaplig och ingenjörsmässig disciplin. Mycket kunskap finns idag och forskning bedrivs vid flertalet av världens universitet och på många stora företag inom området, men fortfarande drivs utvecklingen inte baserat på vetenskapliga resultat utan av starka kommersiella krafter.

2.2.3 Etableringsfasen

Under 1960-talet dominerade stordator, men under årtiondet utvecklades även det som kallas minidator. Utvecklingen inom elektroniken var en förutsättning för utvecklingen av minidatorn. Minidatorn byggde i stor utsträckning på stordatorn, men genom intåget av en mindre dator öppnades en rad nya användningsområden för datorer. Det gjorde också att fler datorer installerades och en marknad för tredjepartsprogramvara öppnades upp, det vill säga företag som inte sålde datorer började utveckla programvara.

IBM var dominerande under 60-talet med en marknadsandel på 70 %. Dock växte också nya företag upp och konkurrerade med IBM. Den kanske starkaste konkurrenten blev DEC (Digital Equipment Corporation) som utvecklade minidatorsystem som gick under namnet PDP (Programmed Data Processor). Deras PDP-8 lanserades 1965 och mer än 50 000 system installerades. DEC gav sig också på stordatormarknaden med sin VAX (Virtual Address eXtension) serie i slutet av 1970-talet. Namnet kommer ifrån att VAX var en vidareutveckling av PDP-11.

DEC:s framgång visade flera saker dels att det var möjligt att konkurrera med IBM och dels att datorn höll på att bli en produkt med en stor marknad. Det senare illustrerades med framgångarna med deras PDP serie som visade att världen behövde inte bara ett antal stordatorer utan att mindre datorer kunde komma till användning på ett stort antal områden. Det fanns dock fortfarande inga tankar på att datorn kunde bli en produkt som skulle finnas i hemmet och på varje arbetsbord i den industrialiserade delen av världen.

1960- och 1970-talet karakteriseras av stor- och minidatorer och utvecklingen av ett antal viktiga saker som möjliggjort intåget av persondatorn under 1980-talet och Internet under 1990-talet. Datorerna vidareutvecklades. Det möjliggjordes för flera användare att dela på en dator genom att varje användare fick tillgång till processorkapaciteten under en given tid för att sedan lämna över den till en annan användare. Tidigare hade i princip varje arbete alltid avslutats innan ett nytt arbete kunde starta.

Utöver utvecklingen av datorn skedde också ett antal andra viktiga landvinningar. En viktig sak skedde 1969 när fyra universitet sammankopplades i ett nätverk. Nätet kallades ARPANET (Advanced Research Projects Agency Network) och får ses som en föregångare till dagens Internet.

Även om den första integrerade kretsen hade konstruerats under slutet på 50-talet så skedde det en mycket snabb utveckling av kretstekniken under de följande 20 åren och utvecklingen har även fortsatt i en oerhörd fart efter det. De första 20 åren gjorde det dock möjligt att börja bygga persondatorer. Det stora genombrottet skedde 1974 när Intel byggde mikrokretsen 8080 som blev en viktig del i de första persondatorerna.

Med utvecklingen av datorerna insågs också att det fanns ett behov av att utbilda programmerare. Under 60-talet utvecklades språket BASIC (Beginner's All Purpose Symbolic Instruction Code) vars huvudsyfte var att fungera som ett enkelt programmeringsspråk för utbildningssyfte. Språket fick en stor spridning och många skrev sina första program i BASIC.

Under 1970-talet blev programmeringen mer allmän. Texas Instrument och HP (Hewlett-Packard) utvecklade programmerbara räkne-

dosor som fick en mycket stor spridning. Detta innebar att eleverna redan i skolan blev familjär med att kunna programmera, även om det var i begränsad omfattning. De första smådatorerna kom under andra halvan av 70-talet. Huvuddesigner för den första persondatorn var Edward Roberts. Datorn kallades Altair. Det fanns dock vissa problem med tanke på att all teknik, operativsystem och kompilatorer hade utvecklats för mycket större datorer. Två studenter vid Harvard bestämde sig för att skriva en BASIC kompilator för Altair. De båda var Bill Gates och Paul Allen. Deras kompilator användes i vidareutvecklingen av Altair, men de sålde aldrig rättigheterna till kompilatorn. Istället bildade de företaget Microsoft och rättigheterna till kompilatorn låg hos Microsoft. Samtidigt etablerade sig ett annat välkänt företag nämligen Apple. Steve Jobs och Steve Wozniak lanserade 1978 Appledatorer i liten skala.

Microsofts stora lyft var när de kom i kontakt med IBM och fick ansvar för framtagningen av ett operativsystem för IBM. Resultatet blev MS-DOS, det vill säga Microsoft Disk Operating System eller PC-DOS som det hette hos IBM. IBM erhöll aldrig rättigheterna till operativsystem utan rättigheterna var hos Microsoft. Det betydde att Microsoft hade möjlighet att låta även andra datortillverkare använda samma operativsystem som IBM hade. Detta smarta drag lade basen för Microsofts framgång och med stor säkerhet lade basen för varför PC:n idag är betydligt större än Macintosh. Det senare är den äpplesort som Apple valde att namnge sina datorer efter. Många jämför Microsoft med Apple, men det finns stora skillnader. Microsoft är ett programvaruföretag medan Apple utvecklar datorer och tillhörande systemprogramvara (i första hand operativsystem).

Den första persondatorn från IBM lanserades 1981 och tack vare sin starka position blev de snabbt ledande. Andra datortillverkare arbetade med att vara IBM-kompatibla och därmed blev IBM:s persondatorer standard på området. Idag är inte IBM lika dominerade, men deras ställning i början på 80-talet ligger till grund för hur persondatorn ser ut idag. Kopplingen mellan IBM och Microsoft har gjort att vi i dag i flertalet datorer som tillverkas har operativsystem utvecklade av Microsoft, till exempel Windows98, Windows NT och Windows XP där vi än idag kan hitta MS-DOS i botten.

Ett annat företag som påverkat utvecklingen i hög grad är Xerox som vi idag i första hand associerar med kopiatorer. Xerox hade dock ett forskningsbolag Xerox-PARC som låg i Palo Alto i det som skulle växa till Silicon Valley i Kalifornien. Vid detta bolag i Palo Alto utvecklades datormusen och även det grafiska användargränssnittet samt Ethernet och laserskrivaren. Xerox hade dock inte förmågan att kommersialisera uppfinningarna. Det grafiska gränssnittet kommersialiserades först av Apple som släppte det i sin Macintoshversion 1984. Ethernet är idag mer eller mindre standard för lokala nätverk. Kommersialiseringen sköttes i detta fall av företag som 3-Com, medan HP var de som drog nytta av laserskrivaren. Xerox bidrag till datorutvecklingen har varit mycket stort även om de inte till fullo lyckades dra nytta av sina innovationer.

Det grafiska gränssnittet som Apple hade på sin Macintosh var i mitten på 80-talet marknadsledande. Detta försprång behöll Apple fram till det att Microsoft introducerade Windows95 och många skulle än idag hävda att Apples användargränssnitt är bättre, medan andra tycker om Microsofts lösning.

Under 80-talet skedde också en utveckling av det som kallas arbetsstationer. Detta är en typ av dator som riktar sig till arbetsplatser ofta med ganska tunga beräkningar. I princip var det en utveckling som satte stordatorn på alla skrivbord istället för att användarna skulle dela på en dator. Den första arbetsstationen kom 1981 och levererades av Apollo. En av de första konkurrenterna till Apollo blev SUN Microsystems som bildades 1982. Arbetsstationerna hade inledningsvis sina egna operativsystem, men har efterhand gått över till UNIX. Det senare är ett operativsystem som började utvecklas på AT&T i USA och som sedan skrevs om i stor utsträckning vid Berkeley University av Bill Joy. UNIX hade en stor spridning och har fortfarande i viss utsträckning inom universitetsvärlden då det i det närmaste var gratis. SUN rekryterade Joy, vilket i stor utsträckning bidrog till företagets tidiga framgångar. Ett UNIX-inspirerat operativsystem för PC utvecklades under 90-talet av Linus Torvalds. Operativsystemet fick namnet Linux.

Utvecklingen av programvaruområdet har också fortsatt. Redan på 1960-talet introducerades objektorientering via Simula. Ett programmeringsspråk som utvecklades i Norge. Språket fick ingen

större genomslagskraft, men objektorienteringen har i många stycken blivit dominerande sedan introduktionen av C++ och Java. Vidare har programvaruutvecklingen utvecklats genom introduktion av metoder och verktyg för design, kravhantering, testning och ledning av programvaruprojekt. Utvecklingen och nuvarande status på programvaruutvecklingsområdet presenteras i respektive kapitel i fortsättningen av boken.

2.2.4 IT-samhället

Utvecklingen av Internet och World Wide Web under 1990-talet har förändrat vår syn på samhället vi lever i. Idag förutsätter vi (nästan) tillgång till Internet. Det är dock värt att betona att World Wide Web introducerades officiellt 1994. 90-talet innebar att mobiltelefonen och datorn blev allmänt tillgängliga. Väldigt många personer och familjer har idag mobiltelefoner och datorer i hemmet. Det har också blivit vanligt med lokala nätverk i hemmet, antalet personer med någon form av bredbandsanslutning växer snabbt och utvecklingen bara fortsätter.

Samhället har idag blivit starkt beroende av datorer och de funktioner som de tillhandahåller. Det är idag inte möjligt att gå till banken eller göra några inköp utan att en dator finns med i bilden. Denna utveckling till ett informationssamhälle har skett på cirka 50 år (mitten på 40-talet till mitten på 90-talet).

Merparten av historiken har varit inriktad på datorn. Programvaran har inte utvecklats på samma sätt, vilket inte heller kan förväntas. Många saker har förbättrats, men det är fortfarande människor som måste klara av att utveckla programvaran och utnyttja de möjligheter som datorutvecklingen ger oss.

Del 2: Byggstenar

Flertalet organisationer idag har någon form av matrisorganisation. Detta innebär att det finns en linjeorganisation, det vill säga en organisation som har hand om resurserna. Dessa resurser kan vara både människor och utrustning. Linjeorganisationer är typiskt uppbyggd kring olika nivåer av enheter och tillhörande chefer. Det kan röra sig om grupper som ingår i en avdelning som tillhör en sektion som slutligen ingår i en organisation. Linjeorganisationen utgör den ena dimensionen i en matris. Den andra dimensionen är ofta projektorganisationen. Den senare karakteriseras av att den byggs upp för en begränsad tid och med ett givit mål. Ett projekt bemannas av personer från linjeorganisationen. Denna del introducerar dessa båda dimensioner, då de ofta är viktiga i organisationer som utvecklar programvarusystem.

3 Projekt

3.1 Inledning

Programvaruutveckling sker normalt i projektform. Projektet är utgångspunkten för uppbyggnaden av denna bok. Målsättningen är att klargöra projektets roll samt hur andra delar i en organisation och olika aktiviteter stödjer genomförandet av ett programvaruprojekt. Först är det dock viktigt att definiera vad ett projekt är. Projekt kan definieras som följer:

Ett projekt är målorienterat, det vill säga har ett specifikt slutresultat, och består av ett antal aktiviteter som skall genomföras under en avgränsad tid med begränsade resurser och budget.

Denna något korta definition kan uttolkas i följande punkter när nyckelorden i definitionen analyseras närmre:

Målorienterat
Ett projekt formuleras normalt kring ett tydligt mål. Tydligt mål innebär också att målet bör vara unikt för varje projekt och därmed blir varje projekt unikt. Om det inte går att formulera ett tydligt och unikt mål är det nog lämpligt att överväga om det verkligen är ett projekt som håller på att formuleras. Det är kanske egentligen flera projekt? Alternativt har målet formulerats alldeles för generiskt, det vill säga ett mål som passar ett stort antal projekt. Ett exempel på ett dåligt mål: Projektet skall leverera produkten till kunden enligt ställda krav. Detta mål säger inget om vad projektet egentligen går ut på utan det kan i princip passa in på alla projekt. Mål av denna typ måste formuleras om.

Slutresultat
Resultatet är nära relaterat till målet. Ett projekt bör resultera i något konkret. Det senare kan vara en produkt, en tjänst eller någon form av dokumentation, till exempel en utredning.

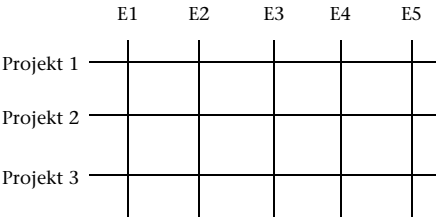
Aktiviteter
Ett projekt byggs normalt upp av ett antal aktiviteter som tillsammans leder projektet fram till målet. Dessa måste normalt genomföras i en viss ordning. Det finns med andra ord beroenden mellan olika aktiviteter. Det betyder inte att det finns beroende mellan alla aktiviteterna. En viktig uppgift för projektledningen är att planera aktiviteterna så att de kan genomföras på bästa möjliga sätt. Aktiviteterna måste anpassas i förhållande till typen av projekt, men består normalt av aktiviteterna både relaterat till projektledning och genomförandet av projektet. Aktiviteterna för genomförandet av ett programvaruprojekt behandlas ingående i Del 3.

Avgränsad tid
Ett projekt karakteriseras av att det är avgränsat i tiden. Det har normalt en startpunkt och en slutpunkt. Ofta vid produktutveckling kan ett projekt följas av ett nytt projekt, men då med ett nytt mål. Att det är avgränsat i tiden betyder också att fast anställd personal är normalt inte anställda i projekt, utan de är anställda i en organisation. Den enda personal som normalt är anställda i projekt är korttidsanställningar i form av projektanställningar. I dessa fall har en eller flera personer anställts för att utföra vissa uppgifter i projektet.

Begränsade resurser och budget
Begränsningen i resurser innebär att projektet förmodligen har tillgång till ett visst antal personer, viss utrustning och vissa lokaler och så vidare. Vad avser personalen har projektledare ibland möjlighet att välja personal, men i andra fall tilldelas ett projekt viss personal. Den exakta fördelningen av personal på projekt bestäms ofta av svåra avgöranden och prioriteringar mellan olika projekt i en organisation. Resursbegränsningen är nära kopplad till en begränsning även i budget, det vill säga projektet förväntas genomföras inom vissa givna kostnader.

Innan vi gör en mer detaljerad genomgång av vad projektarbete innebär och vilka olika faser som ett projekt genomgår är det viktigt att kortfattat sätta projektet i ett sammanhang. Avsikten i detta kapitel är att skapa en förståelse för sammanhanget även om detaljerna lämnas till senare kapitel.

Flertalet företag är organiserade i någon form av matris, se Figur 3.1. I figuren visas fem olika enheter i en organisation (E1–E5) och tre projekt. Den exakta definitionen av en enhet beror på respektive företags interna organisation. Det kan vara grupper, sektioner, avdelningar eller någon annan beteckning som företaget valt. Enheterna innehåller ofta personer med olika kompetens. En organisation kan till exempel ha en designenhet, en testenhet och en enhet för projektledare. Oavsett vilken personal som finns vid respektive enhet är det mycket vanligt med denna typ av organisation. Vi återkommer till organisationen i Kapitel 4.



Figur 3.1 Organisation i matrisform.

Ett projekt producerar dokumentation som bidrar till att dokumentera progressen i ett projekt. Den dokumentation som sker kan delas in i tre delar:

Produktdokumentation
Denna dokumentation relaterar till målet med projektet och dess resultat till exempel i form av en programvara eller en tjänst. Dokumentationen hanteras i anslutning till produkten och efter avslutat projekt överförs ofta ansvaret för dokumentationen till någon som är produktansvarig. Det betyder också att dokumentationen med stor säkerhet kommer att uppdateras av andra efter avslutat projekt, antingen av produktansvariga eller av något annat projekt som syftar till att vidareutveckla produkten. Exempel på dokumentation kan vara kravspecifikationer, design och testplaner, men även protokoll från granskningar och test. Detta betyder att viss produktdokumentation är av karaktären att den kan uppdateras framöver när produkten ändras, medan annan dokumentation beskriver resultat vid en specifik tidpunkt, till exempel ett testprotokoll. Produktdokumentation kommer att beröras i anslutning till de respektive kapitel som beskriver de olika aktiviteterna i samband med programvaruutveckling.

Projektdokumentation
Projektdokumentation karakteriseras av att dess huvudsyfte är att dokumentera och stödja det pågående projektet. Denna dokumentation sparas efter ett avslutat projekt, men uppdateras normalt inte. Typiska exempel på dokument är projektplaner och mötesprotokoll. Projektdokumentationen diskuteras mer ingående senare i detta kapitel.

Organisationsdokumentation
Denna typ av dokumentation är inte alltid aktuell i ett projekt, men den kan förekomma. Med organisationsdokumentation avses beskrivningar av arbetsmetodik, användarhandledningar till verktyg och hjälpmedel och annan dokumentation som berör organisationen som helhet. Normalt arbetar inte utvecklingsprojekt med denna typ av dokumentation, men det kan hända att det under ett projekt görs sådana erfarenheter att organisationsdokumentationen uppdateras. Denna typ av dokumentation diskuteras i mer detalj i Kapitel 4.

Utgående från ovanstående skall vi nedan gå igenom projektarbete i mer detalj. Det är dock viktigt att ha ovanstående i åtanke nedan.

3.2 Projektarbete

Ett projekt bryts normalt ned dels i ett antal aktiviteter som nämnts ovan och dels i ett antal arbetsuppgifter inom varje aktivitet. Denna nedbrytning görs för att arbete skall kunna fördelas mellan personer som arbetar i projektet. Under projektets gång skall sedan resultatet kunna sättas samman för att i slutet av projektet utgöra det resultat som levereras. För att nedbrytning, arbete och sammansättning skall fungera krävs ett antal saker. Det krävs:

- Projektspecifika aktiviteter, till exempel projektledning och definition, planering, organisation, genomförande och avslut av ett projekt. Dessa projektaktiviteter beskrivs nedan i detta kapitel. Innan dess är det dock viktigt att illustrera övriga viktiga aspekter som vi återkommer till senare. Målsättningen här är att ge en viss insikt i dem så att de bidrar till att skapa en helhetsbild över projektarbetet.

- Utvecklingsprocess, vilken rekommenderar i vilken ordning arbetet skall ske. Den övergripande utvecklingsprocessen beskrivs i Kapitel 4. Organisationens utvecklingsprocess instansieras i projektet, vilket betyder att processen ligger till grund för utvecklingen men avvikelser kan ske. De senare skall dock dokumenteras i projektet. De olika aktiviterna i processen beskrivs i Del 3. Aktiviteterna omfattar från det att kravbilden klargörs tills det att systemtesten för programvaran godkänts. I de fall det finns en specifik kund kan även ett formellt acceptanstest ingå. En kravbild kan komma från flera olika källor:
  – krav från en specifik kund,
  – från en förändringsanalys,
  – krav från en marknad,
  – krav baserat på teknisk standard eller lagstiftning, eller
  – tas fram inom företaget.
  Dessa olika sätt att skapa en kravbild diskuteras i Kapitel 5. Från kraven sker en design (Kapitel 6), vilken följs av en implementation i form av kod (Kapitel 7). Det arbete som göra för att utveckla produkten behöver kvalitetssäkras och för att göra detta används inom projektet granskning (Kapitel 8) och test (Kapitel 9).
- Produktstruktur, det vill säga en struktur på det system som skall utvecklas inom ramen för projektet. Ett projekt kan antingen utveckla ett helt nytt programvarusystem eller vidareutveckla ett existerande system. Det senare är det vanligaste fallet. Det betyder att produktstrukturen till en stor utsträckning är given. Inom ramen för ett projekt kan det finnas behov att ändra strukturen på grund av det nya som skall implementeras. Det är dock relativt sällsynt att större förändringar görs inom ramen för ett projekt såvida inte målsättningen med projektet är att omstrukturera ett existerande system. Produktstrukturer presenteras i Kapitel 6 som en del i designarbetet, där strukturerna ses som en design på en hög nivå.

Ovanstående tre delar kopplas i sin tur till de tre olika typerna av dokumentation som finns enligt ovan. Tilläggas skall att även andra aspekter är viktiga som till exempel kvalitetsuppföljning och de enskilda personerna som deltager i projektet. Även dessa andra aspekter återkommer vi till.

Inom ramen för ett projekt finns en rad roller. En roll innebär ett visst ansvar. En person kan inneha flera olika roller. Inom olika företag och organisationer definieras roller olika. Rollerna nedan får ses som naturliga exempel på roller som ofta återfinns inom ramen för ett projekt eller i anslutning till projektet. Exempel på vanliga roller och deras betydelse:

Projektledare
Denna person har det övergripande ansvaret för projektet. I ansvaret ingår att planera, följa upp och rapportera progressen i projektet. Projektledarens arbete beskrivs i Avsnitt 3.3.

Linjechef
Linjechefen eller -cheferna ses normalt inte som en del av projektet. De bidrar till projektet genom att tillhandahålla resurser i form av personal till projektet, vilket då illustreras av matrisorganisationen i Figur 3.1.

Systemarkitekt
För att hålla samman systemet och se till att det får en bra övergripande struktur har många projekt en systemarkitekt eller systemansvarig. Arkitekten är den person som har det övergripande tekniska ansvaret för systemstrukturen. Denna roll är nära förknippad med produktstrukturen som beskrivs i Kapitel 6.

Utvecklare
I denna roll innefattas de personer som bidrar till implementationen av systemet. I större projekt kan denna roll vara nedbruten i flera, till exempel systemdesigner, designer och programmerare. I många fall utför även utvecklare test av enheter medan test på högre nivåer, till exempel funktionstest och systemtest utförs av en separat testenhet. Utvecklingen beskrivs närmre i Kapitel 5–7.

Testare
Rollen syftar på dem som genomför test av till exempel funktioner och systemet. I mindre projekt kan det vara utvecklarna, medan det i större projekt ofta finns speciella testare. I fallet med speciella testare arbetar de ofta med att ta fram testspecifikationer och -instruktioner medan utvecklarna implementerar systemet. Test presenteras i Kapitel 9.

I större projekt kan även andra roller vara aktuella som kvalitetsansvarig och det är också möjligt att det är olika personer som arbetar med kravhantering, design respektive implementation.

Ovanstående är tänkt att ge en kort introduktion till projekt i allmänhet innan vi nu tar upp de olika specifika projektaktiviteterna, det vill säga de aktiviteter som är en följd av att arbetet utförs i form av ett projekt.

3.3 Projektledning

Projektledaren har det övergripande ansvaret för projektet. Denna person skall planera, följa upp och rapportera projektet. I detta ingår ett antal saker som att skriva en projektplan, bemanna projektet, skatta tidsåtgång (både i arbetstimmar och kalendertid), hantera risker, följa upp projektet i förhållande till plan och rapportera projektstatus till berörda personer. Det senare kan vara inom den egna organisationen eller till någon extern kund. Projektledarens arbete beskrivs bäst med hjälp av en genomgång av de olika projektaktiviteterna. Innan genomgången av de olika aktiviteterna i ett projekt är det viktigt att veta vilka som är de största riskerna i ett projekt. Nedan följer en lista på generella projektrisker. I ett specifikt projekt kan det naturligtvis finnas andra risker, till exempel specifika tekniska risker.

Tio vanliga risker i ett projekt är (utan inbördes prioritetsordning):

1. Projektet har ett orealistiskt mål, flera olika mål, saknar mål eller projektet driver ifrån sitt mål under arbetets gång.
2. Endast projektmedlemmarna är intresserade av resultatet.
3. Dålig eller ingen projektledning.
4. Dålig projektplan, till exempel saknar struktur och detaljer.
5. Projektet har fel bemanning, till exempel kan viss kompetens saknas.
6. Orealistiska tidplaner avseende både kalendertid och arbetstimmar.
7. Projektuppföljning genomförs inte, vilket är nära relaterat till dålig intern kommunikation inom projektet.

8 Kravinstabilitet, det vill säga kraven ändras under projektets gång och det finns inte rutiner på plats för att kunna hantera detta.
9 Utveckling av fel funktioner eller användargränssnitt, det vill säga det saknas en förståelse för kravens betydelse.
10 Bristande förståelse för de icke-funktionella kraven, vilket kan inkludera både den miljö som systemet skall verka i och produktattribut som prestanda och tillgänglighet.

Det är också viktigt att betona i samband med projektledarens arbete att en av de viktigare sakerna i arbetet är beslutsfattande. Många gånger är det bättre att ta ett beslut nu istället för att vänta och hoppas på att mer information skall finnas tillgänglig. Det är projektledarens ansvar att ta de beslut som behövs för att säkerställa att chanserna till framgång för projektet maximeras. Målsättningen för projektledaren måste vara att ta beslut baserat på fakta. Det innebär att det är viktigt i projektarbetet att underlag tas fram så att rätt beslut kan fattas. Det senare gäller såväl projektaktiviteter som de aktiviteter som är en del av arbetsmetodiken vid utveckling av systemet.

Som stöd för projektarbetet finns ofta en projektmodell. Denna modell skall inte blandas samman med utvecklingsprocessen. Den senare är fokuserad på det tekniska arbetet medan projektmodellen fungerar som stöd åt projektledaren. Nedan beskrivs denna modell i fem aktiviteter: definiera, planera, organisera, genomföra och avsluta. Varje aktivitet innehåller i sin tur delaktiviteter, vilka beskrivs under respektive rubrik nedan.

Utöver aktiviterna har många projektmodeller milstolpar och kontrollpunkter. En milstolpe är en projektintern punkt som används för avstämning. Detta är viktiga punkter i projektet där projektledaren och -deltagarna kan avgöra att ett delmål med projektet har uppnåtts. Delmålet kan vara att vissa dokument eller delar av systemet skall vara klart. Denna typ av delmål gör att det finns naturliga kopplingar till utvecklingsprocessen (ibland kallad utvecklingsmodell). Anledningen till detta är att det är utvecklingsprocessen som beskriver vilka produktdokument som skall tas fram.

I många organisationer finns även externa kontrollpunkter (på engelska oftast benämnda ”toll gates”). Dessa är beslutspunkter för

företagets ledning, det vill säga punkter där projektet skall rapportera sin progress och ledningen beslutar om huruvida projektet skall fortsätta eller det skall läggas ned. Det är många gånger svårt att lägga ned projekt, samtidigt som det är viktigt att använda företagets tillgängliga resurser på bästa möjliga sätt. Nedläggning av projekt blir naturligtvis svårare och svårare desto längre ett projekt har kommit. Den exakta placeringen av både milstolpar och kontrollpunkter bestäms av respektive projektmodell och dess koppling till organisationens utvecklingsprocess.

3.4 Definiera

Ett projekt måste definieras först. Detta innebär att beslutsunderlag tas fram för att avgöra om projektet överhuvudtaget skall startas eller ej. Den första aktiviteten sammanfattas i ett formellt projektförslag eller projektöversikt som fungerar som beslutsunderlag. I många fall initieras skrivandet av ett projektförslag från ledningen. Det innebär också att många saker är givna från början, till exempel målsättning eller tidpunkt för det att projektet skall vara klart. Detta är ofta en följd av marknadstryck eller andra yttre omständigheter. Svårigheten är att projektet kan hamna i en situation där det definieras och beskrivs från krav och förutsättningar som inte är i harmoni. Det kan innebära att tänkta resultatet från projektet inte är möjliga att ta fram inom den givna tidsramen eller att kvaliteten inte kommer att nå den nivå som önskas.

Varje projekt har fyra huvudaspekter att hantera:

- Funktionalitet, det vill säga vad är det som skall levereras från projektet i form av funktioner?
- Kostnad, vilket relaterar till hur mycket resurser som kan användas under projektets gång. Merparten av kostnaden för programvaruprojekt är personal, även om kostnader för utrustning med mera också måste tas i beaktande.
- Ledtid, det vill säga när skall projektet vara klart och leverera det tänkta resultatet?

- Kvalitet, vilket hänför till hur de icke-funktionella egenskaperna hos ett programvarusystem uppfylls. Kvalitet är ett vitt begrepp och innefattar både externa aspekter som prestanda och tillförlitlighet och interna aspekter som att programvaran skall kunna underhållas och eventuellt också vidareutvecklas.

Det stora problemet i projekt tillstöter när det finns förväntningar på alla dessa fyra utan att en noggrann analys har gjorts av möjligheten att uppnå samtliga fyra på en och samma gång. Det är i dessa fall väsentligt att projektförslaget innehåller en möjlighetsanalys, det vill säga försöker analysera möjligheten att uppnå förväntningarna.

Ovanstående betyder att det är ganska sällsynt att projektledaren får möjlighet att skriva ett projektförslag helt utan vissa givna förutsättningar. Om det finns givna förutsättningar är det av yttersta vikt att projektledaren inkluderar en analys av de givna förutsättningarna och diskuterar eventuella konsekvenser av förutsättningarna.

Ett projektförslag bör åtminstone besvara följande frågor:
- Vilket problem skall projektet lösa?
- Vilka mål har projektet?
- Vilka saker skall uppnås under projektarbetet? Denna fråga kopplas med fördel till milstolparna i projektet.
- Vilka resurser behövs för genomförandet?
- Vilka risker finns med projektet?
- Vilka möjligheter finns att klara projektet under de givna förutsättningarna?

Dessa frågor diskuteras närmre i Avsnitt 3.4.1–3.4.6.

3.4.1 Problembeskrivning

Det första som skall göras är att definiera det problem projektet skall lösa. Alternativt den möjlighet som projektet skulle leda till. Det senare kan vara att företaget kommer ut med en viss produkt först på marknaden. Denna beskrivning måste både inkludera vilket behov projektet avser att möta och hur den egna organisationen kan tjäna på det.

Ovanstående syftar i första hand till att svara på frågan: ”Vad projektet skall åstadkomma?”. Nästa steg är att kortfattat beskriva hur detta skall åstadkommas. Det är viktigt att försöka beskriva vad som behöver göras i projektet och hur framgången i projektet kan mätas. Problembeskrivningen bryts sedan ner i två delar: projektmål (Avsnitt 3.4.2) och milstolpar (Avsnitt 3.4.3).

Det är också viktigt att bestämma vem som är ansvarig för projektet. Detta inkluderar både projektledare och ansvarig linjeorganisation, det vill säga någon måste äga projektet. Projektägaren och eventuellt (andra) personer från ledningen deltar oftast i de kontrollpunkter som omnämndes ovan. Det kan vara projektledaren som skriver denna första projektdefinition, men det behöver inte vara det.

Det är också viktigt att det anges tydligt när projektet skall vara klart. Denna tidpunkt kan antingen planeras eller krävas som diskuterats ovan, det vill säga det kan finnas externa krav som gör att projektet inte kan planeras utan att vissa förutsättningar är givna. Planerad eller krävd färdigtidpunkt är viktig information då en organisation ofta har projekt som konkurrerar om resurserna. Det innebär också att det kan behöva prioriteras mellan olika projekt. Projektdefinitionen skall bland annat vara ett beslutsunderlag för att kunna göra denna prioritering.

3.4.2 Projektmål

Problembeskrivningen behöver brytas ned i mer konkreta projektmål. Varje projektmål bör endast ha ett huvudmål. Detta mål kan sedan brytas ned i delmål, men det är viktigt att det finns ett huvudmål som alla inblandade parter kan relatera. Delmålen bör relatera till milstolparna som diskuteras i nästa avsnitt.

Projektmålet skall definiera det slutliga resultatet från projektet i tydliga termer. Målet skall skrivas kortfattat och vara lättförståeligt för alla inblandade parter. Detta innebär att målet skall kunna fungera som en gemensam referenspunkt som det går att återvända till vid eventuella konflikter. Slutligen är det viktigt att målet är mätbart så att det går att avgöra när målet är uppnått.

3.4.3 Milstolpar

Milstolparna eller delmålen skall beskriva vägen till slutmålet i form av tydliga steg. Tydlighet innebär att delmålen skall vara smarta:

- Specifika, det vill säga ha ett tydligt syfte i linje med projektets mål.
- Mätbara, det vill säga det skall vara enkelt att avgöra om delmålet är uppnått eller ej.
- Ansvarsrelaterade, det vill säga det skall vara möjligt att utpeka en person som ansvarig för ett delmål.
- Realistiskt, det vill säga det skall vara ett rimligt mål inom de givna tidsramarna och den givna kostnaden.
- Tidsrelaterat, det vill säga det skall vara möjligt att säga när delmålet kan vara uppnått.

Första bokstaven i de fem huvudorden bildar ordet SMART, vilket gör det enklare att komma ihåg det som krävs av ett bra delmål eller milstolpe i ett projekt.

Exempel: Ett delmål inom en utbildning kan vara att klara en specifik kurs. Om vi antar att kurs A skall klaras av. Initialt sätter vi kanske upp följande delmål: *Klara Kurs A*. Detta uppfyller dock inte alla orden ovan. Det kan vara lämpligt att ange vid vilket universitet eller högskola som kursen skall klaras av. Vidare kan det vara bra att definiera vad som avses med att klara. På många kurser finns det olika betyg och delmålet bör då tydliggöra om betyget godkänd eller tre är tillräckligt för att delmålet skall vara uppnått. Slutligen behöver även anges när kursen skall klaras av. En betydligt bättre formulering vore: *Klara Kurs A vid Blekinge tekniska högskola med lägst betyget tre senast innevarande år.* Detta delmål uppfyller kraven på ett bra (smart) delmål.

I samband med delmålen skall även anges hur det sista delmålet som sammanfaller med huvudmålet skall utvärderas. Det kan göras i termer av antal sålda licenser under till exempel de första sex månaderna efter projektavslut eller att produkten rankas bland de fem bästa i en specifik utvärdering eller något annat som anses lämpligt.

3.4.4 Resurser

Det finns två principiellt olika sätt som resurser (i form av arbetstimmar) brukar bestämmas. Det första är att det är givet och att projektet därmed planeras utgående från det. I det andra fallet uppskattas antalet arbetstimmar som behövs för att genomföra projektet. Det är i detta sammanhang också viktigt att inte bara tänka i antal timmar utan också ta ställning till vilken kompetens som behövs i projektet. Rätt sammansättning av personal i ett projekt kan vara skillnaden mellan framgång och ett misslyckat projekt. Olika kompetens kan vara allt från att se till att det finns tekniska experter på vissa områden till att projektet har en bra sammansättning vad avser till exempel kompetens inom design och testning.

Det kan också vara viktigt att tänka över kombinationen av personer, då vissa personer arbetar bra tillsammans medan andra inte gör det lika bra. Projektarbete kräver bra samarbete och kommunikation inom projektet. Därmed är det viktigt att få in rätt personal med hänsyn till personliga egenskaper och inte bara med avseende på kompetens.

Om det är aktuellt att försöka skatta antalet arbetstimmar i ett projekt finns det ett antal metoder att använda. Det finns fem principiellt olika metoder:

Algoritmiska modeller
Denna typ av modell syftar på användningen av en ekvation mellan till exempel produktstorlek (oftast i skattat antal rader kod) och antal arbetstimmar. Det skall noteras att det ofta är svårt att skatta storleken i kod tidigt och det finns modeller som försöker ta in mått från kraven för att göra skattning av antalet arbetstimmar. I modellen finns sedan även ett antal konstanta faktorer som bör bestämmas från tidigare genomförda projekt. Den mest välkända modellen av denna typ är COCOMO (Contructive Cost Model). Modellerna är oftast på följande form:

$$Arbetstimmar = a \times Storlek^{b} \times M$$

I formeln är a och b konstanter som bestäms av tidigare projekt och M bestäms av olika produkt- och processattribut. COCOMO finns i olika former och det finns förslag på värden på konstan-

terna beroende på olika situationer. Det är dock vanligt att *b* ligger mellan 1.0 och 1.5 åtminstone för större projekt.

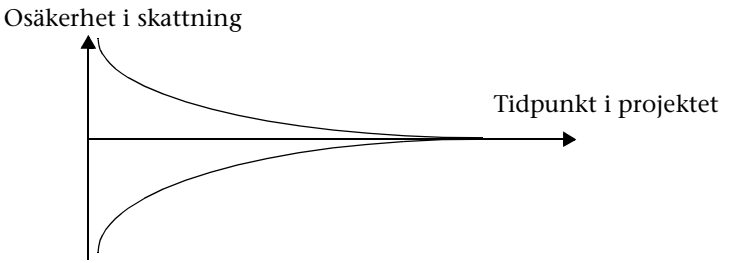
Expertbedömning
Denna metod bygger på att ett antal experter med kunskap om liknande produkter och om utvecklingsprocessen gör skattningar. Det allra bästa är om experterna inte bara gör en punktskattning, det vill säga en siffra för projektet, utan att de gör intervallskattningar. Normalt gör experterna individuella skattningar som sedan diskuteras vid ett möte för att komma fram till en gemensam skattning. Det bästa är om skattningen slutar med att det finns en mest trolig skattning och ett intervall runt det värdet med max- och minvärde. Detta är ett sätt att också erkänna den osäkerhet som finns i samband med skattningen.

Analogi
Vid analogiskattning är ambitionen att hitta liknande situationer eller projekt. Utgående från tidigare erfarenhet identifieras hela projekt eller delar av projekt som anses liknande. Dessa används sedan som utgångsvärden för att skatta för det kommande projektet. Tanken är inte att siffror från tidigare projekt skall tas rakt upp och ned, utan att de skall användas som startpunkt för att göra skattningen.

Parkinsons lag
Denna lag är inte en skattning i riktig bemärkelse. Lagen säger att om det finns en viss tillgänglig kalendertid, till exempel fem månader och sex personer är avsatta för att arbeta i projektet kommer arbetstiden att bli 6 * 5 = 30 arbetsmånader. Denna lag bygger på att alltför ofta bestäms leveransdatum och vilka personer som skall arbeta i projektet utan att arbetstimmarna har skattats. I och med att detta är gjort ges arbetstimmarna av de andra besluten. Det finns flera problem med detta. Det första är att det kan resultera i för få tillgängliga arbetstimmar för det planerade innehållet i produkten. Det andra problemet är att det kan vara för många arbetstimmar, men eftersom det är planerat fylls tiden till leverans. I det senare fallet skulle personal kunnat användas till andra arbetsuppgifter istället för att finputsa på något som kanske redan är tillräckligt bra.

Rätt pris
Om resultatet från projektet skall levereras till en specifik kund är det inte alltid en fråga om skattning av arbetstimmar, utan det är en fråga om vilket pris som kan offereras för utvecklingen. Det senare betyder att priset som satts för kunden skall inkludera utvecklingskostnad och viss vinst. Detta gör att kostnaden i princip sätts via priset som offererats och inte från en skattning av arbetstimmar.

Resursskattningen måste successivt förfinas. Det betyder att i projektplanen som skrivs senare skall en mer aktuell skattning göras. Uppföljning och omplanering är viktiga aktiviteter som diskuteras ytterligare nedan, se Avsnitt 3.5. En viktig anledning till omplanering och förnyade skattningar är att osäkerheten minskar efterhand som tiden går, se exempel i Figur 3.2.



Figur 3.2 Osäkerheten i skattningar.

En viktig aspekt i samband med skattning är att tänka efter om projektet faktiskt har förändrats. I många fall ändras kraven på produkten under utvecklingens gång. Vidare är det inte ovanligt att viss funktionalitet till exempel tas bort för att produkten skall hinna komma ut på marknaden i tid. Dessa saker gör att den skattning som gjorts inledningsvis är kanske inte riktigt aktuell beroende på ändrade förutsättningar. Det är då viktigt att göra en ny skattning.

3.4.5 Riskanalys

Ovan angavs vissa allmänna vanliga typer av risker i projekt. Det är dock viktigt att en seriös riskanalys görs för det aktuella projektet. Vid riskanalys är det bra att dela ned analysen i olika delar för att

därigenom få en bra genomgång. Följande är en möjlig nedbrytning av analysen:

Organisationsrisker
Dessa risker härrör till organisatoriska aspekter som till exempel en omorganisation eller att projektet bedrivs på flera olika orter.

Personalrisker
Dessa risker kan bero på att rätt kompetens inte är tillgänglig eller att projektet är stark beroende av någon nyckelperson.

Kravrisker
Detta är oftast risker baserat på förändringar av krav. Denna typ av risker är oftast störst när kunden inte själv har djup teknisk kunskap inom området.

Tekniska risker
De tekniska riskerna kan beröra många olika aspekter till exempel problem med inköpta programvarukomponenter eller missförstånd av gränssnitten mellan något egenutvecklat och en inköpt standardprodukt.

Verktygsrisker
I många projekt används utvecklingsverktyg och projektet kan då hamna i en beroendeställning till utvecklarna av verktyget. Exempel på risker av denna typ kan vara att verktyget inte vidareutvecklas på förväntat sätt eller att verktygsutvecklaren köps upp och därmed ändras kanske tidigare affärsplaner.

Skattningsrisker
Skattning diskuterades ovan och utgående från den diskussionen framgår det att beroende på hur skattningarna eller olika värden tagits fram finns det risker.

Fördelen med de sex typerna av risker ovan är att diskussionen kring risker kan fokuseras på ett område i taget.

Efter det att riskerna har identifierats skall varje risk bedömas avseende sannolikhet att den inträffar och effekt om den inträffar. Lämpligen sätts en tabell upp med riskerna, sannolikheter för att de inträffar och effekten om så sker. Ett enkelt exempel illustreras i Tabell 3.1. Sannolikheten anges ofta på skalan låg, medium och

hög. Effekten kan anges i obetydlig, liten, medium, allvarlig och katastrofal.

Tabell 3.1 Risksammanställning

Risk	Sannolikhet	Effekt
R1	Medium	Liten
R2	Låg	Allvarlig
R3	Låg	Katastrofal

Baserat på resultatet av denna genomgång måste möjliga åtgärder identifieras. Åtgärderna varierar beroende på den aktuella risken. Vidare beror även åtgärderna på hur kritiskt projektet är antingen ur ett affärsmässigtperspektiv eller därför att produkten i sig själv är säkerhetskritiskt, till exempel programvara till ett kärnkraftverk. I Tabell 3.1 är det risken där effekten är katastrofal som tilldrar sig störst uppmärksamhet. Det är denna risk som måste bevakas mest noggrant under projektets gång.

Slutligen är det mycket viktigt att regelbundet uppdatera riskanalysen. Det innebär dels att nya risker kan uppstå eller något har missats tidigare och dels att sannolikheter och effekter för identifierade risker eventuellt behöver förändras.

3.4.6 Möjlighetsanalys

Utgående från genomgången av mål, resurser, risker och så vidare avslutas definitionsfasen av ett projekt med en möjlighetsanalys. Tanken med denna analys är att utgående från kända fakta görs en bedömning huruvida det är möjligt att genomföra projektet. Denna del av projektförslaget skall fungera som en sammanfattning och utvärdering av förslaget som helhet. Möjlighetsanalysen är viktig indata för att besluta om projektet skall startas eller ej.

3.5 Planera

Efter det att beslut har tagits om att starta ett projekt behöver projektet planeras i mer detalj än vad som gjordes i samband med framtagandet av projektförslaget. Planeringen dokumenteras i en projektplan. Delar av projektförslaget kan eventuellt användas i projektplanen. Projektplanen är normalt en förfining av projektförslaget, det vill säga det är möjligt att finna mer detaljer i planen till exempel i form av nedbrytning av arbetet. Det viktigaste är att planen är komplett och inte utesluter saker därför att det står i projektförslaget. Planen är ett centralt dokument för projektet och därmed är det mycket viktigt att det innehåller all väsentlig information. En projektplan bör innehålla följande:

Introduktion
I denna bör målen återfinnas. Information om begränsningar skall också finnas med till exempel i termer av tid, budget och tillgänglig personal.

Projektorganisation
Varje projekt behöver en organisation. Det första steget är att identifiera viktiga roller i projektet. Därefter är det möjligt att utse ansvariga personer för de olika rollerna. Exempel på ansvariga kan vara systemarkitekt, utvecklingsledare och testledare. Den exakta organisationen beror i stor utsträckning på projektets storlek. I ett mindre projekt kan samma person vara ansvarig för ett antal roller, medan i ett större projekt är rollerna kopplade till olika personer.

Riskanalys
Riskanalysen från projektförslaget är utgångspunkt för denna del i planen. Varje risk måste dock bedömas igen och vidare skall eventuellt nya risker läggas till.

Krav på hårdvara och programvara
Ett utvecklingsprojekt har ofta krav på tillgång till viss programvara och hårdvara. Programvaran kan vara i form av till exempel utvecklingsverktyg eller kompilatorer. Hårdvarukraven kan vara krav på en viss testmiljö, till exempel i form av en miljö som är lik den miljö som den utvecklade programvaran skall finnas i.

Nedbrytning av arbete
Detta innefattar identifiering av aktiviteter och relationer mellan dem. Det inkluderar även att milstolpar och leverabler vid olika tidpunkter i projektet bestäms. Aktiviteterna bör ligga på en sådan nivå att det är så enkelt som möjligt att dels bestämma dess storlek i termer av timmar/dagar eller annan lämplig enhet och dels utse en ansvarig person. Aktiviteterna och beroende mellan aktiviteter listas lämpligen i en tabell, se exempel i Tabell 3.2, där tio aktiviteter har listats. Vidare anges vilka aktiviteter som är beroende av varandra, det vill säga vilka aktiviteter som måste vara klara innan den aktuella aktiviteten kan påbörjas.

Tabell 3.2 Aktiviteter och beroenden mellan dem.

Aktivitet	**Beror av**
A1. Skriv kravspecifikation	-
A2. Design enhet A	A1
A3. Design enhet B	A1
A4. Implementation enhet A	A2
A5. Implementation enhet B	A3
A6. Enhetstest enhet A	A4
A7. Enhetstest enhet B	A5
A8. Integrationstest	A4 och A5
A9. Systemtest	A8
A10. Skriv användarmanual	A2 och A3[1]
A11. Avstämning system och manual	A9 och A10

1 Här antas att det går att skriva manualen efter det designen är klar.

I exemplet kan ses hur designen av de båda enheterna är beroende av att kravspecifikationen är färdig. Efter designmomentet delas arbetet i två delar som kan ske parallellt tills integrationstest. Det senare kan dock inte påbörjas från det att båda enhetstesten har avklarats. Den sista aktiviteten är skrivandet av en användarmanual. Denna aktivitet anses kunna starta när designen är klar. Den kan dock inte avslutas förrän systemtestet är slutfört, då en avstämning måste ske så att det parallella arbetet mellan implementation/test och skrivandet av användarma-

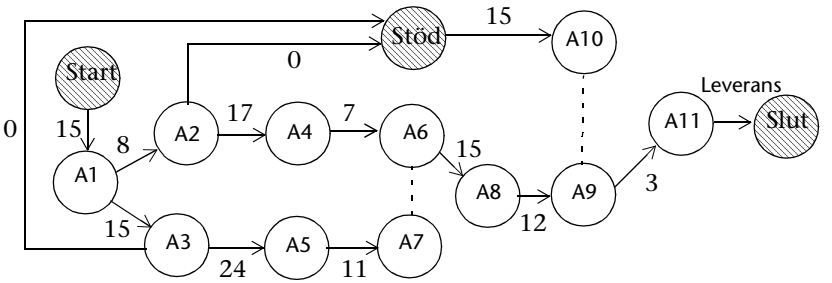
nualen inte resulterat i olikheter, det vill säga att manualen inte beskriver det som slutligen implementerades.

Tidplanering
Utgående från nedbrytning i föregående punkt skall nu aktiviterna planeras avseende både arbetstimmar och kalendertid. Till vår hjälp för att göra detta finns aktivitetsdiagram och Gantt-diagram. Aktivitetsdiagrammet utgår från den tabell som gjordes i föregående punkt. Det kan vara lämpligt att inleda med att komplettera tabellen ovan med tidsuppskattningar. I exemplet antar vi följande siffror (i dagar): 15, 8, 15, 17, 24, 7, 11, 15, 12, 10 och 3. Denna information förs sedan över till ett aktivitetsdiagram för att få en bättre översikt, se Figur 3.3. Beroendet mellan aktiviteter anges genom att aktiviteterna kopplas samman, jämför Tabell 3.2 och Figur 3.3. Längden på aktiviteterna anges på bågarna mellan noderna (aktiviteterna) för att illustrera övergången mellan aktiviteter. Speciellt skall noteras att tidsåtgången anges före aktiviteten. Noderna blir därmed slutpunkt på en aktivitet.

För att kunna beskriva aktiviteterna och deras beroenden grafiskt behövs en del stöd. Stödnoder har markerats med ett speciellt mönster för att markera att det inte är frågan om vanliga aktiviteter. Vidare används heldragna linjer mellan aktiviteter för att markera övergångar, medan beroenden kan lösas på olika sätt antingen genom införande av extranoder eller av streckade linjer. För det första behövs en startnod för att på ett naturligt sätt införa tidsåtgången för den första aktiviteten. Då aktivitet A2 och A3 dels skall leda till implementationen och dels till användarmanualen har en stödnod införts för att markera att både A2 och A3 måste vara klara innan manualen kan påbörjas. Detta har hanterats genom införandet av en stödnod med en längd på noll tidsenheter.

När två eller flera aktiviteter skall vara klara innan en gemensam aktivitet kan påbörjas brukar detta anges med streckade linjer mellan aktiviteterna och den egentliga övergången till nästa aktivitet markeras från en av noderna, se till exempel noderna A9, A10 och A11. Lösningen i anslutning till dessa noder fungerar inte i samband med A2 och A3, då de dels båda skall vara klara för att starta A10 och dels startar andra aktiviteter oberoende av varandra. Slutligen har en slutnod införts för att markera att pro-

jektet är avslutat. I detta specifika exempel innebär det att programvaruprodukten kan levereras.



Figur 3.3 Aktvitetsdiagram.

Aktivitetsdiagrammet gör det också enklare att identifiera den kritiska linjen i projektet. Den kritiska linjen utgörs av de aktiviteter som inte kan försenas utan att projektet försenas. För att identifiera den kritiska linjen måste diagrammet gås igenom och den väg som tar längst tid noteras. Speciell hänsyn måste tas till de beroenden som finns. I diagrammet kan ses att aktivitet A10 inte kan startas förrän A1, A2 och A3 är klara. Den längsta vägen är A1 och A3 som totalt tar 30 dagar. Det betyder att A10 kan vara klar efter 45 dagar. Detta skall nu jämföras med de vägar som leder till A9, men innan detta kan göras måste beroendet mellan A6 och A7 utredas. A6 är klar när A1, A2, A4 och A6 har utförts. Denna väg tar 15+8+17+7 = 47 dagar. A7 å andra sidan är klar när A1, A3, A5 och A7 utförts. Detta tar sammanlagt 65 dagar. Den senare är klart längre och därmed kan A9 vara klar efter 65+15+12 = 92 dagar. Denna väg är avsevärt längre än via A10 och därmed blir det längsta vägen via A9 som styr när A11 kan starta. den kritiska linjen blir således A1, A3, A5, A7, A8, A9 och A11 som sammantaget beräknas ta 95 dagar.

Identifieringen av den kritiska linjen ger också möjlighet att avgöra hur mycket andra aktiviteter kan förskjutas avseende starttidpunkt eller bli försenade utan att påverka sluttidpunkten. Denna analys är baserad på att inte samma person behöver utnyttjas för parallella aktiviteter. Därmed skall analysen avseende kritisk linje ses som en verktyg för att göra personalplane-

ringen. Om det vid personalplaneringen framkommer att till exempel samma person behövs för både A2 och A3, då får analysen av den kritiska linjen göras om baserat på denna kunskap. Å andra sidan kan analysen av den kritiska linjen användas för prioritering av aktiviteter och planering av personal. Ett annat exempel på användningen av analysen av den kritiska linjen är att det kan noteras att det finns 47 dagars marginal i vägen som går via A10, det vill säga (om skattningarna är korrekta) finns det möjlighet att vänta med skrivandet av manualen i 47 dagar innan det blir kritiskt för hela projektet.

Aktivitetsdiagrammet kompletteras lämpligen också med milstolpar. I samband med planeringen kan de milstolpar som bestämdes i samband med definitionen av projektet förfinas och göras mer konkreta.

Informationen från aktivitetsdiagrammet kan föras över i ett Gantt-diagram som lägger in aktiviteterna i kalendern. Diagrammet görs genom att aktiviteterna listas till vänster och överst anges kalendern. Denna planering innebär att det finns möjlighet att dra nytta av informationen från analysen av den kritiska linjen, det vill säga omplanering i tiden kan göras med hänsyn till tillgänglig personal. I diagrammet brukar även anges den tid som en aktivitet kan fördröjas utan att det påverkar den kritiska linjen.

Uppföljning och rapportering

I samband med planeringen måste även bestämmas hur uppföljning och rapportering skall ske. Detta inkluderar hur ofta, men även hur noggrant rapportering skall ske samt rapporteringsvägar. Det första betyder att beslut måste tas om uppföljning sker till exempel varje dag eller varje vecka. Noggrannheten innebär att beslut måste tas på vilken detaljnivå rapportering sker. Räcker det att programmering rapporteras eller bör rapporteringen även ange på vilka delar programmeraren arbetat? Slutligen måste rapporteringsvägar bestämmas, det vill säga vem rapporterar till vem? Går all rapportering direkt till projektledaren eller sker sammanställning av ett antal individer? Allt detta är viktiga saker som måste bestämmas innan ett projekt startar annars är det stor risk att olika personer kommer att hantera det på olika sätt och det gör att en enhetlig uppföljning inte kan göras.

Projektplanen bör kompletteras med information angående hur avsikten är att säkra kvaliteten (kvalitetsplan) och dokumenthantering (konfigurationsplan). Kvalitetsplanen anger vilka aktiviteter som skall genomföras för att säkerställa kvaliteten. Det kan vara i form av projektgranskningar och produktgenomgångar. Den förstnämnda syftar på att en genomgång görs av hur projektet framskrider. Följer det planen? Har projektledaren bra kontroll på uppföljning och riskhantering? Följs de arbetsrutiner som gäller för projektet? Detta är exempel på några frågor kopplade till projektet. Det är också möjligt att göra produktgenomgångar. Dessa skall dock inte blandas samman med de kvalitetshöjande aktiviteter som genomförs inom ramen för projektet, utan produktgenomgångar skall vara oberoende av det egentliga utvecklingsarbetet. Syftet skall vara att få en mer utomstående syn på produkten som är under framtagning. Dessa typer av genomgångar görs företrädesvis av personer som inte direkt arbetar i projektet, utan ofta kan tillhöra någon allmän funktion på företaget.

Konfigurationshanteringen är viktig i alla projekt. Hanteringen avser att av säkerställa att det finns kontroll på olika dokument, inklusive programkod, avseende olika versioner. Konfigurationshantering presenteras utförligare i Kapitel 4. Syftet med planen är att det skall klargöras hur dokument och program hanteras inom projektet. Detta inkluderar backup, ansvar för olika dokument och program, rutiner för hämtning och inläggning av dokument och program i konfigurationshanteringssystemet samt vem som är ansvarig för att godkänna ett specifikt dokument eller program. Backuprutiner är viktiga, då arbete lätt kan gå förlorat om något datorproblem inträffar och backup saknas. I värsta fall kan det innebära betydande förseningar. Det är också viktigt att en ansvarig person utses för de olika dokument och program som skall produceras. Vidare skall det finnas en annan person som är ansvarig för att godkänna resultatet. Godkännande sker efter det att planerade kvalitetshöjande åtgärder är genomförda enligt plan. Det är även viktigt att det finns tydliga rutiner för vem som får hämta ut ett dokument eller program och ändra i det samt hur ofta det uppdaterade dokumentet eller programmet måste läggas tillbaka i konfigurationshanteringssystemet. Det finns många system för konfigurationshante-

ring. Valet av system bestäms utgående från de krav som finns från projektet.

Konfigurationshantering är även en fråga för organisationen, då det är organisationen som måste hålla reda på olika versioner av programvaran (om sådana finns) och vilka releaser som finns. Det senare syftar på en version som är tänkt att ersätta tidigare releaser. Som exempel kan nämnas att operativsystem finns i olika versioner beroende på vilka uppdateringar en användare har laddat hem. De finns också i olika releaser till exempel Windows 2000 respektive XP.

3.6 Organisera

Organisationen av ett projekt är en direkt följd av planeringen. I många fall är det den blivande projektledaren som gör projektplanen och därmed är ledaren redan utsedd. Baserat på analysen som genomfördes i samband med planeringen av projektet och prioriteringen av projektet i förhållande till andra projekt inom företaget kan projektet bemannas. Bemanningen görs genom avvägande kring behov av specialistkompetens, tidigare erfarenheter och tillgänglighet på personal. När det gäller erfarenheter är det speciellt intressant med erfarenhet i anslutning till den typ av produkt som är aktuell i det nya projektet och de metoder och tekniker som skall användas.

Aktiviteterna som identifierades under planeringen tilldelas personer. Eventuellt kan det vara aktuellt att bryta ned aktiviteterna ytterligare för att olika personer skall kunna utföra olika delar av en aktivitet. Organisationen av ett projekt beror i hög grad av det specifika fallet. Det innebär att det viktigaste är att organisationen är genomtänkt och baserad på fakta till exempel i form av tillgänglig personal. Det är nämligen oftast inte möjligt att få tillgång till exakt de personer som önskas eller åtminstone inte exakt vid den tidpunkt som de egentligen behövs. Kunniga och erfarna individer burkar ha en tendens att vara uppbokade och det krävs noggrann planering för att dra bästa möjliga nytta av dem.

3.7 Genomföra

Framgången i genomförandet beror i hög grad på en god planering, men det räcker inte. Projektledaren måste vara beredd att styra projektet för att målen skall uppnås. Rollen som projektledare kan jämföras med att vara coach för ett lag. Projektledaren skall eftersträva att få ut det bästa möjliga ur de individer som deltar i projektet samtidigt som projektdeltagarna strävar efter det gemensamma målet. Detta ställer stora krav på projektledaren i form av social förmåga.

En projektledare skall under genomförandet:
- Lyssna på projektdeltagare och förstå dem.
- Motivera och uppmuntra sina medarbetare.
- Möjliggöra arbetet snarare än att styra det.
- Våga ge spelrum för individerna inom projektets ramar.
- Inse skillnader i olika personligheter, samt sedan kunna förstå och hantera dem.
- Vara tålig avseende fel och problem som uppträder under projektets genomförande.
- Agera buffert mot chefer och yttre händelser.
- Hantera konfliktlösning.
- Vara en medarbetare med speciellt ansvar hellre än chef.
- Vara målmedveten och säkerställa att alla arbetar mot det gemensamma projektmålet.

Sammanfattningsvis skall en bra projektledare ha god förmåga att kunna kommunicera och hantera människor. Projektarbete är i stor utsträckning en social aktivitet som skall ledas och hanteras av projektledaren.

En viktig aktivitet under genomförandet är uppföljning av projektet. Uppföljningen innebär att framåtskridandet skall noteras och speciellt viktigt är eventuella avvikelser från planen. Vid avvikelser måste projektledaren vara beredd att genomföra åtgärder för att komma tillrätta med avvikelserna. Det kan finnas många olika anledningar till avvikelser och de är (tyvärr) en naturlig del av varje projekt. Det är dock ingen ursäkt för att inte ha en så bra plan som möjligt som det sedan är viktigt att följa upp och uppdatera. Avvikelser från plan kan bero på underskattning av problemet, persona-

len var inte tillgänglig vid rätt tidpunkt eller yttre faktorer som ändrade förutsättningar eller sjukdom.

Vid avvikelser från planen, vilket alltför ofta är förseningar måste projektledaren utreda och besluta om planen går att uppdatera inom givna ramar eller om en omplanering krävs. Det senare kan omfatta allt från ändring av leveransdatum till förhandling av det funktionella innehållet. Det är speciellt viktigt med utvärdering och eventuell omplanering av ett projekt i samband med milstolparna.

Uppföljning av projektet bör göras regelbundet och denna regelbundenhet bör bestämmas redan i samband med planeringen liksom rapporteringsvägarna. Det är lämpligt att rapporteringen görs elektroniskt så att siffror för olika personer för samma aktivitet kan slåss samman på ett enkelt sätt. Det skapar en god översikt för projektledaren att kunna göra sammanställningar i olika dimensioner, till exempel per person eller per aktivitet.

Tidsuppföljning och även rapportering kring utvecklingens framåtskridande, speciellt huruvida planerade milstolpar har uppnåtts, är viktig indata för den statusrapportering som en projektledare ofta måste göra. Projektledaren kan behöva rapportera till linjechefer eller till kunder.

3.8 Avsluta

Betydelsen av den sista fasen i ett projekt är ofta underskattad. Fokus är inledningsvis på att leverera och installera den nya produkten. Det kan innebära att produkten installeras hos en specifik kund eller göra tillgänglig på en marknad. När detta är gjort upplevs projektet ofta som avslutat och nya projekt väntar på personalen. Tyvärr innebär detta då att erfarenheterna från projektet oftast inte dokumenteras på ett bra sätt.

Ett projektavslut bör innehålla framtagning av en slutrapport och en slutgranskning av projektet bör göras. Den senare kan antingen rapporteras separat eller vara en del av slutrapporten. Slutgranskningen bör utgå från den initiala projektplanen och sedan följa projektets genomförande. Det är viktigt att notera när förändringar, till

exempel förseningar inträffat och deras orsak, skett. Det är även väsentligt att följa upp de risker som inledningsvis identifierades. Vilka av dessa inträffade? Vad missades i riskanalysen? Projektets mål måste utvärderas. Nåddes målet med projektet? I princip är det viktigt att dokumentera alla erfarenheter, både positiva och negativa. Slutrapporten skall fungera som organisationens minne och historieskrivning av projektet.

Genomförda projekt är en guldgruva för att lära för framtiden. Det är viktigt att lära både från misstag och från framgångar. Ofta sägs det att det är viktigt att lära från sina fel, men det är lika viktigt att lära från framgångarna. Vad gjordes bra, det vill säga vad lade grunden för framgången? Lärdomarna för genomförda projekt är viktiga indata både till kommande projekt och till allmän förbättring av organisationen.

Slutligen är det viktigt att försöka samla projektdeltagarna och göra ett formellt avslut av projektet.

4 Organisation

4.1 Inledning

Organisationer som utvecklar programvara är ofta organiserade i en projektorganisation som nämndes i Kapitel 3. Det innebär att organisationen är nedbruten i ett antal avdelningar, sektioner, enheter eller vad som har valts som beteckning. För att förenkla kallar vi dem enbart enheter. Enheterna kan vara av två huvudtyper: personalresurser till projekt eller mer stabsorienterade enheter. Det senare kan vara allmänna administrativa resurser, ekonomifunktion eller infrastrukturenheter. I detta sammanhang är de senare av störst intresse. Infrastruktur innebär företagsgemensam struktur, till exempel i form av datorstöd, datanät, utvecklingsprocesser, kvalitetssystem och konfigurationshantering. Datorer och datanät sköts vanligtvis av en dataavdelning eller systemansvarig beroende på företagets storlek. Utvecklingsprocesser, kvalitetssystem och konfigurationshantering hanteras ofta av en metodavdelning och används sedan i någon form inom ramen för ett utvecklingsprojekt.

Olika typer av utvecklingsprocesser behandlas i Avsnitt 4.2. Kvalitetssystem behandlas kortfattat i Avsnitt 4.3 tillsammans med en kort diskussion om behovet av en kontinuerligt lärande organisation för att ständigt kunna vara konkurrenskraftiga. Avsnitt 4.4 beskrivs konfigurationshantering och Avsnitt 4.5 tar upp processförbättring. Kapitlet avslutas med ett avsnitt om kunder och marknader i Avsnitt 4.6. Det senare har placerats i detta kapitel om organisationen, då det normalt inte är respektive projekts ansvar att identifiera kunder och marknader för de produkter som utvecklas. Projekten kan ha kontakter med kunder och marknader, men det övergripande ansvaret ligger på organisationen eller företaget.

4.2 Utvecklingsprocess

Utvecklingsprocesser och andra gemensamma infrastrukturer i form av kvalitetssystem med mera ses tyvärr alltför ofta som ett hinder istället för ett stöd. I många projekt upplevs att det skulle gå snabbare att bli klara om de ”bara” fick utveckla produkten.

Problemet med denna syn är att det skulle göra det mycket svårt för andra vid företaget att förstå programvaran i ett senare skede. Det skulle också vara svårt för personal att flytta mellan olika projekt. För en organisation skulle det innebära stora risker och ett stort beroende av enskilda utvecklare. Personalen är naturligtvis en viktig del och varje organisation måste ha som mål att ha största möjliga nytta av sin personal. Men det är inte detsamma som att riskera att inte kunna leverera om en enskild person inte finns tillgänglig längre, till exempel på grund av att personen har lämnat organisationen. Sist men inte minst skulle det innebära att varje projekt skulle behöva lägga innovativkraft på att fundera på hur de skulle arbeta.

Den gemensamma processen skall fungera som ett ramverk för arbetet och den skall göra att personalen i projektet inte behöver fundera på vad som skall göras härnäst. Vidare skall de inte behöva fundera på vilka dokument som skall tas fram. Tanken är att den innovativa kraften skall läggas på produkten och inte på hur arbetet skall genomföras. Detta är målet samtidigt som det skapar en standard för företaget.

Många duktiga programmerare tycker dock att processen är ett hinder och har svårt att förstå behovet. Det kan dock konstateras att alla inte är lika duktiga och därmed finns det många som behöver stöd. Vidare har organisationens storlek också betydelse. I en liten organisation vet alla vad som skall göras och vad olika personer kan. I sådana fall är det ibland inte möjligt att förlita sig helt och hållet på kunskapen hos personalen. Men vad händer när organisationen växer eller nyckelpersoner slutar? I dessa fall är det ytterst väsentligt att det finns företagsgemensamma standards och att dessa följs.

Sammanfattningsvis behövs en balans mellan bra utvecklingsprocess, en god struktur på produkten samt kompetent och engagerad

personal. Vi återkommer till de två sistnämnda i kommande kapitel och fokuserar här på utvecklingsprocessen. Denna process är nära besläktad med en produkts livscykel. Den senare är dock bredare och innefattar utveckling och hantering av olika versioner av produkten tills den slutligen också tas ur drift.

Det finns en rad olika förslag på utvecklingsprocesser. Ett urval av de viktigaste beskrivs i de olika delavsnitten nedan. Det är dock viktigt att notera att processerna har en hel del gemensamt, vilket inte är konstigt. Varje process måste börja med kraven för att baserat på dem genomföra utvecklingen och avsluta med att testa att programvaran fungerar, vilket i sin tur leder till att programvaruprodukten kan levereras. Det är med anledning av detta inte några revolutionerade skillnader mellan processerna, utan det rör sig i första hand om hur olika saker betonas.

Utvecklingsmodeller delas normalt in i ett antal faser, huvudaktiviteter eller utvecklingssteg som kravhantering, design, implementation, test och underhåll. Dessa steg som ingår på ett eller annat sätt i alla utvecklingsprocesser beskrivs närmre i Del 3, där även granskning och verktygsstöd som båda är viktiga ingredienser genom många utvecklingsprocesser tas upp.

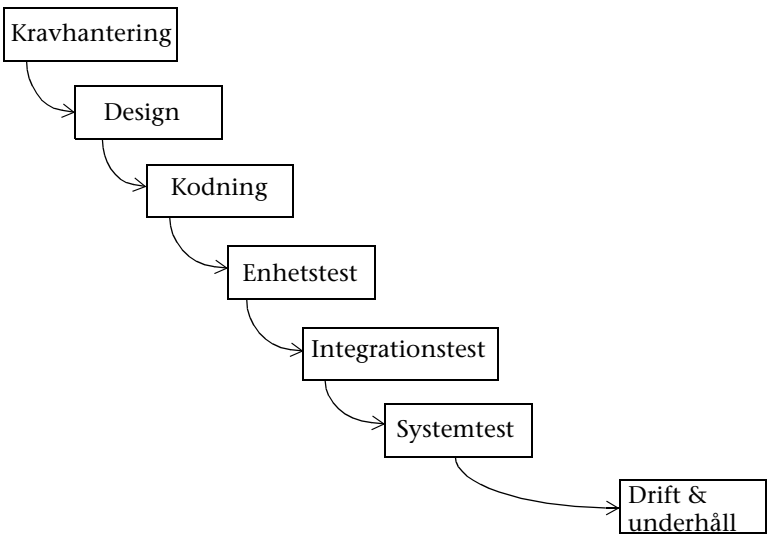
Det finns få helt standardiserade processer som företag använder. De flesta organisationer använder en modell som är baserad på en principiell modell, det vill säga som de som visas nedan. Varje organisation har dock sin egen variant, där namnen på stegen och vad som skall uträttas och tas fram i varje steg kan variera högst väsentligt. Många organisationer har ingångs- respektive utgångskriterier från respektive steg i utvecklingen. Kriterierna kopplas ofta samman med att olika dokument skall vara framtagna. Dessa kriterier kopplas sedan ofta till de milstolpar som respektive projekt ställer upp, se Kapitel 3.

4.2.1 Vattenfallsmodell

Vattenfallsmodellen kallas så på grund av det sätt som den initialt ritades på. Ett exempel på vattenfallsmodellen illustreras i Figur 4.1. Det finns många olika varianter av modellen med olika utvecklings-

faser och olika namn på dem. Grundtanken är dock en och densamma. Tanken bakom modellen är att efter varje genomförd fas sker en övergång till nästa fas. Modellen förutsätter att varje fas avslutas en gång för alla och att det därmed inte är möjligt att gå tillbaka, det vill säga det finns inga pilar tillbaka mellan faserna i figuren. Detta är vanligtvis helt orealistiskt, då det är mycket vanligt att problem upptäcks. Det betyder att det finns ett stort behov av iteration mellan faserna. En alternativ tolkning av modellen som stämmer bättre med verkligheten är att vattenfallsmodellen visar i vilken ordning faserna startar och egentligen ingenting om när de avslutas.

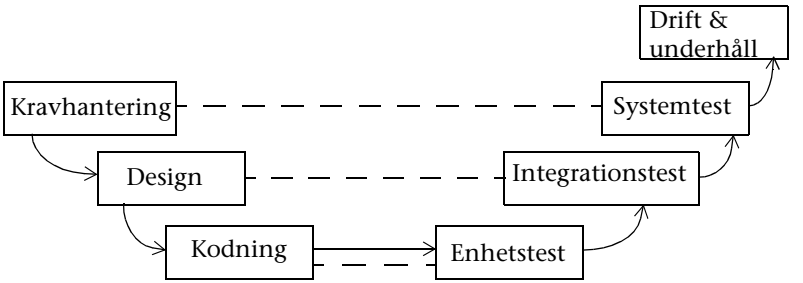
Det kan vara värt att notera att faserna i vattenfallsmodellen ibland har andra namn än de som återfinns i Figur 4.1. Vidare benämns ibland kodning och enhetstest tillsammans som implementation. Detta begrepp används bland annat nedan. Anledningen är att de två faserna ofta genomförs tillsammans och av samma person eller personer.



Figur 4.1 Exempel på vattenfallsmodellen.

Olika förändringar av modellen har gjorts. De inkluderar iteration och även att modellen har ritats som en V-modell för att illustrera hur testaktiviteterna kan kopplas samman med utvecklingsaktivi-

terna. V-modellen illustreras i Figur 4.2. De streckade linjerna anger hur avsikten är att testaktiviteterna skall kopplas samman med utvecklingsaktiviteterna. Det betyder att avsikten är att kodningsfel skall hittas i enhetstest, designproblem skall detekteras när integrationstest genomförs, det vill säga när olika systemdelar sätts samman, och slutligen skall systemtest tillgodose att kraven är uppfyllda.



Figur 4.2 Exempel på V-modellen.

Vattenfallsmodellen är utgångsmodellen och andra modeller bygger ofta på att svagheter i vattenfallsmodellen har identifierats. Den vanligaste industriella utvecklingsmodellen är dock fortfarande varianter av vattenfallsmodellen. Huvudanledningen är att den är enkel och den är lätt att koppla samman med milstolparna i ett projekt.

4.2.2 Inkrementell utveckling

En utvecklingsprincip som bygger på vattenfallsmodellen är inkrementell utveckling. Det inkrementella utvecklingssättet bygger på att det tidigt under utvecklingen identifieras ett antal inkrement som kan fungera som delleveranser, där en delleverans ger en del av funktionaliteten som produkten skall tillhandahålla. Det är dock inte nödvändigt att inkrement levereras till en kund. Det kan vara ett bra sätt för att få en smidig utveckling, där test på tidiga inkrement kan göras parallellt med att nya inkrement utvecklas. En möjlig målsättning kan vara att snabbt kunna få ut något från varje projekt och även erhålla synpunkter tillbaka till exempel från en kund

huruvida det utvecklade stämmer med kraven och förväntningarna.

Varje inkrement utvecklas ofta baserat på vattenfallsmodellen. Fördelen är dock att mindre delar kommer till test när inkrementell utveckling används. Det gör också att det är möjligt att prioritera vad som skall ingå i respektive inkrement och att viktiga delar av en produkt kan komma ut till kunden eller marknaden snabbare om så önskas.

Inkrementell utveckling kan sammanfattas med att målet är att leverera färdig funktionalitet för en del av produktens totala funktionalitet i varje inkrement.

Svårigheten med denna typ av utvecklingen är att det kan vara svårt att identifiera lämpliga inkrement. Detta kan bero på att många av de olika funktionerna i produkten är beroende av varandra.

4.2.3 Evolutionär utveckling

Evolutionär utveckling eller iterativ utveckling är nära besläktat med inkrementell utveckling. Det finns dock en viktig principiell skillnad. I inkrementell utveckling är målet att ta fram fungerande delar, men i evolutionär utveckling är målet hela tiden att arbeta med hela produkten. Det finns två olika typer av evolutionär utveckling, nämligen utredandeutveckling och utveckling av en prototyp.

Vid utredandeutvecklingen är ofta målsättningen att prova om det går att utveckla en produkt på ett specifikt sätt eller med en specifik teknik. En annan möjlighet är att arbetet sker i nära samarbete med en kund för att utvärdera om en viss typ av produkt kan lösa deras problem. Det betyder att utvecklingen ligger till grund för identifiering av krav.

Tanken är egentligen inte att en utredandeutveckling skall leda till en produkt, utan att det i första hand skall vara en utredning. I fall när det visar sig vara en bra lösning händer det ofta att utvecklarna använder det som framkommit som grundstomme för ett system.

Prototyputveckling är närbesläktat med en utredning, men i detta fall är målet inte att utreda och identifiera krav. Målet är oftast att göra utvecklingen för att förstå att rätt sak utvecklas. Byggandet av en prototyp brukar innebära att prototypen inte används. Detta är standard när det gäller till exempel utveckling av en ny bilmodell. Vem vill köpa en prototyp? I programvarusammanhang händer det däremot ofta att prototypen inte läggs åt sidan utan att den vidareutvecklas till den slutliga produkten. I grund och botten betyder det att termen prototyp är rätt missvisande.

Prototyputveckling är mycket bra att använda för utvärdering av användargränssnitt. Det betyder att ett gränssnitt utvecklas för att visa hur den slutliga produkten kan se ut. Funktionaliteten bakom gränssnittet är dock mycket begränsat.

4.2.4 Spiralmodell

En speciell variant av iterativ utveckling är spiralmodellen. Den försöker kombinera riskhantering med iterativ utveckling och den är även influerad av vattenfallsmodellen. Modellen bygger på följande nyckelord:

- Budget
- Bestäm mål
- Alternativ
- Begränsningar
- Riskanalys
- Prototyp
- Utveckla och testa

Modellen är byggd kring en spiral som består av fyra varv och fyra kvadranter. I den första kvadranten och innersta varvet inleds med att budget för projektet bestäms. Baserat på detta sätts mål, dels för helheten och dels för det första varvet. Utgående från målen identifieras olika alternativa lösningar eller angreppssätt. Vidare identifieras begränsningar. Detta genomförs i första kvadranten på det första varvet.

I den andra kvadranten utvärderas alternativen och en riskanalys genomförs. Detta görs genom prototyputveckling. För denna

modell bör prototyp tolkas i bred mening. Det betyder inte nödvändigtvis att olika versioner byggs för de olika alternativen som identifierats utan det betyder att tillräckligt med information tas fram för att kunna göra en ordentlig utvärdering.

De två första kvadranterna i spiralmodellen är likadana för samtliga fyra varv naturligtvis med lämplig förfining baserat på arbetet som utförts under tidigare varv i modellen. I den tredje och fjärde kvadranten skiljer sig de olika varven åt i form av aktiviteter. I tredje kvadranten på första varvet tas en övergripande modell fram som beskriver hur produkten skall fungera. Modellen är på en hög nivå och skall fungera som indata för att kunna bestämma de mer specifika kraven på produkten. I den fjärde kvadranten tas krav på produkten fram och en livscykelplan för produkten skrivs. Den senare kan tala om vilka planer som finns för produkten efter den första versionen och vilken vidareutveckling som finns i sikte.

På andra varvet genomförs den första och andra kvadranten baserat på den information som har tagits fram under det första varvet. I den tredje kvadranten bryts de allmänna produktkraven ned till krav på programvaran. Det sker också en avstämning av kraven gentemot kund eller marknad. Utgående från kraven görs en utvecklingsplan i fjärde kvadranten.

Det tredje varvet inleds på samma sätt som tidigare, men baserar sig nu på kraven. I den tredje kvadranten genomförs design och i fjärde kvadranten planeras för integrationstest. På det fjärde varvet görs de två första kvadranterna som tidigare. Detaljerad design, kodning och test genomförs sedan i den tredje kvadranten och slutligen sker leverans i den fjärde kvadranten.

Den uppmärksamma läsaren ser omedelbart likheten mellan spiralmodellen och de tidigare introducerade modellerna. Vi ser projektfokus via budget, mål och riskanalys. Aktiviterna i tredje och fjärde kvadranten ligger i linje med aktiviteterna i vattenfallsmodellen. Slutligen har spiralmodellen stora likheter med evolutionär utveckling genom de olika varv som finns i modellen och som syftar till att hela tiden genomföra utvärderingar via prototyper.

Denna modell upplevs dock ofta som komplex och svår så även om den inkluderar mycket som finns i de andra modellerna är använd-

ningen av den begränsad. Det skall dock tilläggas att många av de varianter som finns av vattenfallsmodellen gör att gränsdragningen mellan vattenfallsmodellen och övriga modeller blir svår. Många organisationer har baserat sina egna utvecklingsmodeller på vattenfallsmodellen, men anpassningarna som har gjorts har ofta inneburit att de har närmat sig de andra modellerna.

4.2.5 Lätta processer (till exempel XP)

En typ av utvecklingsmodeller som vunnit gehör under 2000-talet är ”lätta processer” (eng. agile processes). Detta är en typ av modeller som på sitt sätt är en reaktion på att utvecklingsprocesserna ofta upplevs tunga och hämmande. Grundtanken är att minska merarbetet i form av framtagning av dokument och öka flexibiliteten i utvecklingen. Flera olika angreppssätt har föreslagits, där det mest omtalade är ”extrem programmering” (eng. extreme programmering) som ofta enbart benämns som XP. Denna metod sätter programmeringen i fokus och kan anses definierad från programmerarens perspektiv, medan tidigare modeller ofta ses som definierade från ett managementperspektiv.

XP är i första hand utvecklat för team av utvecklare. XP utgår från att det finns fyra variabler i programvaruutveckling, nämligen:

- Kostnad
- Ledtid
- Kvalitet
- Funktionalitet, det vill säga produktinnehåll

I varje projekt kan dessa fyra avvägas mot varandra. Enligt XP skall inte kunden kunna bestämma mer än maximalt tre av ovanstående. Den fjärde måste sättas av utvecklarna baserat på övriga tre. XP betonar också fyra viktiga aspekter i arbetet:

- Kommunikation, det vill säga vikten av att ha en god kommunikation mellan utvecklarna. Hypotesen är att många projekt misslyckas för att det är för dålig kommunikation mellan människorna.
- Enkelhet, där syftet är att ta fram den enklaste lösning som kan fungera. Inom XP finns ett starkt fokus på att enbart utveckla

exakt det som efterfrågas och att inte bry sig om eventuella förändringar i framtiden. Detta motiveras med att det är bättre att utveckla det som behövs i framtiden när det inträffar istället för att förbereda för ett antal saker som eventuellt inte händer.

- Återmatning, det vill säga synpunkter på det utförda arbetet eller bristen på utfört arbete måste kommuniceras tillbaka till den person som var ansvarig för arbetet. Denna aspekt är nära förknippad med god kommunikation.
- Mod, det vill säga det är viktigt att våga göra saker. Det är uppenbart att något blir fel, men det är bättre att rätta detta senare än att inte våga. Detta hänger samman med en gammal klyscha: ”den som inte gör något gör heller inget fel”. Det är dock föga produktivt för ett företag att arbeta på detta sätt.

Processen bygger på tolv principer:

- Planeringsspelet: innehållet i nästa release måste bestämmas genom att kombinera affärsmässiga prioriteringar och tekniska skattningar.
- Små releaser: sätt ett enkelt system i drift och sedan är det bättre att göra små releaser ofta. Detta gäller såväl interna releaser och releaser som går till kund.
- Metafor: det skall finnas en enkel metafor för hela systemets funktion så att alla har samma förståelse för målsättningen med systemet som utvecklas.
- Enkel design: systemet skall hela tiden ha en så enkel design som möjligt. Om onödig komplexitet upptäcks skall den tas bort snarast.
- Testning: programmerarna skriver testfall för enhetstest och kunderna för funktioner. XP är en metod som trycker på att testfallen skrivs innan koden. Detta görs för att testfallen skall dokumentera det som programmet skall klara och sedan finns det ingen anledning att skriva något i programmet som inte finns i testfallen.

- Omstrukturering (eng. refactoring): programvaran omstruktureras kontinuerligt för att ta bort överflödig kod, se till att koden är enkel och för att underlätta kommunikationen i ett XP-projekt.
- Parprogrammering: all kod skrivs gemensamt av två programmerare vid en dator. Den som ser en bra lösning programmerar och den andra studerar lösningen konstruktivt för att säkerställa god kvalitet.
- Gemensamt ägandeskap: alla personer i ett XP-projekt har rätt att ändra i all kod. Det betyder att ingen enskild programmerare har ensamrätt på att ändra i vissa delar.
- Kontinuerlig integration: programmet kompileras och länkas flera gånger varje dag. Målsättningen är att detta skall innebära att det hela tiden finns ett fungerande aktuellt program.
- 40-timmars vecka: övertid skall undvikas. Det är endast negativt för produktiviteten. Det är inte tillåtet att arbeta över mer än en vecka i taget, det vill säga inte två veckor med övertid i följd.
- Närvarande kund: kunden skall alltid finnas på plats för att möjliggöra att frågor kan ställas. Det gör också att det är troligare att programmet i slutänden verkligen är det som kunden vill ha.
- Kodningsstandard: för att säkerställa kommunikation genom koden och underlätta att alla kan ändra i koden föreskriver XP att det skall finnas en kodningsstandard.

Ovanstående innehåller många sunda principer. Det finns dock några svårigheter som bör observeras. XP bygger på att utvecklarna kan kommunicera enkelt med varandra. Detta begränsar storleken på projekt där XP kan användas. Vidare finns det en viss risk att XP i första hand passar för en grupp av människor som känner varandra mycket väl och som är mycket duktiga programmerare. Detta är frågeställningar som forskningen behöver studera närmre.

Det finns idag många organisationer som använder en del av principerna i XP, men inte alla. I detta sammanhang är det relevant att fråga sig vad som krävs för att ett projekt skall betraktas som ett XP-projekt.

Utvecklingsprocessen är en viktig faktor för att säkra kvaliteten på programvaran, det vill säga hypotesen är att en bra process ger en bättre produkt. I nästa avsnitt diskuteras vad organisationer mer gör för att säkra kvaliteten.

4.3 Kvalitet i organisationen

Organisationen har inte enbart ansvar för den övergripande utvecklingsprocessen, utan den är även ansvarig för den långsiktiga förbättringen. En organisation är tänkt att vara långsiktig, medan projekten svarar för de projekten som har en begränsad varaktighet. Projekten sätter sina ”kortsiktiga” mål i fokus och skall så göra. Därför behövs en motvikt i form av organisationen som arbetar med det långsiktiga, vilket inkluderar både tillhandahållande av en arbetsmiljö och infrastruktur till projekten samt kontinuerlig långsiktig förbättring. Organisationen måste se till att den är konkurrenskraftig på sikt och inte enbart att nästa projekt går bra.

Det finns mycket litteratur om lärande organisationer och avsikten är inte att gå in på den. Det är dock viktigt att veta att det finns mycket som är relevant inom organisationsteori och lärande organisationer när det gäller produktutveckling i allmänhet och programvara i synnerhet. Lite förenklat kan sägas att en lärande organisation på ett systematiskt och strukturerat sätt tar vara på de erfarenheter som finns. Detta betyder i det ideala fallet att varje enskild person eller projekt inte skall behöva ha samma problem som upplevts tidigare i organisationen, utan den erfarenheten skall ha sparats för återanvändning. Sammantaget innebär att varje organisation som syftar till kontinuerlig förbättring måste bli en lärande organisation. Det måste finnas rutiner och arbetssätt som stödjer lärande från både misstag och framgångar.

Ett stöd för att lära och säkra kvalitet är de kvalitetssystem som finns inom organisationer. De tillhandahåller rutiner och processer för säkring av kvalitet. Det är sedan lämpligt att en person inom ett projekt är kvalitetsansvarig. Det kan vara en fördel om det inte är projektledaren som ofta har ett starkt fokus på tidplaner och kostnader, vilket kan leda till att kvalitetsfrågor hamnar längre ned på

dagordningen. Som stöd för en organisations allmänna kvalitetssystem finns internationella standards. Den mest välkända är ISO 9000 som är en allmän kvalitetsstandard. Det finns sedan stöddokument som beskriver hur till exempel ISO 9000 bör tillämpas för programvaruutveckling.

En central fråga för kvalitetshanteringen är konfigurationshanteringen.

4.4 Konfigurationshantering

Konfigurationshantering syftar till att hålla reda på dokument, kod, olika versioner och releaser av ett programvarusystem. För att underlätta benämns alla saker som behöver konfigurationshanteras för artefakter. Det är oerhört viktigt att ha en god kontroll på sina artefakter. Det gäller allt ifrån att veta vilken som är senaste versionen av ett dokument till att veta vilken version av programvaran som en specifik kund har. För att kunna hantera detta behövs ett system, det vill säga ett konfigurationshanteringssystem. De fyra viktigaste aktiviteterna i konfigurationshantering är:

- Planering av konfigurationshantering
- Ändringshantering
- Hantering av versioner och releaser
- Systembyggande

Planeringen innebär att en plan tas fram. Det kan finnas allmänna riktlinjer inom en organisation. Om det inte finns måste varje projekt ta fram en fullständig plan. En plan bör innehålla följande information:

- Det skall framgå vilka artefakter som skall konfigurationshanteras. I Avsnitt 3.1 introducerades tre olika typer av dokument: produkt, projekt och organisation. Ur ett projektperspektiv är det viktigaste produktdokumentationen, då den har en varaktighet som går bortom ett projekt. Projektdokumentationen behövs oftast inte konfigurationshanteras, då uppdatering normalt inte görs till exempel av mötesprotokoll. Å andra sidan kan det vara bekvämt att helt enkelt förvara all dokumentation i ett och samma system. Det vill säga konfigurationshanteringssystemet

kan användas som lagringsställe även om dess huvudsyfte inte egentligen används. Organisationsdokumentationen måste konfigurationshanteras. Det är dock inte enskilda projekt ansvariga för utan det ansvaret ligger hos organisationen. Konfigurationshantering av organisationsdokument är dock en förutsättning för att kunna kommunicera ut den senaste versionen av till exempel kvalitetssystemet på ett effektivt sätt.

Grundregeln för vad som skall ligga under konfigurationskontroll är att artefakter som genomgår olika versioner och där det kan vara väsentligt att inte bara ha tillgång till den senaste versionen skall definitivt läggas in. Det är viktigt att notera att konfigurationshantering innebär inte endast att den senaste versionen av artefakten är tillgänglig, utan att äldre versioner kan plockas fram vid behov.

- Det skall finnas en tydlig process för konfigurationshanteringen. Ur den skall framgå hur och när olika artefakter skall konfigurationshanteras och hur förändringar av artefakter skall godkännas.

- Det skall tydligt framgå vem som är ansvarig för konfigurationshanteringen. Detta gäller dels för processen som sådan och dels för den specifika tillämpningen av processen. I många fall tas beslut av en styrgrupp som har till uppgift att godkänna huruvida specifika ändringar, till exempel kravändringar, skall tas in i ett projekt. Andra ändringar, till exempel, av kod kan i många fall tas av den enskilde utvecklaren. Exakt hur besluten skall tas måste framgå av processen i föregående punkt.

- I planeringen måste även information om verktygsstöd med mera bestämmas.

Ändringshantering är viktigt. Den ingår som en viktig del i konfigurationshanteringsplanen. Det är ganska vanligt med kravändringar under ett utvecklingsprojekt. I dessa fall är det vanligt med den styrgrupp som nämndes ovan. Den har till uppgift att hantera ändringsförslag och besluta huruvida de skall ingå i ett specifikt projekt eller ej. Dessa beslut tas baserat på en blandning av tekniska skäl (till exempel effekten av ändringen), management skäl (till exempel kostnad för att hantera ändringen) och kundskäl (till exempel vem

inkom med ändringsförslaget). Styrgruppen har till uppgift att göra bästa möjliga avvägande mellan alla synpunkter som inkommer.

Hantering av versioner och releaser är en väsentlig del av konfigurationshanteringen. En ny version innebär ofta mindre uppdateringar i förhållande till en tidigare version. Om fel har hittats kan det leda till en ny version då felet har rättats. En release är ofta en större sammanhållen uppdatering och förbättring av ett programvarusystem. Versionshantering måste skötas dels på systemnivå och dels på enhets- eller komponentnivå.

Om en organisation har ett antal kunder, då är det möjligt att kunderna använder olika versioner. Detta måste finnas dokumenterat. Det är också viktigt att hålla koll på till exempel vilka fel som har rättats i olika versioner. Det är inte acceptabelt att samma fel uppträder igen i samband med att en ny version släpps. Det kan hända om utvecklingen har fortskridit samtidigt som en kund får ett fel rättat i en äldre version. I detta fall är det högst väsentligt att felet också rättats i den version som är utgångspunkten för vidareutvecklingen. När det gäller releaser måste beslut tas om vad som skall ingå i respektive release och när de skall släppas. Vissa organisationer släpper releaser regelbundet, medan andra släpper nya releaser när de har gjort större förändringar och tillägg.

Slutligen är konfigurationshanteringen viktig i samband med systembyggandet (länkningen av systemet). Detta innebär kontroll på vilka versioner av olika enheter och komponenter som skall ingå i en specifik version eller release. När ett nytt system skall genereras måste rätt versioner på respektive enheter och komponenter tas med, annars finns det en risk att systemet inte fungerar. Det kan till och med vara så att rätt version på till exempel en specifik kompilator måste ha använts för att systemet skall fungera.

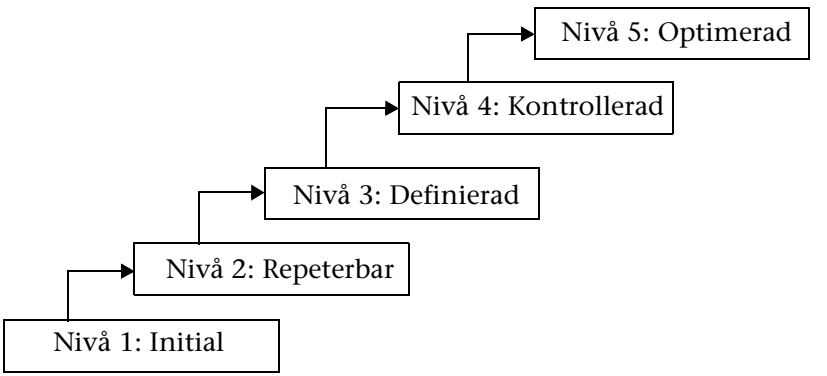
4.5 Allmän processförbättring

Inledningsvis i detta kapitel togs utvecklingsmodeller upp. Dessa modeller har sina egna speciella versioner inom olika organisationer. Samtidigt strävar organisationer efter att förbättra sina produkter och ett sätt att göra det är att förbättra utvecklingsprocessen.

Den allmänna hypotesen är att en bättre utvecklingsprocess skall resultera i en bättre produkt. Det finns två principiellt olika angreppssätt till förbättring av processen: referensmodell och förbättringsanalys. Presentationen nedan inriktar sig på förbättring av processen i allmänhet baserat på dessa två olika angreppssätt. Förbättring av olika kvalitetsattribut behandlas kortfattat i Kapitel 13.

Utgångspunkten för en referensmodell är att baserat på information om vad som anses vara de viktigaste byggstenarna i en bra process formuleras en modell som definierar lämpliga steg för förbättring. Grundtanken i referensmodellen är att det finns vissa saker som måste göras först för att sedan följas av andra. Den första referensmodellen som fick ett stort genomslag inom programvaruutveckling var "Capability Maturity Model" (CMM) som togs fram av Software Engineering Institute i USA. Den har senare följts av andra modeller, till exempel SPICE som har blivit en accepterad standard och även nyare versioner av CMM som tar ett lite breddare grepp kring systemutveckling. Här introduceras CMM som ett exempel på uppbyggnaden av en referensmodell.

CMM är uppbyggd kring fem nivåer, vilket illustreras i Figur 4.3.



Figur 4.3 CMM och dess fem nivåer.

De fem nivåerna är gjorda för att definiera olika grader av mognad i programvaruutvecklingen. Den första nivån kallas ibland hjältarnas. Anledningen till detta är att en organisation på denna nivå kan mycket väl leverera utmärkta produkter, men organisationen har inga fasta rutiner för projektledning och uppföljning. På den andra

nivån har organisationen formell hantering av projekt, kvalitetsuppföljning och konfigurationshantering, men det saknas en formell processmodell. Detta gör att organisationen är mycket beroende av att projektledare motiverar och leder sina team och projekt. Den tredje nivån innebär att det finna en formell processmodell och därmed en fast grund för processförbättring. Det finns också formella procedurer som innebär kontroll av att samtliga projekt följer den definierade processen. På fjärde nivån sker systematisk datainsamling av kvantitativa mått på produkten och processen. Detta gör att det finns en kvantitativ bas för processförbättring. Slutligen, den femte nivån innebär att organisationen arbetar kontinuerligt med processförbättring. Det finns en specifik budget för förbättring och den är välplanerad och integrerad del av en organisations aktiviteter.

CMM har funnits sedan runt 1990, men fortfarande är en majoritet av organisationer som utvecklar programvara på de lägre nivåerna enligt modellen. En möjlig förklaring till detta är att det finns ett antal nyckelområden som skall hanteras på respektive nivå för att kunna komma upp på en högre nivå. För att komma upp på en högre nivå krävs att organisationen uppfyller alla förväntningar på den lägre nivån. Detta betyder att om det finns problem med ett av dessa nyckelområden för övergången mellan nivå 1 och 2 kvarstår organisationen på nivå 1. Grundtanken med CMM var att en organisation skulle ha ett mått på sin mognad. Det är inte alla som har tyckt att det är ett bra angreppssätt. SPICE-modellen är ett svar på detta. SPICE har olika nivåer på olika aktiviteter. Det kan betyda att en organisation har ett högt betyg på en aktivitet (delprocess) och ett lägre på en annan aktivitet.

Som illustration på vad som kan vara en delprocess i detta sammanhang kan sägas att följande är de nyckelområden som måste hanteras på ett bra sätt för komma upp på nivå 2 i CMM, det vill säga för att komma från den initiala nivån till den repeterbara nivån:

- Konfigurationshantering
- Kvalitetssäkring
- Hantering av underleverantörer
- Projektuppföljning
- Projektplanering
- Kravhantering

Referensmodellerna är tänkta att fungera som stöd för processförbättring, det vill säga en referensmodell pekar på områden som måste förbättras via sina nivåer. Detta betyder att de inte bygger på en egentlig analys av problemen inom en specifik organisation, utan på den allmänna bilden av vad som är viktigast. Det andra angreppssättet (förbättringsanalys) utgår från den faktiska situationen och försöker identifiera svaga punkter i processen.

För att genomföra förbättringsanalys kan olika typer av modeller användas. En vanlig basmodell är den som föreslogs av Shewhart som på 1930-talet arbetade med kvalitet och speciellt statiskt kvalitetskontroll. Som en del av sitt arbete tog han fram förbättringscykeln: PDCA – Plan, Do, Check och Act. Stegen kan synas ganska självklara. I det första steget skall en förbättringsmöjlighet identifieras och införandet planeras. Därefter skall det genomföras eller provas i ett fall. Uppföljning och kontroll sker i det tredje steget. Slutligen skall förbättringen permanentas om utfallet var positivt. Därefter kan förbättringsarbetet fortskrida genom att påbörja steg ett igen. En anpassning av denna allmänna förbättringsmodell har gjorts av Basili och hans kolleger inom SEL (Software Engineering Laboratory). Deras modell innehåller sex steg och kallas: QIP – Quality Improvement Paradigm. De sex stegen är:

1 Karakterisera och förstå
  I detta steg görs en nulägesanalys. Målsättningen är att förstå utgångspunkten och identifiera förbättringsmöjligheter.
2 Sätt mål
  Utgående från nuläget sätts förbättringsmål.
3 Välj processer med tillhörande metoder, tekniker och verktyg
  När målet är satt är det dags att välja processer, metoder, tekniker och verktyg som kan bidra till att målet uppnås.
4 Exekvera processen
  Baserat på valen i föregående punkt används processen och tillhörande metoder, tekniker och verktyg. Erfarenheter dokumenteras.
5 Analysera resultaten
  Erfarenheterna från exekveringen analyseras. Det måste utvärderas om de uppsatta målen uppnåddes eller ej, även andra erfarenheter tas tillvara.

6 Paketera och lagra
Erfarenheterna paketeras på lämpligt sätt och lagras. QIP är tänkt att vara en del av en lärande organisation som diskuterades tidigare i detta kapitel. När detta steg är genomfört är det dags att börja på steg 1 igen.

Kontinuerlig processförbättring är väsentligt för att kunna leva upp till de krav som ställs på en organisation idag. Det finns hela tiden krav på ny funktionalitet som skall levereras snabbare med högre kvalitet och till lägre kostnad. Det leder till vikten av att kunna hantera kontakterna med kunder och marknad på ett bra sätt.

4.6 Kund och marknad

I många organisationer finns separat avdelningar som är ansvariga för kund- och marknadskontakter. Det är dock ofta att detta inte är tillräckligt, då utvecklingen måste ta hänsyn till olika kunder och marknaders önskemål. Lite förenklat delas ofta utveckling in i antingen kundutveckling eller marknadsdriven utveckling. I det första fallet syftas det på att det finns en kund som då är beställare. Vid marknadsdriven utveckling sker en mer generell utveckling för att produkten skall kunna säljas på en mer allmän marknad, som till exempel försäljningen av ett ordbehandlingsprogram. I praktiken finns det ett antal varianter mellan dessa båda. I många fall önskar företaget som tar fram en programvara att det skall kunna säljas till en marknad eller flera kunder. Å andra sidan vill varje kund ha sin speciella anpassning. Detta gör att hybridformer mellan kundutveckling och marknadsdriven utveckling blir vanliga.

Oavsett vilket är det viktigt att utvecklarna har en god bild av kundens eller marknadens önskemål. Det tydligaste exemplet är när det gäller utveckling av användargränssnitt, där det blir mycket tydligt att kunden eller marknaden kan ha önskemål. Det är dock inte det enda fallet då hänsyn måste tas. Det finns ofta även starka synpunkter på vilka funktioner som skall implementeras och hur de skall fungera. Därmed är användarorientering mycket väsentlig vid utveckling av programvaruprodukter. Kravet från kunder och marknader kommer bara att öka i framtiden, då dessa icke-experter på

programvarusystem får en större förståelse för programvarans möjligheter. Med anledning av detta är det väsentligt att varje utvecklare har användarna i åtanke vid utveckling av programvara.

Del 3: Utvecklingsfaser

Så snart ett programvaruprojekt uppnår en viss storlek krävs det att det genomgår ett antal utvecklingsfaser. Detta beskrivs i en övergripande process som tillhör linjeorganisationen och processen instansieras normalt i ett antal projekt. Linjeorganisationen och projekt finns beskrivit i Del 2. I denna del beskrivs de delar som normalt ingår i en utvecklingsprocess och som sedan genomförs inom ramen för ett utvecklingsprojekt. Del 3 beskriver följaktligen hur ett antal utvecklingsfaser genomlöps från kravhantering via design och implementation till test. Dessutom tas granskningar upp som kan användas för kvalitetssäkring. Vidare behandlas underhåll och vidareutveckling samt användningen av utvecklingsverktyg.

5 Kravhantering

5.1 Inledning

Den första aktiviteten i utvecklingen är kravhantering. Huvudsyftet är att bestämma ”vad” programvarusystemet skall göra. Under design skall sedan bestämmas ”hur” kravet skall implementeras. Detta är den stora skiljelinjen mellan kravhantering och design. Det visar sig dock ofta svårt att följa detta fullt ut. Det är dock viktigt att eftersträva att uppnå denna skillnad då det gör en tydlig gränslinje mellan kravhantering och design. Vidare är det ofta olika människor och personer med olika kompetens som är inblandade i de olika aktiviteterna. Därmed är det inte alltid att personerna som arbetar med kravhanteringen är bäst på att avgöra hur en viss funktion skall realiseras i ett programvarusystem.

Kravhantering genomförs i de flesta organisationer på två olika nivåer. Anledningen är att olika typer av programvarusystem ofta har en lång livslängd och släpps i många releaser. Det gör att dels måste organisationen hantera krav på produkten och dels måste ett projekt hantera kraven på den närmsta releasen.

För att hantera krav inom organisationen finns ofta en produktledning som ser till produktens långsikta utveckling samt i många fall avgör vad som skall ingå i nästa release. Produktledningen ansvarar i många fall för de externa kontakterna med kund och marknad.

Hanteringen av krav på projektnivå görs genom att det finns en kravspecifikation som gäller för projektet. I vissa organisationer (speciellt mindre organisationer) finns det ingen direkt produktledning utan frågor kring produkten hanteras av det projekt som utvecklar produkten för tillfället.

De två nivåerna på kravhantering som diskuteras här kan även kopplas till tidsperspektivet, det vill säga långsiktighet respektive

kortsiktighet. Flertalet programvaruprodukter utvecklas under en lång tid genom att nya funktioner läggs till. Ett tydligt exempel på det är mobiltelefoner som inledningsvis enbart hanterade vanliga telefonsamtal, men som senare har övergått till att vara små kommunikationsterminaler med möjlighet att bland annat sända e-mail och se på film. I det långsiktiga perspektivet är det således viktigt att arbeta med krav på programvaran som gör att produkten är konkurrenskraftig på marknaden. Det långsiktiga arbetet måste dock balanseras mot det kortsiktiga. För att en produkt skall vara framgångsrik måste produktledningen också kunna avgöra vilka krav på produkten som skall in i nästa release.

Ovanstående betyder att krav måste samlas in och det är också viktigt att välja ut vilka krav som skall in i nästa release. Det kräver en god kunskap om marknaden eller kunden för produkten. Det finns därmed ett stort behov av att förstå kraven på produkten och sedan kunna hantera dem. Det finns tre huvudperspektiv som måste förstås:

- Kravställare, det vill säga vem har intresse i produkten? Det finns ett flertal olika typer av kravställare. Dessa diskuteras i Avsnitt 5.2.
- Kravtyp, det vill säga vilka typer av krav måste hanteras? Det finns flera olika sätt att dela in krav och i Avsnitt 5.3 presenteras fyra olika perspektiv på indelning av krav.
- Kravprocessen, det vill säga hur samlas krav in och hanteras innan de är implementerade och levererade? Processen för detta beskrivs i Avsnitt 5.4.

Utöver dessa huvudperspektiv återfinns i Avsnitt 5.5 en beskrivning av hur en kravspecifikation i ett projekt kan göras.

5.2 Kravställare

Det finns fyra principiellt olika typer av kravställare. Det är marknad, kund, egna organisationen och regelverk. Det är inte säkert att varje programvaruprodukt berörs av alla dessa typer, utan det beror på produkten och målsättningen med densamma.

Vissa produkter utvecklas för en marknad. I dessa fall samlas ofta information in i form av önskemål, det vill säga det är inte direkta krav. Företag gör detta genom att göra marknadsundersökningar och analys av konkurrenter. Typiska exempel är spel och operativsystem. De riktar sig typiskt till en marknad.

Andra produkter utvecklas för en specifik kund. Det kan antingen göras genom att kunden har en egen bild av vad som behövs eller genom att en förändringsanalys genomförs. I det senare fallet analyseras kundens situation. Denna analys görs genom att både nuläget och önskat läge bestäms. Skillnaden mellan nuläge och önskat läge utgör grunden för att identifiera de problem som ett nytt programvarusystem behöver hantera. Typiska exempel kan vara administrativa system (eller en anpassning av ett existerande system) som skall utvecklas till en specifik organisation eller programvara till exempel för en specifik bilmodell. Det finns en rad situationer när kundspecifik programvara behövs. Den stora nackdelen är att utvecklingskostnaden inte delas mellan flera organisationer eller av en marknad.

Den egna organisationen kan generera krav med målet att en bättre produkt skall tas fram. Detta är mycket vanligt, speciellt i samband med utveckling av programvara för en marknad. I detta fall är det många gånger den innovativa kraften i den egna organisationen som bestämmer konkurrenskraften på marknaden.

Slutligen kan olika regelverk vara kravställare på en programvara. Regelverken kan vara lagar, till exempel påverkar skattesystemet de ekonomiska programvarusystemen, eller internationella standards, till exempel protokoll för hur kommunikation skall ske mellan en mobiltelefon och nätverket. I det senare fallet är det den internationella standarden som gör att det är möjligt att köpa mobiltelefoner från olika tillverkare utan byte av system (operatör).

I samband med utveckling av programvara är det viktigt att förstå vilka olika kravställare som finns. Det gäller både mellan ovanstående fyra olika typer och till exempel mellan olika marknader. Det är mycket möjligt att olika marknader, till exempel USA och Europa, har olika krav. En programvaruutvecklande organisation måste då ta ställning till hur kraven från olika marknader skall han-

teras och vad som är viktigast. Detsamma kan gälla om en produkt utvecklas till ett antal specifika kunder.

Sammantaget kan konstateras att det är viktigt att på ett tidigt stadium ha målgruppen för en produkt klart för sig, vilket i sin tur leder till att det är möjligt att prioritera mellan olika kravställare och deras respektive krav. Prioritering behöver göras både mellan olika kravställare och olika krav. Det senare återkommer vi till nedan.

5.3 Kravindelning

En stor svårighet med kravhantering är att det finns så många olika perspektiv på krav och olika sätt att dela in dem. Det är möjligt att dela in kraven beroende på:

- Kravställare
- Olika nivåer som kraven representeras på.
- Olika typer av krav, till exempel om kraven är funktionella eller icke-funktionella.
- Huruvida kraven är riktade mot programvaran som sådan, mot utvecklingsmiljön eller har att göra med programvarans omgivning.

När det gäller kravställare kan kraven delas in i enlighet med indelningen i föregående avsnitt. I denna indelning kan även hänsyn tas till olika marknader, kunder med mera. Denna indelning kan vara relevant då prioritering skall göras mellan marknader eller kunder.

Den andra punkten har att göra med detaljeringsnivå och representationsform för ett krav. Krav som kommer in från en marknadsanalys eller en kund kan ofta vara beskrivna på en ganska allmän nivå och ofta i naturligt språk, till exempel svenska eller engelska. Detta gör att ett krav behöver förfinas och specificeras noggrannare innan det faktiskt kan implementeras. Som exempel kan nämnas att en kund mycket väl kan säga att en produkt skall vara användarvänlig, men vad betyder det egentligen? Olika representationsformer för krav och olika nivåer diskuteras vidare i nästa avsnitt.

En vanlig indelning är att dela in krav i funktionella och icke-funktionella krav. Funktionella krav syftar på funktioner eller tjänster som programvaran skall tillhandahålla, till exempel programmet skall innehålla möjlighet för utskrift. De funktionella kraven riktar sig oftast mot enskilda funktioner. Icke-funktionella krav kan vara riktade mot enskilda funktioner, men är ofta riktade mot hela programvarusystemet. Det kan röra sig om att data skall vara i ett specifikt format eller att svarstiden vid användning av systemet skall understiga en viss tid i 95 % av fallen. Ett tydligt exempel är tiden en användare är beredd att vänta på ton när telefonluren lyfts. Andra viktiga icke-funktionella krav kan röra tillförlitlighet eller säkerhet. Det senare inkluderar både säker användning och att systemet är säkert för dess användare, till exempel gentemot virus och olika typer av intrång.

Slutligen kan krav delas in beroende på om det rör sig om explicita krav på systemet eller om kravet är kopplat till omgivningen, antingen utvecklingsmiljön eller den miljö programvaran skall exekvera i. Krav på systemet kan vara att systemet skall utföra en viss funktion eller ha en viss säkerhet. Dessa krav kallar vi här systemkrav. De andra kraven är organisationskrav. De kan röra sig om krav på användning av en specifik designmetod, programmeringsspråk eller operativsystem, vilket är riktat mot utvecklingsmiljön. Alternativt kan det vara krav som kopplar till miljön i vilken systemet skall användas, till exempel att systemet skall vara kompatibelt med andra system eller att information skall presenteras på ett visst sätt eller i ett visst format.

De olika perspektiven gör att krav kan beskrivas i många dimensioner. Ett enkelt exempel illustreras i Tabell 5.1, där krav har delats in i fyra olika typer baserat på huruvida kraven är funktionella eller icke-funktionella och om det rör sig om systemkrav eller organisationskrav. Denna typ av indelning kan vara väsentlig att göra för att få en god förståelse av kraven och hur de relaterar till varandra. Vidare kan det vara viktigt när det är dags att prioritera kraven.

Tabell 5.1 Exempel på indelning av krav i olika typer.

	Funktionella krav	Icke-funktionella krav
Systemkrav		
Organisationskrav		

5.4 Kravprocess

5.4.1 Fånga

Det första steget i processen kallas ibland kravinsamling. Problemet med detta begrepp är att det antyder att kraven finns där och att det bara att samla in dem. I flertalet fall är det inte så enkelt och därför används här begreppet ”fånga krav” istället, då det åtminstone antyder någonting som är svårare än att samla in dem. Detta steg i kravprocessen är mycket svårt och det är nästan omöjligt att veta om arbetet lyckades förrän programvarusystemet är klart och använt under en tid. Samtidigt är det en oerhört viktig aktivitet då resten av utvecklingen bygger på de krav som finns.

Det finns tre huvudtyper av sätt att fånga krav, sedan finns det ett antal tekniker för att genomföra dem. De tre huvudtyperna är:

Marknadsanalys
Denna typ baserar sig på att en analys sker av marknaden för en specifik produkt eller tjänst. En marknadsanalys i detta sammanhang innebär att en analys görs av potentiella kunder och deras behov.

Kundkontakter
Denna typ kan användas antingen när målsättningen är att utveckla ett system för en marknad eller för en eller flera kunder. Även om målsättningen är en hel marknad kan en befintlig kundbas användas för att identifiera gemensamma intressen för att på så sätt skapa ett system för en marknad. Om målet är en

eller flera specifika kunder gäller det att få reda på vad kunden eller kunderna anser sig behöva. Om målsättningen är att utveckla ett system för en specifik kund används med fördel nästa huvudtyp för att fånga kraven.

Förändringsanalys
I denna typ göra en ingående analys dels av nuläget och dels av syftet med det nya systemet, det vill säga önskat läge efter det nya systemet är tagit i drift. En förändringsanalys genomförs genom noggrann dokumentation av nuläge, önskat läge och därmed också av de problem som det nya systemet förväntas lösa.

För att genomföra ovanstående huvudtyper av analyser, för att kunna utveckla ett bra programvarusystem, finns det ett antal tekniker. Dessa tekniker kan vara riktade mot individer, grupper eller andra system. När det gäller individer innebär det kartläggning av olika typer av intressenter genom kontakt med enskilda personer. Det kan också vara möjligt att genomföra en analys av en grupp av människor till exempel via olika typer av gruppaktiviteter, till exempel användarseminarier. En tredje möjlighet är att studera konkurrerande system, vilket även kan inkludera diskussioner med användare av dagens system.

När det gäller tekniker finns det ett antal olika möjligheter som kan användas på lämpligt sätt beroende på om det rör sig om individer, grupper eller andra system:

Intervjuer
Personer och grupper av människor kan intervjuas. Det kan ske antingen genom strukturerade intervjuer, det vill säga med förberedda frågor, eller ostrukturerat, det vill säga ganska förutsättningslöst. I det senare fallet är det möjligt att använda brainstorming (”spånskiva”) eller olika typer av workshops. En workshop är en förberedd typ av gruppmöten där gruppen eller delar av gruppen får specifika uppgifter eller förutsättningar.

Frågeformulär
Olika typer av frågeformulär kan skickas ut till intressenter. Formulären kan skickas till både potentiella kunder och andra personer som har ett intresse av systemet och dess funktionalitet.

Observation
Det finns även i vissa fall möjligheter att använda observationstekniker. Det kan innebära observation av användning av den nuvarande lösningen (gammalt system eller manuell procedur) eller av arbetet runt omkring det planerade nya systemet. Tanken är att genom observation få en förståelse för behoven, problemen och sammanhanget för det nya programvarusystemet.

Provutveckling
Ett fjärde alternativ är att genom utveckling av en prototyp och utvärdering av densamma säkerställa att det kommande systemet motsvarar förväntningarna. Det är i samband med detta också möjligt att genomföra provexperiment för att avgöra om det är troligt att målsättningarna med det nya systemet uppfylls.

Ovanstående tekniker måste kompletteras med riskanalys, ekonomisk analys och en målanalys. Det senare innebär en analys av om huruvida det är troligt att det nya systemet kommer att uppfylla de uppsatta målen. Riskanalysen skall göras för att identifiera risker och avgöra hur sannolika de är. Den ekonomiska analysen skall innefatta både en kostnadsanalys och en uppskattning av de ekonomiska fördelarna med framtagning av ett nytt system.

5.4.2 Representera

Krav kan representeras på ett stort antal olika sätt. Det finns tre principiellt olika sätt att representera krav:

Naturligt språk
Naturligt språk syftar på användning av vanlig text. Detta är den vanligaste representationsformen. Det positiva är att flertalet människor förstår beskrivningen (under förutsättning att språkkunskapen finns). Nackdelen är att det naturliga språket lämnar alltid utrymme för tolkningar och därmed för missförstånd.

Grafiska beskrivningar
I detta fall används ofta olika typer av grafiska beskrivningsspråk. Det finns ett stort antal av dessa språk och valet av representation varierar beroende på organisation och typen av system.

Matematisk notation
Denna typ av notation används oftast för kritiska funktioner, till exempel för säkerhetskritiska delar i ett system. Fördelen är att det är en exakt definition och nackdelen är att många upplever den som svår att förstå.

De olika sätten att representera krav kopplar även till beskrivningsnivån av krav. Inledningsvis används ofta beskrivningar i textform. En anledning är att det är enkel och snabbt att skriva. En annan anledning kan vara att de personer som skriver ned kraven inte behärskar någon annan lämplig beskrivningsteknik.

Behovet av mer formaliserade formuleringar av kraven ökar efterhand som utvecklingen fortskrider. Det räcker kanske med en beskrivning i textform innan det är helt bestämt om kravet skall ingå i den kommande releasen. Om det beslutas att ett krav skall implementeras i det kommande projektet/releasen då är det kanske viktigt att förbättra beskrivningen och försöka formulera det mer entydigt, det vill säga minska riskerna för misstolkning. En svårighet vid omformulering är att det finns en risk att den som genererade kravet från början inte kan förstå kravet i den nya representationsformen och därmed finns en risk att det trots allt inte riktigt blir som den initiala kravställaren hade tänkt sig det. Samma resonemang gäller en eventuell övergång till den mer matematiska notationen.

Kravnedbrytning är viktig, inte bara för att konkretisera det nuvarande kravet, utan det kan visa sig att det initiala kravet inte är ett krav utan ett antal krav vid närmre studie. Ett exempel kan vara ett krav på att uppfylla en viss standard, men en standard är ofta ett tjockt dokument som i sin tur innebär ett antal krav. För mobiltelefoni kan ett krav vara att systemet skall uppfylla GSM-standarden, men det sin tur innebär ett stort antal saker inklusive bland annat krav på dataformat.

Denna nedbrytning kan också innebära att ett krav (som är beskrivit i text från början) resulterar i ett antal krav som kanske representeras på olika sätt. Vidare kan olika typer av krav kräva olika typer av representation. Det kan till exempel vara så att funktionella krav representeras annorlunda än icke-funktionella krav. Vidare kan vissa typer av krav kräva sin egen representationsform. Ett exempel

på det är krav som kopplar till dataformat, där det kan vara bra att ta med använda specifika notationer som är bra på att presentera data. Ett annat exempel är gränssnitt. Det kanske är lämpligt att genom en bild av en skärm eller ett fotografi representera hur ett specifikt användargränssnitt skall se ut.

5.4.3 Hantera

Kravhantering är långt ifrån bara en teknisk fråga om vad ett programvarusystem skall göra. Det finns många aspekter kopplade till kravhantering som är relaterade till management och beslut. Det viktigaste är att avgöra vilka krav som skall inkluderas i programvaran. Det kan dels härröra till vilka krav som överhuvudtaget är relevanta för produkten och dels vilka krav som skall tas med i nästa projekt eller release. Båda dessa saker är i första hand en fråga för produktledningen eller företagsledningen. Det är viktigt att inte bara bestämma vad ett system skall göra utan även att avgränsa vad det inte skall göra.

Ovanstående gör att prioritering är en mycket viktig aspekt i samband med krav. Det inkluderar både prioritering av vad som skall ingå i produkten och vad som skall tas med i nästa release. För att kunna genomföra en bra prioritering finns det ett antal saker som måste finnas på plats. Det måste finnas stöd för hantering av beroenden mellan krav. Beroenden kan vara positiva eller negativa. Positiva beroenden innebär att om kravet implementeras är det även bra att implementera ett annat krav. Dessa båda krav sägs då ha ett positivt beroende. Identifiering av positiva beroenden kan underlätta vid beslut om vad som skall tas med i nästa release. Ett negativt beroende mellan två krav är när implementationen av ett krav gör det svårare att implementera ett annat krav. Detta i sin tur kan leda till att det är bra att enbart välja att ta med ett av dessa krav.

Förståelse av beroende mellan krav lägger grunden till spårbarhet. Det finns två huvudtyper av spårbarhet: horisontell och vertikal. Den horisontella spårbarheten har att göra med beroende mellan krav. Vertikal spårbarhet i sin tur är relaterad till kopplingen mellan krav och underliggande design samt implementation. Den vertikala

spårbarheten är en förutsättning för att kunna genomföra påverkansanalys. Denna typ av analys görs för att kunna avgöra hur ett specifikt krav påverkar ett existerande system. Med tanke på att många programvarusystem utvecklas över tiden och släpps i ett antal releaser är påverkansanalys en mycket viktig aktivitet för att avgöra vad som skall tas med i produkten som helhet och speciellt i den kommande releasen.

Ett krav som, till exempel, påverkar stora delar av systemet innebär en stor risk. Ett krav som till sin helhet kan implementeras i en enskild komponent innebär ett mycket mindre risktagande. I samband med påverkansanalys finns ett antal vyer när bedömning görs av påverkan av ett specifikt krav. Produktledningen är i första hand intresserade av att avgöra huruvida kravet innebär en stor risk för produkten och den kommande releasen. Projektledningen är intresserad av hur kravet påverkar projektplaneringen och hur arbetet skall genomföras, det vill säga av vem och hur mycket tid behövs. Utvecklarna är intresserade av att göra en mer detaljerad bedömning av den exakta påverkan i termer av design och kod.

Förståelsen för beroenden mellan krav, spårbarheten gentemot design och kod samt påverkansanalysen lägger grunden för att kunna prioritera krav. Det går naturligtvis att prioritera krav utan denna bas, men riskerna och osäkerheten är avsevärt mycket större. Det är därför lämpligt att avråda från prioritering utan tillräcklig kunskap om konsekvenserna. En felaktig prioritering kan få ödesdigra konsekvenser för programvaran, det vill säga det finns en risk att fel produkt utvecklas eller att resurser slösas bort på utveckling av fel funktionalitet i förhållande till önskemål från marknader och kunder. Vikten av en god kravhantering kan inte överdrivas med tanke på dess viktiga roll inför den fortsatta utvecklingen.

Prioritering av krav är svårt. Det är inte ovanligt att möta inställningen att alla krav är viktiga, annars hade de aldrig formulerats. Detta gör det mycket väsentligt att tänka igenom hur krav prioriteras, av vem och för vem. Det senare är kopplat till målgruppen för programvarusystemet och måste bestämmas utgående från detta. Prioriteringen bör genomföras gemensamt av ett antal personer med olika kunskaper om produkten och systemet, det vill säga både personer som ser på de strategiska aspekter för produkten och dels

av dem som har en god överblick och kunskap om det faktiska systemet.

När det gäller hur prioritering görs finns det ett antal alternativa metoder tillgängliga. Några exempel tas upp här. För det första finns det två principiellt olika angreppssätt, nämligen rankning och gruppering. Vid rankning är målsättningen att kraven läggs i en ordning utgående från deras betydelse. Gruppering innebär att kraven delas in i grupper baserat på deras prioritet. Ett exempel på gruppering kan vara: 1) måste tas med, 2) viktiga att ta med och 3) skulle kunna tas bort. Svårigheten med denna typ av gruppering är att det finns en tendens att många krav hamnar i den första gruppen och även den andra gruppen har ett ganska stort antal krav, medan den tredje gruppen är nästan tom. Det kan innebära att prioriteringen inte ger önskad effekt.

För att undvika detta är det möjligt att antingen sätta upp regler för hur många krav som minst måste återfinnas i respektive grupp eller att använda rankning. Rankning innebär dock mer arbete, då alla krav måste placeras på sin egen plats. Fördelen är dock att det finns en tydlig lista sorterad i prioritetsordning. När det gäller rankning finns det två olika typer av rankning. Den första innebär en ren rangordning, det vill säga kraven får enbart en plats i listan. Den andra typen innebär att kraven också åsätts en relativ betydelse, det vill säga det är möjligt att avgöra hur mycket viktigare ett krav är än ett annat krav. Den enklaste modellen för att åstadkomma det senare är den så kallade 100-poängsmodellen. Modellen innebär att den som prioriterar har tillgång till 100 poäng (eller vilket annat antal poäng som väljs) som kan fördelas på kraven. Det innebär att ett krav som får 30 poäng är tre gånger så viktigt som ett krav som erhåller 10 poäng.

Hittills har krav diskuterats utgående från att produktledningen avgör vad som är viktigt för produkten och att det i sin tur avgör vad som skall tas med i nästa release. Detta är dock en något idealiserad bild. Kravhantering är svårt och projekt tar tid att genomföra. Detta innebär att förändring av krav är nästintill oundvikligt. På liknande sätt som nya krav hanteras måste förändringar av krav hanteras. Det är vanligt att projekt har en förändringsgrupp som ansvarar för att göra bedömningar av föreslagna förändringar.

Ändringarna kan vara ett resultat av att kraven initialt har missförstått eller att förutsättningarna ändras under projektets löptid. Oavsett vilket måste beslut tas i anslutning till förändringsförslagen. I samband med detta är än en gång beroenden, spårbarhet och påverkansanalys mycket viktiga. Utgående från påverkansanalys och dess konsekvenser för projektet i form av eventuella förseningar och förändrade kostnader måste beslut om åtgärd tas. Detta är långt ifrån enkelt, då projekt många gånger har lagt ned utvecklingskostnader och nu kan riskera att behöva göra om arbetet. Samtidigt är det inte möjligt att alltid låta bli att ändra.

Ett starkt önskemål är naturligtvis att minimera ändringar efter det att ett projekt har startat. Detta kan endast göras genom en god kravhantering från början, beredskap för förändring samt god projektplanering. Det senare kan innebära att bedömningar görs av hur troligt det är att ett krav ändras. Om risken bedöms som hög för ändring är det kanske möjligt att vänta med utvecklingen av funktionaliteten kopplat till just detta krav. Detta är nog inte möjligt att genomföra fullt ut då det i många fall också krävs att viss funktionalitet måste utvecklas först för att andra funktioner bygger på densamma. Det är dock bra att eftersträva att utveckla de krav som bedöms som mest stabila först.

5.4.4 Validera

Kravvalidering innebär en kontroll att kraven är korrekta i förhållande i kravställarens intention. Avsikten är att försöka säkerställa att utgångspunkten för utveckling i form av krav är korrekt och om utvecklingen sker korrekt kommer rätt produkt att levereras. Detta kan göras på två sätt, antingen genom granskning eller genom att en prototyp utvecklas. En prototyp kan oftast inte tas fram för hela systemet då det är för kostsamt, men det kan vara ett bra alternativ för en speciellt viktig del av systemet till exempel användargränssnittet.

Det andra alternativet är noggrann läsning eller granskning av kraven. Granskning diskuteras i Kapitel 8. För att underlätta denna granskning kan olika saker kontrolleras. För krav finns ett antal saker som bör studeras:

- Är kraven korrekta? Efter bästa förmåga bör säkerställas att formuleringarna motsvarar vad som önskas av systemet.
- Är kraven konsistenta? Det är viktigt att kontrollera att kraven sinsemellan hänger samman, det vill säga att det inte finns något motsägelsefullt.
- Är kraven realistiska? Det är lätt att formulera orealistiska förväntningar på ett system och denna kontroll skall säkerställa att kraven som ställs är rimliga.
- Beskriver varje krav något som kravställaren behöver? Detta är en fråga om att identifiera om det finns onödiga krav, det vill säga att kraven inte inkluderar saker som kravställaren egentligen inte behöver.
- Är kraven testbara? Detta är en fråga om det efteråt utveckling går att avgöra om kraven är uppfyllda. Det är ingen mening med krav som inte går att utvärdera efter utveckling.
- Är kraven spårbara? Detta inkluderar spårning både till kravställare och till ett huvudkrav om det har brutits ned.

Huvudsyftet med valideringen är att göra en sista kontroll att utgångspunkten för det fortsatta utvecklingen är så bra som det någonsin är möjligt.

Hittills har presentationen handlat om krav i allmänhet. En viktig aspekt är dock att kraven är dokumenterade på ett bra sätt i början av ett projekt. Detta innebär att de krav som valts ut för det kommande projektet dokumenteras på ett bra sätt i en kravspecifikation. I nästa avsnitt presenteras ett exempel på vad en bra kravspecifikation bör innehålla.

5.5 Kravdokument

Det finns standards och olika rekommendationer för vad som bör ingå i en kravspecifikation. Nedanstående exempel är baserat på en standard av IEEE (Institute of Electrical and Electronics Engineers).

1 Introduktion

Introduktionen innehåller fem delrubriker. Den första delen skall

informera om syftet med dokumentet. Det bör till exempel tydligt framgå vilket system kravspecifikationen härrör till. För det andra skall omfattningen av produkten beskrivas. Det innebär att det skall framgå vilka avgränsningar som är gjorda. Den tredje delen skall ta upp olika definitioner och förkortningar som används i resten av dokumentet. För det fjärde skall det finnas en lista på referenser. Detta skall finnas för att kravspecifikationen skall sättas in i ett sammanhang. Om kravspecifikationen till exempel hänger samman med en standard (som GSM-standarden) måste det framgå vilken version av standarden som denna kravspecifikation utgår från. Slutligen bör inledningen innehålla en översikt av resten av kravspecifikationen.

2 Allmän beskrivning
Även den allmänna beskrivningen innehåller fem delar. För det första skall produktperspektivet beskrivas. Detta innebär att produktens status och målsättning bör göras tydlig. För det andra skall produktens funktioner beskrivas i allmänna termer. I den tredje delen skall användarna beskrivas. Det kan inkludera en beskrivning av vem den tänkta användaren är. Kräver systemet en viss typ av utbildning eller skall vem som helst kunna använda systemet? Fjärde delen tar upp generella begränsningar, vilket kan röra sig om förutsättningar som dataformat med mera. Slutligen skall den allmänna delen innehålla information om antaganden och beroenden. Det senare kan röra sig om beroenden av andra kringliggande system.

3 Specifika krav
Detta avsnitt är det längsta i dokumentet och innehåller de specifika kraven på programvaran. Avsnittet skall innehålla alla typer av krav strukturerat på ett lämpligt sätt, där det lämpliga sättet är beroende på typen av program som utvecklas. Det är viktigt att både funktionella och icke-funktionella krav tas upp. Ofta tas saker för självklara och skrivs inte in i kravspecifikationen. Det är dock ett misstag då det som kan vara självklart för en person inte behöver vara självklart för en annan person. I princip skall alla typer av krav som togs upp tidigare i detta kapitel tas in i detta avsnitt.

4 Appendix
Appendix kan inkludera beskrivningar av hårdvara och databaser som är av relevans för det aktuella systemet.

5 Index
Ett index kan tas med för att underlätta att hitta i dokumentet.

6 Design

6.1 Vad är design?

Kravspecifikationen beskriver vad systemet skall göra och utgående från denna skall en design göras. Specifikationen kan även ses som en lista på problem som skall lösas av programvarusystemet. Designen är lösningen på dessa problem. Det betyder att designen är den kreativa processen av att överföra problemen (kraven) formulerade i kravspecifikationen till en lösning. Ett enkelt exempel för att förstå skillnaden mellan krav och design är att titta på byggandet av ett hus. En familj har en rad krav när ett hus skall byggas, till exempel om vi antar att familjen har krav på antalet rum och den totala storleken på huset. Familjen har naturligtvis ett stort antal andra krav, men låt oss för enkelhets skull fokusera på detta enkla exempel. Det är sedan arkitektens sak att ta fram en ritning på hur rummen skall placeras och deras respektive storlek givit begränsningen av en total storlek. Detta kan betraktas som en högnivådesign. När familjen och arkitekten är överens om planlösningen tar andra vid utgående från ritningen. De bygger rummen och det är nu även dags att bestämma typen av golv och tapeter samt färger. Detta kan ses som en lågnivådesign, det vill säga när detaljerna klargörs. Slutligen kommer golvläggare och målare. Det sista steget är att jämföra med programmeringen. I detta kapitel är fokus på de båda designnivåerna och programmeringsdelen presenteras i nästa kapitel.

I exemplet ovan fanns två designnivåer. Det är ganska vanligt att hantera hög- och lågnivådesign. Det är dock möjligt att ha fler nivåer beroende på behov och typen av system. För att principerna för design skall framgå räcker dock två nivåer varför vi nedan håller oss till det. Det är också värt att notera att problem kan upptäckas under design som gör att det finns behov att gå tillbaka till kravspecifikationen och eventuellt göra ändringar. Det kan med andra ord

finnas behov av att göra iterationer efterhand som kunskapen om systemet ökar med utvecklingen av detsamma.

Det skall noteras att det finns mycket litteratur på detta område, speciellt finns det många böcker som avhandlar en specifik designmetod. Det finns böcker som beskriver design som en allmän företeelse, men de är betydligt färre. I denna bok är målsättningen med designkapitlet att ge en allmän känsla av vad design är. Det går inte att i denna typ av bok lära ut hur en design skall göras, utan vi får nöja oss med att diskutera principerna och ge en allmän känsla för vad design innebär.

6.2 Designprinciper

Att göra en design innebär att lösningen av problemet som beskrivs av kravspecifikationen skall formuleras i form av ett antal komponenter och deras inbördes kommunikation. Det finns många olika sätt att skapa en design och för varje sätt finns det sedan olika metoder eller notationer. I litteraturen identifieras fem principiellt olika sätt att göra design på. De fem är:

1 Modulär nedbrytning
   Denna typ av design utgår från att funktionerna i systemet fördelas på olika komponenter. Det betyder inte att varje funktion utförs av en komponent, utan funktionalitet måste fördelas så att det blir ett bra system som helhet. Mer specifikt innebär det att återkommande saker som används av flera funktioner måste placeras på ett ställe och inte upprepas för varje funktion. När detta görs formuleras komponenter och deras inbördes kommunikation.

2 Dataorienterad nedbrytning
   Denna designtyp är speciellt lämplig för ett dataorienterat system till exempel design av en databas. Utgångspunkten är externa datastrukturer som sedan bryts ned och representeras på lämpligt sätt internt i systemet.

3 Händelseorienterad nedbrytning
   En händelseorienterad design utgår från externa händelser som systemet måste kunna hantera. Designen utgörs sedan av föränd-

ring av systemets tillstånd baserat på dessa externa händelser. Ett exempel kan vara ett telefonisystem som ändrar tillstånd efterhand som en abonnent utför sina aktiviteter, till exempel ”lyft på luren”, ”slagit nummer” och så vidare.

4 Utifrån och in
Denna typ av design fokuserar på användaren. Tekniken utgår från vad användaren gör och vad det skall resultera i. Designen fokuserar på hantering av indata från användare och vad systemet skall göra med densamma.

5 Objektorienterad nedbrytning
Denna designtyp utgår från klasser av objekt och deras relationer. Utgångspunkten är oftast fysiska saker i anslutning till systemet. Det kan till exempel vara truckar och olika typer av varor i ett automatiskt lagersystem. I designen beskrivs sedan attribut hos dessa objekt och vilka saker som kan göras med respektive objekt.

Ovanstående fem principiella tekniker har naturligtvis saker gemensamt och det är möjligt att se likheter och olikheter. De betraktas dock som olika då de har olika utgångspunkter. Den teknik som vunnit mycket mark under det senaste decenniet är objektorientering som blivit alltmer populär. Dess popularitet kommer från att flera konkurrerande metoder har samlats i en metod, nämligen UML (Unified Modeling Language). Denna designmetod återkommer vi till i Avsnitt 6.6. Vidare har objektorienterade programmeringsspråk också vunnit popularitet under det senaste decenniet med C++ och Java i spetsen.

De två följande avsnitten behandlar de två designnivåerna diskuterade ovan och sedan diskuteras designkvalitet kortfattat i Avsnitt 6.5 innan några specifika designmetoder berörs i Avsnitt 6.6.

6.3 Arkitekturdesign

Det som i äldre litteratur ibland refereras till som högnivådesign har under det senaste decenniet växt fram som ett eget delområde som numera benämns arkitektur. Den stora skillnaden är att det som

tidigare informellt gjordes under högnivådesign numera har formaliserats i mycket större utsträckning. Detta innebär bland annat att vissa återkommande mönster för design av programvarusystem har identifierats.

Designen av ett programvarusystems arkitektur refererar till programvarans komponenter och det externt synbara av desamma samt de inbördes relationerna mellan komponenterna, det vill säga kommunikationen mellan dem. Det finns ett ental definitioner av programvaruarkitektur. Det gemensamma mellan dem är dock i stor utsträckning:

Komponenter/enheter
Varje programvarusystem (bortsett från triviala små program) behöver delas upp i delar, där varje del har ett specifikt ansvar som bidrar till systemets övergripande beteende. Det är viktigt att göra denna uppdelning av funktionalitet på komponenter på ett bra sätt. Detta inkluderar både att tänka på en bra lösning nu och att lösningen skall vara stabil inför framtida ändringar och uppdateringar av funktionalitet.

Komponenternas eller enheternas gränssnitt
För att kunna förstå och arbeta med designen måste gränssnitten på de ingående komponenterna dokumenteras, det vill säga det är viktigt att veta vad komponenterna/enheterna bidrar med. Det senare behövs för att i många fall används samma delfunktionalitet av många funktioner. Ett exempel (utan att veta hur det är löst) är att inom Officepaketet från Microsoft ingår Word (ordbehandling), Excel (beräkning) och PowerPoint (presentation) och inom alla dessa tre programmen behövs ett antal gemensamma funktioner som att kunna spara dokument, skriva ut dokument och så vidare. Denna typ av gemensamma funktioner skulle kunna finnas i komponenter som används av alla tre program som ingår i Officepaketet.

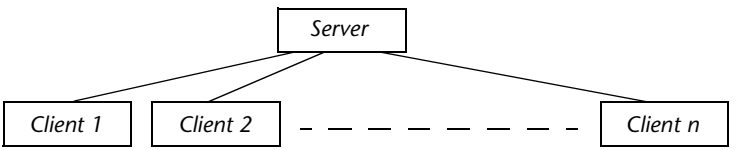
Kommunikation mellan komponenter/enheter
En målsättning med indelningen i komponenter/enheter är att kommunikationen mellan dem skall bli naturlig. Vidare är det viktigt att komponenterna formuleras så att kommunikationen blir ren, det vill säga att alla komponenter inte kommunicerar

med alla komponenter, utan att en bild av arkitekturen blir så enkel som möjligt.

Strukturen på programvaran och kommunikationen mellan ingående komponenter kan diskuteras i termer av designkvalitet. Detta görs i Avsnitt 6.5.

När det gäller beskrivning av programvaruarkitekturer har observerats att det finns vissa strukturer som återkommer i designen. Dessa har fått benämningen designmönster. Ett stort antal sådana mönster har definierats. För att exemplifiera vad designmönster innebär presenteras tre av de tydligaste och också vanligaste. För att undvika missförstånd har de engelska benämningarna bibehållits:

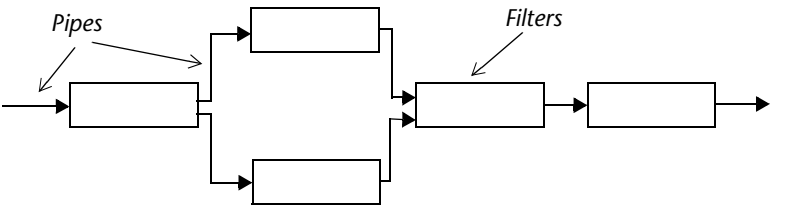
Client-server system
Denna typ av lösning är vanlig för distribuerade system, där en central nod handhar ett antal andra noder. Ett exempel på denna typ av system är mobiltelefoni när en basstation hanterar ett antal mobiler, det vill säga mobiler som finns inom täckningsområdet för basstationen. Det betyder att programvaran för basstation och mobiler måste konstrueras för att kunna hantera denna situation. En illustration av denna typ av system finns i Figur 6.1.



Figur 6.1 Illustration av client-server system.

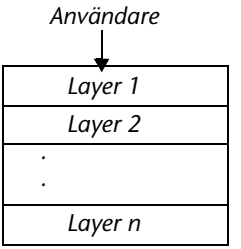
Pipes and filter system
Ett system av denna typ är uppbyggd av ett antal filter som till exempel data skall genomgå. Det betyder att varje filter tar emot ett antal indata genomför någon form av transformation av indata och sänder sedan utdata. De olika filterna skall inte behöva ha kunskap om de andra filterna, bortsett från att de skickar informationen vidare till dem. Anslutningen mellan filter kallas pipes. Ett exempel på denna typ av system är kompilatorer som tar programkod som indata och genom olika översättningar och analyser genererar binär kod som kan tolkas av hårdvaran i

en dator. En illustration av ett system av denna typ återfinns i Figur 6.2.



Figur 6.2 Illustration av pipes and filter system.

Layered system (skiktat system)
Denna typ av system bygger på en indelning i olika lager. Indelningen görs oftast utgående från användaren, det vill säga användaren har enbart tillgång till det yttersta eller översta lagret. Varje lager kommunicerar i det ideala fallet enbart med närliggande lager. Det finns skiktade system som gör avsteg från detta, men det bör nog undvikas om möjligt då det gör arkitekturen mer svåröverskådlig. En illustration av ett skiktat system återfinns i Figur 6.3.



Figur 6.3 Illustration av layered system.

De olika designerna, det vill säga olika mönster och olika lösningar, har olika egenskaper. Det betyder att de är bra på olika saker, vilket gör att valet av lösning beror på kraven på programvaran i form av till exempel olika icke-funktionella krav. Även de funktionella kraven kan avgöra vilken lösning som är lämplig.

När det gäller arkitekturen är det viktigt att inte enbart tänka på en bra lösning när den första versionen av programvaran utvecklas, utan att även ha den framtida utvecklingen i åtanke. En viktig

aspekt med arkitekturen är att kunna utvärdera densamma och även eventuellt förändra arkitekturen på systemet efterhand som det vidareutvecklas. Målsättningen måste vara att ha den bästa möjliga arkitekturen vid varje given tidpunkt under ett systems utveckling. Denna målsättning måste dock vägas mot kostnaden och vinsten med att förändra arkitekturen.

6.4 Lågnivådesign

När programvaruarkitekturen är på plats är det dags att göra en mer detaljerad design. Detta innefattar både den inre strukturen av komponenterna/enheterna och kommunikationen mellan dem. Designarbetet fokuseras först på algoritmer och datastrukturer.

Det kan finnas många olika sätt att lösa ett problem, det vill säga det finns olika alternativa algoritmer. Ett exempel är sortering av data. Ett annat illustrativt exempel är beräkning av $\binom{n}{k}$, vilket också användes för att illustrera representation i form av bitar i Kapitel 2. Utläst innebär denna beteckning att *n!/(k!*(n–k)!)* skall beräknas. En möjlighet är att helt enkelt beräkna *n!*, *k!* och *(n–k)!* och sedan genomföra den andra beräkningen. Problemet med denna lösning är att så snart *n* blir stort kommer datorerna att få problem med antalet siffror, det vill säga datorn kan inte representera tillräckligt stora tal. En alternativ lösning är:

1 Börja med *n!* och *k!*. Ta bort de *k!* sista talen i *n!* som just är *k!*. Detta ger *(n–k)!* tal kvar i täljaren.

2 Dividera med ett tal i taget, det vill säga beräkna först *n/n–k* för att sedan fortsätta med *n–1/n–k–1*. Efterhand som det görs multipliceras resultaten samma och vi fortsätter med detta till vi har nått till talet 1 i nämnaren.

Ett konkret exempel underlättar säkert förståelsen. Antag att vi vill beräkna $\binom{5}{2}$. Om vi tar bort de *k!* sista talen i täljaren har vi kvar: *5*4*3*. I nämnaren har vi kvar *(n–k)!*, det vill säga *3!*. Det senare är lika med *3*2*1*. Nu beräknas först *5/3* och sedan *4/2*. Dessa båda multipliceras varefter *3/1* beräknas som slutligen multipliceras med resultatet från den föregående beräkningen. Resultatet blir *10*.

Samma resultat som om vi hade beräknat *5!*, *3!* och *2!* samt genomfört beräkningen enligt följande: *120/(6*2)*, vilket också hade fungerat när talen är låga. Den andra algoritmen är dock bättre då den klarar större tal innan det blir problem med representationen i datorn.

Det är följaktligen viktigt att välja en bra algoritm för det problem som finns. I exemplet ovan kunde vi råka ut för vissa typer av problem. I andra fall kan det röra sig om att olika algoritmer tar olika lång exekveringstid, vilket förmodligen de båda olika lösningarna i exemplet också gör. Om exekveringstiden är en viktig faktor är det viktigt att välja en effektiv algoritm sett ur det perspektivet.

En annan sak som måste bestämmas är användning av olika datastrukturer. I många fall finns det olika sätt att representera data. Ett typ exempel är huruvida data skall representeras i form av en vektor eller som en länkad lista. En vektor motsvarar det normala matematiska begreppet. En länkad list innebär att data representeras som enskilda enheter som sedan länkas samman genom så kallade pekare. Pekarna brukar peka på föregående och nästa enhet i listan. Vi skall inte gå djupare in på detta exempel, utan använder det enbart för att illustrera att det finns viktiga designval att göra även när det gäller datastrukturer. Det bästa valet avgörs ofta beroende på tillämpning och behov, där det senare refererar till hur avsikten är att arbeta med informationen.

I vissa fall kan det vara viktigt att även låta designen gå ännu djupare, det vill säga att göra en design av exekveringsaspekter som till exempel minnestilldelning och till och med ner till bitnivå. Anledningen till att detta kan behövas kan till exempel vara mycket hårda tidskrav.

6.5 Designkvalitet

När det gäller kvaliteten på själva designen, det vill säga inte kvaliteten på systemet som utvecklas med hjälp av designen, diskuteras normalt två huvudkvalitetsattribut, nämligen koppling (eng. coupling) och samhörighet (eng. cohesion).

Koppling relaterar till relationen mellan olika komponenter/enheter, där god kvalitet karakteriseras av låg koppling. Skälet till detta är att det är lättare att hantera programvaran om dess delar inte är alltför starkt kopplade. Om vi för ett ögonblick återvänder till påverkansanalysen som diskuterades i föregående kapitel framstår det som ganska klart att om kopplingen i designen är hög kommer ny funktionalitet som skall läggas in troligen att påverka stora delar av programvaran. Det är inte önskvärt.

Samhörighet å andra sidan är en fråga om att titta in i komponenterna. Det som eftersträvas för en god design är att funktionalitet i en komponent skall hänga samman, det vill säga den skall inte mer eller mindre slumpmässigt ha placerats i samma komponent. Det är viktigt att eftersträva att innehållet i en komponent har bra samhörighet och därmed att placeringen i samma komponent kan motiveras. En god design kännetecknas av hög samhörighet.

När vi ser till kvaliteten på en design är målsättningen att ha hög samhörighet inom komponenter och låg koppling mellan komponenter. Det finns ett antal mått för designbeskrivningar som försöker fånga dessa båda kvalitetsaspekter. Svårigheten är att det inte är det finns många potentiellt olika sätt att mäta dem och att mätningarna också kan bero på designmetoden. Några olika designmetoder för programvara tas upp i nästa avsnitt. Utmaningen med mätningar inom programvaruutveckling behandlas kortfattat i Kapitel 14.

6.6 Designmetoder

Designprinciper för programvarusystem behandlades tidigare i detta kapitel. När det gäller specifika metoder finns det ett antal metoder för varje princip. Det är därmed omöjligt att ge en fullständig översikt. Valet av metod beror ofta på tillämpning och företagsstandards. Olika företag tar ofta beslut om vilken designmetod som skall användas. Det är därmed inte upp till respektive projekt att besluta det. Anledningen till att en standard används är många. Ett skäl är utbildning av personalen. Ett annat är inköp av verktygs-

stöd. Detta gör att ett enskilt projekt har kanske inte möjlighet att välja den bästa metoden utan styrs av andra faktorer.

Utöver att det finns ett antal metoder för varje princip finns det även metoder som är anpassade för olika typer av tillämpningar. De tillhör någon av principerna omnämnda ovan, men har utöver det även anpassats till olika typer av tillämpningar. Det finns metoder speciellt anpassade för realtidssystem, till exempel telekommunikationssystem, och databassystem.

Exempel på designmetoder:

JSD: Jackson System Development
Detta är en metod som utvecklades under 1980-talet. Den kan beskrivas om bestående av tre steg. Först kommer ett modelleringssteg där problemet analyseras och modelleras i termer av ingående enheter och vad de utför. I det andra steget, nätverkssteget, läggs interaktioner mellan enheter till och kommunikation med omgivningen tas in i modellen. I det sista steget överförs den logiska designen till en fysisk design.

SDL: Specification and Description Language
Detta språk utvecklades också initialt på 1980-talet. Det är ett språk som i första hand bygger på processer som innehåller tillstånd och kommunikationen mellan processerna görs genom skickande av signaler. Signalerna gör att de olika processerna byter tillstånd.

ER-diagram: Entity Relationship Diagrams
Detta är en metod anpassad för datamodellering. Metoden används ofta för utveckling av databassystem. Den bygger på identifiering av enheter och attribut hos dem. Därefter definieras relationerna mellan enheterna. Ett enkelt exempel är en databas för anställda vid ett företag. Företaget är en entitet och ett attribut är företagets namn. Företaget har ett antal personer anställda. Det betyder att det finns en relation mellan ett företag och *n* personer anställda vid företaget. Varje person har attribut som namn och lön. Modellen kan sedan utvidgas till att inkludera andra saker som är relaterade till företaget, till exempel olika produkter eller tjänster som företaget tillhandahåller. Det är också möjligt att definiera roller och ansvar som de anställda har.

UML: Unified Modeling Language
Denna metod är en sammanslagning och i viss utsträckning en kompromiss mellan tre personer (varav en svensk, nämligen Ivar Jacobson, som var huvudmannen bakom metoden ObjectOry) som hade utvecklat vardera en objektorienterad designmetod. Istället för att dessa metoder skulle konkurrera erbjöd ett företag dem att utveckla en gemensam metod. Metoden kom att kallas UML.

Det finns ett antal modeller inom ramen för UML. Den kanske viktigaste och mest centrala modellen är användningsmodellen, som skall fånga in den tänka användningen som olika aktörer kan ha. Begreppet aktör används som ett samlingsnamn på saker som kommunicerar med den programvara som utvecklas. Det betyder att en aktör inte behöver vara en person utan kan mycket väl vara andra system.

Användningsmodellen är klistret mellan ett antal andra modeller. Där analysmodellen skall förfina och strukturera de funktionella krav som fångades med användningsmodellen. Designmodell används för beskriva den fysiska realisationen av användningsmodellen och analysmodellen. Det finns också en implementationsmodell som är nära kopplad till den fysiska realiseringen. Implementationsmodellen är en förfining och detaljering av designmodellen. Vidare finns en testmodell och slutligen en distributionsmodell (eng. deployment model).

UML är en objektorienterad metod, vilket betyder att den i grund och botten bygger på objekt, klasser och deras relationer. Ett objekt är en verklig sak eller ett verkligt koncept. En klass är en samling av objekt som har gemensamma karakteristika.

UML har blivit en defacto standard för objektorienterad programvarudesign. Det innebär dock inte att det inte finns kritik mot modellen. En kritik är att språket blivit för stort, det vill säga många av begreppen som fanns i de tre metoderna som får betraktas som föregångare till UML finns också med i UML. Vidare har det stora antalet modeller kritiserats då det gör det svårt att vara konsistent. Det senare har dock förbättrats efterhand som verktygsstödet har förbättrats.

Det finns hyllmeter skrivit om UML och för en djupare kunskap om någon av de presenterade designmetoderna eller någon annan

metod hänvisas till de böcker som fokuserar på enskilda designmetoder.

Det viktigaste i detta avsnitt är kunskapen om att det finns ett stort antal designmetoder. Vidare är det viktigt att inse att det inte alltid är möjligt att välja den optimala designmetoden, utan att det i många fall redan är bestämt vilken metod som måste användas. Oavsett vilket är det viktigt att veta styrkor och svagheter hos den metod som används.

7 Implementation

7.1 Inledning

Programmering är oftast det första kontakten som personer har med programvaruutveckling. Det kan ske antingen genom programmering hemma till husbehov eller genom att de första kurser som ges i anslutning till programvauutveckling ofta är inriktade på programmering. Detta är naturligt då programmeringen är grunden för att kunna instruera en dator att utföra det som önskas, till exempel visa något speciellt på skärmen eller skicka någon viss signal. Det betyder att först säger krav ”vad” programvaran skall göra. Sedan tar designen hand om ”hur” det principiellt skall genomföras. Slutligen innebär implementation eller programmering att designen översätts till ett språk som med hjälp av verktyg (som kompilatorer och operativsystem) kan tolkas av en dator.

Det är dock lika viktigt att tidigt inse att programmering inte är tillräckligt för att kunna utveckla ett större programvarusystem. Det behövs en rad andra kunskaper och färdigheter varav många berörs kortfattat i denna bok. I ett industriellt perspektiv när större programvarusystem utvecklas är programmeringen endast en relativt liten del. Det finns en rad siffror i litteraturen på detta. Som ett riktmärke kan nog sägas att mellan 5–25 % av arbetstiden läggs på programmering. Det varierar mycket mellan olika företag, tillämpningar och inte minst storleken på programvaran som utvecklas.

Programmeringen upplevs av många som en rolig teknisk utmaning, medan andra tycker att det är en ganska tråkig och asocial aktivitet. Det skall dock betonas att inom ramen för ett stort programvaruutvecklingsprojekt finns plats för både de som tycker om programmering och de som föredrar annan typ av arbete. I större projekt finns ett stort behov av samverkan och samarbete, vilket gör

att det finns många intressanta arbetsuppgifter för dem som föredrar att arbeta tillsammans med andra människor snarare än att sitta på sitt rum och programmera. Å andra sidan behövs duktiga programmerare alltid i projekten också. Den gemensamma och stora utmaningen är dock att lösa problemet att från kravställarnas kravbild skapa ett bra programvarusystem som håller för vidareutveckling över en lång tid.

När det gäller programmering finns det ett stort antal böcker som lär ut olika programmeringsspråk. Vidare måste programmering övas flitigt. Två centrala begrepp i programmering är syntax och semantik. Syntaxen är reglerna som gäller för att skriva språket korrekt. Detta gäller både för naturligt språk (som svenska) och programmeringsspråk. Det gäller att skriva korrekta uttryck. Semantiken definierar hur ett uttryck i ett speciellt språk skall tolkas. Detta är oerhört väsentligt i ett språk som måste vara helt entydigt för att kunna förstås av en dator. I det naturliga språket går det inte att ha lika tydliga regler utan det är till slut alltid en tolkningsfråga. Det närmsta vi kommer gemensamma tolkningar är i samband med lagtext, där jurister enas om vissa tolkningar av lagtexten.

Det som många gånger upplevs svårast när det är dags att lära sig programmera är tankesättet. Det är viktigt att inse att ett programmeringsspråk inte är något annat än ett mycket begränsat språk jämfört med det naturliga språket. Begränsningarna är införda för att göra språket entydigt och därmed tolkningsbart av en dator. Att lära sig att programmera innebär följaktligen att lära sig att lösa problem med en mycket begränsad vokabulär. Varje programmeringsspråk innehåller ett visst antal konstruktioner som kan användas av programmeraren och oberoende av problemet är det dessa konstruktioner som måste användas. I naturligt språk finns det i närmsta oändligt antal olika sätt att uttrycka sig på, men i programmering är det mycket begränsat.

Programmering innebär systematisk och logisk strukturering av ett problem och lösningen dokumenteras sedan av ett mycket begränsat antal konstruktioner, det vill säga de konstruktioner som tillåts av det programmeringsspråk som används.

Konstruktionerna som åsyftas kan exemplifieras med en if-sats, det vill säga en konstruktion som testar om ett visst villkor är uppfyllt

och beroende på om det är uppfyllt eller inte kan olika saker ske. En annan vanlig konstruktion är en for-slinga. Denna utgår från att något skall utföras ett visst antal gånger, vilket görs inom slingan, och när det utförts det angivna antalet gånger går exekveringen vidare efter slingan.

De olika konstruktionerna beror både på typen av språk, det vill säga vilken paradigm tillhör språket, och det aktuella språket. En paradigm är i detta fall en princip på vilken ett språk är uppbyggt, det vill säga vad som är grundidén i språket. Det kan liknas vid de olika designprinciper som diskuterades i föregående kapitel. De fyra vanligaste paradigmerna för programmeringsspråk beskrivs kortfattat i nästa avsnitt.

7.2 Programmeringsparadigmer

Det finns ett antal olika principiella sätt att programmera på, det vill säga olika sätt att bygga upp programmeringsspråk. De fyra vanligaste berörs nedan och exempel på språk inom respektive paradigm ges.

Några saker är värda att notera. För det första god kunskap, kompetens och färdighet i ett programmeringsspråk för en paradigm skall göra det relativt enkelt att lära sig ett annat språk inom samma paradigm. Anledningen är att det i första hand är frågan om att lära sig en ny syntax och semantik, men själva sättet att bygga upp en lösning är densamma.

För det andra går det i många fall att skriva i enlighet med en paradigm även om ett språk från en annan paradigm används. I princip betyder detta att styrkan och möjligheterna i den andra paradigmen och dess språk inte utnyttjas på ett bra sätt. Det finns många exempel på personer som lärt sig ett nytt språk, men i princip löst problemen i enlighet med det språk som användes tidigare. Detta är en anledning att det är viktigare att lära sig språk från olika paradigmer ordentlig än att lära sig flera språk från samma paradigm.

Imperative programmering
Ett program är tänkt som en sekvens av instruktioner och koden organiseras vanligtvis i procedurer och funktioner, det vill säga specifika funktioner kan anropas från en annan del i programmet. Programmet körs genom att sekvensen av instruktioner exekveras. Exempel på språk av denna typ är C och Pascal.

Funktionell programmering
Ett program byggs upp av ett antal funktionella definitioner. Programmet exekveras genom anrop av en funktion med någon eller några parametrar för att få ett svar. Funktionerna i detta fall är grundstommen i programmeringen, medan för imperativa programmeringsspråk finns ofta möjlighet att använda procedurer eller funktioner för att strukturera programmet. Exempel på språk av denna typ är ML och Lisp.

Logikprogrammering
Ett program består av ett antal satser som är skrivna med någon form av logik. Exekveringen av ett program innebär att programmet får vissa fakta som avsikten är att programmet skall bevisa. Exempel på språk av denna typ är Prolog.

Objektorienterad programmering
Ett program ses som ett antal objekt som utbyter information sinsemellan. Exekvering av programmet görs genom att ett meddelande skickas till ett objekt. Exempel på språk av denna typ är Java och C++.

Det är viktigt att vara medveten om dessa olika paradigmer för programmering. Dock kan ett programmeringsspråk stödja mer än en paradigm. Det innebär att det i viss mening är upp till den enskilde programmeraren att avgöra hur språket skall användas och följaktligen vilken paradigm som skall användas. De exempel som är givna ovan skall ses som det språket primärt stödjer. Det är dock möjligt att till exempel med C++ välja att använda det antingen som ett imperativt språk eller ett objektorienterat språk. Det görs genom att programmeraren inte använder de objektorienterade möjligheterna i C++.

Det går inte att säga att en typ av språk generellt sätt är bättre eller mer lämpat. Valet av språk beror många gånger av tillämpningen

eller helt enkelt att just ditt företag har valt att använda ett specifikt språk. Användningen av ett språk medför kostnader i form av verktygsstöd och utbildning, vilket gör att företag oftast väljer att använda ett mycket begränsat antal olika språk. Det är dock odiskutabelt att de imperativa och objektorienterade språken är mest använda och därmed i någon mening viktigast att kunna.

Utöver dessa fyra paradigmer finns det visuell programmering och programmering med hjälp av script. Visual basic är ett exempel på ett visuellt programmeringsspråk. Den visuella programmeringen gör också att gränsdragningen mellan designmetoder med grafisk representation från vilken i vissa fall programkod kan genereras blir otydlig. Detta tas inte upp för att det är ett problem utan för att understryka att gränserna mellan olika aktiviteter inom programvaruutveckling inte alltid är helt klara. Det finns en rad olika scriptspråk också. Ett exempel är Perl.

Språken ovan brukar refereras till som varande högnivåspråk. Det innebär att de av datorn översätts till lägre nivåer för att bli tolkningsbara av en dator. Den lägsta nivån är representation i form av ettor och nollor som omnämndes i Kapitel 2. I undantagsfall kan programmering ske i assembler. Detta är en mer maskinnära form där instruktionerna till datorn är på lägre nivå, till exempel genom att programmeraren instruerar datorn om i vilka register på datorn som olika data skall sparas. Denna form av programmering användes ofta förr för att säkerställa att programmen exekverades så snabbt och optimalt som möjligt. Med hårdvarans utveckling och bättre stöd för högnivåprogrammering har användningen av assembler minskat.

I utveckling av programvara har vi nu översiktligt behandlat hur krav på en produkt fångas in och hur dessa krav hanteras. Vidare har överföringen av kraven från vad programvaran skall göra till hur det skall ske presenterats genom introduktionen av design. Utgående från designen sker därefter programmeringen för att överföra lösningarna på problemen, som representeras av kraven, till en form som kan hanteras av en dator. Hittills har fokus varit på genomförandet av utvecklingsaktiviteter som har till huvudsyfte att skapa något nytt. Det är dock oerhört viktigt att det nya som skapas också är korrekt. Det är långt ifrån trivialt att säkerställa att pro-

gramvaran som utvecklas uppfyller de krav och förväntningar som kravställarna hade från början.

För att försöka säkerställa detta finns två principiellt olika angreppssätt, nämligen statisk och dynamisk verifiering och validering. Den dynamiska verifieringen och valideringen kräver datorexekvering, till exempel genom exekveringen av koden. Allt annat betraktas som statisk verifiering och validering.

Verifiering är kontrollen att transformationen från krav till exekverbar kod har skett på ett korrekt sätt. Validering innebär säkerställning att det är rätt programvara som utvecklas. Det betyder att enkelt uttryck innebär verifiering och validering:

- Verifiering: Är programvaran rätt?
- Validering: Är det rätt programvara?

Fördelen med den statiska verifieringen och valideringen är att den kan genomföras på alla typer av dokument och delprodukter som tas fram under utvecklingen. Den vanligaste formen av statisk verifiering och validering är granskning. En introduktion till granskningar ges i Kapitel 8. När det gäller dynamisk verifiering och validering finns huvudsätt: användning av prototyper och test. Framtagning av prototyper ger kravställaren en möjlighet att tidigt bilda sig en uppfattning om programvaran, eller rättare sagt oftast någon specifik del av programvaran. Det vanligaste exemplet är användargränssnittet. Prototyper behandlas kortfattat i Kapitel 9 tillsammans med test.

8 Granskning

8.1 Programvarufel

Programvarufel är tyvärr både vanliga och kostsamma. Anledningen till detta är att utveckling av programvara är en fråga om design och inte produktion. Det gör att fel smyger sig in då utvecklaren gör misstag eller inte har insett all kopplingar till andra delar eller konsekvensen av en viss lösning. Det som gör programvarufelen kostsamma är det alltför ofta långa avståendet mellan introduktion av fel och detektion av fel. Detta beror på att utvecklingen har fortskridit baserat på felaktiga premisser, till exempel en felaktig design. Konsekvensen blir att felet kan ha propagerat till olika delar i programvaran och uppdateringen blir mer omfattande än om felet hade upptäcks strax efter det att det introducerades.

När det gäller fel är det brukligt att skilja på de fel som utvecklaren gör, ibland kallade misstag, felen som finns i programvaran och felyttringen. Ett fel kan finnas i programvaran utan att det någonsin upptäcks. Skälet är att det är enbart en viss kombination av förutsättningar som gör att felet skulle synas och om de aldrig inträffar upptäcks aldrig felet. Det är därmed viktigt att skilja på fel och felyttring. Antalet fel i en programvara är kopplad till korrektheten, medan felyttringarna är kopplade till tillförlitligheten. Det senare definieras, lite förenklat, av antalet korrekta exekveringar i jämförelse med det totala antalet exekveringar av en programvara. Tillförlitlighet är därmed en extern egenskap hos programvaran medan korrekthet är en intern egenskap.

I det här sammanhanget är det viktigt att nämna att det finns studier som pekar på att ett fåtal fel i programvaran orsakar ett stort antal felyttringar. Detta betyder också att det finns ett antal fel i programvaran som knappast påverkar den externt upplevda kvaliteten

i form av programvarufel. Detta är viktigt att ha i åtanke under utvecklingen i samband med felsökning. Det vore naturligtvis önskvärt att ta bort alla fel, men ofta är detta inte möjligt på grund av de kostnader som är involverade eller den tid det skulle ta att hitta alla felen. Därmed kan det vara viktigt att fundera på hur felen som är mest väsentliga hittas på ett effektivt sätt.

För att adressera ovanstående utmaning med programvarufel har ett antal olika tekniker utvecklats. Det finns två principiellt olika grupper av metoder för feldetektering: statiska och dynamiska. Den dominerade tekniken när det gäller statiska metoder är granskningar, vilket diskuteras i detta kapitel. De dynamiska teknikerna är i första hand arbete med prototyper och olika typer av test. De dynamiska teknikerna, med tonvikt på test, behandlas i Kapitel 9.

Vidare finns det ett antal aktiviteter som har till uppgift att försöka hantera problematiken med programvarufel. Dessa är:

Undvikande
Det bästa är naturligtvis om fel kan undvikas. Det är dock lättare sagt än gjort, men oavsett detta är det viktigt att tidigt ha fokus på stöd för att programvaran utvecklas på ett korrekt sätt. Det betyder att det måste finnas bra stöd i form av utvecklingsprocessen, tidigare produktdokumentation och verktyg. Att undvika fel innebär ofta arbete med olika typer av förbättringsarbete i anslutning till utvecklingsprocessen.

Detektering
Detta är den mest omfattande åtgärden, det vill säga olika metoder för att detektera fel. Ovan nämndes granskningar och test, vilka presenteras i mer detalj nedan respektive i nästa kapitel.

Korrigering
Det räcker inte bara att ha stöd för att hitta fel, utan det måste även finnas bra stöd för rättning. Det kan innebära verktyg som stödjer avlusning (eng. debugging), det vill säga borttagning av fel. Vidare är det viktigt att det finns stöd för att kunna säkerställa att rättningen är korrekt och att den inte har introducerat en rad nya fel. Detta betyder att felrättningar i sin tur måste verifieras och valideras, vilket i sin tur bidrar till kostnaden för fel.

Tolerans

I vissa typer av system arbetas med feltolerans. Det kan innebära ett antal olika saker. För det första kan det innebära att ett system kan gå tillbaka till en gammal version av programvaran om fel uppstår i en ny version av programvaran. En annan möjlighet är att introducera parallella datorer med olika programvarulösningar, det vill säga olika utvecklingsteam (rimligen minst tre) har tagit fram olika lösningar och majoritetsbeslut tas med avseende på svaren. Det senare är kostsamt och det finns ingen garanti att lösningarna blir oberoende, vilket är önskemålet. Problemet med oberoende är att oftast måste de olika teamen utgå från en gemensam specifikation. Detta leder till att eventuella fel i specifikationen eller tolkningsproblem av specifikationen mycket väl kan drabba flera av teamen. Denna typ av utveckling brukar refereras till som varande n-version programmering. Tekniken har använts i en del fall och då i första hand vid säkerhetskritiska tillämpningar.

Utvärdering

Oavsett ovanstående punkter är det viktigt att kunna utvärdera kvalitetsnivån (i form av programvarufel). Kvalitet, olika aspekter på kvalitet och olika kvalitetsattribut diskuteras mer ingående i Kapitel 13. Vi skall här endast beröra kvalitet kortfattat ur ett programvarufelsperspektiv.

Det är viktigt att kunna skatta nuläget, vilket brukar refereras till som att estimera. Vidare är det viktigt att kunna prediktera den framtida utvecklingen. Det är till exempel mycket relevant under testfasen, då det kan vara av stort intresse att veta när en viss kvalitetsnivå har uppnåtts. En viktig fråga är om kvalitetsnivån har nåtts till den inplanerade releasetidpunkten. Om en sådan förutsägelse kan göras erhålls underlag för att besluta om mer testresurser och/eller introducera fler parallella teststationer. Slutligen kan det vara viktigt att i vissa fall kunna certifiera en specifik kvalitetsnivå. Det är dock ingalunda lätt att göra varken estimering, prediktering eller certifiering. Det är dock ingen ursäkt för att inte försöka göra det, då det är en mycket viktig aktivitet om vi vill hävda att vi utvecklar programvara på ett ingenjörsmässigt sätt. Certifiering är en mycket viktig aktivitet om komponentbaserad utveckling skall fungera, speciellt om

avsikten är att använda inköpta komponenter. I detta fall är tanken att komponenter skall kunna köpas från en leverantör och det blir då speciellt viktigt att säkerställa kvaliteten på den levererade komponenten då den påverkar vårt systems kvalitet. Certifiering i dessa fall är ofta svår då det inte är säkert att komponenten levereras med källkoden utan kanske bara som en exekverbar fil.

8.2 Verifiering och validering

Ovanstående pekar på behovet av att försöka säkra programkvaliteten. Som nämnts tidigare finns det två principiella aspekter på detta:

Verifiering: Är programvaran rätt?
Vid verifiering gäller det att avgöra om nuvarande status på programvaran stämmer med det som gjorts tidigare. I princip antas att kraven är korrekt ställda och att målsättningen nu är att avgöra till exempel om designen och koden motsvarar de ställda kraven. I någon mening kan hävdas att programvaran är rätt om design och kod motsvarar kravspecifikationen. Problemet är dock att kravspecifikationen är långt ifrån alltid korrekt och fullständig. Många programvaruprodukter utvecklas åt företag och organisationer som själva inte är experter på programvara. Detta gör att de inte heller är experter på att agera kravställare för programvaruprodukter. Även om kravställarna är kunniga inom programvaruområdet är det ofta svårt att överblicka alla aspekter då kraven först formuleras. Många krav kan saknas, andra krav är fel och flera krav ändras då mer kunskap växer fram under utvecklingen. Detta gör programvaruutveckling till en stor utmaning. Samtidigt är det mycket viktigt att formulera så bra krav som möjligt tidigt, men också vara medveten om att det kommer att ske ändringar under utvecklingen. Om utgångspunkten är de initiala kraven då är det möjligt att programvaran kan verifieras, men å andra sidan är det också möjligt att ingen trots allt är nöjd med slutresultatet. Detta leder till att det inte bara är viktigt att verifiera en produkt utan även validera.

Validering: Är det rätt programvara?
Denna fråga är mer intrikat än den förra. Att göra någonting rätt är bra, men det är mycket bättre om det dessutom är rätt sak som görs. Detta innebär att avstämning måste inte bara ske mot tidigare dokumentation utan mot ofta mot kravställare, marknader, kunder och andra intressenter för att säkerställa att inte bara ställda krav uppfylls utan även outtalade krav, önskemål och förväntningar. En stor svårighet med detta är att det ofta krävs att produkten kan beskådas. Detta betyder att programvaran ofta behöver exekveras. Det är få personer som kan visualisera exekveringen av en programvara från att läsa dokumentationen. I praktiken betyder detta att det ofta är sent i utvecklingen som validering kan ske och att det krävs dynamiska metoder. Dessa metoder behandlas i Kapitel 9.

Det är dock viktigt även med de statiska metoderna då deras stora fördel är att de kan göras genom hela utvecklingsprocessen. Sammantaget betyder det att både statiska och dynamiska metoder är viktiga att använda för att säkerställa att programvaran blir både rätt utvecklad och att det i slutänden är rätt produkt som utvecklats.

Det är inte tillrådligt att endast använda antingen statiska eller dynamiska metoder. Om enbart statiska metoder skulle användas skulle det innebära att programvaran inte exekveras i en testmiljö och att avstämning huruvida det är rätt produkt som utvecklats är oerhört svår att göra. Ingen organisation som utvecklar programvara skulle överväga att enbart förlita sig på statiska metoder.

Vidare är det inte att rekommendera att förlita sig enbart på dynamiska metoder heller. Problemet med detta är att de fel som finns kommer att upptäckas sent och tid har slösats bort på att utveckla programvaran baserat på felaktiga indata i något steg. Tyvärr sker det alltför ofta att de statiska metoderna prioriteras ned i samband med att det blir ont om tid. Den stora risken med detta är att fel hittas sent och att det därmed leder till omarbete som sammanlagt leder till mer totalt arbete än om statiska metoder hade använts systematiskt genom hela utvecklingsprocessen.

En annan möjlig förklaring till att de statiska metoderna försummas är att de ofta kräver manuellt arbete i form av läsning och genomgång av skrivna dokument, inklusive koden. Många utvecklare

tycker att denna genomgång är mindre kreativ, än själva utvecklingen, och därmed mindre stimulerande. Detta gör att det är en aktivitet som få saknar när den prioriteras bort på grund av tidsbrist eller andra skäl. Saknaden uppstår inte förrän det upptäcks alltför många fel sent i utvecklingen. Detta betyder med andra ord att förhoppningen att det inte skulle finnas fel grusas.

Många forskningsundersökningar har visat att tidig felupptäckt är kostnadseffektivt. Statisk verifiering är följaktligen ekonomiskt försvarbar. Med anledning av detta följer i nästa avsnitt en introduktion till den vanligaste formen av statisk verifiering, nämligen granskning.

8.3 Granskningar

8.3.1 Inledning

Det finns två huvudtyper av statisk verifiering: analys och granskning. Analysen kräver att någon form av formell notation har använts, till exempel i form av en designmetod som byggs på en formell notation eller programkoden. Det vanligaste är analys av programkod. Analysen, som oftast görs med hjälp av något verktyg, innebär någon form av identifiering av problem med koden, till exempel i form av kod som inte kan nås vid exekvering eller loopar i koden som aldrig tar slut. Det går att göra en hel del olika typer av analyser, men problemet är att de formella notationerna oftast kommer in sent i utvecklingen, även om det finns metoder för att skriva krav formellt. Om analysen inte kan genomföras förrän koden finns på plats sker den statiska verifieringen bara marginellt innan den dynamiska verifieringen genomförs. Det kan dock vara lönt att göra denna typ av analys då den kan ske automatiskt och det inte behövs några testfall. Det senare återkommer vi till i Kapitel 9.

När det gäller granskning finns det tre olika begrepp som används i samband med olika typer av statisk verifiering av dokument. Deras

exakta betydelse varierar lite beroende på källan, men i princip finns följande tre typer av granskningar:

Granskningar (eng. inspections)
Det är lite olyckligt, men ofta används ordet granskning som ett övergripande begrepp och för en specifik typ av granskning. Denna typ av granskning borde kanske bättre benämnas felfokuserad granskning, då det ger en bättre bild av vad som avses. Det är dock oftast möjligt att ur sammanhanget avgöra om denna specifika granskningsmetod avses eller om avsikten är att diskutera granskning som ett allmänt begrepp.

Denna typ av genomgång syftar till att hitta fel på ett strukturerat sätt och det är den vanligaste typen av granskning. Metoden används ofta för att jämföra det som granskas med något tidigare dokument eller kunskap som granskarna besitter. Denna metod presenteras i mer detalj nedan.

Genomgång (eng. walkthrough)
En metod som är närbesläktad med den föregående är genomgången. En väsentlig skillnad är dock att vid genomgången leder utvecklaren åhörarna genom materialet och förklarar. Åhörarna ifrågasätter och diskuterar. Fel hittas som en följd av genomgången, men huvudsyftet är inte lika tydligt på att identifiera fel utan genomgången kan också ha som syfte att kommunicera den gjorda lösningen till andra.

Besiktning (eng. review)
Målsättningen med en besiktning är också att gå igenom och diskutera, men syftet är oftast att erhålla kommentarer eller att få någonting godkänt. En besiktning kan göras av produktdokumentation, men den sker ofta också för att godkänna till exempel en utvecklingsprocess eller en del av densamma. Besiktningen är inte speciellt inriktad på fel, vilket ofta betyder att en besiktning genomförs utan att det görs en jämförelse med något annat material.

Alla tre metoderna ovan är värdefulla och har sina syften. Nedan är fokus dock på den första av metoderna ovan, då det är den vanligaste och den som är starkast relaterad till verifiering och validering av programvaran.

8.3.2 Granskningsprocessen

Granskningar har säkert informellt genomförts så länge som programvara har utvecklats. De formaliserades dock först i mitten på 1970-talet av Michael Fagan från IBM. Han definierade i första hand processteg och roller för granskningarna. Fagangranskningar genomförs fortfarande idag på många ställen enligt de grunder som Fagan definierade eller med små variationer. Enligt Fagan innehåller granskningen följande processteg:

Planering
Inom ramen för planeringen beslutas om granskningsobjektet. Vidare identifieras lämpliga granskare. Tidplan och plats för möten under granskningen bokas. Tidplanen innefattar resterande steg i processen. Det är viktigt att varje steg tilldelas tillräckligt med tid så att det finns utrymme att genomföra dem på bästa möjliga sätt. Det är speciellt viktigt att tiden för individuell förberedelse planeras så att samtliga granskare hinner förbereda sig på rätt sätt, det vill säga för att granskningen skall bli kostnadseffektiv.

Översikt
Avsikten med detta steg är att den ansvarige utvecklaren (även benämnd författaren) av det objekt som skall granskas skall ge en introduktion och översikt till materialet som ingår i granskningen. Detta steg anses inte obligatoriskt och i praktiken är det inte ofta som det används.

Förberedelse
Enligt Fagans beskrivning av processen är syftet med detta steg i första hand att bekanta sig med materialet. Målsättningen är att inhämta kunskap så att nästa steg, det vill säga mötet, blir effektivt. I många fall har de individuella förberedelserna utvecklats till att vara fokuserade på att identifiera fel. Det senare betyder att varje enskild granskare anstränger sig för att hitta så många fel eller problem som möjligt i granskningsobjektet.

Möte
Då förberedelserna inte, enligt Fagan, primärt var avsedda för att hitta fel och problem skedde felidentifiering i första hand på mötet. Då förberedelsernas karaktär har ändrats till att vara mer

fokuserade på att identifiera fel har även mötets karaktär ändrats. Eftersom felidentifiering i stor utsträckning idag sker under förberedelserna har mötet ändrats till att vara fokuserat på sammanställning av fel samt synergieffekter mellan granskare. Det senare syftar på att målsättningen är att hitta fel som kräver en kombination av olika granskares expertis. Ett protokoll med identifierade fel skrivs i samband med mötet och fungerar som indata till resterande steg i processen.

Omarbete
Efter mötet sker omarbete för att rätta och uppdatera granskningsobjektet i förhållande till de saker som framkom under mötet och som dokumenterades i protokollet. Normalt är det författaren (ansvarig person) av granskningsobjektet som genomför ändringarna. Tidigare i detta kapitel diskuterades skillnaden mellan fel och felyttring. En stor fördel med granskningar är att det som upptäcks är felen, vilket gör att det inte krävs någon felletning ty felet har lokaliserats under granskningen. Detta skiljer sig från test där det som ses är felyttringen, vilket gör att felet måste lokaliseras innan det kan rättas.

Uppföljning
Efter uppdatering och rättning sker uppföljning. Det är oftast upp till mötet att besluta hur uppföljningen skall ske. Detta beslut är avhängigt av felen och problemens allvarlighetsgrad. I normala fall brukar tre olika beslut finnas (från allvarligast till minst allvarligt): omgranskning, ändringarna skall godkännas av utsedd person och författaren får fullt ansvar för att ändringarna genomförs på ett bra sätt. Det är naturligtvis önskvärt att undvika omgranskning då det är stor risk att det leder till förseningar då en ny granskning skall läggas in i ett säkert redan pressat tidsschema.

I tillägg till processen ovan definierade Fagan ett antal roller som ingår i granskningen. Det är viktigt att notera skillnaden mellan personer och roller. Roller är något som personer tilldelas, vilket också betyder att en person i vissa fall kan inneha flera roller. De rollerna som Fagan definierade var:

Moderator
Denna roll är ansvarig för granskningen. Det betyder att moderatorn planerar granskningen och agerar ordförande vid mötet.

Författare
Detta är den person som tagit fram det objekt som skall granskas. Med tanke på att många olika typer av dokument kan granskas kan författaren arbeta som kravingenjör, designer, programmerare, testare eller inneha någon annan roll i organisationen.

Utvecklare
Utvecklaren är en roll som innehas av en eller flera personer som har god kunskap om utvecklingen i anslutning till granskningsobjektet. Rollen ansvarar för att identifiera fel och problem sett ur ett utvecklingsperspektiv.

Testare
På ett liknande sätt som utvecklarrollen är testaren ansvarig för att identifiera fel och problem i anslutning till sin expertis. I testarens fall skall granskare med denna roll identifiera fel och problem i anslutning till testning av det som beskrivs i granskningsobjektet.

Fagans grundtankar gäller fortfarande i mycket stor utsträckning idag, men inom ramen för forskningen har olika förbättringar i anslutning till Fagans idéer tagits fram. Några av dessa presenteras härnäst.

8.3.3 Varianter på granskningsprocessen

Det finns ett antal variationer på granskningsprocessen. Några av de mest uppmärksammade är:

Aktiva designgranskningar (eng. Active Design Review)
Utvecklarna av denna metod ansåg att Fagans granskningar blev ineffektiva genom att de ofta ledde till stora granskningsmöten samt att för stora delar granskades åt gången. De definierade en metod med många små granskningar där avsikten var att samtliga personer som deltog i respektive granskning skulle vara aktiva (i relation till hur det var under en stor granskning).

Två-persons granskningar (eng. Two-Person Inspection)
Ett återkommande problem vid granskningar är hur många personer som skall delta. Det är troligt att fler granskare hittar fler fel. Det är dock inte uppenbart att fler granskare är kostnadseffektivt. Anledningen är att även om tillägg av en granskare innebär att fler fel hittas kommer antalet nya fel som hittas i snitt att minska med antalet granskare. Anledningen till detta är att ju fler granskare som finns redan desto färre nya unika fel finns kvar att hitta. Detta leder till att det i grund och botten är ett kostnadsoptimeringsproblem att bestämma hur många granskare som skall användas. Optimeringen måste ske baserat på kostnaden för att lägga till en granskare till, antalet fel som ytterligare en granskare kan hitta och kostnaden för att inte hitta fel, det vill säga vad fel kommer att kosta att hitta senare i utvecklingen eller i drift.

Problemet med att det inte blir kostnadseffektivt med många granskare resulterade i formuleringen av denna metod. Tanken med metoden är att endast ha två personer med på granskningen: författaren och en granskare. Förslaget är i första hand inriktat på granskningar i små utvecklingsteam. Andra har senare förslagit att små granskningsteam skall speciellt användas sent i utvecklingsprocessen, det vill säga för granskning av kod. Det finns forskning som visar på att vid granskning av krav är det nog lämpligt med fler granskare än av kod, det vill säga det behövs fler granskare vid granskningar tidigt i utvecklingsprocessen.

Parallell granskning (eng. N-Fold Inspection)
Tanken bakom parallell granskning är att adressera samma problem som vid aktiva granskningar, det vill säga att det inte fungerar med ett stort granskningsteam. Förslaget på lösning här är att genomföra flera granskningar i parallell för att få flera oberoende mindre granskningsteam. Denna lösning är tänkt att öka effektiviteten i identifieringen av fel, men frågan är när denna typ av ansats är kostnadseffektiv.

Fasbaserade granskningar (eng. Phased Inspection)
Denna metod bygger på att en stor granskning delas in i olika faser. En av svårigheterna vid stora granskningar är att det är i det närmsta omöjligt att få en överblick och därmed kunna hitta alla potentiella fel och problem. Fasbaserade granskningar bygger på att upp till sex olika mindre granskningar genomförs i serie. Varje

liten granskning har ett specifikt fokus. Avsikten är att detta skall göra granskningen hanterbar och det är även möjligt att ha olika experter inne vid de olika granskningarna. Varje fas kan dessutom bestå av två steg. I det första steget är det en enskild granskare och i det andra steget är det flera granskare. Baserat på det första steget kan avgöras om granskningsobjektet är tillräckligt bra för att bli utsatt för en granskning med flera personer eller om författaren borde åtgärda mer innan det andra steget kan genomföras.

Än en gång är avsikten att lösa problemet med storlek på granskningar, dels storleken på granskningsobjektet och dels med antalet granskare vid en stor granskning. Det som är mest tilltalande ur ett kostnadseffektivitetsperspektiv är att de olika faserna är olika, det vill säga har olika fokus. Samtidigt kommer det att ta lång tid att genomföra granskningarna om faserna skall ske i serie.

Granskningar utan möte (eng. Inspection without meeting)
Ett alternativ som har lanserats för att dels minska arbetstiden avseende granskningar och dels för att minska ledtiden är granskningar utan möte. Undersökningar har visat att ofta tar en granskning lång ledtid eller kalendertid. Anledningen till detta är att det ofta är svårt att hitta mötestider för granskarna, då många personer skall koordineras och de ofta har alltför fyllda kalendrar. Med anledning av att alltmer av felidentifieringen har lagts på de individuella förberedelserna och synergieffekterna i olika studier är mindre än förväntat upplevs mötet mest som en sammanställning av redan hittade fel.

Andra hävdar å andra sidan att många ”fel” som identifieras under de individuella förberedelserna inte är fel och det kan mötet hantera. Vidare bidrar mötet i stor utsträckning till informationsspridning och fyller därmed en viktig funktion även om den inte direkt är relaterad till felidentifiering.

Än en gång är det inte möjligt att säga att detta förslag är bättre eller sämre under alla omständigheter, utan det är än en gång viktigt att ta beslut om möten i det enskilda fallet. Det viktigaste är nog flexibilitet att hålla möte när det bedöms som rätt och vid andra tillfällen inte genomföra mötet.

Om inget möte genomförs skickas granskarnas synpunkter in till en person som sammanställer identifierade fel och problem. Eventuellt skulle denna person baserat på indata kunna avgöra om ett möte behövs eller inte.

Sammanfattningsvis är det svårt att avgöra huruvida några av dessa granskningsvarianter är bättre än Fagans förslag. Det är stor möjlighet att den specifika situationen avgör vad som är det bästa angreppssättet. Det är högst troligt att olika metoder är lämpliga vid granskning av en ordbehandlare och av programvara för ett kärnkraftverk. Det viktiga är dock att i varje enskilt fall försöka tänka igenom vad som är lämpligt. Det gäller inte bara vilken granskningsmetod som skall användas utan även antalet granskare som är lämpligt. Rekommendationerna från forskningen skiljer sig åt, vilket beror på att olika tillämpningar och förmodligen olika ställen i utvecklingsprocessen ställer sina egna krav. Normalt varierar rekommendationerna mellan 2–6 granskare med fler granskare tidigt i utvecklingsprocessen och fler granskare för säkerhetskritiska system. En annan aspekt som kan varieras är det sätt på vilket förberedelse sker. Ett antal olika alternativ har identifierats. Några av dem presenteras i nästa avsnitt.

8.3.4 Lästekniker

Under de individuella förberedelserna kan granskarna ha olika typer av stöd, det vill säga för att gå igenom granskningsobjektet. Detta brukar refereras till som olika lästekniker. Än en gång finns det ett antal förslag:

Ad hoc
Detta är i grund och botten ingen teknik, då den avser att beskriva det fall då inget stöd ges till granskarna. Detta betyder att varje granskare studerar granskningsobjektet efter bästa förmåga.

Checklista
Detta är den metod som tas upp av Fagan. Tanken är att granskarna skall få stöd av en checklista. Den skall innehålla information om saker som granskaren skall leta efter. Principen är den-

samma som för flygkaptenerna innan start, det vill säga ett stöd för minnet att olika saker har tagits om hand. Checklistor kan antingen vara väldigt allmänna och det finns då en stor risk att de ger relativt lite stöd till granskarna, eller mer specifika till exempel för en specifik produkt eller ett specifikt projekt. I dessa fall blir checklistan mer anpassad till situationen och ger förhoppningsvis granskarna ett bättre stöd. En anpassad checklista är dock dyrare än en mer generell checklista, vilket än en gång leder till frågan om vad som är mest kostnadseffektivt.

Stegvis abstraktion
Denna metod är i första hand avsedd för kod. Avsikten är att granskning skall ske från lägre nivå till högre nivå. Detta betyder att istället för att granska koden radvis påbörjas granskningen med de funktioner som ligger i någon mening längst in i programmet. Målsättningen är att granska dessa och se till att de är korrekta, därefter kan granskning ske av nivån ovanför. När denna granskning sker förutsätts att den lägre nivån fungerar, det vill säga underliggande funktioner är korrekta. Detta stegvisa angreppssätt gör det möjligt att granska ganska stora program utan att för den skull behöva förstå hela programmet på en och samma gång.

Scenariobaserad
Metoden kallas ibland felbaserad läsning. Anledningen är att metoden har utvecklats utgående från ett antal scenarier som kopplas till olika feltyper. Olika granskare får sedan olika scenarier och information om att hitta olika typer av fel. Detta är ett sätt att försöka få olika granskare att fokusera på olika saker och därigenom är målsättningen att de hittar olika fel istället för att i princip huvudsakligen hitta samma fel. Scenariobaserade granskningar är ett sätt att försöka adressera det faktum att Fagan rekommenderar att det skall både finnas utvecklare och testare i granskningsteamet. Metoden här försöker formalisera detta genom att ge granskarna olika scenarier.

Perspektivbaserad
En annan lästeknik som också eftersträvar att ta hand om de olika rollerna som nämns av Fagan är perspektivbaserade granskningar. I denna metod eftersträvas att olika roller skall finnas

representerade av granskarna. De vanligaste rollerna som anges är utvecklare, testare och användare. Varje roll får specifikt stöd. Vidare förväntas varje roll som en del i sin individuella granskning att ta fram ett konkret resultat, till exempel kan testare vid granskning av en kravspecifikation bli ombedda att ta fram testfall. Detta gör denna lästeknik mer aktiv än de flesta andra.

Användningsbaserad
En annan variant är att utgå från användningen. Denna metod är utvecklad för att redan i granskningen försöka efterlikna användningen av programvaran. Funktionerna i programvaran prioriteras i förhållande till hur programvaran troligen kommer att användas. Det gör att frekvent använda funktioner granskas innan funktioner som används lite. Tanken är att granskarna blir trötta och då är det viktigt att tidigt granska de funktioner som är viktigast för användaren. Metoden måste dock kompletteras med speciella åtgärder om vissa funktioner är säkerhetskritiska. Det senare gäller till exempel nödstängning av ett kärnkraftverk som är oerhört viktigt, men som i bästa fall aldrig behöver användas.

Än en gång är det omöjligt att rekommendera vilken metod som är bäst. Det finns några undersökningar som pekar på att perspektivbaserade granskningar är bra att satsa på, dock krävs det att personerna som skall ha de olika rollerna verkligen också har erfarenhet från de rollerna. Detta betyder att det inte räcker att tilldelas rollen utan det krävs praktisk erfarenhet från rollen. Detta har konstaterats genom att studier med studenter som tilldelas rollerna, utan att studenterna har erfarenhet av rollen de tilldelas, inte kan påvisa att perspektivbaserade granskningar är bättre än någon annan lästeknik. Däremot när motsvarande undersökningar har gjorts med personer med erfarenhet från industri har det hittills visat sig att perspektivbaserade granskning fungerar åtminstone lika bra eller bättre än andra lästekniker.

Genomgången av granskningsprocessen och lästeknikerna visar tydligt på komplexiteten i programvaruutveckling. Granskningar är en väl avgränsad aktivitet som genomförs ett antal gånger i en utvecklingsprocess och trots det går det inte att rekommendera ett specifikt sätt att genomföra dem. Variationsmöjligheterna är många bara inom granskningar och skalas det upp till hela utvecklingspro-

cessen blir antalet möjligheter och val i det närmsta oändligt. Det är av denna anledning som ett ingenjörsmässigt angreppssätt till programvaruutveckling i stor skala är en nödvändighet.

Innan detta kapitel avslutas skall vi även introducera en intressant aspekt som individuell felidentifiering ger möjlighet till.

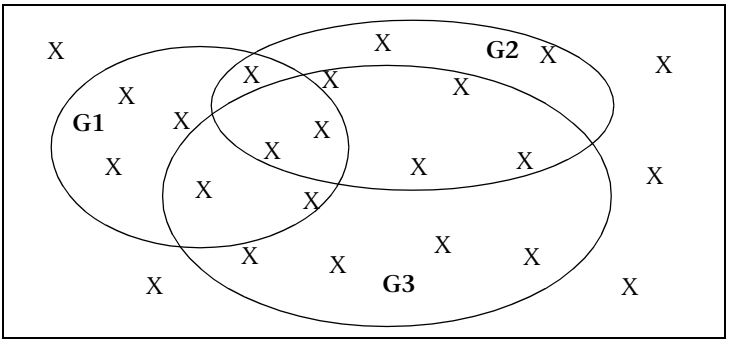
8.3.5 Felskattning

Vid granskningar sammanfattas utfallet som nämnts ovan i ett protokoll. Detta protokoll listar de identifierade felen. Ett stort problem är dock att det är svårt att veta hur mycket fel som finns kvar. Ett stort antal hittade fel kan innebära antingen att granskningsobjektet är dåligt eller att granskningen har varit mycket bra. En intressant möjlighet att få ett grepp om antalet kvarvarande fel kan erhållas genom användning av de metoder som finns för att skatta antalet kvarvarande fel efter en granskning. Det finns några olika möjligheter även här, men enbart ett alternativ presenteras nedan.

Forskare inom biologin använder en metod som kallas ”fånga och fånga igen”, eller på engelska capture-recapture (CRC), för att skatta olika djurpopulationer. Metoden bygger på att olika personer fångar djur, till exempel i en specifik skog, och när de fångat djuret noteras det samt djuret märks. När ett djur fångas som redan är märkt av någon annan noteras även detta. Baserat på de enskilda personernas protokoll kan sedan den totala populationen skattas. Metoden kan överföras till programvaruutveckling genom att låta personerna vara granskare och betrakta felen som djuren.

Grundtanken med metoden illustreras i Figur 8.1 med tre granskare. Fyrkanten i bilden illustrerar programvaran och inom ovalerna vad respektive granskare har hittat. Ovalerna visar följaktligen de tre granskarna (granskare G1, G2 och G3). I figuren syns också några

kryss som ligger utanför ovalerna. Dessa illustrerar fel som inte hittats.



Figur 8.1 Illustration av metoden CRC med tre granskare.

Från Figur 8.1 kan totala antalet hittade fel bestämmas. Vidare är det möjligt att avgöra dels hur många fel respektive granskare har hittat och överlappat mellan granskare. Utgående från denna information är det möjligt att med hjälp av statistiska metoder att skatta det totala antalet fel, vilket gör att antalet icke-hittade fel kan skattas.

Om en metod som CRC skall användas bör de individuella granskarnas protokoll samlas in och en skattning kan göras av den person som samlar in information. Skattningen skulle även kunna ligga till grund för att besluta huruvida ett granskningsmöte behövs eller inte. Om överlappet är litet mellan granskarna återstår troligen mycket fel och det kan vara lämpligt med ett möte. Å andra sidan om överlappet mellan granskarnas protokoll är stort finns förmodligen få fel kvar och ett möte kanske inte behövs. Det kan också vara värt att notera att skattningsmetodens precision blir bättre desto fler granskare och desto fler fel. Detta är dock inte i linje med andra önskemål, det vill säga färre granskare är billigare och färre fel är naturligtvis önskvärt ur ett allmänt kvalitetsperspektiv.

En liten brasklapp är på sin plats angående antagandena i de statistiska metoderna. Det finns lite olika typer av antagande beroende på exakt vilken statistisk metod som används för skattning, men principiellt kan sägas att metoderna kan vara känsliga för lästekniken. Det gäller speciellt om olika granskare har tilldelats till exem-

pel olika perspektiv, vilket gör att lästekniken som sådan bidrar till att minimera överlappet mellan granskare. Det kan vara kostnadseffektivt på sitt sätt, men å andra sidan minskar det möjligheten att skatta antalet kvarvarande fel. Samtidigt är det möjligt att använda skattningsmetoderna ändå på den tillgängliga insamlade informationen, men då måste det tas in i beräkningen att antagandena för de statistiska metoderna inte är uppfyllda. Det är dock möjligt att en organisation kan lära sig att kompensera för detta. Än en gång visar detta på vikten av ett ingenjörsmässigt förhållningssätt till programvaruutvecklingen. Det är enbart genom kloka beslut baserat på fakta som managers kan ta rätt beslut vid olika tidpunkter i utvecklingsprocessen.

9 Test

9.1 Inledning

Den dynamiska verifieringen och valideringen är mycket viktig, då det är det enda sättet att verkligen säkerställa att programvaran fungerar enligt intentionerna. Den statiska verifieringen som diskuterades i Kapitel 8 lägger en viktig bas för att kunna testa programvaran. I Figur 4.2 illustreras hur de olika faserna i utvecklingen kopplas till olika testnivåer. Figuren visar hur det finns olika testnivåer beror på, i någon mening, storleken på det som skall testas.

Dessa nivåer är dock inte det enda sätt som test kan delas in på. För det första finns det två olika principiella sätt att testa på: felletande testning och statistisk testning. Vidare finns det olika testmetoder. Dessa skiljer sig åt till exempel genom sättet som testfall tas fram på. Målsättningen med detta kapitel är att ge en introduktion till test och förklara hur olika nivåer, testtyper och testmetoder är tänkta att fungera. Utöver detta beskrivs användning av prototyper kortfattat.

Innan dessa olika saker berörs är det dock viktigt att belysa ett antal andra viktiga saker i samband med testning. Det första steget vid testning är testplanering. Testning bör integreras redan i den första programmeringskursen. Det är inte bara att skriva program som måste läras ut, utan även hur programmen som utvecklas skall testas. Det är inte ovanligt att testning genomförs genom att den som skrivit programmet själv exekverar programmet lite slumpmässigt med egna indata för att säkerställa att programmet fungerar. För små program med relativt enkel funktionalitet är detta kanske tillräckligt, men så snart programmen blir större är det inte det längre.

Varje program som skrivs borde testas med en eller flera medvetna strategier. Slumpmässig testning kan vara en sådan strategi, men den borde kompletteras även med strukturerad och systematisk

testning. Detta betyder att test måste planeras. För varje programvarusystem som skall testas skall det finnas en testplan, där programvarusystem kan vara allt från ett litet program utvecklat av en person till ett stort system utvecklat av ett stort antal människor över en lång tid. En testplan borde innehålla nedanstående, där djupet på beskrivningar beror av storleken på programvarusystemet. Innehållet i en testplan:

Testprocess
Denna del är den mest omfattande. Den borde innehålla beskrivningarna av på vilka nivåer test skall genomföras och vilka testmetoder som skall användas. Exempel på nivåer och testmetoder presenteras senare i detta kapitel.

Spårbarhet av krav
Kunder och användare är ofta mest intresserade av att kraven och förväntningarna på systemet är uppfyllda. Med anledning av detta är det mycket viktigt att varje krav testas. För att kunna göra detta måste kraven kunna spåras till testfall, det vill säga varje krav måste finnas representerade i form av något testfall som skall exekveras under testningen.

Testobjekt
Det är viktigt att klargöra exakt vad som skall testas. Detta måste relateras till de nivåer och testmetoder som skall användas. Det räcker inte att ange att ett specifikt system skall testas, utan det måste framgå hur test av olika delar skall göras, hur de olika delarna skall integreras och testas samt hur hela systemet skall testas. Det är dessutom viktigt att ange vad som inte testas. Det utvecklade programvarusystemet kan vara en del av en större helhet, det vill säga den nya programvaran interagerar med andra existerande system. Det måste då klart framgå i vilken utsträckning koppling mellan olika system skall testas.

Tidplanering
Test är ofta en tidsödande aktivitet. Det är många olika testaktiviteter som skall genomföras. Vidare måste test göras om när fel har hittats. I många fall är det dessutom inte tillräckligt att köra om det fall som gjorde att en felyttring blev funnen, utan det är nödvändigt att köra om många testfall då ändringen kan ha påverkat

något annat. Att göra om testning brukar kallas att regressionstesta. Detta beskrivs mer utförligt nedan under testmetoder.

Dokumentation av test
Resultaten av testen måste dokumenteras. I testplanen skall beskrivas hur resultaten från test skall dokumenteras. Vid dokumentation av testresultat måste det framgå vilka indata och under vilka förutsättningar ett test genomfördes samt resultatet av testet. Förutsättningarna kan röra sig om programvarusystemets status då testfallet exekverades. Som ett exempel på hur programvarusystemets status påverkas kan nämnas att det är inte ovanligt att utfallet av ett testfall kan bero på tidigare exekveringar. Ett vanligt exempel är minnesproblem. Om inte dynamiskt allokerat minne, det vill säga minne som behövs under exekveringen, släpps efter användning kommer datorns minne efter en viss tids exekvering inte ha mer minne att tillgå. Detta är ett typ exempel på en felyttring som beror av tidigare exekvering. Det visar också på att ett fel kan yttra sig vid en viss exekvering, men felet beror på något helt annat.

Krav på hårdvara och programvara
Genomförande av test kräver ofta tillgång till testmiljöer. För ett större system kan det vara kostsamma miljöer och därmed är det viktigt att tidigt ange vilka behov som finns i form av hårdvara och programvara, samt när behoven finns, för att kunna genomföra testen.

Begränsningar
Slutligen bör eventuella begränsningar dokumenteras. Det kan röra sig om att ange om några delar inte testas. Andra saker som behöver dokumenteras är om vissa typer av test inte kan genomföras på grund av brist på tillgång till rätt utrustning eller kringliggande andra system. En annan sak som bör dokumenteras är om specifik kompetens saknas för att genomföra vissa testaktiviteter.

Ovanstående punkter är exempel på saker som borde dokumenteras i en testplan. För ett enkelt och litet program utvecklat av en person måste naturligtvis planen anpassas. Ett minimikrav är dock att dokumentera hur test skall genomföras samt utfallet av genomförda test.

När det finns en plan för genomförande av test är det dags att ta fram konkreta testfall. Testfall måste definieras. Det betyder att indata, förutsättningar och förväntad utdata måste bestämmas. Testfall måste i första hand formuleras för att kunna avgöra om de ställda kraven är uppfyllda eller inte. Detta kan låta trivialt, men kan vara nog så svårt då att klara ett krav kan innebära att många delar i ett system, eventuellt tillsammans med kringliggande system, skall fungera på ett visst sätt. Det är till exempel svårt att avgöra prestandakrav då test sker av enskilda delar av ett programvarusystem. Detta är svårt då det ofta är oklart hur krav fördelas på olika delar i ett större system. Samtidigt är det oerhört viktigt att testa delarna i ett system på ett adekvat sätt, då det annars blir väldigt sent i utvecklingen som test mot kraven verkligen kan genomföras.

Testfallen måste formuleras med tydliga indata. Samtidigt måste förutsättningarna för ett testfall dokumenteras. Det kan vara stor skillnad att köra ett testfall när systemet precis har startats och när det har exekverat under en tid som även nämndes ovan när det gäller dokumentation av test. Detta problem blir speciellt stort i ett realtidssystem, det vill säga ett system som mottar indata från ett antal ställen när saker i omgivningen händer. Typiska exempel på realtidssystem är telekommunikationssystem och processtyrsystem i industrin. Svårigheten för denna typ av system är att beroende på olika händelsers ordning kan ett system bete sig olika. Detta bidrar till att göra formuleringen av testfall och deras förutsättningar till en stor utmaning. Även för andra typer av system är formulering av testfall långt ifrån trivialt och många gånger kan det vara väl så svårt att testa ett system som att utveckla ett system. Slutligen måste naturligtvis även förväntad utdata specificeras i ett testfall. Om förväntad utdata sammanfaller med verklig utdata finns en god indikation på att systemet fungerar. Det finns dock en viss risk att både utvecklare och testare har misstolkat ett krav på samma sätt. Det senare är vanligare om utvecklare och testare är samma person. Med anledning av detta har många företag valt att ha en oberoende testorganisation som fokuserar på testplanering och framtagning av testfall medan andra personer gör själva utvecklingsarbetet.

För ett mindre program är det ofta ganska enkelt att avgöra att programmet är testat tillräckligt. Vid utveckling av ett större program-

varusystem är det dock inte självklart när tillräckligt med testning är genomförd för att programvaran skall kunna släppas till användning. Vidare genomgår ett större programvarusystem också ett antal olika tester på olika nivåer, vilket vi återkommer till nedan. De olika nivåerna gör också att beslut måste tas när test på en nivå skall avslutas för att fortsätta på nästa nivå.

Ett vanligt kriterium för att avsluta test är att samtliga testfall exekverats utan fel. Svårigheten i detta fall är dock att kriteriet beror på formuleringen av testfall. Det förutsätter att testfallen täcker behoven. Ett annat, mindre lyckat kriterium, är kopplat till tidplanen, det vill säga test avslutas när ett visst datum har nåtts. Det finns alltid en risk att detta sker då det är viktigt att hålla leveranstider. Problemet för test är också att det ligger sist i utvecklingsprocessen och eventuella förseningar i utvecklingen tidigare riskerar att testfasen kommer i kläm. Betydligt bättre kriterier är om avslutning av test kan ske baserat på mätbara kvalitetskriterier, till exempel att systemet har uppnått en viss tillförlitlighet innan det levereras. Kvalitetsaspekter och mätningar behandlas i Kapitel 13 respektive Kapitel 14.

Test är den största och viktigaste typen av dynamisk verifiering och validering. Det finns dock vissa möjligheter att göra dynamisk verifiering och validering tidigare i utvecklingsprocessen genom användning av prototyper. Nedan följer först en presentation av användning av prototyper innan olika typer av test tas upp.

Anledningen till att testplanering med mera har tagits upp innan användning av prototyper är att mycket av planeringen som genomförs för test även borde göras vid användning av prototyper. Testplanen ovan är mycket lämplig att anamma vid användning av prototyper.

9.2 Prototyper

En prototyp är en initial version av programvaran vars syfte är att demonstrera viktiga aspekter på programvarusystemet. Prototypen skall vara en implementation av en mycket enkel form av det planerade systemet. Det är därmed viktigt att avgöra vilka delar som

skall implementeras i prototypen för att få ut mesta möjliga av den med en relativt liten arbetsinsats. Ett vanligt exempel på när en prototyp kan vara mycket användningsbar är för att demonstrera ett användargränssnitt. I detta fall skulle själva gränssnittet visas upp, men normalt saknas all funktionalitet bakom gränssnittet.

Användningen av en prototyp kan bidra till en bättre förståelse för kraven och möjliga lösningar. Tanken är att en prototyp skall byggas tidigt så att det kan bidra i kravhanteringen. Den skall bidra till kravinsamling, det vill säga identifiering av krav som kanske annars hade upptäckts sent i utvecklingen, och kravvalidering, det vill säga att rätt krav formulerats.

Utöver att prototypen bidrar till förbättring av kravbilden kan den ha andra fördelar:

- Missförstånd mellan utvecklare och användare (olika intressenter) kan identifieras och sedermera undvikas.
- Ett exekverbart, om än mycket begränsat system, finns tillgängligt tidigt för att kunna visa vad som kan uppnås.
- Prototypen kan eventuellt även användas för utbildning av användare.
- Den kan ligga till grund för planering av test.

Det finns undersökningar som visar på fördelen med utveckling av en prototyp tidigt. En undersökning visar på att det ger bättre användbarhet, bättre matchning mot användaren behov, högre kvalitet i designfasen, förbättrat underhåll och färre utvecklingstimmar. Anledningen till dessa resultat är den ökande förståelsen som fås tidigt och som därmed kan genomsyra utvecklingen.

I ordet prototyp ligger att det är något som tas fram men inte används senare. Det gäller till exempel inom bilindustrin, där prototypbilar tas fram. Vissa av dessa leder till industriell produktion medan andra stannar på prototypstadiet. Detta synsätt gäller inte alltid vid utveckling av programvara. Inom programvaruområdet finns två olika synsätt på prototyper. Det första stämmer överens med den gängse bilden, det vill säga en prototyp tas fram för att demonstrera och utvärdera en möjlig lösning. Det andra synsättet är en form av evolutionär användning av prototyper. I detta fall tas en prototyp fram för att visa, men den används även i den fortsatta utvecklingen.

Vid evolutionär användning av prototyper ges en möjlighet att stämma av med olika intressenter (kunder, användare och så vidare) genom att en prototyp som successivt förfinas finns tillgänglig. Det innebär att det går att ha en regelbunden kontakt och kommunikation med intressenterna, vilket i sin tur leder till en trygghet att rätt programvara tas fram.

Användningen av prototyper låter i många stycken tilltalande, men dess användbarhet beror även på typen av system som skall utvecklas. Om programvaran som skall utvecklas är en del i ett mycket stort system kan det vara svårt att kunna demonstrera en prototyp. Å andra sidan kan en tidigare version av ett programvarusystem delvis fungera som en prototyp då den kan underlätta dialogen med användare och kravställare.

Användning av prototyper ger följaktligen vissa möjligheter att göra tidig dynamisk verifiering och validering. Samtidigt är det viktigt att notera att en prototyp inte innehåller komplett funktionalitet och därmed är verifieringen och valideringen som görs i första hand en skattning av slutresultatet. Den slutliga verifieringen och valideringen måste göras när programvaran finns i sin helhet och då är det testning som gäller.

9.3 Testtyper

Det finns två principiellt olika angreppssätt till testning. Det är
- Felletande testning
- Statistisk testning

Dessa båda testtyper är i många stycken komplementära. Felletande testning är det vanligaste och där är målsättningen att hitta fel. Detta innebär att ingen hänsyn tas till den faktiska användningen av programvaran. Grundinställningen är att hitta fel och alla fel som hittas skall tas bort oberoende av deras effekt under exekvering. Tilläggas skall att det sedan inte är alltid som alla fel rättas, utan det görs avvägande om det är värt arbetsinsatsen. Fokus för felletande testning är på korrekthet, som är ett internt attribut på programvaran.

Statistisk testning görs i första hand ur ett användarperspektiv. Den viktiga frågan är hur användaren kommer att uppleva programvaran. Testfallen definieras för att efterlikna driftsituationen. I denna typ av testning är fokus på tillförlitlighet, det vill säga den upplevda kvaliteten sett ur ett felperspektiv. Målsättningen är ofta att skatta tillförlitligheten med hjälp av feldata och olika typer av tillförlitlighetsmodeller. Kvalitet och mätningar behandlas i Kapitel 13 och Kapitel 14.

Ovanstående två principiellt olika typer av test kan användas på olika nivåer. Statistisk testning genomförs oftast på hela systemet även om det skulle gå att använda även för enskilda enheter eller komponenter. Vidare kan olika testmetoder, se Avsnitt 9.5, användas för de olika testtyperna.

9.4 Testnivåer

Olika testnivåer hör samman med olika systemnivåer. Ett programvarusystem består ofta av olika delar, vilka kan utvecklas av olika personer eller grupper av personer. Istället för att sätta samman alla delar omedelbart och testa systemet som helhet brukar test genomföras på olika nivåer. Målsättningen är att försöka säkra kvaliteten på de ingående delarna för att sedan successivt bygga samman ett system. Detta gör att det finns ett antal testnivåer som måste beaktas, vilka är allt viktigare desto större system som utvecklas.

I många fall är det önskvärt att inte samma person eller personer testar som har utvecklat programvaran. Anledningen är att det finns en klar risk att en utvecklare i första hand testar det som faktiskt utvecklats och inte det som skulle ha utvecklats. En utvecklare tror naturligtvis att rätt saker har utvecklats och därmed är risken uppenbar att utvecklaren bara bekräftar sina egna tolkningar av kraven. Detta betyder att en utvecklare mycket väl kan testa avseende verifiering, men det är mycket svårt för en utvecklare att testa med fokus på validering. Många organisationer har oberoende testgrupper för att försöka komma åt detta problem. Det finns dock en sorts test som ofta görs av utvecklaren och det är test på första nivån i systembyggandet, nämligen enhetsnivå.

Nedan beskrivs de olika nivåerna kortfattat och framväxten från enheter till ett fullständigt system illustreras i Figur 9.1. Det kan vara värt att notera att namnen på testnivåerna kan variera mellan olika organisationer, men de grundläggande principerna är desamma.

Enhetstest

Detta är den första nivån. Begreppet enhet används här i generiska termer, vilket betyder att en enhet kan vara till exempel en klass eller någon form av komponent i systemet. Den exakta tolkningen av enhet avgörs i det specifika fallet, det vill säga det beror på den utvecklingsmetod som används med mera.

Målsättningen med enhetstestet är att säkerställa att enheten fungerar som tänkt i designen. Det är i första hand funktionen av en komponent som testas, det vill säga att vissa indata genererar förväntade utdata. Utöver detta testas de interna datastrukturerna, logiken i enheten och även att enheten reagerar korrekt när det gäller gränsvärden. Det senare syftar på indata som ligger på gränser där enheten skall reagera olika beroende på vilken sida gränsen som ett specifikt värde befinner sig.

I många organisationer lämnas enhetstest till utvecklaren, utan större krav på dokumentation av vilka test som genomförts och vilka testfall som exekverats. Detta håller dock på att ändras då allt fler organisationer inser vikten av att hitta fel tidigt, det vill säga så nära introduktionen av felen som någonsin möjligt. Enhetstest är i första hand fokuserad på att utvärdera att koden är korrekt, vilket också förklarar varför den ofta utförs av utvecklaren (eller programmeraren i detta specifika fall).

Integrationstest

Denna typ av test syftar till att säkerställa att ett antal enheter fungerar tillsammans. Det innebär att fokus är dels på gränssnitten mellan olika enheter och dels på funktionaliteten levererad av de integrerade enheterna tillsammans. Integrationstest är den första typ av test som brukar genomföras av speciella testare.

Syftet med integrationstest är att utvärdera gentemot designen. När designen gjordes bestämdes hur olika enheter skulle bidra till systemets funktionalitet och hur enheterna skulle kommunicera. Det är detta som nu testas i integrationstestet.

Funktionstest (system)
Detta är första gången hela programvarusystemet testas. Funktionstestet skall avgöra om systemets funktioner finns på plats och fungerar i enlighet med kravspecifikationen. När testet görs i utvecklingsmiljön är det i första hand frågan om en verifieringen, men kan innehålla inslag av validering beroende på hur kontakten med kunder och användare sker.

Funktionstestet är det tredje steget i V-modellen (se Figur 4.2, där funktionstestet benämns systemtest, vilket egentligen är sammanslagningen av funktionstest och prestandatest), det vill säga det steg som skall fungera som en utvärdering mot kraven.

Prestandatest (test av icke-funktionella krav på systemnivå)
Ett system skall inte bara innehålla rätt funktionalitet, utan även utföra funktionaliteten under vissa bestämda förhållande. Det kan röra sig om svarstider, antal saker som skall kunna hanteras under en viss given tid, systemets tillgänglighet och tillförlitlighet eller någon annan kvalitetsaspekt. För att utvärdera de ickefunktionella kraven genomförs ett prestandatest som även det är en avstämning mot kraven.

Efter detta test har utvecklingsorganisationen förhoppningsvis övertygat sig om att programvarusystemet fungerar i förhållande till ställda krav och kända förväntningar.

Acceptanstest
Om det finns konkreta kunder till programvarusystemet brukar ett acceptanstest genomföras. Det innebär att kunden eller kunderna testar det levererade systemet i sin miljö. Programvaran testas i en labbmiljö eller i alla fall under mycket kontrollerade former, det vill säga acceptanstestet genomförs inte i fullskalig drift. Baserat på utfallet av acceptanstestet tas beslut om nästa steg som kan innebära att antingen skall vissa uppdateringar göras eller så kan programvaran tas i drift.

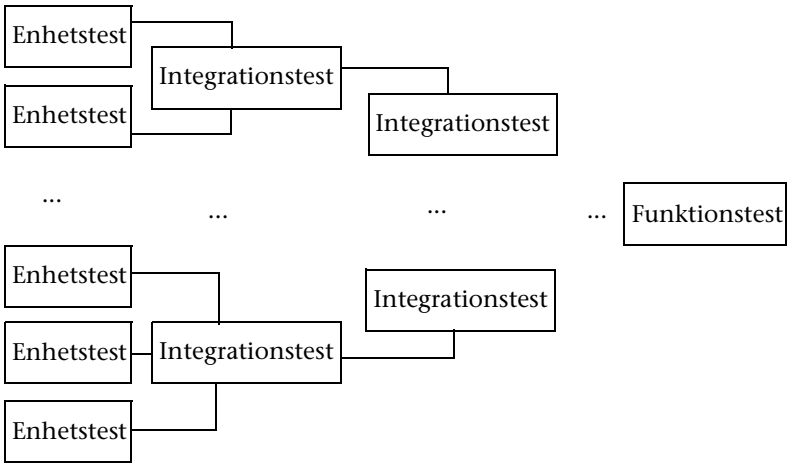
Om det inte finns konkreta kunder utan programvaran i första hand är avsedd för en marknad, så görs programvaran ofta tillgänglig för ett begränsat antal användare för att på så sätt säkerställa att programvaran fungerar tillfredställande. Det senare innebär att användarna är nöjda. Denna typ av test brukar kallas betatest.

Installationstest
När programvaran skall tas i drift genomförs ett installationstest. Det innebär att programvarusystemet testas i den miljö där den verkligen skall användas, det vill säga i rätt omgivning. Det senare inkluderar både rätt hårdvara och annan programvara som finns i anslutning till den nylevererade programvaran.

För att genomföra till exempel enhetstest och/eller integrationstest kan det finnas behov av att ha några hjälpprogram. Det finns två typer av program: testrutiner och teststubbar. När det gäller testrutiner kan det vara frågan om enkla program som kan ersätta någon annan del som skall skicka indata till den del som testas. Testrutinen skall vara en enkel lösning på den andra delen och det enda kravet är att den, ur den testade delens perspektiv, beter sig som den del som skickar indata. Teststubbar är enkla program som kan ta emot utdata från den testade delen och skicka tillbaka ett rimligt svar vid behov. På detta sätt kan en omgivning till den del som testas byggas upp. Målsättningen är att den testade delen skall befinna sig i en omgivning som i möjligaste mån efterliknar den som kommer att finnas när programvarusystemet som helhet finns på plats.

Figur 9.1 illustrerar hur ett antal enheter efter enhetstest sätts tillsammans för att genomgå integrationstest och hur systemet på så sätt växer fram. Detta betyder att delar som genomgått ett integrationstest sedan kan sättas samman med andra delar som också har genomgått integrationstest. På detta sätt sker successiv integration tills hela systemet har byggts upp. När systemet har integrerats sammas genomförs först funktionstest och sedan övriga test i listan ovan. Punkterna i bilden illustrerar att bilden inte är fullständig

utan att andra enhetstest och integrationstest kan behövas innan hela systemet är klart för funktionstest.



Figur 9.1 Successiva test från enhetstest till funktionstest på systemnivå.

9.5 Testmetoder

För genomförande av test på de olika nivåerna finns olika testmetoder att tillgå. Det finns två huvudprinciper för testning. Den första är funktionell testning. Den funktionella testningen bygger på att testobjektet ses som en svart låda, det vill säga alla testfall och observationer av resultat görs i gränssnittet mot testobjektet. Strukturell testning är den andra principen. I detta fall formuleras testfall utgående från kunskap om testobjektets struktur. Det gör att testfall kan formuleras för att säkerställa en viss täckningsgrad, till exempel av vägar genom systemet eller att alla kodrader har exekverats minst en gång.

När det gäller funktionell testning finns det ett antal olika möjligheter. En variant är att studera indata och definiera olika mängder av indata där kravet för en mängd är att programvarusystemet förväntas bete sig likadant. Detta brukar refereras till som ekvivalenstestning, då indata delas in i ett antal mängder där varje mängd av indata förväntar sig ett ekvivalent (likadant) beteende av testobjek-

tet. Tanken är sedan att testfall formuleras för de olika mängderna. Detta gör att testfall inte behöver köras för alla olika indata, utan för ett urval av indata inom den givna mängden. Fördelen är att testfall körs för relevanta urval av indata, samtidigt som inte testfall som inte testar någonting nytt körs i onödan.

En metod som ligger nära ekvivalenstestning är gränsfallstestning. Denna metod kan ses som en blandning av funktionell och strukturell testning. Vid gränsfallstestning är målsättningen att testa alla gränser som finns när det gäller indata, till exempel gränser mellan olika ekvivalenta mängder. Det är också möjligt att komplettera detta med en analys av programvaran för att hitta fler fall då olika saker skall ske i programvaran, även om samma utdata kanske skall genereras.

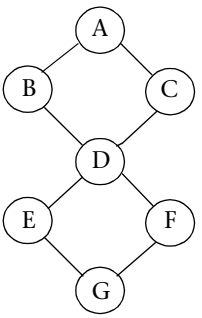
Ett annat sätt att generera testfall ur ett funktionellt perspektiv är att modellera användningen och att ta testfall från den tänkta användningen av programvaran. Detta kan antingen göras genom att testfall väljs för att vara representativ eller att testfall genereras statistiskt ur den troliga användningen. Fördelen med det senare är att det ger en möjlighet att studera programvaran i en situation som efterliknar drift. Vidare ger det genom insamling av feldata också möjlighet att skatta tillförlitligheten, vilket vi återkommer till i Kapitel 14. Det skall tilläggas att det är långt ifrån trivialt att generera testfall på detta sätt.

En tredje form av funktionell testning är gränssnittstestning. I detta fall är målsättningen att utvärdera gränssnittet. Det kan antingen vara ett användargränssnitt eller helt enkelt en komponents gränssnitt. Denna typ av testning är speciellt värdefull vid objektorienterad utveckling för att testa till exempel klasser och då särskilt i samband med återanvändning av objekt eller klasser.

Strukturell testning bygger på att testfall formuleras från kunskap om den faktiska strukturen på koden. Ibland jämförs strukturell testning med en vit låda, då det inre av testobjektet används vid framtagning av testfall. Ett sätt att formulera testfall på detta sätt är att titta på alla exekveringsvägar genom programvaran och att sedan försöka formulera testfall som gör att de olika vägarna testas. Här måste avgöras om all möjliga kombinationer av vägar skall tes-

tas eller om det räcker att varje del av en väg genom programvaran har exekverats. Ett enkelt exempel illustrerar skillnaden i Figur 9.2.

Antag att nod A och D representerar två valmöjligheter i ett program. Två testfall skulle kunna göra att samtliga noder har exekverats. Detta kan göras till exempel genom att ha ett testfall som tar hand om vägen A–B–D–E–G och ett annat testfall som innebär att A–C–D–F–G exekveras. Detta är dock inte alla möjliga vägar genom exemplet. Det totala antalet möjliga vägar i exemplet är fyra stycken, det vill säga även A–B–D–F–G och A–C–D–E–G. Det är följaktligen viktigt att avgöra om två eller fyra testfall behövs här. Det går ganska enkelt att inse att antalet vägar genom ett programvarusystem kan bli mycket stort och därmed är det ofta omöjligt att testa alla möjliga kombinationer. Detta gör sannerligen testning till en stor utmaning. Hur skall test genomföras på bästa möjliga sätt med det minsta antalet testfall?



Figur 9.2 Ett enkelt exempel på vägar genom ett program.

Metoderna ovan är i högsta grad användbara för de flesta typer av programvarusystem, det vill säga oavsett vilken utvecklingsparadigm som följts. Det kan dock vara värt att notera vissa speciella aspekter med objektorientering som påverkar testningen. Detta gör att det i vissa fall talas om objektorienterad testning. Objektorienterad utveckling leder ofta till relativt enkla enheter, men i gengäld erhålls komplexiteten i gränssnitten. Detta gör att enhetstestning blir enklare, men å andra sidan blir integrationstestning svårare. Anledningen till detta är inkapslingen som är en positiv aspekt vid utvecklingen, men som sedan kostar i testfasen. Vidare finns det ett

antal koncept inom objektorientering som ställer speciella krav på test. Som exempel kan nämnas arv och dynamisk bindning. Vid objektorienterad utveckling är det viktigt att hänsyn tas vid formuleringen av testfall till de speciella typer av lösningar som objektorientering gör möjliga.

Slutligen är det viktigt att nämna regressionstestning, det vill säga omtestning. När fel hittas i samband med test leder det till rättningar och uppdatering av programvaran. Detta i sin tur gör att testfall måste köras om. I de flesta fall räcker det inte att köra om det testfall som resulterade i att ett fel hittades, utan helst skall alla testfall köras om. Det skall göras för att säkerställa att rättningen inte har gjort att det blev fel någon annanstans. Det är ofta mycket svårt att inse och därmed förutse alla effekter av en ändring.

För att lyckas med regressionstestning krävs att den automatiseras. Det är naturligtvis önskvärt att automatisera så mycket testning som möjligt, men det är speciellt kritiskt för regressionstestning. Anledningen är att samma testfall kanske behöver köras ett stort antal gånger. Det positiva är att det är relativt enkelt att automatisera regressionstestning. Det som krävs är att exekveringen av lyckade testfall sparas elektroniskt. Vid regressionstest kan sedan testfall köras om och utdata från regressionstestet kan jämföras med första gången testfallet kördes. Om utfallet inte har ändrats har rättningen inte påverkat testfallet. Realtidssystem ger dock vissa problem då händelser i omgivningen kan påverka.

Regressionstestning är oerhört viktigt och det gäller inte bara vid rättningar. Det gäller även att testfall från tidigare versioner av en programvara måste köras igen för att säkerställa att den nya funktionaliteten inte har ändrat på beteendet från det som det tidigare systemet kunde. En användare har mycket svårt att acceptera att en funktion som fungerande tidigare helt plötsligt inte fungerar. Det är mycket lättare att acceptera att det kan vara vissa problem med ny funktionalitet.

9.6 Felklassning

När fel upptäcks i samband med testning, med undantag av enhetstest, brukar felrapporter skrivas. Detta görs av flera anledningar. För det första observeras under test felyttrandet, det vill säga det är inte själva felet som syns. Det gör att det är viktigt att dokumentera observationen, så att själva felet kan identifieras och korrigeras. För det andra rättas ofta felet av någon annan än de som hittade felet. Detta gör att felrapporten blir ett viktigt kommunikationsmedel mellan olika personer i en organisation. För det tredje kan en felrapport användas för att hitta förbättringsmöjligheter. Information om fel kan vara en utmärkt källa för att identifiera saker som behöver förbättras i utvecklingsprocessen.

Ett felyttrande gör att det i första hand är effekten av felet som kan klassas. Många organisationer gör detta genom att klassa felyttrandets allvarlighetsgrad, till exempel i form av kosmetiskt, mindre allvarligt, allvarligt och mycket allvarligt. Ett kosmetiskt fel kan innebära till exempel ett stavfel i en utskrift. Ett mycket allvarligt fel kan innebära att en viktig funktion hos programvaran inte kan användas eller att systemet helt enkelt hänger sig. Den exakta definitionen av allvarlighetsgraderna beror i stor utsträckning på respektive tillämpning och organisation.

I samband med felrättningen, det vill säga när felet som orsakade felyttringen hittats, kan feltypen bedömas. Denna klassning kan innebära att klasser som algoritmiska fel, användargränssnittsfel och tilldelningsfel används. Klassningen sker i detta fall mer kopplat till vari misstaget består. Den ansvarige utvecklaren har uppenbarligen begått ett misstag, vilket är mycket lätt hänt i samband med utveckling av programvara. Det är dock viktigt att få ett grepp om typen av misstag eller missförstånd.

Andra aspekter som är bra att ha med i en klassning, som dock inte är lika vanligt är att peka på var i processen felet troligen introducerades. Om nu utvecklaren har missförstått någonting och därmed introducerat ett fel är det viktigt att förstå när det skedde för att kunna förstå orsaken till felet. I detta sammanhang kan det också vara viktigt att peka på när felet borde ha hittats. Dessa två processklassifikationer ger information om processförbättringsmöjligheter,

dels kopplat till själva utvecklingen och dels kopplat till verifierings- och valideringsaktiviteter.

Sammanfattningsvis är felklassning mycket väsentlig både som kommunikationsmedel och som en källa för att identifiera förbättringsmöjligheter. Fyra klassifikationer rekommenderas som svarar på följande frågor:

- Vad är konsekvensen av felet?
- Vad är det för typ av fel som gjorts?
- När introducerades felet?
- När borde felet ha hittats?

I grund och botten finns det tre saker som felklassningen kan hjälpa till att förbättra. Fokus i förbättringsarbetet skall ligga på:

- Allvarliga fel
  När introducerades dessa och varför upptäcktes de inte tidigare?
- Frekventa fel
  Varför har vi många fel av denna typ?
- Enkla fel (att åtgärda)
  Vad har vi missat som gör att vi gör dessa fel?

Felklassning är ett hjälpmedel. Den ger en bild över problemen som finns. Samtidigt kan felklassning i sig inte hjälpa till att ta bort alla programvarufel. Programvaruutveckling är i stor utsträckning design och därmed kan mänskliga misstag begås. Fel som tillsynes görs slumpmässigt, det vill säga felklassningen visar inte på något mönster är mycket svåra att angripa. Den bästa hjälpen är en god process, bra utbildning av personalen och ett bra stöd åt utvecklarna i form av olika metoder, tekniker och verktyg. Ett exempel på metodstöd är felklassning.

10 Förändring av programvara

10.1 Inledning

Många programvarusystem har blivit för stora och omfattande för att helt ersättas av nya system. Det innebär att kontinuerlig förändring av programvarusystem har blivit mycket viktig. I programvarans barndom var fokus i första hand på underhåll, men under det senaste decenniet har vikten förskjutits från underhåll till vidareutveckling. Allt fler programvaruprodukter kommer ut i nya versioner och releaser. De organisationer som köper programvara har ofta underhållsavtal och de uppdaterar ofta också sina system när nya versioner släpps. I tillägg till underhåll och vidareutveckling av funktionalitet kan det också finnas behov av att antingen underhålla arkitekturen på programvaran eller att förändra arkitekturen.

Ovanstående leder till att det finns fyra huvudtyper av förändring av programvara. De fyra är:

Underhåll
Underhåll brukar innebära rättning av programvarufel, anpassning av programvaran till nya förutsättningar (till exempel ny hårdvara) och tillägg eller modifiering av programvaran.

Vidareutveckling
Vidareutveckling görs då programvarusystemen får en allt längre livstid och genomgår kontinuerlig förändring för att följa med utvecklingen. Vid vidareutveckling avses större och planerade förändringar till skillnad från underhåll. Det finns dock ingen exakt gränsdragning.

Omstrukturering (eng. re-engineering)
Med tanke på alla förändringar som sker är det stor risk att strukturen på programvaran degenererar, vilket leder till ett behov av omstrukturering utan att lägga till funktionalitet.

Arkitekturförändring
Förändringar i tekniken innebär att programvarusystemen måste förändras. Det kan innebära att arkitekturen på programvaran måste anpassas till en annan situation.

Dessa fyra typer beskrivs i lite mer detalj i de följande avsnitten.

10.2 Underhåll

Underhåll innebär tre saker: rättning, anpassning och modifiering. Rättning innebär att fel som inte borde ha funnits i systemet har upptäcks och därmed behöver programvaran rättas. Det är naturligtvis långt ifrån önskvärt att programvarufel inträffar efter det att ett system har satts i drift. Samtidigt är det oerhört svårt att för att större system säkerställa att det är felfritt, speciellt skulle det kosta alldeles för mycket. Detta betyder att för olika typer av programvarusystem görs olika avvägande när det gäller samband mellan olika kvalitetsegenskaper, till exempel korrekthet (fel eller inte fel) kontra kostnaden för ett projekt. Denna typ av avvägande behandlas vidare i Kapitel 13. Kontentan är att fel inte är ovanliga, om än inte önskvärda, och att dessa måste hanteras. Om felet är allvarligt eller inträffar ofta kan det finnas behov av att skicka ut en rättning (eng. patch). Om felet är mindre allvarligt är det möjligt att det är tillräckligt att felet rättas till när nästa version av programvaran släpps. Detta måste avgöras på affärsmässiga grunder.

Anpassningsdelen av underhåll avser mindre uppdateringar på grund av ändringar i omgivning, till exempel uppdatering för att kunna hantera en ny version av operativsystemet. Tekniken och förutsättningarna ändras snabbt inom datorområdet och med anledning av detta kan det finnas behov av att anpassa en programvara till de nya förutsättningarna. I vissa fall görs försök att skriva programvaran så att den skall klara vissa förändringar, men å andra sidan kostar det att göra programvaran flexibel. Det senare gör att en alltför generell lösning inte görs, vilket i sin tur kan leda till anpassningar längre fram. Svårigheten ligger i att förändringarna är ofta okända och det kan kosta mycket pengar att göra en programvara beredd på ändringar som kanske aldrig inträffar. Utveckling av

en flexibel lösning kontra kostnader för anpassningar längre fram är än en gång frågan om beslut på affärsmässiga grunder.

Modifiering avser i första hand de ändringar som blir nödvändiga på grund av att kraven har förändrats. De krav som gällde när programvaran släpptes gäller inte längre, utan nya förutsättningar gäller. Det kan till exempel röra sig om att lagstiftningen har ändrats eller att en standard har ändrats. Större ändringar ingår normalt inte utan bruka hänföras till vidareutveckling som behandlas i nästa avsnitt. Anledningen till denna distinktion är att utgångspunkten inte är att ny funktionalitet skall läggas till. Utgångspunkten är istället att saker i omgivningen har förändrats som gör att programvaran behöver modifieras.

Ovanstående tre typer av underhåll föregås oftast av en analys. I analysen bestäms vad skall ändras och vad effekten av förändringen är. Vid felrättning är det viktigt att en rättning på ett ställe inte ger oönskade sidoeffekter på något annat. När det gäller anpassning och modifieringen är det viktigt att göra påverkansanalys för att avgöra vad förändringen innebär i form av både påverkan på systemet och resursåtgång samt utvecklingstid för att genomföra förändringen. Även vid förändring är det viktigt att överväga sidoeffekter och därmed vad som måste testas efter förändringen. Det skall betonas att efter alla typer av underhåll fyller regressionstest en viktig roll.

Ändringar som går under kategorin underhåll rör sig huvudsakligen om rättningar och mindre förändringar. Nedan följer tre typer av förändringar som är betydligt större.

10.3 Vidareutveckling

Programvarusystemen är idag ofta alltför stora och komplexa för att kunna skrotas och ett helt nytt system utvecklas, även om det naturligtvis händer. Med anledning av detta har vidareutveckling blivit en viktig del av programvaruutvecklingen. Merparten av programvaruutvecklingen idag innebär att existerande system vidareutvecklas. Det är relativt lite utveckling som göra utan att det redan finns ett existerande system.

Ett tydligt exempel är telekommunikationssystemen som inledningsvis hanterade enbart fast telefoni. Utvecklingen har inneburit att systemen har vidareutvecklats för att till exempel kunna hantera analog mobiltelefoni och numera digital mobiltelefoni, datatrafik, Internet och bredbandstjänster. Grundsystemen har inte ersatts fullständigt med ett nytt system, även om merparten av systemen har förändrats. Även om kanske mycket lite finns kvar från det första datorsystemet som togs i bruk för fast telefoni så har aldrig hela systemet tagits bort på en gång. Detta betyder en evolutionär utveckling av systemet. Precis som för underhåll är det viktigt att effekten av förändringen utreds, det vill säga i detta fall av vidareutvecklingen.

Vidareutvecklingen kan dock leda till att strukturen på det existerande systemet efterhand blir sämre och sämre. Detta leder till nästa typ av förändring.

10.4 Omstrukturering

Underhåll och vidareutveckling av ett existerande programvarusystem kan innebära att den initiala strukturen på programvaran, det vill säga arkitekturen försämras eller rentav förstörs. En successiv försämring av arkitekturen innebär att programvaran behöver omstruktureras. Den nuvarande strukturen är helt enkelt inte tillräckligt bra för att klara av fortsatt vidareutveckling och underhåll.

För att hantera problemet med en degenererande arkitektur kan omstrukturering göras. Denna typ av förändring innebär ingen egentlig skillnad i systemfunktionalitet, utan huvudmålsättningen är att få en bättre struktur på programvaran. Detta skiljer denna typ av förändring från de övriga tre som alla har som syfte att någon förändring skall ske.

En omstrukturering påbörjas oftast utgående från det existerande systemet och då i första hand koden. Tyvärr är det alltför vanligt att koden är den enda sanna dokumentationen av programvaran, det vill säga ändringar har gjorts i koden som inte finns dokumenterade i andra dokument. Om utgångspunkten är koden är det brukligt att tala om ”reverse engineering”, det vill säga dokument på högre

nivåer härleds från koden. Det betyder att designdokument och arkitekturbeskrivningar tas fram från koden.

Nästa steg är sedan att strukturera om systemet genom ändring av arkitektur och design innan den nya koden implementeras (eng. re-engineering). Den nya koden blir på detta sätt en implementation av en bättre struktur med förhoppningsvis samma funktionella innehåll. I bästa fall finns testfall för den del som omstrukturerats sparade. Dessa kan användas om de har en extern vy på den del som omstrukturerats ty den nya programvaran skall innehålla samma funktioner som den gamla.

Det stora problemet med omstrukturering är att det ofta ses som en ren kostnad, då programvaran inte innehåller någon ny funktionalitet. Detta gör att det finns ett ganska litet kundvärde i en omstrukturering. Det största värdet är för den organisation som utvecklar programvaran.

10.5 Arkitekturförändring

Mer övergripande arkitekturförändringar kan bli aktuella efterhand som tekniken utvecklas och programvaran skall klara av att leverera sin funktionalitet i en annorlunda miljö. Gränsdragningen mellan underhåll och arkitekturförändring är inte helt entydig, men det är viktigt att inse att det finns olika stora förändringar som måste göras. Arkitekturförändring har särskilts från underhåll då det innebär en utveckling av arkitekturen snarare än en mindre modifiering. Skillnaden inses enklast genom ett exempel.

Många äldre system är utvecklade som centraliserade system där all programvara exekveras på en enda processor. Med förändringar i tekniken och priset för hårdvara är det numera ofta önskvärt att ha distribuerade system, där olika funktioner placeras på olika ställen. Det senare kan vara baserat på till exempel hur användarna vill utnyttja olika funktioner. I telekommunikationsexemplet finns det vissa funktioner som bör ligga nära användaren medan andra kan ligga i en central nod i nätet. Denna förändring ställer krav på uppbyggnaden av programvaran. Kraven innebär ofta ett behov av att förändra programvarans arkitektur.

10.6 Avslutning

Det finns många som hävdar att all programvaruutveckling håller på att övergå i programvaruförändring. Det är sant att andelen nyutvecklad programvara minskar. Samtidigt är metoderna som beskrivit tidigare i denna bok tillämpbara inte bara för nyutveckling utan även för förändring av programvara. Skillnaden mellan nyutveckling och förändring är inte så stor när det kommer till metoder, tekniker och verktyg som bör användas. Den stora skillnaden ligger i att vid förändring krävs att programvaruutvecklingen görs i relation till existerande programvarusystem. Detta innebär i viss mån en begränsning i möjligheterna som inte gör utmaningen med utveckling av programvara mindre. Tvärtom!

11 Verktygsstöd

11.1 Inledning

Verktygsstöd under programvaruutveckling är nödvändigt. I detta kapitel kommer behovet och olika typer av verktyg att tas upp kortfattat. Inget specifikt verktyg kommer att beröras utan fokus kommer att vara på principerna. Verktygsstöd brukar refereras till som CASE (eng. Computer Aided Software Engineering).

I princip kan verktyg stödja det mesta i utvecklingen. All programvaruutveckling har stöd i form av kompilatorer och länkare. Sedan används ofta någon form av editor. Det senare kan i vissa fall vara ordbehandlare, men även grafiska editorer för till exempel design kan användas. Många företag har även stöd för projektplanering och kravhantering. Sammantaget kan sägas att det inte finns många aktiviteter som inte skulle kunna stödjas av verktyg.

Den stora utmaningen är att avgöra vad som skall stödjas och vilka verktyg som skall användas. Det är inte helt oproblematiskt med verktyg då de helst skall vara kompatibla, det vill säga information från ett verktyg skall gärna kunna överföras till ett annat verktyg. Vidare är det viktigt att hitta ett samspel mellan verktygsstöd och de processer och metoder som används. Det bästa är oftast om verktygen stödjer de processer och metoder som används under utvecklingen. I vissa fall kan processer och metoder anpassas till verktygen, men det är högst olyckligt om verktygen styr hur utvecklingen skall genomföras.

När det gäller verktyg finns tre huvudtyper: egen utvecklade, anpassade och generella. Dessa typer beskrivs i nästa avsnitt, där även problemet med kompabilitet mellan verktyg tas upp i lite mer detalj.

11.2 Verktygstyper

Det kan vara frestande att utveckla egna verktyg. Den stora fördelen är att verktygen gör det som behövs i en specifik organisation. Detta skall vägas mot nackdelen att organisationen också har tagit på sig ett fullständigt underhålls- och vidareutvecklingsansvar. Det senare kan bli kostsamt ty all förändring av verktygsstödet måste bekostas av den egna organisationen. Vidare måste det finnas personal som kan arbeta med verktyget. Det är därför mycket viktigt att göra avvägande både tekniska och ekonomiska innan beslut tas om egenutveckling av verktygsstöd.

I den andra änden av spektret finns användning av generella verktyg, till exempel projektplaneringsverktyg eller verktyg kopplade till en specifik notation. Det senare kan vara verktyg som stödjer UML, se Avsnitt 6.2. Fördelen med ett generellt verktyg är att utvecklingskostnaden bärs av den organisation som utvecklar verktyget, vilket i princip innebär att den delas av alla som köper de generella verktygen. Nackdelen är att verktyget ger en viss typ av stöd och det är långt ifrån säkert att detta stöd ligger helt i linje med det stöd som önskas.

En kompromisslösning mellan användning av generella verktyg och egenutvecklade verktyg är kundanpassning av generella verktyg. Detta innebär en större kostnad, men å andra sidan blir verktyget anpassat i förhållande till hur programvaran utvecklas i den egna organisationen. Kundanpassning är ofta förbehållit de stora organisationerna då det troligen är en för dyr lösning för andra.

Ovanstående betyder att det enbart är de stora företagen som kan klara av egenutveckling eller kundanpassning. Mindre företag får antingen köpa licenser till generella verktyg eller klara sig med färre verktyg. I det mindre företaget används ofta enkla ritverktyg och ordbehandlare som stöd under utvecklingen medan de större företagen oftare har tillgång till grafiska editorer.

Oavsett vilka verktyg som används är ett återkommande problem att information och dokument från ett verktyg ofta är svåra att flytta till ett annat verktyg. Detta har vissa verktygsleverantörer löst genom att tillhandahålla en serie av verktyg där information kan föras över från ett verktyg till ett annat.

En annan möjlighet är att i första hand använda verktyg där informationen kan exporteras till något format som kan läsas av andra verktyg. En sådan möjlighet är användning av till exempel XML. Tyvärr är det dock inte ovanligt att verktygsleverantörer vill låsa upp användarna, vilket gör att de ogärna stödjer export till olika format som sedan kan läsas av andra verktyg.

Avslutningsvis kan sägas att verktygsstöd är mycket viktigt i samband med programvaruutveckling. Svårigheten är dock att det finns ett stort antal verktyg tillgängliga och det är långt ifrån enkelt att avgöra vilka verktyg som passar i en specifik situation. Det gör det svårt att välja verktyg som passar den utvecklingsprocess och de metoder som används i en specifik organisation. För att komplicera situationen ytterligare skall verktygen som används gärna passa tillsammans också. Verktygsval är följaktligen svårt om än nödvändigt.

Del 4: Ingenjörsmässighet

Denna del tar upp ett antal aspekter som kopplar till det ingenjörsmässiga genomförandet av programvaruutveckling. Begreppet ingenjörsmässighet används här i bred mening, det vill säga det är primärt inte frågan om att ha en ingenjörsutbildning utan huruvida utvecklingsarbetet bedrivs på ett ingenjörsmässigt sätt. Detta är således mer ett förhållningssätt till utveckling än en fråga om en specifik typ av utbildning. I denna del ingår ett kapitel om kompetenser och egenskaper som underlättar för att kunna arbeta på ett ingenjörsmässigt sätt. Vidare återfinns ett kapitel om olika kvalitetsegenskaper och speciellt avvägningen mellan dem. Det senare är en mycket viktig fråga då ingenjörsmässighet innebär att rätt nivå skall uppnås baserat på de krav och andra förutsättningar som gäller. Det finns även ett kapitel om mätningar som behövs för att kunna hantera och förbättra utvecklingen på ett systematiskt sätt. Slutligen finns det ett kapitel som tar upp kopplingen till olika typer av tillämpningar där programvaran spelar en väsentlig roll.

12 Ingenjörsmässighet

12.1 Inledning

I detta kapitel är inte avsikten att fokusera enbart på den formella ingenjören, det vill säga personer som har gått en ingenjörsutbildning av någon form. Det viktiga är det ingenjörsmässiga beteendet. Ett sådant beteende kräver inte nödvändigtvis en ingenjörsutbildning. Personer med annan utbildning arbetar ofta med samma typ av arbetsuppgifter som den som är utbildad till ingenjör. Vidare arbetar ofta personer med olika utbildning tillsammans i projekt. Det ingenjörsmässiga beteendet är följaktligen viktigt för att fullgöra sina arbetsuppgifter på ett bra sätt. Diskussionen nedan, och även på andra ställen i boken, relaterar därmed i första hand till arbetsrollen snarare än den formella utbildningen när det gäller ingenjörer och ingenjörsmässighet.

Tyvärr är bilden av ingenjören ganska diffus för många människor. Det finns inga tydliga förebilder och det är inte ett yrke som människor har kontakt med i sitt dagliga liv. I allmänhet har de flesta en uppfattning om vad till exempel läkare, tandläkare och affärsbiträden gör, men detsamma gäller inte ingenjören.

I tillägg till att bilden egentligen är diffus finns det även ett antal mer eller mindre felaktiga bilder (stereotyper). Detta gäller inte minst inom programvaruutveckling, där många likställer programvaruingenjörer med programmerare. Dessutom finns det en bild av programmeraren som den ensamma, nästan asociala, datanörden som sitter framför datorn hela dagarna och livnär sig på kaffe och läsk. Det finns naturligtvis personer som motsvarar denna beskrivning, men det är ett fåtal. Vid utveckling av ett större programvarusystem behövs inte bara teknisk kompetens utan även social kom-

petens. Detta leder till förväntningarna på en person som arbetar som ingenjör.

12.2 Förväntningar

Den som arbetar som ingenjör förväntas besitta ett antal kompetenser och egenskaper. Följande kompetenser förväntas:

Teknisk kompetens
Först och främst förväntas en ingenjör att ha teknisk kompetens inom sitt område eller i bredare termer: ämneskompetens. Teknisk kunskap förknippas naturligt med en ingenjör. Det är dessutom den typen av kunskap som utgör större delen av en ingenjörsutbildning.

Social kompetens
Med tanke på att mycket lite ingenjörsarbete görs av ensamma personer utan samverkan med andra finns det starka krav på att ingenjören måste ha en social kompetens. Det krävs samarbetsförmåga, argumentationsförmåga och en allmän öppen attityd till omgivningen. Utan denna typ av kompetens är det svårt att arbeta tillsammans med andra i utvecklingsprojekt.

Administrativ kompetens
Mycket ingenjörsarbete innebär administration. Det kan röra sig om allt från projektadministration till att hålla reda på dokumentationen för systemet eller delar av detsamma. Med tanke på kraven på dokumentation och hantering av densamma är det viktigt med administrativ kompetens. Det betyder att ordningssinne och ett systematiskt och strukturerat arbetssätt är till hjälp för en ingenjör.

Ledningskompetens
Många ingenjörer arbetar i olika ledande roller. Det kan röra sig om att vara ansvarig för en del av ett system eller att vara projektledare på någon nivå eller andra managementfunktioner. Detta betyder att förmåga att leda och motivera andra är en viktig aspekt av ingenjörsarbetet.

Kompetens i muntlig och skriftlig kommunikation
Den sociala kompetensen och även ledningsförmågan hänger nära samman med kompetensen i muntlig och skriftlig kommunikation. Kommunikation är en viktig del av en ingenjörs vardag. Vid utveckling är det mycket som skall dokumenteras och då sätts förmågan till att dokumentera och skriva på ett bra sätt på prov. Vidare är det inte ovanligt att olika typer av presentationer behöver göras för chefer eller kunder. Detta betyder att det även är mycket viktigt med muntlig kommunikation. Dessutom är det inte bara viktigt med muntlig kommunikation vid formella presentationer, utan det är minst lika viktig att kommunikationen med kollegerna fungerar. Det senare är en förutsättning för att utvecklingen av till exempel ett större programvarusystem skall fungera.

Språkkompetens
Många organisationer inom programvaruutveckling har inte bara Sverige som marknad. Det kan finnas kunder runt om i världen. Vidare kan det finnas samarbetspartners eller andra delar av företaget på någon annan plats i världen. Detta gör att språkkunskaper också är viktiga för en ingenjör. Det gäller i första hand engelska, men även andra språk kan behövas ibland. I många större företag med kontor runt om i världen sker dessutom mycket av dokumentationen på engelska, det vill säga det finns ett koncernspråk.

Etisk kompetens
En god ingenjör bör också ha etiskt kompetens. Det är upp till varje ingenjör att värdera sitt arbete ur ett etisk perspektiv. Detta betyder inte att alla personer sätter samma gränser, men det är viktig att en ingenjör har tänkt igenom sitt arbete och vad resultaten från det kan användas till.

Det finns också förväntningar på att en ingenjör skall ha vissa personliga egenskaper. Detta betyder inte att de måste vara medfödda, utan en god ingenjör måste kontinuerligt arbeta med att förbättra sina egenskaper. En ingenjör bör ha följande egenskaper:

Förmåga att ha ett helikopterperspektiv
Det är viktigt att ha ett helhetsperspektiv och en god överblick. Ofta skall en ingenjörsmässig lösning klara av att hantera en situ-

ation. I detta fall är det inte tillräckligt att den egna delen i systemet fungerar bra, utan helheten måste vara bra. En god ingenjör bör ha en förståelse för hur det egna arbetet passar in och bidrar till helheten. Detta gäller dels hur olika programvarudelar tillsammans utgör en programvarulösning och dels hur programvaran kan vara en viktig del i en systemlöning.

Verklighetsförankrad
Ingenjörens uppdrag är oftast att bidra med en lösning till ett problem eller en situation som måste hanteras. Frågeställningen kommer ofta från ett verkligt problem. Det är då viktigt att ingenjören har en god verklighetsförankring och förmåga att se hur olika möjliga lösningar kan bidra till att förbättra situationen eller lösa problemet. Ingenjörer skall i stor utsträckning vara en problemlösare utgående från en given verklighet.

Mål- och resultatinriktad
I rollen som problemlösare ingår att målinriktad arbeta mot en lösning. Det är slutresultatet som är det viktiga. Vidare ingår i målinriktningen även att ingenjören skall eftersträva att hålla sig inom givna tids- och budgetramar. En ingenjör förväntas vara fokuserad på att hitta en bra lösning, vilket inte nödvändigtvis är den optimala lösningen. Det senare var kanske inte möjligt inom de givna ramarna.

Kvalitetsmedveten
En ingenjör skall leverera ett högkvalitativt arbete. Det finns förväntningar på att inte bara en lösning skall levereras, utan att den även uppfyller de uppställda kraven på kvalitet.

Kreativ
Eftersom ingenjören förväntas lösa problem är kreativitet en viktig egenskap. Det gäller att hitta en bra lösning. I många fall finns det kanske ingen tidigare lösning, utan då måste ingenjören komma fram till en bra lösning. Detta kräver ofta att ingenjören är kreativ och tänker i nya banor. Med detta avses inte att ingenjören måste vara en uppfinnare, även om det inte är någon nackdel. Däremot är det viktigt att kunna kombinera och sätta samman tidigare kunskaper på ett nytt sätt.

Flexibel
Denna egenskap är nära kopplad till den sociala kompetensen. Då mycket ingenjörsarbete innebär avvägningar dels mellan olika lösningar och dels mellan olika synpunkter är det viktigt att ingenjören har en flexibel attityd. Det är viktigt att kunna anpassa sig till den givna situationen och inte låsa sig vid någon speciell lösning eller inställning.

Nyfiken på ny kunskap
Ingenjörsarbete innebär utveckling av nya produkter och tjänster. Det innebär ofta framtagning av nya lösningar. Med anledning av detta är det viktigt att ingenjören är nyfiken på ny kunskap. Nya lösningar kan mycket väl kräva nya tekniker eller nya angreppssätt. Detta gör att ingenjören inte får fastna kunskapsmässigt, utan en person som arbetar ingenjörsmässigt måste ständigt söka ny kunskap för att utvecklas i sitt arbete.

Ovanstående kompetenser och egenskaper förväntas leda till att arbetet kan genomföras på ett ingenjörsmässigt och professionellt sätt.

12.3 Arbetet

Ingenjörsarbetet innebär ett antal återkommande aktiviteter. Dessa kan genomföras på olika saker, till exempel på ett projekt eller på en del av ett programvarusystem. Oavsett vilket är de viktiga aktiviteter som också är utmärkande för en ingenjör.

Nedanstående aktiviteter som kan anses utmärkande för ingenjörsarbete är på intet sätt uttömmande. Avsikten är att visa på några olika typer av aktiviteter som ingenjören förväntas arbeta med oavsett sitt ansvarsområde. En ingenjör förväntas:

- Förstå
  Förståelse är viktigt. En ingenjör måste arbeta med att förstå. Detta kan innebära att förstå problemet som skall lösas eller olika lösningsalternativ eller en specifik teknik. Förståelse behövs för att kunna utföra sitt dagliga arbete på bästa möjliga sätt.

- Skatta
  Skattning är en återkommande utmaning för ingenjören. Det kan innebära att skatta tid för att genomföra en specifik aktivitet, till exempel implementation av ett specifikt krav. Det kan också innebära att effekten på systemet, när en ändring skall göras måste skattas. Ingenjören behöver med andra ord kunna göra påverkansanalys. Ingenjörsarbete innebär kontinuerligt avvägande mellan alternativ och för att kunna göra dessa på bästa möjliga sätt krävs en god förmåga att kunna skatta. Skattningarna blir en viktig del i beslutsunderlaget.

- Planera
  Ingenjören skall kunna planera. Det kan innebära allt från sitt eget arbete till ledning av ett större projekt. Detta betyder att ingenjören förväntas kunna bryta ned arbete och problem i mindre delar för att på så sätt kunna planera genomförandet av aktiviteterna.

- Följa upp
  I nära anslutning till planering måste ingenjören också kunna arbeta med uppföljning. Om planer har gjorts upp måste också uppföljning ske. En ingenjör måste kunna göra denna uppföljning och även kunna genomföra omplanering, det vill säga om det visar sig att planen inte håller.

- Utvärdera
  I det ingenjörsmässiga arbetet ingår oftast att välja mellan olika metoder, tekniker och lösningsalternativ. Ingenjören måste kunna sätta upp kriterier för utvärdering och kunna genomföra en sådan för att avgöra vilket alternativ som är bäst. Det kan inte anses förenligt med god ingenjörskunskap att ta första bästa lösning, utan olika alternativ borde jämföras med sina för- och nackdelar innan ett val genomförs.

- Förbättra
  Ingenjörsarbete innebär oftast förbättring i förhållande till en tidigare lösning. På samma sätt som ingenjören eftersträvar att hela tiden hitta bättre lösningar är det viktigt att ingenjörsarbetet också innebär förbättring av det egna arbetet. Kontinuerlig förbättring är en viktig arbetsuppgift för en ingenjör.

13 Kvalitetsegenskaper

13.1 Inledning

Kvalitet är ett mycket allmänt begrepp och det finns de som hävdar att kvalitet är en känsla. Det senare är kanske svårt att hävda med programvara. Det är dock så att efter en viss användning av ett programvarusystem har de flesta bildat sig en uppfattning om huruvida programvaran håller förväntad kvalitet.

När det gäller förväntad kvalitet på ett programvarusystem är det viktigt att notera att det betyder mycket olika saker beroende på typen av system. Det är uppenbart att till exempel säkerhet och korrekt exekvering är mycket viktigare i programvaran till styrsystemet i ett kärnkraftverk och programvara till säkerhetskritiska funktioner hos bilar jämfört med en ordbehandlare eller ett datorspel. Detta betyder att kvalitet är ett relativt begrepp, det vill säga det beror på omgivningen och förutsättningarna. I exemplet är det också klart att kvalitet inte är en sak utan olika saker, det vill säga det finns många olika kvalitetsegenskaper. Ovan nämndes säkerhet (eng. safety), medan i andra sammanhang kan prestanda, till exempel för datorspel där grafiken måste följa med snabbt, och tillgänglighet, till exempel i telekommunikationssystemen, vara viktigast.

Kvalitetsegenskaperna ovan på produkten är i första hand de som kunden eller användaren är fokuserad på. Det finns dock produktegenskaper som i första hand är viktiga för den organisation som utvecklar programvaran. Det gäller aspekter som kopplas till underhåll och vidareutveckling av systemet. I Kapitel 10, diskuterades dessa aspekter och de ställer krav på strukturen i programvaran, det vill säga att det går att arbeta vidare med programvaran. Därmed är det inte bara en fråga om att leverera en produkt som uppfyller kunden eller användarens krav och önskemål, utan programvaran

måste även svara upp mot de interna kraven för den fortsatta utvecklingen.

Ovanstående betyder att olika typer av produkter har fokus på olika kvalitetsegenskaper och dessutom kräver de olika nivå på dem. Det krävs olika säkerhetsnivåer beroende på typen av programvara och det krävs även olika nivåer på underhållbarhet beroende på om den nuvarande programvaran skall ligga till grund för fler releaser eller inte.

Diskussionen ovan relaterar till den upplevda produktkvaliteten sett ur kunden eller användarens perspektiv och utvecklarnas perspektiv, men det finns även ett annat perspektiv. Den organisation som utvecklar programvaran har ofta stort fokus på att programvaran måste levereras vid en viss tidpunkt, till exempel för att hinna ut i rätt tid till marknaden, och till en viss kostnad. Kunderna är med stor säkerhet inte beredda att betala vad som helst för programvaran. Det är också möjligt att det finns ett förutbestämt pris som kunden skall betala. Det gör att utvecklingsorganisationen har ett tydligt fokus även på kostnaden. Slutligen är det också viktigt för organisationen att rätt funktionalitet levereras.

Det finns följaktligen tre kvalitetsperspektiv som måste tillgodoses:

Kunden eller användaren
Denna kategori av intressent är primärt intresserad av produktens beteende. Detta gäller både beteende i form av programvarans funktionalitet och dess icke-funktionella egenskaper, till exempel säkerhet, prestanda och tillgänglighet. Det finns dock även ett intresse av att programvaran levereras vid en viss förutbestämd tidpunkt och till rätt kostnad.

Utvecklarna
Utvecklarna har naturligtvis ett intresse av att produkten är framgångsrik på marknaden, men samtidigt har utvecklarna också andra synpunkter när det gäller kvalitet. De har (eller borde ha) ett intresse av att produkten är välstrukturerad. En god struktur på programvaran underlättar framtida underhåll och vidareutveckling. En annan potentiell aspekt som är viktig för utvecklarna är att produkten innehåller spännande teknik och utmaningar. Detta är inte primärt en kvalitetsaspekt på programvaran

som sådan, men det påverkar definitivt utvecklarnas syn på olika produkter.

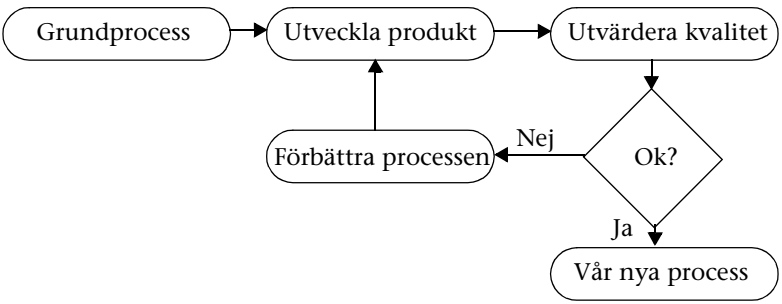
Management
Ledningen har ofta mål som relaterar till tid och kostnader. Det kan vara mycket viktigt att komma ut med programvaran på marknaden vid en viss tidpunkt. Det kan röra sig om att vara först på marknaden med en viss typ av produkt eller att en demoversion måste vara klar till en stor och viktig mässa. Management är också fokuserade på produktens strategiska funktionsinnehåll. Detta betyder att funktioner och tjänster som inte stöds av konkurrenter är speciellt viktiga.

När det gäller kvalitet i programvaruutveckling kan den också delas in i produkt, process och resurs. Produktkvalitet syftar på egenskaper hos produkten. Dessa egenskaper diskuteras i mer detalj i nästa avsnitt, medan kvalitet i relation till processen och resurserna presenteras nedan.

Det är inte bara produktens kvalitet som är viktig, utan även processens kvalitet är viktig. Det finns en ganska stor konsensus kring att processen har en stor påverkan på produktens kvalitet och därmed blir processkvalitet en viktig fråga. Processkvalitet kan betyda många olika saker. Processen skall gå att upprepa, det vill säga processen är väldefinierad så att den går att göra om. Det skall också vara möjligt att relativt enkelt styra projekt, då processen används. Det betyder att utvecklingsprocessen måste stödja styrbarhet. Vidare skall processen vara förutsägbar, det vill säga det skall gå att avgöra i förväg vad som händer. Företrädesvis skall den också stödja förutsägbarhet när det gäller tidsåtgång och resurser, det vill säga både tid i kalendertid och arbetstimmar. Det senare kopplar även till att processen skall vara effektiv. Detta betyder att processen skall producera ett visst resultat på så kort tid som möjligt och med så få arbetstimmar som möjligt i jämförelse med andra alternativa processer. Slutligen skall processen producera samma resultat med samma indata. Det betyder att processen inte skall kunna resultera i olika resultat.

Sambandet mellan processen och produktkvaliteten illustreras i Figur 13.1. Utvärdering och förbättring av processen diskuterades i Kapitel 4. Utöver utvärdering och förbättring av processen är upp-

följning viktig. Det är mycket viktigt att processen kontinuerligt följs upp och att mätningar avseende till exempel ledtid, arbetstimmar, kostnader och programvarufel följs upp genom utvecklingsprocessen. Processens kvalitet kan avgöras genom jämförelser med tidigare genomförda projekt. Om vi antar att avsikten har varit att förbättra processen då är det viktigt att kunna följa upp och se om utfallet efter förändringen också har givit önskat resultat. Detta illustreras i Figur 13.1.



Figur 13.1 Koppling processkvalitet och produktkvalitet.

Slutligen kan kvalitet även hänföras till resurserna, där resurser i första hand är den personal som utvecklar programvaran. Viktiga aspekter som påverkar produktens kvalitet är: utbildning, kompetens, kunskap och motivation. Ett projekt som utvecklar ett programvarusystem måste ha rätt blandning av kompetenser. Det räcker inte att all personal är duktig utan det måste även finnas rätt mix av expertis i projektet. Exakt vad som är rätt expertis beror mycket på den specifika programvaran som utvecklas och även applikationsområdet. Applikationskunskap är mycket viktigt i samband med programvaruutveckling. Det räcker inte att kunna programvaruutveckling, utan det är ofrånkomligt att det även måste finnas mycket god kunskap om själva tillämpningsområdet där programvarusystemet skall användas. Vikten av kunskap inom applikations- eller tillämpningsområdet återkommer vi till kortfattat i Kapitel 15.

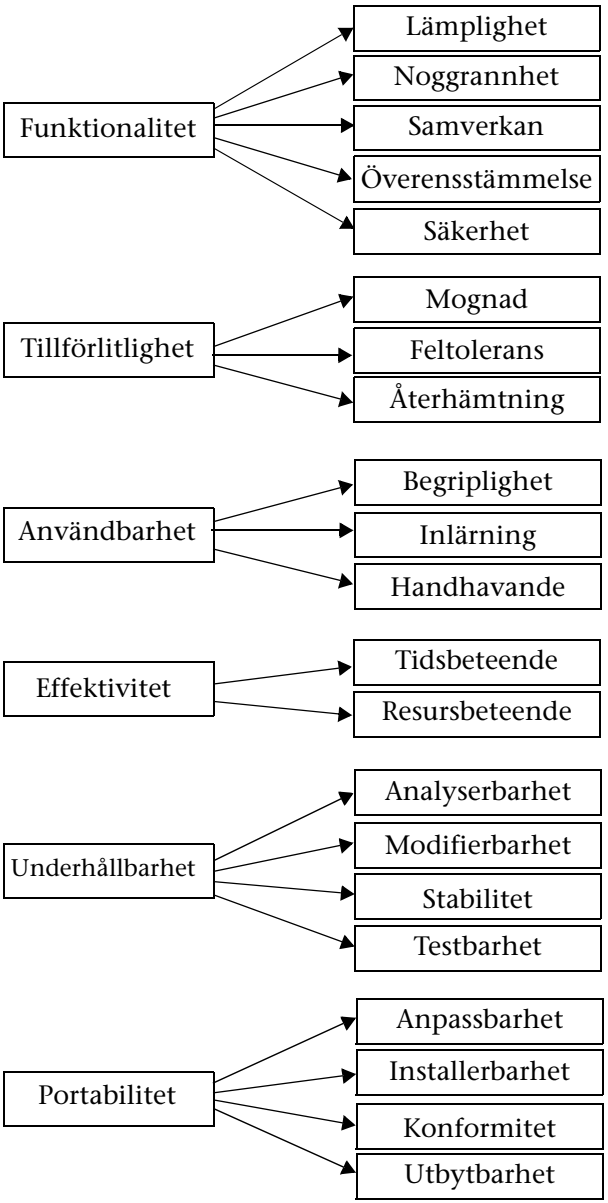
13.2 Produktkvalitet

Produktkvalitet är ett mångfacetterat begrepp som dessutom beror på betraktaren och typen av produkt. För att beskriva produktkvalitet och dess många olika attribut har ett antal kvalitetsmodeller föreslagits. Nedan presenteras en av dessa för att illustrera hur en sådan modell kan se ut samt hur ett antal attribut bygger upp begreppet produktkvalitet.

I Figur 13.2, visas den kvalitetsmodell som är standardiserad av ISO (eng. International Standardization Organization). Begreppen i figuren är fritt översätta från engelska. Modellen är uppbyggd kring sex stycken huvudattribut, vilka i sin tur består av ett antal attribut på lägre nivå. ISO-modellen är en av de enklare genom att attributen på den lägre nivån kopplar enbart till ett attribut på den högre nivån. Andra modeller har inte samma enkla koppling. Att det finns olika modeller visar också på att det inte är enkelt att bestämma exakt vad kvalitet är och hur kvaliteten byggs upp av ett antal attribut.

Den del som ofta ifrågasätts i ISO-modellen är att funktionaliteten är inkluderad som ett attribut. Många föredrar att se kvalitet som en samling av icke-funktionella attribut, det vill säga egenskaper hos funktioner eller programvaran. Distinktionen mellan till exempel funktionella och icke-funktionella krav, se Kapitel 5, blir inte lika tydlig när de funktionella aspekterna tas upp som ett attribut i en kvalitetsmodell. Anledningen är att kvalitet ofta i andra sammanhang ses som något icke-funktionellt.

En kort beskrivning av de olika kvalitetsattributen i modellen återfinns nedan.



Figur 13.2 ISO 9126 kvalitetsmodell.

- Funktionalitet
  Funktionalitet är ett samlingsbegrepp för attribut som fokuserar på existensen av en uppsättning funktioner och deras egenskaper. Funktionerna skall vara de som uppfyller uppställda krav eller förväntningar på programvaran.
  – Lämplighet
  Detta attribut syftar på att rätt funktioner finns och att de tillhandahåller förväntade tjänster.
  – Noggrannhet
  Detta attribut syftar på att funktionerna skall leverera rätt eller förväntade resultat i termer av noggrannhet.
  – Samverkan
  Detta attribut syftar på att funktionerna skall kunna interagera med specificerade system.
  – Överensstämmelse
  Detta attribut syftar på att funktionerna skall uppfylla de standards som finns för tillämpningen eller andra regler och lagar.
  – Säkerhet (i termer av åtkomst)
  Detta attribut syftar på programvarans förmåga att förhindra obehörig tillgång till systemet oavsett om det är avsiktligt eller inte.

- Tillförlitlighet
  Tillförlitlighet är ett samlingsbegrepp för attribut som fokuserar på programvarans förmåga att hålla en given prestandanivå under en given tid och under givna förutsättningar.
  – Mognad
  Detta attribut syftar på frekvensen av fel i programvaran, det vill säga hur ofta det blir fel under exekvering.
  – Feltolerans
  Detta attribut syftar på programvarans förmåga att fortsätta att leverera korrekt funktionalitet även om fel skulle inträffa.
  – Återhämtning
  Detta attribut syftar på programvarans förmåga att återhämta sig efter att fel har inträffat. Attributet inkluderar även tid och arbetsinsats för att uppnå normal servicenivå på programvarusystemet.

- Användbarhet
  Användbarhet är ett samlingsbegrepp för attribut som fokuserar

på det som krävs för att kunna använda programvaran. Detta inkluderar svårighetsgraden att förstå och sätta sig in i programvarans funktionalitet.
– Begriplighet
Detta attribut syftar på den arbetsinsats som behövs av användaren för att förstå programvarusystemet.
– Inlärning
Detta attribut syftar på det arbete som användaren behöver lägga ned för att lära sig använda programvarusystemet.
– Handhavande
Detta attribut syftar på användarens arbetsinsats för att driftsätta och hantera programvarusystemet i drift.

- Effektivitet
  Effektivitet är ett samlingsbegrepp för attribut som fokuserar på programvarans prestanda i termer av exekveringstider och krav på olika resurser.
  – Tidsbeteende
  Detta attribut syftar på exekveringstider, svarstider och genomströmningen av arbete per tidsenhet.
  – Resursbeteende
  Detta attribut syftar på behovet av resurser för att genomföra sina funktioner och effektiv användning av resurserna.

- Underhållbarhet
  Underhållbarhet är ett samlingsbegrepp för attribut som fokuserar på arbetsinsatsen som behövs för att göra förändringar. Begreppet underhållbarhet som kvalitetsattribut i ISO-modellen inkluderar alla aspekter på förändringar som diskuterades i Kapitel 10.
  – Analyserbarhet
  Detta attribut syftar på möjligheterna och arbetsinsatsen för att analysera programvaran i samband med fel.
  – Modifierbarhet
  Detta attribut syftar på det arbete som krävs för att ändra programvaran, rätta fel i programvaran och anpassa den till ändringarna i miljön.
  – Stabilitet
  Detta attribut syftar på programvarans förmåga att hantera oväntade händelser i samband med modifiering.

– Testbarhet
Detta attribut syftar på den arbetsinsats som behövs för att verifiera och validera ändringar i programvaran.

- Portabilitet
  Portabilitet är ett samlingsbegrepp för attribut som fokuserar på programvarans förmåga att kunna flyttas från en miljö till en annan.
  – Anpassbarhet
  Detta attribut syftar på programvarans förmåga att anpassas till nya förutsättningar enbart baserade på de förutsättningar som finns i programvaran.
  – Installerbarhet
  Detta attribut syftar på den arbetsinsats som behövs för att installera programvaran i en specificerad miljö.
  – Konformitet
  Detta attribut syftar på saker som gör att programvaran uppfyller de standards och förväntningar som finns i samband med portabilitet.
  – Utbytbarhet
  Detta attribut syftar på möjligheterna att byta ut en gammal programvara mot en ny programvaran. Detta är ett mått på utbytbarhet hos den nya programvaran, det vill säga dess förmåga att uppfylla de krav och förväntningar som finns på ett redan existerande programvarusystem.

Beskrivningen ovan visar på hur mångfacetterad begreppet programvarukvalitet är. Trots att ett antal attribut beskrivits kortfattat ovan finns det fortfarande några som inte återfinns direkt. De finns med inom ramen för andra attribut, men de finns inte med som egna attribut i ISO-modellen. Detta gäller attribut som tillgänglighet och säkerhet (i termer av personsäkerhet). Detta är inte nödvändigtvis en svaghet i modellen, men det visar på en stor svårighet med programvarukvalitet.

Sammanfattningsvis kan sägas att det är ett svårt begrepp. Det betyder att olika organisationer har sin egen syn på kvalitet och använder kvalitetsattribut som passar dem. De viktigaste kvalitetsattributen är också nära kopplade till den typ av programvara som utvecklas. Om det, till exempel, är programvara till bilar är natur-

ligtvis säkerhetsaspekter (ur personsäkerhetssynvinkel) mycket viktiga. Detta är inte fallet i programvara för telekommunikation, där ofta tillförlitlighet och effektivitet anses viktigare.

Oavsett vilka attribut som är viktiga för olika organisationer och tillämpningar är det viktigt att arbeta med förbättringar av dem. Vidare är avvägning mellan olika attribut en viktig managementfråga. Avvägningen gäller dels mellan olika produktkvalitetsattribut och dels mellan produktkvalitetsattribut och andra kvalitetsaspekter som till exempel ledtid och kostnad. Det är till exempel uppenbart att för datorspel är kostnaden viktig och det innebär att vissa fel kanske måste accepteras. Detta beror på att det helt enkelt skulle bli för dyrt att ta fram felfriprogramvara för denna typ av tillämpning. Å andra sidan om det gäller programvara till ett kärnkraft verk är säkerhet det absolut viktigaste och då har kostnaden en underordnad betydelse.

Resonemanget ovan visar att det inte finns några enkla regler för hantering av programvarukvalitet, utan hanteringen måste anpassas till den rådande situationen och tillämpningen. När det gäller olika kvalitetsattribut är det viktigt att kunna följa upp dem och att kunna förbättra dem. För att kunna göra detta på ett kontrollerat sätt krävs det mätningar. Ett urval av mätningar för några av de mest centrala kvalitetsattributen presenteras i nästa kapitel.

14 Mätningar

14.1 Inledning

Det ingenjörsmässiga förhållningssättet gör mätningar, mått och modeller till viktiga aspekter. Behovet av mätningar kan illustreras med situationen när ett projekt skall starta. För att till exempel kunna avgöra ledtid och arbetstimmar för den förväntade funktionaliteten med en given kvalitetsnivå behövs historiska data. Det är med andra ord viktigt att veta vad som har gjorts innan för att så bra som möjligt kunna skatta viktiga parametrar för det kommande projektet. Tilläggas skall att kvalitetsnivån måste definieras baserat på typen av programvarusystem och icke-funktionella krav i enlighet med diskussionen i föregående kapitel.

Mätningar fyller många olika syften:

- Uppföljning, det vill säga mätningarna skall hjälpa till att avgöra hur långt arbetet har kommit i förhållande till plan.
- Skattning av nuläge, det vill säga det skall vara möjligt att avgöra den nuvarande statusen. Detta är rimligt enkelt när det gäller ledtid (kalendertid) och arbetstimmar, men betydligt svårare när det gäller kvalitetsaspekter på programvaran.
- Förutsägelse av framtiden, det vill säga att kunna avgöra när olika saker sker eller är uppnådda. Detta kan gälla till exempel när den krävda tillförlitlighetsnivån är uppnådd i testfasen.

För att kunna hantera ovanstående behov av mätningar används mått och modeller. Mått är de konkreta sakerna som mäts, medan modeller försöker beskriva samband mellan mått och utfall. Ett exempel är att försöka använda antalet hittade fel i testfasen för att uttala sig om tillförlitligheten i drift. Detta exempel är relaterat till produkten. Ett annat exempel är genomföra omplanering och skatt-

ning av återstående arbetstid baserat på mätningar från tidiga faser. Det kan röra sig om att försöka skatta antalet arbetstimmar som behövs för test givit att ett visst arbete har lagts ned under kravhanteringen, designen och implementationen. Ett sätt att skatta detta är att baserat på historiska data känna till den normala fördelningen av arbetstid mellan olika utvecklingsfaser. Det andra exemplet är ett exempel på mätningar kopplade till processen. Både process- och produktmätningar diskuteras nedan.

Ett återkommande problem med mätningar i industrin är att det är alltför enkelt att definiera olika aspekter att mäta. Detta leder till att det blir för mycket mätningar, vilket gör att data inte samlas in och om de samlas in så används de inte. En följd blir att många programvaruingenjörer får en negativ syn på mätningar och tycker att de utgör en belastning. För att undvika detta är det viktigt att våga göra lagom få mätningar. Detta betyder att varje mätning som görs skall vara:

- Välmotiverad
- Vara på rätt nivå
- Det skall finnas en tydlig användning av mätningen.
- Det skall finnas en intressent som efterlyser mätningen om den inte genomförs.

I anslutning till dessa punkter avseende mätningar skall betonas att de inte bara skall vara uppfyllda utan också tydligt kommunicerade i organisationen. Välmotiverad innebär att mätningen kopplar till någon aspekt som organisationen tycker är högst väsentlig.

Mätningar kan göras på många olika nivåer och det är då viktigt att de genomförs på rätt nivå, där rätt definieras av hur de används. Det senare kan exemplifieras med tidmätningar i form av arbetstimmar. När det gäller arbetstimmar är det viktigt att avgöra om det räcker att räkna arbetstimmar på projektnivå eller om det krävs uppföljning på olika aktiviteter, till exempel, kravhantering, design och så vidare eller om det rentav är viktigt att hålla reda på både aktiviteten och vilken del av systemet som arbetet avser. Det finns inte en modell som passar alla organisationer, utan detta måste avgöras från fall till fall.

Användningen av mätningen är viktig. Om det inte finns tydliga och kommunicerade användningsområden med en mätning är det

inte troligt att den skall genomföras. Varje organisation måste bestämma hur en mätning skall användas. Det är också viktigt att den sedan inte används på fel sätt. Om det är viktigt att hålla reda på hur många fel (och vilka) som en granskare hittar, se Kapitel 8, för att kunna skatta antalet kvarvarande fel då får inte mätningen användas för att betygsätta granskare och sedan koppla detta till lönen. Om det skall finnas sådana kopplingar, vilket inte är troligt, då måste det vara känt och accepterat i organisationen. Missbruk av insamlade mätningar kommer ofelbart att leda till att mätningarna antingen inte rapporteras eller att siffror snyggas till så att de passar. Om till exempel antalet fel som en testare hittar kopplas till lönesättningen, då är det mycket säkert att antalet felrapporter ökar utan att för den skull testningen och sedermera kvaliteten på programvaran har förbättrats. I flertalet fall är det högst olyckligt om mätningar kopplas till personliga prestationer vid åtminstone programvaruutveckling. Intentionen med mätningarna är i första hand för att förbättra projekt, processer och produkter. Om kopplingar dessutom görs till individer då stör detta andra mål.

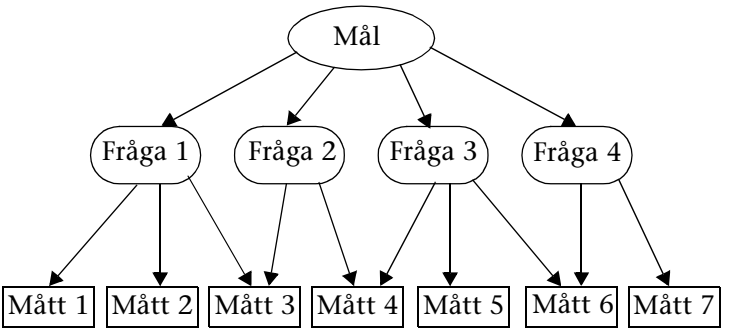
Slutligen skall det också finnas en tydlig intressent eller mottagare av varje mätning. Om det inte finns är det en stor risk att en organisation bygger en datakyrkogård, det vill säga en databas med mätningar som ingen bryr sig. När personalen inser att det inte är någon som bryr sig om en mätning då rapporteras den inte längre. Problemet som uppstår är att när denna insikt kommer infinner sig en allmän misstro mot mätningar som helhet. Detta kan mycket väl leda till att även viktiga mätningar som används inte rapporteras korrekt. Detta gör det oerhört viktigt att definiera vem som är mottagare av en mätning och hur vederbörande kommer att använda mätningen.

För att komma till rätta med ovanstående och definiera rätt mätningar, det vill säga de som verkligen behövs och kommer användas finns en metod som kallas GQM (Goal-Question-Metric). Utgångspunkten är att varje mått och därmed mätning måste ha ett skäl. Metoden har som helhet fyra steg, där det första steget är kopplat till definition av lämpliga mätningar och övriga steg kopplar till genomförandet.

De fyra stegen är kortfattat:

1 Identifiera mål

Detta steg innebär att målen definieras. Frågor som måste besvaras för att avgöra om målen har uppnåtts tas fram. För att besvara frågorna bestäms lämpliga mätningar och tillhörande mått som behövs. Detta steg illustreras generellt i Figur 14.1.



Figur 14.1 Kopplingen mellan mål, frågor och mått i GQM.

I figuren illustreras hur ett mål leder till ett antal frågor. Detta fall leder till fyra frågor. Det åskådliggörs även hur varje fråga kopplas till ett eller flera mått. I figuren kan också ses att ett mått kan vara kopplat till olika frågor. Som ett exempel kan nämnas fråga 3 som kopplar till tre olika mått, där mått 4 också är kopplat till fråga 2 och mått 6 är kopplat till fråga 4. Den hierarkiska uppbyggnaden av GQM gör att det inte finns några mått som inte kopplas till någon fråga och ingen fråga som inte kopplas till ett mål. Detta är en viktig aspekt. Om utgångspunkten är att definiera mått direkt, då är det också stor risk att mått definieras och data samlas in som egentligen inte kopplar till något mål. GQM motverkar följaktligen risken att samla in oväsentlig data som leder till en datakyrkogård.

För att visa på användningen av GQM är ett exempel bäst. Utgångspunkten är kopplad till vad organisationen försöker åstadkomma, det vill säga vad är målet? För att fånga och beskriva målet finns en mall för målformulering.

För det första skall det finnas ett syfte. Syftet byggs upp av tre saker: vad skall göras (till exempel karakterisera, utvärdera, pre-

diktera, motivera och så vidare), med avseende på (till exempel process, aktivitet, produkt, komponent och så vidare) och vad skall åstadkommas (till exempel för att förstå, hantera, förbättra och så vidare). Ett exempel: utvärdera testfasen för att förbättra den.

För det andra kan det finnas olika perspektiv. Detta byggs upp av två delar: vad skall undersökas (till exempel kostnad, effektivitet, korrekthet och så vidare) och ur vems synvinkel (till exempel utvecklarens, kundens, ledningens och så vidare). Fortsättning av exempel: testfasen skall undersökas med avseende på antalet fel utgående från testarens perspektiv.

För det tredje sätts målet i en given miljö, där miljön beskrivs av processen, människorna, problemen, metoderna, verktygen och så vidare. Avslutning av exemplet: testarna tycker att för många fel som hittas sent i testfasen (systemtest) borde hittas tidigare (enhetstest).

Låt oss ta ett konkret exempel på målformulering samt relevanta frågor och mått i anslutning till målet.

Mål: Att förstå vilka faktorer som påverkar projektkostnaden för att förbättra noggrannheten i skattningarna i kravfasen från projektledarens perspektiv på avdelningen X på företaget Y. I denna målformulering kan syfte, perspektiv och miljö identifieras. Syftet är att förstå. Perspektivet är projektledarens som vill förbättra sina skattningar. Miljön är en viss avdelning på ett visst företag.

Ett mål kan leda till en insikt att det inte är rimligt att gå direkt till frågor, utan målet behöver brytas ned i några delmål. Målet ovan kan delas in i följande delmål:

- DM1 (delmål 1): Bestäm nuvarande noggrannhet för projektkostnader.
- DM2: Karakterisera de dokument som ligger till grund för skattningen.
- DM3: Avgör tillförlitligheten i historiska data.
- DM4: Bestäm noggrannheten i den modell som används för närvarande.
- DM5: Identifiera fler parametrar som påverkar skattningen.

Ovanstående fem delmål visar på hur ett övergripande mål leder till ett antal saker som måste adresseras innan huvudmålet kan anses uppnått.

Delmålen leder till frågor. För det första delmålet kan ett antal frågor formuleras. Två exempel:

- Hur bra är noggrannheten på skattningar idag? Denna fråga måste besvaras för att kunna avgöra om någon förbättring sker.
- Är noggrannheten i skattningar lika för utvecklingsprojekt och underhållsprojekt? Detta är viktigt att besvara för att reda ut om det finns några skillnader mellan olika typer av projekt.

Dessa två frågor leder till några konkreta mått. För den första frågan behövs två mått och för den andra frågan behövs ett tilläggsmått. De tre måtten är:

- M1 (mått 1): Skattade projektkostnader idag.
- M2: Verklig projektkostnad idag.
- M3: Karakterisering av projekttyp (utveckling eller underhåll).

Övriga delmål hanteras på samma sätt, det vill säga relevanta frågor och mått bestäms. Det skall noteras att frågor och mått inte ges entydigt från målen, utan olika frågor kan formuleras från mål och frågor kan leda till olika mått. GQM bidrar således inte med någon automatisk formulering av mått, utan metoden skall ses som ett stöd för att kunna härleda relevanta mått och därmed mätningar som behöver göras.

2 Planera mätprocessen
Detta kan innebära till exempel att rapporteringsformulär och verktygsstöd tas fram. Det kan också vara aktuellt med utbildning och information kring mätningarna som skall genomföras.

3 Genomför mätningar
Datainsamling sker. Insamlingen kan antingen ske vid en eller flera givna tidpunkter eller vara en kontinuerlig process. När insamling sker är det viktigt att insamlade data valideras. Detta är speciellt viktigt inledningsvis för att säkerställa att den data som erhålls är korrekt och om data avviker väsentligt från tidigare erfarenheter. Det senare innebär inte att insamlad data nödvändigtvis är inkorrekt, utan den kan vara korrekt men då är det viktigt att förstå förändringen som skett. Speciellt är det viktigt att

förstå de underliggande orsakerna till förändringen i mätdata. Det kan till exempel röra sig om att fördelningen av tid mellan olika aktiviteter har förändrats mycket jämfört med tidigare projekt. Om så är fallet är det väsentligt att förstå varför. Ett skäl kan vara att en ny designmetod har introducerats och då är frågan om förändringen som synes i insamlad tidsåtgång för olika aktiviteterna kan förklaras med den nya designmetoden.

4 Analys och tolkning
Valideringen i punkten ovan är en typ av analys, men målet med den är primärt att säkerställa att insamlad data är pålitlig och att slutsatser kan baseras på den. I detta steg förutsätts att insamlad data (i någon mening) är korrekt. Analysen görs vanligtvis med deskriptiv statistik, det vill säga diagram plottas och enkla mått som medelvärde tas fram. Det kan även förekomma mer omfattande analys i form av statistisk analys. Tolkningen av analysen beror i stor utsträckning på målsättningen och vad mätningarna avser.

GQM kan bidra till att rätt mätningar genomförs i det specifika fallet. För att illustrera några av de vanligaste mätningarna, måtten och modellerna följer nedan en genomgång av några av de viktigaste och vanligaste aspekterna när det gäller mätningar i anslutning till processen och produkten. Det skall dock noteras att det finns väldigt många aspekter som kan mätas. Följaktligen skall mätningarna nedan ses som ett axplock, dock ett viktigt sådant, av möjliga mätningar.

14.2 Processen

I detta avsnitt presenteras i första hand utmaningar i anslutning till mätningar, även om det finns tydliga kopplingar till skattningsmetoderna i Kapitel 3. De absolut viktigaste mätningarna i anslutning till processen är ledtid (kalendertid) och arbetstid. Dessa båda kan tyckas vara trivial att mäta. Tyvärr är det inte alltid fallet.

När det gäller ledtid är det viktigt att definiera när en aktivitet kan anses påbörjad respektive avslutad. I vissa fall sker utredningsarbete av någon enstaka person innan till exempel ett projekt verkligen

formellt startar. Detta gör att starttidpunkten inte är helt enkelt att definiera. Detsamma gäller avslut av olika aktiviteter. Även om en design anses klar kan problem i senare faser göra att designen behöver göras om. Det kan synas uppenbart i ett specifikt fall hur ledtiden skall räknas, men samtidigt är det viktigt att mätningarna från ett projekt kan jämföras med andra projekt. Detta gör att det är viktigt att ha en definition av genomförandet av mätningar som kan hantera olika typer av händelser som inträffar i ett programvaruprojekt samtidigt som siffror mellan projekt blir jämförbara. Detta problem relaterar inte enbart till ledtid utan till alla olika typer av mätningar. Det blir dock extra tydligt när problemet uppstår för en mätning som kan synas tämligen trivial, det vill säga ledtid som enkelt kopplas till kalendern.

För skattning av arbetstid i ett projekt finns ett antal möjligheter som beskrivs kortfattat i Kapitel 3. Både för arbetstid och ledtid är det viktigt att definiera, som nämnts tidigare, hur det skall mätas, det vill säga vad är en aktivitet som skall mätas? En vanlig nivå är att mäta ledtid och arbetstid i anslutning till olika aktiviteter. Detta skapar möjlighet att få en förståelse för tidsfördelningen mellan olika faser och utgående från den kunna arbeta med förbättring.

Ett ständigt problem med ledtid och arbetstid är relationen mellan dem. Det finns alltför ofta en tro att en tioprocentig minskning av arbetstiden också innebär en tioprocentig förbättring av ledtiden. Det är dock inte fallet. I Kapitel 3 introducerades COCOMO som en modell för att skatta arbetstid. I tillägg till den formel som redovisas i Kapitel 3 finns inom ramen för COCOMO även en formel som kopplar samman arbetstiden med ledtiden. Relationen i formeln kompliceras ytterligare av att antalet personer i utvecklingen också påverkar. Om vi inledningsvis antar att antalet personer är konstant och sätter fokus på relationen mellan arbetstid och ledtid. Relationen mellan kalendertid och arbetstid ges av följande formel:

$$TDEV = 3 \times PM^{(0,33+0,2\times(B-(1,01)))}$$

I formeln är *TDEV* kalendertid och *PM* är personmånader. *B* är en multiplikativ faktor som beroende på projektets svårighetsgrad antar värden mellan 1,1 och 1,24. Dessa värden måste ses som riktmärken. Om modellen skall användas i en organisation är det rim-

ligt att börja med dessa värden, men efterhand som erfarenhet erhålls borde varje organisation kalibrera modellens parametrar så att de speglar verkligheten så bra som möjligt.

För att illustrera problemet med relationen, då det gäller förbättring, anta att *B* är 1,2 och att antalet personmånader skattats till 100 personmånader. Detta skulle ge att *TDEV* blir lika med 16,3 månader. Om arbetstiden minskas med 10 % blir *PM* lika med 90 personmånader. Detta minskar dock enbart *TDEV* till 15,7 månader i ledtid. Om förbättringen hade varit 10 % även avseende ledtiden skulle det ha blivit 14,7 månader. Den procentuella förbättringen i ledtiden är därmed väsentligt mindre än minskningen i antalet personmånader.

Även om de exakta parametrarna i en modell som COCOMO inte är universella visar modellen på den normala relationen mellan arbetstid och kalendertid. Det är viktigt att förstå att relationen inte är linjär.

När det gäller genomförandet av till exempel ett projekt motsvarande exemplet ovan bör det också noteras att det oftast inte är en jämn arbetsfördelning. Det betyder att om ledtiden är 16,3 månader och arbetstiden är 100 personmånader betyder inte detta att det är rimligt att 100/16,3 personer arbetar i projektet under hela dess löptid. Ofta startar ett projekt med någon eller några personer. Projektet skalas sedan upp för att i slutet gå ned i antalet personer igen. Efter studier av ett antal projekt har noterats att bemanningen i ett programvaruprojekt närmst liknar en Raleighfördelning. Avsikten här är inte att gå in i de matematiska detaljerna, utan tanken är att beskrivningen ovan åtminstone ger en förståelse för relationerna. När det gäller Raleighfördelningen kan sägas att den till sin grundform har likheter med en normalfördelning.

Det kan också i detta sammanhang vara värt att notera att det ofta är svårt att rädda ett försenat projekt genom att sätta in fler personer. Följden blir oftast att det tar tid att introducera nya personer och därmed sänks effektiviteten hos dem som redan är väl förtrogna med projektet. Det innebär att för programvaruutveckling, som är ett designarbete, går det inte att bara räkna personmånader. Det är stor skillnad mot att gräva ett dike där två personer borde kunna gräva dubbelt så snabbt som en person. Vidare om en person

gräver inledningsvis och en person tillkommer, då går det dubbelt så snabbt. Programvaruutveckling tillhör dock inte denna kategori av arbete.

14.3 Produkten

Det finns ett stort antal olika mått föreslagna i forskningslitteraturen när det gäller programvara. Exakt vilka mått som bör användas i olika situationer måste avgöras med hjälp av GQM och erfarenheter. Det finns ofta mått för olika kvalitetsegenskaper. Svårigheten är dock att många kvalitetsegenskaper är svåra att mäta direkt. Detta gör att mätningar brukar delas in i två typer:

- Direkta mätningar, det vill säga när det faktiskt är möjligt att mäta det som vi vill veta något om. Ett exempel kan vara antal fel hittade i en viss testfas.
- Indirekta mätningar, det vill säga när det inte är möjligt att mäta det som vi är intresserade av. Det är till exempel inte möjligt att mäta tillförlitligheten för en programvaruprodukt under design, men det är kanske möjligt att ha en modell som relaterar antalet kvarvarande fel efter granskning, se Kapitel 8, till tillförlitlighet. Indirekta mätningar karakteriseras av att de är indata till en modell.

Fokus i detta avsnitt kommer att vara på den kvalitetsegenskap som är vanligast att olika organisationer försöker mäta. Tillförlitligheten är av central betydelse för många programvaruprodukter. Den är tydligt relaterad till antalet fel som ofta mäts åtminstone under testfasen via felrapporter från test. För att illustrera användningen av indirekta mått i anslutning till detta tas komplexitetsmätningar upp. Detta innebär sammanfattningsvis att följande tre aspekter kommer att presenteras:

- Komplexitet, det vill säga mätningar på till exempel design eller kod för att försöka fånga svårighetsgraden och därmed felbenägenheten i programvaran.
- Programvarufel, det vill säga de fel som hittas i granskning och under olika typer av test. De fel som hittas under test kan under

vissa omständigheter användas för att skatta tillförlitligheten, vilket presenteras i Kapitel 9.

- Tillförlitlighet som är det externa attributet, det vill säga det som användaren upplever när det gäller korrekt exekvering av programvaran. Programvarutillförlitlighet definieras som sannolikheten för felfri exekvering under en given tid och under givna förutsättningar.

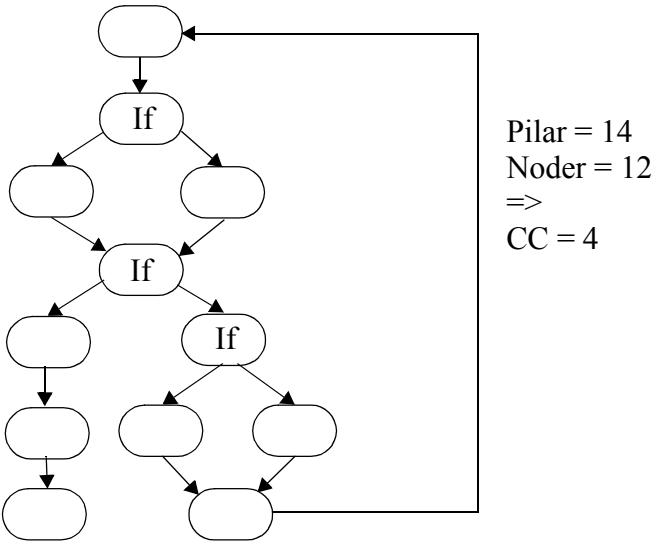
14.3.1 Komplexitetsmätningar

Komplexitet i programvara är ett svårt begrepp. Det är svårt att definiera exakt vad det är och dessutom är det svårt att mäta. Med anledning av detta finns det ett stort antal olika mått som försöker fånga begreppet. Vidare finns det behov av att skilja på till exempel olika typers komplexitet som problemkomplexitet och lösningskomplexitet. Det senare är det som mäts i till exempel design och kod, vilket i och för sig påverkas av problemkomplexiteten. Ett problem kan dock lösas på olika sätt och därmed kan lösningskomplexiteten bli olika även om det är samma problem.

Ett av de äldsta och fortfarande flitigt refererat komplexitetsmått är McCabes cyklomatiska komplexitet (eng. cyclomatic complexity (CC)). Detta mått är lika med antalet oberoende vägar genom ett program. Måttet beräknas enligt nedanstående formel.

$$CC = Pilar - Noder + 2$$

I formeln innebär antalet ”*Pilar*” (eller bågar), det antal övergångar som finns mellan noder i en graf som beskriver programmet. I Figur 14.2 visas en graf av ett enkelt program som illustrerar beräkningen av McCabes cyklomatiska komplexitet.



Figur 14.2 Exempel på graf för att beräkna McCabes cyklomatiska komplexitet.

Det kan också vara värt att notera att för ett program utan goto-satser blir den cyklomatiska komplexiteten lika med antalet if-satser plus ett. McCabes mått har huvudsakligen använts för två syften. För det första är måttet lika med antalet oberoende vägar igenom ett program, därmed visar det också på antalet testfall som behövs för att täcka alla möjliga vägar genom programmet. För det andra har McCabes mått använts som ett komplexitetsmått för att skatta antalet fel i ett visst program eller i en viss komponent. Grundtanken är att ju mer komplext (i termer av McCabes mått) desto fler fel. Om det är möjligt att skatta antal fel i olika delar av ett programvarusystem redan från design eller kod kan det ha stor betydelse för planeringen. Om måttet pekar på att till exempel vissa komponenter innehåller väldigt mycket fel, då det finns det möjlighet att antingen göra en granskning eller är det möjligt att styra testresurserna baserat på denna kunskap.

McCabes mått är ett exempel på mått som direkt mäter antal oberoende vägar igenom ett program. Det kan sedan eventuellt användas

som ett indirekt mått för att skatta antal fel i olika delar i ett programvarusystem.

Det finns ett stort antal andra mått som också försöker fånga komplexitet. Det finns mått på olika nivåer, till exempel mått anpassade för krav respektive kod, och det finns mått för till exempel olika språkparadigmer som presenterades i Kapitel 7. Det finns ett antal mått som är anpassade för till exempel objektorientering. Än en gång är svårigheten att det inte finns något universellt bästa mått, utan val av mått måste göras i den specifika situationen.

14.3.2 Programvarufel

Programvarufel mäts ofta i olika organisationer. Det kan skilja på när fel bokförs, men flertalet organisationer gör det åtminstone under integrations- och systemtest. Huvudanledningen kan vara att test genomförs av en separat testorganisation. Detta innebär att de fel som hittas måste kommuniceras till utvecklarna, då skrivs felrapporter som även diskuterades i Avsnitt 9.6 i anslutning till felklassificering.

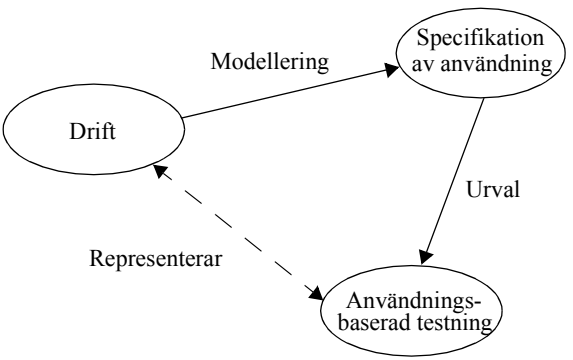
Antalet detekterade fel i olika aktiviteter ger viktig information om effektiviteten i förmågan att hitta fel. Om felklassning används som även pekar på var felet introducerades och var det borde ha hittats, då ger felen även en bra bas för förbättringsarbetet.

Skattningar av kvarvarande fel från granskningar presenterades i Avsnitt 8.3. Dessa skattningar kan ge viktig information för projekt- och kvalitetsledning.

Slutligen finns det en stark koppling mellan antal fel och tillförlitlighet. Detta kopplar till statistisk testning som behandlades i Kapitel 9. Om testning genomförs i form av användningsbaserad testning, det vill säga målsättningen är att efterlikna driftsituationen under i första hand systemtest, då finns det möjlighet att skatta tillförlitligheten som kommer att upplevas under drift. Detta är ett exempel på hur fel kan bli ett indirekt mått på tillförlitlighet.

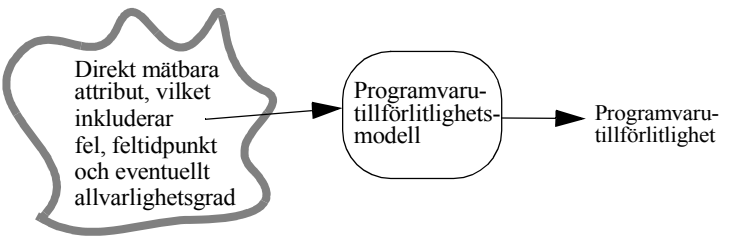
14.3.3 Programvarutillförlitlighet

Den grundläggande idén för att kunna skatta tillförlitligheten från feldata från test illustreras i Figur 14.3. I figuren ses hur driftsituationen modelleras, vilket ger en specifikation av den förväntade användningen. Från denna specifikation eller modell dras testfall som därmed blir ett försök att fånga den verkliga användningen i form av testfall. Dessa testfall driver testningen och tidpunkter för fel noteras noga.



Figur 14.3 Användningsbaserad testning för tillförlitlighetsskattning.

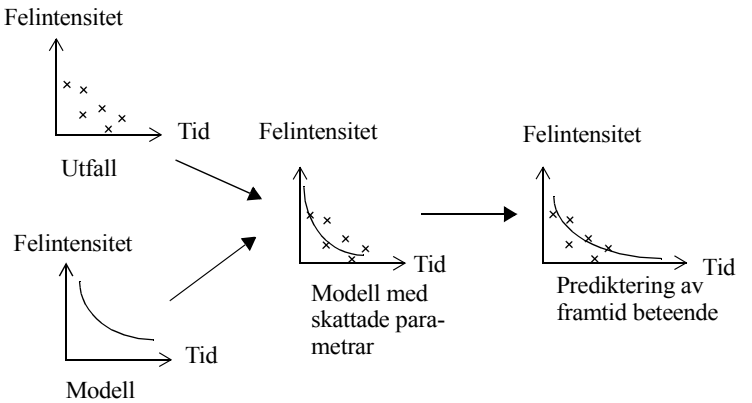
De fel som hittas i testfasen baserat på den användningsbaserade testningen är nu indata till en tillförlitlighetsmodell, vilket illustreras i Figur 14.4.



Figur 14.4 Från mätbara attribut till programvarutillförlitlighet.

Denna typ av modeller baseras ofta på tid mellan felyttringar. Det finns ett antal olika modeller som bygger på olika matematiska modeller av hur programvarufel hittas. I många fall är det svårt att

mäta tillförlitligheten i termer av en sannolikhet direkt. Detta har gjort att det är vanligt att beskriva tillförlitligheten i termer av tid mellan fel (eng. Mean Time Between Failures (MTBF)) eller felintensitet. Det senare är helt enkelt 1/MTBF. Ett exempel visas i Figur 14.5. I figuren visas ett antal saker som diskuteras nedan.



Figur 14.5 Programvarutillförlitlighetsmodellering.

Till vänster i Figur 14.5 visas för det första felintensiteten som har erhållits från feldata. I den nedre vänstra delen visas exempel på en modell som antager att felintensiteten följer en specifik kurva över tiden. Verkligheten är sällan så exakt och avsikten med modellen är inte att exakt modellera utfallet utan att skapa en rimlig bild av utfallet. I mitten i figuren kan ses hur modellens parametrar nu har skattats för att modellen på bästa möjliga sätt skall kunna fånga det verkliga utfallet. Många modeller är uppbyggda kring exponentialfördelningen, då antalet fel förväntas avta med tiden och därmed förväntas tiden mellan fel att öka, vilket följaktligen betyder en minskning i felintensiteten. Vidare innehåller modellerna ofta 2–3 parametrar som skall skattas baserat på det verkliga utfallet. Det rör sig ofta om kurvanpassning, det vill säga hur kan en kurva (givit en viss fördelning) bäst anpassas till en datamängd? Längst till höger i figuren visas hur modellen kan användas för att prediktera framtiden. Det går att besvara frågor som när en viss felintensitetsnivå förväntas uppnås. Detta kan vara mycket viktig information för att avgöra om eventuella tillförlitlighetskrav har uppnåtts vid planerad leveranstidpunkt. Om så inte är fallet då kan det finnas anledning

att antingen försöka ändra i testningen eller att skjuta på leveranstidpunkten.

14.3.4 Avslutning

Avsikten här är inte att gå in på några detaljer när det gäller tillförlitlighetsmodellering, utan det viktigaste är att peka på hur olika typer av mätningar kan ge viktig information för programvaruutveckling. Vidare är det viktigt att betona att tillförlitlighet kan skattas under utvecklingen, det vill säga tillförlitlighetsaspekten kan adresseras på ett ingenjörsmässigt sätt. Fel är en av de viktigare aspekterna att mäta under programvaruutveckling. En anledning till detta är att informationen dels kan ligga till grund för förbättringsarbete och dels kan användas för att skatta ett viktigt kvalitetsattribut i drift, det vill säga tillförlitligheten.

14.4 Sammanfattning

Sammanfattningsvis var avsikten med detta kapitel att visa på att mätningar kan vara ett viktigt hjälpmedel för planering, styrning, utvärdering och förbättring av programvaran. Detta gäller mått både för processen och för produkten, där produkten oftast bedöms som svårast att mäta på. Några exempel på aspekter som ofta mäts har tagits upp i detta kapitel. Det kan dock finnas andra viktiga aspekter och för att avgöra exakt vilka mätningar som bör genomföras finns GQM att tillgå.

15 Tillämpningsområden

15.1 Inledning

Programvara har snabbt blivit en central del av alltfler produkter. Det finns tre huvudtyper av produkter med programvara. Den första är ”rena” programvaruprodukter, till exempel spel och ordbehandlingssystem, det vill säga där det upplevda värdet utgörs av programvara. Det andra är system där många tjänster är realiserade i programvara, till exempel mobiltelefoner och banksystem. Den tredje typen är system som innehåller programvara, utan att användaren är direkt medveten om det. Det senare inkluderar till exempel olika typer av transportsystem (bilar, flyg med flera) och robotar. Detta innebär att programvara har blivit en central del i många produkter och i den infrastruktur som utgörs av datorer och Internet idag. Samhällets allt större beroende av datorer och programvara gör att utbildning och forskning är högst väsentlig för att kunna hantera, utveckla och förbättra de komplexa programvarusystem som idag utgör en väsentlig del av vår vardag idag. Kunskap och kompetens inom området är mycket väsentlig för ett hållbart samhälle, det vill säga där det finns tilltro till programvarusystemen och dess tjänster.

Vi har följaktligen sett en utveckling mot att programvara finns överallt. Detta återkommer vi till i avslutningskapitlet, det vill säga Kapitel 16. Som exempel på programvarans ständiga närvaro tas nedan upp ett antal exempel på systemtyper och olika typer av tillämpningar i anslutning till dem. Avsikten är inte att nedanstående skall vara en uttömmande lista av tillämpningar, utan tanken är att de skall illustrera att det finns behov av kunskap och kompetens om programvara inom ett antal branscher. En annan aspekt på detta är att visa på behovet av att inte enbart ha kunskap och kompetens inom programvaruområdet, utan att tillämpningskunskap är

mycket viktig för att kunna utveckla bra programvarusystem. Ett verkligt exempel på när detta misslyckades presenteras sist i kapitlet.

Det finns ett antal olika kategorier av system där programvara är en central del. Det skall dock betonas att vissa typer av programvarusystem kan hamna i mer än en kategori. Det gör inget, då avsikten inte är att skapa en indelning i olika kategorier. Målsättningen är att ge en förståelse för att kunskap och kompetens inom tillämpningsområdena kan vara mycket viktiga och i vissa fall innebär det att det behövs kunskap och kompetens av olika karaktär. Det är detta som återspeglas genom att några typer av programvara återfinns i mer än en kategori nedan. Följande skulle kunna vara en indelning av programvarusystem för att peka på behovet av olika typer av tillämpningskunskaper:

Programvara
Denna kategori innehåller de programvaror där kunden verkligen medvetet köper programvara. De vanligaste exemplen är datorspel, ordbehandlare och andra program som körs direkt på en dator. Detta inkluderar också de program som kan laddas ned från Internet. Utvecklare av denna typ av system behöver god kunskap om operativsystem och drivrutiner som används på vanliga datorer i hemmet eller på arbetet.

Internet och webben
Det finns mycket programvara kopplade till webben. I dessa fall är det oftast programvaror som tillhandahåller tjänster i någon form. Det kan röra sig om banktjänster via Internetbanker, bokning av resor och hotell, e-handel (till exempel bokhandel), uppslagsböcker och annan typ av informativa tjänster samt deklarationer via nätet. För denna kategori av programvaror krävs det god kunskap om den aktuella tillämpningen, till exempel skattelagstiftningen om det gäller deklarationer. För de andra tillämpningar behövs också specialkunskap. I tillägg till detta behöver den som utvecklar programvara för dessa olika typer av nättjänster god kunskap om Internet och tekniska lösningar i relation till detsamma. Denna kategori av programvara har likheter med andra kategorier nedan, även om de har sina egna karakteristika

också. Det är till exempel uppenbart att systemet som hanterar deklarationer via nätet också passar i nästa kategori.

Informationssystem
Dessa programvarusystem rör ofta administrativa system. Det kan gälla olika typer av ekonomisystem och resursplaneringssystem. Ett exempel på resursplaneringssystem är till exempel olika typer av schemaläggning till exempel av tågpersonal. Hur skall personalen arbeta på tågen för att efter en arbetsdag komma hem igen? Är det överhuvudtaget möjligt? Programvarusystem i denna kategori kan i vissa fall ha tidskrav och beroende på hur kritiska dessa tidskrav är kan eventuellt olika informationssystem också klassas i nästa kategori.

Realtidssystem
En vanlig typ av tekniska system är realtidssystem. De karakteriseras av att händelser sker från omgivningen, som kan vara personer eller andra system, och att realtidsystemet förväntas hantera dessa händelser. Detta betyder att belastningen på systemen förändras över tiden, det vill säga det kan finnas tidpunkter då väldigt mycket sker och andra tidpunkter då det är ganska lugnt. Exempel på system i denna kategori är telekommunikationssystem, där belastningen styrs av abonnenternas beteende och vanor och processtyrsystem i industrin. I det senare fallet kan det röra sig om processövervakning och styrning av till exempel en produktionsprocess. Om det i detta fall kommer in något larm är det viktigt att systemet reagerar snabbt. Det finns många andra exempel på realtidssystem, till exempel inom transportområdet. Dessa tillhör också kategorin inbyggda system och tas upp i nästa kategori.

Inbäddade system
Många realtidssystem är också inbäddade system. Gränsdragningen är inte helt entydig och många tillämpningar kan klassificeras som varande av båda typerna. Vi har här delat på dem för att de har olika egenskaper. I realtidssystem är det tiden som är den kritiska faktorn. Inbäddade system å sin sida karakteriseras av att programvaran finns inuti ett system eller produkt och användaren ser inte systemet primärt som ett programvarusystem. Många tekniska system är av denna kategori. Speciellt kan näm-

nas transportmedel, där till exempel bilar får en allt större andel av sin funktionalitet implementerad av datorer och tillhörande programvara. Ett annat område är medicinsk utrustning. Även telekommunikationsområdet består i stor utsträckning av denna kategori av system. Slutligen kan även robotar vara värt att nämna, där logiken för vad robotarna skall göra är implementerad i programvara.

Distribuerade system
Många både realtidssystem och inbäddade system är distribuerade system. De tas dock upp separat, då det ställer nya krav på programvaran om den skall kunna fungera i en distribuerad miljö. Det gör att kommunikationen måste fungera annorlunda än om hela programvarusystemet hade funnits på en enda dator.

Säkerhetskritiska system
En sista kategori som är värd att nämna är säkerhetskritiska system. Dessa system tillhör med stor säkerhet en eller flera av ovanstående kategorier också. Det som gör det viktigt att lyfta fram detta som en speciell kategori är att det behövs speciell kunskap och kompetens för att hantera säkerhetskritiska tillämpningar. Kraven på formella metoder och en mycket hög kvalitetsmedvetenhet är väsentliga karakteristika för denna typ av system. Säkerhetskritiska system återfinns till exempel inom transportområdet samt inom vissa sektorer av energiområdet, till exempel styrsystemen för ett kärnkraftverk.

Ovanstående kategorier visar tydligt hur kunskap och kompetens inom programvaruområdet måste kombineras med kunskap och kompetens inom tillämpningen och dess speciella egenskaper.

Som ett exempel på när det blev problem på grund av bristande förståelse för en viss tillämpning presenteras här ett exempel från min egen erfarenhet. Vi kom in i detta projekt som konsulter för att försöka komma med förslag på hur nedanstående situation skulle lösas. Förutsättningarna var följande när vi kom in i projektet. En stor kund hade beställt ett nytt programvarusystem. Det var ett stort system och kravspecifikationen var ett tjockt dokument. Bland de icke-funktionella kraven stod att läsa att programvarusystemet skulle vara användarvänligt. Detta är naturligtvis inget bra krav. Vad betyder det egentligen? Problemet var att kravställaren hade en

bild av vad detta innebar och det företag som skulle utveckla programvaran gjorde sin egen tolkning.

Följden blev att utvecklarna tog fram ett menybaserat användargränssnitt. Det var i grund och botten helt i linje med vad kunden ville ha, men två saker missades. För det första förstod inte utvecklarna hur arbetet med systemet egentligen bedrevs. De hade med andra ord ingen uppfattning om i vilken ordning olika funktioner normalt gjordes. Detta var självklart för beställaren som visste att det fanns tydliga kopplingar mellan olika funktioner. Användaren visste att om Funktion A användes så var det mycket ofta att detta följdes av Funktion B. För det andra antog utvecklarna att användarvänligt alltid innebar användning av menyerna. Kunden hade däremot personal som skulle arbeta mycket intensivt med detta system och i sådana fall kan användarvänlighet mycket väl innebära att det skall finnas kortkommando. Det senare önskades som komplement till menyerna. Denna typ av ”dubbelkommando” har vi idag i många programvaror, till exempel ordbehandlare. I detta fall har till exempel många lärt sig att ”Ctrl C” innebär ”kopiera” och så vidare.

Kunden hade följaktligen uppfattningen att funktionerna skulle vara placerade på ett visst sätt i menyerna för att motsvara den vanliga användningen av funktionerna. Vidare hade kunden förutsatt att utvecklarna förstod att kortkommandon behövdes. Utvecklarna å sin sida tyckte att kravet på användarvänlighet var tillgodosett och att kunden varit otydlig i sina krav.

Detta är ett typexempel på där bristande kunskap hos utvecklarna gör att kundens förväntningar inte blir uppfyllda. Kunden har dock varit för dålig på att kommunicera sina krav. Det innebär bland annat att kunden inte förstått att kravet kanske kunde tolkas annorlunda. Å andra sidan borde nog utvecklarna ha frågat kunden om tolkningen av kravet på användarvänlighet.

Avslutningsvis kan nämnas att vårt bidrag till situationen var att genomlysa projektet, både på kundens och utvecklarnas sida, och utgående från detta föreslogs en kompromisslösning. Detta var det enda rimliga då det gick att påvisa att båda parter kunde ha agerat bättre under programvarusystemets utveckling. Kontentan blev att

parterna fick dela på kostnaderna för att korrigera programvaran så att den motsvarande både kraven och icke-uttalade förväntningar.

Sammanfattningsvis är det oerhört viktigt att kunskap och kompetens i programvaruutveckling kombineras med god motsvarande förmåga inom tillämpningsområdet för vilken programvaran utvecklas. En brist på förståelse för omgivningarna och förutsättningarna i olika tillämpningsområden kan få ödesdigra konsekvenser. Ovan visades på ett fall med ekonomiska konsekvenser, men det kan vara mycket allvarligare än så. Det finns många programvarusystem, där problem kan innebära fara för människoliv.

Del 5: Avslutning

Den sista delen innehåller enbart ett kapitel. Detta kapitel sammanfattar mycket kortfattat varför programvaruutveckling är viktigt. Kapitlet avslutas med en diskussion kring åtta trender som har identifierats och som i stor utsträckning kommer att påverka hur programvarusystem kommer att utvecklas och byggas i framtiden.

16 Sammanfattning

16.1 Inledning

Baserat på utvecklingen av IT-samhället och det beroende som därmed har uppstått avseende datorer och programvara är det enkelt att konstatera att kunskap och kompetens inom programvaruutveckling kommer att vara viktiga för en lång tid framöver. Programvarusystem har blivit allt större och har en allt längre livslängd. Detta gör att programvaruprojekt som arbetar med vidareutveckling av programvarusystem blir allt vanligare och viktigare. Det i sin tur leder till utmaningen att inte bara klara av utveckling utan att klara av den i anslutning till ett existerande system.

Ovanstående gör att kunskap kring programvaruprojekt, arkitekturer, processer och de olika aktiviteter som bygger upp utvecklingsprocessen är mycket väsentliga. Vid kontakter med industrin idag betonas ofta områden som kravhantering, test, arkitekturutveckling och processförbättring som viktiga områden att arbeta med. Det är ofta inte programmeringen som uppleva vara den svaga punkten i dagens utveckling, även om den säkert också kan förbättras.

Detta är kortfattat vad som är viktigt idag, men en fråga är naturligtvis vad som händer i framtiden. Vad är det för trender som finns? Vad är de troliga effekterna av dessa trender?

16.2 Framtiden

Följande åtta trender är resultatet av diskussioner som förts inom ett internationellt konsortium. Konsortiets målsättning har varit att identifiera viktiga forskningsområden i anslutning till programvaruprocessen. En förutsättning för att kunna göra det är att först

ha en förståelse för den troliga utvecklingen inom området som helhet. Nedanstående trender identifierades som tämligen säkra, även om den exakta effekten är mycket svårare att förutse. Det finns naturligtvis sedan även saker som kan hända, men där utvecklingen är mer osäker. Det trender som bedömdes som tämligen säkra var:

Korta ledtider
Det kommer att vara ett fortsatt tryck på korta ledtider. Att träffa marknadsfönstret blir bara viktigare och viktigare. Detta kommer också att innebära krav på ökad produktivitet inom programvaruutveckling. Samtidigt som det kommer att vara ett stort fokus på ledtid och kostnad kommer kunderna och marknaderna att bli allt mer kvalitetsmedvetna. Detta gäller speciellt kvalitetsattribut kopplade till säkerhet (både ur synvinkeln kritiska system och ur sårbarhetssynvinkel) och personlig integritet.

Systemstorleken ökar
Systemen kommer att öka i storlek. Vidareutveckling kommer att dominera över nyutveckling. Det blir allt vanligare att system byggs av andra system. Andelen komponenter och därmed sammansättning av system från delar kommer att öka.

Ett trådlöst samhälle med sensorer
Den trådlösa trenden kommer att fortsätta. Den kommer att sprida sig till enheter som kommunicerar autonomt, det vill säga kommunikationen är inte ett resultat av att en människa initierar händelseförloppet. Denna trend är nära kopplad till att vi kommer att se allt fler mindre datorer som till exempel sensorer som kommunicerar med varandra och med system. Det kan röra sig om övervakningssystem för sjuka människor, där sensorer kan göra mätningar som sedan skickas vidare för analys. Det kan komma system som till exempel när man lämnar arbetet noterar att du är på väg hem och i samband med det skickar över din inköpslista till affären. Personalen i affären plockar samman dina varor och levererar dem till dig lagom till du kommer hem. Det finns ett antal framtidscenarier av denna typ som är teknisk möjliga, sedan är det en fråga om vad vi i grund och botten vill.

Ökad globalisering
Det kommer att ske en ökad globalisering. Denna trend kan potentiellt hindras av större terroristdåd som i sin tur leder till en

ökad protektionism. Grundtrenden är dock att globalisering kommer att öka för att bättra utnyttja den kunskap och kompetens som finns runt om i världen och för att utveckling kommer att ske där det är mest ekonomiskt lönsamt. En möjlig utveckling är att programmering och sammansättning av programvarusystem kommer att ske på olika ställen. Samtidigt kommer vi att se alltfler utvecklingsprojekt som spänner över flera orter runt om i världen. Ett möjligt scenario är att det sker försök att utföra utveckling 24 timmar om dygnet. Detta kan ske till exempel genom att tre olika orter, strategiskt placerade i världen, kommer att arbeta tillsammans för att åstadkomma 24 timmars utveckling.

Värdebaserad utveckling
Kunderna kommer att bli bättre kravställare och värdet på funktionaliteten och kvaliteten som levereras kommer att bli allt viktigare. Det räcker inte med ny teknisk funktionalitet, utan programvaruprodukterna måste medföra ett tydligt mervärde för slutanvändaren.

Programvaran alltmer kritisk
Programvaran är redan idag kritisk i många system och utvecklingen kommer att fortsätta i denna riktning. Detta kommer speciellt att innebära ett starkt fokus på olika kvalitetsaspekter. De mest uppenbara är de aspekter som kopplar till säkerhet och systemförtroende. Efterhand som användaren mognad ökar, se föregående punkt, kommer kraven på ökad funktionalitet att kombineras med starka krav på säkerhet och personlig integritet.

Snabb teknisk förändring
Det kommer att ske en fortsatt snabb teknisk förändring, vilket bland annat påverkar behovet av vidareutveckling och underhåll av system. Samtidigt kommer de nya teknikerna, till exempel i anslutning till sensorteknik att ge helt nya möjligheter och därmed även hot. Det måste därmed finnas en beredskap för en fortsatt teknisk utveckling.

Interoperabilitet
Denna punkt hänger samman med ett par av ovanstående punkter, men förtjänar att separeras då det kommer att bli en mycket viktig aspekt i framtiden. Kraven på att system skall kunna kom-

municera med varandra kommer att öka. Detta ställer stora krav på interoperabiliteten mellan olika system och speciellt olika dataformat. Idag finns, till exempel, många olika system inom sjukvården som inte kan kommunicera med varandra. System är ofta inköpta oberoende av varandra på någon nivå i en organisation och det har saknats en god förståelse för att systemen på sikt borde kunna utbyta information. Detta kan gälla mellan olika avdelningar på ett sjukhus och naturligtvis även mellan olika sjukvårdsområden. Sjukvården är naturligtvis inte det enda området med detta behov. Därmed kommer interoperabilitet att bli en allt viktigare kvalitetsaspekt efterhand som systemen blir allt fler. En viktig utmaning är att kunna koppla samman flera system samtidigt som det inte får påverka till exempel personlig integritet negativt.

Ovanstående visar på några av de trender som kommer att påverka oss i hög grad i framtiden. Påverkan kommer att ske på oss både som medborgare och som utvecklare av programvara. Ovanstående trender och dess följder kommer att ställa stora krav på programvaruutvecklande organisationer. Samtidigt som det ger ett stort antal intressanta utmaningar att arbeta med i framtiden.

Om författaren

Dr. Claes Wohlin är sedan 2000 professor i programvaruteknik vid Blekinge tekniska högskola och prorektor sedan 2004. Tidigare har han varit professor vid Lunds universitet och Linköpings universitet. Claes Wohlin blev civilingenjör i elektroteknik vid Lunds tekniska högskola 1983. Efter forskarstudier och arbete i industrin disputerade han 1991 i teletrafiksystem med en avhandling inom programvarutillförlitlighet och -prestanda. Efter fem års arbete i industrin som programvaruingenjör, kvalitetssamordnare, projektledare och konsult återvände han 1993 till universitetsmiljön. Hans forskningsområden är i första hand: empiriska studier inom programvaruteknik, kravhantering, mått och modeller inom programvaruteknik, programvarukvalitet och processförbättring.

Claes Wohlin har utöver denna bok författat eller varit medförfattare till en bok och mer än 120 artiklar publicerade i internationella tidskrifter och på internationella konferenser inom programvaruteknik. Utöver detta är han redaktör för två böcker tillsammans med forskare från Australien. Han är chefredaktör för tidskriften Information and Software Technology som publiceras av Elsevier i Nederländerna. Han är dessutom i redaktionen för tre andra tidskrifter: Empirical Software Engineering: An International Journal, Software Quality Journal och Requirements Engineering Journal.

Claes Wohlin erhöll 2004 Telenors Nordiska Forskningspris för sina insatser inom programvaruteknik och förbättring av programvarutillförlitlighet inom telekommunikationsområdet. Han har två år i följd (2003 och 2004) rankats bland de 15 mest publicerade forskarna inom området ”system and software engineering” av tidskriften Journal of Systems and Software. Han är sedan 2005 även gästprofessor vid Chalmers tekniska högskola, där han är verksam vid IT-universitet i Göteborg som drivs gemensamt av Chalmers tekniska högskola och Göteborgs universitet.

Sakregister