Et lite kompendium i Systemutvikling

01-14 Kapittel 2

Noen viktige årsaker til for sein levering:
- Programmerer ga feilaktig status for egen kodeutvikling
- **Arbeidet var underestimert – blir forelest**
- **Dårlig eller ufullstendig planlegging – blir forelest**
- Ingen oversikt over prosjektstatus
- Lavere produktivitet enn forventet / planlagt
- **Kunden visste ikke hva han ønsket – blir forelest**

Viktig å være klar over at programvare ikke blir utviklet alene – det er alltid mer rundt som for eksempel brukere, prosedyrer for bruk, maskinvare og dokumentasjon. Alt dette må være på plass for å få gjort de oppgavene som systemet er tiltenkt. Husk. Ingen kjøper et datasystem fordi de gjerne vil ha et datasystem. De kjøper et datasystem fordi de ønsker å få løst et problem eller utført en oppgave. Mister vi det av synet blir resultatet misfornøyde kunder. **Vis fig 2.1.**

Vi må starte med planlegging. Når prosjekter går galt – og det gjør de inne imellom – så ligger mye av årsaken i dårlig planlegging.

Hva bør være med i planleggingen:
- **Prosessmodell – hvordan skal vi utvikle. Dette skal vi se på senere**
- Prosjektorganisasjon.
- **Standarder, retningslinjer og prosedyrer. Vi skal se på noe av dette senere**
- Ledelsesaktiviteter
- **Risikoanalyse. Dette skal vi se på senere**
- Personell
- **Metoder og teknikker. Vi skal se på mye av dette senere.**
- Kvalitetssikring. Vi kommer ikke til å si så mye om dette. Det er et eget kurs som behandler området.
- **Arbeidspakker. Skal snakke om dette under prosjektplanlegging**
- **Budsjett og tidsplan. Blir behandlet under planlegging**
- Endringer og endringsplanlegging
- Leveranse

Utviklingsprosesser – kapittel 3

For å kontrollere prosjekter deler vi de opp i faser og aktiviteter. Skal se på flere modeller:
- Fossefall
- Prototyping
- Inkrementell utvikling
- RAD
- Spiralmodellen

Fossefall

Fig 3.1. hovedformålet er – eventuelt var – å unngå å kaste bort arbeid. Dette skulle oppnåes ved å fokusere på å ha en solid (”frosset”) kravspesifikasjon. Modellen legger vekt på at hver fase blir avslutta med et dokumentert resultat – kravspesifikasjon, overordnet design dokument osv.

Hovedproblemene – som gjør at modellen i sin rendyrkede form sjelden blir brukt - er at:
- Store problemer hvis kunden har tatt feil, gitt ufullstendig informasjon eller bare forandrer mening.
- Det er vanskelig å gjøre ferdig for eksempel all design før man begynner å kode. Prosessen detaljert design – kode er i seg selv iterativ. Vi ser for eksempel at ca. 34% av design blir gjort i kodingsfasen og ca. 10% blir gjort under integrasjonstesting.

Tegn tabellen i **fig. 3.2** på tavla og kommenter.

Prototyping

Dette er løsningen på problemet med ufullstendig eller misforstått kravspesifikasjon og kunder som forandrer mening – ofte som konsekvens av at de får mer kunnskap eller erfaring med systemet. Vi kan bruke prototyper på flere måter:

- For å forstå kundens krav – vis **fig 3.3**
- For å eksperimentere med en algoritme eller løsning

Prototyping kan gjøres billig ved å gjenbruke komponenter man allerede har, for eksempel Excel og Access pluss litt ”lim” for å lage et enkelt lagerstyringssystem. Man kan også bruke deler av systemer man har laget før for å lage deler av den funksjonaliteten kunden etterspør.

Vanligvis vil en prototyp ha dårligere ytelse og være vanskeligere å vedlikeholde pga. at kodekvaliteten jamt over er dårligere – mer uryddig kode. Dette kan avhjelpes med streng kvalitetskontroll, men dette vil gjøre hele prosessen tyngre og langt på vei gjøre prototypingen om til inkrementell utvikling eller til xP – ”extreme programming”, som boka dessverre ikke behandler. De fordelene forfatteren beskriver med evolusjonær prototyping er også relatert til xP.

Ved å bare implementere det kunden ser han har behov for vil man i noen tilfeller lage et system med mindre, men mer brukerretta funksjonalitet.

Prototyping setter vanligvis større krav til utviklerne – det er en jobb som krever en god del erfaring hvis det ikke skal bli fullstendig ”skjærereir” , også kalt ”write only” kode.

Prosjekter som bruker prototyping må huske på at dette ikke er en unnskyldning for å kaste alle planer over bord. Slike prosjekter må også planlegges og styres.

Inkrementell utvikling

I denne utviklingsmetoden er kravene kjent ved starten av prosjektet. Når systemet likevel leveres i inkrementer er det fordi kundene erfaringsmessig:

- Ikke helt vet hva de vil. I noen tilfeller leder dette til at systemet får for mye funksjonalitet. Dette gjør systemene dyre og vanskelige å vedlikeholde.
- Endrer oppfatning etter som
    - tiden går – de får mer erfaring
    - de får erfaring med de delene av systemet de har fått levert – ser nye muligheter

Inkrementell utvikling vil vanligvis gjøres slik at man implementerer de viktigste funksjonene først. Etter hvert legger man til ny funksjonalitet og lar kundene prøve denne for å se om den er nyttig. Dette setter grenser for hvor lite et inkrement kan være – det må gi kunden ny funksjonalitet.

RAD

RAD – Rapid Applicaiton Development - har mye til felles med evolusjonær prototyping og inkrementell utvikling. I tillegg kommer time boxing, som ikke finnes i de andre metodene.

En metode som bruker time boxing har følgende egenskaper:

- Lengden på tidsboksen blir bestemt først – dermed ligger leveringstiden fast.
- Hvis vi oppdager at vi ikke kan bli ferdige med det vi hadde planlagt innenfor tidsboksen fjerner vi noe funksjonalitet. Denne blir da overført til en annen tidsboks.

RAD har fire faser i følgende rekkefølge:

1. Kravplanlegging
2. Brukerdesign
3. Konstruksjon
4. Idriftsetting

Kravplanlegging og brukerdesign blir gjort i samarbeid ned kunden. Til det bruker vi workshop’er.

- Kravplanlegging – JRP (Joint Requirement Planning). Her prøver man å bli enige om så mange som mulig av kravene slik at de ikke må endres senere. I tillegg blir krava prioritert.

- Brukerdesign – JAD (Joint Application Design). Det er vanlig å ha to slike workshoper.
    - Første JAD – første, initiell design av systemet, som er basis for en prototyp. Brukerne får denne for å eksperimentere med.
    - Andre JAD – brukernes evaluering av prototypen fra første JAD. Ut fra dette lager vi den endelige designbeskrivelsen.

Utviklingen av systemet skjer i en sekvens av tidsbokser som lager evolusjonære prototyper. Disse blir evaluert av brukerne og utviklerne i felleskap ved slutten av hver tidsboks.

Spiralmodellen

Hovedideen i spiralmodellen er å utvikle prosjektet ved å:
- Identifisere de problemene i prosjektet som for øyeblikket har den største tilhørende risiko
- Finne løsninger på disse problemene.

Modellen er derfor problem / risiko-drevet. **Se fig 3.6**. Selve programvareutviklingen foregår ved å lage et sett av prototyper som brukes til å identifisere problemer og kartlegge risiko.

01-17 Kapittel 7

All kostnadsestimering kan sees fra to sider:
- Utviklerne og prosjektleder – hva vil det kost oss.
- Selgerne – hva skal vi ta som utgangspunkt for salget

Det er uansett viktig å vite hva det kommer til å koste og hvor lang tid det vil ta. Hvis vi ikke vet det så blir det vanskelig å planlegge prosjektet.

Læreboka mener at det har katastrofale følger å bruke politiske beslutninger når man estimerer. Hverdagen for mange firmaer er nok ofte annerledes. Det er flere hensyn enn de reint datafaglige som spiller inn. Det etterfølgende er noen eksempler:
- Hva må vi by hvis vi skal få prosjektet. Hvis vi ikke får det, hva er alternativet? Har vi noe annet produktivt arbeid å sette folk til eller må vi si opp folk og hvilke konsekvenser har det?
- Hvis vi får dette prosjektet, hvilke nye muligheter gir det oss og hva er de verdt?
- Vi må komme på markedet med noe før konkurrenten ellers vil han ta en stor del av markedet. Eller tilsvarende – vi må ha noe å demonstrere på Hannovermessa.

Selv om disse punktene er viktige må vi skille skarpt mellom to ting:
- Det vi vil selge produktet for
- Det som det koster å utvikle det.

Tilsvarende:
- Hva vil det koste å lage all funksjonaliteten vi har planlagt
- Hva vil det koste å få til en første versjon med nok funksjonalitet til at det er interessant for kundene

Generelt gjelder at de fleste – kanskje alle – estimeringsmodeller er det vi kan kalle bruttomodeller. Dette innebærer at selv om de tar utgangspunkt i linjer kode, antall skjermbilder eller liknende, så er alt det andre – dokumentasjon, design, testing osv – inkludert i sluttestimatet.

Nye algoritmiske modeller

Walstone og Felix

Vis generell form – $E = (a + b*KLOC^c)*(f(X_1, X_2,\ldots X_n)$. Forklar parameterne og hvordan man kan komme fram til tallverdier for disse – lineær regresjon fra tidligere prosjekter. De enkleste formlene er den første versjonen av Walstone og Felix formel: $E = 5.2*KLOC^{0.91}$. På denne formen vil første konstanten vise produktiviteten i form av månedsverk / 1000 linjer kode. b = 5.2 betyr derfor at vi har en produktivitet på 0.19 * 1000 linjer = 190 linjer kode pr. månedsverk eller 9 – 10 linjer pr. dagsverk.

De fleste av dere vet at dere skriver mye mer kode enn det på en dag – gjerne ”noen hundre linjer”, for eksempel 200 – 300 linjer kode pr. dagsverk. I et industrielt prosjekt vil imidlertid mesteparten av tida ikke gå med til

koding, men til interaksjon med kunder, design, dokumentasjon, dokumentgransking osv. 250 linjer kode pr dagsverk tilsvarer 5000 linjer kode - altså 5 KSLOC - pr månedsverk. Dette gir en konstant på 0.2 i stedet for 5.2 som Walstone og Felix bruker.

Hovedproblemet med disse metodene er å finne brukbare estimater for KLOC tidlig i prosessen.. Det vanligste er å bryte det planlagte systemet ned i prosedyrer / moduler og bruke erfaring fra tidligere systemer til å anslå KLOC for hver prosedyre. Denne måten å estimere på tar ikke hensyn til at forskjellige prosedyrer eller subsystemer har ulik kompleksitet og derfor vil kreve ulike mengder ressurser.

For å ta hensyn til dette fant Walstone og Felix et sett med faktorer som de mente hadde stor påvirkning på produktiviteten – **se fig. 7.6. Vis og kommenter**. De definerte en produktivitetsindeks PC definert som Abs(maks – min) og derfra en vekt W = 0.5*log(PC). Videre innførte de en X, gitt som X = +1 for mindre enn normalt, 0 for normalt og -1 for mer enn normalt. På den måten gir "mer enn normalt" lavere produktivitetsindeks, mens normalt (0) gir uforandret produktivitet. Produktivitetsindeksen er gitt som I = $\text{Sum}(X_i W_i)$.

Eksempel:
Vi har anslått at vi trenger ca. 2000 linjer kode. Den enkle modellen gir oss da $E = 5.2*2^{0.91}$ som gir E lik ca. 10 månedsverk.

COCOMO

Tar utgangspunkt i same formen som Walstone og Felix, men opererer med tre typer prosjekter:

- Organisk – lite team med mye erfaring i kjente omgivelser. Vanligvis små prosjekter, b = 2.4, c = 1.05
- Embedded – store beskrankninger gitt av omgivelsene, for eksempel maskinvare, b = 3.6, c = 1.20
- Semidetached – mellomform mellom de to forgående, b = 3.0, c = 1.12

Selv om de alle bruker samme formel vil konstantene a og c variere. For å ta hensyn til forskjeller i krav og kompleksitet for de enkelte modulene innfører COCOMO et sett av kostnadsdrivere Xi. Den komplette formelen blir $E = b*KLOC^c* \text{Prod}(X_i)$. Denne tabellen har noe til felles med den vi så tidligere. **Vis tabellen fra COCOMO I.** Kommenter og regn gjennom et eksempel

Funksjonspunkter

Basere seg på et volummål som kan finnes ut fra brukergrensesnittet. De bruker følgende egenskaper:

- Antall input typer (I) – gjelder bare input som endrer data i en datastruktur, ikke input som bare styrer programmets oppførsel. Hver type som har et nytt format eller blir behandlet ulikt blir regnet med.
- Antall output typer (O) – samme forbehold som over.
- Antall spørsmålstyper (E) – forespørsler til systemet om informasjon. Dette er spørsmål som ikke forandrer data i programmet, for eksempel valg i menyer.
- Antall interne logiske filer (L). Vi bryr oss ikke om hvordan filene er fysisk organisert, men bare om antall logiske filer.
- Antall grensesnitt mot andre applikasjoner (F) – bruk eller utveksling.

Ujusterte funksjonspunkter – UFP = 4I + 5O + 4E + 10L + 7F. De vektene vi har brukt her er normalverdier. Det er mulig å i stedet bruke ulike verdier for om for eksempel en inputtype er enkel, normal (gjennomsnittlig) eller kompleks. **Se fig 7.9 og eksemplet for input type og antall dataelementer i fig 7.10**.

For å ta hensyn til en del faktorer som påvirker prosjektet – for eksempel data kommunikasjon, distribuert funksjonalitet og krav til ytelse – blir det beregnet en justeringsfaktor. Hver påvirkningsfaktor skal graderes på en skala fra 0 til 5. Summen av disse faktorene kalles DI (Degree of Influence) og vi finner en total justeringsfaktor gitt som TCF = 0.65 + 0.01*DI. Derfra finner vi justerte funksjonspunkter som UP = UFP*TCF.

Det finns en tilsvarende estimeringsmetode som bygger på use case punkter. Her teller man use case enheter i stedet for funksjonspunkter. Metoden er ny og det finnes lite erfaring med å bruke den. Vi vil derfor ikke gå inn på denne metoden her.

Eksempel:

- Antall input typer (I) - 5. Alle er enkle (3)

- Antall output typer (O) – 5. To enkle (4) og to normale (5)
- Antall spørsmålstyper (E) – 12. Tre enkle (3), to normale (4) og fem komplekse (6)
- Antall interne logiske filer (L) – 8. Alle normale (10)
- Antall grensesnitt mot andre applikasjoner (F) - 0

Dette gir UFP = 5*3 + (2*4 + 2*5) + (3*3 + 2*4 + 5*6) + 8*10 = 160

- Data kommunikasjon - 3
- Distribuerte funksjoner - 3
- Ytelse - 0
- Mye brukt - 1
- Transaksjonsrate - 0
- Online data input - 5
- Sluttbruker effektivitet - 0
- On-line oppdatering - 0
- Kompleks prosessering - 0
- Gjenbruk - 0
- Lett å installere - 4
- Lett å drive / operere - 3
- Flere installasjoner - 1
- Lett å endre – 0

DI = 20 og TCF = 0.65 + 0.01 * 20 = 0.85
FP = 160 * 0.85 = 136

Produktivitet 0.1 FP pr dagsverk => E = 136 / 0.1 = 1360 dagsverk = 68 månedsverk eller 6.2 årsverk

COCOMO II

COCOMO II er en videreutvikling av COCOMO I. Den opererer med tre detaljeringsnivåer:

- Applicaiton composition modell (APM), som hovedsakelig er beregna for prototyping.
- Tidlig design modell, som kan brukes når vi har en arkitektur design
- Post arkitektur modell, som brukes for utviklingen av programsystemet. Det er dette som er den modellen som er en videreutvikling av COCOMO I.

For APM går vi fram som følger:

1. Finn antall skjermbilder, rapporter og 3GL komponenter i systemet.
2. For skjermbilder og rapporter må vi bestemme kompleksitetsgraden som enkel, middels eller vanskelig. For rapporter finnes det en tabell for å anslå kompleksiteten.
3. Bruk tabellen i **fig 7.13** for å finnes objektpunktene for hvert enkelt objekt (ikke objekt i OO betydning)
4. Summer objektpunktene som gir oss OP
5. Estimer gjenbruksprosenten og finn nye objektpunkter som:
   NOP = OP*(1 – gjenbruksandelen)
6. Finn produktiviteten for bedriften: PROD = NOP / antall månedsverk. PROD vil avhenge av erfaring , verktøy osv og kan variere fra 4 til 50. Hver bedrift må finne sin PROD verdi.
7. Nødvendig kostnad er E = NOP / PROD

Tidlig designmodell bruke UFP. Deretter blir UFP konvertert til SLOC (antall kildekodelinjer). Konverteringsfaktoren vil avhenge av språk, omgivelser, måten man legger ut koden på etc. Noen typiske verdier er 128 linjer C og 29 linjer C++ per funksjonspunkt. Hver bedrift må finne sin egen konverteringsfaktor.

For å finne det endelige estimatet bruker COCOMO II sju kostnadsdrivere:

- En kombinasjon av kravene til pålitelighet, databasestørrelse, produktkompleksitet og dokumentasjonsbehov.
- Gjenbruk – samme som for post arkitektur modellen
- Plattform – som er en kombinasjon av krav til eksekveringstid, lagerplass og erfaring med utviklingsverktøy
- Utviklernes erfaring med applikasjon, plattform og utviklingsverktøy
- Utviklernes dyktighet som systemanalytikere, programmerere og hvor lenge de har vært i bedriften
- Utviklingsverktøy og geografisk spredning av prosjektet.
- Kravene til kalendertid

Hver av disse sju kostnadsdriverne graders på en sjupunkts skala fra ”Ekstra lav” til ”ekstra høy”. Ut fra dette finner man er verdi for hver kostnadsdriver som multipliseres sammen for å justere estimatet.

Post arkitektur modellen er den samme som COCOMO I bortsett fra to ting:
- Noen av kostnadsdriverne er forandret.
- Isteden for å ha tre typer utvikling har den en korreksjonsfaktor som bygger på et sett av skaleringsfaktorer som kan ha verdier fra 0 (svært høy) til 5 (svært lav).

Kostnadsdriverne er som vist i **fig 7.14**. Korreksjonsfaktoren b blir beregnet ut fra følgende fem skaleringsfaktorer:
1. Graden av ukjent – nytt arbeid, applikasjonsområde etc. Alt nytt = 5, ingenting nytt = 0
2. Behovet for å være konform med tidligere systemer. Bundet opp av gamle systemer = 5, helt fritt = 0.
3. I hvor stor grad vi eliminert i boende risiko i utviklingen og hvor mye av arkitekturen er spesifisert. Lite (20% eller mindre) = 5, alt = 0.
4. I hvor stor grad fungerer prosjektteamet. Dårlig kommunikasjon og samarbeid = 5, meget godt samarbeid = 0.
5. Hvor moden er organisasjonene når det gjelder planlegging, estimering og utvikling. Helt umoden = 5, meget moden = 0. COCOMO II bruker CMM her, men har også ei egen sjekkliste til dette bruket.

VI finner da $b = 1.01 + 0.01*\text{Sum}(W_i)$ og $E = a*\text{KSLOC}^b*\text{Prod}(\text{kostdrivere})$. Antall kodelinjer er her det tallet vi får ved å konvertere funksjonspunkter til kodelinjer.

Vi kan ta hensyn til gjenbruk ved å justere anslaget for antall linjer kode. I COCOMO antar man at man har følgende fordeling:
- Design. 40% - DM
- Koding (utvikling) 30% - CM
- Integrasjon av komponentene 30% - IM

Man må så anslå hvor mye av systemet som må designes, kodes og integreres på nytt. Ut fra dette får ,man en ny korreksjonsfaktor AAF = 0.4DM + 0.3CM + 0.3IM. I COCOMO I nøyde man seg med dette. I COCOMO II ser man i tillegg på to andre forhold.
- SU som tar hensyn til graden av modularitet i koden. Dette er et tall varierer fra 0.10 til 0.50 for meget ustrukturert, tett koblet kode.
- AA som tar hensyn til innsatsen vi trenger for å vurdere om komponenten er brukbar for det planlagte systemet.

Vi kan nå finne et justert antall kodelinjer gitt som AKLOC = KLOC * (AAF + SU + AA). Vi kan nå bruke ALOC i stedet for KLOC i de formlene vi har sett på tidligere.

Kost / nytte analyse

Når man har en kostnadsmodell som fungerer – gir fornuftige svar – kan den brukes til å se på nytten av endringer i prosjekt, metoder eller opplæring. Fra tabellen over kostnadsdrivere i COCOMO II ser vi at faktoren for programmerernes kunnskap varierer fra 1.37 (dårligst kunnskap) til 0.74 (best kunnskap). Dette vil utgjøre en faktor 1.85 fra dårligste til beste programmerer. Har vi for eksempel et prosjekt på 100 årsverk med dårlige programmerer kan vi redusere det til 54 årsverk ved å bruke de beste programmererne. Ett årsverk koster ca. 1 million kroner slik at besparelsen er på 46 millioner.

Tilsvarende betraktninger kan gjøre for andre faktorer – for eksempel kostnader ved en ekstra fil eller et ekstra skjermbilde.

Eksempel

Vi har et system med kostnadsdrivere som vist i fig. Ut fra de vurderingene som er gjort finner vi at Prod(kostnadsdrivere) = 1 * 0.93 * 1 * 0.91 * 0.95 * 1.11 * 1 * 0.87 * 1.22 * 1 * 1.10 * 1 * 1.12 * 1 * 1.12 * 1.25 * 1 = 1.63

For å bestemme b ser vi på skaleringsfaktorene:

1. Graden av ukjent – nytt arbeid, applikasjonsområde etc. Alt nytt = 5, ingenting nytt = 0. **1**
2. Behovet for å være konform med tidligere systemer. Bundet opp av gamle systemer = 5, helt fritt = 0. **2**
3. I hvor stor grad vi eliminert i boende risiko i utviklingen og hvor mye av arkitekturen er spesifisert. Lite (20% eller mindre) = 5, alt = 0**. 4**
4. I hvor stor grad fungerer prosjektteamet. Dårlig kommunikasjon og samarbeid = 5, meget godt samarbeid = 0. **2**
5. Hvor moden er organisasjonene når det gjelder planlegging, estimering og utvikling. Helt umoden = 5, meget moden = 0. COCOMO II bruker CMM her, men har også ei egen sjekkliste til dette bruket. **5**

b = 1.01 + 0.01*(1 + 2 + 4 + 2 + 5) = 1.15

I det forrige eksemplet fant vi UFP = 160. Hvis vi skal utvikle produktet i C finner vi at dette tilsvarer 20480 kodelinjer og KSLOC = 20.5.

$E = a * KSLOC^b * Prod(kostnadsdrivere)$ gir oss $3*20.5^{1.15}*1.63 = 157.7$ månedsverk eller 14 årsverk. Hvis vi bruker a = 0.2 – som eksempel på et ”studentprosjekt med studentkvalitet” finner vi i stedet E = 10.5 månedsverk.

Kalendertid

De fleste estimeringsmodeller har en formel som gjør at man kan utlede kalendertid fra utviklingskostnader. Det er ikke slik at prosjektet må ta denne tiden – det kan godt ta lengre tid hvis vi vil det. Det er imidlertid vanskelig å få gjennomført prosjektet på kortere tid.

Walstone og Felix: $T = 2.5*E^{0.35}$

COCOMO I – organsikk prosjekt $T = 2.5*E^{0.38}$

COCOMO II – nominell tid $T = 3.0*E^{0.33 + 0.2*(b - 1.01)}$

Det er to hovedgrunner til at vi ikke kan korte ned på tida på et pågående prosjekt ved å sette inn flere folk:

- Mengden av nødvendig kommunikasjon og koordinering øker. Dette krever tid.
- Når vi tar inn nye folk et prosjekt må de ha opplæring. Dette betyr at de ikke er spesielt produktive den første tiden. I tillegg tar de ressurser fra prosjektet ved at de som kjenner prosjektet må fungere som veiledere for de nyansatte.

Data som er samlet inn tyder på at produktiviteten L – antall linjer pr. månedsverk – er avhengig av temastørrelsen P på følgende måte: $L = 777*P^{-0.5}$. Mange synes nok 777 linjer per månedsverk er lite. Dette tallet inkluderer imidlertid ikke bare koding, men også overordna design, detaljert design, integrasjon, alle typer dokumentasjon, prosjektmøter – interne og med kunde – og så videre. Mange bedrifter bruker lavere tall – for eksempel 200 linjer kode pr. månedsverk totalt over hele prosjektet.

Eksempel

I forrige eksempel fant vi at det prosjektet vi så på ville kreve 158 månedsverk. For COCOMO II vil dette gi en gjennomføringstid på $T = 3.0*158^{\,0.33 + 0.2*(1.15 - 1.01)}$. Dette gir oss T = 18.4 måneder. Dette tilsvarer en bemanning på for eksempel 8 personer på full tid og en person på deltid. Å sette på flere personer vil antakelig ikke være effektivt. Men bare føre til større kostnader.

Studentprosjektet hadde 10.5 månedsverk og den samme formelen gir i dette tilfellet T = 7 måneder

01-17 Kapittel 8

Prosjektkontroll behandler tre elementer:

- Det vi skal kontrollere – selve prosjektet
- Det som kontrollerer prosjekt – prosjektleder, den organisasjonen han har bak seg og de reglene og metodene han bruker
- Den informasjonen han bruker til å styre sine beslutninger og dermed prosjektprosessen. Denne informasjonen kommer fra to kilder:
    - Fra prosjektet – for eksempel et notat om et problem som har oppstått
    - Fra prosjektets omgivelser – for eksempel ordre om å fjerne funksjonalitet, kutte kostnader eller levere tidligere.

For å kunne kontrollere et prosjekt effektivt er det viktig å kjenne prosjektets mål. De fleste prosjekter har flere mål. Typiske prosjektmål kan være:

- Kortest mulig utviklingstid
- Mest mulig fornøyde kunder
- Mest mulig fornøyde sluttbrukere
- Mest mulig gjenbruk
- Mest mulig gjenbrukbar kode
- Størst mulig fortjeneste

Det er ikke alltid mulig å oppfylle all prosjektets mål i like stor grad. Noen ganger vil to eller flere mål til og med kunne være motstridende – for eksempel god vedlikeholdbarhet og stor tidseffektivitet. Noen ganger vil ledelsen være i stand til å prioritere, andre ganger er det umulig å få til – ”Alt er like viktig”.

For å få effektiv prosjektledelse må følgende betingelser være oppfylt. – vi må

- Kjenne målene til prosjektet
- Ha tilstrekkelige virkemidler som vi kan bruke slik vi mener det er nødvendig
- Kjenne prosjektets tilstand – ressursforbruk, hva er gjort.
- Vite hvordan en endring i prosjektets parametere påvirker resultatene og målene.

Hvis vi hadde visst alt dette kunne prosjektledelse vært en enkel, rasjonell prosess. Antakelig kunne den vært helt eller delvis automatisert. Dessverre er de fire betingelsene over bare delvis oppfylt. Mulige problemer er at vi ikke:

- Kjenner alle målene eller hvordan de egentlig er prioritert

- Har tilstrekkelig med virkemidler. Vi er begrenset av lover (interne og eksterne), vi har de folkene vi har og kan ikke få noen andre, det er bare 24 timer i døgnet, folk kan bli syke eller tatt av prosjektet for å gjøre noe som haster mer – brannslukking.
- Kjenner prosjektets tilstand. Det er vanskelig å vite hvor langt har X kommet med modul Y når vi får svaret at han er "snart ferdig" eller er "90% ferdig med kodingen".
- Vet hva som vil skje når vi endrer en grensebetingelse – for eksempel pålegger alle i prosjektet 2 timer overtid pr. dag. Noen vil produsere for to timeverk ekstra, noen vil jobbe langsommere, noen vil bli småsyke osv. Mennesker er ikke maskiner selv om prosjektleder gjerne vil tro det.

Tre viktige faktorer:
- Produktvisshet
    - Hvor klart er kravene spesifisert – vet vi hva vi skal lage
    - Hvor stabile er kravene – har kunden bestemt seg for hva han vil ha
- Prosessvisshet
    - Kan vi endre prosessen. Dette er viktig for våre muligheter til å styre / påvirke prosjektet
    - Er det mulig å måle viktige parametere i prosessen. Dette er viktig for å kjenne prosjektstatus.
    - Hvor godt forstå vi effekten av endringer i en eller flere prosessparametere. Dette er viktig for våre muligheter til å påvirke prosessen i den retningen vi ønsker.
- Ressursvisshet
    - Hvor godt kjenne vi tilgangen på ressursene i prosjektet. Dette gjelder primært personell, men i mindre grad også maskinvare – særlig viss prosjektet er avhengig av spesiell maskinvare.

Hvordan vi vil estimere avhenger av graden av ressursvisshet og prosessvisshet. Når denne er lav vil det være gunstig å gjøre en følsomhetsanalyse. Dette skjer som følger:
- En hver estimering bygger på et sett av forutsetninger. Disse må identifiseres og dokumenteres. Eksempler er kvalifikasjonene til personellet, antall utviklere, kundens krav til for eksempel pålitelighet eller tilgjengelige verktøy.
- Hva skjer med estimatet hvis en av disse forutsetningene må forandres?
- De forutsetningene som fører til store endringer i estimatet er kritiske. De må derfor vurderes ekstra nøye og overvåkes under hele prosjektet slik at vi kan passe på at de holder stikk eller foreta nødvendig re-planlegging av (deler av) prosjektet hvis de svikter.

Fire viktige typer av prosjektsituasjoner avhengig av de tre faktorene over og hvorvidt de tre typene visshet er høy eller lav:

	Realisering	Allokering	Design	Eksplorering
Produktvisshet	Høy	Høy	Høy	Lav
Prosessvisshet	Høy	Høy	Lav	Lav
Ressursvisshet	Høy	Lav	Lav	Lav

Ved å kombinere rimelige situasjoner får vi fram fire problemsituasjoner med hvert sitt fokus:
- Realisering – hvordan oppnå måla på den mest effektive måten. Dette er et ønskeprosjekt og forekommer dessverre sjelden. Vi kan bruke en enkel, lineær prosessmodell. Prosess og nødvendig kunnskap kan fastsettes på forhånd. Jobben kan deles opp i et sett av arbeidsoppgaver som kan styres og kontrollers. En av de forslåtte metodene for kostnadsestimering kan brukes direkte.
- Allokering – usikkerhet med hensyn til ressurser. Hovedproblemet er tilgjengelighet av personell. Hvem vil jobbe på prosjektet, hvor mye vil de jobbe, når kan de gjøre det? Viktige momenter her er å standardisere så mye som mulig. Dette vil gjøre det lettere å bytte folk. Vi kan stadig bruke en enkel utviklingsmodell og en av de estimeringsmetodene vi har snakka om tidligere, men når vi ikke vet hvem som skal gjøre jobben vil det være flere parametere som er ukjente eller vil variere over tid. For eksempel COCOMO kostnadsdrivere og hvordan de avhenger av bemanningen til prosjektet.
- Design. Design refererer her til **design av prosjektet** – ikke til design av produktet / systemet. Krava er fastlagt, men vi har ikke bestem oss for hvordan vi vil realisere systemet – lav visshet på prosess og ressurser. Viktige spørsmål er valg av milepæler, valg av dokumenter som skal lages, valg av personell og fordeling av ansvar. Vi trenger standardiserte løsninger, dokumenter og prosedyrer. Dermed er output gitt og vi kan styre gjennom valg av prosess og ressurstilgang. Vi trenger generell overkapasitet siden det kan bli mye skifte av personer og dårlig tilgang i noen perioder. Siden vi ikke kan regne med overkapasitet i personer, må vi prøve å få ekstra resurser i form av mer tid – timeverk og kalendertid.

Vi kan ikke bruke noen av de beskrevne estimeringsmodellene. I stedet må vi bruke det vi har av erfaringer og data fra tidligere, liknende prosjekter. Viktig med følsomhetsanalyse. Dette vil fortelle hva som er kritisk og hvilke muligheter man har for å endre prosjektet.

- Eksplorering – lite kunnskap om krav, ressurser og prosess. Dette er en vanskelig, men slett ikke uvanlig situasjon. Innenfor enkelte markedssegmenter er dette nesten den mest vanlige måten å jobbe på. Dette gjelder for eksempel SME i deler av internettmarkedet. Styring og prosess vil være ad hoc. Den kritiske suksessfaktoren er i hvor stor grad deltakerne er engasjert i å få til et godt produkt. Prototyping og tett kobling mellom utviklere og kunder er absolutt nødvendig. Estimering vil stort sett måtte skje via ekspertvurderinger. Eksperter er folk som har deltatt i slike prosjekter før.

Viser til fig 8.2, som oppsummer hele problemstillingen.

Risikostyring

Risikoanalyse og risikokontroll er viktig i all prosjektstyring. Det har vært en tendens i mange programvareprosjekter til å ignorere risiko – "vi vil bekymre oss om det viss det hender" er en vanlig respons. Praktisk erfaring viser imidlertid at det da ofte er for seint. Vanligvis vil en risikoanalyse foregå som følger:

1. Identifiser mulige risikofaktorer. Dette skjer via en eller annen form for idemyldring – alt fra helt uformelt til det å følge en eller annen dokumentert prosess. Dersom det er mulig bør vi også prøve å identifisere risikoindikatorer – altså ting vi kan observere når risikoen er på vei til å inntreffe. Dette vil gi oss en tidlig advarsel og gi oss bedre tid til å finne og iverksette tiltak.
2. Finn effekt (K for kostnad) og sannsynlighet (p). Risikoen (R) er da gitt som R = K*p. I prinsippet kunne man tenke seg å finne tallverdier – for eksempel tapet er NOK 100 000 og sannsyneligheten er 0.01 – men dette er vanligvis vanskelig. Derfor opererer de fleste med en eller annen skåringsmekanisme, for eksempel bruke H, M og L (høy, middels og lav) eller tallverdier, for eksempel 10, 5 og 1. Tallverdiene gjør det lettere å beregne et risikotall. Dette gjør at man ikke kan finne noen absoluttverdi fro hver risikofaktor, men må nøye seg med å gradere og rangere de. Dette er nok for å lage ei prioritert liste over risikoer og tiltak.
3. For alle viktige risikoer må vi definere tiltak. Disse vil tilhøre en av følgende kategorier:
   - Unngå – innføre tiltak for å fjerne risikoen. Er det en spesiell funksjon vi tror det blir vanskelig å implementere kan vi ha et forprosjekt for å forstå den bedre eller vi kan prøve å forhandle den bort med kunden. Er det problemer med å bruke et spesielt verktøy kan vi gjøre en rådgiving og veiledningsavtale med en konsulent eller et firma osv.
   - Overfør – flytt risikoen ut av prosjektet, for eksempel til et forprosjekt – vanlig måte å behandle ustabile krav på - eller et parallelt forskningsprosjekt.
   - Aksepter – dette er noe vi ikke kan gjøre noe med nå. Det må lages en plan for hva vi skal gjøre hvis det inntreffer og sørge for at en person er ansvarlig for å sette den i verk hvis risikoen skulle gå over til å bli et problem.
4. Håndter risikoen. Overvåk de viktige risikoer og eventuelt deres indikatorer. Siden et prosjekt ikke er en statisk enhet er det viktig å gjenta prosessens trinn 1 – 3 med jevne mellomrom. Det kan dukke opp nye risikoer, noen risikoer vil kunne forsvinne eller deres sannsynlighet eller konsekvens vil forandre seg.

Resultatet bør dokumenteres i en risikotabell som minimum bør inneholde følgende informasjon:

Aktivitet	Kostnad (K)	Sannsynlighet (P)	Risiko (R = K * P)	Tiltak	Ansvarlig

Bruk et regneark. Da er det lett å oppdatere og sortere. Dersom det er mulig bør man også ta med indikatorer for hver enkelt risiko.

Prosjektplanlegging

Prosjektet består av mange aktiviteter. Det er viktig å få oversikt over aktivitetene før vi begynner å planlegge. En måte å få oversikt over disse på er å lage en WBS – Work Breakdown Structure.

Det er viktig at alle i prosjektet er med på denne prosessen. Aktiviteter som blir glemt må legges inn i prosjektet seinere – ofte med store ekstrakostnader. Det kan være lurt å lage WBS i flere omganger etter at man har laget det øverste nivået – hovedaktivitetene. Viktig input er:

- Den erfaringa hver enkelt deltaker har
- Resultater – WBSer - fra tidligere prosjekter.
- Den utviklingsmodellen / prosessen vi har valgt

To tommelfingerregler for å lage en WBS:

- Ingen arbeidspakke bør være større enn 40 – 100 timeverk
- Bryt alltid ned til et nivå der du forstår hva som skal gjøres, enten fordi det er opplagt eller fordi den som skal gjøre jobben kjenner den igjen.

I henhold til disse reglene vil et prosjekt på ett årsverk (1600 tv) bestå av 16 til 40 arbeidspakker.

Når vi har identifisert alle arbeidspakkene må vi:

- Finne ut hvor mye ressurser (Dagsverk - Dv)vi trenger for å lage hver enkelt pakke. Deretter må vi bestemme hvor mange personer (n) vi sette på jobben. Dette vil definere varigheten (T) på aktiviteten ved at T = Dv / n. Her må vi passe på to ting:
    - Det er ikke alle Dv / n som er meningsfylte. For eksempel vil det være urimelig å tro at vi kan få gjort unna en 100 Dv jobb på en dag ved å sette på 100 personer. I tillegg er det en del aktiviteter som er avhengige av en modningsprosess. Slike aktiviteter er det vanskelig å gjennomføre på kort tid.
    - Vi har gjort en del forutsetninger under estimeringa om kvalifikasjonene til de som skal gjøre jobbene. Disse forutsetningene må respekteres, eller må vi estimere om igjen.
- For hver arbeidspakke, bestemme hvilke andre arbeidspakker som må være ferdige før vi kan starte arbeidet med denne pakka – avhengigheter.

Denne prosessen kan føre til at vi må endre WBS eller estimatene. Dette er en fordel, ikke et problem:

- Eksempel på et tilfelle der vi må endre WBS er hvis vi finner ut at pakke B må starte etter pakke A, men ikke hele pakke A må være ferdig, bare noen deler. Da er det lurt å dele pakke A opp i to pakker A1 og A2 og si at B kan starte når pakke A1 er ferdig.
- Endring av estimatene kan være aktuelt i flere tilfeller. Vi vil se på to slike:
    - Vi har en ferdig WBS og har kostnadssatt alle pakkene slik at summen blir lik det totale estimatet. Det viser seg imidlertid at flere av pakkene ser ut som de er svært mye underestimert. Løsningen på dette er å identifisere de underestimerte pakkene, gi de et realistisk estimat og summere på nytt.
    - Vi har en fordelingsnøkkel, for eksempel at design tar 40% - 10% på overordna design og 30% på detaljert design - og koding og testing tar 30% hver. Da kan vi benytte samme strategi som over, men mer detaljert.

Resultatet skal oppsummeres i en tabell med følgende format:

Arbeidspakke	Før-jobber	Dagsverk (Dv)	Antall personer (n)	Varighet (T = Dv / n)

Ut fra denne tabellen kan vi nå lage PERT diagram og Gantt diagram som er de vanligste diagrammene for å planlegge et prosjekt.

Som eksempel kan vi se på følgende tabell:

Arbeidspakke	Før-jobber	Dagsverk (Dv)	Antall personer (n)	Varighet (T = Dv / n)
Design	-	10	1	10
Test plan	Design	5	1	5
Kode A	Design	10	1	10
Kode B	Design	5	1	5

Testing	Test plan Kode A Kode B	10	1	10

Dette gir følgende PERT diagram., Se fig. 8.8. Dette gir oss en kritisk vei som er 30 dager lang. Dette må sees sammen med den varigheten vi estimerte eller lovet kunden tidligere. Hvis kritisk vei er kortere enn den estimerte / lovede tiden er alt OK. Dersom dette ikke er tilfelle må vi prøve å fikse det ved å:

- Prøve å korte ned på varigheten til en eller flere aktiviteter på den kritiske veien ved å sette på flere folk.
- Prøve å fjerne avhengigheter som gjør at vi kan oppnå en større grad av parallellitet i prosjektet.
- Dele opp aktiviteter slik at vi får redusert aktivitetene på den kritiske veien.

De aktivitetene som ligger på den kritiske veien i PERT diagrammet må overvåkes ekstra nøye. En hver forsinkelse på en av disse vil forsinke hele prosjektet. De andre aktivitetene har det vi kaller en slakk. Dette medfører at forsinkelser i disse aktivitetene ikke er kritiske siden de ikke påvirker den totale varigheten av prosjektet. De kan imidlertid raskt bli så mye forsinka at vi må redefinere den kritiske veien og sluttidspunktet.

Ut fra PERT diagrammet kan vi lage et Gantt-diagram – se fig. 8.7. Ut fra Gantt-diagrammet kan vi lage oss en bemanningsprofil ved å summere antall planlagte dagsverk pr. dag eller måned. Denne profilen er viktig fordi den kan hjelpe oss med å foretelle når vi:

- Trenger ekstra folk
- Kan avgi folk til andre prosjekter

For det eksemplet vi så på i fig. 8.7 finner vi følgende bemanningsprofil.

Aktivitet	5	10	15	20	25	30
Design	1	1				
Testplan			1			
Kode A			1	1		
Kode B			1		1	1
Test						
Sum	1	1	3	1	1	1

Vi ser at vi bare trenger en person unntatt i dag 10 – 15, der vi trenger tre personer. Da er det viktig at det virkelig er tre personer tilgjengelig på det planlagte starttidspunktet – altså etter 10 dager. Dersom de ikke er tilgjengelig vil det medføre forsinkelser i hele prosjektet – en dag for hver dag de ikke er tilgjengelig. Noen tror at det går an å ta igjen dette, men erfaring viser at det skjer bare i noen få tilfeller.

WBSen, PERT diagrammet og Gantt-diagrammet bør oppdateres med jamne mellomrom. Det er derfor viktig med verktøystøtte i disse aktivitetene. Diagrammene gir oss en god mulighet til å følge opp prosjektet. Dette kan skje på flere måter:

- Når noen aktiviteter er ferdige kan vi sammenlikne medgåtte resurser mot estimerte ressurser. Dersom det er en systematisk forskjell bør vi vurdere å justere resten av estimatene også. Denne justeringa vil gi en god pekepinn på hvordan vi ligger an i forhold til planen.
- Vi kan la det ***estimerte ressursforbruket*** for hver pakke representere verdien av pakka. Dette er rimelig viss vi har solgt prosjektet for den estimerte verdien. Hvis vi er ferdige med arbeidspakker som representerer X % av estimatet vil det være rimelig å si at prosjektet er X % ferdig.
- Hvis det er aktiviteter på kritske vei som ikke er ferdige når de skulle være det, bør man vurdere å flytte folk fra ikke-kritiske til kritiske aktiviteter. Pass på at de virkelig kan gjøre nytte for seg i denne aktiviteten ellers går det bare fra galt til verre. Trenger de for eksempel opplæring?

01-21 Fowler kapittel 2

UML er et modelleringsspråk, ikke en utviklingsmetode. RUP – Rational Unified Process er en metode som er spesielt utviklet for å gå sammen med UML. Man kan imidlertid bruke UML uansett hva slags prosess og utviklingsmetode man velger. Fowler har definert en liten, enkel, iterativ prosess som passer godt for mange små og middels store prosjekter. Grovt sett består denne prosessen av følgende:

- Inception – finner ut om det foreslåtte prosjektet er nyttig og mulig å realisere innenfor rimelig tid og ressursforbruk
- Elaboration – samle inn krav og lage en overordnet analyse og design
- Konstruksjon – en serie av iterasjoner for å realisere systemet
- Transisjon – vi slipper ut første versjon – betaversjonen

Vi vil se på hver av disse fasene i mer detalj.

Inception

Det er mange måter å gjennomføre dette på, måten avhenger av hvem kundene er og hvor stort prosjektet er. Viktige ting å få på plass på et overordnet nivå er:

- Hva skal vi lage
- Hvem skal ha det
- Hva skal det koste
- Tror vi at vi kan få det til

Hvis det ser ut til at det er "la-seg-gjørlig" så går vi over til neste fase, ellers er det bare å glemme hele greia.

Elaboration

Dette er primært en risikodrevet fase. Vi trenger oversikt over de risiki som ligger i prosjektet. Det kan være praktisk å operere med fire risikokategorier:

- Kravrisiki – analysen av kravrisiki begynner med use-case.
- Teknologiske risiki
- Ferdighetsrisiki
- Politiske risiki

- Kravrisiko – hva er krava til systemet, bygger vi det riktige systemet
- Teknologisk risiko – har vi den riktige teknologien for denne jobben
- Kunnskapsrisiko – har vi den kunnskapen vi trenger tilgjengelig for dette prosjektet
- Politisk risiko – finnes det politiske (her i betydningen intern bedriftspolitikk) krefter som kan komme i veien for prosjektet.

Vurdering og handtering av disse risiki kan gjøres slik vi har snakket om før. Kategoriene og det vi sier om de her, kan brukes som ei sjekkliste for å øke sannsynligheten for at vi har fått med oss alle de viktige risikofaktorene. Vi skal se litt på hver av disse kategoriene.

Kravrisiko

Før vi kan se på kravrisiko må vi raskt definere et use case. Et use case er en funksjon som brukeren trenger, som har verdi for han og som er beskrevet på en måte som han forestår. Use case er i UML den viktigste måten å kommunisere med kundene på. På dette tidspunktet behøver ikke use casene å være detaljert – noen avsnitt med tekst er nok. Det forteller hva funksjonen skal hete og hva kunden ønsker å oppnå med funksjonen.

I tillegg til et sett av - de viktigste – use case, trenger vi en **konseptuel modell av domenet** (fagområdet). En av oppgavene til denne modellen er å beskrive og definere hvordan dette fagområdet bruker en del viktige termer. Dette er viktig for å forstå den verden systemet skal fungere i. Tre teknikker er viktige. Vi skal se på de i mer detalj senere.

- Klassediagram, tegnet fra et konseptuelt perspektiv.
- Aktivitetsdiagram, for å identifisere hva som skal gjøres.
- Interaksjonsdiagram, for å identifisere hva som skal gjøres.

For alle disse diagrammene gjelder at man på dette stadiet i prosjektet kan og bør være svært uformell. I tillegg til diagrammene og den tilhørende notasjonen kan det være lurt å ha en del forklarende, uformell notasjon. Hovedpoenget er å få oversikt over domenet, ikke å lage en korrekt, rigorøs modell. Modellen skal være et **skjellet, ikke en overordna modell**. En overordna modell er en høynivå beskrivelse av systemet og inneholder derfor ingen detaljer. Et skjellett kan godt inneholde detaljer, men bare de vi synes er viktige for øyeblikket.

For å få til en brukbar domenemodell er det viktig å lage modellen sammen med kunden. Dette er nok en viktig grunn til å holde notasjonen enkel og uformell.

Klassediagram

Klassediagrammer beskriver typer av objekter i systemet og de statiske relasjonene som kan være mellom de. Det finnes to typer av relasjoner:

- Assosiasjoner, for eksempel en kunde leier en video
- Subtyper, for eksempel und-ass er en subtype av student.

Vis fig. 4-1 fra Fowler. Noen kommentarer på en del egenskaper, i det minste generalisering, klassebetegnelse med navn, attributter og operasjoner og relasjoner og multiplisitet.

Aktivitetsdiagram

Aktivitetsdiagrammer består – ikke uventa – av et sett av aktiviteter og hvordan de er kobla sammen. Aktivitetsdiagrammer kan i noen tilfeller være et godt alternativ til interaksjonsdiagrammer. Hver aktivitet er enten en prosess – noen gjør noe - eller en eksekvering av et program. Diagrammet viser sekvenseringen av aktivitetene. **Vis fig. 9-1 fra Fowler**. Kommenter sentrale begreper som ”Fork”, ”Branch” og ”Join”. Aktivitetsdiagrammer blir ikke behandla i kurset ut over dette.

Interaksjonsdiagram

Av samarbeidsdiagrammer er det interaksjonsdiagrammer som blir prioritert i dette kurset. Disse diagrammene viser hvordan klassene arbeider sammen og utveksler meldinger. Klassediagrammer og interaksjonsdiagrammer blir utvikla i en iterativ prosess**. Vis fig 5-1 i Fowler**. Kommenter sentrale begreper som ”Objekt”, ”Message” og ”Creation”.

Mer om kravrisiko - 1

Kombinasjonen av domenemodell og use case gir en god kombinasjon for å forstå kravrisikoen i prosjektet. Husk at formålet er å forstå risiko og gjøre noe med den, ikke å lage et systemdesign. Dette impliserer at vi bruker mye uformell notasjon og ekstra kommentarer i tillegg til de diagrammene vi har nevnt tidligere.

Når vi har en god domeneforståelse kan vi ta en runde til på use casene og til slutt ta en foreløpig siste runde på domenemodellen. I dette arbeidet må vi inkludere en eller to domeneeksperter. Dersom domenemodellen er stor kan vi prøve å dele den opp i pakker og gjerne begynne å se på tilstandsdiagrammene til klasser som har særlig interessant eller vanskelig oppførsel.

Pakker

Et pakkediagram er et klassediagram som viser bare pakker av klasser og avhengigheter mellom de. Vi har en avhengighet mellom to klasser viss endringer i definisjonen i den ene medfører endringer i den andre. Avhengighetene kan være at den ene sender en melding til den andre, en klasse bruker deler av en annen klasse som parametere i en operasjon eller ved at den ene bruker den andres grensesnitt. **Vis fig 7-1 i Fowler**. Vi har avhengigheter mellom pakker viss vi har avhengigheter mellom en klasse i den ene pakka og en klasse i den andre.

Tilstandsdiagrammer

Et tilstandsdiagram viser alle tilstandene til et **objekt** og hvordan tilstanden til objektet endrer seg som resultat av hendelser. Tilstandsdiagrammet berører altså objekter – instansinerte klasser – og ikke de definerte klassene**. Vis fig 8-1 fra Fowler** og kommenter. Overgangene er merket med ”Hendelse [Betingelse] / Handling”. ”Handling” er det samme som en tilstandsendring – transisjon.

Mer om kravrisiko – 2

Pass hele tiden på at vi lager et skjellett, ikke en overordna – høyere ordens – modell. Viss vi prøver å ta med alle detaljer over alt på dette tidspunktet vil det ta alt for lang tid og vi kan lett glemme hensikten, nemlig å få en oversikt over kravrisikoen. Vi må passe på å hele tiden jobbe videre med use casene for å se om de introduserer elementer som gjør at vi må endre noen av de andre diagrammene i skjellettet.

Viss vi føler oss usikre på hvordan vi skal realisere en del av domenemodellen, er det lurt å lage en prototyp. Ikke lag prototyper av alt, men bruk domenemodellen til å oppdage ting som dere føler dere usikre på.

Den kritiske delen av arbeidet med kravrisiko og domenemodellen, er tilgang på domeneekspertise. Vanligvis betyr dette tilgang på eksperter / brukere fra kunden. Vi må altså jobbe tett sammen med kunden i denne perioden.

Teknologisk risiko

Det viktigste vi kan gjøre her er å lage en eller flere prototyper for å prøve ut den teknologien vi skal bruke senere i prosjektet. Eksempler på slik teknologi kan være databaser, kommunikasjonspakker og brukergrensesnittpakker. Den største risikoen er ikke hvordan hver pakke fungerer eller er designet, men hvordan pakkene passer sammen med hverandre og med det du skal lage selv. Dette må derfor prøves ut så tidlig som mulig. Vi må også vurdere arkitekturbeslutningene på dette trinnet. Dette er særlig viktig viss vi skal lage et distribuert system.

Det er viktig å se på de beslutningene som det er vanskelige å gjøre om seinere. Tre viktige spørsmål:

- Hva vil skje viss en del av teknologien ikke fungerer
- Hva må vi gjøre viss to dele av teknologien ikke fungerer sammen
- Hvor sannsynlig er det at noe går galt her og hva kan vi gjøre viss det inntreffer.

I denne prosessen er det igjen viktig å se på use case, klassediagrammene og pakkediagrammene for å se om det er noe du har oversett.

Kunnskapsrisiko

At vi mangler ekspertise er en vanlig grunn til at prosjekter går galt. Vanligvis er årsaken at man ikke vil bruke penger på kurs – rett og slett fordi det koster penger. På den andre siden ser det ikke ut til at de bryr seg om at overskridelsene vanligvis koster langt mer. Kursing av prosjektdeltakerne – og av ansatte generelt – er en god investering. Problemet er imidlertid ofte at mange prosjektorienterte bedrifter mangler et investeringsbegrep og investeringsopplegg.

Det beste er å ha et opplegg med korte, målretta kurs etterfulgt av eget arbeid med hjelp av en mentor. I begynnelsen vil mentoren være en del av prosjektet for så seinere å trekke seg tilbake og fungere som orakel og rådgiver. Viss man ikke kan skaffe en mentor så pass på å hold hyppige gjennomganger av det som blir gjort slik at man kan gi hverandre råd, diskutere problemer og lære av hverandres tabber og suksesser.

Vær på utkikk etter områder i prosjektet der du mangler kunnskap selv. Dette er områder der vi trenger kurs, en eller flere mentorer eller å leie inn en erfaren konsulent.

Politisk risiko

Dette angår risiko som oppstår internt i bedriften og i forholdet mellom bedriften og kundene. De mest vanlige konfliktene – og dermed risikofaktorer – er:

- Allokering av personell. Vi har blitt lovet fire personer, men får bare tre, eller vi får personer med lavere eller annen kompetanse enn den vi hadde regnet med.
- Tilgang til personell. Vi må avgi folk før vi hadde tenkt pga. at et annet prosjekt blir prioritert høyere av ledelsen eller fordi ”det brenner” i et annet prosjekt eller fordi en viktig kunde har rapportert en feil.

Andre risiki som hører til her er at vi blir lovet kurs som så blir kutta pga. kostnader eller – viss vi skal lage produkter til et generelt marked – bedriften bytter markedsstrategi. Endring i prioriteter pga. at vi blir oppkjøpt er også noe som har blitt vanlig også i norsk industri.

Dette er problemer som må løses av ledelsen, ikke av de som jobber i prosjektene. Det kan imidlertid være lurt å vurdere de og bestemme seg for hva man skal gjøre på et tidlig tidspunkt. Det kan være nyttig å informere ledelsen på et tidlig tidspunkt om at disse risiki finnes og hva som kan bli konsekvensen viss de inntreffer.

Når er ”Elaboration” ferdig

Tommelfingerregel: Denne fasen tar ca. en femtedel av det totale prosjektet. Det finnes to indikatorer som sier at vi er ferdige:

- Utviklerne synes det er OK å prøve å estimere hvert use case – kostnader og kalendertid – til nærmeste ukeverk.
- Alle signifikante risiki har blitt identifisert og vurdert så godt at du føler at du kan handtere de uten å se store problemer.

Konstruksjonsfasen

Planlegging

Konstruksjonsfasen består ev en serie av iterasjoner. Det kan være lurt å lage mange små use case og realisere et use case pr. iterasjon. Igjen må vi passe på å involvere kundene. Begynn med å kategorisere use casene. Dette gjøres som følger:

1. Kunden deler opp use casene i henhold til deres verdi for bedriften – bruk kategoriene høy, middels og lav
2. Utviklerne deler opp de samme use casene i henhold til risiko – samme kategoriene.
3. Utviklerne estimerer hvert use case – kostnad og ledetid. Dette er netto tid – full tid på produktivt arbeid. Etter som vi får erfaring – f.eks. etter første iterasjon – må vi justere for overhead.
4. Bestem hvor lang du vil ha hver iterasjon – dette må være nok til å gjennomføre 3 – 6 use caser. Uansett bør en iterasjon ikke være mye lenger enn en til halvannen måned.
5. Bestem prosjektets hastighet. Dette gjøres på følgende måte:
   - N = antall utviklere
   - L = lengde på en iterasjon – se punkt 4.
   - F = last faktor – se punkt 3. Viss vi f.eks. bruker fire ukeverk på å få gjort det som er estimert til å ta tre ukeverk er load faktoren 4/3 = 1.3
   - Dette gir hastigheten V = N * L / F. Viss vi har fem utviklere og to ukers iterasjoner med en lastfaktor på 1.3 finner vi V = 5 * 2 / 1.3 = 7.7, som er innsats pr. iterasjon.
6. Når vi har prosjekthastigheten V kan vi finne antall iterasjoner IT som følger:
   IT = 1 + Sum(Ukeverk pr. use case) / V. Viss vi har funnet at vi totalt trenger 77 ukeverk for å realisere alle use casene finner vi IT = 1 + 77 / 7.7 = 11 iterasjoner.

Ut fra kundens og utviklernes vurderinger kan vi nå lage følgende tabell over use casene

		Kundens vurdering av verdi for kunden		
		H	M	L
Utviklernes vurdering av risiko	H			
	M			
	L			

De use casene som havner i høy risiko og / eller stor verdi må taes tidlig i prosjektet.

Sett av tid for integrasjon og levering av betaversjon. Sett av 10% viss du har gjort det (mange ganger) før og 35% viss dette er nytt for deg. Til slutt bør du legge til 10 – 20% som reserveressurser.

Lag en releaseplan – når kan vi demonstrere hvert inkrement og hvilke use case vil det inneholde.

Konstruksjon

Hver iterasjon kan sees som et miniprosjekt med analyse, design og koding etterfulgt av testing og integrasjon. Når iterasjonen er realisert kan du demonstrere det nye systemet for kunden. Iterasjonene er både inkrementelle og iterative.

- Inkrementell fordi den legger til noe ny funksjonalitet hver gang.
- Iterativ fordi vi hver gang må skrive om noe av den koden vi allerede har for å tilpasse nye inkrementer.

Tre aktiviteter som må gå kontinuerlig:

- Refaktoring. Når vi jobber med inkrementer må vi vanligvis endre noe kode for hvert inkrement vi legger til. Refaktoring er endringer som ikke påvirker funksjonaliteten, men i stedet rydder opp i koden, ofte ved å endre klasser, gjøre enkel redesign osv. Det finns noen regler – de tre viktigste kommer her:
  1. Ikke gjør refaktoring og implementasjon samtidig eller om hverandre. Ha er klart skille mellom disse to aktivitetene

2. Pass på å ha et komplett sett tester før du begynner refaktoring slik at du kan validere at det som var OK før stadig er OK.
3. Ta korte steg – som bare inneholder noen få endringer i koden. Test etter hvert steg.

- Integrering. Integrer ny kode så snart den er tilgjengelig. Kjør alle tester etter hver integrasjon.
- Testing. Lag tester kontinuerlig slik at du til en hver tid kan teste det du har implementert pluss all tidligere funksjonalitet. Lag et opplegg for automatisk testing slik at du slipper å sjekke alt for hånd.

Konstruksjon begynner med design. Første skritt i design er å gå gjennom alle klassene som er beskrevet tidligere og se hvordan de bidrar til å realisere den funksjonaliteten som er beskrevet i use casene. Fowler anbefaler CRC kort. **Vis fig 5-6 i Fowler og forklar**. Ut fra CRC korta finner vi ansvar og operasjoner som skal inkluderes i et eller flere klassediagram. Diskuter klassediagrammene – videre utarbeidet fra de skjelettene vi laget tidligere – med kolleger.

Nå er vi klare til å kode. Koding og etterfølgende testing vil vanligvis føre til endringer i design. Lag ikke komplett designdokumentasjon – i det minste ikke nå. De vil sikkert bli endra seinere. Bruk heller Javadoc eller likende – for eksempel BlueJ - til å generere designdokumentasjon ut fra den koden som finnes. Da blir det enkelt å oppdatere dokumentasjonen og den vil alltid være riktig - så lenge vi ikke får feil i Javadoc. Pakkediagrammene viser hvordan systemet er satt sammen og hvilke logiske deler det har. Dette diagrammet er vanligvis mer stabilt enn de mer detaljerte klassediagrammene.

I tillegg vil det være praktisk / nyttig å oppdatere de viktigste, mest kompliserte tilstandsdiagrammene. I slike tilfeller bør man også vurdere om man trenger aktivitetsdiagrammene.

Transisjon

I denne fasen skal det ikke legges til funksjonalitet, unntatt viss endingene er små og absolutt nødvendige. I tillegg må vi rette feil. Det kan være aktuelt å gjøre optimalisering viss systemet er for tregt – så tregt at det irriterer kunden. Husk at optimalisering – både for tid og plassforbruk – har en tendens til å gjøre koden vanskeligere å forstå, og dermed å endre(for eksempel for å rette feil eller legge til ny funksjonalitet senere).

24-01 Kapittel 13

På et overordnet nivå finnes det bare to typer testing:

- Testing for å se om vi har feil i koden, for å hjelpe til med å finne de og sørge for at feilene blir rettet. Dette er den testingen som bedriften gjør under utvikling. Hele opplegget er internt og under bedriftens kontroll.
- Testing for å bygge tillit til at produktet gjør jobben sin eller tilfredsstiller kundens krav. Ofte vil kunden si at ”Vi godkjenner systemet når følgende tester er kjørt uten feil…”.

Det finnes mange måter å test på. Alle har som formål å gi et best mulig utbytte av testingen. Noen interessante varianter er:

- Dekningsbasert testing. Det finnes mange måter å definere dekningsgraden på, for eksempel
  - kodedekningsgrad – alle **kodesetninger** er kjørt minst en gang.
  - stidekningsgrad – alle mulige **veier** er kjørt minst en gang. Stidekning og kodedekning er ikke det samme. **Vis og kommenter fig 13.3.** Linje 7 har en if-setning. Hva skjer hvis betingelsen ikke er sann?
  - løkkedekning – for eksempel **alle løkker skal kjøres null, en og N ganger**
  - definisjon – bruk dekningsgrad – **alle par av tilordninger og bruk av en variable** skal utføres.
  - kravdekningsgrad – **alle krav skal dekkes**. Hva vi må gjøre avhenger av hvor detaljerte vi er med hensyn på hvert krav.
- Feilbasert testing. Vi setter inn en del feil i koden i tillegg til de som allerede er der. Vi sier at koden er godt nok testet når vi har funnet en gitt andel – for eksempel 99% - av alle innsatte feil.
- Problembasert testing. Vi bruker den kunnskapen vi har om hva som er vanskelig – dvs. ofte feiler – og setter inn mesteparten av innsatsen der.
- Risikobasert testing. Vi ser på hva som medfører de alvorligste konsekvensene for kunden hvis det feiler og tester mest der. Dette vil for eksempel bli gjort for sikkerhetskritiske systemer.

Det kriteriet vi velger vil bestemme når vi er ferdige med å teste. Vi behøver ikke å kreve 100% dekning. I mange tilfeller vil det ikke engang være realistisk. Dette gjelder bla. for stidekning siden antall mulige veier gjennom et program fort blir stort.

Det er også vanlig å dele opp testing i to strategier:

- ”Black box” testing. All testing – input og output – tar utgangspunkt i kravene til systemet, modulen eller pakka.
- ”White box” testing. All testing tar utgangspunkt i strukturen til koden. Dette passer godt for de fleste dekningsgradsmål.

Testing sammenfattes ofte under tittelen V&V – Verifisering og Validering. Disse to begrepene dekker to viktige sider:

- Validering – har vi laget det systemet vi skulle lage, dvs. riktig funksjonalitet
- Verifisering – har vi laget systemet på en riktig måte, dvs. har vi tilfredsstilt krav til pålitelighet, vedlikeholdbarhet osv i tillegg til kundekrav. I tillegg vil vi vite om vi har brukt den prosessen vi skulle – har vi for eksempel gjort gransking av alle moduler?

Det er ikke uten videre enkelt å definere hva en feil er. **Tegn diagrammet med mulig input og hva slags input som aktiverer feil** og hvilken som ikke gjør det. Si noe om koblingen mellom brukerprofil – bruksmåte – og hvor feil ligger i systemet.

Vis og kommenter fig 13.2 Si noe om problemene med å finne et automatisk orakel. Dette medfører at vi nesten alltid må ha et menneske som orakel.

Partisjonstesting

Partisjonstesting går ut på å dele opp inputrommet i ekvivalensklasser, der data i hver klasse vil aktivere samme del av koden. Enkle eksempler:

IF (a > 5) P1( ) ELSE P2( ) deler inputrommet i to deler. For å teste dette trenger vi minst å ha a større enn 5 og a mindre eller lik 5.

IF (a > 5) {…IF (b < a) P1( ) ….ELSE ….} deler inputrommet i tre deler. For å teste denne biten trenger vi å ha a større enn 5 og a mindre eller lik 5 og b mindre enn a og b større eller lik a.

Simulering har vist at ved å velge et stort antall tilfeldige inputverdier vil man vanligvis få et like godt resultat som ved partisjonstesting. Når strukturen til inputrommet blir kompleks vil vi uansett ikke kunne lage et ryddig sett med partisjoner og generering av tilfeldige verdier vil være et godt alternativ.

Testing og utvikling

Det er for seint å begynne å teste for å finne og rette feil når systemet er ferdig. I den grad det er mulig må testing foregå under alle aktiviteter i et utviklingsprosjekt. Vi vil kort se på noen av aktivitetene og hva vi skal gjøre der:

- **Krav** – finnes ofte bare som naturlig språk eller i en eller annen uformell notasjon – for eksempel use case. Det er fire ting som er viktige med krav til et programvaresystem. Det skal være:
    - Komplett – alle elementer av kravet skal være på plass
    - Konsistent – ingen av kravene skal stride mot noen av de andre
    - Gjørlig – kravene skal kunne realiseres med kjent teknologi. I tillegg mener noen av det i dette også ligger at kravene gir et system som er kostnadseffektivt – det koster mindre å lage systemet enn det man sparer ved å ta det i bruk.
    - Testbare – vi skal kunne teste om kravet er oppfylt.

  For å teste krav er det ingen annen teknikk som kan brukes enn en eller annen leseteknikk – dokumentgranskning, inspeksjon eller liknende.
- **Design**. De samme kravene som vi hadde til kravspesifikasjon er også viktige her. I tillegg er det viktig å ha sporbarhet. Dette impliserer at det må være mulig å se hvor hvert enkelt krav er realisert. Teknikkene for å teste er vanligvis de samme – inspeksjon og granskning. I tillegg vil det være mulig å implementere deler av systemet – i det minste klasseskjelletter og tomme prosedyrer.
- **Implementering**. Det er her vi gjør ”ordentlig” testing siden det er her vi har kode vi kan eksekvere. Sporbarhet er stadig viktig og gjelder begge veier:
    - Fra krav til test – denne testen er her for å teste følgende krav…
    - Fra test til krav – dette kravet blir testet av følgende test…

Bare på den måten kan vi være sikker på at vi ikke har mistet krav underveis i prosessen fram til nå.

- **Vedlikehold**. En stor del av utgiftene med å ha et programvaresystem kommer etter at systemet er ferdig – kanskje så mye som 2/3. Dette er vedlikehold, som har to årsaker:
  - Feil – ca. 30%
  - Endringer – ca 70% - fordi kundens behov eller omverdenen har endra seg.

  Når vi gjør en endring – uansett årsak – er det tre ting det er viktig å teste:
  - Har vi oppnådd den effekten vi ønsket
  - Fungerer alt annet som før
  - Er koden stadig like lett å forstå og å finne fram i.

Manuelle testteknikker

Den mest brukte manuelle testteknikken er å lese dokumenter – krav, design, kode og tester. Det mest effektive er å få to – tre andre personer til å gå systematisk gjennom det du har skrevet, men hvis det er vanskelig så vil det også hjelpe om du får en kollega til å "se på det" eller bare leser gjennom det selv etter å ha latt det ligge urørt en dag eller to. Vi vil se kort på noen teknikker:

- **Walkthrough**. Dette begrepet brukes på to måter:
  - Formell teknikk for å gå gjennom kode, ved å håndeksekvere den ved hjelp av enkle testdata. Hensikten er ikke å teste koden, men å bruke testdata til å forstå hvordan koden fungerer og bruke det som utgangspunktet for en diskusjon om kodekvaliteten. Det er denne betydningen som står i læreboka.
  - Uformell teknikk for å presentere et løsningsforslag / konsept for andre utviklere for å samle ideer til å forbedre løsningen. I denne formen brukes teknikken før man har skrevet ferdig koden, men etter at man har kommet fram til hvordan man vil løse problemet.. Det er denne betydningen som blir mest bruk, i det minste i Europa. Jeg vil anbefale at dere holder dere til denne måten å gjøre en walkthrough på.
- **Inspeksjon**. En formell prosess for å godkjenne et dokument. Den kan brukes på alle typer dokumenter selv om den nok blir brukt mest på kode. Det er imidlertid viktig å også inspisere kravdokumenter, designdokumenter, testplaner osv. En gruppe av – vanligvis – fire personer går gjennom dokumentet og ser etter inkonsistenser og feil. "Fagan-inspeksjon" – slik den er presentert her – legger lite vekt på forberedelsene, mens andre mener at dette er den viktigste aktiviteten. Andre ting som varierer fra bedrift til bedrift er hvorvidt forfatteren skal være til stede. T. Gilb, som også har arbeidet mye med inspeksjoner, har et opplegg det man alltid har et walkthrough tidlig i utviklingsprosessen. Nyere forskning synes å vise at de fleste feil finnes under de individuelle forberedelsene og at nytten av selve møtet er marginal. Jeg mener dere bør bruke følgende opplegg:
  1. Skisser en relativt detaljert løsning. Det må være noe mer konkret og forpliktende enn bare noen løse tanker.
  2. Når dette er ferdig så bør det presenteres for ei gruppe av medarbeidere – minst tre – fire stykker. Hensikten er å oppnå konsensus – dette er en fornuftig måte å løse problemet på – og å få innspill ut fra de erfaringene de andre har. Hva man får avhenger av hvem man trekker inn i prosessen.
  3. Ta med den informasjonen du samlet under det foregående møtet og gjør ferdig dokumentet.
  4. Lever dokumentet til den som er ansvarlig for inspeksjoner. Han vil gjøre følgende:
     - Går gjennom dokumentet for å se om det har en reell sjanse for å bli godkjent – eventuelt med noen endringer. Hvis han ikke tror det så må han levere dokumentet tilbake til forfatteren med beskjed om å forbedre det.
     - Sette sammen et inspeksjonsteam. Deltakerne velges ut fra applikasjonskunnskap og kunnskap om prosess og metode som er brukt i det aktuelle prosjektet.
     - Alle deltakerne for utlevert dokumentet og bakgrunnsmateriale som for eksempel kravspesifikasjon – hvilket problem skal vi løse, hvem er kunden, hva er kundens krav, relevante standarder
     - Deltakerne får en viss tid på å gå gjennom dokumentet og lage ei liste over ting de mener er feil, ting som kan være feil og ting de ikke forstår.
     - Etter at forberedelsestida er over vil lederen for inspeksjonen kalle inn til et møte der man går gjennom hver enkelt persons problemlogg og lager en felleslogg for inspeksjonen. I noen tilfeller vil fellesgjennomgangen føre til at man oppdager nye problemer, men dette er dessverre sjelden.
     - Etter at man er ferdige med loggen må man bli enige om hvorvidt man vil
       - Godkjenne dokumentet som det er – noe som skjer svært sjelden

- Godkjenne dokumentet med endringer. Dette er OK hvis vi tror at det er en liten jobb å rette opp problemene – for eksempel en til to dagsverk
- Større endringer – ny inspeksjon etter at alle endringene er utført. Dette kan også bli resultatet hvis det er store mengder småfeil.
- Forkastet – dette dokumentet er hinsides hjelp. Behold erfaringene og start prosessen på nytt.

5. Når endringene er utført i henhold til den beskjeden man har fått fra inspeksjonen er dokumentet godkjent og vi kan gå videre i prosessen.

Dette opplegget er altså forskjellig fra det som står i læreboka. Dere finner et sett med foiler til denne måten å gjøre det på i dagens innslag i forelesningsplanen på fagets hjemmeside.

- **Scenariobasert** evaluering. Dette er en manuell testing – ofte kalt desk checking – av kode ut fra et use case eller et scenario – situasjonsbeskrivelse. Boka beskriver SAAM –Software Arkitektur Analyse Metode - prosessen som er tilpasset arkitekturevaluering. Metoden kan også brukes i vanlig testing. Siden arkitekturen i stor grad vil bestemme hva vi kan og ikke kan gjøre seinere er det viktig å gjøre en god jobb her. I tillegg til selve evalueringa har metoden også andre fordeler, for eksempel den å tjene som et utgangspunkt for diskusjoner om hva systemet skal gjøre. Dermed vil det tjene til å lage forståelse og konsensus i prosjektet og mellom prosjektet og kunden. Selve SAAM prosessen er som følger:
    1. lag scenarier. Disse kan inneholde scenarier både fra kravspesifikasjonen og fra situasjoner som vi tror vil kunne dukke opp senere.
    2. beskriv de arkitekturene vi vil evaluere.
    3. klassifiser scenarioene i relasjon til hver av arkitekturene som direkte (fullt støttet) og indirekte (delvis støttet)
    4. beskriv alle nødvendige endringer i arkitekturen som er nødvendige for å støtte de indirekte scenariene.
    5. vurder interaksjonene mellom de indirekte scenariene. Hvis flere indirekte scenarier krever endringer i samme komponent kan dette være en indikasjon på at vi har fordelt funksjonaliteten på en uheldig måte i arkitekturdesignet.
    6. oppsummering. Gi en vekt til hvert scenario etter hvor viktig det er. På den måten kan vi finne den arkitekturen som støtter de fleste og viktigste scenariene. .

Dekningsbaserte testteknikker

Dette er en enkel og viktig måten å vurdere tester på. Vi vil se på noen varianter:

- **Kontrollflytbasert** dekning. Kravet går ut på at man skal teste alle veier gjennom programmet. Tatt bokstavelig er dette kravet vanskelig å oppfylle hvis vi har løkker. Det er derfor definert to varianter av dette kriteriet som er noe enklere å forholde seg til:
    - Setningsdekning. Alle programsetninger blir utført minst en gang
    - Stidekning. Alle stier blir gjennomført minst en gang.
    - Utvidet stidekning. Alle mulige kombinasjoner av predikater blir eksekvert. Hvis vi for eksempel har betingelsen (a > b && x = 10) må vi ha fire datasett siden vi har to relasjoner som da kan være henholdsvis FF (a $\leq$ b, x =7), FT (a $\leq$ b, x= 10), TF (a > b, x = 7) og TT (a > b, x = 10).

  Vis fig 13.10 og diskuter noen tilfeller.
    - Vi kan oppnå full setningsdekning ved følgende parametervalg:
      n = 2, a[1] = 5, a[2] = 3. Det gjør imidlertid ikke noen forskjell for resultatet om betingelsen på linje 9 endres fra "$\geq$" til "=". Vi får heller ikke eksekvert THEN grena i linje 6 eller den implisitte ELSE greina i linje 9.
    - Hvis vi legger til datasettet n = 2, a[1] = 3, a[2] = 5, får vi også eksekvert THEN greina i linje 6.
- **Dataflytbasert dekning**. Ser her på dekning av alle par av tilordninger – bruk for alle definerte variable. Det er vanlig å skille mellom to typer bruk – P-bruk når den variable brukes i et predikat så som j > 5 – og C bruk som er all annen bruk. Et annet sentralt begrep i dataflytbasert testing er at en sti kan være "definition clear" med hensyn til en gitt variable. Dette betyr at den variable ikke endrer verdi når vi beveger oss langs denne stien. Vi skiller mellom to strategier:
    - "All Bruk" dekning. Dette kravet går ut på å lage testdata slik at vi får gjennomløpt alle "definition clear" tilordninger – bruk stier pluss den setningen som kommer etter bruken. Det siste er nødvendig for at vi skal være sikre på å få med begge grenene i en IF-setning.

    - Alle DU stier (Define Use). Dette er det samme som ”All bruk” kravet, men har som en tilleggsbetingelse av alle ”definition clear” stier er uten repeterbare løkker (FOR- eller WHILE-løkker).
    - I tillegg finnes det mange varianter av dette. Se i læreboka side 452.
- **Kravspesifikasjonsbasert** dekning. Her omformer vi systemkravene til en rettet graf. Tekstlige use case og UML aktivitetsdiagrammer er et godt utgangspunkt for slike grafer. Vi kan definere dekningsgradsmål på slik grafer som på alle andre grafer – for eksempel alle mulige stier.

Feilbasert testing

Formålet med feilbasert testing er å teste slik at vi har et mål for hvor godt testen dekker systemet – hvor stor sannsyneligheten er for at vi har funnet all eller de fleste feil når testingen er ferdig. Vi vil se på et utvalg av teknikker:

- Feilsåing går ut på å legge inn et sett av feil i systemet og deretter teste systemet. Kvaliteten på metoden avhenger av om vi setter inn realistiske feil eller ikke. La oss anta at vi har N1 feil i systemet og legger inn N2 nye feil. **Tegn figur på tavla**. Vi kjører et sett av tester og finner n1 feil vi ikke har lagt inn og n2 av de feilene vi har lagt inn. Under forutsetning av at alle feilene – både de virkelige og de som er lagt inn etterpå – er like lette å finne kam vi nå sette at n1/N1 = n2/N2 og dermed **N1 = N2*n1/n2**. I stedet for å lage nye feil kan vi utnytte de feila vi finner i systemet.
    - La to grupper teste programvaren uavhengig av hverandre, men slik at gruppe 1 tester først og gruppe 2 etterpå. **Tegn opp den vedlagte skissa**. Feilene blir ikke fjerna før gruppe 2 er ferdige med å teste. La n1 og n2 være antall feil funnet av henholdsvis gruppe 1 og gruppe 2. Antall feil som forekommer hos både gruppe 1 og gruppe 2 vil bli betegna med c. Da gjelder n1/N1 = c/n2 og dermed **N1 = n1*n2/c**.
    - Alternativt kan vi bruke metoden med feilsåing, men sette inn feil som tidligere er fjerna fra systemet under annen testing, for eksempel enhetstesting.

  Uansett valg av metode vil det være slik at vi kan ha tillit til testingen hvis vi finner alle eller de fleste av de feilene som er sådd, og få andre.
- Mutasjonstesting. Denne testformen har mye tilfelles med feilsåing, men bygger på to hypoteser:
    1. **Hypotesen om kompetente utviklere** sier at den programvaren som blir utvikla er svært nær den riktige. Avvik fra programvaren vil derfor være feil.
    2. **Hypotesen om kobling mellom feiltyper** sier at de tester som finner små feil også finner store – her i betydningen alvorlige – feil.

  Mutasjonstesting foregår derfor ved at vi lager små variasjoner av den koden vi har utviklet. Disse nye versjonene av koden inneholder feil (hypotese 1) og kalles mutanter. Hvis en test avdekker denne feilen er mutanten fjernet – død. Vi må altså fortsette å lage nye tester helt til vi har funnet mutanten.

Noen egenskaper ved testkriterier

Det store spørsmålet for testing er: Når har vi testet nok. Boka lister opp 11 kriterier av varierende nytte og interesse. Vi har valgt ut noen av kriteriene som vi finner nyttige:

- To programmer som er semantisk like – gjør samme jobben – kan ikke nødvendigvis testes like godt med samme datasett.
- To programmer som er syntaktisk nær hverandre – samme struktur og dataflyt – kan ikke nødvendigvis testes like godt av samme datasett.
- Det at et program er testet tilstrekkelig for en brukergruppe eller et bruksområde betyr ikke at det er testet tilstrekkelig for et annet brukesområde.
- Det at komponenter er testet hver for seg og funnet feilfrie er ingen garanti for at de fungerer sammen
- Minstekravet for et sett av testprogrammer er alle setninger i programmet blir eksekvert minst en gang.

Det som er gjort av eksperimenter synes å vise at det ikke finnes ***en beste testmetode***. Det er enighet om at:

- Inspeksjon er mer effektivt til å finne feil enn testing. I tillegg har inspeksjoner den fordelen at det kan brukes mye tidligere enn testing – allerede ved overordnet design.
- Jo tidligere i utviklingsprosessen man begynner med inspeksjoner jo bedre er det.
- For tradisjonell programvareutvikling blir det mer kostbart å rette feil, jo senere i prosessen man gjør det. Det ser imidlertid ut til at dette ikke gjelder for utvikling der man begynner å eksekvere / simulere allerede i designfasen – noe som er mulig med SDL og Java / BlueJ.

Det er imidlertid ikke sikkert at man får mer pålitelig programvare av dette. Problemet er at inspeksjon finne feil jevnt spedt utover i koden mens vi - når systemet er i bruk – vil eksekvere forskjellige deler av koden med svært forskjellig intensitet. De fleste feilhendelsene blir forårsaket av et lite antall feil i programmet – kanskje så lite som 5%. Det er derfor viktigere å finne de få, men kritiske feilene enn å kvitte seg med feil generelt. Av den grunn er det utviklingsteknikker som fraråder å bruke enhetstesting og bare teste med brukerspesifikke data. Dette gjelder for eksempel for Cleanroom.

Top-Down vs. Bottom-Up

Top-down og Bottom-up er to populære teststrategier – begge med sine fordeler og ulemper. Vi skal raskt se på hva de innebærer:

- Top-Down: Vi begynner med hovedprogrammet og legger til moduler etter hvert som de blir ferdige. Der vi mangler ferdige moduler setter vi inn tom kode – også kalt stubber.
- Bottom-Up: Her begynner vi med enkeltmoduler så snart de er ferdige og legger til moduler etter hvert. For å teste moduler der nivået over i kallhierarkiet ennå ikke er ferdige må vi lage ekstra kode – også kalt drivere.

I begge tilfeller må man altså skrive kode som senere skal kastes – henholdsvis stubber og drivere. Dette koster penger og i tillegg kan vi få problemer pga. feil i denne ekstrakoden.Hvis man lager et systemskjelett først kan man sette inn moduler etter hvert som de blir ferdige. Dette er antakelig en bedre løsning og kan gjøres blant annet ved hjelp av BlueJ.

01-31 kapittel 9

Ifølge IEEE er et krav ”en betingelse eller evne som trengs av en bruker for å løse et problem eller nå et mål”. Vi vil bruke begrepet ”Krav Engineering” i stedet for kravanalyse for å få fram at dette er en iterativ samarbeidsprosess som inkluderer:

- Problemanalyse
- Dokumentering av foreløpige resultater
- Sjekke vår forståelse
- Gjenta trinnene hvis nødvendig.

Selv om kravhåndtering og design strengt tatt er separate aktiviteter, er det neppe mulig å skille de helt. Lista overfor vil derfor ofte bli utvida med **preliminær design**.

Læreboka opererer med tre prosesser i kravfasen. **Vis fig. 9.1**. Disse er:

- **Finne/ekstrahere/oppdage krav**. Her er det viktig å forstå fagområdet. Hvis det er flere fagområder involvert, må man forstå dem alle. I det minste på et nivå der det er mulig å kommunisere. Det er viktige forskjeller på å utvikle programmer for en kunde og utvikle programvare for et marked.
- **Spesifisere krav**. Beskrive kravene slik at de kan implementeres. Vi må fokusere på produktet, prosessen må komme senere. Det finnes mange måter å beskrive krav på: fra naturlig språk - brukt mer eller mindre formelt – til formelle notasjoner som Z og VDM. Vi skal se på en av disse teknikkene – tilpasset OO – senere, i kapittel 12.

- **Verifisering og validering av krav**. Vi må sjekke om vi har de rette kravene (validering) og om vi har formulert disse kravene på en riktig måte (verifisering).

Finne/ekstrahere/oppdage krav

Strengt tatt går dette ut på å modellere en liten del av verden. Den modellen vi lager blir kalt en eksplisitt, konseptuel modell. En slik modell inneholder den bakgrunnskunnskapen som alle de som arbeider med det aktuelle domenet har. All denne kunnskapen er implisitt for de som arbeider i domenet og vår oppgave er å gjøre denne kunnskapen eksplisitt, slik at den blir tilgjengelig for andre. Viktige problemer som vi må takle i denne prosessen er:

- Mange personer er ikke bevisst den kunnskapen de har – dvs. de er ikke vant til å uttrykke den, det er bare ”noe alle vet”.
- Kunnskapen forandrer seg over tid
- Systemutviklerne og kundene/brukerne snakker forskjellige språk. Det er ikke sikkert de brukerne vi snakker med skjønner hva en fil er, eller at vi vet hva FAS-data er.
- Det er ikke sikkert at brukerne er i stand til å gi en konkret, eksplisitt beskrivelse av all den kunnskapen de sitter inne med.
- Det vil være nyanser mellom personer når det gjelder måten å beskrive kunnskap på, enten fordi de er uenige eller fordi de representerer ulike grupper av personell.

I tillegg til disse problemene har vi andre, mer ulne problemer. Noen viktige av disse er:

- Folk flest har en svært begrenset korttidshukommelse. Dette betyr at de alltid vil huske best det som skjedde sist og vil la dette dominere for eksempel en samtale om hva de gjør, hva slags problemer de har og hva de trenger av datastøtte i sitt daglige arbeid.
- Mennesker er svært selektive i hva de husker – enten ting som har gått svært bra eller ting som har gått svært dårlig. Det som er vanelig er det ikke alltid like lett å huske.
- Vi skal ikke overvurdere menneskers evne til rasjonell tenking. Ofte bruker de forenklede eller direkte gale modeller når de skal forklare ting.

Alt dette til sammen gjør at vi ikke kan satse på å bare få informasjon fra brukere og andre kunder. Vi må i tillegg ha tilgang på skriftlig informasjon og bruke kundens erfaring og kunnskap til å tolke/forstå det skriftelige materialet.

Vi kan ikke forvente at kundene alltid vet hva de vil ha – noen ganger er det bare på jakt etter en eller annen løsning på et problem. Man risikerer for eksempel at mens kunden vil ha et program som løser problemet, så er det egentlige problemet ikke løsbart ved hjelp av et datasystem. Det kan for eksempel være et menneskelig problem, et administrativt problem eller et kompetanseproblem. Første fase i kravarbeidet må derfor være å forstå problemet eller – sammen med kunden – å komme fram til en felles forståelse av problemet.

Det finnes grovt sett to typer problemer vi kan komme ut for når det gjelder kravarbeid:

- **Tekniske problemer** – implementer ei algoritme som finner verdien til et ikke-komplett Gammaintegral. Dette er stort sett enkle problemer. Alt arbeidet ligger i å finne en eller flere matematiske formler og implementere de.
- **Organisatoriske problemer** – lag et nytt system for prosjektstyring. Dette er vanskelige problemer i og med at de
    - Griper inn i måten organisasjonen arbeider på
    - Angår mennesker

Teknikker for å finne krav

Vis fig 9.3 og gå raskt gjennom teknikkene. Vi har valgt ut noen få som vi vil si mer om: nemlig intervju, sammen med idémyldring, ”task” analyse, use case analyse, domeneanalyse og prototyping. Det kan være greit å tenke tilbake på den prosessen som ble foreslått av Fowler, kapittel 2, som foreslår å kombinere use case og klassediagram/aktivitetsdiagram (domeneanalyse) sammen med brukeren/kunden.

Intervju

Begynn med å spørre hva brukerne forventer fra systemet. Uansett hva vi leverer – hvis det er noe annet enn det de forventer ligger vi i utgangspunkter dårlig an. Vi kan bruke gruppeprosesser, spørreskjema eller snakke med hver enkelt bruker. I grupperprosesser må vi passe på at ikke en eller noen få dominerer prosessen, mens mange

andre ikke kommer til ordet eller ikke tør å si noe. Det finns flere metoder for å drive anonyme gruppeprosesser. Vi skal se kort på en av de:

1. Førstemann skriver en ide, et behov eller et krav på et ark og sender det til neste mann.
2. Nestemann kan da:
   - Gi ytterligere bidrag til den ideen som allerede er skrevet ned
   - Bare sender arket videre – hvis han ikke har noe nytt å bidra med
   - Starte en ny ide.

   Deretter sender han arket/arkene videre til neste mann i ringen.
3. En ide/et ark er ferdig når den person som "startet" arket ser at det ikke har kommet på noe nytt i den siste runden.

På denne måten kan man raskt samle mange ideer og flere innspill til hver ide, samtidig som alle er sikret en viss grad av anonymitet. En lik prosess kan også lett implementeres i et nett av datamaskiner, for eksempel på toppen av Netmeeting.

Taskanalyse

Vi lager en liste av jobbene brukerne skal utføre. Deretter blir hver enkelt jobb brutt ned i stadig mer detaljerte beskrivelser – noe av det samme som vi gjør når vi lager en WBS.

Use case analyse

Dette likner på taskanalyse, men i stedet for å gå systematisk gjennom alle mulige – og umulige – arbeidsoppgaver, ser vi her på et sett av scenarier. Scenariene kan være virkelige eller teoretiske.

- **Virkelig** – vi observerer brukeren mens han gjør en jobb som vi senere skal implementere. Ved å la brukeren forklare hva han gjør og hvorfor, kan vi følge med i prosessen og finne de kravene som prosessen må tilfredsstille.
- **Teoretisk**- vi sier til brukeren: "Tenk deg at du skal…. Hva ville du gjøre da?"

De viktigste utfordringene ved bruk av denne metoden er å finne ut når vi har nok use case og hvor detaljerte de skal være. Vi skal se mer – mye mer – på use case i UML litt senere.

Domeneanalyse

Domeneanalyse går ut på å forstå det området – domenet – systemet skal fungere i. Ofte vil denne analysen ta som utgangspunkt allerede eksisterende systemer innenfor det samme området. Man kan se på dette som et forsøk på å finne eller utvikle en referansemodell for området – hvordan løser man problemer av type X innefor dette området. Domeneanalyse vil vi behandle videre i UML, der vi bruker klassediagrammer i denne prosessen.

Prototyping

En prototyp tar utgangspunkt i de første kravene man kan få samlet inn på "en eller annen måte". Deretter viser vi prototypen til brukerne og spør: "Var det dette dere trengte?" Svarene vil bestemme den videre utviklingen av en blanding av nye prototyper, diskusjoner og demonstasjoner.

Dokumentasjon av krav

Det som kommer ut av denne prosessen er en kravspesifikasjon, som er utgangspunktet for høynivå design. En måte å dokumentere den på, er å bruke IEEE Std 830. Dere finner den i vedlegg C i læreboka – side 663 og utover. Det er noen viktige krav som en kravspesifikasjon må oppfylle.

- **Korrekt**. Vi kan ikke garantere dette, men vi må sjekke så godt vi kan mot kunden og mot andre relevante dokumenter. Feil i KS er kostbare å rette. Hvis vi oppdager de seint kan vi risikere å måtte gjøre om på alt fra og med overordnet design.
- **Entydig**. Dette betyr at de som har skrevet den og de som skal lese den får samme oppfatning av hva som skal gjøres**. Vis diagram med kommunikasjon mellom to grupper og avhengigheten av valgt, implisitt modell**. Ofte er problemet her at det finnes ting som kunden eller utvikleren anser for å være opplagt, men som ikke er det for den andre gruppa.
- **Komplett**. Alt som er viktig for kunden eller for å realisere systemet må stå her. Dette gjelder for eksempel sånt som respons på feil data eller ulovlige kommandoer eller kombinasjoner. All erfaring viser at det er vanskelig å få en komplett KS tidlig, rett og slett fordi kunden lærer etter hvert.
- **Internt konsistent**. Dette betyr at ingen del av KS må stride mot en annen del. Dette kan bare sjekkes ved hjelp av inspeksjon. Ryddig oppstilling av alle krav – for eksempel ved å gi alle krav nummer og

undernummer og sette de opp i tabeller - vil hjelpe godt. I tillegg må vi passe på å alltid bruke samme ord for samme ting.

- **Indikasjon og rangering ut fra viktighet og stabilitet**. Begge deler er viktige å vite.
    - Viktighet – hva skal vi begynne med, hvor er det kritisk hvis det går feil
    - Stabilitet – hvor sannsynlig er det at kravet endrer seg. Høy sannsynlighet betyr at vi ikke bør begynne å implementere dette, men det vil være lurt å begynne med en liten, enkel prototyp for å komme videre.

  Den jobben det er å få på plass disse indikatorene vil også være nyttige for kunden slik at han kan bedre forstå sine egne prioriteringer og hvor han har behov for å undersøke videre.
- **Verifiserbare**. Det må altså være mulig å teste at vi har oppfylt kravene. Dette sette betingelser til hvordan vi kan beskrive et krav. Det vil være gunstig å tenke "Hvordan kan vi teste dette?" når vi skriver kravene. Dette er koblinga i V-modellen mellom KS og FAT/SAT. Noen krav er vanskeligere å beskrive på en testbar måte enn andre. Hva betyr for eksempel "systemet skal være lett å bruke" eller "systemet skal gi rask respons?"
- **Modifiserbart**. Siden kunden med stor sannsynelighet vil forandre mening underveis er det viktig at det er mulig å endre krav uten "store katastrofer" – det vil si vi må kunne beholde alle de egenskapene vi har listet opp så langt. Felles begrepsapparat, god struktur på kravene og fravær av redundans er viktige faktorer her.
- **Sporbarhet**. På dette nivået betyr sporbarhet av det er mulig å finne ut **hvorfor** et krav finnes. To spørsmål er viktige:
    - Hvor kommer kravet fra? Er det et originalt krav, er det utledet av - en konsekvens av – ett eller flere andre krav eller er det ikke et krav i det hele tatt – bare "det hadde vært fint hvis?"
    - Hva er årsakene til kravet, hvorfor trenger vi dette og hva er målet med dette kravet?

En måte å organisere kravene på er vist i **fig 9.6. og fig. 9.7**. Krav kan bli organisert langs flere akser:

- System mode: ulike modi kan ha ulike krav – trening, drift, avansert bruk, enkel bruk.
- Brukergruppe: ulike brukergrupper kan ha forskjellige behov og derfor forskjellige krav til systemet.
- Responstype: for eksempel data inn, data ut, spørre etter data
- Hvis vi ikke har noe annen måte å organisere på, vil vi vanligvis bruke et eller annet funksjonshierarki.

I fig. 9.7 har vi brukt brukerklasser for å organisere funksjonelle krav. Hvert funksjonelt krav kan beskrives som gitt i standarden i vedlegg C. Har foreslår de følgende beskrivelse:

- Innledning: hva er hensikten med denne funksjonaliteten, hva ønsker brukerne å oppnå?
- Input: hva slags input forventer denne funksjonen?
- Hva gjør funksjonen med dataene? Dette kan for eksempel være en beskrivelse av algoritmene som blir brukt, hva slags sjekking som blir gjort med dataene og hva slags tilstand systemet er i etterpå.
- Output: hva gir funksjonen oss? Viktig å beskrive reaksjonen både ved riktige og ved feil data. Hva slags feilmeldinger vi skal gi, osv

KS dokumentet

Det er mange måter å skrive en KS på – fra et dokument i naturlig språk, for eksempel norsk eller engelsk - til svært formelle metoder som for eksempel VDM (Vienna Development Method). Vi skal i det etterfølgende se på følgende metoder:

- Naturlig språk – vanligvis norsk eller engelsk
- ER diagrammer – brukt mye i applikasjoner med databaser
- Tilstandsdiagrammer
- SADT diagrammer – Strukturert Analyse og Design Teknikk

Vanligvis vil det ikke være noe enten – eller. De fleste KS dokumenter vil inneholde innslag av flere metoder, rett og slett fordi det er ting som er enklest å si ett språk og andre ting det er lettest å si i et annet. Legg merke til at **use case diagrammer alene ikke er KS** –de er bare et middel til å lage en KS.

Uansett teknikk er det viktig at KS er på en slik form at de kan leses av kunden. Dette legger begrensninger på hva slags språk og formalismer vi kan bruke. Det er viktig at kundene kan lese KS for å kontrollere at:

- Vi har en felles forståelse av det systemet vi skal lage
- Alle krav er formulert slik som kunden vil ha de
- Kunden - alene eller sammen med uviklerne - kan inspisere KS dokumentet og godkjenne det.

Naturlig språk

Dette er den naturlige måten å beskrive KS på. Dette gjelder ofte for kundene og svært ofte også for utviklerne. De formelle språkene som finnes er stort sett kjent hos noen få spesialister – for eksempel spesialister på formelle metoder. Uansett er det bare unntaksvis at kundene kjenner de. Vi må passe på en del problemer som er knyttet til bruk av naturlig språk i en KS:

- **Støy**. Naturlig språk er svært fleksibelt. Det er derfor lett å beskrive samme tingen på mange forskjellige måter. I tillegg er det lett å fylle på med ting som ikke har noe med KS å gjøre.
- **Taushet**. Det er lett å glemme å skrive ting som er selvsagt for den som skriver det, selv om det ikke nødvendigvis vil være selvsagt for den som skal lese det. I en formell notasjon er man nødt til å også ha med det selvsagte – for eksempel definisjoner av alle parametere eller alle betingelser på en variabel.
- **Motsigelser**. Hvis vi skal beskrive samme tingen mer enn en gang, er det lett å bruke nye ord og begreper. Dette kan føre til subtile selvmotsigelser, noe som er vanskelig å oppdage senere. For eksempel ”x skal være større enn fire” ett sted og ”x skal være større eller lik fire” et annet sted. Det kan være greit å huske at det gjelder forskjellige regler for det å skrive en roman og det å skrive en KS.
- **Flertydighet**. Naturlig språk – særlig den muntlige varianten – er av natur flertydig. Konteksten vil vanligvis gi den informasjonen man trenger for å kunne tyde setningen. Slik er det imidlertid ikke alltid i skriftlig materiale.
- **Forover referanser**. Dette er særlig alvorlig i store dokumenter. Ofte er det en måte å utsette en beslutning eller en beskrivelse på og det er ikke sjelden at man glemmer å ta beslutningen eller ”skrive det siden”. Ofte er det et strukturproblem i dokumentet som forårsaker slike problemer.
- **Forhåpninger**. Noen ganger er en KS bygd mer på håp og tro enn på realiteter. ”Det hadde vært fint om…” er populært. I andre tilfeller er det beskrevet løsninger som er så avanserte at det vil være vanskelig å realisere dem med dagens teknologi – i hvert fall til en overkommelig pris og innenfor realistiske tidsrammer.

Det er imidlertid viktig å huske at naturlig språk er den representasjonen det er lettest å utveksle mellom mennesker eller organisasjoner og den som flest behersker. Noen har eksperimentert med følgende løsning på problemet:

1. Beskriv og analyser problemet i et formelt språk – for eksempel VDM, Z eller OBJ.
2. Skriv dokumentet om til naturlig språk og vis det til kunden.
3. Alle endringer tas over til det formelle dokumentet slik at vi er sikker på at KS er konsistent, entydig osv.
4. Gjenta fra punkt 2 viss nødvendig.

Det er neppe praktisk å gjøre dette manuelt for større dokumenter eller systemer. Et verktøy ville hjelpe mye, men vi vil kunne få problemer som er vanskelig å oppdage hvis det er feil i verktøyet. Et mer praktisk alternativ er å støtte/utfylle beskrivelsen i naturligspråk med diagrammer av forskjellig type – for eksempel ER diagrammer, tilstandsdiagrammer eller SADT diagrammer. Vi vil kort se på disse tre diagramteknikkene.

ER diagrammer

Et ER diagram er en semantisk struktur. Det finnes fem modellbegreper:

- Entitet – et objekt. Dette kan være både abstrakte og konkrete objekter.
- Entitetstype – type for en entitet/objekt
- Attributtverdi – innformasjon som helt eller delvis beskriver en entitet.
- Attributt – type for et sett av attributtverdier
- Relasjon – en assosiasjon mellom to eller flere entiteter

Vis eksempel i fig. 9.11 og kommenter hva vi har av entitetstyper – for eksempel ”member” – relasjon – for eksempel ”borrow” og attributter – for eksempel ”name” og ”identification”.

ER diagrammer ble opprinelig laget for databasemodellering, men er nå en del av flere teknikker som bruker strukturert analyse. Det er en grei måte å forstå hvordan begreper henger sammen, men man kan like gjerne bruke klassediagrammer for dette formålet.

Tilstandsdiagrammer

Tilstandsdiagrammer viser:

- Hvilke tilstander en enhet har
- Hva slags input som er lovelig i en gitt tilstand
- Hvilken tilstand input fører til

Tilstandsdiagrammer har et bredt bruksområde innenfor mange fagfelt og er en effektiv måte for å beskrive forandringer i et element forårsaket av stimuli. Diagrammene kan brukes til å beskrive mennesker, maskiner, økonomi og alt annet innimellom. **Vis fig. 9.12** og kommenter.

For å lage et tilstandsdiagram begynner man med det man skal studere tilstanden til og stiller to spørsmål:

- Hva slags input/stimuli kan denne enheten få nå?
- Hvilken tilstand får den – hva skjer med enheten - for denne inputen?

Svaret på det andre spørsmålet er en ny tilstand og man kan da gjenta spørsmålene for den nye tilstanden. Prosessen vil gi god oversikt og forståelse for det objektet vi studerer.

SADT diagrammer

SADT står for "Strukturert Analyse og Design Teknikker". Dette er primært en diagramteknikk. Hvert enkelt diagramsymbol har en presis mening – det har altså en streng syntaks og semantikk. Vi skal ikke holde noe SADT kurs, men kort se på de viktigste notasjonselementene. **Vis fig 9.13 og fig 9.15**.

Ikke-funksjonelle krav

Ikke-funksjonelle krav er krav som angår hvordan et system løser en oppgave. Eksempler på slike krav er krav til pålitelighet, responstid og brukervennlighet. Utfordringen ligger i å formulere disse kravene slik at de er testbare og – hvis mulig – sporbare. Mange ikke-funksjonelle krav er ikke sporbare ned i produktet – de er bare sporbare ned i prosessen. Selv om vi ikke kan identifisere hvor i produktet de er realisert, kan vi peke på hvor i prosessen vi har tatt hensyn til kravene.

Det finnes en enkel prosess – definert av Tom Gilb – som heter "Management by Objective". Denne prosessen vil hjelpe til med å definere ikke-funksjonelle krav på en testbar måte.

1. Start med et ikke-funksjonelt krav A
2. Spør "Hva mener du med A?"
3. Dette vil gi en ny tolking av A eller en dekomponering av A i komponentene a1, a2, …
4. Hvis den nye tolkinga av A eller komponentene a1, a2, … er målbare, så er vi ferdig. Ellers gjentar vi prosessen fra tinn 2 med den nye tolkinga av A eller med hver av komponentene a1, a2, …

Vi vil se på et eksempel – brukervennlighet.

- Hva mener du med "brukervennlig"
  - Lett å lære
  - Lett å bruke
- Hva mener du med "lett å lære"
  - Skal kunne gjøre de vanligste jobbene mine etter et todagers kurs.
- Hva er de vanligste jobbene dine
  - Oppgavene X, Y og Z
- Hva mener du med "lett å bruke"
  - Skal kunne gjøre de vanligste jobbene mine like raskt som jeg gjør de nå
- Hvor mye tid bruker du nå
  - Vet ikke, men det kan vi vel finne ut.

Videre arbeid nå vil starte med å samle data om hvor lang tid personene nå bruker på de tre oppgavene X, Y og Z. Dette kan vi gjøre ved å overvåke personer i bedriften og se hvor lang tid de bruker hver gang de gjør jobben. Vi kan bruke største og minste tid – et intervall – eller vi kan beregne middelverdi og standardavvik Når vi har dette klart kan vi definere testen for brukervennlighet på følgende måte:

1. Få kunden til å velge ut for eksempel tre brukere. Disse vil få et todagers kurs i å bruke systemet
2. Disse brukerne skal deretter gjøre jobbene X, Y og Z i hver sin rekkefølge. For eksempel kan bruker 1 gjøre X, Y, Z, X, Y, Z, bruker 2 gjøre Z, Y, X, Z, X, Y og bruker 3 gjøre Z, Z, Y, Y, X, X. Registrer tida hver bruker trenger på oppgaven.
3. Hvis alle jobbtidene faller i det tidligere målte intervallet for hver jobb, er brukervennligheten godkjent. Ellers må vi identifisere årsakene til problemet, rette det og deretter gjenta testen med tre nye – hvis mulig – brukere.

Legg merke til at dette ikke er en test av datasystemet alene. Testen inkluderer også opplæringsmaterialet og selve kurset. Dersom vi vil unngå det, kunne vi for eksempel erstatte kurset med at hver bruker får en "fomle- og prøvefase" av en gitt lengde før vi begynner å teste – samle data.

Rammeverk

Det finnes fire perspektiver som vi kan bruke på et datasystem:

- **Foretningsperspektiv**. Hvordan driver vi forretning? Hvis man ikke kan knytte prosjektet – og produktet – til forretningsdriften i firmaet, vil vi før eller siden få problemer med prioritet og oppmerksomhet. Dessuten gir dette innsikt i en god del underforståtte – implisitte – krav og/eller forventninger.
- **Informasjonsperspektiv**. Her ser vi på den informasjonen som flyter inn i og ut av systemet og som er nødvendig for forretningsdriften. Her må vi også se på hva som skal være med i datasystemet og hva vi vil realisere utenfor i form av manuelle rutiner. Vi trekker altså systemgrensene her. Dette er viktig for senere brukerdeltakelse og kommunikasjon.
- **Funksjonsperspektiv**. Hva skal systemet gjøre? Dette er det eksterne synet – systemet sett utenfra. Her må vi involvere de som skal bruke systemet – sluttbrukerne, ikke deres sjefer. Legg merke til at det godt kan – og ofte vil – være forskjell på kunde og sluttbrukere. Vi har to typer av brukere:
    - De som bruker datasystemet - primærbrukere
    - De som bruker de dataene som kommer ut av datasystemet – sekundærbrukere

  Begge parter må involveres, fordi de ser på systemet fra hvert sitt ståsted og vil kunne ha ulike behov.
- **Implementeringsperspektiv**. Dette er det interne systemet – systemet sett innenfra. Hovedproblemet her er å realisere den funksjonaliteten som er identifisert i funksjonsperspektivet. De som er involvert er programvare eksperter – systemanalytikere, designere, kodere, testere og prosjektledere.

Disse perspektivene vil kreve ulike teknikker og kunnskap. Det er neppe mulig å gjøre dette alene og ved å bruke bare en metode eller teknikk. Vi kan ikke lære dere alt dere trenger i dette faget, men nøyer oss med den EDB-faglige biten. Det dere imidlertid må lære er at det finnes flere typer problemer med å lage et effektivt system for en bruker – ikke bare det å programmere en ferdig tygget løsning.

02-04 – Fowler kapittel 3

Et use case er et uformelt scenario beskrevet i en uformell notasjon. Når man tenker på all den formalismen som er utviklet for å dokumentere kundekrav, er det nærmest underlig at use case fungerer så godt som de egentlig gjør. Siden use case er en type scenario, vil vi først se på hva et scenario er:
Et scenario er – i denne forbindelsen - en sekvens av trinn / handlige som beskriver en interaksjon mellom en bruker og et system.

Tilsvarende kan vi si at:
Et use case er et sett av scenarier som er bundet sammen av et felles brukermål.

Ofte vil et use case ha en hovedversjon – hovedscenario – der alt går som planlagt i det enkle tilfelle. I tillegg vil det ha alternative sekvenser hvis ett eller annet går galt. I tillegg til slike spesielle sekvenser kan du, hvis du vil – legge til tekst som sier noe om betingelsene for at dette use caset er relevant. Pass på å ikke overbelaste use case med notasjon. Det er bare det som er viktig for kommunikasjon med brukerne og det som er viktig for å forstå brukernes behov som skal være med.

Mengden av detaljer i et use case avhenger av risikoen. Jo større risikoen er desto mer detaljer trenger du. Den risikoen vi ser på her er bestemt av konsekvensene hvis vi misforstår dette spesielle brukerkravet.

Det finns flere måter å dokumentere et use case på og vi skal se litt på hver av disse:
- Vanlig, fri format tekst – beskrive en historie med fokus på hva som skjer.
- Strukturert tekst – fast, tabularisk oppsett
- Diagram

Fri format tekst

Dette er en prosabeskrivelse – muligens med noe struktur – om hva som skal skje når man ønsker å bruke systemet til å utføre en beskrevet oppgave. Ofte vil den ha karakter av en spørsmål og svar sesjon der brukeren svarer på spørsmål fra den som skal lage use caset.

Diagram

Ta utgangspunkt i **fig 3-2 hos Fowler**. Viktige elementer i diagrammet er:
- Aktør – en som gjør noe eller initierer en handling. Det vil ofte, men ikke alltid, være en person. UML skiller imidlertid ikke mellom personer og andre aktører, som for eksempel et annet datasystem. I fig 3-2 er for eksempel ”Accounting System” en aktør som ikke er et menneske. Det kan være lurt å starte med å identifisere alle aktørene og deretter se på hva enkelt aktør gjør. En viktig kilde til å identifisere alle use case er å se etter hvilke eksterne hendelser som skjer og hvordan systemet skal reagere på de.
- Include relasjoner, som viser aktiviteter som blir brukt flere steder, men som vi vil slippe å beskrive flere ganger. Vi skiller ut disse aktivitetene i en egen use case og inkluderer det i de use casene der vi trenger den. I fig. 3-2 er ”Valuation” et eksempel på dette.
- Generalisering som viser at et use case likner på et annet, men gjør litt mer. Dette er en måte å beskrive alternative scenarier på. I fig 3-2 har vi denne situasjonen i ”Capture deal”. Dersom kredittgrensene er overskredet vil vi få en spesialbehandling. Dette er stadig ”Capture deal” use caset, men i en annen form. Et slikt use case skal altså stadig angå samme oppgave.

I **fig 3-3** hos Fowler ser vi et nytt diagramelement – ”extend” eller utvid.. Dette likner på generalisering, men use caset som skal ha en utvidelse må ha et utvidelsespunkt. Dette må være erklært på forhånd. Det er fullt mulig å ha flere utvidelsespunkter.

Følgende tommelfingerregler kan være greie å bruke:
- Bruk ”include” når man begynner å gjenta ting i to eller flere use case.
- Bruk generalisering når du beskriver en variant av den normale oppførselen.
- Bruk ”extend” når du beskriver en variant og ønsker å ha mer kontroll over den. Det er dette vi oppnår ved å ha definerte utvidelsespunkter

Det er mulig å utvikle use case i mer detaljer etter hvert som vi får bedre forståelse. Det er også mulig å splitte opp et use case hvis vi synes det blir for stort og uhåndterlig. Noen ganger kan det være praktisk å gjøre dette ved å organisere flere samhørende use case i pakker. I tillegg er det ingenting i veien for å legge på forklarende tekst i den grad det er nødvendig.

Uansett er det viktig å huske på at use case er et hjelpemiddel i kommunikasjon med brukerne – det er ikke en metode for (overordna) design.

Strukturert tekst

Mange – særlig folk som kjenner et domene godt og som har mye erfaring i systemutvikling – synes at use case diagrammer er litt for upresise. Særlig er det alvorlig at det ikke er mulig å vise sekvenser av hendelser, bare selve handlingene. Av den grunn er det mange som foretrekker tekstlige use case.

Det finnes ikke noe i UML som definerer maler eller strukturerte tekst use case. Det kan være lurt å starte med use case diagrammene for *deretter* å lage strukturerte tekstlige use case som inneholder med detaljer, sekvenser, triggere osv.

02-07 kapitel 12

Vi kan se objektorientering fra flere sider. De viktigste for oss er:

- **Modellering.** Viktige egenskaper er at hvert objekt har
    - En identitet som skiller det fra andre objekter
    - Egenskaper som kan finnes ved å se på / studere objektet.

  Forskjellige objekter ligger på forskjellige steder i hukommelsen. Et objekt kan betraktes som en abstrakt datatype (ADT) som består av et sett med data og operasjoner for å manipulere de. Det skal i prinsippet bare være mulig å endre disse dataene via de definerte operasjonene. Med dette perspektivet kan vi sette:
    - **objekt = identitet + data + operasjoner**.

  eller
    - **objekt = identitet + tilstand + oppførsel**

  Modelleringen blir da å finne og beskrive de enkelte delene av ”likninga”.
- **Utviklerne – bruker O-O**: Et objekt er en abstraksjon og en innkapsling av data og deres operasjoner. Imidlertid vil ikke alle O-O språk kreve abstrakte datatyper og et objekt behøver ikke være en abstrakt datatype. Et eksempel på en abstrakt datatype kan være en kø. Den inneholder et array for køen og to eksternt tilgjengelige operasjoner – sett inn i køen og ta ut av køen. I tillegg vil den ha interne operasjoner som for eksempel sjekk for full eller tom kø.

  Et objekt kan egentlig være hva som helst. Data abstraksjoner og objekter er på mange måter ortogonale – uavhengige – begreper. Mange vil kalle språk som bare har abstrakte datatyper for objektbaserte språk, men de vil reserver O-O for språk som har arving.
- **Implementering – lage O-O**. Her er et objekt en sammenhengende struktur i hukommelsen – en samling av data og kode. Objekter kan ha mindre objekter inne i seg – det er en rekursiv struktur. Det laveste nivået består av heltall, boolske variable osv. Det finnes mange interessante problemer med å realisere O-O, men vi går ikke inn på de her.
- **Formelt sett**: Et objekt er en tilstandsmaskin med et sett av tilstandsvariable og et sett av funksjoner som manipulerer disse tilstandsvariablene.

Oppførsel hos objekter er knyttet til endringer i tilstanden til objektet – dvs. til endring i verdien til en eller flere variable. Disse variable er attributter til objektet. En dør kan for eksempel være et objekt. Den kan være i flere tilstander som for eksempel ”åpen” / ”lukket”, ”låst” / ”ulåst”. Legg merke til at ikke alle kombinasjoner er realistiske. ”Lukket” og ”ulåst” er OK, mens ”åpen” og ”låst” ikke nødvendigvis er like greit. I tillegg vil vi ha betingelser for hvilke overganger som er lovelige. For eksempel vil kan det bare være lov å gå fra ”lukket” til ”åpen” viss vi allerede er i tilstanden ”ulåst”. Dette dokumenteres best i et enkelt tilstandsdiagram.

Vi vil nesten alltid realisere et system med flere klasser. Derfor er forbindelsene mellom klassene viktig. Viktige forbindelser er:

- **Spesialisering**. For eksempel vil ”bord” og ”stol” være spesialiseringer av klassen ”møbler”. I ER diagrammer vil vi tilsvarende ha en ”is-a” relasjon: ”A table **is-a** piece of furniture”.
- Det inverse av spesialisering er **generalisering**. ”Møbler” er en generalisering av ”stol”, ”bord”, ”sofa” osv.

Vis fig 12.2 som viser et spesialisering / generalisering hierarki. Dette er et diagram sin viser enkel arving og det er derfor et tre. Hvis vi har multippel arving vil vi i stedet få en rettet, asyklisk graf (DAG). Objekthierarkiet kan også betraktes som et typehierarki.

O-O analyse og design – Notasjon

Vi trenger tre typer diagrammer for å gjøre O-O analyse og design:

- **Klassediagrammer** for å vise dekomponeringen av systemet. Klassediagrammene har noder – klasser – og kanter – relasjoner mellom klassene. Vi kan få fram mer informasjon ved å legge til tekst på en eller flere noder og kanter.
- **Tilstandsdiagrammer** for å vise dynamisk oppførsel. Her bruker de fleste O-O metoder en eller annen form for tilstandsmaskin. Nodene i diagrammet viser lovelige / mulige tilstander, mens kantene viser lovelige overganger fra en tilstand til en annen.
- **Interaksjonsdiagram** for å vise sekvensen av meldinger mellom objekter. Vi har to typer:
    - Sekvensdiagrammer, som legger vekt på rekkefølge / tidsaspekter
    - Samarbeidsdiagrammer, som legger vekt på objektene og deres relasjoner for å implementere en gitt interaksjon

Klassediagram

Klassediagrammene viser den statiske strukturen til systemet. En klasse trenger tre typer info:

- Navn på klassen
- En liste av attributter
- En liste av tjenester som klassen tilbyr

Vi tenger ikke ha med alle attributtene eller tjenestene i starten av fasen – bare de vi synes er viktige her og nå. Før vi kan begynne å implementere noe må de imidlertid være på plass.

I tillegg til generalisering og spesialisering som vi allerede har sett, er følgende relasjoner viktige:

- **Assosiasjoner** – med og uten assosiasjonsklasser. **Vis fig. 12.5**. I UML vil en assosiasjon forbinde to eller flere klasser. Dette vises med ei heltrukket linje. Vi kan – viss vi vil – sette navn på assosiasjonen. I tillegg kna vi legge på retning (Fylt trekant) og multiplisitet. For eksempel står det her at en klient kan tilhøre ett eller flere biblioteker og at et bibliotek kan ha null eller flere klienter.

  Når vi trenger mer info kan vi legge til en assosiasjonsklasse. Dette er en vanlig klasse med attributter og operasjoner. I eksemplet i fig 12.5 vil det være praktisk å ha attributt medlem-ID og operasjon ”bli medlem”.
- **Aggregater**. Må skilles fra sammensetning – ”composition”. Aggregater vises med **ikke-fylte ruter** symboler. Aggregater er ”is-part-of” relasjoner. Hver klasse som deltar i et aggregat kan tilhøre flere helheter.
- **Sammensetning – ”composition”** – er også ”is-part-of” relasjoner, men det som er modellert kan bare tilhøre **ett hele**. Sammensetning vises med en **fylt ruter** symbol. Når dette hele forsvinner også delene mens dette ikke er tilfelle for aggregater. Sånn setter sammen setning et sterkere begrep enn aggregater. **Vis fig 12.6** i boka.

For å forklare forskjellen vil vi se på følgende figur – vis **fig. 6-6 hos Fowler**. Denne figuren viser at i denne modellen kan et punkt kan bare tilhøre en sirkel eller et polygon av gangen. Beskrivelsesstile – ”style” – kan imidlertid høre til flere sirkler og polygoner og kan leve sitt eget liv selv om vi fjerner alle sirkler og polygoner.

Tilstandsdiagram

Et objekt vil ha et livsløp – det blir skapt, det blir oppdatert en del ganger (muligens null ganger) og tilslutt dør det. Et tilstandsdiagram er en god metode for å dokumentere et slikt livsløp. UML legger en del ekstra info til de tradisjonelle tilstandsdiagrammene for at de skal være enkle å bruke i modellering. Vi vil se på følgende mekanismer:

- Tilstander har lokale variable. Dette gjør diagrammet enklere enn om vi hadde all info i sele tilstandsbeskrivelsen. Et eksempel kan være en person som låner bøker på et bibliotek. Etter som han tar ut bøker kunne vi tenke oss at han gikk fra tilstand ”ikke lånt bøker” til tilstanden ”lånt en bok”, deretter til tilstanden ”lånt to bøker” osv. Dette vil kunne bli et stort - og ikke spesielt interessant – diagram. **Vis hvordan dette ville bli**

seende ut på tavla. Alternativet er **fig. 12.7 i boka**, som er mye enklere, uten å gi slipp på noe info.

- Som for alle andre tilstandsmaskiner vil det også i UML være slik at input vil kunne forårsake overgang fra en tilstand til en annen. I forbindelse med overgangen vil man kunne ha handlinger – f.eks. å oppdatere en variabel eller å gi output.
- Tilstandsdiagrammer kan fort bli store og vanskelige å lese. Derfor er det innført en metode / notasjon for å lage litt struktur i de.

Tilstander tegnes som rektangler, men med **runde hjørner**. I tillegg til alle de virkelige tilstandene finnes det to pseudotilstander – start og stopp. Et objekt kan ikke være i noen av disse tilstandene. I starttilstanden er ikke objektet instansiert og i sluttilstanden er det borte.

Overgangene mellom tilstander er markert med:

- Hendelsen som forårsaker overgangen – "Lån bok". Denne må alltid være tilstede.
- Aksjoner – N = N + 1. Aksjonen er skilt fra hendelsen med en "/". For eksempel kan vi sette "lån bok / N = N+1"
- Betingelser for at overgangen skal være lovelig – med notasjonen <overgang> [N = 0]
- Output som følger av overgangen

For å lettere bevare oversikten kan det være lurt å ikke ha med alle detaljer hele tiden, men bare de detaljene vi trenger for øyeblikket. **Fig 12.8** viser hvordan vi kan organisere tilstandsdiagrammet med trinnvis forfining. Dette gjør det enklere å lage tilstandsdiagrammet i flere trinn etter hvert som vi får mer info, bedre forståelse eller trenger en mer komplett beskrivelse av endringene i tilstand.

Sekvensdiagram

Objekter kommuniserer ved å sende meldinger til hverandre. Dette blir i UML viset med sekvensdiagrammer. Disse er omtrent det samme som MSCer i telekom-verdenen.

- Objekter blir lagt horisontalt, vanligvis slik at de som ligger lengst til venstre blir aktivert først.
- Meldinger fra et objekt til et annet vises som horisontale linjer med piler - **fra** det objektet som sendte dem, **til** det objektet som mottar dem.
- Tiden vises langs den vertikale aksen slik at en hendelse som ligger høyre i diagrammet hender før den som ligger lavere.

Vis fig. 12.9 og kommenter. Nummereringa som er vist i figuren er ikke nødvendig. Det finnes mange andre mekanismer i sekvensdiagrammer i UML enn de som vises her. F.eks. kan vi skille mellom synkrone og asynkrone meldinger, vi kna vise hvordan objekter starter og dør og vi kan vise iterasjoner. Se kapitel 5 i Fowler for mer info. Uansett er hensikten å vise hvordan objekter utveksler info. Erfaring viser at de fleste forstår sekvensdiagrammer etter litt enkelt forklaring og de er derfor effektive når vi skal diskutere med brukere og kunder.

Det er mulig å gjøre sekvensdiagrammer komplekse, særlig ved å legge på mange betingelser og betinga hendelser. Dette ødelegger imidlertid hensikten med diagrammet som hjelpemiddel for kommunikasjon. Det er bedre / enklere å se på hvert diagram som en beskrivelse av et scenario og heller lage ett diagram for hvert scenario.

Siden use case også er scenarier vil det være mulig å gå fra use case til sekvensdiagrammer og bruke dette som en metode til å identifisere objekter / klasser og de meldingene vi trenger for ar klassene skal samarbeide.

Samarbeidsdiagrammer

På samme måter som sekvensdiagrammer er også samarbeidsdiagrammer en metode for å vise hvordan en gruppe objekter samarbeider for å realisere et scenario. Slike diagrammer kalles ofte også objektdiagrammer. Et samarbeidsdiagram består av:

- Noder som er enheter – ofte objekter
- Kanter mellom nodene. Disse kantene er nummerert for å vise rekkefølgen til hendelsene.

Man trenger ikke begge deler og hva man velger av samarbeidsdiagram og sekvensdiagram er ofte et spørsmål om smak og behag.

O-O analyse og design – Metoder

Viktige komponenter i analyse og design er:
- Identifiser objektene
- Bestem de attributtene og tjenestene objektene skal tilby
- Bestem relasjonene mellom de

Dette er ikke sen sekvens av oppgaver – de må løses iterativt. Resultatet er en statisk struktur av objekter og relasjoner mellom disse objektene. Livsløpet til et objekt kan beskrives i et tilstandsdiagram og hvordan de samarbeider, kan vises i et sekvensdiagram. De fleste måter å finne den nødvendige informasjonen bygger på lingvistiske begreper – f. eks at substantiver blir objekter og verb blir tjenester. Vi skal se litt på tre metoder. Alle tre har hvert sitt sett av retningslinjer – det er ikke algoritmer for problemløsning.

Hver metode er definert ved et sett av aktiviteter og et prosessdiagram som viser hvordan aktivitetene er relatert til hverandre. Alle har en iterativ analysefase, men de to siste metodene har ikke en iterativ designfase. At analysefasen er iterativ henger sammen med at det er her vi bygger opp vår forståelse av systemet – både hva det er og hva det skal gjøre.

Booch metoden

Vis fig 12.17 i boka. Det første gjennomløpet i modellen er analyseorientert mens de senere vil være designorienterte. Vi vil kommentere hver aktivitet kort:
- **Identifiser klasser og objekter**. Her skal vi avgrense problemet – hva er innefor datasystemet og hva er utenfor – og lage en første dekomponering.
    - Ved analyse: finn gode abstraksjoner, basert på vår forståelse av applikasjonsdomenet.
    - Ved design. Legg til klasser fra løsningsdomenet. Her vil vi også lage en fortegnelse over alle abstraksjonen (data dictionary) med komplette definisjoner, slik at vi er enige om hva de begrepa vi bruker betyr.
- **Identifiser semantikken og attributtene** til hver enkelt abstraksjon – hva gjør den med hva? Denne informasjonene kommer fra scenarier – f.eks. use case. Ut fra denne prosessen får vi en konsistent og komplett lite av ansvar / operasjoner for hver enkelt abstraksjon. Denne informasjonen blir også satt inn i data dictionary. Vår forståelse av use case blir videreutviklet og koblet sammen med abstraksjonene i sekvensdiagrammer eller samarbeidsdiagrammer.
- **Identifiser relasjoner mellom objektene**. Her må vi igjen skille mellom analyse og design.
    - Analyse: finn relasjoner mellom abstraksjonene.
    - Design. Vi tar taktiske beslutninger om arv mellom klassene og instanser for objektene. Dette gir oss klassediagrammer, samarbeidsdiagrammer og moduldiagrammer. Moduldiagrammene viser modulstrukturen til systemet.
- **Identifiser grensesnitt og implementasjon**. Her velger vi algoritmer og eventuelle nye klasser som vi trenger av løsningsmessige årsaker.

OMT metoden

Vis fig 12.18 i boka. Denne metoden har et klart skille mellom analyse og design som Booch metode mangler. Metoden er implementasjonsorientert i langt sterkere grad enn Booch metode. Skrittet fra analyse til design er stort og ikke enkelt. Analysefasen er iterativ, mens designfasen – i prinsippet - ikke skal være det.
- Analyse
    - Finn objektmodell. Dette er den statiske strukturen i systemet. Identifiser objekter, deres relasjoner og attributter. Lag klassediagram og data dictionary.
    - Finn dynamisk modell. Dette er oppførselen til systemet. Lag sekvensdiagrammer.
    - Finn en funksjonell modell. Denne beskrives med et data flyt diagram. **Vis fig. 11.8** fra boka.

  Disse tre trinna mp gjentaes – det er en iterativ prosess.
- Design

- System design. Her lager eller velger vi en arkitektur for systemet. Dette inkluderer oppdeling i subsystemer, plassering i prosessor (viss flere) osv.
- Objekt design. Her lager vi de algoritmene vi trenger for hvert enkelt objekt.

Fusjonsmetoden

Vis fig 12.19 i boka. Som for OMT er det skilt klart mellom analyse og design, men i motsetning til OMT er designdelen her langt mer detaljert. Som for OMT er analysefasen iterativ.

- Analysefase
  - Finn objektmodell – klassediagrammene. Som før blir dette dokumentert i data dictionary.
  - Finn grensesnittmodell. Dette viser den dynamiske oppførselen. Den innholder en livssyklusmodell for hver klasse i tillegg til regulære (analytiske) uttrykk for tilstandsmaskinen. I tillegg er det en spesifikasjon av semantikken for hver operasjon ved hjelp av pre- og postbetingelser. Prebetingelse – hva er sant før operasjonen blir utført – og postbetingelse – hva er sant etter at operasjonen er utført.
- Designfase
  - Lag objektsinteraksjonsdiagram. Dette er omtrent det samme som samarbeidsdiagrammer.
  - Lag synlighetsgraf. Denne beskriver hvordan kommunikasjonen mellom objekter blir realisert. Vi finner ut hvilke andre objekter hvert enkelt objekt må kommunisere med. Synlighetsgrafen beskriver hvordan dette blir gjort.
  - Utvikl klassebeskrivelser. Denne bygger på det vi laget i de to foregående aktivitetene – objektinteraksjonsdiagrammer og synlighetsgraf.
  - Lag arvingsgraf. Arvingsgrafen lages ut fra klassebeskrivelsene vi allerede har laget.

Fusjonsmetoden legger mye mer vekt på designfasen enn de to andre metodene. At analysemetoden er så enkelt og kort betyr at metoden er avhengig av en ferdig, komplett kravspesifikasjon.

Noen generelle kommentarer

Vi ser det er stort spenn i graden av formalisme – fra Booch metode som er meget uformell til Fusjonsmetoden som er svært formell og matematisk. OMT metoden ligger litt midt i mellom.

Det er en svakhet og en styrke for O-O metodikken at det er liten forskjell mellom analyse og design. Ikke alle er imidlertid enige i dette. Det er helt klart at analyse og design har forskjellige hensikter:

- Analysen er problemorientert og skal gi en forståelse av domenet og problemet vi skal løse
- Design er løsningsorientert og skal dekomponere løsningen i moduler – delløsninger – og bestemme hvordan disse skal samarbeide.

Imidlertid vil det alltid være behov for å revurdere eller oppdatere vår forståelse av problemet etter som vi lager en design og blir tvunget til å ta flere og flere beslutninger. Det kna være nyttig å tenke på veien fra problem til løsning som ei rett linje, der O-O analyse **stort sett** er nær problemet mens design **stort sett** er nær løsningen. I de fleste tilfeller vil det kunne være en stor grad av overlapp på midten.

Erfaringene med O-O er noe blandet. Det er ikke urimelig å hevde at O-O er en ”riktigere” måte å se problemer på. Tilsvarende er det heller ikke urimelig å hevde at verden ”består” av, eller kan modelleres som, objekter. Det viser seg imidlertid at de fleste mennesker tenker lettere ved hjelp av handlinger eller oppgaver. Derfor er erfaringene med O-O metoder blandede og resultatene av de få kontrollerte eksperimentene som er utført er langt fra entydige.

02-11 Fowler kapittel 4

Forfatteren skiller mellom tre perspektiver på klassediagrammer:

- **Konseptuelt syn**: klassediagrammet viser konsepter fra det domene vi ser på. Vi behøver ikke / skal ikke tenke noe særlig på hvordan vi skal implementere dette. Det er derfor også uanhengig av utviklingsspråk. Dette hører primært hjemme i analysefasen.

- **Spesifikasjonssynet**: fokus er på klassens grensesnittet mot omverdenen. Nøkkelen til god O-O programmering ligger i å programmere mot klassenes grensesnitt, ikke mot deres innmat – implementasjonen. Dette er viktig i designfasen.
- **Implementasjonssynet**: dette angår siste fase – implementasjon - og er det eneste der vi tenker realisering av innmaten i klassen.

Grensa mellom perspektivene er ikke skarp. Skille mellom konsept og spesifikasjonen er ikke alltid like viktig. Det er mest viktig å holde implementasjonsperspektivet adskilt fra de to andre. Vi vil se på en del egenskaper ved klasser i de tre perspektivene.

Assosiasjoner

- Konseptuelt. Konseptuelle relasjoner mellom klassene. Dette inkluderer navn og multiplisitet. Her pleier vi ikke å legge på navigeringsinfo – retninger på assosiasjonene.
- Spesifikasjon. Her representerer assosiasjonene ansvarsfordeling mellom klassene. Ansvar viser f.eks. hva en klasse er i stand til å gi fra seg av informasjon og hvem som oppdaterer hvilken info. Det ligger imidlertid ingen implementasjonsbeslutninger i dette. Hvordan det gjøres, må bestemmes seinere. Vi legger på piler for å vise hvem som har ansvar for hva. I **fig. 4.2** vil ”Order” ha en peker til ”Kunde”, mens det motsatte ikke vil være tilfelle. Fowler anbefaler at vi lar ”ingen pil” betyr at vi ikke har bestem oss ennå.
- Implementasjon. Her vil vi legge til pekere i de nødvendige retninger – avhengig av pilene - og erklæringer av pekere og metoder til å bearbeide relevante data.

Assosiasjoner kan ha navn. De øker imidlertid mengden av info i klassediagrammet. Det er derfor lurt å bare gi navn til de assosiasjonene der dette øker forståeligheten av diagrammet. Assosiasjoner er permanente forbindelser mellom to objekter.

Attributter

- Konseptuelt. Her markerer vi bare at attributtet finnes – ikke hva det er eller hvordan vi vil realisere det.
- Spesifikasjon. Når vi går over til spesifikasjonsnivået så sier attributtet at klassen har et ansvar for å gi deg sin verdi.
- Implementasjon. Nå er dette et felt i klassen som må ha type, synlighet ol.

Det er ingen forskjell mellom attributter og assosiasjoner på konseptuelt nivå. Forskjellene er bare viktige på spesifikasjons- og implementasjonsnivå. Attributter er gjerne variable med en enkelt verdi.

Operasjoner

- Konseptuelt. På dette nivået bør man bare bruke operasjoner til å vise de viktigste ansvarsområdene til en klasse.
- Spesifikasjon. ”Public” metoder i klassen. Det er vanligvis ikke nødvendig å vise metoder som bare manipulerer attributter, siden de er underforstått.
- Implementasjon. Her må man ta med alle metoder siden de nå skal realiseres.

Full UML-syntaks for en operasjon er
<synlighet> <navn> ”(” <parameterliste> ”):” <returtyper> ”{” <streng av egenskaper>”}”

- Synlighet: ”+” som betyr åpen / ”public”, ”#” som betyr beskyttet og ”-” som betyr privat.
- Navn er en tekststreng
- Parameterliste er ei liste av parametere, adskilt med komma. Hver parameter har syntaksen **<retning> <navn> ”:” <type> ”=” <default verdi>**. <retningsnavn> er ”in” – input - ”out” - output - eller ”inout” – både input og output. Default verdi er ”in”.
- Returtype - type på resultatet. Strengt tatt er det lov med flere typer, men det vil vanligvis bare være en.
- Egenskaper – liste av egenskaper, adskilt med komma ,som definerer egenskaper ved resultatet.

Eksempel - + balansePaa (dato: Dato): Penger. Denne operasjonen er synlig ”public”, har navnet balansePaa og parameterliste som bare består av en parameter – Dato som er en ”in” parameter og har type ”dato” og ingen defaultverdi.

Generalisering

- Konseptuelt. Her vil vi si at A er en subtype av B viss alle instanser av A også er instanser av B. Sånn sett er A en spesiell type av B. Alt vi kan si om A – assosiasjoner, atributter og operasjoner – er også sant for B.
- Spesifikasjon. Her er vi hovedsakelig interessert i grensesnitt. Generaliseringen sier hvis A er en subtype av B så vil alle grensesnitt som finnes i B også vil finnes i A- grensesnittet i subtypen må være konformt med grensesnittet i supertypen. Sagt på en annen måte: Viss jeg har laget kode for B så kan jeg i stedet sette inn kode for en vilkårlig subtype av B, f.eks. A.
- Implementasjon. På dette nivået er vi opptatt av arving. Hvis A er en subklasse av B så arver A alle Bs egenskaper.

Oppsummering

Hvordan skal vi bruke klassediagrammer? Ett av problemene er at det er en svært rik notasjon. Fowler har noen viktige tips:

- Ikke prøv å bruke all den notasjonen som er tilgjengelig. Bruk de avanserte konseptene bare hvis du virkelig trenger dem.
- Bruk riktig nivå – konseptuelt nivå i analysefasen – hva er problemet - spesifikasjonsnivået når du begynner å tenke programvare og implementasjonsnivået når du begynner å tenke på en spesiell måte å implementere løsningen på.
- Ikke prøv å modellere alt med en gang – hold deg til det som er viktig.
- Pass på å ikke kjøre deg fast i detaljer som bare angår implementasjonen for tidlig i prosessen.

02-14 Fowler kapittel 5

Sekvensdiagrammer

Viser til **fig 5-1 i Fowler**. Den notasjonen vi ikke har sett tidligere er:

- Meldinger og parametere, eventuelt med et iterasjonssymbol knyttet til seg. Iterasjoner har syntaksen ”*” ”[” <årsak til iterasjonen> ”]” <melding>
- Betingelser på meldingene. Dette har formatet ”[” <betingelse> ”]” <melding>
- Start nye objekter
- Kall til seg selv
- ”return” som viser retur av kontroll, men uten at en melding blir sendt. De kan for så vidt være med hver gang, men det er fornuftig å la være å tegne de inn hvis de ikke gir ekstra info eller gjør diagrammet lettere å lese.
- Sende en melding til seg selv – ”self call”.

Sekvensdiagrammer er også viktige for parallell programmering / parallelle prosesser (**se fig 5-2 i Fowler)**. Legg merke til det nye symbolet – en pil med bare ”halvt hode”. Dette er sekvensdiagrammets notasjon for å sende en asynkron melding. Her ser vi hvordan en transaksjonskoordinator starter to kontrollrutiner. Disse kan starte opp og kjøre i parallell. I tillegg sender transaksjonskoordinatoren spørsmål / meldinger til seg selv med forespørsel om alle kontrollsprosessene er ferdige – ”all done?” Når det siste sjekk-objektet er ferdig sender den en – stadig asynkron – beskjed om at transaksjonen er OK – meldingen ”beValid”.

En asynkron melding venter ikke på svar og blokkerer derfor ikke det objektet som sender den. Generelt kan en asynkron melding gjøre ett av tre:

- Starte en ny tråd (thread)
- Skape et nytt objekt - en ny instans
- Kommunisere med en tråd som allerede kjører.

Tråder (threads) kan være, men behøver ikke være egne prosesser. Dette avhenger av kjøresystemet og operativsystemet. F.eks. vil Linux lage en prosess for hver tråd, mens Solaris ikke vil gjøre det. Hvis det lages prosesser vil disse – avhengig av forholdet mellom antall tråder og antall tilgjengelige CPU'er – kjøre i parallell eller i kvasiparallell.

I fig 5-2 i Fowler ser vi også bruk av det å stoppe / drepe et objekt. Dette markers med et stort kryss ved enden av "livslinja" til objektet. Vi ser at når sjekk-prosessen er ferdige så avslutter de seg selv. I fig. 5-2 i Fowler avslutter de to sjekkprosessene seg selv etter å ha returnert info om at resultatet av sjekken var OK. I fig 5-3 i Fowler slutter en av sjekkene med at det er noe galt – melding "Fail" – og koordineringsprosessen avslutter da den andre sjekkeprosessen uten å vente på resultatet fra den.

I fig 5-3 hos Fowler er det også vist en nyttig detalj – tekst på siden av sekvensdiagrammet som i naturlig språk, beskriver hva som foregår. Brukt med måte – kommentarer kun til viktige ting – vil dette øke lesbarheten til diagrammet.

Samarbeidsdiagrammer

Vi har sett på samarbeidsdiagram før. Her skal vi bare se på to ting:

- En forbedret sekvensnummerering
- Annotering i tillegg til sekvensnummereringen.

Vi kan bruke alle de samme notasjonene som i sekvensdiagrammene. Iterasjon markeres ved å sette: "*" "[" <årsak til iterasjonen> "]" <melding>, mens meldinger som det er knyttet betingelser til skrives som: "[" <betingelse> "]" <melding>. (Se eksempler på dette i fig. 5-4 i Fowler.)

I fig 5-4 er det brukt den samme nummereringsmetoden som den vi har sett tidligere. UML har imidlertid innført desimalnummerering for å kunne vise hvilken operasjon som aktiverer hvilken annen operasjon (se fig 5-5 fra Fowler). Regelen for nummerering er som følger:

- Hver gang vi går ut fra et objekt går vi ett nivå ned i nummerering. Hvis vi går ut fra objekt N vil første melding få nummer N.1, neste få nummer N.2 osv.
- Vi kan finne kildemeldingen – og dermed også kildeobjektet - ved å suksessivt fjerne siste nivå i sekvensnummeret. F.eks. vil melding 1.1.2.2 komme fra melding 1.1.2 som igjen kommer fra melding 1.1.
- All notasjon som vi kan sette på et sekvensdiagram settes etter sekvensnummeret i samarbeidsdiagrammet.

CRC kort

Hensikten med både sekvensdiagrammer og samarbeidsdiagrammer er å vise hvordan klasser / objekter samarbeider for å realisere løsningen på en del av det identifiserte problemet. Husk at begge diagrammene vanligvis vil være scenarier – det blir for komplisert på vise alle alternativene i ett diagram. Hensikten med begge diagrammene er å realisere et scenario, beskrevet i et use case. Dersom vi ser på en enkelt klasse må vi i stedet bruke et tilstandsdiagram.

Ulempen med både sekvensdiagrammer og samarbeidsdiagram er at de er vanskelige å bruke for å diskutere alternative løsninger. Vi skal ikke ha diskutert lenge før diagrammet blir uleselig som følge av endringer, radering / utvisking og alternative streker og symboler. Vi kan forbedre denne situasjonen noe ved å bruke et dataverktøy, men det er vanligvis vanskelig for mer enn en til to personer å se på en dataskjerm samtidig – i hvert fall når vi er avhengig av at alle skal delta aktivt samtidig.

CRC-kortet ble innført for å kunne diskutere løsninger på en enkel og *inkluderende* måte. At den er *inkluderende* er viktig hvis vi skal få fatt i all relevant informasjon.

Klassenavn	
Ansvar 1	**Samarbeider med - a**
Ansvar 2	
Ansvar 3	**Samarbeider med – b**
Ansvar 4	**Samarbeider med – c**

Hva det betyr:

- **Ansvar**: Dette er en høynivå beskrivelse av hva klassen gjør eller hensikten med klassen.

- **Samarbeider med**: Dette er andre klasser som denne klassen må samarbeide med – få utført tjenester hos. Legg merke til at et ansvarsområde kan samarbeide med flere andre klasser.

Den videre prosessen består av følgende trinn:
1. Velg ut et use case
2. Definer / navngi klasser som vi trenger for å realisere det scenariet som use caset beskriver. Navn på klassene kan f.eks. hentes fra substantiver i use caset.
3. Se etter hvilke klasser du trenger å samarbeide med. Hvis de
   - allerede finnes så må navnet på "din" klasse inn som en samarbeidspartner
   - ikke finnes så må vi definere en eller flere nye klasser
4. Se etter ansvar som denne klassen må ivareta. Slike momenter vil komme fra under veis. Dette kan realiers med en kombinasjon av:
   - Metoder
   - Attributter

   Fokuser på ansvar på øverste nivå. Det er ikke lurt å fylle på med alle mulige lavnivå ansvar på dette stadium i prosessen. Husk at vi er på konsept eller spesifikasjonsnivå.
5. Fortsett med trinn 2 – 4 til vi har et sett av klasser som tilfredsstiller use caset. Det vil / bør bli færre og færre nye klasser og ansvar etter hvert. Hvis ikke det er tilfelle bør vi tenke nøye gjennom det vi holder på med en gang til – noe er antakelig helt galt.
6. Gjenta prosessen fra trinn 1 – nytt use case / scenario

Når vi har gått gjennom alle use casene og funnet fram til den kombinasjon av klasser og ansvarsforhold som vi trenger, kan vi dokumentere det hele i et interaksjonsdiagram.

02-21 Fowler kapittel 7

Pakker

Pakker brukes til å gruppere elementer som hører sammen – hovedsakelig klasser. Det beste / viktigste "hører sammen" begrepet er avhengigheter. Et pakkediagram inneholder:
- Pakker / samlinger av klasser
- Avhengigheter mellom pakkene

Avhengigheter har vi sett før. Vi har en avhengighet mellom to klasser dersom en endring i den ene klassen kan føre til endringer i den andre. Eksempler på slike avhengigheter er:
- Et grensesnitt blir endra
- Sende meldinger til en annen klasse
- Bruke en annen klasse som en del av sine data
- Bruke en annen klasse som parameter i en operasjon

Vis Fowler fig. 7-1 og kommenter. Avhengigheter er markert som før med stipla piler. Vi har avhengighet mellom to pakker hvis vi har avhengighet mellom en klasse i den ene pakka og en klasse i den andre. Legg merke til at pakkeavhengigheter ikke nødvendigvis er transitive.

Vi ser at hvis for eksempel pakke "Order" endres så behøver vi ***ikke nødvendig*** å endre pakka "Order Capture UI". Vi må imidlertid sjekke om det er nødvendig å endre "Order Capture Application". Denne pakka beskytter "Order Capture UI".

Det er lurt å ha så få avhengigheter som mulig mellom pakker. Nyttige råd fra Fowler:
- La alle klasser ha synlighet "privat". Da kan de bare sees av andre klasser i samme pakke.
- Lag egne klasser som er "public" for det som skal synes ut av pakka. Disse ekstra klassene kalles en "fasade".

Det å bruke pakker er ikke løsningen på avhengigheter i systemet. Det er imidlertid en hjelp i å se hvor avhengighetene er. Mange avhengigheter er dårlig nytt for en del sentrale egenskaper til et programvaresystem. Dette gjelder særlig:

- Vedlikeholdbarhet. Endringer blir verre og verre jo flere avhengigheter som finnes.
- Testbarhet. Mange avhengigheter gjør at det blir vanskeligere å finne ut hver feil kommer fra. I tillegg vil endringene bli vanskeligere pga lav vedlikeholdbarhet.

På den andre siden må det åpenbart finnes avhengigheter mellom pakkene, ellers er det ikke et system, bare en masse uavhengig pakker. Vi kan få et mål på graden av avhengigheter ved å se på to tall:
- Fan-in: antall andre klasser / pakker som kaller / bruker denne klassen / pakka
- Fan-out: antall klasser / pakker denne klassen / pakka kaller / bruker

Vi kan sette en grense på tallet Fan-in * Fan-out som det ikke er lov å overskride uten en nærmere begrunnelse. På den måten kan man ha en viss kontroll på strukturen. Tallet vil avhenge av hvor gode prosjektdeltakerne er – særlig hva slags kompleksitet de kan handtere. Det er viktig å også ta hensyn til kompleksiteten slik at det tallet vi må forholde oss til er (Fan-in * Fan-out)^2 * Kompleksitet. For kompleksitet kan vi bruke enkle ting som antall linjer eller mer kompliserte ting som McCabes kompleksitetstall. Vanligvis vil ”lenge” være godt nok. Hvis vi sier at vi maksimalt kan handtere fire kall inn og fire kall ut, når vi har 100 linjer kode finner vi ei øvre grense på 25 600. Viss vi velger å øke lengden til 500 linjer finner vi at Fan-in * Fan-out må være mindre enn sju noe som gjør at vi for eksempel må redusere Fan-out til to. Dette gir strengt tatt et noe større tall enn 25 600, nemlig 32 000. Slike tall må brukes med varsomhet og er bare ett av flere hensyn vi må ta når vi tar beslutninger.

Fowler, fig 7-2 viser en del nye muligheter. Først og fremst har vi samla ”Orders” og ”Customer” i ei ny pakke, kalt ”Domain”. I det generelle tilfellet kan vi beskrive ei pakke på tre måter - Sett navnet i ruta øverst til venstre og innholdet inne i hovedboksen. Dette kan være
- Ei liste av klasser, som for ”Common”
- Et nytt sett av pakker, som for ”Domain”
- Et sett av klasser.

Vi kan vise avhengigheter på flere detaljeringsnivåer. I Fowler fig 7-2, ser vi at ”Order Capture Application” bare er avhengig av pakka ”Domain”, mens vi er mer eksplisitte for pakka ”Mailing List Application” som bare er avhengig av ”Customer”.

Viss vi setter en avhengighet til ei pakke som har underpakker som for eksempel ”Domain” så sier vi egentlig at vi har avhengighet til en eller flere av underpakkene. Vi viser imidlertid ikke dette i dette diagrammet. Man må et nivå ned i detaljering for å se mer pressist hva som er avhengig av hva.

Vi har bruket stereotypen <<global>>, nederst i fig 7-2. Dette betyr at all pakker i systemet har en avhengighet til denne pakka. **Bruk dette med varsomhet**.

Vi kan bruke generalisering for pakker som vist nederst i fig 7-2. Vi har ei abstrakt pakke – leg merke til begrensningen {abstrakt} - ”Database Grensesnitt”. Denne viser at vi kan ha enten et Oracle eller et Sybase grensesnitt. Denne notasjonen impliserer en avhengighet fra subtypene til den abstrakte supertypen. Vi har altså nå ei abstrakt pakke med grensesnitt og andre pakker som implementerer hvert enkelt grensesnitt. Database grensesnittet er ansvarlig for å lese og skrive ”Domain” objekter til databasen.

Unngå løkker i avhengighetsgrafen. Ikke skriv om for enhver pris – det kan være gode grunner til å ha løkker. Det er imidlertid lurt å ha så få som mulig.

Samarbeid i pakker

På samme måte som pakker kan ha klasser kan de også ha samarbeid. Et samarbeid er en interaksjon mellom to eller flere klasser. Et slikt samarbeid ka vise en operasjon eller realisering av et use case. **Se Fowler fig. 7-3.** Dette viser et samarbeid mellom tre instanser av klassen ”Party”, kalt henholdsvis ”Selger” og ”Kjøper 1” / ”Kjøper 2”.

Vi kan ofte trenge et klassediagram som viser hvilke klasser som deltar i et samarbeid. **Se Fowler fig. 7-4**. Vi ser at salg involverer tre klasser, nemlig ”Party”, ”Tilbud” og ”Varemengde”.

Et samarbeid kan forekomme flere ganger i samme system, men med forskjellige klasser involvert. Dette kan vi dra nytte av ved å lage et parametrisert samarbeid. UML kaller dette for et mønster – ”pattern”. De som arbeider med mønster i andre miljøer er ikke uten videre enige i denne ordbruken. Rollebasert modellering ligger også i dette området. Trygve Reenskau er en av guruene på dette området med metoden OORAM. **Se Fowler fig. 7-5**.

Legg merke til at dette er et rollediagram, ***ikke*** et klassediagram. De rollene som er involvert står i firkanten øverst til høyre. Det tilsvarende klassediagrammet er vist i **Fowler fig 7-6**.

Noen gode råd

Det er lurt å innføre pakker hver gang klassediagrammet blir for stort. Fowlers definisjon av ”for stort” er noe som ikke går inn på et A4 ark.

Pakker er et bra nivå for testing. Det er mulig å teste en og en klasse, men erfaring viser at det er mer effektivt å teste ei pakke av gangen. Det kan værte praktisk å ha en testklasse pr. pakke.

02-21 Fowler kapittel 8

Vi har allerede sett på tilstandsdiagram i noen detalj. Dette er siste strøk og innføring av noen nye elementer:

- ”Guard”, som er en betingelse og aktivitet, som er en aktivitet oms blir utført når vi går over til tilstanden.
- Supertilstand, som er en måte å samle tilstander på.
- Generering av hendelser – ”event”.
- Parallelle tilstandsdiagrammer.

Guard og aktivitet

Fowler fig. 8-1 viser flere eksempler på denne konstriksjonen. En guard har følgende syntaks: ”<hendelse> [<betingelse>] / <aksjon>”. Vi kan utelate både hendelse og betingelsen og skriver da ”/ <aksjon>”. Alternativt kan vi utelate aksjonen og skriver da ”[<betingelse>]”.

Når vi går over til en ny tilstand kan vi utføre en ny aktivitet. Dette er betegnet med syntaksen ”do / <aktivitet>”. Viss man har en hendelse som gir en overgang og dette skal føre til en aksjon så kan man vise dette med notasjonen ”<hendelse> / <aksjon>”. Legg merke til at de tilstandene som ikke har noen aktivitet, bare har tilstandsnavnet inne i tilstandssymbolet.

I **Fowler fig. 8-2** har vi tatt med en ny tilstand – kansellert. Vi har ingen betingelse eller aksjon her, bare hendelsen ”kansellert”.

Supertilstander

Fowler fig 8-2 ser allerede noe uoversiktlig ut selv om det bare er fem tilstander der. Vi skal ikke legge til mange flere før det blir vanskelig å lese diagrammet. For å bevare lesbarheten slik at diagrammet virkelig kan fungere som et kommunikasjonsmiddel i systemutviklinga er det mulig å innføre supertilstander – nærmest et pakkebegrep for tilstander. Dette er vist i **Fowler fig. 8-3**. Supertilstander har navn som står i en egen boks. Det fins to typer overganger ut av en supertilstand:

- De som kommer fra en spesiell tilstand inne i supertilstanden. De blir tegnet på normal måte – fra den tilstanden de kommer fra. I Fowler fig 8-3 gjelder dette overgangen til ”Levert”.
- De som kommer fra alle tilstander inne i supertilstanden. De blir tegnet fra supertilstanden. I Fowler fig. 8-3 gjelder dette ”Kansellert”.

Genererte hendelser

Generell står hendelsene på den kanten som markerer overgangen til den nye tilstanden. En hendelse i et tilstandsdiagram – overgang til ny tilstand - kan bli generert på følgende måter:

- Etter en spesifisert ventetid. Dette blir markert med konstruksjonen ”**after** (<tid>)”. Dette kan være praktisk for eksempel for å forhindre at systemet henger ved at det venter på input. Vi kan bli enige om å vente i maksimalt 10 sekunder før vi går til sluttilstand. Det vil da være en kant *fra* den tilstand vi er i *til* sluttilstanden som er markert med ”after(10 sekunder)”.
- Når en betingelse blir oppfylt. Dette er for eksempel nyttig i kontrollsystemer og blir markert ved hjelp av konstruksjonen ”**when** (<betingelse>)”. Viss vi vil gå over i en ny tilstand når en robot har aktivert en gitt sensor kan vi for eksempel skrive ”when (signal fra sensor 4)”.

De genererte hendelsene kan også ha en betingelse og en aksjon knyttet til seg. Det komplette notasjonen er derfor ”<nøkkelord> (<parameter>) [betingelse>] / <aksjon>”.

Det finnes to spesielle hendelser – ”entry” og ”exit” som er knyttet til henholdsvis inngang til og utgang fra en tilstand. Viss vi har en transisjon som går tilbake til samme tilstand vil vi få eksekvert følgende aksjoner:
1. aksjoner knyttet til ”exit”
2. aksjoner knyttet til overgangen
3. aksjoner knyttet til ”entry”
4. aksjoner knyttet til tilstanden selv, for eksempel do/<aksjon>

Parallelle tilstandsdiagrammer

Tar utgangspunkt i **Fowler fig. 8-4**. Vi starter med å sjekke betaling i den første tilstanden. Dette er indikert med ”do / sjekk betaling”. Dette kan gi to resultater – OK eller ikke OK med sine respektive overganger.

I **Fowler fig. 8-5** har vi kombinert Fowler 8-1 som inneholder sjekk, pakk og lever med Fowler fig. 8-4 som viser betalingsautoriseringen. Når vi starter diagrammet vil vi begynne ***samtidig*** med å sjekke om vi har varen og om betalinga er OK. Som før kan betalingssjekken ende med at transaksjonen blir forkasta. Varedelen kan stadig når som helst bli kansellert. Den av de to tilstandsmaskinene som er ferdig først vil vente på den andre. Hvis begge når sin sluttilstand vil vi gå over i tilstanden ”levert”.

Noen gode råd

Tilstandsdiagrammer er nyttige for å beskrive oppførselen til et objekt over flere use case. Det er ikke så velegna til å beskrive oppførselen til flere samarbeidende objekter. Det er derfor lurt å kombiner tilstandsdiagrammer og for eksempel sekvensdiagrammer.

Ikke lag tilstandsdiagrammer for alle klasser – bare for de klassene der det gir ekstra innsikt eller informasjon. Vanligvis er det lurt å lage tilstandsdiagrammer for brukergrensesnitt og kontrollobjekter.

02-25 kapittel 10

Definisjon:
Programvarearkitektur er den strukturen som innbefatter programvarekomponenter, de egenskapene ved komponentene som er eksternt synlige og relasjonene mellom disse komponentene.

Arkitekturen i et programvaresystem har tre oppgaver:
- Den er et kommunikasjonsmiddel mellom alle interessenter. Ofte er den på diagramform og lett å lese og lett å forstå også for brukere.
- Den dokumenterer mange tidlige designbeslutninger. De tidlige designbeslutningene er viktige fordi de legger mange føringer på hva vi kan gjøre og ikke gjøre seinere i prosjektet. I tillegg vil det påvirke strukturen av prosjektet - for eksempel WBSen.
- Siden arkitekturen dokumenterer all viktig funksjonalitet i form av subsystemer eller funksjoner er den en viktig basis for å utvikle ei produktlinje – en serie av produkter, basert på den samme arkitekturen, men med variasjoner som avhenger av marked / brukergruppe eller applikasjonsområde.

Når vi skal velge arkitektur er det flere forhold som spiller inn. De viktigste er:
- Systemets funksjonelle krav og de datastrukturene systemet skal jobbe med. Dette er formalisert i metoden som heter JSP – ”Jacksons Structured Programming”.
- Utviklingsorganisasjonen vil kunne ha spesielle ønsker ut fra tidligere erfaring, hva som er vanlig i denne typen organisasjoner etc.
- Arkitektens tidligere erfaringer med hva som fungerer og hva som ikke fungerer. Det er ikke lett å gi opp noe som har fungert før.
- Det området systemet skal fungere i. For noen områder kan for eksempel offentlige bestemmelser gi føringen på hva som er lov og ikke lov. I tillegg vil den maskinvaren som skal brukes ha en del innflytelse, rett og slett fordi noen måter å gjøre ting på vil være enklere / mer effektive.

Det er lett å se at de forholdene som påvirker arkitekturen vil kunne forsterke hverandre. Viss vi har god erfaring med en spesielle måte å dele opp systemet på vil vi etter hvert kunne få avdelinger som er spesialister på hvert enkelt subsystem. Da har vi sementert denne arkitekturen og det blir vanskelig å endre den – det vil fort kunne kreve en omorganisering av bedriften.

Det finnes flere perspektiver på en programvarearkitektur. Noen av de vil vi kjenne igjen fra Fowlers bok.

- Konseptuelt eller logisk syn. Her ser vi bare på hovedkomponentene og deres interaksjoner.
- Implementasjonssynet. Her ser vi på moduler eller pakker og lag – for eksempel datalag, beregningslag og lag for brukerkommunikasjon.
- Prosessyn. Her ser vi på systemets dynamiske oppførsel – aktiviteter, prosesser, kommunikasjon mellom prosessene og allokering av prosessene. Dette synet er bare viktig viss vi har mye parallellprosessering.
- "Deployment" synet. Her ser vi på allokering av aktiviteter til fysiske noder, for eksempel i et nettverk. Dette er bare viktig viss vi har et distribuert system.

I tillegg vil det finnes andre, applikasjonsspesifikke syn som er viktige. For eksempel vil det være viktig å ha et sikkerhetsperspektiv viss vi lager et e-handel system.

Evaluering av en arkitektur

Det er mange faktorer som spiller inn. Noen viktige er hvor godt arkitekturen understøtter:

- Endringer i datarepresentasjonen
- Endringer i algoritmer
- Endringer i funksjonalitet
- Separat utvikling av hver enkelt modul.
- Forståelighet
- Ytelse
- Gjenbruk

Når man skal vurdere en arkitektur må man ta hensyn til alle disse faktorene. Alle vil imidlertid ikke være like viktige. Det er derfor nødvendig at man blir enige om hvor viktig hver enkelt faktor er før man begynner med vurderingene. Man kan for eksempel bruke en skala fra 1 til 10 for å gi både viktighet og subjektiv vurdering og summere produktene av de to tallene. Vi vil se på hvert av disse momentene under.

Endringer i data

Viktige momenter er datavolum og datarepresentasjon.

Det er ikke sikkert at det er de samme valga som er fornuftige ved små datamengder som ved store datamengder. Ved store datamengder vil det for eksempel være aktuelt å ha egne moduler som henter data, mellomlagrer dem og sjekker dem før man begynner å behandle dem. Det kan også være lurt å buffre data i dette tilfelle.

Datarepresentasjonene vil antakelig endre seg flere ganger i et produkts levetid. Det kan gjelde både format og innhold. Det kan komme til ekstra datafelt og det kan være endringer for eksempel fra tall til bokstaver slik det var for NTNUs karaktersystem.

Endringer i datamenge vil ofte forplante seg til endringer i algoritmer. Det er andre algoritmer som blir effektive for store datasett enn for små datasett. Et eksempel er matriseoperasjoner. Viss vi skal operere med spredte matriser – (store) matriser der bare noen få elementer er forskjellige fra null – er det ikke lurt å bruke de vanlige matrisealgoritmene for multiplikasjon, addisjon osv.

Endringer i algoritmer

Problemet med å endre algoritmer kommer vanligvis ikke fra algoritmen som sådan, men fra de krava algoritmene setter til datarepresentasjon eller den datamengden algoritmen er velegnet for.

I sanntidssystemer vil endringer i algoritmer kunne være vanskelige fordi det endrer tidsforbruk i en eller flere moduler. Viss man for eksempel har 10ms på å gjøre en oppgave og må bruke ei ny algoritme som bruker 12ms så har man i utgangspunktet et problem. Løsningen kan sette store krav til hvor lett det er å endre systemet.

Endringer i funksjonalitet

Det er rimelig sikkert at funksjonaliteten til systemet vil måtte endre mange ganger i løpet av levetiden til systemet. Bare systemer som er ubrukerlige vil være stabile over lang tid. Årsakene til endringene er at:

- Brukernes behov endrer seg etter som de får mer erfaring med systemet
- Brukernes krav endrer seg ettersom de ser flere og flere oppgaver som kan løses ved hjelp av datasystemer
- Omgivelsene endrer seg – bedriftens kunder, lover og regler, hva som er vanlig i bransjen. Alt dette fører til endringer i funksjonalitet.

Siden funksjonaliteten er en av de viktige parametrene når vi skal velge arkitektur vil mange funksjonsforandringer bli vanskelige. Dette gjelder særlig funksjoner som er spredt over mange moduler eller subsystemer. Det finnes to måter å resonere på:

1. Vi må forberede arkitekturen på så mange endringer som mulig slik at endringene blir lette å inkludere. Dette kan vi gjøre for eksempel ved å parametrisere funksjoner.
2. Vi vet ikke på forhånd hvor endringene kommer. Parametrisering av kode gjør koden mer kompleks og fører derfor til flere feil. Det er derfor ikke lurt å forberede systemet for endringer i funksjonalitet.

Dere må selv velge hvilken strategi dere vil bruke. Det ser imidlertid ut til at den mest brukte strategien for øyeblikket er strategi nummer 2.

Separat utvikling av moduler

Tid til market – TTM – er blitt en stadig viktigere parameter for programvarebedrifter. Innenfor utvikling av websystemer har kravene blitt ekstreme.

En måte å oppnå lav TTM på er å gjøre så mye som mulig av utvikling og testing i parallell. Dette krever at alle grensesnitt og fordeling av funksjonalitet - hvilken modul gjør hvilken oppgave - blir klarlagt på et tidlig tidspunkt. Arkitekturen kommer inn her ved at den bestemmer hvordan systemet blir delt opp og dermed bestemmer hvilke grensesnitt som må finnes og hva slags informasjon som skal utveksles.

Feil fordeling av funksjonalitet – hvilken modul skal gjøre hvilken jobb – vil kunne få katastrofale konsekvenser når kravene til parallell utvikling blir store.

Dersom mange moduler skal bruke de samme dataene må også datastrukturer defineres tidlig, noe som igjen vil kunne påvirke valg av algoritmene.

Forståelighet

En arkitektur må være lett å lese og lett å forstå. Dette er viktig for at alle involvert skal forstå konsekvensene av all beslutningene som taes i denne aktiviteten. Dette er spesielt viktig når vi senere skal gjøre forandringer. Det er også viktig når man skal definere grensesnitt – særlig viktig at man her har en felles forståelse av hvordan funksjonaliteten er fordelt over systemet – hvem har ansvar for hva.

Ytelse

Med ytelse kan vi mene tid – rask respons – eller stor lagringskapasitet.

Ytelse i betydningen responstid har i stor grad med flytting og tolking av data å gjøre. Alt som har med bergning å gjøre har utviklet seg mye raskere enn aksess til lagringsmedier som disk og lignende. Systemer med store krav til ytelse må ordne seg slik at data – når de først er lest – blir brukt i alle nødvendige beregninger. Det er uheldig å først laste dataene ned på disk og deretter behandle dem får så å lagre dem om igjen.

Ytelse i betydningen ”behandle store datamengder” setter krav til lagringsmedier og til algoritmer fordi vi likevel ønsker rask respons.

Gjenbruk

Gjenbruk kommer inn ved valg av arkitektur på to måter:

- Utvikling ***med*** gjenbruk
- Utvikling ***for*** gjenbruk.

Utvikling ***for*** gjenbruk:
Det viktigste vi må spørre oss selv om før vi lager noe for gjenbruk er: ”Hvor sannsynelig er det at vi kan gjenbruke denne modulen”. Siden det koster ekstra å lage noe gjenbrukbart er dette en investering og må behandles som det. Det betyr blant annet at vi må ha en idé om når vi får igjen de pengene vi har investert. Valg av moduler vil påvirke arkitekturen og visa versa gjennom oppdelingen i moduler.

Utvikling ***med*** gjenbruk:
For at gjenbruk skal fungere kan man ikke først lage arkitektur og overordna design og så begynne å lete etter passende komponenter. Den eneste måten å få dette til å fungere på er å ta utgangspunkt i kundens krav – hva skal systemet gjøre for meg – for så prøve å finne komponenter som gjøre hele eller deler av jobben. Når man har funnet et sett med komponenter man vil bruke har man lagt sterke føringer på hva slags arkitektur man kan ha.

Beskrivelse av arkitekturer

Boka foreslår å beskrive arkitekturer på et overordna plan ut fra følgende faktorer:

- Problem – hva slags problem vil vi løse med denne arkitekturen? En kort problembeskrivelse.
- Kontekst – i hva slags omgivelser skal det systemet som er lagt på basis av denne arkitekturen fungere i. Dette kan angå både maskinomgivelser – maskinvare, databasesystem og operativsystem.
- Løsning – hvordan ser løsninga ut på et overordna plan. Arkitekturen kan stort sett beskrives ut fra to elementer:
    - Komponenter – systemets byggesteiner.
    - Konnektorer – hvordan komponentene skal utveksle informasjon.

  Disse to valgene er vanligvis ikke uavhengige. Valg av komponenter vil legge begrensninger på valg av konnektorer og omvendt.
- Varianter av arkitekturen for å ta vare på spesialtilfeller.
- Eksempler – ett eller to for å gjøre ideene klarere for leseren.

Boka har flere eksempler som vi vil gå gjennom, men først må vi se på to kategoriseringer av henholdsvis komponenter og konnektorer.

Mulig kategorisering av komponenter:

- Beregningskomponenter. Vanligvis enkel input og output. Vanligvis er de hukommelsesløse – de oppbevarer ikke tilstander, men transformerer hver input til en output etter et sett av formelle regler. Eksempler er matematiske funksjoner.
- Hukommelseskomponenter. Lagring av persistente data som skal brukes av en eller flere komponenter. Eksempler er databaser, filsystemer og symboltabeller.
- Administrasjonskomponenter. Dette er komponenter som har et sett av tilstander med tilhørende operasjoner. Vanligvis vil de velge operasjoner ut fra tilstand – hva har hendt før – og fra input – hva hender nå – til å velge ut den riktige operasjonen. Alle systemer har minst en slik for å velge ut og utføre oppgaver.
- Kontrollkomponenter. Disse styrer sekvensen av hendelser i et system. Det er en glidende overgang mellom administrative og kontrollerende komponenter, men kontrollere styrer sekvenser, mens administrative komponenter stort sett velg ut funksjoner.

Mulig kategorisering av konnektorer:

- Prosedyrekall. Dette er en enkelt kontrolltråd mellom to komponenter. Kontrollen blir overført til en prosedyre som gjør en jobb og – før eller siden – gir kontrollen tilbake.
- Dataflyt. Hver enkelt komponent er styrt av datatilgang. Når en komponent ser at det er datadelen skal behandle blir den aktivert, gjør jobben sin og går over i ventemodus igjen. Jobben vil vanligvis medføre at data endres eller legges til i nye datastrømmer.
- Implisitt aktivering. En aktivitet blir startet når en definert hendelse inntreffer. Den som initierer hendelsen vet ikke hvem som kommer til å reagere og den som reagerer vet ikke hvem som har initiert hendelsen.
- Meldinger. Et element sender en melding til et annet element med beskjed om å handtere meldinga. Dette kna være synkront – sender er blokkert til den får svar – eller asynkront – sender fortsetter uten å vente på svar.

- Delte dataområder. Flere moduler / komponenter deler det samme dataområdet og utvekseler informasjon ved å lese / skrive data. Vi trenger en mekanisme for å hindre at to eller flere komponenter bruker de sammen dataene samtidig.
- Instansiering. En komponent – den som instansierer – skaffer plass for tilstanden til en annen komponent – den som blir instansiert.

Boka har flere eksempler på bruk av dette skjemaet – **vis fig 10.7 til fig 10.12.** Vi vil gå kort gjennom hver enkelt:

- Hovedprogram med et sett av subrutiner – fig .10.7. Passer best for utvikling i språk som Pascal, Modula II osv. Arkitekturne er et strengt eller svakt hierarki – forklar forskjellen. **Tegn figur**
- Abstrakte datatyper – fig. 10.8. Hver datatype er innkapslet og kommuniserer med omverdene gjennom et grensesnitt. Hver komponent er et objekt eller en tjener som tilbyr tjenester. Kontrollen er desentralisert – hver pakke kan aktivere andre pakker gjennom prosedyrekall eller meldinger.
- Implisitt aktivering – fig. 10.9. Systemet blir realisert gjennom et sett av prosesser. Prosessene er uanhengige og reagerer på eksterne hendelser. Det må finnes en felles hendelse handteringsmekanisme. **Tegn figur**. Legg merke til forskjellen fra "Abstrakte datatyper". Her er det ingen meldinger fra et objekt, men en felles handteringsmekanisme som sier fra når en spesiell prosess skal gjøre noe – systemet er hendelsesdrevet.
- "Pipes" og filter – fig 10.10. Systemet er realisert gjennom en serie av bergninger / transformasjoner. Dette er filter som er realisert som uavhengige prosesser. Filtrene kommuniserer ved hjelp av "pipes", der man kan sende inn og hente ut informasjon. En "pipe" er en FIFO kø. Avhengig av operativsystemet vi har må vi noen ganger selv passe på å ha eksplisitte sjekker for tomme og fulle "pipes".
- Felles datalager – fig. 10.11. Datalageret er hovedsaken her. Vi ønsker å vedlikeholde et datalager slik at det alltid er oppdatert – gir et riktig bilde av nå-situasjonen. Dette er viktig for lagersystemer i butikker, banksystemer og lignende. **Tegn figur**. Det vil vanligvis være mange systemer som opererer uavhengig mot ett datalager. Ofte vil det være transaksjonsorienterte systemer.
- Lagdeling – fig. 10-12. Vi har identifisert et sett av tjenester som an arrangeres i et hierarki. Hvert lag ***bruker*** tjenester på laget under og ***gir*** tjenester til laget over. Akkurat som for hierarkiske systemer finnes det en sterk – kan bare kalle på nivået direkte under – og en svak – kan kalle alle tjenester som ligger under – versjon. **Tegn figur**. En fire lags struktur blir ofte brukt:
    - Operativsystemlaget. Her finnes alle rutiner som gir informasjon om maskinens tjenester.
    - Utstyrslaget. Her finnes rutiner for I/O kontrollere osv.
    - Logisk ressursstyring. Her finnes abstraksjoner av maskinvareobjekter – for eksempel disker og skrivere – og programvareobjekter.
    - Tjenestelaget. Her finnes applikasjonsfunksjonaliteten.

  Mange systemer – særlig i en klient-tjener arkitektur – bruker en trelags modell. Vi har følgende lag:
    - Datalagring – ofte en eller annen form for database.
    - Datahandtering – algoritmer for bergning, transformasjoner og lignende.
    - Brukerkommunikasjon – alle typer brukergrensesnitt.

  Avhengig av hvordan man deler disse lagene på klient og tjener får vi systemer med tykke eller tynne klienter. **Tegn figur**.

En domenespesifikk arkitektur består av:

- Referansearkitektur - et spesielt valg av arkitekturstil, men uten det semantiske innholdet i komponentene – altså hva hver komponent gjør.
- Komponentbibliotek – gjenbrukbare biter av domenekunnskap. Disse er resultatet av en eller flere domeneanalyser.
- En konfigurasjonsmetode for å velge ut og konfigurere – tilpasse- komponenter til arkitekturen slik at det endelige systemet kan tilfredsstille kraven til applikasjonen.

Designmønster – pattern

Et designmønster er en gjentatt struktur av kommuniserende komponenter som løser et generelt designproblem i en spesifisert kontekst. Først og fremst må vi se på hva et designmønster er og hvorfor det er lurt å ha det. Et godt designmønster må:

- Angå et designproblem som vil forekomme flere ganger.

- Balansere et sett av motstridende eller konkurrerende krav. Dette er alltid et problem i design og et mønster må gjøre dette på en god måte for å være interessant.
- Dokumentere eksisterende, velprøvd designerfaring. Et mønster blir ***ikke oppfunnet***, det utvikler seg etter hvert, basert på praksis i utviklingsprosjekter. Slik sett er det dokumentasjon av "beste løsning".
- Berøre flere komponenter og hvordan de samarbeider.
- Gi en felles plattform – ideer, konsepter og begreper – å samarbeide ut fra. Slik sett er et mønster en del av et språk for å snakke om arkitektur og design.
- Være en måte å dokumentere på. Det må for eksempel være mulig å dokumentere arkitekturdesignet ved å referere til mønsternavn. Dette forutsetter igjen en generell og felles kunnskap om hva som ligger i begrepet.
- Gi et system eller en systemarkitektur som har definerte egenskaper. Dette gjelder primært funksjonalitet, men ikke-funksjonelle egneskaper vil også kunne være involvert. Dette gjelder særlig ting som fleksibilitet og endring av brukergrensesnitt.

Det er vanlig å beskrive et mønster med tre sett av informasjon:
- Kontekst – hva skal vi lage og for hvem?
- Problem – hva slags problemer løser dette mønsteret?
- Løsning – beskrivelse av hvordan mønsteret løser problemet

Andre bruker større, mer avanserte beskrivelser, for eksempel:
- Navn – hva heter mønsteret
- Synopsis – kort beskrivelse av mønsteret – en til tre setninger
- **Kontekst** – som før
- Historie ("Forces") – hva var det som ledet fram til å definere den løsningen som er beskrevet i dette mønsteret.
- **Løsning** – som før
- Konsekvenser – hvilke konsekvenser (positive og negative) får det viss man velger en løsning i henhold til dette mønsteret
- Implementering – ting man må passe på når man skal realisere dette mønsteret
- Java API bruk – viss det finnes eksempler på kode i Javas API kjerne så står de her
- Kode eksempel – inneholder et kodeeksempel for dette mønsteret
- Beslekta mønster – henvisninger til mønster som for eksempel takler beslekta / nærliggende problemer

Det mest kjente designmønsteret som er laget er MVC – "Modell View Controler". Dette er ikke fordi det er det "beste" mønsteret, men fordi det har gått igjen i mange lærebøker. Essensen i dette mønsteret er at det er en god idé å skille data ("model"), måten de presenteres på ("view") og den nødvendige logikken ("controller"). **Vis foiler med diagram, beskrivelsen og sekvensdiagram av MVC**.

Et annet mønster som er blitt mye bruket er "proxy" mønsteret. **Vis foil**. Hensikten her er å kunne bruke et objekt på en server som om det lå lokalt gjennom et stedfortredende objekt – en "proxy".

02-28 Kapittel 11

Hovedproblemene med å designe programvare er at:
- Det finnes ikke en endelig formulering av problemet. Det er vanskelig å skille design fra
    - Krav – som kommer før designprosessen
    - Implementering – som kommer etter.

  Derfor vil krav – design – implementering i de fleste tilfeller være en iterativ prosess. Dette har man tatt hensyn til i utviklingsmodeller som XP og inkrementell utvikling.
- Siden det ikke finnes en endelig problemdefinisjon er det heller ikke noen måte for å finne ut man er ferdig. Det finnes mange kvalitetskrav man kan sette til en design – for eksempel vedlikeholdbarhet, testbarhet, forståelighet – men det er ikke uten videre gitt at man kan tilfresstille alle.
- Designer er ikke riktige eller gale. De kan imidlertid være mer eller mindre velegna for det problemet vi holder på med eller mer eller mindre lette å implementere.
- Endringer i krav eller i implementasjon vil kunne ha store konsekvenser for design selv om endringen i utgangspunktet ser liten ut.

Vi kan formulere designproblemet på følgende måte: Hvordan kan vi dekomponere et system i deler slik at:
- Hver del har en lavere kompleksitet enn det systemet vi startet med

- Delene løser problemet når vi setter de sammen.
- Vi ikke flytter kompleksiteten ***fra*** systemet og delene ***til*** måten de er koblet sammen på.

Hva er en god design

En god design kan bedømmes ut fra flere kriterier. Vi vil se på følgende:

- Abstraksjon
- Modularitet
- Informasjonsskjuling
- Kompleksitet
- Systemstruktur

Abstraksjon

Å abstrahere betyr å konsentrer seg om de viktigste momentene og ignorere det som ikke er relevant på dette nivået. I systemdesign operere vi med to former for abstraksjon – dataabstraksjon og prosedyreabstraksjon. Vi ser kort på begge:

- Prosedyreabstraksjon – vi kan bryte problemet ned i prosedyretrinn som igjen kan brytes ned i nye trinn osv. Eventuelt kna vi bryte problemet ned i subproblemer som så igjen brytes ned på nytt til vi enten kommer til noe vi kjenner igjen eller til noe vi mener det er mulig å løse ved å skrive kode. Denne måten å jobbe på gir **en hierarkisk struktur** som vist tidligere.
- Dataabstraksjon – også kalt objektorientert design. Dette har vi sett på før. Istedet for å fokusere på aksjoner – prosedyrer – fokuserer vi på data og deres tilstander. Dette gir andre systemstrukturer med sine fordeler og ulemper.

Modularitet

Dette sier noe om sammenhengen mellom komponentene inne i en modul – kohesjon - og sammenhengen – koblingen – mellom modulene. Vi bruker to de to strukturkriteriene – kohesjon og kobling for å beskrive dette. Vanligvis vil man tilstrebe sterk kohesjon og svak kobling. Dette styrer mange designbeslutninger og også viktige beslutninger i en arkitektur.

- Kohesjon – limet som holder komponentene sammen i en modul sammen. Det finnes mange former for kohesjon. Vi kan liste de opp etter økende styrke – 1 lavest og 7 høyest - på følgende måte:
    1. Tilfeldig kohesjon – komponentene i modulene er gruppert på en eller annen tilfeldig måte.
    2. Logisk kohesjon – komponentene i modulene er gruppert sammen fordi de logisk hører sammen – f.eks. alle modulene som handterer data input.
    3. Temporær kohesjon – komponentene i moduler er gruppert sammen fordi de er aktive i same tidsområde – f.eks. alle modulene som deltar i initialisering.
    4. Prosedural kohesjon – komponentene i moduler som er gruppert sammen fordi de skal gjennomføres i en gitt rekkefølge for å realisere en bestemt funksjon – f.eks. lese data, oppdatere data og skrive data ned på en fil.
    5. Kommunikasjonskohesjon – komponentene i modulen opererer på samme data. Det er imidlertid ikke nødvendig at det forgår i en gitt rekkefølge.
    6. Sekvensiell kohesjon – output fra en komponentene i modulen er input til den neste osv.
    7. Funksjonell kohesjon – gruppering av alle komponentene som sammen realiserer en funksjon i en modul.
    8. Datakohesjon – komponentene som kobler sammen prosedyrer og de data de opererer på – abstrakte datatyper – er samlet i en modul.

    For å finne ut hva slags kohesjon man har kan man gjøre følgende enkle test. Skriv ned en setning som beskriver funksjonaliteten eller hensikten med strukturen. Ut fra dette kan man gjøre følgende observasjoner. Viss setningen inneholder:
    - Komma, ”og” eller mer enn ett verb, har det mest sannsynelig en sekvensiell eller kommunikasjonsmessig kohesjon.
    - Ord som ”før”, ”etter” osv, så er det mest sannsynelig sekvensiell eller temporal kohesjon.
    - Ord som ”initialiser” ol så er det mest sannsynelig en temporal kohesjon.

    Denne testen gjelder bare for prosedyrer / moduler.
- Kobling – graden av sammenheng mellom modulene. Akkurat som for kohesjon finns det mange former for kobling. Vi kan beskrive de fra sterkest – 1 – til svakest – 6 - på følgende måte:

1. Innholdskobling – en modul har direkte innvirkning på en annen modul. Dette kan f.eks. skje viss en modul direkte kan manipulere data i en annen modul som ved bruk av ”public” eller globale data.
2. Felles kobling – ”common” er brukt i FORTRAN for å definere et felles dataområde. I denne type kobling har flere moduler adgang til et felles dataområde der de kan manipulere et sett av data. **Tegn en figur** som viser forskjellen på kobling av type 1 og type 2.
3. Ekstern kobling – moduler kommuniserer gjennom et eksternt medium som f.eks. en fil. Dette har mye til felles med type 2 kobling(”common”).
4. Kontrollkobling – en modul styrer eksekveringen av en annen modul gjennom kontrollinformasjon. Dette gjøres ofte ved å sette kontrollparametere i den ene modulen og bruke de til å styre utføringa i en annen modul.
5. Stempelkobling – data er stemplet med informasjon om et felles, kjent format. Dette er f.eks. den koblingen vi har mellom moduler som utveksler pakka data – f.eks. i et datanett.
6. Datakobling – bare enkle data blir utvekslet mellom modulene, f.eks. i form av parametere.

Kobling er ikke kommutativ / gjensidig. A kan være datakoblet til B (f.eks. gjennom parameteroverføring av ett eller flere heltall) mens B kan overføre kontrollinformasjon til A og er dermed kontrollkoblet til A.

Husk - det er ønskelig å ha ***stor*** kohesjon inne i modulen og ***løs*** kobling mellom modulene. **Tegn aksekors** med navn på noen viktige koblinger og kohesjoner og vis hvor det er gunstig å være.

Sterk kohesjon mellom komponentene inne i en modul gjør at det er mulig å forstå komponentene – og dermed modulen – som et hele. Sterk kobling mellom moduler betyr at modulene at det er vanskelig å endre en av de uten å måtte endre mange andre. Dette kalles rippel eller bølgeeffekten. Svak kobling betyr at det er lettere å endre i en modul uten å måtte endre i mange andre. I tillegg blir det enklere å gjenbruke modulen.

Informasjonsskjuling

Informasjonsskjulig påvirker både kobling og kohesjon. Informasjonsskjuling fjerner mange av mulighetene for felleskobling, eksternkobling og kontrollkobling og oppmuntrer til datakobling. Dette skjer fordi den nødvendige informasjonen ikke er tilgjengelig – den er skjult.

På samme måte oppfordrer informasjonsskjuling til å la de komponentene som skal jobbe på samme data ligge i samme modul – datakohesjon eller abstrakte datatyper.

Kompleksitet

Boka påstår at kompleksitet er egenskaper som bare angår programvaren. Det jeg sitter inne med av erfaring tilsier imidlertid at den enkelt utviklers kunnskap og erfaring også må spille inn. Det som er vanskelig / komplekst for en nybegynner behøver ikke være vanskelig for en som har lang erfaring.

Det er rimelig å bruke flere former for kompleksitet:

- **Ekstern kompleksitet** – knyttet til kostnadene ved å løse problemet.
- **Intern kompleksitet**. Sett fra et designsynspunkt vil vi la alle faktorer som påvirker kostnadene – antall timeverk – som er nødvendige for å lage eller endre en programvarekomponent.
- **Tids- eller plasskompleksitet** – den tiden eller plassen komponenten bruker på eksekvere.

Det er vanlig å snakke om to deler av kompleksiteten – **intramodul** kompleksitet som beskriver egenskaper ved en enkelt modul og **intermodul** kompleksitet som beskriver forholdet mellom et sett av moduler i et system.

Det er grovt sett to sett med parametere som kan brukes til å estimere kompleksitet – henholdsvis størrelsesbaserte og strukturbaserte egenskaper.

- Størrelsesbasert metrikker er mål knyttet til modulens størrelse. Dette er greit definerbare mål. Eksempler på mål som har vært eller er brukt er:
    - Antall kodelinjer eller kodesetninger. Det siste har den fordelen at det ikke er avhengig av forskjeller i måten å skrive koden på.
    - Antall funksjonspunkter, objektpunkter (COCOMO II) eller use case punkter.
    - Antall genererte instruksjoner

- Strukturbaserte metrikker er mål knyttet til strukturen av modulen. Noen eksempler er
    - McCabes syklomatiske tall. Dette er det som er mest kjent og nest (mis)brukt.
    - Fan-in / Fan-out
    - Antall "knuter" i grafen.

Kompleksitetsmål er omstridte både i akademia og industrien. Det finnes ingen generell enighet om at ett mål er bedre enn et annet og deres verdi er tvilsom. Praktisk kna de brukes som **en av flere indikatorer** på at en komponent eller en modul begynner å bli komplisert.

Systemstruktur

Systemstrukturen består av relasjoner mellom moduler. Disse relasjonene kna ta mange former, f.eks.:

- Modul A inneholder modul B.
- Modul A kommer etter modul B – "etter" er her brukt i temporal betydning.
- Modul A leverer data til modul B.
- Modul A bruker modul B.

Generelt sett må den informasjonen en modul trenger om en annen modul i systemet holdes på et minimum. Det er derfor viktig å vite – for hver modul – hvilke andre moduler den bruker. Dette vises i en kallgraf. **Vis fig. 11.5 i boka**. Det er mulig å måle mange aspekter / egenskaper ved en kallgraf. Noen eksempler er:

- Størrelsen - antall noder, antall kanter eller summen av begge disse talla.
- Dybde – lengste sti fra rotnode til løvnode. Dette forutsetter at grafen ikke har løkker.
- Bredde – det maksimale antall noder på et gitt nivå.

Det blir vanligvis antatt at jo nærmere en graf er et tre, desto bedre er det. Viss grafen ikke er et tre finnes det minst en kant som kan fjernes uten av grafen slutter å være sammenhengende. Vi kan forstsette å fjerne kanter helt til vi får en graf. Resultatet når vi ikke kan fjerne noen kan mer uten av grafen slutter å være samehengende kalles en grafen vi da har for en spenntre. Antall kanter vi har måttet fjerne er et mål for hvor langt den opprinnelige grafen var fra å være et tre.

Et vanlig mål for dette avviket er følgende:
En komplett graf med n noder har n (n –1) / 2 kanter. Et tre med n noder har (n – 1) kanter. Viss vi har en graf G med n noder og e kanter, vil vi definere urenhetsmålet på følgende måte: $m(G) = 2 (e – n + 1) / (n – 1) (n – 2)$. **For utledning, se vedlagt lapp**. Legg merke til at det ikke alltid er et overordna mål å få en graf som er så nær et tre som mulig. Som fører dette bare en av mange input i problemet med å velge en god designstruktur.

Et mye brukt kompleksitetsmål for en modul M er Henry og Kafuras mål: $(\text{Fan-in}(M) * \text{Fan-out}(M))^2$ som et mål på kompleksitet. Vi har sett på dette tidligere.

Designmetoder

Det finnesen stor mengde designmetoder. Vi skal bare se på tre av de i noen detalj. Dette er:

- Funksjonsdekomposisjon
- Dataflytbasert design
- Design basert på datastruktur – JSP

Funksjonsdekomposisjon

I denne metoden deler vi systemet opp i funksjoner som skal oppfylle systemets krav. Hver funksjon kan deles opp i nye funksjoner osv, helt ned til kode. Dette ligger nær til det vi kaller trinnvis forfining eller detaljering, men foregår på et høyere nivå. Det er vanlig å skille mellom top-down og bottom-up design.

- Top-down - Vi starter på toppen av systemet og bryter det ned i stadig mindre funksjonsbiter. For at vi kan gjennomføre dette helt rigid må vi kjenne all funksjonaliteten på forhånd. Det hadde naturligvis vært kjekt, men inntreffer dessverre sjelden.
- Bottom-up - Her begynner vi med de funskjonsbitene vi kjenner. forstår eller har tilgjengelig. Deretter bygger vi på toppen av disse for tilslutt å realisere all den funksjonaliteten som skal være i systemet.

Ingen av disse strategiene kna vanligvis brukes på en rigid måte. Det er flere grunner til dette, bl.a.:

- Kundene vet ofte ikke hel hva de vil ha og selv om de visste det er de ikke alltid i stand til å forklare det på en slik måte at det kan tjene som grunnlag for en design
- I tillegg til kundekravene er det mange andre krav et programvaresystem må møte. Dessverre blir disse bare klare etter hvert som vi jobber med problemene.
- De fleste prosjekter vil gjennomgå en eller flere endringer i løpet av sitt liv. Disse endringene er uforutsigbare.
- Folk gjør feil
- I design bruker folk den kunnskapen de har, erfaringer fra det de har gjort før og lignende.
- Mange prosjekter bygger på det som er gjort før – vi lager ofte ikke system fra grunnen av, men bygger på / utvider systemer som eksistere fra før.

Systemdesign vil måtte foregå etter ”yo-yo” prinsippet. Det finnes noen få regler – de som kommer her er hentet fra David Parnas:

- Begynn med å identifisere subsystemer. Start med det minst mulige subsystem og utvid etter hvert, inkrementelt
- Bruk prinsippet om informasjonsskjuling – objektbasert utvikling.
- Legg på utvidelser trinn for trinn. Bygg systemet lagvis slik at funksjonalitet på ett lag understøtter det som leggs over.
- Bruk ”uses” / ”includes” relasjonen fra UML. Dette gitr et sett av avhengigheter. Prøv å bygge et hierarki ut fra dette.

Som alltid - dette er ikke en sekvensiell kokeoppskrift. I stedet må vi se på det som et sett av trinn som vi kan gå gjennom en eller flere ganger for å lage en systemstruktur. Ofte vil systemene få en blanding av lagstruktur og hierarki som den som er vist i **fig. 11.6**.

Parnas ide om det minimale subsett som så senere kan utvides har fått stor innflytelse på arbeidet med produktfamilier.

Dataflytbasert design

På toppnivå er dataflytbasert design det samme som funksjonell dekomposisjon, bare at fokus er på data i stedet for funksjoner. Vi fokuserer på dataflyten. Komponentene er i utgangspunktet svarte bokser som transformerer data. Selve designprosessen består av to trinn:

- Logisk design ut fra dataflyt. Dette kalles strukturert analyse – SA. Det å klarlegge dataflyten er på mange måter en kravanalyse. Siden vi etter hvert legger på mer detaljer i dataflytdiagrammet foretar vi en implisitt top-down funksjonell dekomponering av systemet. Hovedresultatet av denne aktiviteten er et sett av dataflyt diagrammer. Diagrammene har fire typer av komponenter:
    - Eksterne elementer / entiteter. Dette er kilder eller sluk for en transaksjon. De blir markert med firkanter.
    - Prosesser som omformer data. Disse blir tegnet som sirkler i diagrammet.
    - Dataflyt mellom to elementer – eksterne enheter, prosesser eller datalager. Disse blir tegnet som linjer med piler som viser retningen.
    - Datalager som ligger mellom to prosesser. Dette blir markert med to parallelle linjer med navnet på datalageret mellom linjene.

  På øverste nivå har vi et kontekstdiagram**. Vis fig 11.7 og kommenter**. Deretter kan vi bryte systemet videre ned. Starten på dette er **vist i fig 11.8**. Videre nedbryting stopper når prosessene blir så enkle at vi ser hvordan vi kan realisere de. Erfaring spiller en stor rolle her. En slik prosess kalles en minispek. **Se fig. 11.9**. Resultatet av denne fasen er et sett av dataflytdiagrammer og minispeks.
- Den logiske designen blir omformet til en programstruktur i ett eller flere strukturdiagrammer. Dette kalles strukturert design – SD. Siden forrige fase har generert et sett av minispeker som er prosesser, så vil vi nå kunne beskrive hvordan vi kan gjøre

om en eller flere prosesser til kode. Metoden hadde opprinnelig få retningslinjer for dette, men noen har kommet til etter hvert – basert på erfaring. Se også **fig 11.12**.

Metoden gir en arkitektur som klassifiseres som produsent – konsumer modellen. Prosesser produserer data som neste prosess konsumerer. Slike arkitekturer kan blir ganske kompliserte. Får å kunne rydde opp prøver man å redusere kobling og øke kohesjon mellom grupper av prosesser.

Design basert på datastrukturer – JSP

Hovedmomentet i JSP / JSD er at en god design gjenspeiler datastrukturen. Siden datastrukturen er mer robust enn de funksjonelle kravene får man en struktur som er både logisk og robust. JSP bygger på tre basisstrukturer – sekvens, iterasjon og valg – seleksjon. **Se fig. 11.33. Kommenter** hvordan de tre datastrukturene kan se ut og hvordan koden passer til å handtere dette. JSP som metode består av følgende sekvens av aktiviteter:

1. Bruk strukturdiagrammer til å modellere input og output.
2. Sett sammen strukturdiagrammene for input og output for å få en helhetlig struktur. Denne strukturen vil automatisk bli hierarkisk.
3. Løs opp i det Jackson kaller strukturkollisjoner. Dette er tilfeller der vi ønsker å koble sammen to ulike syn i trinn 2 – altså input og output strukturene.
4. Optimaliser koden.

I de to metodene vi har sett på tidligere går vi fra problemstruktur til funksjonsstruktur og deretter til en programstruktur. I JSP går vi fra problemstruktur til datastruktur og så til en programstruktur. JSP har hatt fokus på den siste overgangen – datastruktur til programstruktur. For å få en ryddig overgang fra problemstruktur til datastruktur innførte Jackson JSD – Jackson Strukturert Design. Denne prosessen består av tre trinn:

1. Modellering. Dette er en modell av den delen av verden vi er interessert i - det samme som en domenemodell. **Se fig. 11.21**. Legg merke til symbolene for iterasjon ”*” og seleksjon ”o”. Disse diagrammene er tilstandsmaskiner og kalles prosess-struktur diagrammer. Modellen tar utgangspunkt i elementer i den delen av verden vi er interessert i. Hvert element i diagrammet er en aksjon – f.eks. ”Lån” eller ”Lever tilbake”. I tillegg vil hvert av elementene ha attributter. F.eks. vil ”Lån” ha attributter som beskriver hvilken bok han låner.
2. Nettverk. Her setter vi systemet sammen til et nett av kommuniserende, parallelle prosesser. Nettverket er et system spesifikasjonsdiagram. JSD har to metoder for kommunikasjon mellom prosesser:
   - En enhet kan se på tilstanden til en annen enhet.
   - En enhet kan sende en asynkron melding til en annen enhet via en datastrøm.

   Disse to kommunikasjonsmetoden har ulike notasjoner. Se på tilstand – en ruter med ”SV” inne i og datastrøm - en sirkel med ”SD” inne i.
3. implementasjon. Her gjør vi om nettverket fra trinn 2 til en sekvensiell design.

Valg av designmetode

Ingen designsmetode gir oss en garanti for et fantastisk resultat. Det som teller er kombinasjonen av metode og våre erfaringer. En god metode skal hjelpe oss med å gjøre det beste ut av de erfaringene og den informasjonen vi har. Det at en metode er beskrevet på en algoritmeliknende måte gjør at den er lett å ta i bruk – viss den passer for det vi skal bruke den til. Metoder som er mer deskriptive vil være vanskeligere å ta i bruk, men vil være lettere å tilpasse til det problemet vi har tenkt å løse.

Noen erfaringer:

- JSP / JSD egner seg best der datadefinisjonene er gitt og statiske.
- Dataflyt teknikker vil gi oss ekstra hjelp viss dataene er dynamiske. Den passer best der vi skal erstatte et manuelt system med et datasystem.

Fig 11.24 viser en enkel måte å klassifisere designmetoder på. Den har to dimensjoner:

1. Problemorientert vs. produktsorientert. Problemorienterte metoder fokuserer på det problemet som skal løses og er menneskeorientert. Man kan beskrive, kommunisere og dokumentere designbeslutninger. Disse metodene vil ofte ha aspekter av kravanalyse. Produktorienterte metoder fokusere på en korrekt transformering av krav til implementasjon.
2. Konseptuel vs. formell. Konseptuelle metoder er deskriptive og tar utgangspunkt i den verden vi skal modellere og realisere. Formelle metoder er preskriptive – de viser en beskrivelse av det systemet som skal

realiseres. Viss vi har en formell beskrivelse kan vi også bevise designet – men bare at det er et design av det systemet vi har beskrevet, ikke at det tilfredsstiller kundens krav.

De fire kvadrantene har følgende tolking eller hovedfokus:
1. Forstå problemet – få fram en løsning på en form som gjør at løsningen kan kommuniseres, forståes og diskuteres av andre.
2. Transformer til implementasjon – fra beskrivelse av applikasjonsdomenet til kjørbar kode.
3. Representer egenskaper – metoder for å diskutere et problem og mulige løsninger.
4. Lage implementasjonsenheter – er beregnet for å lage f.eks. kodemoduler.

Det ovenstående angår det problemet som skal løses – hva vi ønsker å oppnå. Det er mange andre faktorer som også påvirker designbeslutningene:
- Domenekunnskap – viss vi har mye domenekunnskap vil en top-down teknikk basert på datastrukturene være effektiv. Viss vi er i en eksperimentell fase – prototyping – vil det likevel lønne seg å bruke en bottom-up metode.
- Designernes erfaringer – hva har de brukt før. De fleste er mest effektive når de kan bruke en metode de har mye erfaring med fra før.
- Tilgjengelige verktøy – en metode med verktøystøtte vil være å foretrekke viss alt annet er likt.
- Total utviklingsfilosofi – hvordan lager vi system: fra de første brukerintervjuene til drift og vedlikehold. En designmetode som passer inn i denne totaliteten vil være å foretrekke.

Notasjoner

Det er viktig å dokumentere designet man til slutt velger. Det er imidlertid også viktig å huske op at koden vil bli endra – noe som vil kunne føre til at deler av designet også blir endra. Dersom vi ikke passer på å oppdatere designet vil ”kart og terreng” snart være forskjellige. Kostnaden med å oppdatere design etter hver kodeendring er en vesentlig kostnad i vedlikehold av programvare. To ting kan gjøre dette enklere: automatisks oppdatering av design når koden oppdateres og automatisk kodegenerering fra design - vedlikehold av design, ikke av kode.

En design vil vanligvis være kompleks og en enkel diagramteknikk er ofte ikke nok. Dessuten er diagramteknikkene ikke rigide – vanligvis kan vi tegne det vi vil, uten noen form for formell sjekking. I tillegg trenger vi en god del beskrivende tekst. Denne kan være kort, lesbar og flertydig eller lang, vanskelig å lese, men rimelig presis.

Dokumentasjon av design

En design skal oppfylle flere roller i et prosjekt. De viktigste er:
- Prosjektledelse – trenger info for å planlegge og kontrollere prosjektet.
- Konfigurasjonsansvarlig – finne ut hvilke komponenter som skal inngå hvor i systemet.
- Designer – funksjonaliteten til hver komponent og hva slags grensesnitt hver enkelt komponent har.
- Utviklerne – hva skal hver komponent gjøre (algoritmene) og hvordan skal den utveksle info med andre komponenter (grensesnitt).
- De som utfører enhetstester – hvordan skal komponenten oppføre seg for hver enkelt inputklasse.
- De som gjør integrasjonstest – hvordan skal komponentene i systemet jobbe sammen for å realisere kravene til subsystemer og system.
- De som gjør vedlikehold – hvilke brukerkrav og hvilken funksjonalitet er realisert i hvilke komponenter.

I IEEE standard 1016 er det spesifisert ti attributter som hver designenhet – komponent – trenger. Disse er som følger:
- Identifikasjon – et navn slik at vi kan henvise til den.
- Type – f.eks. om det er et subsystem, en prosedyre eller en fil.
- Hensikt – hvorfor trenger vi denne enheten, f.eks. gjennom å referere til kravspesifikasjonen.
- Funksjon – hva gjør komponenten rent konkret, i form av oppgaver, tjenester. Igjen er det viktig å referer tilbake til kravspesifikasjonen.
- Deler – hva er denne enheten satt sammen av.
- Avhengigheter – dette er både funksjonelle avhengigheter – hva må gjøres før / etter denne enheten – og definisjonsmessige avhengigheter – hva deler denne enheten med andre av data, grensesnittdefinisjoner osv.
- Grensesnitt – hva er grensesnittet til denne enheten. Vi må også si hvordan den kommuniserer – prosedyrekall, meldinger, software bus etc.
- Ressurser - hva trenger vi av maskinvare, drivere, biblioteksprosedyrer etc.

- Prosessering – hva slags algoritme bruker vi for å få gjort jobben. Det er viktig å skille mellom dette og ”funksjon”.
- Data – hva trenger denne enheten av data for å gjøre jobben sin med den valgte algoritmen. Her må vi også ha med dataformat og interne data.

For å gjøre det lettere tilgjengelig deler IEEE 1016 denne informasjonen opp i tre grupper – dekomposisjon, avhengigheter, grensesnitt og detaljert beskrivelse. Se beskrivelsen i **tabell 11.4. Kommenter.**

03- 04 Kapittel 14

Det er vanlig å operere med fire typer vedlikehold:
- Korrektivt vedlikehold – feilretting
- Adaptivt vedlikehold – tilpasninger til endringer i maskinvare, operativsystem, andre systemer dette systemet må kommunisere med, bedriftens interne organisasjon og bedriftens omverden.
- Perfektivt vedlikehold – forbedring av koden med hensyn til brukergrensesnitt og tidsforbruk
- Preventivt vedlikehold – opprydding for å forbedre vedlikeholdbarheten.

Se fig 14.1

Vedlikeholdbarhetene til systemet / koden er vesentlig for alle systemer som skal leve mer enn ett år. Det er mange måter å forbedre den på, både når det gjelder kode, struktur og brukeregenskaper. Dette er noen momenter:
- God - i betydningen enkel - kode, god dokumentasjon, kodestandard
- En endringsvennlig arkitektur.
- Oppmerksomhet mot kundens behov i alle utviklingsaktiviteter.
- Hold volumet av nyutviklet kode nede – mindre kode å endre / vedlikeholde.

Manny Lehman har formulert et sett av lover for programvare. De to som interesserer oss her er som følger:

Kontinuerlig endring – et hvert system vil gjennomgå hyppige endringer. Dette vil pågå helt til det er så vanskelig å endre systemet at endringene stopper av seg selv. Deretter vil systemet leve en periode uten å bli endret, for så å bli erstattet av et nytt system.

Økende kompleksitet – etter hvert som et system blir endret blir det mer og mer komplekst. Dette fører til at hver ny endring blir mer kostbar og før eller siden vil det ikke bli mulig å fortsette.

Disse to lovene sier begge to at det er en grense for hvor lenge et system kan leve før det ikke lenger kan tilpasse seg en verden i stadig endring. Det er imidlertid flere årsaker til at man nøler med å erstatte dette systemet med et nytt:

- For administrative systemer er det mye forretningslogikk i de eksisterende systemene som ikke lenger eksisterer i eksplisitt form andre steder.
- Det er vanskelig å skaffe programmerere med tilstrekkelig applikasjonskunnskap
- Nyutvikling er kostbart
- Nye systemer vil inneholde flere feil enn det gamle systemet. I tillegg bytter man feil man kjenner – og dermed kan ta hensyn til – mot nye feil som man ikke kjenner og dermed ikke kan passe seg for.
- Det kan være vanskelig å lage et nytt system med samme funksjonalitet fordi vi ikke kjenner den totale funksjonaliteten. Ofte er den heller ikke dokumentert.

Noe kan gjøres med enkle midler:
Vi kan øke kodens lesbarhet ved å strukturere den bedre – ”prettyprinters”. Vi kan få oversikt over strukturen ved hjelp av verktøy som er spesielt beregnet for dette formålet.

I tillegg til alt det andre er vedlikehold ofte blitt sett på som en lavstatus jobb, sammen med testing. Dette gjør at utviklere skyr vedlikehold som pesten og mange vil slutte hvis de får for mange vedlikeholdsoppgaver. De fleste føler det som en karrieremessig blindvei.

To løsninger har vært brukt med noe hell:

- Høyere lønn for vedlikeholdsavdelingen – f.eks. 25% i snitt. Dette vil man imidlertid neppe få gjennomslag for i Norge. Andre løsninger som ligner på dette er å la vedlikeholdsfolk få adgang til ledelsens lunsjrom – noe som har fungert godt i noen engelske bedrifter
- Vedlikehold som en plikt – alle i utvikling må være i vedlikeholdsavdelingen to måneder i året. Har i det minste den effekten at folk blir mer oppsatt på å lage vedlikeholdbar kode siden de vet at de snart må endre i den selv.

Reverse engineering

Reverse engineering ble på et tidspunkt sett på som medisinen som skulle redde oss ut av problemene med vedlikehold av store, gamle systemer. Det har dessverre ikke blitt den suksessen som man trodde. Det er flere grunner til det, men den viktigste er at det er for vanskelig å finne igjen det som egentlig var hensikten med koden. Uten den innsikten kan vi bare gjøre kosmetiske forandringer. Disse hjelper, men de løser ikke problemet. Det vi kan gjøre er å:

- Bytte ut dårlige navn som X1 og AB med mer meningsfylte navn som angår den oppgaven denne variable virkelig har.
- Gi koden en layout som gjør den lettere å lese. Dette er en av de tingene vi kan gjøre automatisk.
- Sørge for at de som endrer noe i koden samtidig rydder opp i det området de er inne i og dokumenterer ordentlig, bla. ved å sette inn kommentarer. Dette er effektivt, men berører bare små deler av koden av gangen. Generelt sett er endringer farlige fordi de kan innføre nye feil.

Det finnes noen typer av verktøy som vil være til noe nytte i dette arbeidet:

- Pretty-printers, som vi har snakka om et par ganger før.
- Verktøy som hjelper oss å få oversikt over strukturen til programmet. Dette kan være:
    - Enkle verktøy som tegner kallgrafene eller lager kryssreferanselister
    - Avanserte, hypertekstbaserte verktøy som hjelper oss å vandre rundt i koden på en effektiv måte.
    - Verktøy som beregner en del metrikker som Fan-in / Fan-out eller McCabes syklomatiske tall. Dette kan hjelpe til med å finne potensielle problemområder i systemet
- Verktøy for å måle testdekningsgrad – f.eks. kodedekningsgrad eller stidekningsgrad. Selv om disse verktøyene ikke direkte hjelper oss med å strukturere koden, hjelper de oss ved å vise hva salgs konsekvenser endringer har og de viser også hvordan dele ar systemet henger sammen.
- Dynamiske debuggere, gjerne med et grafisk grensesnitt.

Ledelsens rolle

Boka innfører en typologi for programvareorganisasjoner. Dette er bare relevant for store bedrifter. For de fleste norske bedrifter er ikke dett så viktig. Vi vil derfor bare berøre det ganske kort.

- W bedrifter – delt opp etter type arbeidsoppgaver, f.eks. en analysedel og en programmeringsdel. Vi får flinke spesialister på hver enkelt oppgave, men det er vanskelig å koordinere. ”Over the wall” engineerng er et hyppig problem.

- A bedrifter – delt opp etter applikasjonsområde. Igjen får vi dyktige spesialister, men må betale for det med koordineringskostnader når flere applikasjonsdomener må samarbeide.
- L bedrifter – delt opp etter livssyklus for systemet. Eksempler er analyse, utvikling, vedlikehold og drift. Problemene er de samme som før.

Uansett modell er den viktigste årsaken til problemene dårlig kommunikasjon – særlig mellom utvikling og vedlikehold. Det å skille utvikling og vedlikehold har både fordeler og ulemper.

Fordeler:
- Greie ansvarsforhold. Vi får god oversikt over hvor mye vi bruker til utvikling og hvor mye vi bruker til vedlikehold. Dette hjelper til med å forstå kostnadsbildet bedriften har og hva høy vedlikeholdbarhet er verdt.
- Et alvorlig problem i all prosjektplanlegging er at folk blir tatt ut av prosjektet for å gjøre endringer eller rettinger i andre systemer. Ved å ha to avdelinger med hvert sitt ansvarsområde blir dette problemet kraftig redusert.
- Vedlikeholdsavdelingen blir sterkt motivert til å ha gode tester før et produkt sendes til en kunde. De feilene de finner her kan sendes tilbake til utviklingsavdelingen som stadig ”eier” produkter. Hvis det er mye feil i det som blir levert til kunden blir det vedlikeholdsavdelingen som får alt ekstraarbeidet.
- Ved å ha vedlikeholdsspesialiteter vil det vanligvis bli et bedre og mer effektivt vedlikehold.

Ulemper:
- Siden vedlikehold tradisjonelt er en lavstatus jobb kan man risikere å få bare folk uten ambisjoner eller med dårlig kunnskap og erfaring i denne avdelingen. Dette gjør alle vedlikeholdsproblemer gradvis verre over tid.
- Når ansvaret for systemet flyttes fra en avdeling til en annen vil vi miste mye erfaring. Selve systemet og dets dokumentasjon er enkel å overføre, men vesentlig info som ***hvorfor*** vi valgte en spesiell løsning forsvinner i prosessen.
- Formell overføring av ansvar fra en avdeling til en annen koster tid og penger. Bedrifter som har dette skillet har opplevd lange tider med forhandlinger fordi utvikling og vedlikehold ikke kan bli enige om status for produktet, hva slags info som skal overføres, osv.
- Kunder som kjenner utviklingsbedriften har en tendens til å hoppe over vedlikeholdsavdelingen – ”de vet jo aldri noe likevel” – og går rett på utviklerne for å få fikset feil. Dette gir uklare ansvarsforhold og problemer med fakturering – hvem skal betal hvem for hva. **Eksemplet fra NOVIT**.

Vedlikehold som en tjeneste

Vedlikehold er en tjeneste og mange kunder ser på leverandørens service - kvalitet og innstilling - som viktige faktorer når man skal velge leverandør. For noen kunder blir det bedømt som mer positivt å ha problemer og få god behandling enn ikke å ha problemer i det hele tatt. Når vi selger så selger vi altså både et produkt og en tjeneste. Hvor mye som er en tjeneste og hvor mye som er et produkt varierer. **Fig 14.10** viser hvordan dette blandingsforholdet varierer for programvare.

Et viktig hjelpemiddel i analyse av tjenester er gapanalyse – gapet mellom det vi yter og det kunden forventer. **Fig. 14.11** identifiserer fem viktige, mulige gap som vi skal kommentere.
- Gap 1 – gapet mellom hva leverandøren synes er rimelig – bra nok – og det kunden forventer. Det er viktig for leverandørene å vite hva kundene forventer. For å forstå kundene bedre trenger vi å gjøre markedsanalyser.
- Gap 2 – gapet mellom hva bedriften ønsker å vise fram / yte og interne standarder i bedriften – hva de ansatte synes er viktig og riktig.
- Gap 3 – gapet mellom hva vi ønsker å gjøre og hva den enkelte ansatte gjør. Gapet kan komme av faktorer som stor arbeidsbelastning, lite kunnskap eller generell mangel på interesse. Som en så vakkert sa det ”Jeg er utvikler. Jeg utvikler programmer. Hva kunden synes om det er totalt irrelevant”.
- Gap 4 – gapet mellom det vi lover i kontrakter, brosjyrer og i kommunikasjon med kundene og det vi egentlig leverer. Erfaring viser at det er lettere å love enn å holde – her som overalt ellers. Dette blir særlig tydelig når bedriftens økonomi begynner å bli trang.
- Gap 5 – dette er den samla effekten av de fire andre gapene.

Det er viktig å finne ut hvor fornøyde kundene er med vedlikeholdstjenestene vi gir. Dersom de ikke er så fornøyde som vi ønsker må vi gjøre noe med det. Gapmodellen er en måte å finne ut hva vi skal gjøre og hvor vi skal sette inn innsatsen.

Kontroll med vedlikehold

Vedlikehold består av minst tre aktiviteter som må kontrolleres nøye. Her er de listet sammen med noen minimumstiltak for å klare å holde hodet over vannet.

- Identifiser og klassifiser endringsønsker. Tre ting er viktig. Alle endringsønsker må få en:
    - Id slik at de kan refereres til.
    - Analyse som konkluderer med at endringen **aksepteres** – vi gjør noe med dette, **forkastes** – dette vil vi ikke gjøre noe med, eller at vi **trenger mer info**. Det siste er aktuelt hvis vi ikke forstår hensikten (eller bare ønsker å utsette hele greia).
    - Prioritet ut fra en første analyse av viktigheten.
- Analyse av endringen. Hensikten er å se på konsekvensene av å utfører endringen. Dette gjelder også konsekvenser for resten av systemet. Aksepten fra forrige fase er bare midlertidig. Det er viktig å se på mulige løsninger, hva de vil koste og hvor lang tid det tar å implementere den.
- Implementering. Det som er viktig her er å passe på at:
    - Endringen får den effekten som var planlagt.
    - Ikke får andre effekter – for eksempel at det som fungerte før ikke fungerer lenger.

For å få til dette er det viktig å ha et godt testbibliotek og god sporbarhet mellom tester, kode og krav. Det er nå vi får glede av den innsatsen vi la ned tidligere i prosjektet med å få til god sporbarhet.

Dette opplegget er det samme som man ser i iterativ utvidelse. Det er rimelig siden skille mellom vedlikehold og utvidelse / videreutvikling er mildt sagt utydelig. Mange bedrifter – både leverandører og kunder – ser ingen hensikt i å skille mellom de to aktivitetene uansett. Noen bruker spørsmålet om nytte for å avgjøre problemet. Nøkkelspørsmålet er: ”Vil jeg stadig ha nytte av systemet hvis denne endringen ikke gjøres?”

- ”Ja” – dette er videreutvikling. Jeg legger på ny funksjonalitet for å få mer nytte ut av systemet
- ”Nei” – dette er vedlikehold. Endringen er en reparasjon og blir den ikke gjort kan jeg ikke bruke systemet mer.

Dette er ikke en endelig beslutningsmekanisme. Det finnes antakelig mange unntak og / eller tvilstilfeller. I tillegg til dette finnes det en glidende overgang fra videreutvikling til gjenbruk. Det er ikke uten videre lett å avgjøre når man går over fra å videreutvikle et system til å gjenbruke deler av det for å lage et nytt system. Det er derfor naturlig å se på vedlikehold – videreutvikling – gjenbruk, ikke som tre kategorier, men som delvis overlappende områder i et kontinuum.

Se fig 14.12 og fig 14.13. I tillegg til den relativt ordnede prosessen vi så på, finnes det dessverre også en annen prosess – ofte kalt ”quick fix” prosessen. Å kalle den lurvete er antakelig en underdrivelse. Mange av de problemene man etter hvert får med store programvaresystemer skriver seg fra denne og liknende vedlikeholdsprosesser.

Når man har gjort et sett av endringer til et programvaresystem er det vanlig å foreta en ny leveranse. Denne må tilfredsstille følgende krav:

- Den funksjonaliteten som fungerte før må stadig være på plass.
- De feilene man har ment å rette må være borte
- Den nye funksjonaliteten må fungere som planlagt.

Det er ikke nok å oppdatere programvaren. Hvis det er gjort endringer, altså ikke rettinger, må man også oppdatere den dokumentasjonen (for eksempel brukerveiledningen) som gjenspeiler endringene.

Det er flere måter å planlegge leveranser på. Måten vi gjør det på vil også avhenge om vi har en - eller noen få - kunder eller om vi leverer et system på det åpne marked:

- Fast stab og variabel plan. Man har en fast vedlikeholdsstab og faste leveranse intervaller – for eksempel hver sjette måned. Man implementerer endringene etter hvert som de kommer og tar med så mange som man rekker fram til neste leveranse. Vi leverer altså alltid i tide, men med varierende innhold. Løsningen er fleksibel, men kundene vet ikke hva de får på forhånd.
- Variabel stab og fast plan. Man starter med å se på hva man vil gjøre før neste leveranse. Her er det viktig at vi har en klar prioritering. Deretter estimerer man nødvendig innsats og setter inn de nødvendige ressursene for å holde planen. Kunden får de N viktigste endringene, men han har ingen kontroll over N.
- Variabel stab og variable planer. Dette er den mest fleksible måten å drive vedlikehold på. Som ventet blir den også den mest kostbare pga. mye planlegging og estimering. Hvilke endringer som skal gjøres blir avtalt med kunden ut fra dens behov. Ut fra dette blir det laget en plan med tid og kostnader, noe som igjen vil bestemme antall utviklere vi trenger. Dette opplegget vil gi kunden best muligheter til å påvirke vedlikeholdsopplegget.

Den løsningen vi velger vil avhenge av flere ting, sånn som:

- Vår generelle ressurssituasjon. Har vi lite folk å sette på vedlikehold vil den første løsningen være den eneste farbare.
- Vedlikeholdskontrakten. Vi kan ha en kontrakt som forplikter oss til å ha et visst antall personer engasjert i vedlikehold eller som forplikter oss til å løse alle problemer av prioritet A innefor et gitt tidsrom osv.
- Det generelle forholdet til kunden. For store, viktige kunder vil vi vanligvis være innstilt på å yte best mulig service – også når det gjelder vedlikehold. Dette kan tvinge oss til å velge det siste alternativet.

Når et system er vedlikeholdt over lang tid vil strukturen gradvis bli dårligere og dårligere. De to viktigste årsakene til dette er at:

- Vi har ikke tid / råd til å rydde etter oss pga. av press på ressursene.
- Mange utvidelser går på tvers av den opprinnelige strukturen og burde strengt tatt ført til en restrukturering av systemet – som vi ikke har tid til.

Alt dette fører til en oppmagasinering av behov for å rydde i systemet med jevne mellomrom. Det er flere strategier for å planlegge dette. Det etterfølgende er bare noen av mange muligheter:

- Se på kohesjon og kobling. Prøv å ”beveg” hver enkelt modul mot løsere kobling og sterkere kohesjon. De beskrivelsene vi så på sist kan være gode retningslinjer.
- Vurder hvordan modulene ligger an i henhold til verdiene på fan-in / fan-out. Vurder først de modulene med de største verdiene.
- Se på endringstrafikken. De modulene der det er endret mest – enten i antall ganger eller i prosent av antall linjer endret – er de mest populære kandidatene for opprydding. Vær imidlertid klar over at det ofte er forskjell på hvor vi endrer og hvor vi *burde* ha endret. Dette avhenger bla. av hvor enkelt det er å endre den eksiterende koden. Det finnes altfor mange eksempler på at kode som burde vært endret ikke blir det fordi ingen tør. I stedet endrer man i andre moduler for å få fram den ønskede effekten.

Det er neppe lurt å bruke disse reglene slavisk. Se på de som en måte å styre oppmerksomheten på. Før man endelig bestemmer seg for hva man skal gjøre bør det nok en mer detaljert analyse til.

03-11 Kapittel 6

Det er to sider ved kvalitet – produkt og prosess. Hovedsaken er produktet, men det ligger underforstått i mye av kvalitetsarbeidet at vi får god kvalitet på produkter ved å ha en god prosess. Før vi ser mer på dette må vi definere kvalitet. ISO har definert kvalitet på følgende måte:

”Helheten av egenskaper en enhet har og som vedrører dens evne til å tilfredsstille uttalte og underforståtte behov”

Det finnes også andre definisjoner på kvalitet, for eksempel:
- Egnethet – ”fitness for use”. Denne definisjonen er i slekt med ISO definisjonen.
- Overensstemmelse med kravspesifikasjon – ingeniørdefinisjonen. Den har den fordelen at den til en viss grad er objektiv.
- Noe flott. Dette er populærdefinisjonen. Etter denne definisjon vil for eksempel en BMW ha høyere kvalitet enn en Volvo, mens dette ikke behøver å være tilfelle med de andre definisjonen

Det er to ting vi må legge merke til:
- Målet er å tilfredstille kundens behov. Uten kunde er det fåfengt å snakke om kvalitet. Kvalitetsvurderingen er derfor avhengig av hvilken kunde vi har med å gjøre er som sådan rent subjektiv.
- Definisjonen inkluderer både eksplisitte – uttalte – og implisitte – underforståtte – behov. De eksplisitte behovene kommer fra krav, men de implisitte behovene stort sett kommer fra kunnskap og forventninger knyttet til domenet.

Ut fra det faktum at kvalitet er en subjektiv oppfattelse følger at mange av de viktige kvalitetsmålene også må være subjektive.

Det er likevel et behov for å kunne måle / vurdere kvalitet, både når vi lager kravspesifikasjon og når vi skal godkjenne / akseptere et nytt system. ”Vi kan ikke forstå det vi ikke kan måle” sa lord Kelvin. På en annen side ”Vi kan ikke måle det vi ikke forstår” som en eller annen frustrert utvikler formulerte det. Derfor – forståelse og måling må skapes over tid og i en iterativ prosess.

For å kunne snakke om produktkvalitet på en ryddig måte trenger man en nedbryting av begrepet i et sett av underbegreper. Den vanligste måten å gjøre dette på er å lage en faktor – kriterier – metrikker modell. En av de eldste, men stadig mye brukte, slike modeller skriver seg fra McCall.

ISO har definert og standardisert et sett med kvalitetsattributter. De fleste faktorene er de samme som hos McCall, men de er organisert på en annen måte. ISOs måte å gjøre dette på er samlet i standarden ISO 9126. Vi vil se på hvert enkelt attributt, hva det betyr og hvordan man kan vurdere eller måle noen av de. **Vis fig fra ISO 9126**. den lista vi ser på er noe forskjellig fra den læreboka bruker, men den er i det minste konform med ISO 9126.
- Funksjonalitet – systemets evne til å gi funksjonalitet som tilfredsstiller eksplisitte og implisitte behov
- Pålitelighet – systemets evne til å opprettholde sin ytelse
- Brukbarhet – hvor lett er det å lære å forstå hva systemet gjør og hvor godt liker brukerne systemet
- Effektivitet – hvor godt opprettholder systemet sin ytelse for en gitt tilgang på ressurser.
- Vedlikeholdbarhet – hvor lett er det å endre systemet
- Bærbarhet - hvor lett er det å flytte systemet fra en omgivelse til en annen

Dette kalles kvalitetsfaktorer. Hver enkelt faktor er definert av et sett av kriterier. Disse er som følger:
- Funksjonalitet
    - Nøyaktighet – resultat som er riktig eller i henhold til en avtale.
    - Egnethet – et sett av funksjoner som er egnet til å kunne brukes for å løse brukerens problemer / oppgaver.

    - Overensstemmelse – overensstemmelse med gitte standarder, regler og konvensjoner i det gitte applikasjonsområdet.
    - Sikkerhet (i betydningen security) – hindre uautorisert tilgang til å se eller endre data. Dette gjelder både tilgang som følge av "angrep" og som følge av uhell.
- Pålitelighet
    - Modenhet – ikke feile pga. feil i koden.
    - Feiltoleranse – fungere som planlagt selv om den får feil input eller blir brukt på feil måte.
    - "Recoverability" – komme raskt opp igjen og miste få / ingen data eller transaksjoner. Reparasjon.
    - Tilgjengelighet – være tilgjengelig når vi trenger systemet. En funksjon av feilrate og reparasjonsrate (MTTF og MTTR).
- Brukbarhet
    - Forståelighet – hvordan kna systemet hjelpe meg med å få gjort mine oppgaver.
    - Lett å lære – lett å lære seg å bruke systemet til å få gjort sine arbeidsoppgaver.
    - Lett å bruke – lett å få gjort det man ønsker, lett å kontrollere systemets oppførsel. Ingen negative overraskelser.
- Effektivitet
    - Tidseffektivitet – rask responstid
    - Ressurseffektivitet - bruke de tilgjengelige ressursene (plass, sekundær lagring, etc) på en effektiv måte
- Vedlikeholdbarhet
    - Analyserbarhet – lett å finne ut hvor en feil er eller hvor vi må endre for å få til en gitt effekt. Dette er i stor grad en funksjon av **sporbarhet,** kohesjon og enkel kode**.**
    - Endringsvennlighet – hvor lett er det å endre i koden.
    - Stabilitet – lett å endre ett sted uten å få en masse uventa effekter rundt omkring. Dette er i stor grad avhengig av sterk kohesjon og svak kobling.
    - Testbarhet – hvor lett er det å sjekke at vi har oppnådd det vi ville med endringene uten å ødelegge det som fungerte før.
- Bærbarhet
    - Tilpassningsdyktighet – kan enkelt tilpasses til nye omgivelser, så som nye operativsystemer, nye nettverksløsninger osv.
    - Installerbarhet – lett å installere, både i det planlagte miljøet og i et nytt miljø
    - Sameksistens – kunne eksisytere / fungere sammen med andre systemer uten å lage problemer.
    - Konformitet – overensstemmelse med standarder og konvensjoner (god programmeringsskikk) som angår bærbarhet
    - Erstattbarhet – kan erstatte (bli brukt i stedet for) annen programvare med samme funksjonalitet som allerede finnes i systemet.

ISO hadde som mål å også definere et sett av metrikker som gjorde at man på en entydig måte kunne gi tallverdier til hvert enkelt kriterium, men det viste seg at det var umulig å bli enige om et slikt sett av metrikker

Ved å rangere hver enkelt faktor kan vi komme fram til en verdi som sier noe om kvaliteten på produktet. Dette kan brukes til å sette krav til produktkvaliteten. Vi kan for eksempel gjøre følgende:

- Hvert kriterium i gies en vekt W_i.
- Vi definerer en evalueringsprosess – test eller annet – som kan gi kriteriet en verdi mellom 0 – ikke til stede i det hele tatt – til 1 – kravene er helt oppfylt. Dette er kriterieverdien C_i. Eventuelt kan vi ha flere mål som kombineres - metrikkene M_{ij} .
- Kvalitetstallet blir da gitt som $\text{Sum}(W_i * C_i)$

Et alternativ er å stille spesifikke krav til hver enkelt kvalitetsfaktor. Dette er vanligvis enklere, både for kunde og utviklere. Et eksempel er som følger:

- Funksjonalitet – systemets evne til å gi funksjonalitet som tilfredsstiller kundes behov
  Velg ut 5 brukere. Disse skal bruke systemet i tre dager og deretter gradere funksjonaliteten på en skalla fra – til 5. Minst 3 skal gi en karakter på 4 eller 5 og ingen skal gi det en karakter dårligere enn 3.
- Pålitelighet – systemets evne til å opprettholde sin ytelse
  Ingen feil skal inntreffe i løpet av tredagersperioden der vi tester funksjonaliteten.
- Brukbarhet – hvor lett er det å lære å forstå hva systemet gjør og hvor godt liker brukerne systemet
  Brukerne i funksjonalitetstesten skal få et en dags kurs i å bruke systemet. Deretter skal de fylle ut et spørreskjema som sjekker deres forståelse. Ingen skal få mindre enn 75% riktig på denne undersøkelsen.

- Effektivitet – hvor godt opprettholder systemet sin ytelse for en gitt tilgang på ressurser.
  Responstiden på den avtalte konfigurasjonen skalregistreres automatisk under hele testperioden. Midlere responstid skal være 0.1 sekund. Ingen transaksjon skal ta mer enn 2 sekunder.
- Vedlikeholdbarhet – hvor lett er det å endre systemet.
  Leverandøren vil få tilsendt to endringsønsker angående layout av skjermbilde etter en dag av testperioden. Etter at leverandøren har akseptert endringene skal endringen være tilgjengelig for brukerne etter en dag.
- Bærbarhet - hvor lett er det å flytte systemet fra en omgivelse til en annen
  Ingen spesielle krav.

Uansett hvilken måte vi velger å spesifisere kvalitetskravene på er det viktig å huske at det som vi ikke setter testbare krav til, det frå vi ikke oppfylt.

Flere av de definerte kvalitetsfaktorene er generelt i motsetning til hverandre – det er vanskelig å oppnå alle samtidig. Skal man få høy skår på det ene er det ikke nødvendigvis slik at man kan få høy skår på alle andre. **Vis fig. 6.7 og kommenter**. Dette er riktignok McCalls modell, men ideen er generell.

Perspektiver på kvalitet

Vi har allerede sett ISOs definisjon og så vidt berørt et par andre definisjoner. Her skal vi prøve å være litt mer systematiske. Boka snakker om fem perspektiver vi kan ha på kvalitet:

- Transcendental – det uutsigelige. Jeg vet hva kvalitet er, men jeg kan ikke forklare det. Jeg kjenner det igjen når jeg ser det.
- Brukerbasert – egnethet for bruk. Hvor godt tilfredsstiller produktet mine behov.
- Produktbasert – for eksempel lav kobling og høy kohesjon, lett lesbar kode, laget i henhold til god utviklingsskikk.
- Produksjonsbasert – her heller utviklingsbasert – i overensstemmelse med de beskrevne krav. Ofte kalt ingeniørdefinisjonen.
- Verdibasert - lønner det seg å ta dette i bruk, tjener jeg mer enn det som det koster meg å kjøpe og drive systemet.

ISO prøver å ta opp i seg alle disse perspektivene unntatt det første. Det siste perspektivet – verdibasert er imidlertid ikke så altfor godt ivaretatt. Den verdien vi får ut det å ta i bruk et nytt system kan være flere ting. Det etterfølgende er noen eksempler:

- Økt effektivitet – spare penger
- Økt effekt – bedre informasjon og bedre grunnlag for å ta gode beslutninger, bedre resultater generelt.
- Ekstra verdi – vi kan gjøre ting vi ikke kunne gjøre før.
- Nytt produkt – vi kan lage / selg noe vi ikke kunne lage / selg før. Dekker både produkter og tjenester.
- Bedre struktur på info. Vi kan få mer ut av de andre systemene i bedriften som bruker eller gir info som vi trenger idet daglige arbeidet.

Kvalitetssystemer

Hovedhensikten med et kvalitetsystem er å

- Ha ryddige kontraktsforhold. Dette inkluderer en ryddig spesifikasjon av hva som skal gjøres. Dette impliserer ***ikke*** at all funksjonaliteten må være bestemt på forhånd. Det er for eksempel fult mulig å ha en kontrakt som bestemmer at man skal ha et forprosjekt der man definerer kravene til systemet eller en kontrakt som sier at man skal bruke XP, som er en ekstremt iterativ og inkrementell utviklingsmodell.
- Gi trygghet for at kunden får det han er lovet – både når det gjelder produkt (det han får til slutt) og prosess (måten vi lager det på). Tilliten kommer primært fra prosessen. Vi kan ikke velge en leverandør ut fra detaljkunnskap om de folka han setter til å gjøre jobben. Vi kan imidlertid velge ut fra graden av ryddighet i prosessen.

Standarden skal sørge for at vi har en prosess som gjør at vi fokuserer på å oppfylle kundens krav – både de funksjonelle – hva systemet skal gjøre – og de ikkefunksjonelle kravene – hvordan systemet skal gjøre det.

Fig 6.10 viser innholdsfortegnelsen for standarden ISO 9001. Se også vedlegg A i boka. Denne standarden er imidlertid laget fro all produksjon av varer og tjenester. Problemet vårt er at programvare ikke blir produsert på

samme måten som jagerfly eller lenestoler – den blir utviklet. Når det gjelder produksjon – som stort sett går ut på å brenne CD-er – er programvarebransjen verdens beste og mest effektive bransje.

Kvalitetsystemer for programvare

ISO 9001 er en generell standard. For å få den mer tilpassa til programvare er det laget en veiviser for bruk av ISO 9001 i programvare. Denne heter ISO 9000-3. boka har imidlertid valgt i stedet å se på IEEE Std 730. denne finnes i appendiks B i læreboka. Innholdsfortegnelsen for denne standarden er vist i **fig. 6.11**. Vi skal gå kort gjennom de viktigste punkta i denne standarden.

- Hensikt – hvilket produkt angår denne planen. Forskjellige produkter vil ha forskjellige planer, bla. fordi konsekvensen ved å feile er forskjellige, kundens krav er forskjellige osv.
- Refererte dokumenter – hvilke dokumenter refererer denne planen til.
- Ledelse – hvem er ansvarlige for hva i prosjektet. Her finner vi ledelsesstruktur, kommandostruktur, hvem som kan ta hvilke beslutninger, hvem som kan godkjenne hva osv. Dette inkluderer også hvem som kan beslutte at vi skal kunne ta snarveier gjennom systemet.
- Dokumentasjon – hvilke dokumenter skal lages, hvordan skal de se ut, hvordan skal de godkjennes. Følgende er minimum:
    - Kravspesifikasjon
    - Designbeskrivelse
    - Testplan
    - Testrapport
    - Brukerdokumentasjon
    - Konfigurasjonsstyring
- Standarder, praksis, konvensjoner og mål – hvilke standarder skal gjelde, regler for god praksis og konvensjoner. Dette kan inkludere alt fra kodestandarder til navnekonvensjoner for dokumenter. Det må være med en beskrivelse av hvordan vi skal sjekke at reglene / standardene blir fulgt.
- Gransking og revisjoner – hvordan skal granskinger og revisjoner utføres. Dette gjelder både for dokumenter (inklusive kode) og for QA planen.
- Testing – hvordan skal vi teste moduler, subsystemer og systemet. Ofte kan vi klare oss med en henvisning til testplanen.
- Rapportering og retting – korrektive tiltak. Prosedyrer for å rapportere feil og sørge for at de blir behandla på en riktig måte – analysert, prioritert og retta.
- Verktøy teknikker og metoder – hvilke skal vi bruke i dette prosjektet. Alle verktøy, metoder og teknikker må være beskrevet. For verktøy må det også være med versjon, slik at alle bruker det samme.
- Kodekontroll – hvordan vi tar vare på kode, sørger for at det som er godkjent ikke blir endra uten at de som skal bruke den får beskjed, hvem som er ansvarlig for dette.
- Mediekontroll – hvordan beskytter vi de fysiske mediene så som disker, disketter, CD-er og taper
- Kontroll over underleverandører – hvordan sikrer vi kvaliteten til de som utvikler deler av systemet for oss. Det blir mer og mer vanlig at bedrifter gjør det de er gode til og så setter de bort resten til andre bedrifter. Det er flere måter å gjøre det på, for eksempel
    - Stille krav til underleverandørens QA-plan
    - Stille krav til underleverandørens testprosedyrer
    - Ha en person utplassert der for å sjekke at de jobber ordentlig – i henhold til avtalte prosedyrer og metoder
- Data innsamling og lagring – hvordan blir QA aktivitetene dokumentert og kontrollert basert på innsamla data
- Opplæring – hvilken opplæring er nødvendig for at denne QA-planen skal virke. Ofte er dt nødvendig å lære utviklerne teknikker oms dokumentgransking, innsamling av testdata osv. Dette må planlegges på forhånd, bla. for å sikre at det blir satt av ressurser til det.
- Risikostyring – hvordan blir risiko handtert i QA. Legg merke til at risikoanalysen er gjort i prosjektplanen. Her er det snakk om hvordan vi aktivt bruker resultatet av risikoanalysen i løpet av prosjektet.

Det å ha en QA-plan garanterer ikke et godt resultat. Det er ingen erstatning for kompetente utviklere, et godt programmeringsspråk og gode utviklingsomgivelser. Det kommer ***i tillegg*** til at dette for at hver enkelt i prosjektet skal kunne bidra til et godt sluttresultat på den mest mulig effektive måten.

I det øyeblikket vi begynner å gjøre ting bare fordi det står i QA-planen er vi på ville veier. Hensikten er å gi kunden et best mulig produkt til avtalt tid og kostnad. Det som kommer i tillegg bør man se svært nøye på. Nyttige ting som kommer i tillegg kan for eksempel være at:

- Bedriften / utviklerne skal lære av erfaringene på en systematisk måte
- Vi ønsker å forbedre prosessen
- Vi ønsker å lage komponenter 7 subsystemer som vi skal gjenbruke i senere prosjekter

Alt dette må imidlertid være planlagt. Det er dumt å gjøre en masse ekstraarbeid fordi det muligens kan komme til nytte.

03-18 Kapittel 17

For å få til gjenbruk må vi se på tre viktige faktorer:
- Gjenbruk av komponenter
- Gjenbruk av design
- ”Lim” for å sette sammen komponenter.

Alternativt kan vi se på det boka kaller dimensjonene til gjenbruk. Forfatteren opererer med seks dimensjoner:
- Substans – det man gjenbruker. Dette er komponenter i en eller annen form. Komponentene kan angå produkt eller prosess:
    - Produktkomponenter kan være alt fra subsystemer til små ”kodesnutter”. Komponentene kan være generiske – for eksempel en praktisk datastruktur – eller spesialiserte ting.
    - Prosesskomponentene kan være beskrivelser av hvordan man gjør deler av en jobb, for eksempel hvordan man bør gå fram når man skal gjøre kodegranskning.
- ”Scope” – vi skiller mellom horisontal og vertikal gjenbruk. Vi ser at jo smalere domener vi opererer med, desto bedre sjanser får vi for at det vi lager kan gjenbrukes i nye systemer i dette domenet. Derimot vil så spesialiserte komponenter ha liten sjanse for gjenbruk utenfor domenet.
    - Horisontal gjenbruk er gjenbruk av generelle komponenter – for eksempel et matematisk bibliotek.
    - Vertikal gjenbruk er gjenbruk av domenespesifikke komponenter, for eksempel gjenbruk av en lageradministrasjonsrutine når vi lager et nytt lagersystem.
- Tilnærming – vi opererer med to former for gjenbruk:
    - Planlagt, systematisk – her identifiserer vi komponenter som skal lages gjenbrukbare og legger ekstra innsats ned i dokumentasjon og testing. Dette må sees på som en investering og behandles deretter.
    - Opportunistsk, ad hoc – folk gjenbruker det de måtte ha liggende av kode. Det de gjenbruker er ting man gjør i løpet av mange prosjekter, så som I/O rutiner. Unix filosofi med ”pipes” og filter passer godt inn her.
- Teknikk – igjen har vi to metoder:
    - Komposisjon – vi setter sammen eksisterende komponenter til et nytt system. Hovedproblemet er å finne passende komponenter, enten fordi de er vanskelige å identifisere eller fordi vi ikke har de.
    - Generativ. Her gererer vi nye komponenter ut fra for eksempel ”patterns”.
- Bruk. Her skiller vi mellom.
    - ”White box” gjenbruk. Her endrer vi deler av komponenten. Dette skal gjøres med forsiktighet. Data indikerer at viss vi endrer mer enn 25% avkoden er det like billig å lage komponenten på nytt.
    - ”Black box” gjenbruk. Komponentene brukes som den er. Ofte er dette COTS komponenter, der vi ikke en gang kjenner innholdet – bare grensesnitt og (deler av) funksjonalitet
- Produkt – hva gjenbruker vi. Mulighetene er mange – kildekode, design, arkitektur, klasser osv. Det er imidlertid mest vanlig å gjenbruke kode. Imidlertid er det en økende trend mot å bruke design ”patters” etter som disse blir gjort kjent.

Vi vil se på gjenbruk av forskjellige typer av artefakter. Dette inkluderer:
- Biblioteker.
- Maler
- Design
- Arkitektur

Biblioteker

Den største suksessen her er gjenbruk av matematiske biblioteker. Viss vi sjekker hvorfor dette er vellykka finner vi følgende momenter:

- Standardisert terminologi. Dette skyldes at fagfeltet er veletablert med et felles vokabular.
- Lite, veldefinert grensesnitt for de fleste funksjoner. Det er almen enighet om hva slags info som skal inn og hva vi får ut igjen. Det enste som skiller er info om feilhendelser, udefinert input osv.
- Standardisert dataformat – vi trenger sort sett heltall og reelle tall og de har begrensa variasjonsmuligheter.

I de tilfellene de momentene som er nevnt over ikke er tilstede, blir det mer vanskelig. Generelt har man identifisert fire viktige momenter:

- Søking etter komponenter – det å finne riktig komponent er kritisk. Vi kan ha enkel søking som setter krav til valg av riktige søkeord, eller vi kan ha en avansert søkemekanisme, som vil sette store krav til metoden for å legge inn nye komponenter.
- Forstå komponenter – for å gjenbruke en komponent vil vi gjerne forstå hva den gjør. Dette gir tillit og uten tillit til komponentene tør vi ikke gjenbruke de.
- Tilpassing av komponenter – viss ikke komponenten er direktegjenbrukbar må vi modifisere den dette stiller store krav til vedlikeholdbarhet viss det skal lønne seg.
- Sammenstilling – hvordan skal vi ”lime” sammen komponentene.

Når man lagrer komponenter i en database bruker man ofte metoder fra biblioteker – særlig forskjellige former for indeksering. **Fig. 17.3** viser et eksempel. Begrepene har følgende betydning:

- Ukontrollerte indekser – vi kan velge de termene vi har lyst til.
- Kontrollerte indekser kan være:
    - Ei liste av lovlige ord – nøkkelord.
    - Et klassifikasjonsskjema definert som en trestruktur. Selv om det begynner som en trestruktur går det vanligvis raskt over til å bli et nettverk pga. multippel klassifikasjon.

Det viser seg vanskelig å få til et opplegg som både er effektivt å søke i og lett å legge inn nye komponenter i. Et nett med fasetter – gjerne med forskjellige vekter avhengig av hvor nær hverandre begrepene er – gir en effektiv søk, men kostnadene ved å legge inn nye komponenter er høy og øker for hver ny komponent som legges inn.

Når vi skal velge om vi vil gjenbruke en komponent eller ikke er det mange ting som er viktige i tillegg til hva komponenten gjør. Dette gjelder bla.

- Kvalitet – hvor lett er komponenten å endre, hvor robust er den mot feil input osv.
- Administrativ info – hvem har laget denne komponenten, hvor / til hva har den vært brukt før, hva slags erfaringer har andre med denne komponenten, osv. Dette er innfor som gir tillit til komponenten.
- Dokumentasjon – vanlig programvaredokumentasjon, særlig viktig for vedlikehold / endringer.
- Beskrivelse av grensesnitt – hva slags parametere, hva slags dataformat osv.
- Testinfo. Dette kan både være testlogger fra tidligere tester, nyttige tester å kjøre osv.

Generelt sett vil det være mer å tjene på gjenbruk dess større komponenten er. På den andre siden er det vanskeligere å finne store, gjenbrukbare komponenter, særlig fordi vi ikke ønsker å ta med oss en masse funksjonalitet som vi ikke trenger. Store komponenter er ofte spesialiserte og derfor vanskelig å gjenbruke.

For å komme i gang med gjenbruk må noen fylle biblioteket med gjenbrukbare komponenter. Dette er en kritisk suksessfaktor. Når folk søker i biblioteket og ikke finner noe de kan bruke vil de snart slutte å bruke det. Voss de derimot finner noe nyttig vil de bli motivert til å søke flere ganger og bli motivert tikl å legge inn nye komponenter. På den måten kan vi få en god sirkel.

Maler

Maler faller i to kategorier:

- Dokumentskjeletter for rapporter, programvaredokumentasjon, brukerdokumentasjon osv. IEEE standardene faller i denne kategorien.
- Kodeskjeletter – ufullstendig kode som er knyttet til spesielle applikasjonstyper. Dette er kode som er halvferdig – utvikleren må fylle inn resten. For eksempel kodeskjelett får å lese ei pakke fra en fil, velge aksjon ut fra et typefelt i pakka og skrive ny info ut på ei ny fil.

Design

Det er en flytende overgang mellom gjenbruk v8a kodeskjeletter og gjenbruk av en design. Mange bruker til og med kodeskjeletter til å dokumentere design. Ofte vil man ha ferdige prosedyreskjeletter eller klassestrukturer som definerer de overordna designet. Utviklerne må fylle inn kode selv.

Vi kan også gjenbruke design i form av sekvensdiagrammer, gjerne kombinert med klassediagrammer.

Arkitektur

Vi har stadig de samme basisideene som ved maler og design, men har flyttet oss et trinn oppover i abstraksjon. Det finnes få vellykka eksempler på gjenbruk på dette nivået.

Gjenbruk og livssyklus

Det finnes to viktige aktiviteter her. Begge opererer stort sett på komponentnivå.

- Utvikling med gjenbruk. Vi finner gjenbrukbare komponenter i en skuff eller i en database. Vanligvis er dette noe vi eller noen andre har laget før, men ikke nødvendigvis for at noen skal gjenbruke det.
- Utvikling for gjenbruk. Vi lager komponenter for at noen andre skal kunne gjenbruke de.

Utvikling med gjenbruk

Gjenbruket skjer på komponentnivå. Vi kan imidlertid ikke vente til kodefasen med å tenke gjenbruk. Vi må tenke gjenbruk helt fra vi begynner å se på arkitekturen. **Vis fig. 17.5** og kommenter. Det å oppnå en stor gjenbruksandel vil være et internt krav til utviklingsprosjektet. Dette kan / vil medføre at vi i stor grad må bygge opp systemet rundt de komponentene vi velger å gjenbruke. Som en konsekvens av dette får vi en iterativ prosess som inneholder søking i databasen, arkitektur og overordna design.

Utvikling for gjenbruk

Utvikling for gjenbruk må komme først ,ellers er det jo ingen ting å gjenbruke. **Fig. 17.6** viser en mulig prosess for dette. Legg merke til at det ikke er nødvendig å ha mer enn ett planlagt prosjekt i domeneanalysen. Det er tilstrekkelig å vite / anta at man skal lage senere prosjekter der man kan få til utvikling med gjenbruk.

Beskrivelsesspråk

Det er definert flere språk som understøtter gjenbruk. Mange har prøvd å innføre skille mellom ”programming in the large” – systemprogrammering – og ”programming in the small” – komponentprogrammering. Et eksempel på hva som har kommet ut av dette er språket MIL – ”Module Interconnection Language”. Et eksempel er vist **i fig 17.7**. kommenter bruken av nøkkelorda ”provide”, ”require” og ”implentation”.

Her beskriver vi et abstrakt system som består av flere moduler. De tre første modulene blir kalt henholdsvis ”control_mod”, ”input_mod” og ”store_mod”. Hver av modulene har et sett av prosedyrer og / eller pakker.

Selv om vi ikke har noen form for kompilator som kan handtere dette så kan beskrivelsen være nyttig fordi den viser strukturen i systemet på en måte som utviklere vil oppfatte som enkel.

Utvikling av gjenbruk

Vi har tidligere sett at matematikkpakker er ett av de områdene der gjenbruk har vært svært vellykka. Mange ser ikke på dette som gjenbruk – det er bare funksjoner som er tilgjengelige, akkurat som I/O rutiner er tilgjengelig i de fleste språk. Dette har fått boka til å identifisere fire faser i gjenbruk:

1. All programvare blir skrevet fra bunn av. Domenet er lite utforsket og oppdeling i pakker, klasser eller prosedyrer er ad hoc. Det er lite eller ikke noe gjenbruk.
2. Vi begynner å se de samme problemene med de samme løsningene dukke opp i flere sammenhenger innefor domenet. Vi går gjennom en prøving og feilingsfase for å finne praktiske, abstrakte konsepter. All gjenbruk er ad hoc.
3. I denne fasen vil vi begynne med gjenbruk. Terminologien i domenet her begynt å stabilisere seg. Dette øker kommunikasjon og sannsynligheten for at noen lager komponenter som er relatert til denne terminologien. Vi har begynt å gjenbruke eksisterende komponenter på en systematisk måte fra prosjekt til prosjekt.

4. Sluttfasen. Domenet er gjennomarbeidet og forstått helt og fult. Utvikleren lager ikke programvare for dette domenet og vi tenker ikke engang på dette som gjenbruk. I stedet har vi et sett av standardkomponenter som alle bruker.

Det ser ut til at det ikke er lurt å prøve å hoppe fra fase 1 til fase 3 uten å ha vært innom fase 2. Vi trenger denne fasen for å samle erfaring nok til å bygge opp det beste, mest brukbare settet av abstraksjoner for terminologien. Det ser altså ut til at vi må ha et evolusjonært syn på gjenbruk – i det minste for bransjen som helhet. Situasjonen vil antakelig være noe annerledes innefor en bedrift.

Ikke-tekniske problemer

Som innefor mange andre områder i systemutvikling er det også mange problemer innenfor gjenbruk som ikke er rent tekniske og derfor ikke kna ha rent tekniske løsninger. Vi skal se på tre områder – ledelse, økonomi og prosjektregnskap.

Ledelse

For at gjenbruk skal fungere i en bedrift er det nødvendig at:

- Ledelsen støtter innføring av gjenbruk. Som vi senere skal se vil effektivt gjenbruk medføre investeringer både i form av kode og opplæring og dette er det vanskelig å få til uten ledelsens godkjenning.
- Man oppmuntrer til gjenbruk eller forlanger en god begrunnelse for å lage en komponent selv viss det allerede finnes en komponent som kan gjøre jobben. Boka påstår at utviklere lider av NIH – ”Not Invented Here” syndromet, men det er lite som tyder på det når man prøver å finne ut hva som fremmer gjenbruk. NIH syndromet fungere nok mer som en unnskyldning når man ikke får til gjenbruk. Den største hindringen for gjenbruk ser ut til å være manglende tillit til komponenter.
- Man utfører en domene analyse slik at man vet hva det er teknisk gunstig å lage av gjenbrukbare komponenter.
- Man tenker gjenbruk fra starten av prosjektet. Gjenbruk er ikke noe som kan vente til slutt. Ut fra dette er inkrementell utvikling antakelig mest effektivt viss man vil oppnå høy gjenbruksgrad.

Økonomi

For at gjenbruk skal kunne lønne seg kreves det investeringer. Det er et problem at mange programvarebedrifter totalt mangler et investeringsbegrep og en internrente. Uten dette kna de ikke handle økonomikk rasjonelt – og det gjør de da ofte ikke heller. Når vi utvikler gjenbrukbare komponenter finnes det tre kostnader vi ikke kommer unna, nemlig kostnadene forbundet til:

- Utvikling av komponenten. Dette er kostnader som blir absorbert av det enkelt prosjekt og derfor blir dekket av kundene.
- Å gjøre komponenten gjenbrukbar. Dette er kostnader til mer dokumentasjon slik at andre kna forstå hva komponenten gjør og hvordan den gjør det, ekstra testing for å gi mer tillit og ekstra kostnader i kvalitetssikringen for å få en høy kvalitet på komponenten. Dette er den største kostnaden.
- Drift og vedlikehold av et komponentbibliotek. Denne kostnaden vil i de fleste tilfeller være liten.

Når vi skal gjenbruke noe kommer det til to nye kostnader:

- Søke i biblioteket for å finne gjenbrukbare komponenter. Dette er vanligvis en liten kostnad, men den medfører at det er liten vits i å gjenbruke små komponenter viss vi må bruke mye tid på å finne.
- Gjøre eventuelle modifikasjoner for tilpassning. Dette kan fort bli dyrt og enkelte hevder at det bare er gjenbruk med meget små endringer som egentlig lønner seg.

Det er noen kritske spørsmål som må besvares før man tar beslutningen om å lage en komponent gjenbrukbar:

- Hva er sannsyneligheten for at vi kommer til å gjenbruke denne komponenten?
- Hvor mange ganger kommer vi til å gjenbruke denne komponenten?
- Når kommer vi til å gjenbruke denne komponenten?

Den informasjonen man får fram kan man bruke på flere måter. Vi skal se på to – en enkel, men kanskje bra nok og en som bruker ideer fra investeringsanalysen.

- Enkel tilnærming: Vi bestemmer vår planleggingshorisont. Dette kan være hva som helst fra 12 måneder til 4 – 5 år. Det vil antakelig ikke lønne seg viss:

    - Sannsynligheten for gjenbruk er vurdert som lav
    - Første gjenbruk ligger mer enn ett til to år fram i tid
    - Det neppe blir mer enn ett til to gjenbruk av komponenten.
- Avansert tilnærming. Innfører følgende notasjon:
    - Utviklingskostnader i år i er U_i. Legg merke til at kostnadene ved å lage en komponent vil endre seg – den vil antakelig synke over tid.
    - Internrente er r.
    - Første utvikling koster U, med et tillegg U_g for å gjøre komponenten gjenbrukbar.
    - Antall gjenbruk er N

  Hvert gjenbruk gjør at vi sparer ca. U_i kroner. Redusert til nåverdi blir dette $U_i / (1 + r)^i$. Tiden til gjenbruk og intern-renten påvirker altså resultatet. Viss vi sumerer dette for alle år der vi gjør gjenbruk finner vi ut hva vi har spart: Spart = SUM($U_i / (1 + r)^i$). Dette må være større enn investeringen som er U_g .

L oss se på et enkelt eksempel. En modul koster 200 timeverk å utvikle. Internt blir dette vurdert til 500 * 200 = 100 000 kroner. For å få den gjenbrukbar må vi legge på 100 timeverk til – altså 50 000 kroner som er en investering. Bedriften bruker ei internrente på 5%. Vi regner med å kunne gjenbruke denne modulen tre ganger – en gang i hvert av de tre neste åra. Vi ser ikke for oss at det blir noe særlig enklere å lage slike moduler i planperioden.

Det vi sparer, redusert til nåverdi (NPV) er 100 000 * ($1 / 1.05 + 1 / 1.05^2 + 1 / 1.05^3$) som er omtrent lik 2.7 * 100 000 eller 270 000 kroner. Vi ser at dette lønner seg. Viss vi bare får ett gjenbruk og det skjer om tre år så er NPV av innsparinga bare 100 000 * $1 / 1.05^3$, som er omtrent lik 86 000 kroner om stadig er en grei investering.

Endringer i utviklingskostnader og internrente vil kunne forandre bildet. Igjen – dette bør ikke være den eneste input i beslutningsprosessen , men det vil være nyttig å være klar over.

Ellers kan det være greit å se på forholda som illustreres i **fig. 17.11** for å forstå hvorfor mange bedrifter ikke kommer i gang med gjenbruk i stor skala.

Prosjektregnskap

Gjenbruk behøver ikke nødvendigvis å være et godt valg for et utviklingsprosjekt. La oss se på et dessverre ofte forekommende forhold.

- Prosjekt A finner ut at en av deres komponenter passer svært godt for gjenbruk og ønsker derfor å lage komponenten gjenbrukbar selv om dette ikke var planlagt på forhånd. I stedet for å koste X kroner så vil nå denne komponenten koste et sted mellom 1.5 * X og 2.0 * X. Dette kommer fram som en overskridelse på prosjekt A og prosjektleder får kjeft. Mengden varierer fra bedrift til bedrift.
- Prosjekt B får en gjenbrukbar komponent og trenger bare å bruker 0.1 * X i stedet for X som planlagt. Dette prosjektet kommer derfor ut med en kostnad lavere enn planlagt og får masse skryt av ledelsen – kanskje til og med et lønnspålegg.

Slike opplevelser gjør ikke at de som deltok i prosjekt A får lyst til å lage gjenbrukbare komponenter. Det er flere måter vi kan bøte på dette på. De to som kommer her blir begge brukt i bedrifter som satser seriøst på gjenbruk:

- Det å lage en gjenbrukbar komponent er et eget, separatfinansiert prosjekt for de som lager gjenbrukbare komponenter.
- Etter at hvert prosjekt er ferdig går en egen gruppe gjennom det de har laget og velger ut komponenter som skal lages gjenbrukbare. Selve jobben gjøre at Gjenbruksgruppa i bedriften.

Begge disse opplegga har den fordelen at de synliggjør gjenbruk som en investering.

Et siste problem er at det er mye morsommere å lage ting selv enn å "skru sammen" ting andre har laga. Det er to effekter som motvirker dette:

- Gjenbruke der man ikke er ekspert slik at man kan bruke tida si på det man er god til – noe som gir bedre kode. Dette krever litt planlegging av både gjenbruk og bemanning før man starter prosjektet.
- Man slipper å vedlikeholde de delene som er gjenbrukt. Det er det andre som er ansvarlig for. Dermed blir det mindre å gjenbruke og mer tid til å lage nye ting.

Det man må passe seg for er å lage et A-lag som har det morsomt med å lage nye ting og et B-lag som bare får gjenbruke og lime sammen det andre har laget. Ingen liker å tilhøre et offisielt eller uoffisielt B-lag.

04 – 04 Kapittel 18

Vi skal se på to viktige forhold:

- Feiltoleranse – kunne bli utsatt for feil uten å krasje eller forårsake andre alvorlige konsekvenser, for eksempel ødeleggelse av viktige data.
- Pålitelighet – gi riktige / forhandsdefinerte svar eller oppføre seg riktig i henhold til innholdet i en kravspesifikasjon.

I tilelegg vil vi bruke noe tid på å diskutere to viktige spørsmål innefor pålitelighet i programvare. Dere skal gjøre diskusjonen – jeg skal bare oppsummere på tavla.

Feiltoleranse

Det er enklest å ta utgangspunkt i et eksempel. Boka ser på et eksempel med en feiltolerant disk. Vi vil unngå to typer feil:

- En eller flere blokker blir umulig å lese. Hyppigheten av dette problemet er kjent fra en analyse av maskinvaren – disk pluss kontroller.
- Prosessoren som styrer dette krasjer. Det er i dette eksemplet bare krasj under skriving til disk som forårsaker uleselige blokker.

En robust løsning finnes i **fig. 18.1**. Gå gjennom koden på figuren. Vi bruker en lese / skrivesetning med følgende format:

- Parameter 1 – disken vi leser fra. Her er dette disk1 eller disk2.
- Parameter 2 – blokk adresse på disken
- Parameter 3 – resultat av lesningen, altså innholdet i blokka.

All info blir skrevet til begge diskene. Når vi leser, leser vi først fra disk1. Viss denne lesinga feiler leser vi fra disk2. Årsaken til at dette er fornuftig er som følger:

Vi antar at maskinvaren feiler uavhengig. Dette behøver ikke alltid være sant – vi kan tenke oss flere grunner til at de feiler samtidig, for eksempel at de er produsert i samme bedrift, til samme tid, er installert samtidig og er brukt like mye.

Viss feilsannsyneligheten er p vil sannsyneligheten for at begge feiler **samtidig** – gitt at de feiler uavhengig – være lik p* p. Dette vil altså være feilsannsyneligheten for ”saferead” under de gitte forutsetningene.

For å få litt bedre oversikt over systemet kan vi tegne et tilstandsdiagram – **se fig. 18.2.** Tegn også opp denne figuren slik at det blir klart ***hvordan*** vi lager den. Parametrene r, s og R er ikke sannsyneligheter – de er rater og har dimensjon 1/t. Diagrammet / modellen gir en full oversikt over tilstander og overganger i systemet. I tillegg til feilratene for de to diskene – s- har de også en reparasjonsraten r, for ”recover”. Det er altså ikke en fysisk reparasjon vi ser på. Vi ser videre at vi kan få en dobbeltfeil viss den andre disken krasjer mens vi holder på med reparasjon / ”recovery”. I dette tilfellet må vi gjøre reparasjon fra et eller annet reservemedium – for eksempel magnetbånd eller en disk som er fysisk tatt ut og lagret et annet sted. Hendelsen at begge diskene feiler samtidig er så usannsynlig at den ikke er tatt med i tilstandsdiagrammet.

Vi kan nå sette opp likevektslikningene for dette systemet. Dette gjøre ut fra prinsippet at i likevekt må trafikken inn og ut av en node være like store. Viss vi forutsetter konstant feilsannsynlighet får vi følgende modell:

$2 * s * p(1,1) = r * p(0,1) + r * p(1,0) + R * p(0,0)$

$s * p(1,1) = s * p(0,1) + r * p(0,1)$

$s * p(1,1) = s * p(1,0) + r * p(1,0)$

$s * p(0,1) + s * p(1,0) = R * p(0,0)$

I tilegg må summen av alle sannsynelighetene være 1 – ett sted må vi jo være. Derfor har vi også at:

$p(1,1) + p(1,0) + p(0,1) + p(0,0) = 1.$

Ut fra symmetri ser vi at $p(0,1) = p(1,0)$ viss diskene er fysisk like. Derfor mister vi en av de fire likningene ovenfor, men har stadig fire likninger og fire ukjente – de fire tilstandssannsynlighetene. Det nye linkningssettet blir:

$2 * s * p(1,1) = 2 * r * p(0,1) + R * p(0,0)$

$s * p(1,1) = (s + r) * p(0,1)$

$2 * s * p(0,1) = R * p(0,0)$

$p(1,1) + 2 * p(0,1) + p(0,0) = 1$

en enkel måte å løse dette systemet på er å bruke de to midterste likningene til å erstatte p(0,0) og p(1,1) i den siste likninga. Etter litt enkel manipulasjon finner vi at:

$p(0,1) = p(1,0) = (R * s) / (2 * s * s + R * r + 3 * R * s)$

$p(1,1) = (R * (s + r)) / (2 * s * s + R * r + 3 * R * s)$

$p(0,0) = (2 * s * s) / (2 * s * s + R * r + 3 * R * s)$

I de fleste tilfeller vil feilraten (antall feil pr. time) være vesentlig lavere enn raten for reparasjon, enten det er ”recovery” eller ”roll back” – altså har vi at r, R >> s. Detter gjør at vi kan forenkle likningen over vesentlig. Pass på at vi ikke lenger har at summen av sannsynlighetene er lik 1:

$p(0,1) = p(1,0) = s / (r + 3 * s)$

$p(1,1) = r / (r + 3 *s)$

$p(0,0) = 2 * s * s / (R * (r + 3 * s))$. Viss vi ønsker å bevare summen lik en kan vi i stedet sette $p(0,0) = 2 * s / (r + 3 * s)$

Strengt tatt kunne vi brukt bare r under brøkstreken, men dette ville gitt $p(1,1) = 1$, som er noe uheldig. Dessuten er vanligvis r > s, men ikke nødvendigvis mye større. La oss se på noen typiske tall for en 40 GB disk:

- Det tar ca. 20 000 sekunder, som tilsvarer 5.5 timer, å gjøre en full ”roll back”. Dette gir $R = 1 /5.5 = 0.18$..
- Reparasjon – full les / skriv av hele disken - tar ca 8000 sekunder, som tilsvarer 2.2 timer. Dette gir oss en verdi på $r = 1 / 2.2 = 0.017$.
- En disk feiler i gjennomsnitt en gang pr. en million timer. Dette gir $s = 10^{-6}$. Vi ser at allerede her er **s** svært liten i forhold til R.

Med disse tallene finner vi $p(0,1) = p(1,0) = 5.9*10^{-5}$, $p(1,1) = 0.9998$. Da har vi sannsynligheten for at vi må kjøre ”roll back” på $1 – 2*p(1,0) – p(1,1) = 0.0001$. etter som r øker i forhold til s vil p(1,1) gå mot 1.0.

Pålitelighet

Modeller for programvare pålitelighet blir brukt på to måter.

- Stille krav – dette systemet må ha en feilrate på mindre enn en feil pr. måned.
- Under testing – vi må teste til sannsyneligheten for mer enn en feil pr. måned er mindre enn 5%.

For å få til dette må vi ha en statistisk modell. Det finnes mange modeller for programvarepålitelighet – antakelig mye mer enn 100. vi skal nøye oss med å se på to. Ingen av de er spesielt avanserte, men derfor er de enkle å forstå. I tillegg viser de mange av de

problemene vi støter på når vi skal lage og bruke slike modeller. De to modellene vi skal se på er:

- Basis eksekveringstidsmodell
- Logaritmisk Poisson eksekveringstidsmodell

Før vi begynner må vi se på noen viktige momenter:

- Vi bekymrer oss ikke primært om antall feil, men om antall feilhendelser.
- Vi vil bruke eksekveringstid – ikke klokketid. Hovedårsaken til dette er at eksekveringstid bedre gjenspeiler bruken av programmet enn klokketid gjør. Personlig mener jeg at brukstid er mer relevant enn eksekveringstid, men eksekveringstid er tettere relatert til eksekveringstid enn til klokketid så det er vel egentlig OK.
- Vi vil estimere pålitelighet ut fra tidligere erfaringer – for eksempel testing. Den parametriserte modellen vi kommer fram til forutsetter strengt tatt at framtida likner på fortida. Derfor – viss framtidig bruk skiller seg mye fra måten vi har testa på er den modellen vi lager helt uinteressant.
- Programvare er deterministisk. Det som var OK i går vil være OK i morgen. Programvare kan derfor bare feile under følgende forutsetninger:
    - Noe har endra systemet eller dets omgivelser
    - Systemet får ny input – input det ikke har opplevd før
    - Systemet får input det har opplevd før, men er i en ny tilstand.

  Estimering av et programs pålitelighet kan derfor ikke skilles fra estimering av brukernes oppførsel.
- Vi kan beskrive systemets oppførsel på to måter:
    - Antall feil funnet i en bestem periode
    - Tid mellom feil i en bestemt periode.

  Valget vil bestemme hva slags modeller som er realistiske.
- Under testing vil feil bli fjernet etter hvert som de blir funnet. Vi vil derfor anta at programmet blir mer og mer pålitelig etter hvert. En tilsvarende prosess vil ikke uten videre være på plass når systemet er i drift.

Litt notasjon:

- Gjennomsnittlig antall feil i intervallet [0, t] – $\mu(t)$
- Feilintensitet – $\lambda(t)$. Dette er gjennomsnittlig antall feil pr. tidsenhet. Derfor har vi også at $\lambda(t) = d[\mu(t)] /dt$

Forholdet mellom disse parametrene er vist i **fig. 18.3**.

Basis modell - BM

Her antar vi at reduksjon i feilintensitet - gitt som funksjon av antall feil fjernet – er konstant. Lar vi μ være det gjennomsnittelige antall observerte feil, kan vi sette $\lambda(\mu) = \lambda0(1 - \mu / N0)$, der N0 er det ukjente totale antall feil i systemet. Av dette følger at $d\mu/dt = \lambda0(1 - \mu / N0)$. Denne differensiallikninga kan vi løse enkelt ved hjelp ved å bruke differensialoperatoren og tilpasse et konstantledd. Vi finner direkte at $\mu(t) = N0[1 - \exp(-t * \lambda0 / N0)]$ og derfra direkte at $\lambda(t) = \lambda0*\exp(-t * \lambda0 / N0)$.

Denne modellen skriver seg fra Jelinski og Moranda og er en av de eldste og enkleste modellen for programvarepålitelighet. Ofte skrives den på formen:

$\mu(t) = N0[1 - \exp(-t * \Phi)]$, $\lambda(t) = \Phi N0*\exp(-t * \Phi)$.

Logaritmisk Poisson modell – LPM

Her antar vi at feilintensiteten avtar eksponentielt, altså at vi har $\lambda(\mu) = \lambda 0 * \exp(-\theta*\mu)$. Vi kan omforme dette til følgende differensiallikning: $d\mu/dt = \lambda 0 * \exp(-\theta*\mu)$ som er direkte integrerbar med grensebetingelsen $\mu = 0$ for $t = 0$. Vi får da $\exp(\theta\mu) = \theta\lambda 0*t + 1$ som igjen gir oss at $\mu = \ln(\theta\lambda 0*t + 1) / \theta$. Herfra finner vi direkte at $\lambda = \lambda 0 / (\theta\lambda 0*t + 1)$.

Bruk av modellene

Når vi slutter å rette eter hvert som feilene dukker opp – etter levering til kunde – vil feilraten etter disse to modellene være konstant. Vi erstatter da t med T – den tiden vi har brukt til testing. Vi får da en homogen Poisson prosess med feilrate $\lambda(T)$ og sannsynligheten for n feil i løpet av tidsrommet t blir

$P_n(t) = (\lambda t)^n * \exp(-\lambda t) / n!$

Vi ser direkte at $P_0(t) = \exp(-\lambda t)$.

Vi kan estimere modellparametrene ut fra vanlige statistiske metoder – for eksempel MLE (Maximum Likelihood Estimation) eller LSE (Least Square Error). Alternativt kan vi bruke en grafisk estimering – kurvetilpassning. **Se fig. 18.5 i boka**. Vi skal ikke se mer på dette her. Det er viktigere å se på hvordan modellen kan brukes i praktisk arbeid. La oss anta at vi for øyeblikket har en feilrate på λ_P mens kravet er ei feilrate mindre eller lik λ_F . Vi er interessert i å vite hvor mye lenger vi må teste for å oppfylle kravet til feilrate. Vi går fram på følgende måte:

- Basismodellen: $\lambda_P = \Phi N0*\exp(-t_P * \Phi)$, $\lambda_F = \Phi N0*\exp(-t_F * \Phi)$. Ved å dividere den ene likningen på den andre finner vi direkte at $\Delta t = \ln(\lambda_P / \lambda_F) / \Phi$.
- For den logaritmiske modellen løser vi dette lettest ved å skrive om likningen for λ til uttrykket $(\lambda 0 / \lambda_p - 1) = \theta\lambda 0*t_P$ og tilsvarende for λ_F. Ved å finne utrykk for t_P og t_F og subtrahere finner vi $\Delta t = (1/\lambda_F - 1/\lambda_p) /\theta$.

Alternativt kunne v ta utgangspunkt i $\mu(t)$ og finne ut hvor mange flere feil vi må finne for å tilfredsstille kundens krav til pålitelighet. Det kan være greit å huske på at disse – og andre modeller innfor programvarepålitelighet – bare er tilnærminger. De viktigste problemene med å modellere programvarepålitelighet er at:

- Hvordan og hvor ofte et system feiler avhenger av hvordan det blir brukt. Modellering av feilforekomster medfører derfor at vi må modellere bruksmåten. Siden programvare er deterministisk kan systemet – under testing - bare feile for *ny input eller for gammel input i en ny tilstand.*
- En modell som gir gode resultater for en måte å bruke systemet på, vil kunne feile for andre bruksmåter.

Vi bør derfor ikke bygge beslutningene bare på resultater fra de formlene som kommer ut av pålitelighetsmodellene. Boka foreslår å ha en bibliotek av pålitelighetsmodeller og deretter velge den modellen som passer best til hvert enkelt prosjekt. Dette vil sikkert fungere i mange tilfeller, men det blir litt for ugjennomsiktig til at vi skal ha tillit til resultatene. Vi savner jo svar på ”Hvorfor passer denne modellen og ikke de andre?”

Til diskusjon

En løsning på problemet med å lage høypålitelig programvare var lenge – og er stadig i neon miljøer – å innføre N-versjon programmering. Ideen er som følger:

- Skriv en kravspesifikasjon og distribuer den til N utviklingsmiljøer – for eksempel tre stykker.
- Hvert miljø lager sin versjon av systemet. Det er viktig at disse tre miljøene ikke utvekseler informasjon, diskuterer problemer og løsninger eller samordner seg på noen måte. Hensikten er å hindre felles feil i systemet – feil som finnes i to eller flere av systemene.

- Bruk alle systemene sammen. De får samme input, men leverer hver sin output. Resultatene blir sammenliknet. Vi kan nå velge mellom flere strategier:
    - OK bare viss alle systemene gir samme resultat. Ellers rapporterer vi en feil
    - OK viss flertallet – for eksempel 2 av 3 – gir samme resultat. Feil bare viss det ikke er mulig å finne et resultat som har flertall

Dette ble imidlertid ikke den suksessen mange hadde håpet på. Diskuter hva som kan være mulige årsaker til dette. .

04-08 Kapittel 4

Vi har behov for konfigurasjonsstyring / konfigurasjonskontroll av to grunner:

- I løpet av et prosjekt blir det produsert en god del dokumenter – design, kode, tester osv. I løpet av prosjektet må de fleste av disse endres en eller flere ganger pga. feil, inkonsistens, noen – for eksempel kunden - har forandret mening, ting passer ikke sammen osv. Lista med årsaker kan gjøres lang.
- Etter at vi har levert første versjon for første plattform må vi endre systemet selv om det ikke er feil i det. Som før er årsakene mange. Eksempler er: selge produktet på ny plattform, lage spesialversjon for en viktig kunde, komme med nye, forbedra versjoner – ny funksjonalitet, nye muligheter.

Utfordringene i de to tilfellene er litt forskjellig:

- I løpet av prosjektet – passe på at alle bruker nyeste godkjente versjon av alle dokumenter og at det samtidig er mulig å gå tilbake til en tidligere versjon viss det viste seg at vi gjorde noe dumt.
- Etter at vi har levert – sørge for at vi kan gjenskape de versjonene som hver enkelt kunde har. Dette kna bli komplisert når vi begynner å få inn feilrapporter som bare angår systemene til noen kunder.

Grunnleggende begreper

Vi vil starte med å anta at det finnes bare en offisiell versjon av systemet. Dette kalles en ”baseline”. Alt som tilhører en baseline er formelt godkjent ved hjelp av tester, granskning, inspeksjoner og lignende. Den første oppgaven til konfigurasjonskontroll er å sørge for at ikke ”uoffisielle” dokumenter kommer å forurenser baseline. Dette kan for eksempel være dokumenter oms ennå ikke er granska eller moduler som ikke har en godkjent test. Baseline inneholder alt som beskriver / dokumenterer systemet. Dette inkluderer:

- Kildekode og objektkode (kode som er kompilert og klar for lenking)
- Kravspesifikasjon og designdokumenter – både overordna og detaljert design.
- Testplan og testresultater - testlogg
- Brukermanual

Det er lett å se for seg at de fleste av disse dokumentene vil bli endra en eller flere ganger i løpet av et prosjekt og at det kna være en utfordring å holde de oppdatert og konsistente. For å få til dette vil de fleste bedrifter velge å ha en dokumentert prosess. Det er interessant å se at mange bedrifter som ellers ikke har noen utviklingsprosess i det hele tatt – og kanskje til og med synes det er dum ide – likevel har en meget streng kontroll over hva som ligger i baseline. En slik prosess er vist i boka – **fig. 4.1**. Vis og kommenter. Legg merke til følgende:

- Når vi skal vurdere effekten av å tillate en endring må vi ta hensyn til både produktet, det dokumentet det gjelder og selve utviklingsprosessen / prosjektet.
- Vi må velge om vi vil:
    - Forkaste – dette vil vi ikke gjøre noe med
    - Sette på venting – dette vil vi ikke gjøre noe med nå
    - Akseptere endringa. I dette tilfellet må vi også gi endringa en prioritet.
- Pass på at baseline er konsistent etter endringa. Dette kan føre til at vi – i tillegg til endringa – også må initialisere andre endringer. Viss vi for eksempel endrer funksjonalitet må vi i tillegg endre brukerdokumentasjon og en del systemtester.

Det er den siste jobben som er vanskelig. Intensiv bruk av sporing hjelper noen, men generelt sett er det så mange mulige sammenhenger i et system at det er vanskelig å holde styr på alle.

Siden det kan være flere som ønsker å endre en komponent mer eller mindre samtidig er det viktig å ha en mekanisme som sørger for at bare en person ad gangen kan andre noe. En vanlig teknikk er kjent som "check-in / check-out" der alle baseline dokumentene ligger i en database. **Forklar dette konseptet kort**. Det er vanligvis CCB – "Change Control Board" – som kontrollerer når en komponenten er ferdig og dermed kna legges inn i databasen. Komponenten er da under konfigurasjonskontroll / versjonskontroll. **Fig. 4.2 i boka** viser hvordan dette blir ivaretatt.

For å holde styr på utviklingen av hver enkelt komponent er det vanlig å innføre en faset nummerering. En mulig å måte å gjøre dette på er å bruke nummer som er bygd opp som X.Y.Z osv. Dette har følgende tolking:

- Første siffer identifiserer hovedversjonen. Det er vanlig å bruke 0 for den opprinnelige utviklingen, 1 for første release og deretter 2, 3, osv. for nye hovedreleaser. Hovedreleaser inneholder en konsolidering av de endringene som er gjort til nå pluss eventuell ny funksjonalitet.
- Andre siffer viser midlertidige endringer – små feilrettinger osv. Det er vanlig å operere med X.1, X.2 osv så lenge man retter og deretter samle alle endringene i en ny release som da får nummer (X+1).0.

Vis og kommenter fig. 4.3. Legg merke til at vi legger til ny funksjonalitet i den ene greina – X.2 -> X.3 – mens vi bare gjør små endringer i den andre – X2.1 -> X2.2 -> X2.3. Når vi har laget ett system og ønsker å lage nye versjoner – for eksempel for en ny plattform – får vi flere parallelle utviklingsspor – greiner. Erfaring har vist at når vi først har laget flere greiner i utviklinga av et produkt er det vanskelig å få slått de sammen igjen.

Ofte lager man ikke fysisk komponentene X2.1, X2.2 osv. I stedet lagrer man basisversjonen – i dette tilfelle X.2 – og deretter bare endringene, også kalt deltaer. Versjon X.3 består derfor av versjon X.2 pluss alle deltaer som tilhører X.2. Først når vi går til en ny hovedversjon blir alle deltaene fysisk inkludert i modulene. Fordelen med dette er at vi kan lage hele opplegget mer fleksibelt og brukervennlig:

- Deltaene kan gies navn etter hvilken effekt de har – "rett problem med store tabeller, feil R.12.
- Vi kan enkelte ta med bare noen av rettingene i stedet for å måtte ta med alt som er gjort til nå.

Slike systemer er best kjent under akronymet SCCS – "Source Code Control Systems". Problemet med dette – og alle liknende opplegg – er at vi får problemer så snart vi begynner å endre i den koden som allerede er endra før vi lager en ny release.

Det finnes verktøy som automatisk lager oppdaterte versjoner av koden, kompilerer de nye versjonene og lenker det hele sammen til et nytt system. Input til slike verktøy er en konfigurasjonsbeskrivelse som består av:

- En liste over baseline komponentene som skal være med
- En liste av deltaer for hver baseline komponent

Konfigurasjonsplan

Som mye annet i et større programvareprosjekt er det også kjekt å ha en plan for konfigurasjonsstyringen. **Fig. 4.4** viser innholdet i en slik plan i henhold til IEEE Std. 828. Opplegget kan se overveldende og vel byråkratisk ut, men erfaring viser at man trenger dette for å komme i mål med prosjektet på en ryddig måte. De viktigste delene er:

- Ledelse – det viktige her er hvordan ansvaret er fordelt innad i prosjektet. Hvordan blir en fase avslutta, hvem bestemmer når en enhet er ferdig osv. I tillegg blir samarbeidet mellom CM, QA og utvikling fastlagt her.
- Aktiviteter – her sider vi noe om hvordan vi vil holde styr på enhetene. I tillegg må vi si noe om selve godkjenningsprosessen og hvordan vi holder oversikt over status – hva skal endres, av hvem og hvorfor. Hva har vi av utestående endringsforslag, deres prioritet osv.

Diskusjonstema

CM ble laget og definert som oppgave og problem den gangen man hadde en tradisjonell fossefallsmodell. Må våre ideer, verktøy og rutiner endres når vi går over til inkrementell utvikling eller er de metodene og standardene vi har stadig tilstrekkelige.

Diskuter. Diskusjonen oppsummeres på tavla.

Kapittel 5

Dette handler primært om menneskene i prosjektet. På mange måter er det den viktigste delen, men også den delen som er vanskeligst tilgjengelig. Dette er ikke ment som en opplæring i det å lede mennesker i et prosjekt. Det er mer en liste over momenter dere må passe på når / viss dere skal lede et prosjekt. Selve opplæringa i å takle de situasjonene som oppstår er ikke en del av systemutvikling som fag, det hører til psykologi og sosiologi.

Dette er et sett av regler – ting det kan være lurt å passe på:

- Et prosjekt bestå av individer – hver med sine egne mål. Det er prosjektleders oppgave å sørge for at alle jobber sammen mot det felles mål som er **prosjektets mål.** En vesentlig del av dette er at prosjektleder gjør det klart for prosjektdeltakerne hva han forventer av dem – **hva er hver enkelts oppgaver i prosjektet**. Erfaring viser av viss ikke dette er klart fra start, vil deltakerne snart definere sine egne mål og jobbe for å oppnå disse – uansett hva de måtte være og uansett om det angår prosjektets mål eller ikke.
- Når man først er blitt enige om målene og hvem som skal gjøre hva er det viktig å følge opp at hver enkelt gjør jobben sin innefor de tids og kostnadsrammene som er definert. Dersom det viser seg at det ikke er mulig å gjøre jobben innefor disse rammene må prosjektleder sørge for å:
    - Gi utvikleren nye rammer. Selv om det er prosjektleder som gir rammene så må utvikleren være med å bestemme hvor mye ekstra ressurser som trengs for å fullføre jobben.
    - Replanlegge prosjektet. Dette består både av å replanlegge de aktivitetene som har fått nye ressurser og de aktivitetene som ikke er påbegynt ennå.
- At de som jobber i prosjektet gjør jobben sin – at det er framdrift i prosjektet. Den mest effektive måten å følge opp et prosjekt på er så se på ressurser brukt og verdi skapt. Dette måles ved å se på:
    - Antall timeverk brukt i en aktivitet – ressurser brukt
    - Antall arbeidspakker ferdige – verdi skapt. Verdien til en arbeidspakke er lik det vi har estimert at den skal kost. Det er dette kunden betaler og det er det enste verdimålet det er mulig å bli enige om.

  Man kunne i stedet se på antall kodelinjer produsert. Forfatteren av læreboka er engstelig for at dette vil føre til at folk skriver kode bare for å skrive kode. Dette vil gjøre målet ubrukelig, men de enste utviklerne egentlig lurer er seg selv. Viss man bruker dette målet kna det være lurt å telle linjer **skrevet, endra eller sletta** for å få et best mulig bilde av utviklinga.
- Det å jobbe i et prosjekt betyr samarbeid. Et prosjekt er ikke en egotripp. Ideelt sett er det ikke noe som heter ***min kode*** i et prosjekt – i det minste ikke før den skal vedlikeholdes. Samarbeide i et prosjekt kan ta mange former – fra det helt uformelle ”buddy” systemet til formaliserte kodegranskninger.
- At kommunikasjonen fungerer i prosjektet. Det er flere måter å få til dette på – noen formelle og noen uformelle. Det etterfølgende er neon eksempler
    - Prosjektmøte. Disse bør være på en fast tid, for eksempel hver mandag kl 10:00. bruk de bare til å gjøre ting som alle må være med på ellers kaster vi bort verdifull tid. Dette betyr at prosjektet må se på framdrift – hvor er vi - eventuelle problemer – hvordan kommer vi videre – og viktig innfor som alle trenger.
    - Uformelle kontakter, for eksempel rund kaffemaskinen eller Colaautomaten. Felles kaffe-, lunsj- og tidsskriftareal er også nyttig. Generelt er det viktig å skape mange muligheter / anledninger til uformell kontakt. At dette er viktig er en av årsakene til at geografisk distribuerte prosjekter ofte får problemer.

Hvordan vi koordinerer arbeidet vårt vil avhenge av flere faktorer. De viktigste er:

- Type arbeidsoppgave – for eksempel komplett definert vs. løst definert.
- Hva slags folk har vi – for eksempel nyansatte uten erfaring vs. kompetente ansatte med lang og erfaring og spesialister vs. folk med svært diversiv bakgrunn.
- Organisasjon – sentralstyrt firma vs. firma der hver enkelt del har en stor grad av selvstyre. Dette bestemmer f.eks. hvilke muligheter vi har til selv å velge hvordan vi skal jobbe.

Noen erketyper av koordineringsmekanismer:

- Enkel struktur. Koordinering skjer gjennom direkte kontroll. Det finnes noen få ledere og mange utviklere. Det finnes lite spesialisering, opplæring eller formelle rutiner. Dette fungerer bare i små organisasjoner med få og små prosjekter.
- Maskinbyråkrati. Arbeidet er spesifisert helt og fullt og arbeidet foregår etter spesifiserte instrukser. Spesialiserting er viktig mens opplæring blir holdt på et minimum.
- Avdelinger / divisjoner. Avdelingene er her prosjekter som har en stor av selvstyre. Målene blir satt utenfra, men prosjektene har stor grad av frihet til å bestemme hvordan de vil nå målene. Koordinering mellom prosjekter skjer gjennom standardisering av resultater – hva skal prosjektet levere og på hvilken form. Koordinering innefor prosjektet skjer gjennom prosjektmøter.

- Profesjonelt byråkrati. Når det ikke er mulig å spesifisere verken sluttresultat eller innhold kan man oppnå koordinering gjennom standardisering av hva den enkelte utvikler skap kunne. Fokus er altså på dyktige medarbeider som vet hva som trengs å gjøre.
- Ad-hoc-krati. I prosjekter med en stor grad av innovasjon er det vanskelig å definere konkrete, avgrensa arbeidsoppgaver. Vi trenger mennesker som er i stand til å nå vage mål på en koordinert måte. Koordineringen skjer gjennom *gjensidige* justeringer.

Det er ikke slik at hele bedriften behøver å ha de samme koordineringsmekanismene. Særlig i store bedrifter vil man finne mange måter å koordinere på, avhengig av hva avdelingen lager, hva slags omgivelser de fungere i osv.

I tillegg vil det ikke nødvendigvis være lurt å ha samme måten å lede et prosjekt på gjennom hele levetiden til prosjektet. For eksempel vil analyse og designaktiviteter ofte være innovative, mens selve kodingen må være strengt kontrollert. Det er bra med kreativitet i problemløsning, men kanskje ikke fullt så bra når vi skal lage kode i henhold til en kodingsstandard.

Ledelsesstil

Det er to akser vi kan bruke for å beskrive / vurdere en ledelsesstil:
- Relasjonsoppmerksomhet – graden av fokus på hver enkelt persons relasjoner til andre personer i prosjektet.
- Oppgaveoppmerksomhet – graden av fokus på hva vi skal oppnå og hvordan vi skal oppnå det.

Dette kna illustreres med følgende diagram. **Vis fig 5.1 i boka**. De fire ledelsesstilene er som følger:
- Separasjon – passer best for rutinearbeid. Byråkratisk, hierarkisk opplegg med rutiner og prosedyrer. Vi får en stabil og effektiv prosjektorganisasjon, men det er vanskelig å få til noen virkelig innovasjon.
- Relasjoner – folk trenger å bli motivert, koordinert og tenger opplæring. Hver enkelt oppgave vil være gitt til en person. Jobbene er komplekse, spesialiserte og innovative. Beslutninger skjer gjennom møter og konsensus. Den viktigste suksessfaktoren er en prosjektleder som får dette til å funger i stedet for at det skal degenerer til en evigvarende serie av møter og diskusjoner.
- Engasjement – mest effektivt når vi må jobbe under press – for eksempel tidspress. Denne ledelsesformen avhenger i stor grad av en dyktig leder. Beslutninger blir ikke gjort i møter, men er i stor grad implisitt ved at deltakerne deler en felles visjon om ha de skal lage.
- Integrasjon. Fungere best når målet er uklart. Arbeidet vi i stor grad være eksplorativt – vi prøver oss fram, for eksempel ved hjelp av prototyper og andre former for eksperimenter. Prosjektleders oppgave er å stimulere og motivere. Beslutningene blir tatt ”bottom-up” på en uformell måte. Ledelsen må passe på at vi ikke mister målet av syne i all denne kreativiteten og eksperimenteringen.

Prosjektorganisering

Vi trenger en prosjektorganisasjon for å få folk til å samarbeide mest mulig effektivt for å nå et felles mål. Det er mange måter å gjøre dette på og vi skal bare kort vise noen av de prosjektorganisasjonene som finnes i programvareindustrien.

Hierarkisk organisasjon

Hierarkiske prosjekter er ofte modellert etter prosjektstrukturen og produktstrukturen. Det finnes ett subprosjekt for hvert delsystem, ett for integrasjon, ett for testing osv. **Vis fig 5.2**. De øverste lagene i prosjektstrukturen koordinerer arbeidet ved hjelp av standardisering og ved hjelp av regler og prosedyrer. Lenger ned i strukturen vil det kunne være andre måter å koordinere arbeidet på – avhengig av hva slags arbeid det er. Noen av subprosjektene kan være svært standardiserte, mens andre kna være meget innovative. Dette er både styrken og svakheten i denne måten å jobbe på:
- Styrke – delprosjektene kan organiseres på den måten som er mest effektiv for den jobben de skal gjøre akkurat nå.
- Svakhet – flere delprosjekter med hver sin organisasjonsform vil kunne lede til kraftige kulturkollisjoner som kan bli et problem i prosjektet.

I tillegg har vi det problemet som alle hierarkiske strukturer har: ordre går ovenfra og ned gjennom forsterkere – alle legger på litt ekstra ”trøkk” - og nedenfra og opp gjennom filter – ingen har noensinne blitt belønna for å komme med dårlige nyheter.

Matriseorganisasjon

Dette blir brukt i organisasjoner som egentlig er organisert i henhold til kompetanse. Det finnes grupper for databaser, grafikk, nettverk osv. hvert prosjekt settes sammen av folk fra de enkelte faggruppene og folk går tilbake dit når prosjektet er ferdig.

Sammenhengen i prosjektet må ivaretaes av prosjektleder. Viss vi igjen ser på **fig 5.1** vil denne måten å organisere prosjektet på kreve en prosjektleder med høy oppgaveoppmerksomhet mens den som leder hver faggruppe må ha høy relasjonsoppmerksomhet.

”Chief programmer” organisasjon

En slik organisasjon består i utgangspunktet av tre personer – ”chief programmer”, en assistent og en bibliotekar. Disse har hver sine oppgaver:

- Chief programmer. Han gjør all design og implementerer de viktigste / vanskeligste delene av systemet.
- Assistent. Han er vikar for chief programmer og vil ofte ta seg av resten av implementeringa.
- Bibliotekar. Tar seg av alt administrativt arbeid og dokumentasjon .
- Ekspertgruppe. Dette er bare en del av CPT viss jobben er for stor for chief programmer eller viss man trenger ekspertise som chief programmer ikke har.

Det finns en del gode resultater fra å ha brukt CPT. Det første problemet som melder seg er ”Finnes det nok folk med så høy kompetanse at dette er generelt gjennomførbart?” Det ser ut til at svaret er ”Nei”. Den store fordelen med CPT ligger i det at det er lettere å få et lite antall mennesker til å jobbe effektivt sammen. På den andre side – det er begrenset hva en liten gruppe mennesker kan får gjort på rimelig tid.

En variant av CPT har innført ei lita gruppe av likeverdige personer i stedet for chief programmer og hans assistent. I tillegg har man innført en ny person – tester. Det er også mulig å ta med et par nybegynnere i opplæringsstillinger. Noen ganger kan man la en av disse være bibliotekar.

SWAT

SWAT er en forkortelse for ”Skilled With Advanced Tools” og beskriver en organisasjon som stort sett bare passer i RAD – ”Rapid Application Development”. Et prosjekt organisert etter SWAT består av maks fire personer - generalister, som helst skal jobbe i samme rom. Dette skaper effektiv kommunikasjon i prosjektet. SWAT prosjekter opererer med et minimum av dokumentasjon – ofte bare noen slisser på en whiteboard.

Hovedideen i SWAT er å drive inkrementell utvikling basert på utstrakt gjenbruk, svært høynivå programmeringsspråk og utstrakt bruk av programgeneratorer – for eksempel 4GL.lederen i et SWAT prosjekt fungerer omtrent som en forman på et bygg.

For at et SWAT prosjekt skal lykkes, må deltakeren være svært motivert.

Åpen struktur

Prosjekter som er organisert etter denne modellen prøver å kombinere to viktige faktorer:

- En åpen ledelsesstil med fokus på samarbeid. Prosjektet må ha en leder med teknisk kompetanse som kan ta avgjørelser når deltakeren ikke klarer å arbeide seg fram til et konsensus. Det er viktig å ha engasjenent i prosjektet slik at alle jobber mot et felles mål uten at man skal måtte ty til detaljoppfølging.
- En ryddig / streng prosjektstruktur som sørger for at prosjektet oppfører seg på en forutsibar måte, for eksempel i forholdet til planer og kostnader.

Noen gode råd

- Prioriter kvalitet framfor kvantitet. Små prosjektgrupper fungerer best, blant annet pga. at kommunikasjon er en kritisk suksessfaktor.
- Ta hensyn til den kompetansen, motivasjonen og interessen som finnes i prosjekt når oppgaven blir delt opp i mindre oppgaver og distribuert til deltakerne.

- Pass på at alle i bedriften får anledning til å:
    - Lære ny ting
    - Delta på områder de har lyst til – ikke bare der de er best for øyeblikket.

  Dette er særlig viktig å ta med i betraktning når vi ser det sammen med foregående punkt.
- Pass på å ha ei prosjektgruppe som er velbalansert – man trenger for eksempel både eksperter på enkeltområder ***og*** noen som kan sørge for at ekspertenes bidrag blir sydd sammen til et brukbart hele.
- Folk som ikke passer inn i prosjektgruppa bør flyttes til et annet sted jo før jo heller dersom dette er mulig. Det er dessverre mange eksempler på prosjekter om har blitt ødelagt av interne konflikter. Jo mer et prosjekt er avhengig av samarbeid, entusiasme og motivasjon, jo farligere er interne konflikter.

04-15 Kapittel 19

Bruk av verktøy i utvikling av programvare blir ofte referert til som CASE – "Computer Aided Software Engineering". På et tidspunkt var det mange som trodde at innføringen av CASE verktøy skulle løse alle eller de fleste problemer. Dette har vist seg å være alt for optimistisk. Denne forlesningen har ikke til hensikt å gi noen innføring i bruk av CASE verktøy eller gi en komplett oversikt. I stedet er det noen smakebiter på hva som finnes og hva det kan – og ikke kan – hjelp til med. Slik det ser ut nå vil nok programvareutvikling være mye mer menneskeintensivt enn verktøyintensivt i all overskuelig framtid.

Det finnes flere typer CASE verktøy. **Vis fig. 19.1**. Legg merke til at vi deler CASE inn i to hoveddeler – verktøy og utviklingsomgivelser. Her kommer noen få kommentarer om hvert verktøy:

- Verktøy – støtter en avgrensa aktivitet i utviklingsprosessen, for eksempel testing eller overordna design
- Verktøybenk – en samling av (to eller flere) integrerte verktøy som understøtter et avgrensa sett av aktiviteter, for eksempel analyse og design.
- Utviklingsomgivelser skal – i det minste i prinsippet – understøtte hele utviklingsprosessen, fra kravanalyse til systemtest og vedlikehold. Utviklingsomgivelser kan videre deles opp i:
    - Verktøysett - en samling av verktøy, vanligvis ikke fullstendig integrert.
    - Språkomgivelser – et sett av verktøy tilpasset utvikling i et gitt programmeringsspråk, for eksempel Java. Slike verktøysett kan dra nytte av egenskaper ved språket og dermed gjøre en god del sjekking automatisk.
    - Integrerte utviklingsomgivelser – integrasjon av informasjon. Alle verktøyene jobber mot en felles database. All informasjon om systemet og dets komponenter ligger her.
    - Prosessorienterte utviklingsomgivelser – er omtrent det samme som integrerte systemomgivelser, men er tilpasse den utviklingsprosessen vi bruker. Dette kan for eksempel innebære at det ikke er lov å levere en komponent til CM-delen av databasen uten at det finns info om at det kan gjennomgått enhetstesting og kodegransking.

Ofte finner det sted en kontinuerlig utvikling av CASE verktøyene. Det kna begynne med ett eller flere frittstående verktøy som så samles til en verktøybenk. Etter som man lager flere og flere verktøy vil man sette de sammen til en utviklingsomgivelse som man så senere integrerer. Når man har fått og erfaring med dette vil man kanskje til slutt legge på prosessinfo for å få en fullstendig utviklingsomgivelse.

For å kunne karakterisere CASE verktøy er det forslått en flerfasettert modell, som vist i boka **fig. 19.2**. Dette er primært en måte å beskrive eller diskutere CASE verktøy på. Det etterfølgende er neon korte kommentarer til hver:

- Supportbredde – hvilken av de fire kategoriene over hører verktøyet til
- Problemtype – sier noe om hva slags systemer / problemer verktøyet passer for
- Størrelse på systemet – her delt opp i tre kategorier etter hva slags systemutvikling det passer til: store systemer, middels store systemer eller bare til utvikling av små systemer.
- Brukerskala – sier noe om forholdet mellom CASE verktøyet og antall personer i prosjektet.
    - En enkelt bruker. Dette er stort sett verktøy som editorer, debuggere og kompilatorer
    - Grupper av brukere. Her må man i det minste – i tillegg til det vi har over – inkludere CM verktøy og verktøy som bygger systemer.
    - Store grupper av brukere. Nå blir det nødvendig å legge på en god del regler for hvordan samarbeid skal skje, ansvar for aktiviteter og lignende.
    - Organisasjoner. Her trenger vi i tilegg dokumentmaler, standarder, regler og støtte for gjenbruk på alle plan osv .
- Prosesskala. Understøtter verktøyet produkt, folk eller begge deler. Eksempler på disse to aspektene er:

    - Produkt – skrive kode, teste, konfigurere
    - Folk – planlegging av granskingsmøter, holde styr på tidsfirster, ressursforbruk osv.
- Prosesstøtte. Verktøy som støter prosessen hjelper – eller tvinger – utvikleren til å følge en definert prosess, bruke de riktige dokumentene osv. neon verktøy støtter ingen prosesser, noen støtter en fast definert – hardkoda – prosess, mens det også finnes verktøy der man selv kan definere prosessen. Eksempel er ”Process Weaver”.
- Eksekveringsparadigme. Dette går også under betegnelsen ”enactment” og forteller i hvilken grad verktøyet, ut fra et sett av regler, gjør ting på egen hånd. Dette kna gjøre på flere måter:
    - Tilstandsmaskin – når verktøyet er i en gitt tilstand utfører det et sett av definerte aksjoner. Eksempel: Når en modul er i tilstand ”ferdig gransket” blir det lagt inn i CM databasen
    - Petrinett – når all nødvendig input til en aksjon foreligger, blir aksjonen utført. Eksempel: når alle modulene som tilhører et subsystem er i tilstand ”Ferdig” blir de lenket sammen til et subsystem
    - Produksjonsregler – bruker ideer fra både tilstandsmaskiner og Petrinett. Er vanligvis på formen ”Vi kan bygge X når a, b og c foreligger”.

Vi vil se på noen eksempler på verktøy.

Verktøysett - Unix

Unix er det fremste eksempel på et verktøysett som blir mye brukt. Det har en del egenskaper som gjør at det er enkelt og fleksibelt, på tross av et brukergrensesnitt som ser nærmest uforståelig ut. De viktige egenskapene er:

- Filsystemet har en trestruktur. Alle filer har et eget, definert stinavn. Filkataloger er også filer.
- Filstrukturen er enkel. Alle Unix filer er sekvenser av bytes og det finnes ingen filtyper.
- All systemprogrammer og de fleste brukerprogrammer tar input fra terminalen og gir output til den samme terminalen. Dette kna imidlertid enkelt endres ved å definere ny inputkanal ”< input navn” og ”> output navn”.
- Unix har en stor mengde små, enkle og praktiske programmer for de fleste ting en utvikler har behov for. Populære eksempler er ”wc” – teller ord, tegn og linjer – ”lpr” – skriver ut ei fil, ”grep” – finner tegnmønster, ”ls” lister opp alle filer i stående katalog. .
- Lett å skjøte sammen programmer ved hjelp av en ”pipe” – symbol ”|”. Enkelt eksempel. ”ls | lpr” lager en liste av alle filer i stående katalog og printer den ut.

Unix er populær hos en gruppe utviklere som nærmest har laget seg sin egen stamme – med språk, ritualer osv. noen ganger kna man få inntrykk av at det at Unix har en kryptisk brukergrensesnitt er et gode i seg selv – det holder fremmede ute.

På den annen side er det enkelt å lage et brukervennlig språk på toppen av Unix og bruke alle de gode ideene som finnes der.

Språkavhengig omgivelser

Omgivelser som er språkavhengige bruker egenskaper ved det valgte språket til å hjelpe og støtte utviklerne. Viktig hjelp er for eksempel:

- Mer effektiv editering – når man taster inn et reservert ord så kommer resten av strukturen på plass og er bare å fylle inn. Dette sikrer syntaktisk korrekt kode.
- Hjelp til debugging – når programmet eksekveres kan systemet vise fram variabelnavn med verdier, hvilken gren av en IF..THEN… som blir valgt osv. det vil også være mulig å definee at vi skal ha tilbake kontrollen på et gitt sted i koden – ”breakpoint”. Her kan vi se på variable, endre verdier og så fortsette eksekveringen.
- Browser – dette er et hjelpemiddel for å se på programverdier. Dette gjelder objekter, attributter, meldingskøer osv. Referanse skjer via de navna vi har gitt når vi skrev koden.
- Prettyprinter – et godt hjelpemiddel når vi skal lese koden, for eksempel i en kodegransking. Dette programmet gir innrykk i kode på definerte steder, for eksempel etter hver IF-setning, etter hver ”{” osv og tilsvarende når vi kommer til for eksempel ”}”.
- Eksekvering av delvis komplette programmer. Dette gjør at du kam skrive et program delvis ferdig. Når eksekveringen når deen uferdige biten for utvikleren kontrollen tilbake og kan for eksempel
    - Skrive inn manglende kode, kompilere og fortsette eksekveringen
    - Sette inn nødvendige dataverdier og forsette på et valgt sted i programmet

Slike løsninger krever vanligvis rask responstid og et godt grafisk brukergrensesnitt for å være effektive.

Verktøybenk

Det finnes flere slags verktøybenker. Det etterfølgende er bare eksempler. Se også **fig 19.3**.

- AWB – Analyst Work Bench, for de som gjør systemanalyse
- PWB – Programmer Work Bench, for utviklere, de som implementerer
- MWB – Manger Work Bench, for prosjektledelsen
- IPSE – Integrated Project Support Environment.

Det etterfølgende er noen få kommentarer til hver av disse verktøybenkene:

- AWB: kravanalyse og overordna design. Dette skjer ofte ved hjelp av diagrammer og en gode grafiske verktøy er viktige. Resultatene lagres i en database. I tilegg er det viktig å kunne:
    - Oppdater og endre diagrammer
    - Analysere krav og designbeslutninger med hensyn til kompletthet og konsistens.
    - Generere rapporter
    - Lage enkel prototyper for å kunne utforske designalternativer
- PWB: implementasjon - editering og kompilering - og testing. Verktøy for kontroll med kildekode er viktig. Et eksempel på et slikt verktøy er SCCS – ”Source Code Control System”. **Se fig. 19.5**. Ofte vil slike verktøy også inneholde muligheter for generering av testsdata og for simulering.
- MWB: dette inkluderer verktøy for konfigurasjonsstyring, kostnadsestimering, planlegging – hvem gjør hva når. I tillegg kna det være aktuelt med verktøy for pålitelighetsestimering, statusvurdering – prosjektframdrift, oppnådd testdekningsgrad og lignende.
- IPSE: Dette er en samling / integrasjon av AWB, PWB og MWB. Mye av arbeidet som er gjort på dette området har hatt som mål å få til en god infrastruktur som gjør det lett å få alle verktøyene til å samarbeide. Ett eksempel på en slik infrastruktur er PCTE – ”Portable Common Tool Environment”. PCTE inneholder fire klasser av tjenester:
    - Basismekanismer, for å sette sammen, eksekvere og kommunisere med de enkelte verktøyene. Det er viktig å kunne starte og stoppe prosesser, I/O, filesystemer, meldingsutveksling mellom prosesser osv.
    - Mekanismer for å kunne kjøre systemet på en distribuert maskinvareplattform.
    - Brukergrensesnitt – uniformt for alle verktøy. Dette er viktig slik at man ikke skal behøve å holde styr på like mange brukergrensesnitt som det er verktøy.
    - Verktøy for å manipulere de objektene som finnes i databasen. Dette er det nye i IPSE, og kan best oppfattes som et generalisert filsystem. De viktigste egenskapene er et sett med attributter som beskriver hvert enkelt objekt og lenker mellom objektene.

Eksempel på et modulelement i en IPSE database.

- Modulen har en Id og en status – for eksempel under implementasjon
- Den kan være lenket til det subsystemet den tilhører og til den personen som er ansvarlig for å implementere den

Det finnes mange IPSE systemer, men ingen av de har tatt helt av. De fleste er noe i bruk.

Prosessentrerte omgivelser

I en prosessentrert omgivelse er en IPSE eller en eller flere verktøybenker koblet sammen med en prosessbeskrivelse. Ofte inkluderer dette et verktøy for prosessmodellering. Det finnes spesialverktøy som ”Process Weaver”, men det er også mulig å bruke enkel, generelle notasjoner som Petrinett. **Vis fig 19.7** og forklar de viktigste momentene i en Petrinett modell.

Formelle modeller av utviklingsprosesser må rimeligvis bli rigide. Det at det stadig finnes unntak fra de definerte reglene, gjør at disse verktøyene blir tungvinte å bruke. Viss vi ser på modellen i **fig 19.7** kan vi tenke oss følgende unntak – som eventuelt også må modelleres inn:

- Noen ”roter bort” et møtereferat
- Ledelsen bestemmer at et planlagt review ikke er nødvendig – for eksempel på grunn av tidspress
- Møte må utsettes på grunn av sykdom

Alt dette kna modellers inn, men vil rakst føre til en svært komplisert modell. Det å utvikle, definere, debugge og vedlikeholde modellen kan bli en stor oppgave, som vil kunne stjele tid fra den egentlige utviklingen. På den måten går de prosessentrerte utviklingsomgivelsene over fra å være en hjelp til å bli et ekstra problem.

Sluttkommentarer

Mange trodde på et eller annet tidspunkt av CASE verktøyene skulle løse mange av de problemene programvareutviklerne sto overfor og dermed gi oss bedre programmer – bedre kvalitet – raskere og billigere. Dette har imidlertid ikke hendt. Det kna være mange grunner til dette og det som kommer her er bare noen av mulighetene, men det er de ***jeg*** tror er viktigst:

- Hovedproblemene i programvareutvikling er ikke primært programvareorientert – de skriver seg fra problemer med å forstå kundens behov og problemer. Det vi skulle trenge er verktøystøtte til å forstå og løse problemer. Dette er imidlertid langt fra enkelt.
- Vi kan bare gi verktøystøte til det som kan formaliseres. Programvareutvikling er primært en innovativ, kreativ prosess og dermed vanskelig å formalisere. De delene vi forstå – for eksempel språkdefinisjoner – er allerede formalisert og automatisert gjennom kompilatorer.
- Store formaliserte systemer er
    - tunge å bruke og krever stor maskinkapasitet. Det kan føre til at vi får problemer ved at verktøyet blir en flaskehals i utviklingen
    - vanskelig å oppdater eller legg inn nye verktøy. Dette fører lett til at de snart gjør at vi må bruke gårsdagens løsninger på morgendagens problem – en ikke altfor lystig tanke.