inf1050-notater

31 May 2010

Contents

1 Kapittel 1 - Hva er programvare?

1.1 Introduksjon

Programvare brukes over alt i dagens samfunn. Har ingen direkte fysisk begrensing, og kan derfor fort bli svært komplekst og uoversiktelig. Prisantydning kan være vanskelig og prosjekter slår ofte feil og overskrider budsjettet. Software Enginering prøver å endre dette ved å gi verktøy for prosjektanalyse etc.

1.2 Hva er programvare?

- Software (en samling programmer og tilhørende dokumentasjon etc.). Vi har govt sett to typer:

  1. Standard konsumer-programvare.
  2. Kustomisert programvare (spesialdesignet for ett firma etc.)

Fem viktige attributter til god programvare:

- God, forståelig kode for senere forbedringer.
- Stabil
- sikker og fortrolig programvare.
- Effektiv.
- Brukervennlig.

1.3 Hva er systemutvikling?

- Systematisk og organisert fremgangsmåte. Tar også hensyn til organisatorisk og økonomiske begrensninger i prosessen.

På grunn av stor innvirkning av samfunnet er software engineers nødt til å opptre etisk og tenke konsekvenser.

2 Kapittel 2 - Sosiotekniske systemer

Et sosio-teknisk system består av mer enn bare programvare (maskinvare, mennesker).
Et system er en sammensetning av komponenter som sammen jobber mot et mål.

2.1 Definisjon teknisk system

Et teknisk system er systemer som inneholder både programvare og maskinvare, men prosedyrer eller prosesser.

2.2 Definisjon Sosio-teknisk system

Et sosio-teknisk system inneholder i tillegg operatører som har kunnskap om hvordan systemet skal brukes. (Karakteristikker er, ikke-deterministisk output ved lik input,

2.3 Om systemer

Et system er en sammensetning av flere deler som virker sammen. Ikke enkeltkomponenter, men samspillet. Eksempler: Fly, biler, bankterminaler, etc.

Typer av egenskaper: Funksjonelle og ikke-funksjonelle egenskaper. Altså være et transportmiddel (funksjonell) eller sikker, pålitelig, ytelse, etc (ikke-funksjonell).

Et system har tre punkter som avgjør pålitligheten:

1. Hardware-feil.
2. Software-feil.
3. Operatørfeil.

Sikkerheten til et system er det vanskeligste å anta. Man vet aldri om et system er helt sikkert, kun når det ikke er sikkert.

2.4 Systemutvikling

Systemutvikling gjelder for spesifisering, design, implementering, validering, iverksette og vedlikeholde sosio-tekniske systemer. Siden mange utviklingsavgjørelser gjøres på grunnlag av systemutviklingsprosessen er det også viktig at utviklerene vet om prosessen.

2.4.1 Vi har tre former for system-krav

1. Abstrakte funksjonskrav: Basisfunksjonene til systemet.
2. Sysem-egenskaper: De ikke-funksjonelle kravene.
3. Hva systemet ikke må gjøre.

2.4.2 Systemmodelering

Deler opp systemet i sub-systemer og ser på avhengighetsforhold. Sub-systemene kan så igjen være nye uavhengige systemer, eller komponenter man henter inn. I systemintegreringsprosessen tar du alle de uavhengig utviklede sub-systemene og setter sammen til et komplett system. Pleier å gjøres inkrementelt. (sub-systemene er sjeldent ferdig samtidig og gjør feilsøking letter og billigere).

2.4.3 System-evolsjon

Det gjøres etterhvert endringer i systemer, både hardware og funksjons/software-messig. Dette blir etterhvert dyrere, og man må være forsiktig, siden det kan oppstå problemer med sub-systemene og hvordan de spiller sammen.

2.4.4 System-dekomponering

Hardware-messig kan det bety å pakke sammen utstyr og sende til gjenvinning. Software kan bety å hente ut verdifull data, til en ny database etc.

2.4.5 Større systemer

På større systemer har man ikke mulighet til å gjøre hele system-utviklingsprosessen selv, og man setter ut f.eks. deler av sub-systemene på anbud.

2.5 Rammer

Menneskelige, politiske og organisatoriske bestemmelser har en stor effekt på sosio-tekniske systemer.

3 Kapittel 3 - Kritiske systemer

Kostnadene for kritiske systemer er MYE høyere. Må gjennomgå mye dyr testing og bevisføring av systemet og koden. Det er også mye mer som står på spill om noe galt skulle skje, både økonomiske skader og menneskelige skader.

Den mest viktige emergens-egenskapen til et kritisk system er pålitelighet (som igjen kan bestå av tilgjengelighet (at den på ett gitt punkt faktisk svarer deg på forespørsel), reliability (sannsynlighet for at systemet fungerer), sikkerhet (safety) og sikkerhet (security).

3.1 Sikkerhets-kritiske systemer:

Kan resultere i skade, tap av liv eller miljøskader.

3.2 Oppgave-kritiske systemer:

Feil gjør at hovedoppgaven feiler. F.eks. styringssystem til fly.

3.3 Business-kritiske systemer:

Fører til store økonomiske skader for bedriften.

3.4 De tre hovedårsakene til feil

1. hardware-feil
2. feil i spesifiseringen av systemet
3. operatør-feil.

Må ta hensyn til menneskene som er med i systemet, samt alle komponentene som jobber sammen.

4 Kapittel 4 - Systemutviklingsprosess

Hvordan programvare skal produseres. Organiseringen rundt det å skrive kode etc.

- Økt produktivitet og kvalitet.

4.1 Programvare er en del av samfunnet.

Utfordringer med å levere tilfredsstillende kvalitet. Tidsplan og budsjett overskrides svært ofte. Mange prosjekter feiler.

- Livssyklus: Fra oppstart av utvikling til nedleggelse.
- Utviklingsprosess: Fasene fra oppstart, utvikling til leveranse.

Store kostnader med utvikling av systemer og dermed har man over tid fått mange forskjellige utviklingsmodeller som kan hjelpe oss å minimere risiko og kostnad for utviklingen.

4.2 Fossefallsmodellen:

1. Requirements analysis and definition: Tjenester og mål for systemet defineres, etter samtaler med bruker.
2. System and software design: Beskriver system-arkitekturen.
3. Implementation and unit testing: Sjekker hver komponent om de går sammen etc.
4. Integration and unit testing: Komponenter settes sammen og testes som et system. Krav sjekkes. Leveres!
5. Operation and maintanane: Systemet innstaleres, og eventuelle usette feil rettes,nye krav oppdages og legges til.

Gjør få iterasjoner. Fasene avsluttes for ny startes. Dyrt med endringer. Kun når kravene er meget godt definert, og man føler seg sikker på prosessen.

4.3 Evoutionary development:

Kommer med raske utkast, viser dem til kunden og får tilbakemeldinger og gjør endringer. Dette gjøres til resultatet er akkseptabelt.

1. Exploratory develpoment: Vidreutvikler de delene man forstår, har samtaler underveis.
2. Throwaway prototyping: Eksperimenterer med de dårlig forståtte kravene, og kaster det.

Problemer: Litt vanskelig for ledere å vite lengde, estimere etc. Samt at systemet blir dårligere strukturert når man bare legger til og legger til.

En kombinasjon av evolusjonær og fossefall kan være bra. Definere krav med evo. og fossefall på de godt forståtte delene.

4.4 Component based software engineering:

Tar sikte på å gjenbruke komponenter.

1. Component analysis: Ser på krav, og leter etter komponenter. Passer ikke alltids nødvendigvis.
2. Requirements modifications: Ser på kravene igjen i forhold til kompoonenter. Endrer på kravene hvis det lar seg gjøre. Hvis ikke letes det etter nye komponenter.
3. System design with reuse: Framework settes sammen. Noen komponenter må kanskje designes, hvis det ikke finnes noe å gjenbruke.
4. Development and integration: Noe kode kan måtte skrives her. Systemet settes så sammen.

Billigere med gjennbruk, prosessen kan gå raskere, men kompromisser på alltids gjøres i forhold til kravene.

4.5 Process iteration:

Kravene forandrer seg stort under en prosess. Ny teknologi, press utenfra, forandring i ledelse etc. Inkrementelle prosesser prøver å ta forbehold om dette.

- Systemspesifikasjonene er ikke ferdig før siste inkrement er ferdig.

4.5.1 Incremental delivery:

Skriver generelle outline-krav og struktur. Så skriver man krav til enkeltinkrementer, utvikler dem og leverer deler, og skriver nye krav til nye inkrementer.

1. Fordeler: Kan få ett fungerende system tidlig. Mest kritisk uvikles først. Kan ut fra erfaring forme nye krav til inkrement. Lavere risiko for total prosjekt-kolaps.
2. Spiral development: Utviklingsfasen er illustrert i en sipral som går utover.
3. Objective setting: Formål med fasen settes. Risiko settes.
4. Risk assesment and redction: Analyse av risk blir gjort. Nødvendig handling utføres.
5. Development and validation: Utviklingsmodell velges. Fossefall, inkrementell, evolusjonær etc.
6. Planning: Prosjektet evalueres og man ser om det er nødvendig å gå en spiral til.

4.5.2 Process activites

- Software specification:

  1. Feasibility study: Ser om det er mulig å utvikle, om det er lønnsomt og om det kan gjøres innenfor budsjett. Gjøres raskt. Gir klarsignal for vidre arbeid.
  2. Requirements elicitation and analysis: Spesifiserer krav gjennom observasjon av tidligere systemer og ved å snakke med potensielle brukere.
  3. Kravspesifikasjon: Definere informasjonen samlet i punktet over i kravdokumenter av funksjoner.
  4. Krav-validering: Kontrolerer og sjekker over kravene.

  Software design and implementation: Fører spesifikasjonene til kjørbart system.

  - Designer en skisse av systemet og arkitektren på forskjellige abstraksjonsnivåer.

- Software validation:
  Kontrollerer at programvare er slik den skal være, og tilfredsstiller kunden. Hvis godkjent fortsette på ny iterasjon, hvis ikke gå den samme iterasjonen på nytt

- Software evolution:
  utvikling og vedlikehold knyttes mer og mer sammen, og software-produksjon er en evolusjonær prosess som utvikler seg.

4.6 Prøv-og-feil

Dårlig idè, man har ingen forutsetninger for å planlegge eller estimere tid

4.7 TODO Oblig-metoden. Funker dårlig på større systemer. Vanskelig for samarbeid.

Utdype?

4.8 Prototyping

Introduser for å avhjelpe problemer ved fossefallsmodellen. Lager protyper som du kan vise frem. Greit å vise noe visuelt. (Har både kast-prot. og evolusonær-protyp. (bruker siste utkast).

4.9 Evolusjonære modeller

Fossefall forutser forutsigbarhet og repeterbar. Det er det ikke. Evolusjon er ett svar på disse modellene. Har iterasjoner av Iterasjonsplan, analyse og design, programmering, test. Inkrementerer for hver iterasjon.

- Kravene kan også komme etterhvert. Støtter også endringer under veis.

Men mindre formalisme, krever disiplin.

4.9.1 RUP

Arkitektursentrert, objektorienterte utviklingsprinsipper, UML-modelering er sentralt. ModellDrevet utvikling
Hybrid prosessmodell. Basert på UML.

1. Inception: Buisness-analyse og eventuelt klarsignal.
2. Elaboration:
   - Forstå problem-området
   - få oversikt over rammeverk-arkitekturen
   - risk-analyse.
   - Ved slutt har man use case UML.
3. Construction: Programmet utvikles og dokumenteres.
4. Transition: Fører det over til brukere.

4.9.2 Computer-Aided Software Engineering (CASE)

Kan automatisere noe av utviklingsprosessen. Blant annet grafisk system modeller, generere grafisk brukergrensesnitt, programdebugging, oversette programkode fra gammel til ny.
Blant annet Genova. Tegner UML og får generert kode.

- Datamodell og klassediagram. Brukerdialoger(skjermbilder) for tilbakemeldinger fra interessenter.

4.9.3 Smidige (agile) metoder

Mindre formalisme og krav. Oppfordrer til direkte muntlig kommunikasjon. Eksempler er XP (Extreme programming) og Scrum. Individer og kommunikasjon fremfor prosesser og verktøy. Samarbeid med kunder fremfor kontrakter.

- Endrighetsvillig.
- XP (Extreme programming)
  Veldig rettet mot hvordan gjennomføringsrettet, med programmeringsteknikker for eksempel par-programering (to programerere på en maskin, pilot og kopilot der piloten programmerer og kopiloten validerer koden fortløpende) og også måter å gjennomføre arkitektur og testing av systemet. Metodene er ikke nødvendigvis bare gjennomførbare med XP, og brukes ofte som teknikker under andre systemutviklingsprosesser.

  Fokus på programmering, test og tilhørendeteknikker. Få krav til spesifikasjon og planlegging. Raske iterasjoner (1-3 uker). Ineresentene integrert i prosjektorganisasjonen.

  Prosjekt og prosjektarbeid

  Engangsoppgave som ikke er utført tidligere. Skal lede til ett bestemt resultat. Krever ulike tverrfaglige ressurser. Begrenset i tid.

4.10 Vikige elementer i prosjektplanlegging:

Prosjektarbeid er å organisere og kontrollere prosessen. Ulike utviklingsmodeller er forslag/tilnærminger til en løsning.

For å Planlegge: brukes resursene riktig?. Oppføligng. Og korreksjon (budsjett, leveransedato).

Styringsgruppen: De økonomiske ineressene. Overordnet styring. Prosjektlederen: Daglig ledelse av prosjektet. Ansvar for fremdrift etc. Skriver rapporter.

Viktige faktorer

1. Kostnadsramme
2. Tidsramme
3. Personalramme (antall prosjektdeltagere og kompetanse)
4. Utstyrsramme (maskiner, programvare, nettverk, etc)
5. Krav til leveransene
6. Offentlige krav (lover, retningslinjer, etc)
7. Produksjonstekniske krav

Viktige elementer i prosjektplanleggingen

1. Identifisere og planlegge mål og delmål.
2. Prioritere oppgaver.
3. Estimere arbeidsomfang.
4. Beslutte start og sluttdato.
5. Holde oversikt over avhengigheter mellom aktiviteter.

Bruker timeboxing for å holde angitt tid. Starter med høyeste priorierte oppgaver.
Gjør en oppfølging av fremdrift. Alle rapporterer tidsbruk. Planer oppdateres gjenvlig.
Ved korte iterasjoner har man råd til å feile.

4.10.1 EØS lov om anbud

Hvis det er et offentlig firma/en offentlig etat som skal leie inn leverandør må det være en åpen anbudsrunde dersom estimert pris er over 500 000 kroner

5 Kapittel 5 - Prosjekthåndtering

5.1 Management activities:

Prosjektleder holder orden på alt. Snakker med prosjektarbeidere. Kan opdage problemer tidligere en å bare vente på at de opptrer.

5.2 Prosjektplanlegging:

Nøye planlegging. Forutse problemer som kan oppstå, klare å komme med løsninger. Planer endres underveis hele tiden. Prosjektplanleggingen går i en løkke til prosessen er ferdig. Og ser man noe går feil må man gjøre nye avtaler med kunden. Man bør bygge inn litt tid til å feile.

5.3 Prosjektplan:

Setter opp resurser tilgjengelig, arbeidsoppgaver og en tidsplan for arbeidet.

- Introduksjon: Kort forklaring, pluss budsjett, tid etc.
- Prosjektorganisering: Roller og personer i utviklingsteamet.
- Riskanalyse: Riskanalyse og riskhåndering.
- Hardware og software-krav: Hva som kreves/trengs.
- Arbeidsoppgaver: Milestones, aktiviteter og estimert levering.
- Prosjektkalender: Avhengighetsforhold og estimert tid til vær milestone.
- Monitor og rapporterings-systemer: Hvordan prosjektet monitores og hvilke rapporter som skal skrives.
- Denne planen kan endre og utvikle seg.

5.3.1 Milestones og leveringer:

Siden man ikke ser prosessen fysisk utarte seg, må det leveres rapporter som forklarer hvor man er i prosessen etc. Milestones er logiske målpunkter i prosjektutviklingen. Deliverables (leveringer) er resultater man leverer kunden. Som oftes er dette milestones, men ikke andre veien.

5.3.2 Prosjekttidestimering og aktivitetsnett

Vanskelig jobb. Opdateres gjevnlig etterhvert som mer informasjon kommer inn. En aktivitet bør ta fra 1 til 8-10 uker. Samt må resursser estimeres. Mennesker og eventuelt hardware.

Noen legger til 30% så 20% tid for uforutsette hendelser.

- Setter ofte opp en tabell med Task, duration og dependencies. Lager så aktivitetsnettverk, og finner tiden prosjektet tar ved å finne den kritiske veien.

For å få oversikt over tidsdisponeringen kan man også bruke Gantt charts. Kan også brukes til "staff allocation". Med navn og oppgaver nedover.

- Eksempel på aktivitetstabell

navn	Tidsestimat	Oppgave	Dep	kommentar
T1	3 dager	Kravspesifisering	none	funksjonalitet
T2	3 dager	Prosjektevaluering	T1	Tidligere systemer?
T3	20 dager	Database og sentralsystem	T2	
T4	5 dager	Faktureringssytem	T3	
T5	15 dager	Webgrensesnitt	T3	eksterne designere?
T6	10 dager	Internt grensesnitt	T3	
T7	20 dager	Integrering av systemene	T4,T5,T6	
T8	10 dager	Sluttesting og bugfiksing	T7	

5.3.3 Risk management

1. Prøver å se problemer som kan oppstå på forhånd og ta forhåndsregler. Riskplan dokumenteres i prosjektplanen.
2. Project risk: Forskyver prosessen eller endrer resursene.
3. Product risk: Feil som påvirker kvaliteten og ytelsen til software-produktet.
4. Business risk: Risker som kan angå firmaet. F.eks. ny konkurranse etc.

Risk identifisering -> Risk-analysering -> Risk-planlegging -> Risk-overvåking.
Så graderer man risikoen etter sannsynlighet og utfall.

- Eksempel på risikoanalyse (fra oblig 1)

1. Databasen blir kapret av uønskede
   - Kundens vurdering:
     * sansynlighet: 3
     * konsekvens: 4
   - Leverandørens vurdering:
     * sansynlighet: 2
     * konsekvens: 5
   - Tiltak: Adskille webgrensesnittet og det interne grensesnittet i metoder og generelt minske mulighet for databasemanipulering fra web. Og at innlogging fra interne terminaler er passordbeskyttet uten at pas- sordene er lett tilgjengelig.
   - Ansvarlig begge
2. Nøkkelpersoner forsvinner fra prosjektet
   - Kundens vurdering:
     * sansynlighet: 2
     * konsekvens: 4
   - Leverandørens vurdering:
     * sansynlighet: 2
     * konsekvens: 4
   - Tiltak: Tett sammarbeid og god kommunikasjon fra nøkelpersoner og videre til deres kolegaer, slik at plassen kan fylles. Fjerne urelevante arbei- dsoppgaver fra personell som er engasjert i prosjektet
   - Ansvarlig begge

6 Kapittel 6 og - Kravhåndtering, Programvare-krav og Kravspesifisering

Kravspesifikasjon kan deles inn i to grupper.

- *Bruker-krav.* Et løst definert krav-dokument, som spesifiserer kravene til systemet på ett overordnet nivå.
- *System-krav.* En mer detaljert beskrivelse av delene til systemet, og nøyaktig hva som skal bli implementert. Kan være en del av kontrakten.

6.1 Foranalyse

Uten en skikkelig foranalyse kan det være vanskelig å vite hvilke risikoer som er involvert, om systemet trengs, eller om det har livets rett økonomisk sett.

Forutsigbarheten for prosjektet øker og man vil være bedre i stand til å levere prosjektet til rett tid eller minimere overskridelser av planen

de første spørsmålene man må stille seg er

- er prosjektet gjennomførbart?
- teknologisk gjennomførbart?
- er det tidsmessig og kostnadsmessig lønnsomt
- og samsvarer systemet med målene til selskapet
- hvem vil berøres av systemet (interessenter)

Man finner så informasjon i forhold til disse problemene, og lager en rapport på bakgrunn av dette. Man ser også på om dette vil være en forbedring for organisasjonen, i forhold til gamle systemer, takler den gammel data fra tidligere som organisisajonen måtte ha. Man spør også alle som måtte ha interesse av systemet om det er gjennomførbart, og om det bør gjennomføres. Vanlig tid for prosessen er 2 til 3 uker.

6.2 Interessenter(stakeholders) og kravinnsamling

Krav kommer ofte fra interessentene rundt systemet kort fortalt er en interessent en hvilken som helst gruppe/individ som berøres av systemet direkte eller indirekte.

eksempler på interessenter:

- sluttbrukere av systemet(interessert i enklere jobb)
- organisasjonen rundt sluttbrukere(kan bli påvirket av bedre systemer)
- kjøper av systemet(vil at det skal være økonomisk lønnsomt)
- Leverandør av systemet
- Leverandører av lignende systemer
- etc

Hvorfor er det vanskelig å finne og forstå krav fra interessenter (fra boka)

1. Interessenter vet ofte ikke hva de vil ha "jeg vet hva jeg vil ha når jeg ser det"
2. Interessenter vil ofte beskrive krav ut i fra sin egen implisitte kunnskap, og å forstå hva de faktisk vil ha/trenger kan ofte være en utfordring
3. Interessenter kan ha motsigende krav, fordi forskjellige interessenter er interessert i forskjellige mål
4. Politiske faktorer kan gi gi innflytelse på systemkravene eks: overvåkning eller logging av bruksmønstre etc (datatilsynet!)
5. Organisasjonen kan bli berørt på mange måter av et nytt system, og nye interessenter kan dukke opp midt i utviklingsprosessen, og gi nye krav eller forandre viktigheten av eksisterende krav.

Viktig å få de riktige kravene helt fra begynnelsen. Det kan bli fryktelig mye dyrere å gjøre endringer senere i prosessen, selv om man må huske på at endringer i krav er uungåelig.

ved endringer må endringen dokumenteres, det må gjøres konsekvensanalyse og implementere. Spor gjerne endringene i et verktøy.

Til å hjelpe med å finne kravene, kan man se på spesifikasjonen til tidligere, lingnende systemer, gjøre intervjuer med stakeholder, kjøre tester med prototyper.

Det kan også ofte være lurt å sette opp scenarioer for brukere og interessenter, slik at det blir lettere for dem å sette seg inn i problemstillingen. Man kan f.eks. skape disse scenarioene ved tekst, prototyper eller skjermbilder.

Vi kan også lage **use cases** for å hjelpe oss med kravinnsamlingen, eller benytte **etnografi**, hvor man overvåker bruksmåten, f.eks. på en arbeidsplass, og henter kravinformasjon på den måten. Man får ofte informasjon som man ikke ville fått ellers.

Metoder for kravinnsamling

- Intervjuer
- spørreskjemaer
- observasjon
- studere dokumenter
- eksisterende systemer
- idédugnad(brainstorming).
- Prototyp: Bruk og kast ideer du viser til brukere etc. F.eks. ved Genova.

flyt

1. Kravinnsamling og -analyse: Identifiser krav, prioriteter, løs konflikter mellom interessenter. Ofte lurt å visualisere på forhånd for interessenter. Da vet de lettere hva de ønsker etc. prioriter krav
2. Kravspesifikasjon: Organiser og kategoriser alle kravene. Spesifiser presist, f.eks. ved UML.
3. Validering av kravspesifkasjon: Forståelighet, konsistens, testbarhet, sporbarhet (hvem er kilden til kravet), endringsevne (konsekvenser av å endre?), kompletthet, nødvendige krav, realistiske krav, for tidlig design.
4. Dokumentering av krav, kravdokumenter skrives

6.3 Krav-validering

Krav-validering kontrollerer at kravene som er samlet, faktisk representerer det kunden ønsker. Feil i kravene kan bli dyrt, siden det kan føre til at endringer i systemet må gjøres på et sent tidspunkt. Ting man bør kontrollere er blant annet,

- Krav ikke kolliderer med hverandre.
- Slå sammen og gjøre kompromisser mellom krav.
- At kravene definerer alt som er ønsket av systemet.
- Muligheten for å implementere kravene.
- At kravene er testbare/verifiserbare i ettertid, i forhold til kunden.

Man bør så ha en **kravgjennomgang** med klient og tilbyder, for å snakke seg gjennom kravene, og avdekke feil og mangler.

6.3.1 hvorfor trenger vi veldefinerte krav

"a camel is a horse designed by a committee"

Vi trenger veldefinerte krav for å ha holdeplasser i virkeligheten når vi utvikler et system, hva skal systemet gjøre og hvordan skal det fungere. Krav er et slags sikkerhetsnett for at man lager et system som samsvarer med kundens behov.

6.3.2 Krav til krav

gode krav bør bestå disse kravene

- Er de forsåelige?
- Er det konsistens?
- Er det kompletthet?
- Er de testbare?
- Er de verifiserbare?
- Er de relevante?

6.4 Kravhåndtering

Kravene til et system, spesielt av litt størrelse, vil altids endre på seg, siden man ser problemet og systemet i et nytt lys, etterhvert som det trer frem. Det er derfor viktig å ha en evolusjonær løsning, som lar deg endre på kravene underveis. For store systemer kan det ta flere år å finne kravene, og da kan påvirkende elementer i miljøet rundt allerede ha endret seg. Dette må man også ta hensyn til.

Man setter gjerne opp en avhengighets-matrise på kravene, for å kunne kontrollere om enkelte endringer av krav, vil påvirke andre deler av systemet. For større systemer setter man opp egne databaser, som kan gjøre dette automatisk.

6.5 Funksjonelle og ikke-funksjonelle krav

Programvare-system-krav klassifiseres ofte i tre kategorier funksjonelle, ikke-funksjonelle eller domene-krav.

6.5.1 Funksjonelle krav

De funksjonelle kravene bør være **komplette**, altså alle krav er definert, og **konsistente**, ingen selvmotsigende krav.

sier hvilke tjenester systemet skal utføre, hvordan det skal oppføre seg i spesielle sitasjoner, hvordan svare på input.

- Tenk hva slags krav dere vil ha for å kunne lage use cases
- Konkrete oppgaver som skal utføres
- Enten/eller-scenarioer
- Eksempler fra Oblig 1
  - Systemet må kunne vise oversikt over ledige hotellrom
  - Resepsjonister og nettbrukere må kunne booke hotellrom
  - Systemet må kunne skrive ut raporter for alle hoteller
  - Nettbrukere må kunne få opp oversikt over sine reservasjoner

6.5.2 Ikke-funksjonelle krav

* Produktkrav

- Brukervennlighet
- Effektivitetskrav
- Pålitelighetskrav
- Portabilitetskrav

* Prosesskrav

- leveransekrav
- Implementasjonskrav
- Krav til standard

* Eksterne krav

- Lovmessige krav
- Etiske krav

Ikke-funksjonelle krav bør, så langt det lar seg gjøre, skrives som testbare krav, slik at man kan avgjøre om kravet er møtt. Ungå vagt definerte krav.

- Elsempler
  - Ytelse
    * Systemet skal behandle alle responser på under 1 sekund
    * Systemet skal ha en oppetid på 99,9%
  - Sikkerhet
    * Systemet skal tilby full backup 6 måneder tilbake i tid
    * Systemet skal ha sikker og kyptert forbindelse mellom hotellene og databasen
  - Andre ting
    * Krav til brukervennlighet
    * juridiske krav Personopplysningsloven etc.
    * Ikke gjennbruk av mailadresser til spam?
  - Testing
    Whitebox og blackbox-testing

6.6 Bruker-krav

Bør skrives, så langt det lar seg gjøre, så enkelt og lett forståelig som mulig. Skal kunne beskrive funksjonelle og ikke-funksjonelle krav til folk uten særlig teknisk kunskap. Unngå programvare-sjargon. Men man må passe på, for man mister ved dette ofte mye av klarheten og entydigheten. De bør heller ikke være for detaljerte, siden det minsker muligheten til utvikleren for gode og kreative løsninger på problemet.

6.7 System-krav

System-krav er en utvidet utgave av bruker-krav. De legger til ett høyere detaljnivå, og brukes som ett startpunkt for systemutviklerene i deres systemdesign. Det brukes også ofte i kontrakten, og bør derfor være detaljert og nøyaktig. Egentlig skal systemkrav ikke inneholde *hvordan* et system skal designes eller implementeres, men dette er ofte vanskelig å ungå. Blant annet fordi systemet kanskje skal fungere med tidligere systemer, etc. Vi kan bruke **strukturert, formatert spesifikasjon** hvis vi ønsker litt mer presisjon i krav-beskrivelsen, enn hvis vi bruker naturlig språk.

6.8 Interface specification

I de fleste tilfeller skal nye systemer jobbe sammen med tidligere systemer. Interface-spesifikasjonen (grensesnitt-kommunikasjonen mellom de to) må derfor være svært tydelig, så det ikke oppstår kommunikasjonsproblemer mellom systemene, og det bør komme tidlig i krav-dokumentet. Det finnes flere former for Interfaces man må ta hensyn til.

- Eksisterende APIer.
- Representasjon av data.

6.9 Programvare-krav-dokumentet

Software requirements document er det offisielle utsagnet om hva utviklerene skal implementere. Dokumentet skal rekke ut til mange forskjellige lesere, fra senior management til utviklerene. Detaljgraden til dokumentet avhenger litt av utviklingsprosessen. Hvis utviklingen skal outsources til ett eksternt selskap er man nødt til å beskrive kravene mye mer detaljert, så det ikke oppstår feiltolkninger.

7 Kapittel 8 - System-modeller

En systemmodel representerer systemet på en mer overordnet, abstrakt og helhetlig måte.

7.1 Context models

Her jobber man med å finne grensene til systemet, og grensene til de forskjellige delene i systemet.

7.2 Behaviour models

Jobber med å beskrive oppførselen til systemet. F.eks. dataflyt gjennom programmet, eller hvordan systemer reagerer på spesielle hendelser.

7.3 Datamodeller

Datamodeller er digrammer som beskriver systemet på en objektorientert-lignende måte, bare med dataen i systemet, istedenfor objektene. Man tegner opp enheter, attributter, tjenester og deres relasjoner (med navn og antall/forhold/relasjoner).

7.4 Objektmodeller

Mye brukt i programvareutvikling, med objektorienterte språk. Representerer systemet i en objektorientert måte, med UML. Ofte er denne metoden veldig naturlig, siden den beskriver virkeligheten på en naturlig måte, med objekter.

Objekter kan også arve egenskaper fra mer genrelle objekter i ett klassehierarki. Spesialiserte objekter kan så legge til egne atributter og egenskaper.

7.5 Structured methods

Karakteristikker:

- Brukes til kravspesifisering og systemdesign.
- Dele prosjektet i veldefinerte aktiviteter.
- Bruke diagrammodelering etc. på prosjektet.
- Gi en god og strukturert definisjon av systemet.
- Skal forstås av klient og utvikler.

  Ofte bruker man avanserte **CASE** (Computer-aided software engineering)-verktøy, for å hjelpe til med denne prosessen. Det er verktøy som kan alt fra datamodelering, automatisk generering av kildekode, og brukergrensesnitt-manipulering.

8 Kapittel 13 - Applikasjons-arkitektur

8.1 Data-processing systems

Får input-data fra database, eller filer, som den utfører en prosess på og sender det ut igjen, enten tilbake i databasen, eller printe ut i ny fil.

Slike prosesser er naturlig å representere i data-flyt-diagram-representasjoner.

8.2 Transaction-processing system

Programmer som prosesserer spørringer mot databaser, eller oppdateringer av databaser. Man sørger for at alle handlinger utføres trygt, før det speiles i databasen, slik at ikke databasen blir kurupt eller inkonsistent.

8.3 Event-processing system

Programmer som prosesserer hendelser i interfaces, gjort av brukeren, i en tilfeldig rekkefølge. F.eks. Word Processors, bildebehandlingsprogrammer og spill.

En del av disse, f.eks. teksteditorer har behov for svært rask behandling av data, etter spesifikke eventer, og endringen foregår direkte i en buffer i minnet.

8.4 Language-processing system

Tar et naturlig, eller artificial konstruert språk, og generer en annen form for representasjon som output. Vanligste eksemplet er **compilers**. Denne komponenten kan være hardware (de fleste kompilatorer) eller software (slik Java er).

9 Kapittel 23 - Software-testing

9.1 Kvaliteter vi trenger i produktene:

- Korrekt programvare
- Pålitelighet
- Robust
- Ytelse

- Brukervennlig

Vi kan lete etter feil allerede i dokumentasjonen.
Continuity properties gjelder ikke for software engineering, slik det gjør for andre ingeniørdisipliner.

- Komponent-testing: Teste deler av systemet.
- System-testing: Teste hele systemet. Tester også funksjonelle og ikke-funksjonelle krav.

Tester av to grunner:

- For å demonstrere for kunde og utvikler at det fungerer tilfredsstillende.
- For å avsløre feil eller mangler i programvaren.
- Skrive tester for å sjekke krav. Trenger ofte flere tester for å få det til. Testing kan kun avsløre feil, ikke fraværet av feil.

Problemer oppstår ofte når man kombinerer funksjoner i programmet. Vanlig prosedyre er at utviklere tester sine egne komponenter og sender de videre til teamet som integrerer dem, og tester hele systemet.

9.2 System-testing

Å teste integreringen av to eller flere komponeneter, og at alt går som det skal. Man tester for hvert komponent-inkrement, for å teste alle kombinasjoner.

9.2.1 Release-testing:

Sender beta-utgaver til folk, for å kontrollere at den tilfredsstiller kravene og at den ikke feiler ved vanlig bruk.

Man prøver altids input som har størst sannsynlighet for å gi en feil i programmet. (Test alle error-meldinger, test buffer overflow, repeter samme input flere ganger, tving frem feil output, tving frem for store eller for små outputs).

9.2.2 Performence-testing:

Tester om hastighet/påletelighet/stabilitet er som det skal være. Stresstester systemet utenfor for kravene som er definert i kravene, og sjekker om den feiler "soft" eller skaper store problemer.

9.2.3 Komponent-testing

Å teste enkeltkomponeneter. Hovedsaklig utvikler som gjør den jobben. Enten indivudelle metoder eller funksjoner i ett objekt, eller objekt-klasser med flere attributter og tilhørende metoder.

9.2.4 Interface-testing:

Teser interface-komunikasjon mellom flere komponenter. Test Case-design

Designer tester. Input og predikert output. Skal være effektive til å teste systemet, og finne eventuelle feil og at systemet tilfredsstiller kravene.

9.2.5 Kravbasert testing:

tester kravspesifikasjonene, om de er møtt. Gjøres i system-testing-delen.

9.2.6 Structural testing:

Passer på at alle delene av kodene kjøres minst en gang.

9.2.7 Partition testing:

Kan teste med kun en input av input med like karakteristikk. F.eks. ett positiv tall, ett negativt etc. Prøver ofte med atypiske verdier, siden utviklere ofte glemmer dette. ved inputverdier velger du verdier midt i områdene og verdier på begge sider av verdigrensene.

9.2.8 Structural testing(white-box):

9.2.9 Path testing:

Vi tegner opp ett diagram over alle mulige grenveier det er mulig å kjøre igjennom. Alle løkker og if-else-settninger blir grener og looper i programmet. Så sørger vi for å gjøre testinger så hver kodeblokk blir kjørt minst en gang, og får testet alt med både true og false.

9.3 Test-automatisering

Testing er dyrt. Kom derfor raskt en del programvare som kan automatisere prosessen. Kan holde orden på testdata, generere testdata, Oracle som predikerer forvenet resultat, File comperator som sammenligner tester, Dynamic analyser som sjekker hvor ofte de firskjellige statementene blir kjørt. Og simulering i forskjellige varianter. F.eks. bruker-interaksjon.

Testingsfasen er ofte estimert til å være 50% av utviklingskostnadene.

10 Kapittel 26 - Software cost estimation

10.1 Software productivity

Handler om å kartlegge produktiviteten til utviklerene, effektivitetskostnadene for jobben, og fordele arbeidet utover. Har ofte behov for dette, for å senere kunne gi en prisestimering. Kan ofte bruke antall linjer kode estimert, som en basis for videre estimat. Eller hvor lang tid man bruker per funksjon i systemet.

LOC/pm er en måte å måle produktivitet på. LinsOfCode / programmer-months. Denne effektiviteten avhenger av hvilket språk som brukes, da forskjellige språk, krever forskjellig antall linjer kode, for å utføre samme prosedyre.

Man kan også funksjons-punkt-estimering. Denne metoden kan gjøres på et tidligere stadie en LOC, siden man vet det meste man trenger etter kravspesifisering etc.

10.2 Estimation techniques

Det finnes flere teknikker for å estimere prisen:

- Estimering basert på kostander til lignende prosjekter.
- Leie inn flere eksperter, for så å sammenligne og diskutere deres estimeringer.
- Parkison's Law: Sier at arbeid vil fylle ut tid som er satt til side.
- Det kan avhenge av hva kunden er tilgjengelig av budsjett.

Det er ofte lurt å bruke flere estimeringsmetoder, for så å sammenligne dem til slutt. Varierer estimeringene kraftig, kan man regne med at man ikke har nok informasjon.

10.3 Algoritmic cost modeling

Algorthmic cost modeling bruker en matematisk formel til å beregne tid og pris-estimatet, som er regnet ut fra tidligere, fullførte prosjekter. En vanlig formel kan se slik ut

$Effort = A \times Size^B \times M.$

- **A** er en faktor, som avhenger av organisasjonen sin praktisering og type programvare som skal utvikles.
- **Size** er LOC eller funksjons-estimering, representert i funksjon eller object points.
- **B** ligger mellom 1 og 1.5.
- **M** er en multiplikator, avhengig av ting rundt utviklingsprosessen og erfaring til utviklere etc.

I starten av estimeringsfasen vil nøyaktigheten variere fra 0.25X til 4X. Utover i prosessen vil det bli tydeligere og tydeligere hvor lang tid det vil ta.. Det er viktig å derfor legge til rette, for å kunne endre rundt dette.

COCOMO er en avansert standard for prisestimering, som har tre forskjellige detaljnivåer, og bruker en rekke forskjellige input til å estimere kostnader og tid, med varierende treffsikkerhet.

10.4 Product duration and staffing

Programvare forventes å komme på markedet raskere og raskere, for å kunne konkurere med motstanderne. Forholdet mellom utviklere og tid er ikke lineært, på grunn av kommunikasjon og management.

COCOMO-modellen har en modell for estimere kalendertid (TDEV),
$TDEV = 3 \times (PM)^{(0.33+0.2\times(B-1.01))}$.

- PM er innsats-estimering.
- B er utregnet eksponent for COCOMO-modellen.

  Ofte trenger man heller ikke mange arbeidere i startfasen, men heller flere etterhvert. Det er da viktig å ikke ta inn for mange arbeidere av gangen, da dette fører til problemer og kan senke prosessen. Det bør gjøres gradvis.

11 UML

Notasjon som støtter opp under modellbasert systemtvikling.

- Godt utgangspunkt for dokumentasjon.

Kan brukes til datamodeliering, arbeidsflyt-modelering eller objektmodelering.

11.1 Use Case

Use case: Beskrive funksjonelle krav ved use case. Beskriver systemet utenfra, og bruksmønsteret. Tegnes med Akøtrer (mennesker og andre komponenter som interakter med use casen) og Use Case (oval).

- Kan også bruke extend og include.

Kan bruke struktrert tekstlig spesifikasjon til hver use case.

Vedlagt ligger bilder av en kravtabell med use-case diagram

11.1.1 Hovedflyt

Brukes for å beskrive hvordan systemet skal fungere ut i fra Use case diagrammer og gjør det enklere å se hva som kan gå galt, og dermed gi alternative flyt

eks. basert på use case diagrammet over

1. Søkeren fyller ut online lånesøknad
2. Søkeren sender søknaden til banken via internett
3. Systemet validerer informasjonen i lånesøknaden ved å sjekke at den er så korrekt og komplett som mulig

4. Systemet innhenter kredittrapport for søkeren fra et eksternt kredittbyrå for kredittraport.
5. Systemet henter søkerens kontohistorie med banken fra kontosystemet
6. Systemet beregner søkerens kredittscore basert på kredittrapport og kontohistorie
7. Systemet informerer søkeren via e-mail om at søknaden er mottatt og blir vurdert
8. Systemet setter status på lånesøknaden til "Initiell kredittsjekk ferdig
9. Systemet allokerer lånesøknaden til en lånekonsulent for videre behandling

11.1.2 Pre- og postbetingelser

Use case "Vurder lånesøknad":
Prebetingelse:

- Lånesøknaden har status "Initiell kredittsjekk ferdig"

Postbetingelse: (en av de)

- Lånesøknaden har status "Godkjent"
- Lånesøknaden har status "Informasjon mangler"
- Lånesøknaden har status "Avslått" og søker har fått beskjed om at søknaden er avslått

11.2 Domenemodeller

Enkle klassediagrammer, uten metoder. Skal beskrive virkeligheten. Tegne opp relasjoner.
Domenemodeller er nyttig i forbindelse med use case modellering fordi

1. Domenemodellen fanger opp informasjon om objekter i use casene og er et viktig verktøy for at use casene er beskrevet med riktig detaljeringsnivå
2. Klassene i domenemodellen kan brukes i utforming av mer presise pre- og postbetingelser

Hensikten med domenemodellen er å forstå objektene og få en oversikt over terminologi.

11.2.1 Forretningsobjekter

De har evig/langt liv. Lagrer i daabase.

11.2.2 Kontrollobjekter

Kontrolerer handlingsforløpet.

11.2.3 Kantobjekter

Kommniserer med brukere /aktørene.

11.3 Sekvensdiagrammer

Bygger på domenemodellen og objektene som er definert der og viser en interaksjon mellom aktører og objekter i systemet for et bestemt bruksmønster.
Det er ofte nyttig med sekvensdiagrammer for å identifisere (og spesifisere bruken av) metodene til objektene i systemet
vedlagt er sekvensdiagram for å melde seg på kurs på uio

11.4 convensjoner

11.4.1 extend

Utvider funksjonalitet, f.eks. i alteriantiv flyt.

11.4.2 include

F.eks. hvis flere use case utfører samme prosedyre, kan vi lage use case av det, og include.

12 PS2000

12.1 Bruksområde:

1. Drift av IT- løsninger
    - Utstyr
    - Programmer
2. Livsløpsperspektiv
    - Etablering
    - Ordinær drift
    - Avslutning

Eies av kunden og/eller leverandøren -Hvor det ikke er mulig eller hensiktsmessig å etablere nøyaktige eller detaljerte kontrakter.

En IT-leveranser er for Maskinvare, programvare og tjenester. Formålet med kontrakt er konfliktforebygging og gjensidig forpliktelser. Mest viktig å fordele risiko?

Har både interne og eksterne rammebetingelser man må ta hensyn til.

12.2 prisformer

Har tre prisformer:

- Fastpris, knyttet til avtalt omfang.
- Løpende timer, fakurerer timer og annet.
- Målpris, basert på estimater og risikovrderinger. Justeres etterhvert.

PS2000 regulerer iterative eller smidige (agile) prosesser. Kan benyttes både av private og offentlige aktører. Utviklet så begge parter tas vare på.

- Kontrakten er i større grad ett styringsverktøy. Basert på iterative metoder.
- Regulerte forpliktelser i begge parter.
- Håndtering av usikkerhet tilrettelagt.

Ukentlig oppdatering av risikomatrise.

Jo lengre ut i prosjektet man er, jo dyrere er endringer, men desto mer vet man hva man ønsker av produktet, og desto større nytte-effekt av endring. Dilemma!

Fleksibel og oversiktelig.

12.3 PS2000 vs andre kontrakter

Kontraktsstandarden skiller seg vesentlig fra andre standarder i markedet. Spesielt kan følgende trekkes frem:

- Kontraktsstandarden er utviklet av kunder og leverandører i samarbeid, slik at begge parters interesser er ivaretatt og balansert.
- Kontraktsstandarden tilrettelegger for å fange opp den læring som foregår under gjennomføring av prosjektet. Gjennomføringsmodellen består av 4 faser
  - behovsfasen
  - løsningsbeskrivelsesfasen
  - en trinnvis konstruksjonsfase
  - godkjennings- og avslutningsfasen
- Det er tilrettelagt for utstrakt bruk av motiverende økonomiske modeller i form av incentivordninger for at eventuelle tids- og kostnadsbesparelser kommer begge parter til gode og vice versa. Det skal utarbeides en usikkerhetsanalyse som legges til grunn ved valg av spesifikke incentiver.
- Samhandling mellom kunde og leverandør forbedres ved at kontraktsstandarden legger opp til et integrert samarbeid og en effektiv og separat prosess for eventuell konfliktløsning.
- Kontraktsstrukturen med forhåndsutfylte bilag og veiledning gjør det enklere å utforme spesifikke kontrakter tilpasset ulike behov. Alle referanser i de generelle kontraktsbestemmelsene er utdypet i bilagene.

12.4 Oppbygging av PS2000

Kontrakten er delt inn i
Del1: Kontraktsdokument Del2: Generelle kontraktsbestemmelser Del3: Kontraktsbilag

13 Kapittel 11 (Ikke hoved) - Arkitekturdesign

Valg av arkitektur-rammeverket influerer viktige deler av systemet, som hastighet, sikkerhet og tilgjengelighet.

Store systemer deles inn i subsystemer. ARkitekturdesign er å fastsette disse subsystemene, og kommunikasjonen dem imelom.

Skisse en grunnleggende struktur av systemet, og hovedkomponentenene og kommunikasjonen i mellom.

Er mer en bestemmelseprosess enn en aktivitet.

Finnes to typer dataarkitekturer. Sentral database, eller lokal database, hvor subsystemene sender dataen til og fra hverandre. Fordelen med klient-server-modellen er at den er disitrbuert, og det er lett å oppgradere og fordele belasten.

Layered-modellen organiserer systemet i lag. Kan endres, så lenge interfacet er det samme. Negative siden er at flere lag kan gjøre at systemet går tregere.

Et subsystem skal være uavhengig, og fungere uavhengig av de andre subsystemene. Mens en modul er en litt mindre enhet, som ofte benytter seg av tjenester i andre moduler etc.

Finnes to måter å visualisere systemarkitekturen på.

1. Objektorientert der hvert objekt er ett subsystem. En fordel med dette er at objektene kan gjenbrukes.
2. Funksjons-orienter der man har input-data som går gjennom forskjellige funksjoner (subsystemer) og får til slutt en output.

14 Kapittel 12 (Ikke hoved) - Distributed system architecture

Så og si alle større systemer i dag, er distrubert over flere maskiner. Det har en del fordeler, blant annet

- Deling av resursser, både hardware og software.
- Ofte designet rundt åpne standarder og protokoller.
- Skalerer lett.
- Stor feiltolleranse, annen server kan raskt slå inn i steden hvis man har flere.

Ulempene er

- Kompleksitet.
- Sikkerhet. Traffiken må gå over nettverk, mulighet for eavsdropping.
- Kan være vanselig å drifte store distribuerte systemer.

Generelt sett kan vi si at vi har to typer distribuerte systemer.

- **Klient-server**. Tilbyr tjenester, som klientene spør etter.
- **Distribuert objekt-arkitektur**. For objektene er forespørlser fra klient og andre objekter akkurat det samme.

Multiprosessor-arkitektur, lar flere prosesser i samme system, kjøre på forskjellige prosessorer.
Klient-server-arkitetur kan ha to former.

1. Tynn-klient.
2. Tykk-klient.

Mobile klienter er ofte en liten mellomting av disse.
CORBA er en standard som støtter mange av disse arkitekturene.
Peer-to-peer er desentralisert, distribuert system, der alle klientene er direkte koblet til hverandre i ett nettverk, og beregninger gjøres på flere maskiner/klienter.

15 Kapittel 16 (Ikke hoved) - Brukergrensesnitt-design

15.1 Et godt grensesnitt er viktig, for god programvare.

Noen ting å huske på er blant annet,

- Mennesker har dårlig hukomelse. Ikke forvirr bruker med for mange valg.
- Må ta i betraktning at enkelte ser dårligere enn andre, noen hører dårlig etc. Ungå å ekskludere folk.
- Bør være familiært.
- Konsistent.
- Forklaringer og forståelig/intuitivt grensesnitt.

Vanlig å ha flere grensesnitt til programmet, egnet til forskjellige bruksmåter og personer. F.eks. web-interface, kommandolinje, grafisk brukergrensesnitt eller meny-basert grensesnitt.

For å kunne ha flere grensesnitt er det lurt å bruke MVC (Model-View-Controller)-metoden, utviklet av Trygve Reenskaug. Den går ut på å skille ut datamodellen i ett objekt, View (grensesnitt) i hvert sitt objekt, og en controller mellom disse, for å håndtere kommunikasjonen.

Feilmeldinger i programvare bør ungå å være negative, bør være lett forståelige og tilby videre hjelp og være smarte.

15.2 Prosessen ved GUI-utvikling:

- Start med analyse av brukere og bruken av programmet.
- Lag en prototype, test den og utvikle den videre.

16 Kapittel 25 (Ikke hoved) - Managing people

En av de viktigste rollene ved prosjektledelse, er håndtering av medarbeidere og holde dem motivert.

16.1 valg av arbeidere

Å velge nye medarbeidere til et prosjekt kan ofte være svært vanskelig, og man er nødt til å analysere hvilke kvaliteter til hvilke stilling man er mest opptatt av. (Teknisk, sosialt, etc.).

16.2 Motivering av arbeidere

Motivasjon er også svært viktig. Uten motivasjon vil arbeidet gå saktere, og arbeiderene vil ikke bidra like konstruktivt til utviklingen eller for firmaet, og kan lettere gjøre feil. En måte å motivere på er å gi oppgaver som er utfordrende, men gjennomførbare, sørge for at arbeidere hele tiden lærer etc. Samt sørge for et godt sosial miljø.

16.3 Generalisering rundt mennesketyper

Generelt sett har man tre kategorier mennesker, som verdsetter forskjellige ting:

- Oppgave-orienterte: Motivert av oppgaven de gjør.
- Selv-orienterte: Motivert av suksess.
- Interaksjons-orienterte: Motivert av interaksjon med medarbeidere.

16.4 Inndeling i grupper

Hvis det blir for mange medarbeidere på ett prosjekt, deler man inn i grupper, med maks 8-10 medlemmer. Dette gjør kommunikasjon enklere, og det er mulig å holde ett møte sammen. Det er også mange positive sider, ved å skape en gruppetilhørighet. De jobber bedre sammen, pusher hverandre fremover og jobber ofte mer målrettet.

17 Kapittel 29 (Ikke hoved) - Konfigurasjonsstyring (Subversion)

Siden systemkrav forandrer seg under hele utviklingsprosessen er det vikig med versjonskontroll.

Man kan også branche baselinen ut i flere brancher, f.eks. om man lager flere versjoner for flere operativsystemer (Solaris, UNIX, Windows, etc.).

Når man skal bruke versjonkontroll på ett prosjekt er det lurt å bruke en sterkt hierarkisk filstruktur. På større prosjekter har man også en **konfigurasjonsdatabase** der man lagrer relevant informasjon til hver versjon.

17.1 Versjoner

Til **versjonsidentifisering** kan man bruke *versjonnummerering*. Hver system release får ett hovedtall f.eks. 1.0, og mindre endringer øker desimaltallet, f.eks. 1.1. v2.0 trenger ikke nødvendigvis branche fra siste (1.1), men kan godt branche fra 1.0.

Det finnes CASE-tools som kan ta hånd om alle disse prosessene og endringhåndtering etc.

17.2 Versjonskontroll

Hvis flere jobber på samme kildefiler, kan det fort bli kluss med forskjellige versjoner av filene. Derfor bruker vi versjonskontrollsystem. En klar deling mellom disse systemene er for eksempel distribuerte systemer som for eksempel **git**, **Bazaar** eller **Subversion** (kan også brukes sentralisert) og sentraliserte systemer som **CVS** (Concurrent Versions System) og **Subversion**

17.2.1 Sentraliserte systemer

Subversion løser dette ved å låse tilgang til filen, hvis den allerede blir jobbet med. Man *sjekker inn* og *sjekker ut* filen. Det finnes også innsjekkingssystemer som kan sammenlignede endringene som har gjort i filene, og eventuelt gi tilbakemeldinger om det er gjort endringer på samme sted, ellers spleiser den endringene inn i den nye filen.

Fordelen med et slikt system er selvfølgelig at man har veldig god kontroll på koden (det finnes bare en kode), og man vil alltid kunne teste den nyeste koden

Ulempen er selvfølgelig at ved f.eks filkorrupsjon vil man få større problemer, med mindre man har en god og fullstendig backup. En annen ulempe er at man må jobbe online(ustabilt trådløst nettverk?), og kanskje eieren av prosjektet har sperret koden for ekstern tilkobling av sikkerhetsgrunner og man må alltid være på et sted når man koder. Overføring må også være kryptert som kan gi ytelsestap i en rekke tilfeller.

17.2.2 Distribuerte systemer

Alle som kobler seg til kildekoden for å redigere laster ned en fullstendig kopi av all kode og historie, man får da et stort nettverk av versjoner som utvikles i parallell.

Ulempen er at det kan være vanskelig å "lime" sammen filer etter omstendig redigering og sikkerheten for et firma som vil beskytte koden sin er redusert. Man må også laste ned all kildekode og historie til et prosjekt for å være en del av det. Dette kan ta sin tid med store prosjekter, men er i stor grad en engangskostnad.

Fordeler med distribuerte systermer er selvfølgelig at man har minst like mange backuper som kodere, man kan jobbe offline (fordel ved ustabile trådløse nett), og fra hvor som helst i verden, å redigere kode lokalt vil også være raskere og sikrere.