

inf1050-notater

Sjur Hernes og Eivind Lindbåten

31 May 2010

Contents

1	Kapittel 1 - Hva er programvare?	4
1.1	Introduksjon	4
1.2	Hva er programvare?	5
1.3	Hva er systemutvikling?	5
2	Kapittel 2 - Sosiotekniske systemer	6
2.1	Definisjon teknisk system	6
2.2	Definisjon Sosio-teknisk system	6
2.3	Om systemer	6
2.4	Systemutvikling	6
2.4.1	Vi har tre former for system-krav	6
2.4.2	Systemmodellering	7
2.4.3	System-evolsjon	7
2.4.4	System-dekomponering	7
2.4.5	Større systemer	7
2.5	Rammer	7
3	Kapittel 3 - Kritiske systemer	7
3.1	Sikkerhets-kritiske systemer:	7
3.2	Oppgave-kritiske systemer:	7
3.3	Business-kritiske systemer:	7
3.4	De tre hovedårsakene til feil	8
3.4.1	Faser i en livssyklus	8
4	Kapittel 4 - Systemutviklingsprosess	8
4.1	Utviklingsmodeller.	8
4.1.1	Fossefallsmodellen:	8
4.1.2	Evolutionary development:	9
4.1.3	Component based software engineering:	9
4.1.4	Process iteration:	9
4.1.5	Incremental delivery:	9
4.1.6	Process activities	10
4.1.7	Prøv-og-feil	10
4.1.8	TODO Oblig-metoden.	10

4.1.9	Prototyping	10
4.1.10	Evolusjonære modeller	10
4.1.11	Computer-Aided Software Engineering (CASE)	11
4.1.12	Smidige (agile) metoder	11
4.2	Vikige elementer i prosjektplanlegging:	11
4.2.1	EØS lov om anbud	12
5	Kapittel 5 - Prosjekthåndtering	12
5.1	Management activities:	12
5.2	Prosjektplanlegging:	12
5.3	Prosjektplan:	13
5.3.1	Milestones og leveringer:	13
5.3.2	Prosjekttidestimering og aktivitetsnett	13
5.3.3	Risk management	14
6	Kapittel 6 og 7 - Kravhåndtering, Programvare-krav og Kravspesifisering	15
6.1	Foranalyse	16
6.2	Interessenter(stakeholders) og kravinnnsamling	16
6.3	Krav-validering	17
6.3.1	hvorfor trenger vi veldefinerte krav	18
6.3.2	Krav til krav	18
6.4	Kravhåndtering	18
6.5	Funksjonelle og ikke-funksjonelle krav	18
6.5.1	Funksjonelle krav	19
6.5.2	Ikke-funksjonelle krav	19
6.6	Bruker-krav	20
6.7	System-krav	20
6.8	Interface specification	20
6.9	Programvare-krav-dokumentet	20
7	Kapittel 8 - System-modeller	21
7.1	Context models	21
7.2	Behaviour models	21
7.2.1	Use Case	21
7.3	Datamodeller	22
7.4	Objektmodeller	23
7.4.1	Domenemodeller	23
7.4.2	Sekvensdiagrammer	24
7.5	Delegert vs sentralisert kontrollsti	24
7.6	Structured methods	25
7.7	Metode for ansvarsdrevet OO - INF1050-metoden	25
8	Kapittel 13 - Applikasjons-arkitektur	26
8.1	Generiske systemer	26
8.2	Data-processing systems	26
8.3	Transaction-processing system	27

8.4	Event-processing system	27
8.5	Language-processing system	27
9	Kapittel 23 - Software-testing	27
9.1	Kvaliteter vi trenger i produktene:	27
9.2	System-testing	28
9.2.1	Release-testing:	28
9.2.2	Performance-testing:	28
9.2.3	Komponent-testing	28
9.2.4	Interface-testing:	28
9.2.5	Kravbasert testing:	28
9.2.6	Structural testing:	28
9.2.7	Partition testing:	29
9.2.8	Structural testing(white-box):	29
9.2.9	Path testing:	29
9.3	Test-automatisering	29
9.3.1	Enhets-testing	29
9.3.2	Integrasjons-testing	29
9.3.3	System-testing	29
9.3.4	Black box testing	29
9.3.5	White box testing	30
10	Kapittel 26 - Software cost estimation	30
10.1	Software productivity	30
10.2	Estimation techniques	31
10.3	Algorithmic cost modeling	31
10.4	Product duration and staffing	31
11	PS2000	32
11.1	Bruksområde:	32
11.2	prisformer	32
11.2.1	Fastpris	32
11.2.2	Løpende timer	33
11.2.3	Målpris	33
11.3	PS2000 vs andre kontrakter	33
11.3.1	Andre kontrakter	34
11.4	Oppbygging av PS2000	34
12	Kapittel 11 (Ikke hoved) - Arkitekturdesign	34
13	Kapittel 12 (Ikke hoved) - Distributed system architecture	35
14	Kapittel 16 (Ikke hoved) - Brukergrensesnitt-design	36
14.1	Et godt grensesnitt er viktig, for god programvare.	36
14.2	Prosessen ved GUI-utvikling:	36

15 Kapittel 25 (Ikke hoved) - Managing people	36
15.1 valg av arbeidere	37
15.2 Motivering av arbeidere	37
15.3 Generalisering rundt mennesketyper	37
15.4 Inndeling i grupper	37
16 Kapittel 29 (Ikke hoved) - Konfigurasjonsstyring (Subversion)	37
16.1 Versjoner	37
16.2 Versjonskontroll	38
16.2.1 Sentraliserte systemer	38
16.2.2 Distribuerte systemer	38
17 Jus, etikk og personvern	38
17.1 sjekkliste for kontroll av POL	38
17.1.1 Sjekkliste, trinn 1	38
17.1.2 Sjekkliste, trinn 2	39
17.1.3 Sjekkliste, trinn 3	39
17.1.4 Sjekkliste, trinn 4	39
17.1.5 Sjekkliste, trinn 5	40
17.1.6 Sjekkliste, trinn 6	40
17.1.7 Sjekkliste, trinn 7	40
17.1.8 Sjekkliste, trinn 8	41
17.2 Oppsummering av lovens ni kapitler	41
17.2.1 Kapittel I	41
17.2.2 Kapittel II	41
17.2.3 Kapittel III	41
17.2.4 Kapittel IV	41
17.2.5 Kapittel V	41
17.2.6 Kapittel VI	41
17.2.7 Kapittel VII	42
17.2.8 Kapittel VIII	42
17.2.9 Kapittel IX	42
17.3 Virkeområde og hovedregler	42
18 Databaser	42
18.1 Klient-server databaser	42
18.2 integrerte databaser	43
19 Appendix - Ordliste	43

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 uoversiktlig. Prisantydning kan være vanskelig og prosjekter slår ofte feil og overskrider budsjettet. Software Engineering 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.

Software engineering er et tverrfaglig foretagende!

Baserer seg på de systematiske ingeniørprinsippene:

- Planlegging og forutsigbarhet
- Oppdeling og strukturering av problemer i mindre komplekse bestanddeler
- Modularitet og gjenbruk
- Abstraksjon og modellering
- Systematisk kvalitetssikring

En enkel oppdeling og strukturering av prosessen kan se slik ut:

- Tid (faser):

- PS2000-kontraktstandarden: Behovsfase, Løsningsbeskrivelse, Iterativ konstruksjonsfase, Godkjenningsfase.
- Timeboxing/tidsavgrensning.
- Oppgaver (aktiviteter og tilhørende leveranser):
 - Analyse, design, programmering, testing,...
- Tekniske aspekter:
 - Kvalitetsaspekter, Funksjonalitet, Moduler, Komponenter.
- Modeller på forskjellige abstraksjonsnivåer
 - Kravspesifikasjoner versus Objektorientert analyse versus Detaljert design versus Kode.

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. System-egenskaper: De ikke-funksjonelle kravene.
3. Hva systemet ikke må gjøre.

2.4.2 Systemmodellering

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 lettere og billigere).

2.4.3 System-evolusjon

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.

3.4.1 Faser i en livssyklus

1. Idefase om et system – foretningsanalyse (lønner det seg)
2. Kravinnsamling og kravanalyse (hva skal systemet gjøre?)
3. Design (hvordan skal det konstrueres?)
4. Programmering (konstruksjon)
5. Test (ble det riktig?)
6. Installasjon, integrasjon, driftsetting
7. Vedlikehold (feilretting og videreutvikling)

4 Kapittel 4 - Systemutviklingsprosess

Hvordan programvare skal produseres. Organiseringen rundt det å skrive kode etc. 76% av alle prosjekter overskrider budsjett. 19% bruker mindre. Økt produktivitet og kvalitet.

4.1 Utviklingsmodeller.

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.

4.1.1 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 maintenance: Systemet installeres, 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.

4.1.2 Evolutionary development:

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

1. Exploratory development: Videreutvikler 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.

4.1.3 Component based software engineering:

Tar sikte på å gjenbruke komponenter.

1. Component analysis: Ser på krav, og leter etter komponenter. Passer ikke alltid nødvendigvis.
2. Requirements modifications: Ser på kravene igjen i forhold til komponenter. Endrer på kravene hvis det lar seg gjøre. Hvis ikke leter 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 alltid gjøres i forhold til kravene.

4.1.4 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.

4.1.5 Incremental delivery:

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

1. Fordeler: Kan få ett fungerende system tidlig. Mest kritisk utvikles først. Kan ut fra erfaring forme nye krav til inkrement. Lavere risiko for total prosjekt-kolaps.
2. Spiral development: Utviklingsfasen er illustrert i en spiral 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.

4.1.6 Process activities

- 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 videre 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: Kontrollerer 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.

4.1.7 Prøv-og-feil

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

4.1.8 TODO Oblig-metoden.

Funker dårlig på større systemer. Vanskelig for samarbeid. Utdype?

4.1.9 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 evolusjonær-protyp. (bruker siste utkast).

4.1.10 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
- RUP
Arkitektursentrert, objektorienterte utviklingsprinsipper, UML-modelering er sentralt. ModellDrevet utvikling
 1. Hybrid prosessmodell. Basert på UML.
 2. Inception: Buisness-analyse og eventuelt klarsignal
 3. Elaboration:
 - Forstå problem-området
 - få oversikt over rammeverk-arkitekturen
 - risk-analyse.
 - Ved slutt har man use case UML.
 4. Construction: Programmet utvikles og dokumenteres
 5. Transition: Fører det over til brukere.

4.1.11 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. Bruker-dialoger(skjermbilder) for tilbakemeldinger fra interessenter.

4.1.12 Smidige (agile) metoder

Mindre formalisme og krav. Oppfordrer til direkte muntlig kommunikasjon. Eksempler er XP (Extreme programming) og Scrum. Individuer 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-programmering (to programmerere 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 tilhørende teknikker. Få krav til spesifisering og planlegging. Raske iterasjoner (1-3 uker). Ineressentene 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.

4.2 Viktige elementer i prosjektplanlegging:

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

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

Styringsgruppen: De økonomiske interessene. 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 prioriterte oppgaver.

Gjør en oppfølging av fremdrift. Alle rapporterer tidsbruk. Planer oppdateres gjenvlig.

Ved korte iterasjoner har man råd til å feile.

4.2.1 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 riskhåndtering.
- 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. Oppdateres gjerne 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.

HUSK:

- Estimering er i beste fall kvalifisert gjetning
- Krav og forutsetninger endres underveis
- Historisk sett har vi uansett lav sansynlighet for å treffe
- Nettverksdiagram(aktivitetsdiagram)

Beskriver mulige gjennomføringer av aktivitetene Nettverksdiagrammet bestemmes ved:

- Hver aktivitet defineres som en enhet i diagrammet
- mulige rekkefølger angis som piler mellom to aktiviteter
- starttidspunkt og ferdigstilling bestemmer diagrammets mulige veier

Nettverksdiagrammet gjør at vi enklere kan se om vi er i rute på prosjektet eller ikke.

- Kritisk vei
den veien i nettverksdiagrammet som bruker lengst tid, som dermed er korteste tiden det er mulig å bli ferdig med prosjektet på
- Slack
den veien i nettverksdiagrammet man kan ha størst forsinkelser på uten at det den veien blir den kritiske. Slack kan beskrives som der marginene er, eller der man må vente når man er ferdig, hvis man regner med at alle aktiviteter starter med en gang de kan og går etter planen.
 - * **Positiv slakk** betyr det er tid å gå på, i forhold til sluttdato.
 - * **Negativ slakk** betyr at aktiviteten medvirker til at prosjektet ikke blir ferdig tidnok.

- 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	Faktureringsystem	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	

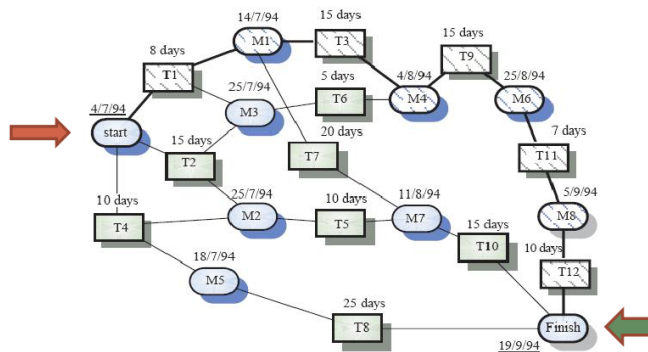


Figure 1: Eksempel på nettdiagram til en aktivitetstabell

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 samarbeid og god kommunikasjon fra nøkelpersoner og videre til deres kollegaer, slik at plassen kan fylles. Fjerne urelevante arbeidsoppgaver fra personell som er engasjert i prosjektet
- Ansvarlig begge

6 Kapittel 6 og 7 - 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 kostnadmessig 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 organisasjonen 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 kravinnnsamling

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 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, lignende 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. Kravinnnsamling 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 kravspesifikasjon: 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 forståelige?
- 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 alltid 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 selvmot-sigende krav.

sier hvilke tjenester systemet skal utføre, hvordan det skal oppføre seg i spesielle situasjoner, 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 rapporter 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 kryptert 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 kunnskap. 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 unngå. 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

Notasjon som støtter opp under modellbasert systemtvikling.

- Godt utgangspunkt for dokumentasjon.

Kan brukes til datamodeliering, arbeidsflyt-modelering eller objektmodelering.

En systemmodell representerer systemet på en mer overordnet, abstrakt og helhetlig måte. Kan ses på den tekniske motvekten til Krav. Da krav skal bli forstått av også ikke tekniske mennesker, er det naturlig for utvikleren å lage en mer teknisk modell basert på kravene. Man baserer ofte modellene sine på UML-standardene

7.1 Context models

Her ser vi på systemet og konteksten i forhold til andre systemer, og hvilke undersystemer som har kontakt med verandere, og i hvilken rekkefølge? Hvilket system er databasen koblet til, hvor kommer lenken mot internett hvis noen, er det et sikkerhetssystem mellom internett og andre systemer?

7.2 Behaviour models

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

Dette kan ofte modeleres med use-case diagrammer, direkte spesifisert på kravene fra interessentene

7.2.1 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.

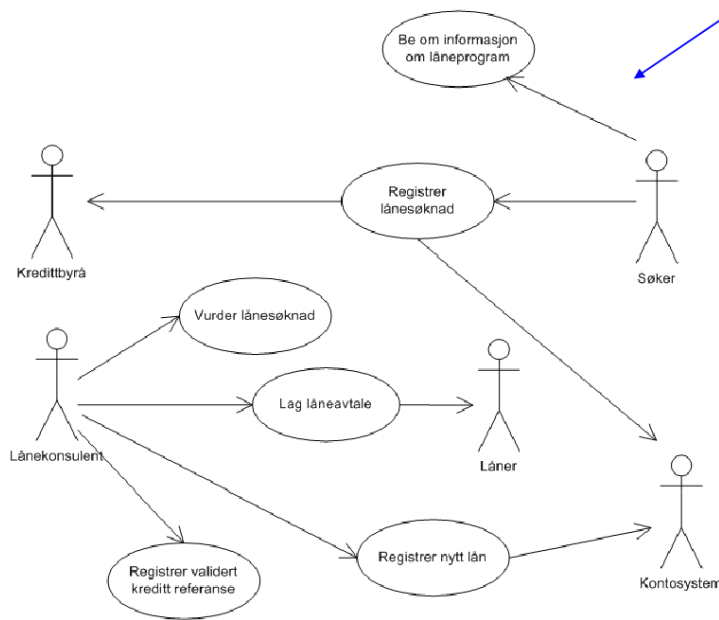
Tekstlige krav som utgangspunkt for å identifisere aktører og use cases

#	Krav	Aktør	Use case
1	Systemet skal på oppfordring vise informasjon om alle tilgjengelige låneprogrammer.	Søker	Be om informasjon om låneprogram
2	En lånesøker skal kunne søke om lån online ved å registrere all nødvendig informasjon i et skjema. Informasjonen sjekkes automatisk og gis en kreditt score basert på informasjon fra kredittbyrå samt kundens forhold til banken.	Søker, Kredittbyrå, Kontosystem	Registrer lånesøknad
3	Søkeren skal til enhver tid kunne se status for lånesøknaden.	Søker	Se status
4	Registrerte lånesøknader skal vurderes for godkjenning. Hvis søknaden godkjennes skal dette markeres i systemet slik at et nytt lån kan opprettes.	Lånekonsulent	Vurder lånesøknad
5	Lånekonsulenten skal kunne registrere kredittinformasjon for lånekunder som valideres manuelt som for eksempel eksterne bankkonti og inntektskilder.	Lånekonsulent	Registrer validert kredittreferanse
6	Låneavtaler skal kunne opprettes og sendes til kunden for elektronisk godkjenning via internett.	Lånekonsulent, Låner	Lag låneavtale
7	Nye lån skal registreres i bankens system for kontooversikter.	Lånekonsulent Kontosystem	Registrer nytt lån

Figure 2: Eksempel på kravtabell til et casediagram

- extend
Utvider funksjonalitet, f.eks. i alteriantiv flyt.
- include
F.eks. hvis flere use case utfører samme prosedyre, kan vi lage use case av det, og include.
- 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 kredittrapport.
 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 allokere lånesøknaden til en lånekonsulent for videre behandling

Use case diagram



Assosiasjon

- Viser kommunikasjon mellom en aktør og et use case.
- Pil kan brukes til å vise hvor kommunikasjonen er startet (men NB! Må ikke forveksles med retningen på "dataflyt")

Figure 3: Eksempel på casediagram fra kravtabellen

- 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

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).

Herunder kommer CRC-kort som beskriver hva objektene må vite om seg selv, og hvilke andre objekter som den har relasjoner til, hvordan skal den få tak i informasjon, hva slags objekt det er - kant, kontroll, eller foretningsobjekt

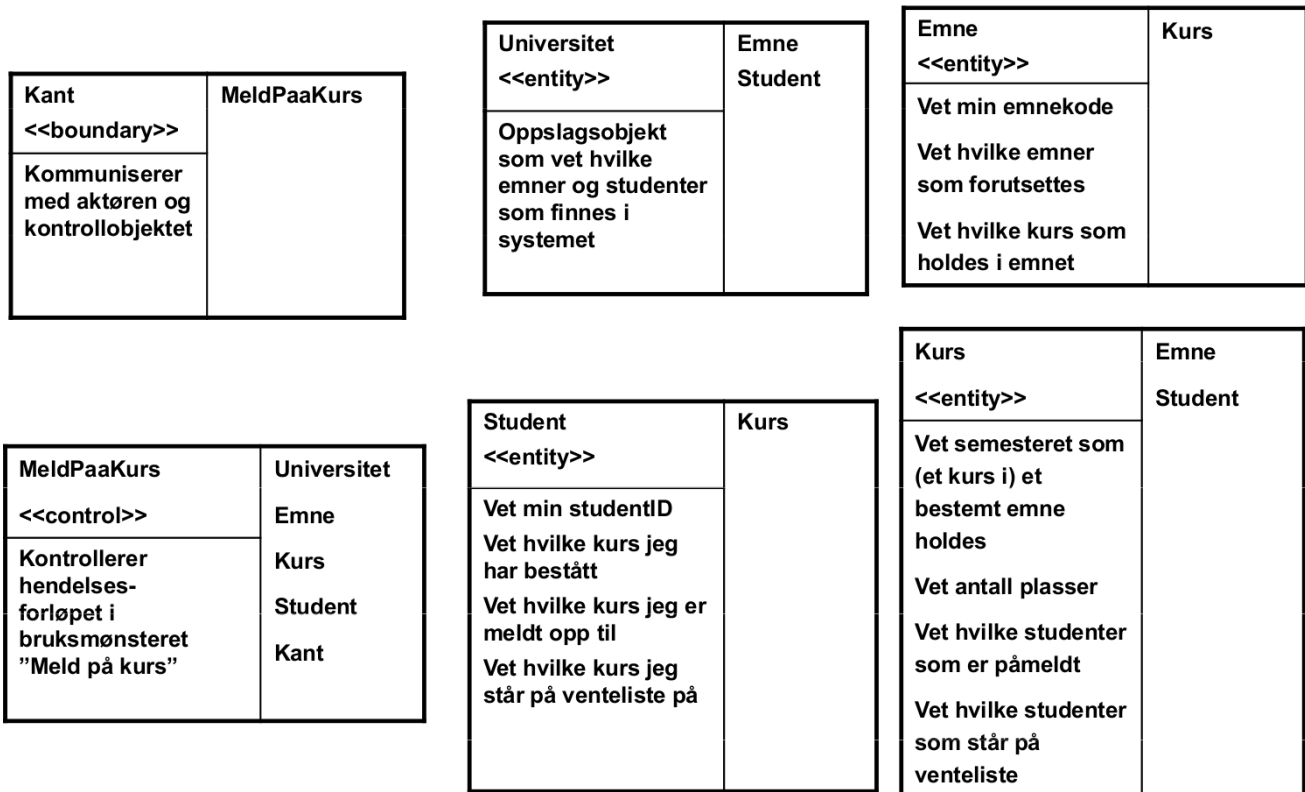


Figure 4: Eksempel på CRC-kort

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 attributter og egenskaper.

7.4.1 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.

- Forretningsobjekter
De har evig liv. Lagrer i database.

- Kontrollobjekter
Kontrollerer handlingsforløpet.
- Kantobjekter
Kommuniserer med brukere.

7.4.2 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

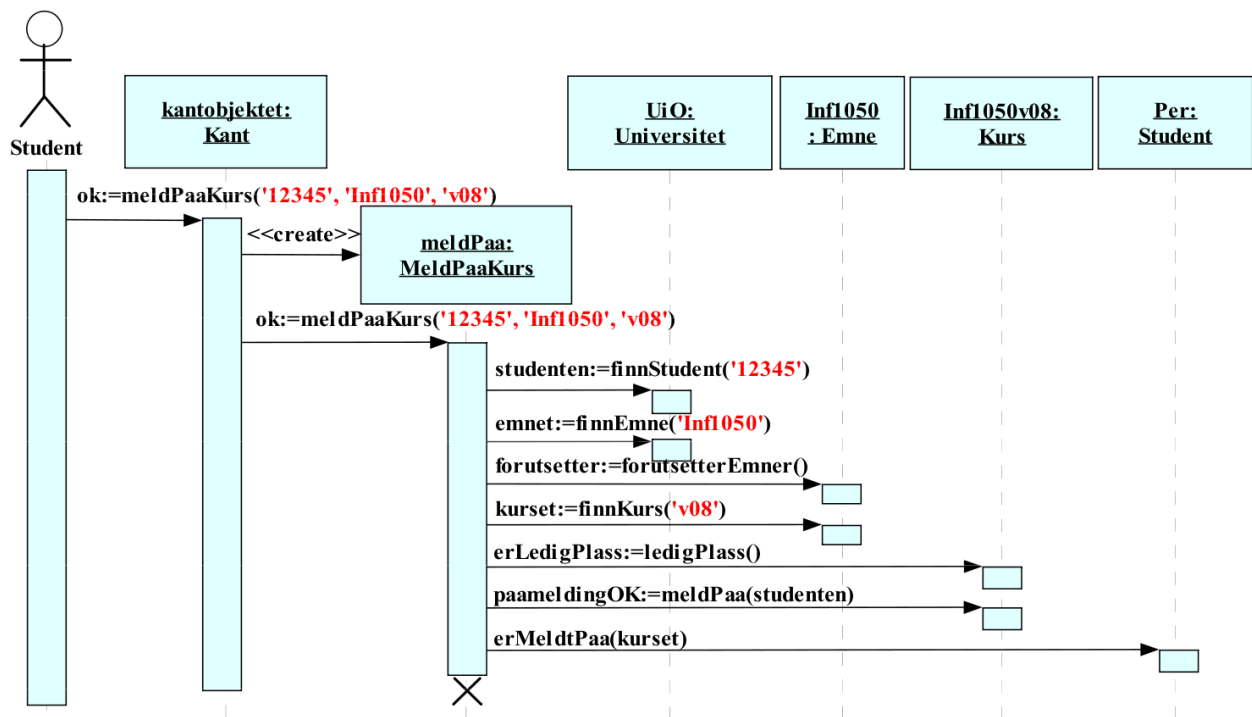


Figure 5: Eksempel på sekvensdiagram

7.5 Delegert vs sentralisert kontrollsti

Design av ansvar og samarbeid mellom klassene i et system kan gjøres på flere måter, og to av de vanligste fremgangsmåtene for dette er delegert kontrollsti og sentralisert kontrollstil. Delegert kontroll vil si at ansvar er fordelt utover flere klasser i applikasjonen. Sentralisert kontroll innebærer at applikasjonen har noen få, store kontrollklasser som koordinerer et sett med relativt enkle klasser.

7.6 Structured methods

Karakteristikk:

- Brukes til kravspesifisering og systemdesign.
- Dele prosjektet i veldefinerte aktiviteter.
- Bruke diagrammodellering 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 datamodellering, automatisk generering av kildekode, og brukergrensesnitt-manipulering.

7.7 Metode for ansvarsdrevet OO - INF1050-metoden

Analyse av krav

- Identifiser krav
- lag høynivå bruksmønsterdiagram
- Spesifiser hvert bruksmønster tekstlig med hovedflyt og alternativ flyt

Objektdesign, for hvert bruksmønster:

- Identifiser objekter, fordel ansvar (CRC)
- Lag sekvensdiagram
- Lag klassediagram
- Lag klassediagram på systemnivå

8 Kapittel 13 - Applikasjons-arkitektur

8.1 Generiske systemer

Som regel vil to systemer av samme type ha veldig mange felles attributter, det finnes i dag mange generiske systemer for en rekke systemtyper **drupal**, **wordpress**, **SAP** og flere er systemer som man utvider med egen funksjonalitet/konfigurasjon. Som en applikasjonsarkitekt kan du bruke disse generiske systemene på en rekke måter:

1. Som en grunnplattform å bygge videre på
2. Som en sjekklister for å passe på at man har med alt man trenger
3. Som en måte å organisere arbeid på (man ser på modeller og etterligner etter eget behov)
4. Evaluere komponenter for gjennbruk
5. Som en måte å finne krav på/sammenligning

8.2 Data-processing systems

Uten brukere som bruker systemet men heller info den skriver ut

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.

For eksempel folding@home og lignende programmer som rett og slett knuser tall og skriver tilbake resultater for forskning

8.3 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 krupt eller inkonsistent.

De fleste websider er slik og det kan her være en fordel å skille nøkkelkomponentene i tre lag Modell som integrerer databasen, kontroller som spør databasen og returnerer View denne arkitekturen er mest kjent som MVC

Tegn opp:

- Web browser <-> Web Server <-> Application Server <-> Database Server

8.4 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.

Det eneste disse trenger å tenke på er som regel hvordan reagere på det som skjer da disse systemene som regel er små og lokalt på en datamaskin trenger vi ikke tenke så mye på flere systemer, men heller hvordan metodene fungerer.

8.5 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).

Man kan også argumentere at alle “what you see is what you get”-teksteditorer som f.eks word er et slags language-processing-system da den mottar klartekst og lagrer det som xml eller en annen dokument-standard.

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 komponenter, 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 alltid 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 Performance-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 enkeltkomponenter. Hovedsaklig utvikler som gjør den jobben. Enten individuelle metoder eller funksjoner i ett objekt, eller objekt-klasser med flere attributter og tilhørende metoder.

9.2.4 Interface-testing:

Tester interface-kommunikasjon 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 positivt 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 comparator som sammenligner tester, Dynamic analyser som sjekker hvor ofte de forskjellige statementene blir kjørt. Og simulering i forskjellige varianter. F.eks. bruker-interaksjon.

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

Man kjører flere tester, siden de tester forskjellige aspekter ved systemet.

9.3.1 Enhets-testing

Vanligvis en test av spesifikke seksjoner av koden, for eksempel en funksjon. I et objektorientert miljø, vil dette typisk være på klassenivå. Testene skrives ofte av utviklerene selv, og er en form for white-box-testing.

9.3.2 Integrasjons-testing

Integrasjons-testing, tester interaksjonen (interfacet) mellom forskjellige komponenter. Det beste er å innføre komponentene på en iterativ måte, og teste fortløpende.

9.3.3 System-testing

Tester systemet i sin helhet, i forhold til kravspesifikasjoner. Det er en form for black-box-testing, siden det ikke skal kreve noe kunnskap om "innmaten" i systemet.

For eksempel:

- Testing av GUI.
- Testing av sikkerhet, hastighet, veldlikeholdbarhet og andre ikke-funksjonelle krav.

9.3.4 Black box testing

- Tester systemet, sett fra utsiden, uten kunnskap om koden/system.

takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid input and determines the correct output. There is no knowledge of the test object's internal structure.

- Vet ikke hvor store deler av koden som faktisk testes.
- Kan kontrollere at programmet følger kravspesifikasjoner.
- Skalerer godt.
- Man vet ikke hvor feilen oppstår, når man får feil output.

9.3.5 White box testing

- Tester laget av folk med store kunnskaper om koden, for å teste alle mulige “veier” i koden etc.

uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate outputs. In electrical hardware testing, every node in a circuit may be probed and measured; an example is in-circuit testing (ICT).

Eksempler på hva som testes:

- Control flow testing
- Data flow testing
- Branch testing
- Path testing

Fordeler ulemper

- Man er sikker på at alle “kriker” av koden blir testet.
- Skalerer ikke bra. Antall muligheter øker eksponentielt.
- Oppdager ikke manglende funksjonalitet.

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. LinesOfCode / 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 Algorithmic cost modeling

Algorithmic 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 konkurrere 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 PS2000

Utviklet av Dataforeningen

11.1 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.

11.2 prisformer

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 oversiktlig.

11.2.1 Fastpris

knyttet til avtalt omfang.

- Kundeperspektiv
 - Forutsigbart, men risikerer å få et avkortet produkt dersom leverandøren ikke vil jobbe utover fastpriskostnadene.
- Leverandørperspektiv
 - Også forutsigbart, men risikerer å jobbe gratis etter at fastpriskostnadene er brukt

11.2.2 Løpende timer

fakurerer timer og annet.

- Kundeperspektiv:
 - Lite forutsigbart og risiko for store budsjettoverskridelser. vil dog kunne få levert et fullstendig produkt.
- Leverandørperspektiv:
 - Også lite forutsigbart, men får dekket faktiske utgifter

11.2.3 Målpris

basert på estimater og risikovurderinger. Justeres med incentiver og saksjoner.

- Kundeperspektiv og leverandørperspektiv
 - Forutsigbart og eksplisitt håndtering av risiko og ansvarsfordeling. Rom for erfaringsbaserte vurderinger. Støtter samarbeid mellom kunde og leverandør. Krever ekstraarbeid for å vurdere risiko og usikkerhet. Krever at begge parter forstår incentiv-modeller.
- Bruksområder
 - Systemutvikling basert på relativt godt spesifisert behovsanalyse
 - Godt nok grunnlag for realistiske estimater
 - Erkjennelse av at krav og prioriteringer kan endre seg underveis
- Fordeler og ulemper
 - Relativt forutsigbar mht. kostnader
 - Klare leveranseavtaler mht. omfang og kalendertid
 - Rom for erfaringsbaserte justeringer i iterasjoner
 - Støtter integrert samarbeidsmodell
 - Felles incentiver om å levere kvalitet til avtalt tid og kostnad

11.3 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 parter interesser er ivarettatt 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.

11.3.1 Andre kontrakter

Begge de to andre avtalene som er presentert på forelesninger er basert på å ivareta en interessant sin interesse, og ikke begge, PS2000 er unik der og det er derfor vi nesten utelukkende har om PS2000

- IKT Norge:
Laget for å ivareta leverandørs interesser
 - Standardavtale om systemutvikling
 - Standardavtale om systemutviklingsprosjekt
- Statens standardavtaler (DIFI):
Laget for å ivareta kunders interesser
 - Programutviklingsavtalen

11.4 Oppbygging av PS2000

Kontrakten er delt inn i

Del1: Kontraktdokument Del2: Generelle kontraktsbestemmelser Del3: Kontraktsbilag

12 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 imellom.

Skisse en grunnleggende struktur av systemet, og hovedkomponentene 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-orientert der man har input-data som går gjennom forskjellige funksjoner (subsystemer) og får til slutt en output.

13 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 feiltoleranse, annen server kan raskt slå inn i steden hvis man har flere.

Ulempene er

- Kompleksitet.
- Sikkerhet pga 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 forespørsler 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.
 - klienten vil bare mota og sende informasjon, bare ansvarlig for å presentere informasjonen

- Serveren gjør all prosessering og datamanagement

2. Tykk-klient.

- Klienten har applikasjonslogikk i tillegg til brukerinteraksjon
- Serveren har bare ansvar for datamanagement (serveren brukes som database)

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.

14 Kapittel 16 (Ikke hoved) - Brukergrensesnitt-design

14.1 Et godt grensesnitt er viktig, for god programvare.

Noen ting å huske på er blant annet,

- Mennesker har dårlig hukommelse. 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 unngå å være negative, bør være lett forståelige og tilby videre hjelp og være smarte.

14.2 Prosessen ved GUI-utvikling:

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

15 Kapittel 25 (Ikke hoved) - Managing people

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

15.1 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.).

15.2 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.

15.3 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.

15.4 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.

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

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

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

Når man skal bruke versjonskontroll 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.

16.1 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 branchen fra siste (1.1), men kan godt branchen fra 1.0.

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

16.2 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**.

16.2.1 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 innsjekkingsystemer 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. filkorupsjon 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.

16.2.2 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 systemer 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.

17 Jus, etikk og personvern

Personopplysningsloven er alltid relevant ved informasjonssystemer som skal behandle personopplysninger. Loven er basert på et EU-direktiv. Under følger en sjekkliste for å kontrollere om du følger POL riktig. (Hentet direkte fra forelesningsfoil).

17.1 sjekkliste for kontroll av POL

17.1.1 Sjekkliste, trinn 1

Gjelder loven for mitt informasjonssystem?

- Hvis jeg er etablert i Norge (hovedregel)
- Hvis PO blir behandlet elektronisk

- Behandling = alt du kan gjøre med opplysninger/data
- Helt eller delvis elektronisk behandling
- Uansett tekst, bilde, lyd mv
- Spesielle regler gjelder for fjernsynsovervåking

17.1.2 Sjekkliste, trinn 2

Har jeg lovlig adgang til de opplysningene jeg ønsker å behandle (eller kan jeg skaffe slik adgang?) Mulige rettslige grunnlag

- Lovhjemmel
- Samtykke
 - Frivillig
 - Informert (jf § 19)
 - Uttrykkelig
- Nødvendig (slik det er angitt i pol §§ 8 og 9)

17.1.3 Sjekkliste, trinn 3

Hva skal/kan jeg benytte PO til?

- All behandling av PO må skje for ett eller flere bestemte formål
- Formålene må være angitt før behandling begynner
- Formålet spiller en viktig rolle bl.a i forhold til krav til opplysningskvalitet, info.sikkerhet og lagringstid
- Formålet kan endres, men det er begrensninger

17.1.4 Sjekkliste, trinn 4

Har opplysningene den nødvendige *kvalitet*?

- Kvalitetskravene skal alltid bedømmes ut i fra formålet
- Kvalitetskrav etter loven:
 - tilstrekkelige
 - relevante
 - korrekte
 - oppdaterte
- Det skal alltid vurderes om det er behov for tiltak for å fremme opplysningskvalitete

17.1.5 Sjekkliste, trinn 5

Er personene tilstrekkelig sikkert identifiserte?

- Begrensninger i adgang til å benytte “entydige identifikasjonsmidler”:
 - fødselsnummer
 - biometriskeidentifikasjonsmetoder
- Må være
 - saklig behov for og
 - nødvendig å benytte slike identifikasjonsmidler

Datatilsynet kan gi pålegg om bruk av entydige identifikasjonsmidler

17.1.6 Sjekkliste, trinn 6

Hva er tilfredsstillende sikkerhetstiltak?

- Skal sikre
 - konfidensialitet
 - integritet
 - tilgjengelighet
- Skal vurdere behov for å iverksette tiltak
- Tiltakene må være
 - planlagte
 - systematiske
 - dokumenterte
- Tiltakene kan være av et hvilket som helst slag

17.1.7 Sjekkliste, trinn 7

Hva må jeg gjøre for å være sikker på at loven, forskrifter og eventuelle enkeltvedtak blir etterlevet?

- Internkontroll må gjennomføres for å sikre etterlevelse av krav i:
 - lov og forskrift
 - konsesjoner og andre enkeltvedtak
- Tiltakene må være
 - planlagte
 - systematiske
 - dokumenterte

17.1.8 Sjekkliste, trinn 8

Kan jeg starte behandlingen?

- Melding skal sendes minst 30 dager før hvis meldepliktig
- Konsesjon må søkes hvis sensitive PO
 - Sensitive opplysningstyper er definert i loven
 - Kan ikke starte behandlingen før konsesjon er gitt og vilkår som stilles i konsesjonen kan etterleves

17.2 Oppsummering av lovens ni kapitler

17.2.1 Kapittel I

Kapittel I. Lovens formål og virkeområde inneholder definisjon av grunnbegreper som brukes i loven, pluss bestemmelser om hva loven gjelder for og hvor den gjelder. Dette er nesten alltid nødvendig å ha oversikt over.

17.2.2 Kapittel II

Alminnelige regler for behandling av personopplysninger er også viktig fordi i dette kapitlet fastsettes de viktigste vilkårene for at det over hode skal være lov for skolen og andre å behandle opplysninger om elever og lærere. Dette kapitlet inneholder mange viktige plikter for den som er ansvarlig for at loven etterleves, for eksempel den øverste skoleledelsen i en kommune (“behandlingsansvarlig”).

17.2.3 Kapittel III

Informasjon om behandling av personopplysninger Personer som det er registrert opplysninger om (elever, lærere mv) har rettigheter etter loven. Kapittel III inneholder bestemmelser om de viktigste rettighetene.

17.2.4 Kapittel IV

Enkelte andre rettigheter finnes her. Andre rettigheter for den registrerte, bl.a. rett og plikt til å slette og rette opplysninger.

17.2.5 Kapittel V

Overføring av personopplysninger til utlandet får i praksis kun betydning i forhold til overføring av personopplysninger til og fra land utenfor EØS-området. Bestemmelsene er imidlertid ofte vanskelige å anvende fordi de er basert på situasjonen for mer enn ti år siden og ikke er godt tilpasset moderne bruk av Internett.

17.2.6 Kapittel VI

Melde- og konsesjonsplikt Meldeplikt til Datatilsynet gjelder ofte, mens konsesjonsplikt primært gjelder behandling av personopplysninger som loven definerer som sensitive.

17.2.7 Kapittel VII

Fjernsynsovervåking gjelder enhver bruk av kamerateknologi, eventuelt med lyd, men ikke når kameraet brukes på “vanlige”, manuelle måter (f.eks. fotografering på klassetur).

17.2.8 Kapittel VIII

Tilsyn og sanksjoner har bestemmelser som gir Datatilsynet og Personvernemnda myndighet til å bestemme. Særlig viktig når det er konflikt og uenighet om hva som er lov. Kapitlet har også bestemmelser om straff og erstatning ved brudd på lovens bestemmelser, men dette er kun aktuelt i sjeldne tilfelle.

17.2.9 Kapittel IX

Ikraftsetting. Overgangsregler. Endringer i andre lover har i dag neppe betydning.

17.3 Virkeområde og hovedregler

Loven stiller noen grunnkrav til all behandling av personopplysninger: Den behandlingsansvarlige må ha rettslig grunnlag for behandlingen, dvs. enten samtykke fra den registrerte, hjemmel i lov eller behandlingen må være nødvendig av hensyn til et av formålene som er nevnt i § 8.

- Opplysningene skal ha tilfredsstillende kvalitet
- Opplysningene skal ikke brukes senere til formål som er uforenlig med det opprinnelige formålet med innsamlingen, uten at den registrerte samtykker
- Opplysningene skal ikke lagres lengre enn det som er nødvendige for behandlingen

For sensitive personopplysninger er det enkelte tilleggskrav: behandlingen må som regel ha konsesjon (tillatelse) fra Datatilsynet og behandlingen må oppfylle de skjerpede krav til rettslig grunnlag i § 9. Hvilke opplysninger som skal regnes som sensitive, er uttømmende definert i § 2 nr 8. Helseopplysninger og opplysninger om politisk oppfatning er eksempler på sensitive opplysninger.

18 Databaser

I en relasjonsdatabase har vi en eller flere tabeller, kalt relasjoner. Hver rad må kunne entydige identifiseres av f.eks. ett ID-nummer. Hver kolonne har en attributt. Cellene inneholder verdiene til attributtene. Tabellene kan kobles sammen med primary og foreign keys.

18.1 Klient-server databaser

Mye brukt språk mot databaser er SQL, med databaser som **mySQL**, **PostgreSQL** men også **ORACLE** er en stor distributør innen databaser. De baserer seg på at programmet som bruker databasen logger seg inn i databasen med brukernavn/passord

18.2 integrerte databaser

er ofte sett på som en alternativ måte til å bruke flat filstruktur for et program vil ofte ikke ha like stor concurrency som Klient-serverdatabaser, men fungerer mer som en integrert lagringsløsning for programmet som minimerer datakorupsjon og lignende **SQLite** er en av de større navnene innen dette, spesielt til mobiltelefoner o.l.

19 Appendix - Ordliste

Under følger noen korte forklaringer på ord og begreper og av og til dets engelske oversettelse.

Norsk	Engelsk	Forklaring
Domene-kunnskap		Kunnskap innenfor et bestemt område, f.eks. innenfor ett bestemt prosjekt eller
Sosio-teknisk system		System der også mennesker, med kunnskap om systemet, inngår i prosessen.
Kritiske systemer		Hvor feil i systemet vil være svært kritisk. F.eks. flystyring, kontrolltårn etc.
Domenemodell		Modell (ofte UML) over den delen av virkeligheten vi skal lage system for.
Ikke-Funksjonelle krav		legger føringer på oppførselen i form av krav til ytelse, sikkerhet, brukervennlig
Funksjonelle krav		beskriver oppførselen (funksjonene) til systemet
Milepæl	Milestone	Beskriver konkret delmål som skal oppnås