Forelesning 11: Arkitektur og brukergrensesnitt

Introduksjon

Forelesningen om arkitektur og brukergrensesnitt beskriver historien om utvikling av et virkelig system som er i produksjon den dag i dag. Disse notatene omhandler den teori som vi kommer inn på i løpet av forelesningen er ikke en direkte beskrivelse av det vi går i gjennom i forelesningen. Det er to hoveddeler, det ene er om arkitektur og den andre delen om brukergrensesnitt.

Arkitektur

Begrepet arkitektur brukes i dag om mange nivåer innen IKT og er nærmest blitt et moteord i bransjen. I dette kurset bruker vi arkitekturbegrepet til å beskrive struktur i et program i større skala. En passende definisjon av ”arkitektur” er: «Et systems komponenter og deres relasjon til hverandre.» Vi tenker da på større komponenter som moduler, prosesser eller pakker i et UML diagram. Hver enkel klasse og dens attributter er for detaljert til at vi ville beskrive det som arkitektur. Det er som om å beskrive arkitekturen til en bygning. Vi vil beskrive størrelse, stil og hovedroms utfordring, ikke dykke ned i detaljer som dørhåndtak eller låsemekanismen på dodøren.

En god arkitektur er en forutsetning for å lykkes med et hvert system av noe størrelse. Vi trenger en god oversikt over hvordan systemet henger sammen for å bli i stand til å utvikle et system. Ingen er i stand til å holde oversikt over et større system uten at arkitekturen er vel-definert og dokumentert. En god arkitektur hjelper også den enkelte utvikler i programmering siden hovedtrekkene i hvordan systemet skal bygges er allerede bestemt og vi trenger ikke sitte og fundere over de samme spørsmålene gang på gang. Videre må arkitekturen ta høyde for de viktigste ikke-funksjonelle kravene som ytelse, sikkerhet, pålitelighet etc.

Siden arkitekturen er selve grunnsteinene i systemet må den også bygges tidlig i en systemfase. Det nytter ikke å starte å tenke på arkitektur når du er ferdig. Da er det for sent.

For å komme frem til en god arkitektur må vi som alltid gå tilbake til forretningskrav. Det spesielle med arkitekturutvikling er at det ofte er de ikke-funksjonelle kravene som betyr mest for valg av arkitektur. Det er også viktig å prioritere mellom kravene siden alle ikke kan oppnås i samme grad i et enkelt system. Alle kunne sikkert ønske seg et sikkert system, som er lynraskt, kjører på hvilket som helst skrap av billig hardvare, koster ingenting i drift, utvikles på en uke og oppfyller alle tenkelige funksjonelle krav. Den beklagelige sannhet er at vi kan ikke oppnå alt og må ta noen valg.

En praktisk fremgangsmåte for arkitekturutvikling er å fokusere på arkitektur i de tidlige iterasjonene, leke CRC kort med komponenter akkurat som med klasser. Så lager man arkitekturdiagrammer med hovedkomponentene inntegnet. Deretter simulerer man på en tavle de mest sentrale use cases med arkitekturen (eventuelt med sekvensdiagrammer). Simuleringen foregår ved at man diskuterer i gjennom arkitekturen sett i forhold til både funksjonelle og ikke-funksjonelle krav. Vi går flere runder

med endringer i skissene og diskusjoner til vi har dekket use cases og til vi tror vi har dekket ikke-funksjonelle krav.

Den virkelige testen på om arkitekturen er god kommer dessverre ikke før helt til slutt. Men det finnes målinger som kan gjøres løpende i prosjektet:

- Pålitelighetstester, hvor man kjører systemet under belastning over lengre tid for å måle feilraten og oppetid.
- Ytelsestester, hvor den enkleste formen er ganske enkelt å måle responstid på en operasjon i fra brukeren gjør en aksjon til han får svar.
- Kapasitetstester, hvor man kjører systemer under høy last til vi finner det punktet hvor systemet starter å gi dårligere ytelse.

Et viktig men ofte glemt ikke-funksjonelt krav er at systemet skal være lett å vedlikeholde. Vedlikehold betyr i praksis det vi gjør av feilrettinger og forbedringer etter at systemet i første omgang er regnet som ferdigstilt. Dette omfatter, for et typisk system, 70 % av den totale kostnaden ved et system. Vedlikehold skiller seg i bunn og grunn ikke fra utvikling av første versjon av et system, med unntak av en vesentlig faktor. Nemlig at andre personer er nødt til å sette seg inn i systemet og forstå hvilke konsekvenser en endring vil få. Et veldefinert system med høy kohesjon og reduserte avhengigheter på tvers av delsystemer gjør denne jobben vesentlig enklere. Problemet med å gjøre endringer i andres kode er at du fort kan lage nye feil fordi du ikke forstår alle konsekvenser av endringen. Det kan fort bli slik at retter du en feil dukker det opp et par nye et annet sted i programmet.

Arkitekturen må dokumenteres hvis den skal ha noen verdi for andre arkitekten selv. Vi trenger i det minste å kjenne til hovedkomponentene i løsningen og relasjonene mellom dem.

Beskrivelse av design er i praksis ofte gjort med diverse konseptuelle skisser og samling av tekstlige dokumenter. UML-diagrammer av typen component, deployemnt og overordnet klasser egner seg til arkitekturnivå. Det finnes mange måter å beskrive arkitektur på, men for alle arkitekturbeskrivelser er det et prinsipp som gjelder og det er at vi må skille mellom forskjellige konseptuelle aspekter eller views. Dette går ut på at for et kompleks system er det umulig for vår (min i hvert fall) lille hjerne og ta inn over oss alle detaljer. Vi må lage abstraksjoner for å få oversikt, og vi må se på et utvalg aspekter av gangen og ikke alle på en gang. Vi kan for eksempel se på logisk arkitektur, og fysisk arkitektur. Fysisk arkitektur er maskiner og hva som kjører på dem, mens logisk arkitektur er programvarekomponenter sett i fra utvikler/drift.

En mye brukt logisk arkitektur er en trelags logisk arkitektur. Her vil alt som har med bruker-interaksjon ligge på hva vi kaller presentasjonslaget, forretningslogikk i et eget lag og til slutt et datalag. Selv med en trelags-arkitektur kan vi fortsatt velge en tolags fysisk arkitektur hvor vi kjører en tykk klient. Den tykke klienten vil i så fall inneholde to logiske lag. Klienten er programmet som kjører presentasjonslaget men vi kan, som vi ser, ha flere lag i tillegg til presentasjonslaget. Hvorvidt en klient kalles tykk eller tynn går på hvor mye ressurser den krever og hvor mye kode den kjører. Skillet mellom tykke og tynne klienter er spesielt uklart i nettleser-baserte klienter hvor vi kan kjøre deler av koden i nettleseren. Det er da mer en diskusjon om hvor tykke, eller små-lubne klientene er.

I eksemplet i forelesningen utvikler vi en Web-applikasjon, karakterisert ved at klienten er HTML-sider som vises i en nettleser. Nettleseren kan inneholde skripting som gir mer dynamiske sider, men det er fortsatt Web Server som genererer sider. Det er tilstandsløs kommunikasjon mellom nettleser og Web server (http/s) som gir utviklerne noen ekstra utfordringer. Det er også vanskeligere å kontrollere presentasjonen enn på en tykk klient. En Web-applikasjon tillater brukeren å utføre forretningslogikk via en nettleser. Dette i motsetning til de fleste sider på Internett som bare serverer innhold. En netthandel er for eksempel en Web-applikasjon, mens VG.no er i hovedsak publisering som bare presenterer innhold.

Brukergrensesnitt

For brukerne er grensesnittet selve løsningen. Det "er" produktet. Det er jo det de ser og jobber med. Alt det andre som ligger bak ser brukene aldri og bør strengt tatt ikke bry seg med. Nå i dag er det kun grafiske brukergrensesnitt som teller. Gode brukergrensesnitt er selvfølgelig viktig for brukerne som tilbringer store deler av arbeidstiden sin med å jobbe med dem. Men godt design og brukervennlighet selger også. GUI er implementert i presentasjonlaget i trelagsarkitekturen.

Vi skiller mellom det som er rent design og brukervennlighet selv om de to begrepene går inn i hverandre. Med design tenker vi på hva du ser av layout, ikoner, farger, fonter osv. Brukervennlighet derimot, går på hvordan systemet er å bruke, om det er lett å lære og hvordan det oppfører seg. Vi ser på om systemet er forståelig for en bruker, om feilmeldinger er forståelige, om det er konsistent i oppførsel mellom forskjellige skjermbilder osv. Brukervennlighet er avhengig av hvilken brukergruppe systemet er rettet mot. En bruker som jobber med systemet hele dagen og etter hvert kjenner det veldig godt ønsker hurtigvalg og prioriterer rask respons. Er du innom systemet en sjelden gang er behovet for veiledning og wizards ønskelig, mens ekspertbrukeren ville bli drevet til vannvidd hvis en wizard eller en pratsom binders dukket opp i operasjoner han skal gjøre hundrevis av ganger om dagen.

Det kan være vanskelig å måle brukervennlighet på samme måte som feks. responstid og ressursbruk. Men det finnes måter som f. eks.

- Hvor lang tid tar det å lære systemet for nybegynnere?
- Hvor mange "brukerfeil" oppstår med erfarne brukere?
- Hvor ofte får brukerne meningsløse tilbakemeldinger?
- Responstid er også veldig viktig for brukeropplevelsen og kommer inn også her.

Nå finnes det heldigvis en del retningslinjer for å bygge gode brukergrensesnitt. Her er et slikt sett:

- Brukeren i førersetet, det vil si at brukeren føler at det er han som kontrollerer systemet og ikke omvendt.
- Konsistent, for eksempel ved at samme operasjon gjøres likt i alle skjermbilder.
- Kan personifiseres og konfigureres.
- Er tilgivende og gir deg mulighet til å rette opp.
- Gir deg tilbakemeldinger, gjerne forståelige.
- Tilpasset oppgavene og brukergruppen (usability).
- Designmessing tiltalende, uten forstyrrende element.

Disse tommelfingerreglene er i fra Maciaszek sin bok. Det finnes mange varianter over disse som stort sett går ut på det samme. I Visma har vi et sett med ti regler og vi har laget egne guidelines rundt dem. I tillegg til de syv punktene over, fokuserer vi mye på at brukergrensesnittet skal være enkelt, kutte ut overflødig funksjoner, osv.

Utvikling av brukergrensesnitt er veldig godt egnet til prototyping og iterativ utvikling. Vi starter med skisser og mock-ups laget med presentasjonverktøy, Flash eller likende. Da er det enkelt å få umiddelbare tilbakemeldinger fra brukere. I design av brukergrensesnitt er det sjelden at det er noe utstrakt bruk av UML. For de aller fleste prosjekter av noe størrelse vil vi velge oss et GUI rammeverk som legger føringer på design av klassestrukturen.