UNIVERSITETET I OSLO
Institutt for informatikk

Effektiv modernisering av Enterprise Informasjons-Systemer

Begrepsapparat for bruk av agile metodikk, forsterket av dimensjonene arkitektur og teknologi

Hovedoppgave

Arve Klev

1. mai 2007

Forord

Opprinnelig var planen å skrive om modeller og metoder for systemutvikling i Forsvaret. Inspirert fra ulike kilder om mere smidighet og enklere teknologi, og mye arkitektur-fokus i Forsvaret, ble min interesse vekket for å se alle tre dimensjoner samlet.

Min veileder Dr. Tone Bratteteig, penslet meg over på riktig spor ved å se dimensjonene arkitektur og teknologi som mulige forsterkere til metodikken. Hun ga meg videre de viktige og nødvendige tilbakemeldinger til at det kunne bli en oppgave—tusen takk!

Takk til sivilingeniør Kjetil Johnsen, som har gjennomgått hele oppgaven og gitt kommentarer. Videre takk til Kurt Veum, Principal Scientist ved Nato's gruppe for operativ arkitektur (NC3A), og tidligere forsker ved FFI, for innspill spesielt vedrørende arkitektur. Og takk til stipendiat Jon Vatnaland ved UiO for hjelp med stoff omkring "outsourcing og norsk politikk". Takk til min arbeidsgiver, og dyktige kolleger i Forsvaret for all støtte.

Til slutt går min hjertelighet til mine to barn Julie (6) og Kristoffer (8) som jeg ikke har sett mye til i innspurten med oppgaven. Min kone Siss ga meg nødvendig puff til å bli ferdig, og muligheten til å kunne skrive kontinuerlig den siste tiden—tusen takk.

Innhold

Kapittel 1

Oppgaven

Tenk om vi hadde et stort enterprise informasjonssystem som virksomheten var helt avhengig av, men hvor alle endringer medførte store kostnader og tok lang tid før de var materialisert. Tenk videre at ønsker som web-basert brukergrensesnitt og portlets i en enterprise portal ikke lot seg realisere med dagens arkitektur og teknologi på et fornuftig sett. Og som om ikke det var nok, tenk om høynivå integrasjon mot andre systemer virket som ren ønsketenkning.

Hva hvis det kunne finnes en enkel tilnærming til modernisering av virksomhetens informasjonssystem, hvor videre vedlikehold skjedde kostnadseffektivt og hurtig, og web-grensesnitt inngikk i moderniseringen. Samtidig som vi kunne vite at portlets og høynivå integrasjon var uproblematisk.

1.1 Mål

Min **hypotese** er at agile metodikk er velegnet for modernisering eller re-design av enterprise informasjonssystemer. Men agile metodikk må følges av arkitektur og teknologi som forsterker de viktige aspektene ved agile metodikk.

Målet med oppgaven er å se om den nevnte hypotesen holder, gjennom å ta utgangspunkt i, og argumentere for, en hensiktsmessig metodikk. Videre vil jeg gå inn på dimensjonene for henholdsvis arkitektur og teknologi, og se på sider som jeg mener er avgjørende for modernisering av informasjonssystemer. Disse dimensjonene vil jeg så trekke med meg tilbake til metodikken i form av mulige forsterkere for denne.

Jeg ser primært på skreddersydde enterprise informasjonssystemer som følger en to-lags klient-tjener arkitektur hvor persistering av data skjer på en sentral tjener, og hvor brukergrensesnitt sammen med hele eller deler av logikken er distribuert ut til klientene. Jeg forutsetter normalt god båndbredde i nettverket.

Jeg avgrenser meg vedrørende sikkerhet i form av autorisering og autentisering. Her er det mange ulike muligheter og valg som vil avhenge av hva virksomheten allerede benytter, eller ønsker og trenger.

En annen avgrensning jeg gjør er å ikke gå inn på problematikken som kan oppstå når et team settes sammen med utviklere og bruker-representanter, hvor gruppedynamikken kan lede til konflikter, som i sin tur svekker teamet. Dette er et område jeg er klar over, men allikevel velger jeg ikke å behandle det her.

1.2 Metodebruk

Forskning kan være i form av en systematisk undersøkelse man selv foretar—eller man kan gjøre bruk av andres forskning, gjennom å se på deres empiriske undersøkelser. Jeg har sett til andres forskning, hvor jeg benytter deres empiri hvor det er relevant som støtte til egne erfaringer, og til underlag i min argumentasjon.

Hovedvekten av egen empiri beskrives i appendiks A. Jeg har kunnet følge en liten systemutviklingsgruppe som utfører skreddersøm av informasjonssystemer i Forsvaret. Empirien er i form av observasjoner av flere prosjekter som er utført siden våren 2004, i tillegg til at jeg selv siden høsten 2006, også har kunnet ta del i deler av dette arbeidet. Prosjektene ligger tett opp mot den metodikk, arkitektur, og teknologi som oppgaven omhandler, og hvor effekten og resultatet av disse dimensjonene og innbyrdes avhengighet, er observert. Jeg ser en rekke ting som jeg kunne konkludert med ut fra praksis, men kan allikevel ikke gjøre slike konklusjoner grunnet manglende egnet datamateriale. I ettertid ser jeg at systematiske undersøkelser skulle vært foretatt for å skaffe til veie slikt datamateriale, men jeg så ikke nødvendigheten tidlig nok. I mangel på systematisk datainnsamling, må jeg støtte meg på egne erfaringer, og velger av den grunn å allikevel å benytte egne observasjoner som datamateriale—som en illustrasjon, og gjør bruk av andres forskning som et supplement.

Videre empiri er opparbeidet gjennom eget arbeide i Forsvaret med systemutvikling av informasjonssystemer. Gjennom å ha vært aktivt deltagende gjennom mange år i ulike prosjekter, har jeg kunnet se og erfare hvordan IT-verdenen sett fra deler av Forsvarets side har utviklet seg. Jeg mener å kunne påberope meg bred praksis på området, men mangler som sagt de systematiske undersøkelsene som jeg i ettertid ser mangler.

Gjennom kontakt med forskere fra Forsvarets Forskningsinstitutt (FFI), fra Nato's gruppe for operativ arkitektur, og ulike fagmiljøer i Forsvarets tekniske IT-organisasjon, FLO/IKT, kontakt med flere av konsulenthusene som leverer utviklingstjenester til både offentlig og privat sektor, har jeg kunnet drøfte problemstillinger og erfaringer fra reelle prosjekter. En slik uformell og åpen diskusjonsutvekling er en kvalitativ god informasjonskilde som ofte ikke kommer fram gjennom formelle intervjuer. Nevnte erfaringsutvekslinger har gitt meg litt flere nyanser i måten å se systemutviklingsprosjekter på. Ulempen er at formell dokumentasjon mangler, og at jeg ikke kan påberope meg objektivitet.

Endel teoristoff er hentet fra pensum i faget IN364. Dette er ytterligere komplettert med teori som omhandler nyere retninger for systemutvikling. Jeg har videre innhentet teori fra kilder som omhandler dimensjonene arkitektur og teknologi, som jeg har funnet relevant for oppgaven. Mange synspunkter kommer fra grunnleggende vitenskaplige artikler, andre fra bøker av profilerte navn innen den del av IT-verdenen som er betraktet, og til slutt i form av ulike nett-sider med informasjon som ikke nødvendigvis er publisert i form av vitenskapelige artikler. En god del stoff og tanker er av relativ ny art.

Underveis i min kildegranskning har det dukket opp spørsmål som kunne vært grunnlag for intervjuer. Slike intervjuer er ikke foretatt, men jeg ser i ettertid behovet. Jeg jobber selv i en stor virksomhet, og mottar regelmessig henvendelser om intervjuer til besvarelse. I en travel hverdag er det sjelden jeg finner tid til å imøtekomme henvendelsene. Dette har nok også vært en faktor

for at jeg ikke tidlig vurderte å gå ut med slike henvendelser selv. Jeg opplever i ettertid at slik systematisk datainnsamling ville gitt meg et bedre metodisk redskap til å underbygge min argumentasjon, i tillegg til å høyne forskningsfaktoren og grunnlag for senere gjenbruk hos andre.

Forsvaret var for meg en naturlig undersøkelsesenhet. Mitt arbeidssted der har medført lett tilgang på observasjoner. Videre som en stor bedrift med over 15 000 ansatte, og fordi de har brukt IT-systemer på godt og vondt gjennom mange år. Forsvaret ved Forsvarets ForskningsInstitutt (FFI), har i mange år vært med som en av pådriverne innen IT-utviklingen i Norge. I tillegg til Forsvaret som undersøkelsesenhet, burde jeg også ha innhentet erfaringer fra prosjekter som er utført i andre virksomheter, slik at ytterligere nyanser kunne kommet fram.

En tradisjonell kritikk til mine erfaringer ("case"), er at metrikker skulle vært satt opp i forkant, med påfølgende målinger utført underveis. Det nærmeste jeg har å støtte meg til av målinger er et elektronisk oppfølgingsystem hvor ønsker, behov, og feil registreres. Videre angis medgått tid for implementering av en utført registrering. I tillegg har jeg tatt ut statistikk på kodelinjer, ulike utviklere, og tidsperioder fra versjons-håndteringssystemet. Highsmith mener forøvrig at vi ikke kan støtte oss på målinger som er egnet i en definert prosess, men må finne støtte i å måle resultatet i form av produktet kunden vil ha[46]. En slik undersøkelse kunne vært utført i form av formelle intervjuer med både utviklere og kunder for å dokumentere deres erfaringer av prosessen, og kundene alene for å registrere deres grad av tilfredshet med resultatet. Videre kunne jeg ha skrevet dagbok som dokumenterte de ulike prosjektenes historie. Dette valgte jeg ikke å utføre. På den annen side har jeg uoppfordret mottatt responser på tilfredshetet av artefaktet for ulike prosjekter.

3

Kapittel 2

Bakgrunn

Den gang da datamaskiner begynte å bli allment tilgjengelige—altså epoken etter hullkortmaskinenes tid—lå programmenes logikk og presentasjon på sentrale maskiner med karakterbaserte terminaler tilknyttet. Etter hvert kunne grafiske terminaler benyttes, slik som X-terminalene, hvor bruker-opplevelsen ble bedre gjennom en grafisk arbeidsflate med flere vinduer som kunne brukes samtidig. Logikk og presentasjon lå fortsatt sentralt, sammen med persistering av systemenes data. Sentralisering var dominerende.

Etter at PC'ene gjorde sitt inntog, skulle brukeren få det lettere og spare tid, og samtidig få råderetten over sin datamaskin. Dette medførte at mange enterprise informasjonssystemer (EIS[1]) måtte skrives om for å kunne installeres lokalt hva angikk logikk og presentasjon. På større arbeidsplasser skapte dette nye ressursmessige utfordringer som måtte håndteres. Programvaren til den enkelte PC skulle ikke bare installeres, men siden vedlikeholdes, konfigureres og oppgraderes. Andre utfordringer kom som en følge av ulike versjoner av operativsystemer og annen programvare hos brukeren, i tillegg til varianter av maskinvaren. Ikke desto mindre var desentraliseringen av logikk og spesielt presentasjon toneangivende.

Med den stadige ekspanderende globaliseringen i samfunnet og utbredelsen av Internet og bedriftenes intranett, skrives i økende grad enterprise informasjonssystemer for en sentralisert mellomvare-plattform, med fortsatt bruk av sentralt persisterte data. Brukertilgangen til de sentrale programmene foregår i økende grad gjennom standardiserte weblesere med nettverkstilgang, og andre systemer kan integreres mot de samme grunnleggende tjenestene tilbudt via såkalte webtjenester og portlets. Vi er tilbake til full sentralisering av EIS'ene, men mere standardisert og med utvidet anvendelse.

En samfunnstrend vi har kunnet observere de siste årene er en økning av bortsetting. Denne trenden har for virksomheter først og fremst bestått av bortsetting av oppgaver definert utenfor bedriftenes kjernevirksomhet[43]. Sentralisering av EIS vil kunne gi grobunn for ytterligere bortsetting av oppgaver relatert til drift og forvaltning av EIS. Bortsettingen kan veksle mellom ulike driftsoperatører av konkurransemessige årsaker eller ved at en operatør går konkurs. Bortsatte systemer kan senere ønskes tilbakeført til egen virksomhet. Avhengig av hvordan denne pendelen svinger, og hvor mange

[1] EIS—Enterprise Informasjon System. Se definisjon på side 119

oppgaver som settes bort, kan også en del av virksomhetens kunnskapskapital forsvinne i prosessen[2]. Gitt en antagelse om at systemene inngår i virksomhetens kjerne, fordrer dette at kunnskapen om systemene må være en del av kunnskapskapitalen hos bedriften.

Alt er i endring, eller pendler; samfunnet, virksomheter, prosesser, mennesker, krav og teknologi. Eksisterende enterprise informasjonssystemer vil påvirkes av slike endringer. Konsekvensen er at systemene må være endringsvillige, slik at de ikke fordrer uforholdsmessig høye utgifter gjennom sitt livsløp.

I en verden hvor grunnleggende premisser[3] gir krav om lavere kostnader *(billigere)*, hurtigere tempo og resultater *(raskere)*, og økende kvalitetskrav og flere anvendelsesområder *(bedre)*: Hvordan kan vi håndtere utvikling og vedlikehold av programvare på en mer tilfredsstillende måte? Og mer spesifikt; hva gjør vi med eksisterende enterprise informasjonssystemer som mangler viktige anvendelsesområder som web/portlet-basert brukergrensesnitt, tjenestegjenbrukbarhet, eller hvor endringsvillighet, integrerbarhet, og deploybarhet bør forbedres? Legger vi til en fjerde premiss som påvirker eller innvirker på de tre tidligere nevnte; hvordan kan vi unngå unødvendig kompleksitet og sørge for at systemet bare omfatter akkurat det vi trenger *(enklere)*?

Med premissene som bakgrunn kan de totale kostnadene til systemet (TCO[4]) bli uforholdsmessig høy i forhold til nytteverdien. Høy TCO for EIS kan kjennetegnes ved at systemet er ressurskrevende å drifte og vedlikeholde, endringer, utvidelser og deployering tilsammen tar lang tid og koster mye, funksjonalitet mangler, men ingen tør endre systemet, nye anvendelsesområder og kvaliteter mangler eller er utilfredsstillende.

Målet er å kunne modernisere eller re-designe et EIS til et system som er klargjort for hurtige endringer på en kost-effektiv måte, og samtidig tilføre kvalitet og viktige anvendelsesområder som er nødvendige. Unødvendig kompleksitet og funksjoner som er framkommet gjennom årenes løp forsøkes fjernet. I denne prosessen ønsker jeg å kombinere de tre dimensjonene metodikk, arkitektur, og teknologi.

2.1 Premisser

Premissene påvirker hverandre i ulik grad avhengig av hvor de anvendes. Eksempelvis kan et system som er enkelt og raskt å endre og deployere, oppleves som som mindre rikt i bruker-grensesnittet. Hva som er bra for én gruppe brukere, trenger ikke være til det beste for virksomheten. Kompromisser vil måtte inngås.

Billigere. Etter at programvaren er utviklet, starter tradisjonelt de største utgiftene å rulle[72]. For noen virksomheter har det vært estimert at 80% av kostnadene for programvare kommer fra vedlikehold og videreutvikling[85]. "Chaos"-rapporten[73] viser at prosjekter ofte blir dyrere, tar lengre tid, og ikke gir den funksjonaliteten som prosjektet var ment å avstedkomme. Andre studier viser til at re-design kan være opp til fire ganger billigere enn nyutvikling[72].

[2] Prosess—De mentale og fysiske aktivitetene vi utfører for å oppnå et resultat
[3] Premiss—i betydningen forutsetning. Ikke premiss brukt for å sette fram en hypotese.
[4] TCO—Total Cost of Ownership

Hvis vi kan re-designe systemer og flate ut kostnadskurven for vedlikehold og videreutvikling, mener jeg det vil gi merkbare besparelser for virksomhetene.

Lønnskostnader i Norge er relativt høye sammenlignet med for eksempel den nye IT-nasjonen India. For å redusere kostnader knyttet til lønninger i norske utviklingsprosjekter, kan vi redusere tiden prosjektet varer og antall personer som deltar. Dog er det viktig å påpeke at kostnader er relative i den forstand at desto mer verdi et system har for bedriften, jo mer regningsvarende er det og høyere kostnader kan forsvares. Men dette tilsier ikke at vi kan sløse, og i et konkurransemarked kan det utmerket godt være at ett konsulentselskap leverer et bedre system på kortere tid enn hva andre er i stand til. Det gjelder i slike tilfeller å ha nok kunnskap i virksomheten til å ha *kunnen*[5] til å velge riktig.

Uavhengighet til leverandører kan være et annet kostnadsreduserende tiltak, og som fremmer konkurranse[30]. Hvis denne uavhengigheten i tillegg støttes av kostnadsfrie[6] åpen kildekode produkter med tilsvarende kvalitet som kommersielle produkter[75], kan vi unngå bindinger til én leverandørs produkter med store initielle og årlige kostnader og senere reduserte valgmuligheter. Forutsetningen er at produktene støtter den teknologien vi ønsker å benytte.

Sentralisering av applikasjonene, slik det er mulig for enterprise informasjons systemer, kan redusere utgiftene betraktelig gjennom enklere og raskere tilgjengeliggjøring (deployering) og administrasjon. I tillegg kommer effekten av redusert ventetid for brukerne for tilgang til nye endringer og eventuelle feilrettinger.

Raskere. Med samfunnets akselererende endringshastighet som også gjør seg gjeldenende i virksomheter, med følger for organisasjonsstrukturer og arbeidsprosesser som endres, er det problematisk for IT-systemene som støtter en gitt struktur og prosess å henge med. Om virksomheten kunne plukke ut og sette sammen ny IT-støtte fra gjenbrukbare tjenester i informasjonssystemene, ville bildet sett anderledes ut. For EIS'er uten slik anvendelse for tjenestegjøring, fordres endringer.

Hvis hastigheten for re-design og senere vedlikehold skal økes, trenger vi korte iterasjoner, samt hurtig og enkel deployering for raskt å gi ut nye versjoner med ny og forbedret funksjonalitet. Men da må metodikken være smidig, programvaren enkel å endre, og plattformen sentralisert.

Raskere—hvis hvis vi kan finne rammeverk som støttes av en hensiktsmessig arkitektur, som gjør det lett å bytte ut komponenter, tjenester og teknologier som applikasjonen kan nyttiggjøre i form av byggeklosser. Vi bør kunne gjøre endringer og utskiftninger raskt og billig på et senere tidspunkt.

Raskere—hvis vi kan finne metodikker som bedre er tilpasset endringer i all faser av et utviklingsløp, og som kun vektlegger artefakter som er helt nødvendige.

Raskere må også tolkes i en videre forstand for å kunne forsvares opp mot 90-tallet's til dels meget effektive utvikling av store informasjonssystemer med hjelp av CASE[7] eller verktøy for 4. generasjons språk (4GL). Vi må ta hensyn til andre aspekter enn bare utviklingstiden, og muligheten og fordelene med verktøystøttet hurtig prototyping som, når prototypen er godkjent, faktisk

[5] *Kunnen*— en av de gamle greske dygder
[6] Kostnadsfritt ved anskaffelser og ingen lisensutgifter, og dugnadsdrevet support
[7] CASE—Computer Aided Systems Engineering

representerer det kjørbare systemet. En effektivitetsårsak er at disse proprietære verktøyene fra en gitt leverandør, var meget datamodell-sentriske og gjorde det tilnærmet automatisk å lage brukergrensesnitt fra en datamodell som var definert eller importert inn i verktøyet, gitt at man fulgte vertøyets filosofi. En annen grunn var at systemene kun var ment til bruk for en begrenset mengde brukere, og hvor tilgjengeliggjøring for Internet ikke var et ikke-funksjonelt krav. Det er denne kategorien av systemer, tykke klienter med direkte nettverkstilgang mot en database-tjener, som er oppgavens fokus.

Hurtig applikasjons utvikling (RAD[8]) må fortsatt være et kriterie, men på samme tid må åpne standarder, leverandør-, verktøy-, og lisens-uavhengighet vurderes. Og ikke minst må vi ta hensyn til at systemene blir mer og mer agnostisk til antall brukere og den geografiske plasseringen, samtidig som andre systemer skal kunne aksessere tjenester eller integreres mot disse.

Bedre. Mange prosjekter har endt opp med et EIS som ikke gir kunden hva som forventes når leveransen kommer. Systemet er en implementasjon av en spesifikasjon som ikke lengre stemmer med virkeligheten. For å sikre at systemet har riktig funksjonalitet og er det kunden egentlig vil ha, kreves en høy grad av brukermedvirkning og forankring i ledelsen, og er helt avgjørende og rangeres som de to viktigste faktorene for vellykkede prosjekter[74]. Kodebasen fordrer i tillegg endringsdyktighet for ikke å gi en vesentlig forhøyet kostkurve[13].

Bedre gjennom at virksomhetens kildekode kan forvaltes, genereres og driftes internt eller settes bort om de strategiske ledelsesvurderinger tilsier det—og uavhengig av om maskinvare og operativsystem byttes. Hvis situasjonen senere endres, må man enkelt kunne flytte kodebasen til det sted hvor det er mest hensiktsmessig. Det bør være unødvendig å senere måtte endre eller re-designe fordi virksomheten eller en tjenestetilbyder, ASP[9], ikke kan tilby riktig plattform til forvaltning, generering, og kjøring av systemene.

Nye anvendelsesområder som høynivå integrasjon, webtjenester, portlets, enkle standardiserte grensesnitt kan gjøre systemene bedre. Integrasjon på et høyere abstraksjonsnivå gjør det mulig å bygge tjenester av andre gjenbrukbare inter-operable tjenester. Deler fra informasjonssystemene kan inngå i virksomhetens portal, hvilket øker mulighetsrommet og nedslagsfeltet for EIS.

Enklere. Enkelhet—ikke primitivt og lite sofistikert—men lett å forstå, brukervennlig og intuitivt.

Kompleksitet koster penger, tar lengre tid, og gjør ikke nødvendigvis sluttproduktet bedre. Det samme kan sies om unødvendig funksjonalitet utviklet i tilfelle noen kan trenge den. Et eksempel er "Word" fra Microsoft, som er overlastet med funksjonalitet de færreste bruker eller trenger. "Ut-esing" av programvare er en negativ trend som stadig brer om seg[40]. I tillegg til unødvendig funksjonalitet, kommer kompleksitets-økning av arkitekturen promotert fra applikasjons-tjener leverandører, for å kunne rettferdiggjøre kompliserte og kostbare produkter. Arkitekter som kun tegner arkitekturen kan bidra til å lage en unødvendig komplisert løsning. De som selv deltar i implementeringen, vil trolig lage enklere løsninger, med bedre resultat, lavere kostnad, og komme i mål med kjørbar kode raskere[53].

[8] RAD—Rapid Application Development
[9] ASP—Application Service Provider

Prosjektledere og oppdragsgivere sverger ofte til prosessmodeller for systemutvikling som lager mange, og i praksis unødvendige, artefakter i form av mange og store dokumenter, pent og sirlig oppsatte og syntaktisk riktige UML-diagrammer som det er brukt unødvendig lang tid på, men som egentlig ikke gjør det endelige produktet, kjørbar kode av det riktige systemet, bedre[24, 40].

2.2 Alternativer for modernisering

For eksisterende enterprise informasjonssystemer som trenger modernisering, har vi flere muligheter. Tar vi de nevnte premisser i betraktning, kan ulike valg avhengige av situasjonen. Den totale systemporteføljen må også sees under ett, og om virksomheten har mange kritiske systemer som skal spille sammen, kan det på sikt bli kostbart å velge feil strategi.

For å synliggjøre ulike alternativer vil jeg først gi en kort beskrivelse av noen varianter som kan være aktuelle. I seksjon 2.2.1 kommer jeg nærmere inn på det valget jeg legger til grunn for oppgaven.

Terminal-tjenere. Man kan modernisere EIS ved å flytte programvare over til sentrale terminal-tjenere. Dette kan være et alternativ for systemer installert rundt omkring på brukernes PC'er som tykke klienter med logikk og presentasjon samlet. Systemene kan være både hyllevare eller skreddersydde løsninger hvor vedlikeholdskostnadene er akseptable, og hvor ikke høynivå integrasjon eller tilgjengeliggjøring for VVV[10] er påkrevd. Valget forutsetter kontroll av nettverket og klientene som bruker systemene.

Teknologi-overbygg. Man kan velge å legge et teknologilag over det eksisterende systemet som fra før ligger på en sentral server og er utviklet i en monolittisk arkitektur. Det eksisterer kommersielle produkter i denne kategorien, og et slik valg treffer typisk applikasjoner med en monolittisk arkitektur, hvilket ligger utenfor min avgrensning.

Konvertere. En annen mulighet for modernisering er å velge én leverandør og tilhørende proprietære verktøy som konverterer eller tilgjengeliggjør aktuelle to-lags klient-tjener systemer til å kunne kjøres sentralt med tilgang gjennom weblesere. Et eksempel er Oracle's tradisjonelle "Forms"-applikasjoner som er installert på brukernes PC'er. Disse kan ved hjelp av standard J2EE teknologi, blandet med Oracle's proprietære utvidelser og verktøy, kjøres som en 3-lags applikasjon med tynne klienter (nesten, skjermbildene går som "applets" i webleseren, og trenger en proprietær "plugin" for å virke). I noen tilfeller kan en også relativt enkelt utvide funksjonaliteten med webtjenester og portlets. Ulempene er at man også velger leverandør-, verktøy-, og lisens-avhengigheter, hvilket kan gi en høy kostnad og et senere begrenset mulighetsrom.

Med India's vesentlig lavere lønnskostnader i forhold til Norge, og dyktige IT-folk, kan en vurdere å sende bort konverteringsjobben for manuell utførelse. Man kan da sette nye krav til arkitektur og teknologi som systemet konverteres til. En viktig forutsetning og begrensning for denne muligheten er om man makter å overføre nok kunnskap og teori om systemet som skal konverteres.

[10] VVV—VerdensVeVen eller World Wide Web (WWW)

Hyllevare og "Halvfabrikata". Man kan kjøpe ny hyllevare som dekker hele eller deler av den aktuelle porteføljen, men som passer premissene bedre. Innkjøp av hyllevare til erstatning for eksisterende informasjonssystemer krever spesialtilpasset datakonvertering. Integrasjon mellom ulike hyllevaresystemer eller mellom eksisterende systemer og ny hyllevare, skjer normalt nede på database-laget ved at data kopieres og oppdateres mellom systemenes databaser. Denne integrasjonstypen kan enten foregå direkte mellom to systemer, eller ved hjelp av en integrasjonsløsning som kan gjenbruke og holde oversikt over ulike integrasjoner.

En variant til hyllevare er "halvfabrikata" med muligheter for større tilpasninger gjennom konfigurering innenfor gitte rammer. SAP[11] kan være et eksempel på et produkt innen denne kategorien. Konvertering av eksisterende data fra gamle EIS må utføres på lik linje med konvertering til hyllevare.

En forstudie før valget kan treffes om bruk av hyllevare eller "halvfabrikata" må utføres for å avdekke forventet dekningsgrad. Lav treffprosent eller dekningsgrad indikerer at løsningen bør forkastes. Situasjonen kan bli at systemer som var planlagt sanert, likevel må leve fordi kritisk funksjonalitet for organisasjonen ikke blir tilfredsstillende dekket. Og for å sikre at gamle systemer kan saneres, kan vi oppleve at innføringen av "halvfabrikata" utarter seg til et utviklingsprosjekt med både store tilpasninger og skreddersøm, som medfører en sterk leverandør- eller produkt-låsing.

Store vedlikeholdsutgifter, slik det er vanlig for utviklingsprosjekter[72], kan også komme i denne kategorien som en følge av stor andel skreddersøm for å dekke kravene. Dette skal ikke være nødvendig om man konfigurerer innenfor rammene. Store tilpasninger i form av skreddersøm kan tyde på lav dekningsgrad som burde vært oppdaget i en tidlig forstudie.

Avhengig av merke "halvfabrikata", kan man oppnå bedre integrasjonsmuligheter og andre anvendelsesmuligheter enn hva ren hyllevare kan gi. Men ikke alltid—i forbindelse med Forsvarets GOLF-prosjekt[12] hvor SAP ble valgt som løsning, hevdet leverandøren som er ansvarlig for implementeringen at integrasjon mot eksisterende systemer innebar høy risiko, med den følge at flere eksisterende systemer utvikles, eller skal utvikles, på nytt ved hjelp av SAP-verktøy og skreddersøm.

Uansett må man forvente leverandør-, verktøy-, og lisens-avhengigheter som låser organisasjonen i mange år framover—spesielt for store og kostbare prosjekter.

Nyutvikling. Systemer kan moderniseres ved å utvikles helt på nytt og skreddersys for entreprisen gjennom et utviklingsprosjekt, fortrinnsvis med god brukerdeltagelse og ledelses-forankring. Denne valgmuligheten er typisk når man ikke ønsker, eller har anledning til å ta utgangspunkt i det eksisterende systemet. Et eksempel kan være at utviklingen sist gang ble satt bort, og at man nå ikke har tilgang på kode eller hodene som utførte jobben.

[11] SAP—www.sap.com ("systems, applications and products in data processing")

[12] Program GOLF ble etablert året 2000, for å samle all informasjon om økonomi og materiellforvaltning for hele Forsvaret i en felles løsning. Målet er bedre styring og kontroll på tvers av forsvarsgrenene. Programmet er senere omdøpt til program LOS, hvor fokus er spesielt rettet mot logistikk. Prosjektene i programmet kjøres serielt ved at et prosjekt avsluttes før neste starter. Dette medfører at gjennomføringen nødvendigvis tar lang tid, og at prosjektene utsettes for store endringer i de teknologiske omgivelsene.

En erfaringer jeg har opplevd i Forsvaret, er ved ett tilfelle da en avdeling ikke hadde annet eierskap til et system, bortsett fra bruk, at nyutvikling for samme system ble bestilt flere ganger. Den iboende kunnskapen, eller teorien til programmet, som design og implementering av systemet gir, kan ikke eksplisitt uttrykkes i dokumentasjon—den ligger i menneskene som lager sytemene[61].

2.2.1 Re-design

2.2.1.1 Forskjell på re-design og nyutvikling

Den kritiske forskjellen på re-design og nyutvikling er oppstartspunktet for utviklingen[72], hvor vi i re-design kan bruke det gamle systemet som spesifikasjon. Etter min mening, vil vi kunne gjenbruke datamodellen i stor grad, og hvis systemet oppfyller de funksjonelle behov, er forretningsreglene allerede gitt. Men underveis kan man oppdage, fordi kunden er med og gir innspill, at regler og funksjonalitet må endres, fjernes, eller legges til. Klargjøring for enkel endring er derfor meget viktig[24, 13]. Her trekker jeg parallell til re-design, slik at framgangsmåten eller prosessen for re-design tilnærmes den for nyutvikling, og vil etter endt re-design kunne være lik. Etter min mening gir det mest nytte om de samme teknikker og metoder følges i hele livsyklus-håndteringen. Og sannsynligheten for å bli fulgt øker hvis metodikken er enkel og inspirerer til bruk.

En annen ulikhet mellom nyutvikling og re-design er at sistnevnte normalt ikke fordrer organisasjonsendringer.

Innen brobygging utgjør design-delen av et prosjekt bare omkring 10%, mens den resterende delen (cirka 90%) er konstruksjon. For programvare er forholdet nesten snudd om[35]. Design-delen kan etter min mening reduseres betraktelig ved bruk av en referansearkitektur, samtidig som vi gjenbruker database-skjema og forretningsregler.

Forskjellen på re-design og nyutvikling, bortsett fra redusert omfang og derved redusert tid og prosjektstørrelse, er trolig mindre ved bruk av agile prosesser. I metodikk-kapitlet (3) går jeg nærmere inn på hvordan bruk av smidig metodikk utvisker mye av skillet mellom nyutvikling og re-design.

2.2.1.2 Fordeler ved re-design

En viktig grunn for å velge re-design av et system, er når organisasjonen er avhengig av det, og videre vedlikehold skjer på regulær basis[72]. Re-design forbedrer systemstrukturen, og gjør systemet lettere å forstå og høyner moralen for de som vedlikeholder det, ved at systemet er av høyere kvalitet. Re-design er spesielt viktig når datasystemer som har vært i bruk over lengre tid trenger videreutvikling for å tilpasses til ny teknologi[17]. I lys av dagens muligheter og krav, kan slik teknologi eller nye anvendelsesområder, være høynivå integrasjon og tjenestebygging ved webtjenester (WS[13]). Ytterligere muligheter er enklere deployering og bruk av tynne klienter som er agnostiske til brukerens arbeidsstasjon og geografisk plassering så lenge det finnes nettverkstilgang.

Premissene jeg satte opp i seksjon 2.1 gir utfyllende vurderingsgrunnlag for

[13] WS—Web Services

om re-design bør velges. Hvis premissene har svak gyldighet med hensyn på eksisterende system, indikerer det at re-design bør vurderes under forutsetning av at en kost/nytte analyse også tilsier et slikt valg.

Andre fordeler med re-design kontra nyutvikling fra bunnen av er følgende[72]:

Redusert risiko Det er forbundet med høy risiko å skrive om systemene som er essensielle for organisasjonen fra grunnen av uten å gjøre nytte av det eksisterende systemet. Spesielt ved at vi kan ta utgangspunkt i en ferdig spesifikasjon og som oftest et ferdig modellert database-skjema.

Statistikk fra "The Stadish Group" viser i tillegg at bare 28% av prosjektene lykkes med å levere i tide, innenfor budsjett, og oppfylle kravene til funksjonalitet og egenskaper[74].

Reduserte kostnader Kostnadene med re-design er langt lavere enn ved full omskriving av systemene. Ulrich[72, [79]] henviser til et eksempel hvor han fant at det var fire ganger rimerigere å re-designe enn å utvikle alt på nytt.

2.2.1.3 Kandidater for re-design

Kandidater for re-design innenfor oppgavens kontekst er skreddersydde enterprise informasjonssystemer med logikk og presentasjon distribuert ut til brukernes klienter, som persisterer sine data på en sentral tjener—altså systemer i en to-lags klient-tjener programvarearkitektur. Andre kandidater er de systemene som innehar en monolittisk systemarkitektur med tilsvarende persistermingsmekanismer, men den siste kategorien systemer faller utenfor min avgrensning av oppgaven.

Gitt disse kandidatene, er beveggrunnen for re-design de systemer som mangler de gitte fordeler jeg nettopp omtalte.

Når kandidater for modernisering vurderes, vil vi kunne støte på systemer hvor tidligere implementerte forretningsregler ikke lengre etterspørs, eller som har feil funksjonalitet i forhold til dagens krav. Følgelig kan deler av databasemodellen være delvis utdatert. Slike problemstillinger kan håndteres på ulike måter. En enkel løsning er å ikke implementere slik funksjonalitet videre ved å utsette til etter endt forbedringsprosjekt, og overgang til normalt vedlikehold trer i kraft. Re-design av feil spesifikasjon må unngås, og dermed gir en alternativ løsning seg i form av å først endre spesifikasjonen.

2.2.1.4 Kost/nytte analyse

I forkant av oppstart for den typen prosjekter som er perspektivet for oppgaven, må to faktorer analyseres for å kunne gi svar på om re-design skal utføres. Den første faktoren er *kostnad*, og den andre er *nytteverdi*. Kostnad henger sammen med premissene "billigere" og "raskere", mens nytteverdi dekkes gjennom premissene "bedre" og "enklere".

Jeg ser forutsetningen for re-design som at kostnad er lavere enn nytteverdi, og som jeg har introdusert en mulig beregningsformel for:

$$Kostnad < Nytteverdi$$

hvor nytteverdi utgjør produktet av levetid, kritikalitet, og endringsbehov:

$$Nytteverdi = Levetid \times Kritikalitet \times Endringsbehov$$

og følgelig vil nytteverdien bli lav eller nærme seg null hvis en av faktorene er tilnærmet null. Nytteverdiens faktorer kan beskrives ved:

Levetid Levetiden estimert i antall år etter endt re-design.

Kritikalitet Angis i forventet inntjening/besparelser per år ved å ha systemet. Et høyt antall brukere av systemet øker kritikaliteten. Hvis liv og helse kan stå på spill tallfestes ikke kritikalitet, men angis som en uendelig (∞) verdi.

Endringsbehov Angis i inntjening/besparelser per år for endringer som innføres eller muliggjøres gjennom re-design. For eksempel kan man se besparelser i form av mindre kostnader til deployering eller senere videreutvikling, eller bedre inntjening ved at hurtigere beslutninger og endring av virksomhetsprosesser kan tas ved at EIS'et integrerer sine tjenester med andre systemer.

Når endringsbehov vurderes, bør man også se på graden av leverandør-avhengighet ("vendor-lock-in"), som kan gi følger for vedlikeholdbarheten til programvaren. Er vi et offer slik "prinsipal-agent-teorien" beskriver? I så tilfelle vil re-design hvor vi makter å dreie vekk fra slik låsing være et godt bidrag til å ytterligere kunne redusere vedlikeholdskostnader.

Sist og ikke minst—for re-design av et system mener jeg en viktig økonomisk driver er å etterstrebe en ut-flatet kostkurve for den videre livsyklus-håndteringen.

2.3 Dimensjoner

Jeg har valgt tre dimensjoner for oppgaven; *metodikk*, *arkitektur*, og *teknologi*. Disse områdene, som jeg mener har sterke avhengigheter til hverandre, ønsker jeg å se på i et "Agile"[14] (agile) perspektiv.

Trenger vi en tung og besværlig *metodikk* med mange støtte-artefakter, eller kan denne være agile? Hvordan kan vi legge grunnlaget for videreutvikling gjennom realistiske rammer? Metodikken, enten man re-designer eller utvikler nye systemer, kan kanskje være den samme—man har bare kortere og mindre prosjekter uten samme grad av problemløsning inn mot problemområdet ved re-design? Men et re-designet system er ikke endelig, det er utgangspunktet for videre utvikling. Og om denne metodikken passer like godt for re-design som for nyutvikling, vil nytteverdien for vedlikehold økes ytterligere?

Skal *arkitekturen* være tilpasset en gitt leverandør's produkter eller en konsortium bestemt plattform, eller kan den designes til å være uavhengig og smidig? Hvilke mønstre kommer til anvendelse når vi arbeider med arkitekturen? Hvordan sikrer vi gjennom arkitekturen enkel høynivå integrasjon, videre tjeneste-bygging, og enkel utplassering, drift og forvaltning?

Hvilken *teknologi* bør vi vektlegge når vi re-designer? Og hvilke rammeverk og verktøy implementerer teknologien? Kan åpen kildekode gi nødvendig støtte til effektivitet og leverandøruavhengighet, og samtidig redusere kostnader?

Og til slutt; vil vi finne fordeler av å se disse tre dimensjonene samlet for å forsterke virkningen av en agile metodikk?

[14] Med "Agile" menes smidig, lettvekts, og nødvendig—ikke monumentalt, tungt, og unødvendig. Jeg velger i fortsettelsen å angi begrepet—agile.

Dette er en ytterligere detaljering til den innledende hypotesen og spørsmål, som jeg vil forsøke å besvare i de påfølgende kapitlene 3, 4, 5.

Kapittel 3

Metodikk

> "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> | | |
> |---|---|
> | **Individuals and interactions** | over processes and tools |
> | **Working software** | over comprehensive documentation |
> | **Customer collaboration** | over contract negotiation |
> | **Responding to change** | over following a plan |
>
> That is, while there is value in the items on the right, we value the items on the left more."
>
> — *Agile Manifesto[12]*

Generelt er en metode en tilnærming for å utføre en oppgave, mens en metodikk er studiet av en familie av metoder. Innefor programvare området omtales vanligvis en metodikk på samme måte som generelt for en metode; en tilnærming for å utføre og løse en oppgave[66]. Alternativt kan metodikk beskrives som alt hva vi vanligvis gjør for å få programvaren ut[24].

Jeg vil i dette kapittelet vise at metodikk innen systemutvikling er i endring. Fra ett ytterpunkt med fravær av prosesser for programvareutvikling, kom tilsvar i form av byråkratiske og monumentære metodikker for å håndtere svakhetene i den initielle tilnærmingen. Dette tilsvaret ser vi nå nye alternativer til i form av lette og smidige metodikker, med fokus på hurtig å lage vedlikeholdbar kode av høy kvalitet. Selv store monumentære metodikker justeres og kommer i "lettvekter" utgaver.

I henhold til den skandinaviske skolen for systemutvikling hvor brukerne har en sterk innvirkning[19], skal vi se at smidige metodikker faktisk forutsetter slik aktiv brukermedvirkning. Med utgangspunkt i at lette og smidige metodikker kan fungere godt for nyutvikling, ønsker jeg å forfekte re-design i en agile kontekst.

Målet med kapittelet er å gi en referansemetodikk for bruk senere i oppgaven. Metodikk-kapittelet er delt inn i følgende seksjoner:

Tradisjonelle modeller og metoder I seksjon 3.1 gir jeg en bakgrunn og argumentasjon til tradisjonelle modeller og metoder.

Abstraksjonsnivå	Rolle
Visjon	Virksomhet
Strategi	Virksomhet
Modell	Arkitekt/Utvikler
Metode	Utviklere/Bruker
Teknikk	Utvikler/Bruker
Verktøy	Utvikler/Bruker

Tabell 3.1: Abstaksjonsnivåer

Agile metodikker I seksjon 3.2 omtales en agile metodikk-tilnærming, med størst vekt på utvalgte elementer som kommer til anvendelse ved re-design. "Referansemetodikken"[1]er beskrevet til slutt i denne seksjonen.

Metodikk-forsterkere I seksjon 3.3 definerer jeg begrepet metodikk-forsterker til bruk i de etterfølgende kapitlene.

3.1 Tradisjonelle modeller og metoder

De primære funksjonene for en programvare prosess-modell er å bestemme rekkefølgen for stegene involvert i programvareutvikling, og etablere overgangskriterier fra et steg til det neste. Dette inkluderer avslutningskriterier fra ett steg, samt valgkriterier og inngangskriterier for det neste. En prosessmodell reiser to spørsmål for programvareutvikling. Det første omhandler hva vi skal gjøre etter nåværende steg, og det andre spørsmålet dreier seg om hvordan vi skal fortsette å gjøre det. Metoder fokuserer på hvordan en skal navigere gjennom en fase, og hvordan produktene for fasen skal representeres[15].

Noen typiske modeller er; "kod-og-fiks" modellen, den stegvise modellen og fossefalls-modellen, evolusjonær utvikling, transformerings-modellen, og spiralmodellen.

For å plassere prosessmodeller/systemutviklingsmodeller, metoder, og teknikker i en kontekst, vil jeg kort angi disse sammen med andre tilstøtende abstraksjonsnivåer. Deretter følger en beskrivelse av ulike tradisjonelle modeller og metoder for systemutvikling, slik de har vært, og fortsatt er i bruk, samt hvilke fordeler og ulemper de innehar.

Ulike abstraksjonsnivåer

Forestillinger og framgangsmåter kan uttrykkes på ulike abstraksjonsnivåer med dertil forskjellige detaljeringsgrader. Dette gjelder generelt for alle organisasjoner og virksomheter. Rollene derimot vil variere for type av virksomhet. En skjematisk framstilling er gjengitt i tabell 3.1. En presisering; man kan utføre oppgaver på ulike abstraksjonsnivåer samtidig, og samme person kan inneha flere roller.

På aristotelisk vis, eksemplifiserer jeg de ulike abstraksjonsnivåer og tilhørende roller i en imaginær systemutviklings-organisasjon:

[1]"Referansemetodikken" består av utvalgte elementer, og omtales heretter; referansemetodikk (uten anførsel).

Visjon Virksomheten setter opp en visjon. Dette er virksomhetens langsiktige mål. Et tenkt tilfelle:

> Systemutviklingsvirksomheten "Re-Design A/S", setter opp i sin visjon at de skal være raskest, best, og billigst innen sitt område om 5 år.

Strategi Virksomheten lager strategien. Dette er overordnede langsiktige beskrivelser som staker ut kursen framover for å nå visjonen. Et tenkt tilfelle:

> Systemutviklingsvirksomheten "Re-Design A/S", setter opp i sin strategi at de skal fokusere på effektiv re-design av enterprise informasjonssystemer, med fornøyde kunder som deltar aktivt i prosjektene sammen med fornøyde utviklere.

Modell Ulike modeller for, i vårt tilfelle, systemutvikling. En modell ligger under strategi-nivået, og kan romme flere metoder. Typisk rolle for modell-nivået er arkitekten. Et tenkt tilfelle:

> Systemutviklingsvirksomheten "Re-Design A/S" har en gruppe systemutviklere som utøver re-design aktivitetene. Gruppens arkitekt har vurdert, og kommet fram til at de skal skifte fra en dokument-sentrisk fossefalls-modell som få likte og som ga store vedlikeholds-kostnader, til en enklere evolusjonær utviklingsmodell. Den nye modellen passer bedre til virksomhets-strategien.

Metode Ulike metoder innenfor en modell. Ofte blir en samling metoder omtalt som en metodikk, hvilket jeg mener er det samme som en modell. Unntak kan være at samlingen med metoder ikke nødvendigvis passer inn i én gitt modell. Både utviklere og brukere som deltar i utviklingsprosjektet har roller her. Et tenkt tilfelle:

> Systemutviklingsvirksomheten "Re-Design A/S" har en gruppe systemutviklere som utøver re-design aktivitetene. De ønsker blant annet å bruke "eXtreme Programming" (XP) som metode innenfor en evolusjonær modell. Dette støttes av oppdragsgiverne, som samtykker i å allokere to superbrukere til å delta på fulltid i det lille og samlokaliserte utviklingsteamet.

Teknikk Innenfor en metode kan vi benytte ulike teknikker. Disse har ofte en form for verktøystøtte. Både utviklere og brukere som deltar i utviklingsprosjektet har roller her. Et tenkt tilfelle:

> Systemutviklingsvirksomheten "Re-Design A/S" har en gruppe systemutviklere som utøver re-design aktivitetene. De benytter TDD som er en fundamental teknikk innen blant annet XP. Superbrukerne på teamet bistår i å skrive test-senarier for funksjonelle tester.

Verktøy Verktøy kan være alt fra programvare til gråpapir og tavle. Verktøy kan ofte understøtte en gitt teknikk. Både utviklere og brukere som deltar i utviklingsprosjektet har roller her. Et tenkt tilfelle:

> Systemutviklingsvirksomheten "Re-Design A/S" har en gruppe systemutviklere som utøver re-design aktivitetene. For å få verktøystøtte til TDD, bruker utviklerne Eclipse med det integrerte åpen kildekode test-rammeverket JUnit. Brukerne støtter seg på noe tilsvarende CRC-kort til å skrive sine funksjonelle tester på.

3.1.1 "Kod-og-fiks"

"Kod-og-fiks" var basismodellen som opprinnelig ble brukt i de tidlige dager av systemutviklingen (brukes trolig framdeles), og besto av to enkle steg:

1. Skriv litt kode.
2. Fiks problemene i koden.

Denne modellen hadde tre hovedproblemer[15]: For det første ble koden stadig mer ustrukturert etter som antall endringer økte, og endringenes kostnader økte med antall endringer. Dette førte til behovet for en designfase.

For det andre, til tross for godt designende programmer, var disse lite sammenfallende med de egentlige behovene. Resultatet var enten å forkaste løsningen, eller utvikle på nytt med tilhørende kostnader. Dette førte til behovet av en kravfase før designfasen.

Til slutt kom problemer med kostnader til endring av koden som skyldtes manglende forberedelser for testing og modifisering av koden. Dette førte til at man også måtte vektlegge slike faser, i tillegg til planer og utarbeidelse av rutiner til testing og videre utvikling.

3.1.2 Den stegvise modellen og fossefallsmodellen

Den stegvis modellen kom i 1956 som en følge av problemene i "kod-og-fiks" modellen. Idéen var at programvare skulle utvikles i suksessive steg[15].

Fossefallsmodellen omtales også som "faseorientert", og har siden 70-tallet vært en innflytelsesrik modell. Den hadde to forbedringer til den stegvise modellen[15]: Den første forbedringen kom som en følge av behovet for iterasjoner mellom en fase og den foregående. Samtidig ville en forhindre iterasjoner mellom mer perifere faser fordi dette påvirket de mellomliggende fasene og førte til større ressursbruk. Den andre forbedringen var en innledende innarbeidelse av prototyping i programvare livssyklusen, via et "build it twice"-steg som gikk i parallell med kravene for analyse og design.

En del av de initielle problemene som oppsto med fossefallsmodellen, har vært løst ved å utvide den til også å dekke inkrementell utvikling, parallell utvikling, bruk av fjerde generasjons verktøy, plass for evolusjonære forandringer, formell programutvikling og verifikasjon, og stegvis validering og risikoanalyse. Selv med disse omfattende revisjoner og forbedringer, har fossefallsmodellen truffet på flere fundamentale vanskeligheter som har gitt opphav i nye prosessmodeller[15].

En hovedkilde til problemer med fossefallsmodellen er dens vektlegging av dokumenter som kriterier for ferdigstillelse av krav og designfaser. Dette er effektivt for noen typer av programvare som for eksempel kompilatorer, men fungerer dårlig for interaktive sluttbrukerapplikasjoner. Prosjekter som ikke passer inn i fossefallsmodellen feiler fordi fasene for disse kommer i gal rekkefølge. Dokumentdrevne standarder har påtvunget mange prosjekter å skrive detaljerte spesifikasjoner av dårlig forstått funksjonalitet, som etterfølges av design og implementering av løsning for feil problem[15].

Jeg mener utgangspunktet for fossefallsmodellen er en statisk verden, hvor en går fra spesifikasjon—gitt at denne er riktig, til implementasjon i en forutsigbar faseorientert prosess. Mange av idéene til Taylor[77] om "Scinetific Management" gjør seg gjeldende i denne prosessen. I tillegg til å være svært dokumentsentrisk, er modellen etter mitt syn lite egnet for brukermedvirkning utover spesifikasjonsfasen, og analysefasen.

Den mest kjente inkarnasjonen av denne modellen er, etter min mening, RUP[2]—som bygger på den generiske prosessen "Unified Process" (UP).

3.1.3 Den evolusjonære modellen

Modellen kom med bakgrunn i svakhetene ved fossefallsmodellen. Den evolusjonære modellen har også steg eller faser, men her består de i å utvide et allerede fungerende programprodukt, hvor retningen for utviklingen bestemmes av erfaringer som oppnås underveis. En annen fordel er at brukerne får sterk innvirkning i et utviklingsløp etter denne modellen[15]. Begge fordelene finner jeg igjen som sterke likhetstrekk med XP (se seksjon 3.2.1).

Modellen er ideell ved bruk av fjerde generasjons språk, eller når en kunde sier:

> "jeg vet ikke hva jeg vil ha, men jeg vet det når jeg ser det"

Utsagnet indikerer høy grad av usikkerhet, hvor usikkerhet er en typisk parameter for vurdering i forbindelse systemutvikling [58, 23].

Modellen har også sine svakheter, og er i følge Boehm[15], vanskelig å skille fra "kod-og-fiks" modellen. Videre er den basert på en urealistisk antagelse om at brukerens operasjonelle system har fleksibilitet nok til å ta høyde for ikke planlagte utviklingsretninger. Argumenter mot en slik urealistisk antagelse er[15]: Når flere uavhengig utviklede applikasjoner senere må integreres. Eller når midlertidige omgåelser på grunn av mangler i programvaren hindrer videre utvikling på et senere tidspunkt. Til slutt; i situasjoner hvor ny programvare inkrementelt erstatter et stort system. Hvis det eksisterende systemet ikke er godt modularisert, er det vanskelig å støtte en god sekvens av "brobygging" mellom gammel og ny programvare.

Jeg vil her kort gi noen motargumenter (omtales ytterligere i seksjon 3.2 og kapittel 4). En tjenesteorientert arkitektur har som mål å kunne integrere uavhengig utviklede applikasjoner, og denne tjenesteorienteringen er ikke avhengig av en gitt systemutviklingsmodell. Jeg mener at dette er anvendelsesområder som moderne systemer trenger, eller enkelt skal kunne utøkes til.

XP som metodikk planlegger for endring, men endring er her av en art som hovedsaklig treffer Boehm's andre argument mot den evolusjonære modellen.

[2]RUP—Rational Unified Prosess. Finnes på VVV som `http://www-306.ibm.com/software/awdtools/rup/`

Vi kan ha situasjoner, slik som modernisering av EIS med relasjonsdatabase og eksisterende forretningsregler, hvor nettopp ny programvare inkrementelt erstatter et system. Den eneste modulariseringen jeg da forutsetter er at databaseskjemaet betraktes som en modul.

Boehm hevder at ved å følge de ulike steg i feil rekkefølge har evolusjonære modeller feilet på samme måte som fossefallsmodellen, med konsekvenser hvor en lager kode som er vanskelig å endre før en ser på langsiktige betraktninger rundt arkitektur og bruk[15]. Dette mener jeg står i kontast til agile metodikker med utstrakt bruk av automatiserte tester og refaktorering, og spesielt for EIS hvis metodikken kan forsterkes av hensiktsmessig arkitektur. Dette kommer jeg også tilbake til.

Pelle Ehn omtaler den evolusjonære systemutviklingsmodellen[26], hvor prototyping (se seksjon 3.1.6) er et sentralt sett av aktiviteter om vi ser på en modell som en prosess. Dette er også i tråd med Sommerville's[72] argument for den evolusjonære utviklingen, som han for øvrig mener er mer effektiv enn "vannfalls-prosessen", når svakhetene oppveies med bruk av prototyping. Tilsvarende finner jeg hos de agile metodikkene som XP kan være en representant for, hvor hyppige iterasjoner og inkrementer og kjørbare systemer ligger innenfor den evolusjonære utviklingen med bruk av prototyping.

3.1.4 Transformasjonsmodellen

Ustrukturert kode som en følge av "kod-og-fiks"modellen og den evolusjonæremodellen, kan også gi seg utslag for utvikling basert på fossefallsmodellen, hvor optimaliserings-hensyn går foran hensynet til senere endringer. Transformasjonsmodellen er foreslått som en løsning. Tanken er at en skal kunne gå automatisk fra en formell spesifikasjon av et programvareprodukt, til et kjørbart program som tilfredsstiller spesifikasjonene. Svakheten med den automatiske transformasjonsmodellen er at den bare er tilgjengelig for små produkter i et fåtall begrensede områder[15].

Her trekker jeg paraleller til en nyere variant av denne modellen, MDA[3] (Modell Drevet Arkitektur), hvor man utvikler UML (Uniform Modelling Language) modeller i form av en uavhengig modell (POM), og en implementasjonsmodell (PIM). Implementasjonsmodellen skal så kunne genereres til kjørbart system, men også her er tilgjengeligheten bare for små produkter og begrensede områder.

3.1.5 Spiralmodellen

Spiralmodellen er en *risikodrevet* tilnærming til programvareprosessen. Dette i kontrast til prosesser som primært er dokumentdrevet eller kodedrevet. Spiralmodellen baserer seg på andre modeller's styrke, samtidig som den skal løse flere av deres problemer[15].

Boehms' spiralmodell[15] introduserer en systematisk måte for å mikse prototyping og tilnærminger ved spesifisering, hvilket er i tråd med Mathiassen[57]—se seksjon 3.1.7. Valg av tilnærming er basert på en analyse av risikofaktorene for det aktuelle utviklingsprosjektet som betraktes. Med spiralmodellen er systemutvikling vanligvis delt opp i et antall sykler hvor hver sykel involverer en framgang, og innbefatter samme type og sekvens av aktiviteter[3].

[3]MDA—Model Driven Architecture. Finnes på VVV som `http://www.omg.org/mda`

At hver sykel involverer en framgang, mener jeg tilsvarer den evolusjonære modellens utvidelse av et allerede fungerende programprodukt.

Spiralmodellen har utviklet seg gjennom mange år med basis i flere forbedringer av fossefallsmodellen. Den kan ta opp i seg mesteparten av andre modeller som spesialtilfeller. Videre kan den gi veiledning om hvilke kombinasjoner av modeller som passer best i en gitt programvare situasjon. Risiko vurderes basert på systemet som skal utvikles og effektiviteten av de ulike metoder og teknikker som benyttes. En sykel består av flere steg[15]:

- En typisk sykel starter med å avgjøre målet for sykelen. Dette medfører siktemål, alternativer, og innskrenkninger for produktene som utarbeides i denne sykelen.
- Hovedsaken i sykelens neste steg er å identifisere usikkerhet som kan bidra til risiko. Dette gjøres gjennom evaluering av alternativer relatert til siktemål og innskrenkninger. Til slutt inkluderer dette steget formulering av en kosteffektiv strategi for å bestemme hovedkildene til risiko. Aktiviteter her kan være prototyping, simulering, gjennomgang av brukerspørsmål.
- Det tredje steget utgjør utvikling og verifikasjon av produktet i sykelen. Hvis risikoen er høy, brukes innsats på å bestemme kilden til usikkerhet. Dette kan involvere både spesifisering og prototyping.
- Steg fire er ment for å legge planer for neste sykel.

Når alle hovedkilder for risiko er bestemt, følger utviklingen en faseorientert modell. Dette kan inkludere deling av produktet i komponenter som leveres separat. Overgangen fra en sykel til neste er basert på en gjennomgang av produktene fra nåværende sykel og planer for den neste[15].

Boehm ser likhetstrekk mellom en disiplinert prosess og XP, og hevder de følger en spiral-filosofi; begge er både risiko og inkrementdrevet[16]. En stor forskjell, etter min mening, er at XP's sykler er ekstremt mye kortere, og ikke følger noen faser med mindre man hevder at fasene er ekstremt korte. Mitt hovedargument mot spiralmodellen er at den bruker mye tid på en slags "spesifisering" av risiko, for deretter å være faseorientert. Kjørbare artefakter tidlig i prosjektet vil ikke typisk tas fram bortsett fra prototyper, og dette mener jeg kan være risikodrivende. Brukermedvirkning vil således også i denne modellen være unaturlig som en kontinuerlig del av prosjektet grunnet modellens faseorientering som kommer i store sykler.

3.1.6 Prototyping - en metode

Utviklingsstrategier gir retningslinjer for å konstruere programsystemer i en prosjektsituasjon. En studie fra USA i 1979 konkluderte med at fiaskoene med systemutviklingsprosjekter ofte hadde årsak i utilstrekkelig oppgavebeskrivelser fra brukersiden. Ett ytterpunkt i tilnærming som kan løse problemene er å bruke eksperimentelle metoder (prototyping) som grunnlag for å utvikle anvendbar programvare[3]. Dette som en kontrast til formalisering av aktiviteter gjennom formelle metoder i tidlige faser for å oppdage "problematiske" krav.

Idéen med prototyping kom som en reaksjon på problemene med tradisjonelle framgangsmåter ved programvareutvikling[3]: Ofte er det først når systemet

er ferdig at brukerne og deres organisasjoner er i stand til å eksplisitt formulere krav. Utviklerne vil foretrekke å utsette endelig spesifikasjon til konstruksjonsprosessen er i gang. Gjensidig koordinering er påkrevd mellom utviklere og brukere gjennom hele utviklingsprosessen, og begge parter lærer av hverandre. IT-leverandører med stor avstand til brukerne og deres organisasjoner har tradisjonelt kun akseptert "endelige spesifikasjoner", for så å levere "ferdige" implementasjoner tilbake.

En måte å takle slike problemer i tradisjonell programvareutvikling på, er ved å ha en evolusjonær utviklingsprosess med utstrakt bruk av prototyper.

I systemutviklings-prosessen kan prototyper ha ulike oppgaver. Den første er å vise en driftsklar modell for deler av det framtidige systemet. Den andre er som en basis for å diskutere vanskeligheter, oppklare problemer, forberede valg mellom utviklere, brukere og ledelse. Videre kan den være et grunnlag for neste prototype eller neste versjon av systemet som skal lages. En siste oppgave kan være i forbindelse med brukergrensesnitt ved å vise funksjonalitet og oppførsel mest mulig lik det endelige systemet. Av prototyper finner vi tre ulike *oppgaver*[3]:

Ekte prototype er et provisorisk driftsklart system, og utvikles parallelt med modellen av informasjonssystemet. En slik prototype brukes vanligvis for brukergrensesnitt eller for å vise utvalgt funksjonalitet slik at eventuelle problemer kan avklares. Referansegruppen i Forsvaret (se appendiks A) bruker en variant av denne, men i stedet for å kaste prototypen, anvendes elementer direkte i det videre arbeidet.

Testbenk prototype designes hovedsakelig for å klargjøre spørsmål relatert til konstruksjon sett fra utviklernes side.

Pilotsystem har vi når prototypen er blitt kjernesystemet i applikasjonsområdet. Da opphører klare skiller mellom prototypen og systemet. Når prototypen har oppnådd et gitt nivå av "forfining", installeres denne som et pilotsystem og utvikles derfra i sykler. Tilveksten i de ulike syklene skal utelukkende være fra brukernes prioriteringer. Jeg mener denne type prototype ligger tett opptil den kodedrevne prosessen og planleggingsspillet som vi finner i XP. Hos referansegruppen i Forsvaret (se appendiks A) finner jeg tilsvarende anvendelse.

Vi kan avdekke flere *typer* av prototyping relatert til ulike mål med prototyping, i forhold til hvor kjent problemområdet er[3]:

Utforskende prototyping brukes når et problem er uklart. Spesielt viktig når en skal se på ulike valg i design. Her har vi ulike varianter av ekte prototype. Utforskende prototyping er spesielt viktig i prosjektsituasjoner der utviklere og brukere tilhører forskjellige organisasjoner/firma.

Eksperimentell prototyping er en form for prototyping som fokuserer på teknisk implementasjon av et utviklet mål. Brukerne kan da bli i stand til å ytterligere spesifisere sine idéer om hvilken datastøtte som det er behov for. Utviklerne på sin side gis en vurdering på hvor velegnet løsningen så langt er. Essensielt her er kommunikasjonen mellom brukerne og utviklerne. Både ekte og testbenk prototyper klassifiseres inn her.

Evolusjonær prototyping er en metode innenfor modellen evolusjonær systemutvikling. Korte utviklingssykluser er målet, og da er det mest effektivt å eliminere forskjellen mellom prototypen og applikasjonssystemet ved å utvikle pilotsystemer. Her skifter utviklernes rolle til tekniske konsulenter som arbeider i tett samarbeid med brukerne. Dette er etter min mening svært likt det vi finner i XP (se seksjon 3.2.1), og slik referansegruppen i Forsvaret (se appendiks A)

ofte anvender prototyper som en naturlig del av prosessen.

Horisontal og vertikal prototyping. Vi kan se på systemutvikling som design og implementering av en rekke lag. Fra brukergrensesnitt laget og ned til basislaget, normalt bestående av operativsystemet eller database spørrespråket. Inndelingen i horisontal og vertikal prototyping kan være nyttig i denne sammenhengen: I horisontal prototyping lages bare spesifikke lag av systemet, for eksempel brukergrensesnittet med sine skjermbilder og menyer, eller funksjonelle kjernelag slik som database transaksjoner. Vertikal prototyping velges når en del av målsystemet implementeres fullstendig gjennom alle lagene. Denne teknikken er egnet når systemets funksjonalitet og implementasjonsvalg fortsatt er åpne. Dette er vanligvis tilfelle ved pilotsystemer. Vertikal prototyping ligger nært opp til XP's tilnærming for design av arkitektur hvor en brukerhistorie implementeres gjennom alle lag.

Siste kriteriet for å klassifisere prototyper på, er ved å se *sammenhengen mellom prototyper og applikasjons systemet*[3]:

Prototyper inkrementeres for å produsere applikasjonssystemet. Arbeidsstasjoner og meget høynivå programmeringsspråk (VHLL) utvisker det tekniske skillet mellom prototyper og applikasjonssystemene. En gradvis overgang fra prototype til system er av den grunn mulig.

En prototype er en del av applikasjonssystemets spesifikasjon. Applikasjonssystemet bygges på basisen av en akseptert prototype. Her er prototypen bare brukt som spesifikasjon og ikke som en byggestein i applikasjonssystemet. Den er en såkalt "throw away". Prototyping-en består i å modellere og demonstrere utvalgte aspekter som brukergrensesnitt og funksjonalitet ved systemet. Feiltoleranse og effektivitet sees bort fra for å kunne utvikle prototypene så hurtig som mulig.

Prototyper som brukes kun for å klargjøre problemer. Dette er en ytterlighet med prototyping hvor intensjonen ikke er å bygge et system, men tilegnelse av kunnskap.

3.1.7 Miks av spesifikasjon og prototyping

Større systemutviklingsprosjekter burde bruke kombinasjoner av tilnærminger til systemutvikling basert på spesifikasjoner og prototyper[57].

Med utgangspunkt i *spesifikasjoner*, transformeres disse gjennom en rekke faser. For hver fase blir nye spesifiseringer eller beskrivelser produsert gjennom transformering av beskrivelsene i forrige fase. Bruk av spesifisering har tett sammenheng med en analytisk bruksanvisning. Systemutviklere rådes til å ta fordel av abstraksjon for å redusere kompleksitet. Tilnærmingen har noen fundamentale begrensninger. Primært støtter den seg på tilgjengelig informasjon, og forutsetter videre at denne er riktig. En slik forenkling begrenser hvordan aktørene i organisasjonen kan kommunisere og lære om det framtidige systemet[57]. Jeg mener at spesifikasjoner kan i en agile kontekst sees som tester som skrives først, og som nevnt for spiralmodellen i form av risiko som først vurderes.

Når systemutviklingen er basert på design, implementering, og evaluering av *prototyper* som modellerer deler av det totale systemet, vil systemets design være basert på mer eller mindre realistisk bruk av prototyper. Denne tilnærmingen har en tett sammenheng med en eksperimentell utvikling. Systemutviklere rådes til å høste fordeler av mulighetene til å lære gjennom eksperimenter.

Prototypetilnærmingen er et konstruktivt tilsvar på problemene og svakhetene med spesifikasjonstilnærmingen, men reiser andre typer av problemer. Ved bare å betrakte enten prototyping eller spesifisering binder vi oss opp i et alt for enkelt rammeverk[57]. Her trekker jeg paralleller til både spiralmodellen, og som vi senere skal se, XP som bruker prototyping som viktige elementer i sine modeller.

Situasjoner for systemutvikling er på den ene siden beskrevet som grad av kompleksitet og usikkerhet som utviklerne møter. Grad av kompleksitet representerer mengden av relevant informasjon, som er tilgjengelig i en gitt situasjon for å ta beslutninger om designvalg. På den andre siden er graden av usikkerhet representert som tilgjengelighet og pålitelighet av informasjon som kan være relevant for samme formål[57].

Tilnærming i systemutvikling er på den andre siden karakterisert av termer som bruksanvisning og type utførelse. Bruksanvisningen gir råd om hvordan utviklerne skal behandle informasjon for kunne ta designvalg. Ytterlighetene her er enten en analytisk eller eksperimentell oppskrift. Når systemutviklerne opererer etter en analytisk oppskrift forenkler de tilgjengelig informasjon gjennom abstraksjoner. Når oppskriften er eksperimentell, lærer de av erfaring og genererer derved ytterligere informasjon. Type utførelse brukes for å beskrive og dokumentere designforslag og beslutninger. Spesifikasjoner kan brukes som abstrakte beskriver av egenskaper og oppførsel for et system. I motsetning til dette kan forskjellige modeller i form av prototyper brukes for å illustrere konkret oppførsel for systemet[57].

I et enkelt rammeverk basert på forskjellen mellom spesifisering og prototyper har det vært tatt for gitt at en analytisk oppskrift og bruk av spesifikasjoner henger sammen, og at det samme er tilfelle med bruk av eksperimentering og prototyper. Likeledes har det vært tatt for gitt at spesifisering er mest effektivt når kompleksiteten er stor, mens eksperimentering skulle være mest effektivt i forbindelse med usikkerhet. Et fundamentalt premiss for en slik teori er at kompleksitet og usikkerhet er uavhengige karakteristikker i en designsituasjon. Det er vanskelig å finne bevis som støtter et slikt synspunkt[57].

I en *analytisk tilnærming* må vi stole på en forenklet imaginær verden, og som en konsekvens innføres nye kilder til usikkerhet avhengig av graden av dårlig sammenheng med den komplekse virkeligheten. For en *eksperimentell tilnærming* hvor vi produserer informasjon etter som vi eksperimenterer, og som en konsekvens innføres nye kilder til kompleksitet[57].

En antagelse er at kompleksitet og usikkerhet er innbyrdes relatert. Som en konsekvens finnes ingen enkel måte å relatere en oppskrift og utførelsen av denne. Når vi betrakter kompleksitet og usikkerhet som avhengig av hverandre, kan vi ikke redusere den ene uten at det får konsekvenser for den andre. Dette er uttrykt i følgende basis prinsipp for systemutvikling, "Prinsippet om begrenset reduksjon"[59]

> Når en stoler på en analytisk adferd for å redusere kompleksitet introduseres nye kilder for usikkerhet som krever en eksperimentell motvekt. Samsvarende, når en stoler på eksperimentell adferd for å redusere usikkerhet introduseres nye kilder for kompleksitet som krever en analytisk motvekt.

Prinsippet beskriver relasjonen mellom situasjonen og oppskriften som an-

vendes. I stedet for å betrakte ulike framgangsmåter, foreslås i stedet at anvendelsen for utføring, enten den er som spesifikasjoner eller prototyper, krever en viss blanding av en analytisk og eksperimentell tilnærming. Effektiv systemdesign krever en systematisk innsats for å kombinere både en analytisk og en eksperimentell tilnærming. Dette er vist gjennom eksperimenter. Spiralmodellen ansees som et nyttig rammeverk for å kombinere tilnærmingene[57]. I seksjon 3.2.1 vil jeg argumentere for at også XP kombinerer disse tilnærmingene, og stiller spørsmål ved om XP's fokus på enkelhet kan svekke anvendelsen av prinsippet.

3.2 Agile metodikker

Siden årtusenskiftet har idéer om en smidigere systemutviklings-prosess som vektlegger individuelle metodikker som Crystal, eXtreme Programming (XP), og annen tilpasningsdyktig programutvikling, stadig oftere fått innpass. Samtidig skaper de heftige debatter[46].

Fra en kaotisk prosess med "kod-og-fiks" endte vi opp i systemer med mye feil og omfattende testing og vanskelig vedlikehold i etterkant. Dette ledet over til prosesser dominert av tunge byråkratiske metodikker som ikke har vært særlig effektive og populære, og som har avstedkommet mange kostbare prosjekter. Nå ser vi en framvekst av en gruppe lettvekter metodikker som har kommet de siste årene[35]. Den voksende interessen for lettvektsmetodikker indikerer en motvekt til både byråkratiske metodikker og full frihet til å kode i vei ala "kod-og-fix". Denne typen metodikker er tilpasningsdyktige av natur, promoterer en utflatet kostkurve, og setter menneskene i fokus. De skiller seg fra tradisjonelle engineeringsmetoder ved at de er mindre dokument-orienterte, og krever mindre tradisjonell dokumentasjon for en gitt oppgave. De vektlegger koding i mye sterkere grad. Kode som er sluttproduktet—programvaren. Naur[61] hevder at programmering (design og programmering) er teori bygging, hvilket gir, om Naur har rett, et svært tungtveiende argument for vektlegging av kode og kjennskap til denne.

Sammen med utbredelsen av lettvekter metodikker, er det interessant å observere at tunge metodikker som RUP og MDA, begge har kommet i agile versjoner. Cockburn[24] foreskriver slike tilpasninger av metodikker. Lettvekterutgavene av etablerte metodikker er ikke et bevis på at den agile retningen er best, men vitner om at retningen har noe bra ved seg. Jeg tolker framveksten av agile metodikker og "agileringen" av etablerte prosesser som en erkjennelse av at noe har vært galt, og kan gjøres bedre. To andre tolkninger av "agileringen" av etablerte prosesser mener jeg kan være av kommersiell art: For det første ser en leverandør at deres produkt mister markedsandel til agile metodikker. For det andre ønsker en leverandør å øke markedet slik at mindre prosjekter også bruker varianter av deres metoder. Om jeg i tillegg framhever hvilke tungvektere innen systemutvikling som skrev "Agile manifesto"[12][4] styrker det ytterligere mitt syn i behovet for endrede prosessmodeller og vektlegging av smidighet og enkelhet.

[4]De 17 var; Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas.

Omfattende byråkratiske metodikker mener jeg ikke er noe middel for *raskere*, *bedre*, *billigere*, og *enklere* framskaffelser av enterprise informasjonssystemer, og spesielt hva angår modernisering. Dette avhenger dog av prosjektets størrelse, type, og kritikalitet. Større prosjekter og prosjekter med høy grad av kritikalitet fordrer tyngre metodikker. Prosjekter som tilhører andre typer enn enterprise informasjonssystemer, som kompilatorer og operativsystemer, ligger også utenfor hva jeg her tar stilling til. Et eksempel fra et systemutviklingsprosjekt illustrerer resultatet etter bruk av en tung byråkratisk metodikk: Et utlånssystem skulle utvikles for en stor bank i Singapore, og 50 utviklere jobbet i 15 måneder[46]:

> "Et stort velkjent systemintegrasjons-firma hadde brukt to år på prosjektet, for så å erklære at det ikke var mulig å fullføre. De produserte artefaktene var 3500 sider av bruker-historier, en objektmodell på flere hundre klasser med tusenvis av attributter uten metoder—og ingen kode."

Prosjektet ble senere reetablert, alt tidligere arbeid forkastet, og fokus ble rettet mot sluttproduktet med bruk av en agile metodikk. Prosjektet kom denne gangen i havn—på kortere tid, og med færre ressurser[46]. At prosjektet kom i havn den andre gangen *kan* ligge i årsaker som at utviklerne nå var dyktigere, men jeg antar at sannsynligheten for dette er liten i og med at prosjektet kunne karakteriseres som prestisjefylt.

For bedre å klargjøre skillet mellom tradisjonelle og agile prosessmodeller, kan vi sette disse opp mot hverandre[24, 35]:

- I gamle modeller er byråkrati til hindrer for dynamikk, mens agile prosesser er mindre byråkratiske og mere dynamiske.

  Vi har den senere tid også fått varianter av tradisjonelle metodikker som er mindre byråkratiske og mer dynamiske, men da i form av agile utgaver.

- I gamle modeller er det unødvendig mye fokus på artefakter som ikke forbedrer produktet, mens de agile har hovedfokus på produktet som skapes, og mindre vektlegging på dokumenter, sirlige tegninger og modeller som ikke fremmer det egentlige produktet.

  Videre argumentasjon er lagt til seksjon 3.2.3.7.

- I gamle modeller tar endringer for lang tid og koster for mye, mens de agile modellene ønsker å maksimere produktivitet og promoterer en utflatet kostkurve for endringer sent i utviklingsløpet, og senere ved vedlikehold.

  Alle ønsker å maksimere produktivitet, men bare den agile retningen promoterer en utflatet kostkurve sent i livsyklus-håndteringen. De tradisjonelle metodikkene har sin styrke ved store prosjekter og prosjekter med meget høy seremoni—men det er slike store prosjekter som ofte ikke makter å levere i henhold til Chaos-rapporten[73].

- I de gamle modeller; når produktet tilgjengeliggjøres for bruk, dekker det ikke dagens behov og ønsker, mens den agile tilnærmingen fokuserer mere på på iterativ og inkrementell utvikling, samt å forbedre kommunikasjon og koordinering.

Spiralmodellen er bedre her enn andre tradisjonelle modeller, ved at inkrementer eller mindre komponenter tas fram i store sykler og integreres i ettertid. Et problem med spiralmodellen, og generelt for tradisjonelle modeller, er at vi heller ikke her oppnår et kjørbart resultat som kan anvendes før prosjektet går inn i en siste sykel. Men til gjengjeld skal det ikke kunne komme ut et produkt som ikke langt på vei dekker dagens behov. I så tilfelle har risikohåndteringen feilet.

- Når testing ikke er en integrert aktivitet i hele prosessen, blir det en lavstatus-aktivitet som nybegynnerne må ta hånd om, mens testing inngår som en integrert aktivitet i de agile prosessmodellene.

  Når jeg trekker parallellen til RUP, som en modernisert tradisjonell faseorientert metodikk, med en disiplin for testing, vektlegges tester hovedsaklig i konstruksjonsfasen. Etter min mening er rollen "tester" den som utfører denne lavstatus-aktiviteten (eller disiplinen).

- I tradisjonelle prosesser dreies fokus vekk fra dyktighet, kunnen og brukermedvirkning, mens de agile prosessene gjør det artig for dyktige utviklerne og superbrukere å jobbe sammen i prosjektet. Menneskene er i fokus, i motsetning til tradisjonelt fokus på roller.

  Desto mindre kunnskap og faglig dyktighet et team innehar, jo mere nytte kan de dra ut av dokumentbaserte prosesser med mange detaljerte steg som må gjennomføres for om mulig komme i mål[46]. I tillegg til manglende kunnskap og dyktighet, gir fravær av brukermedvirkning størst risiko for prosjektet[74]. Videre er agile metoder orientert mot *mennesker* og mindre mot prosesser som skal hjelpe alle, uansett kunnskapsnivå. De agile prosessene skal fremme teamet bestående av gode utviklere og gode brukere med kompetanse om problem-området[35].

Enhver metodikk har sitt "sweet spot", hvilket kan anvendes som identifikasjon på om metodikken er agile[24, ss. 178-179]. Tilsvarende kan det brukes for å identifisere egnede metodikker til ulike former for utviklingsløp. Agile utvikling har tre karakteristikker: For det første er perspektivet erkjent og akseptert som økende nivåer av uforutsigbarhet i en turbulent økonomi. For det andre samarbeidende verdier og prinsipper, og til slutt—en så vidt tilstrekkelig metodikk[46]. Dette kan lett lede til motforestillinger mot agile utvikling. Av disse kan en årsak ligge i metodikkenes bekjentgjøring av usikkerhet[47]:

> "I store bedrifter er det ofte mer akseptabelt å ta feil enn å være usikker".

> "En del av motforestillingene til agile tilnærminger er at talsmannen erklærer at prosjekter har usikkerhet når de har det. Mange mennesker ønsker ikke å høre sannheten, de vil heller høre løgnen om at vi vet eksakt hva som kommer til å skje."

En annen årsak til motforestillinger kan ligge i at agile utvikling vektlegger en god relasjon mellom enterprise og IT, hvilket medfører at denne typen metodikk fungerer dårlig opp mot organisasjoner som fra før ikke fungerer[47].

En vesentlig forskjell mellom de tradisjonelle engineeringsdisiplinene som gjelder for konstruksjon av broer og skyskrapere—og systemutvikling av

Figur 3.1: Planer og metodikk fra Dilbert.

programvare, er at førstnevnte i mye større grad trenger planer og tegninger for å konstruere de *forutsigbare* artefaktene riktig første gangen. En skyskraper som er halvveis bygd lar seg ikke lett rive for så å prøve på nytt. Man kan på forhånd i stor grad forutsi hvordan byggverket skal ende opp. For enterprise programvare som skal supportere enterpriser og krav i stadig endring, hvor forutsigbarhet ikke er en grunnleggende forutsetning, og hvor kreativitet spiller en viktig rolle, trenger vi andre tilnærminger enn å følge en fastsatt plan[35, 24, 13, 14, 40]. Vi trenger en prosess som gir kontroll over manglende forutsigbarhet for å unngå ukontrollert kaos, og Fowler[35] foreslår at en tilpasnings-orientert prosess følges når: Kravene er usikre eller endres, vi har ansvarlige og motiverte utviklere, kundene forstår og er villige til å involvere seg, omfanget ikke er endelig fastsatt på forhånd, og teamet ikke er for stort. En smidig og lett prosess har i tillegg større sjanse for å bli fulgt enn byråkratiske prosesser.

I følge Fowler[35], er de agile metodene mer rettet mot *tilpasning* enn å følge en fastsatt plan, hvilket fremmer *endring* i motsetning til å motsette seg endring ved blindt å følge en plan. Highsmith fremmer tilsvarende syn gjennom å sammenligne lederen for en militær kamp og en agile prosjektleder[46]:

> "Just as winning is the primary measure of success for a battlefield commander, delivering customer value (however the customer defines it) measures success for the agile project manager. Conformance to plan has little meaning in either case. If we want to be agile, we have to reward agility."

Med hensyn på planlegging, planlegger de militære for kamp, men vet at planene vil endres, akkurat som endring av planene er typisk for agile metodikker. Et humoristisk eksempel på vektlegging av planer er vist i figur 3.1[5].

Re-design vil også ha en viss grad av utforskning, og en hver aktivitet som har grader av utforskning, som Highsmith uttrykker det, kan ikke basere seg på målinger som er egnet i en definert prosess, som for eksempel CMM[6], men må støtte seg på å måle resultatet, eller vinne slaget, i form av produktet kunden vil ha[46]. Re-design er et skritt på veien. Videre utvikling og endring kommer i de etterfølgende rundene. Og velger vi først en agile tilnærming, er det etter

[5]Dilbert-stripen er gjengitt med tillatelse fra 'PIB Copenhagen A/S'
[6]CMM—Capability Maturity Model

min mening hensiktsmessig å videreføre samme tilnærming for systemets videre livssyklus.

Når Naur[61] argumenterer for at programmering (design og programmering) er teoribygging, kan dette sprike med re-design? Hva når vi tar utgangspunkt i gamle systemer med et eksisterende databaseskjema og forretningsregler, og kanskje bare brukernes erfaringer med det eksisterende systemet, og vi ikke får overført innebygd kunnskap fra de opprinnelige utviklerne? Har vi da mistet dette aspektet ved at programmering er teoribygging? Jeg mener at vi prøver å bygge teorien slik at den endelig sitter. Men idéelt sett skal de samme, eller et utvalg av de samme, utviklerne som utviklet det opprinnelige systemet, stå for moderniseringen sammen med et utvalg superbrukere. Derved vil den iboende kunnskapen enklere overføres.

Med utgangspunkt i agile tilnærminger har jeg valgt XP og TDD som representanter. I de etterfølgene seksjonene beskrives disse, før jeg argumenterer for elementer til bruk i en referansemetodikk som kan anvendes for re-design.

3.2.1 eXtreme Programming (XP)

Innen genren Agile metodikker finner vi blant andre Scrum[7], Alistair Cockburn's Crystal[8], og XP[13]. Jeg velger her XP som et eksempel på en konkret agile metodikk fordi den har klart mest interesse blant de agile tilnærmingene. XP er i hovedsak utviklet av Kent Beck, Ward Cunningham, and Ron Jeffries[46].

XP beskriver en tilnærming til utvikling som kombinerer beste praksiser brukt av suksessrike utviklere, som tidligere ble nedlesset av massiv litteratur fra ulike systemutviklingsprosesser[13]. Interessen for XP kommer fra grasrota, fra utviklere og testere som ikke lengre ønsker overdreven formalisme—ofte forvekslet med disiplin, ikke ønsker overdreven dokumentasjon, metrikker, og hindrende prosesser. De har funnet alternative måter å levere kvalitets programvare raskere og mer fleksibelt[46]. For informasjonssystemer vil ikke målinger som veies opp mot metrikker kunne jamføres med en brukers "tommel opp eller tommel ned". Det spiller liten rolle om en måling er god hvis ikke kunden er fornøyd. På den annen side ser jeg dilemmaet med at ulike brukere har ulike oppfatninger av hva som er bra.

XP er en lettvekts og smidig metodikk for små til mellomstore grupper av utviklere med normale ferdigheter—sammen med brukere som representerer enterprisen, for å framskaffe kjørbar programvare i en kontekst hvor kravene er skiftende eller uklare. XP lover å redusere risiko i systemutviklingsprosjekter, forbedre responderingen til endringer i virksomheten, forbedre produktiviteten gjennom hele livsløpet til et system, og å gjøre det gøy å utvikle programvare i team[13]. Dette mener jeg klassifiserer XP inn i en evolusjonær modell med bruk av prototyping slik Budde[3] beskriver. Videre promoteres en utflatet kostkurve, hvilket generelt er mulig for agile prosesser, som kan oppnås ved bruk av objekter som nøkkelteknologi, kombinert med enkelt design, automatiserte tester, og til slutt; mot og erfaring til å modifisere[13].

XP tar utgangspunkt i gode praksiser og gjør disse ekstreme i den forstand at om noe er bra, så blir dette noe enda bedre ved å ta det til det ekstreme. Disse praksisene er avhengige av hverandre, og for å være ekte XP kan ingen

[7] http://www.controlchaos.com
[8] http://alistair.cockburn.us

praksis utelates—hvilket gir disiplinering. Og disiplin er viktig for å gi balanse til de agile metodikkene[16]. XP's ulike *praksiser* og hva de bygger på er[13]:

Par-programmering Gitt at *kodegranskning* er bra. I XP foreskrives par-programmering for kontinuerlig kodegranskning (forutsetter at begge programmerene vet sine oppgaver og ikke misforstår sitt ansvar slik vi ser i figur 3.2[9]).

Testing Gitt at testing er bra. I XP foreskrives at alle tester hele tiden (enhets-testing). Også brukerne skriver prosa for funksjons-tester, som deretter implementeres (se seksjon 3.2.2).

Design Gitt at design er bra. I XP inngår design i daglige oppgaver gjennom *refaktorering* (se seksjon 3.2.3.4).

Enkelhet Gitt at enkelhet er bra. I XP skal systemet alltid dreies mot det enkleste designet som støtter aktuell funksjonalitet ("the simplest thing that could possible work").

Arkitektur Gitt at arkitektur er viktig. I XP inngår konstant definering og forbedring av arkitekturen gjennom bruk og kommunikasjon av *metaforer*.

Integrasjons-testing Gitt at integrasjons-testing er viktig. I XP skal man integrere og teste flere ganger om dagen.

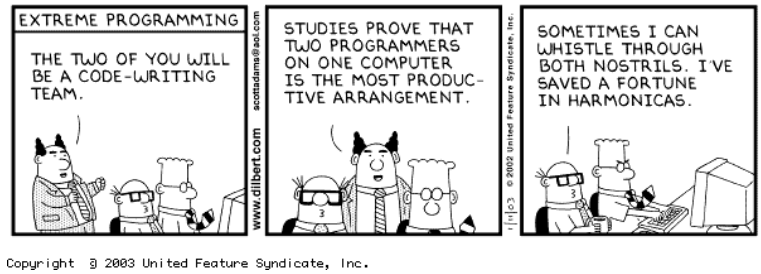
Korte iterasjoner Gitt at korte iterasjoner er viktig. I XP gjøres iterasjonene ekstremt korte—minutter og timer, gjennom sitt *planleggingsspill*, hvor planer forventes å bli endret.

Ekstreme praksiser til tross, på ledelsessiden er XP hverken ekstrem i retning av sentralisert kontroll, eller at alt er opp til hver enkelt. XP forfekter middelveien på dette området gjennom en desentralisert beslutningstaging. Egne eksplisitte disipliner for prosjekt-styring (ala RUP), er fraværende. Om XP skulle brukes for store team hvor eksplisitt prosjekt-styring ville vært påkrevd, ville metodevekten måtte økes for dette. I de tradisjonelle metodikkene beskrives ikke ledelsesforankring eksplisitt, men vil typisk være prosjektleders ansvar å sørge for slik forankring gjennom prosjekt-oppdraget. Tilsvarende vil det være for XP. Ledelses-forankrig er like fullt en viktig faktor for at prosjekter skal lykkes[74], og et prosjekt som igangsettes, enter det er nyutvikling eller for å modernisere eksisterende system, må være forankret i ledelsen hvis gjennomføringen er viktig for virksomheten.

Med XP består aktivitetene forbundet med produktutvikling av *planlegging*, *testing*, *utvikling*, *design*, og *tilgjengeliggjøring*[13]. Disse aktivitetene kan ikke forveksles med fossefallsmodellens faser, hvor en fase avsluttes før neste kan påbegynnes. XP har som nevnt meget korte iterasjoner, og aktivitetene vil av den grunn hurtig skifte fram og tilbake. Det kan også være at aktiviteten for tilgjengeliggjøring, helt eller delvis overlates til en integrasjons-tjener som kontinuerlig tilgjengeliggjør produktet (programvaren) i sin gjeldende tilstand eller versjon. Tilsvarende "kontinuerlig integrasjon" som gjelder for testing.

I tillegg til sine praksiser og aktiviteter har XP fire verdier; *Kommunikasjon*, *Enkelhet*, *Tilbakemelding*, og *Modighet*. XP forsøker å holde den riktige

[9]Dilbert-stripen er gjengitt med tillatelse fra 'PIB Copenhagen A/S'



Figur 3.2: Misforstått par-programmering

kommunikasjonen flytende ved å benytte aktiviteter som fordrer kommunikasjon mellom utviklere, kunder, og ledelse, slik som muntlig tale, testing, par-programmering, og estimering. Enkelhet er ikke alltid enkelt. Men XP gjør et veddemål om at det er bedre å gjøre en enkel ting i dag og betale litt ekstra senere hvis det må endres, enn å gjøre en komplisert ting i dag som kanskje aldri skal brukes. Enkelhet og kommunikasjon har en iboende gjensidighet ved at mere kommunikasjon klargjør hva som må gjøres og ikke gjøres, mens jo enklere systemet er jo mindre er det å kommunisere om[13]. Her mener jeg vi ser et annet syn på verden enn hva vi så i forbindelse med miks av spesifikasjon og prototyping i "Prinsippet om begrenset reduksjon" hos Mathiassen.

På den ene siden, gitt at enkelhet og kommunikasjon har en iboende gjensidighet, kan holdbarheten av Mathiassen's prinsipp svekkes. På den andre siden, at altså en slik gjensidighet ikke eksisterer, og Mathiassen's prinsipp fortsatt står sterkt, vil jeg påstå at XP er en metodikk som kombinerer tilnærmingene i det gitte prinsipp. XP har tester som spesifikasjon, etterfulgt av kjørbar kode re-faktorert til enklest mulig kode-design (pilotsystem prototype) i korte iterasjoner. Eller er det slik at antagelsen om kompleksitet og usikkerhet som innbyrdes relatert, bare har en motvekt i enkelhet og kommunikasjon? Uansett må ikke dette forstås dithen at usikkerhet og utforskning ikke lengre gjelder. Vi tar bare i bruk en annen utradisjonell tilnærming.

Ulike praksiser og verdier som jeg mener kommer til anvendelse for re-design av EIS, beskrives i seksjon 3.2.3.

3.2.2 Test Drevet Utvikling (TDD)

Test Drevet Utvikling (TDD[10]) er ikke en metodikk, men snarere en teknikk som brukes blant annet innen XP, hvor TDD er én av kjerne-praksisene[53]. Som for XP hører denne inn i en evolusjonær tilnærming[7]. TDD snur opp ned på tradisjonell utvikling. I stedet for å spesifisere, designe, og skrive funksjonell kode først—og så teste i ettertid, skrives testene først, etterfulgt av funksjonell kode. Dette gjøres i meget små steg. Jeg mener vi kan se slike tester som først skrives som en alternativ form for spesifikasjon som vi så i "Miks av spesifikasjon og prototyping" i seksjon 3.1.7.

Målsetningen til TDD kan være todelt, avhengig av perspektivet. På den ene siden kan målet være spesifisering og ikke validering. Teknikken blir da en

[10]TDD—Test Driven Development

tilnærming til design av det funksjonelle systemet[7, 14]. På den andre siden kan vi se TDD som en programmeringsteknikk, hvor målet er ren kode som virker, og som kontrollerer gapet mellom beslutning og tilbakemelding. Begrunnelser for teknikken er[14]:

- Utviklingen er forutsigbar med kode som alltid er ferdigtestet, slik at man vet når man er ferdig.
- Det muliggjør læring fra koden, hvilket er i tråd med "Programming as Theory Building"[61].
- Programvaren blir satt pris på av brukerne ved at disse er med å skriver historiene som systemet skal dekke, og lager (ikke koder) de funksjonelle testene som skal dekke systemet.
- Styrker team-følelsen ved at utviklerne og brukerne gjensidig stoler på hverandre.
- Det er artig å utvikle.

TDD er en kombinasjon av Test Først Utvikling (TFD[11]) og refaktorering. TFD foreskriver først å skrive testen, for deretter å skrive kun nødvendig produksjonskode for å tilfredsstille testen. Deretter re-faktoreres koden, samtidig som testen tilfredsstilles[7].

Når vi skriver testen først oppnår vi en iterativ bevisbasert prosess til kodingen, og unngår mange problemer som kan oppstå ved å skrive testen i ettertid; testen påvirkes av koden, koden oppnår ikke tilstrekkelig testdekningsgrad. Dette fører i sin tur til økt risiko ved refaktorering. Og til slutt—når vi skriver testen først, blir den faktisk skrevet[53]. Dette er forøvrig i henhold til mine egne erfaringer og synspunkter.

Utviklingen drives framover gjennom automatiserte tester som utviklerne koder i forkant av hver beslutning som tas, det være seg oppgave, forretningsregel, eller designbeslutning som skal kodes. Koden kjøres og gir umiddelbar tilbakemelding etter at en beslutning er tatt[14]. Ny kode skrives bare om en test feiler, eller den må re-faktoreres ved funn av dupliserende kode[53, 14].

Sykelen som det jobbes etter i TDD er som følger[14, 7]:

1. Skriv en liten test som først ikke virker.
2. Implementer den funksjonelle koden slik at testen hurtigst mulig virker.
3. Refaktorer til slutt den funksjonelle koden og fjern eventuelle kodeduplikater. Testen skal fortsatt virke—kjøre og vise at alt er OK.

En underliggende forutsetning for TDD er tilgjengeligheten for et enhets-test rammeverk. Uten slike rammeverk i form av konkrete verktøy, er TDD i praksis umulig[7]. Forslag til verktøystøtte for teknikken er beskrevet i seksjon 5.4.3.5.

Bruk av testing vil etter min mening naturlig inngå i moderniseringsarbeidet, og beskrives i seksjon 3.2.3.3.

[11] TFD—Test First Development

3.2.3 Re-design og referansemetodikk

Ingen metodikk kan brukes på alt, og alle metodikker må tilpasses prosjektene. Min intensjon er ikke å lage en komplett "kokebok" for re-design av enterprise informasjonssystemer, ei heller å foreskrive komplett bruk av XP. I stedet ønsker jeg å argumentere for viktige elementer som kommer til anvendelse i lys av en agile tilnærming.

Å foreslå ekte XP, altså alle praksisene, kan være vanskelig å få til i første omgang. Man kan starte med noen, og utvide etter hvert[13]. For modernisering av EIS har jeg ikke tatt stilling til ekte XP, men jeg kan ikke se sterke grunner som taler i mot annet enn at min arkitektur-tilnærming, som jeg kommer tilbake til i seksjon 4.4, delvis bryter med XP's arkitektur-praksis.

Cockburns beskriver prinsipper for design og evaluering av metodikker[24], og herfra har jeg valgt ut de prinsippene som jeg finner mest relevante:

- Interaktivt, ansikt-til-ansikt er billigste og raskeste kommunikasjonskanal: Hvis produktivitet og kostnader er viktige fordres små grupper i samme rom.

  Større grupper svekker god direkte kommunikasjon. Dette er godt underbygget av Hohmann, hvor han angir ulike potensielle kommunikasjons-stier etter som team-størrelsen vokser[48]. Eksempelvis gir et team på fire bare 28 mulige stier, mens et team på 11 gir hele $11,253$.

- Ekstra metodikkelementer har en høy pris: Ved å legge ekstra elementer som midlertidige artefakter til en metodikk, kreves flere ting som skal utføres. Resultatet av økt metodevekt blir at at man trekkes bort fra kjernen av utviklingen slik at effektiviteten går ned.

- Større team krever tyngre metodikk: Over 8–12 mennesker på et team medfører at kommunikasjonen svekkes, fysisk plassering og samlokalisering vanskeliggjøres, overlapp og duplisering av arbeid økes ved at det er problematisk å vite hva andre gjør. Tyngre metodikker skal hindre dette.

- Økt tilbakemelding og kommunikasjon reduserer behovet for mellomprodukter: Tomme løfter unngås gjennom å produsere kjørbar kode i små iterasjoner som kan vises fram, og reduser team-størrelsen til at direkte kommunikasjon fungerer godt.

- Disiplin, evner, og forståelse overgår prosesser, formalisme, og dokumentasjon: Disiplin og prosess er forskjellig. Prosess er å følge instruksjoner, mens disiplin er å velge å arbeide slik at det krever konsistens.

  Evner og formalisme kan heller ikke forveksles. Utfylling av skjemaer kan ikke erstatte evner.

  Forståelse og dokumentasjon er ikke det samme. Mye kunnskap er i form av iboende kunnskap, og kan ikke erstattes av dokumentasjon.

  Tunge metodikker støtter seg på prosesser, formalisme, og dokumentasjon som viktige faktorer, hvor optimalisering er viktigere enn tilpasning til forandringer. Optimalisering foregår mer i en statisk verden uten store endringer av teknologi, arkitektur, prosesser, og problemområder.

Med utgangspunkt i Cockburns premisser, oppgavens argumentasjon for redesign, kombinert med agile metodikk-deler og teknikker, har jeg i de følgende åtte seksjoner satt opp grupperinger av elementer som jeg anser relevante innenfor min referansemetodikk. Jeg har i tillegg gruppert inn risikohåndtering som spiralmodellen bygger på, som jeg mener XP har sine strategier for. Jeg oppsummerer disse metodikkelementene i tabellen 3.2 for oversiktens skyld.

Metodikk-del	**Seksjon**
Planlegg for endring—Utflatet kostkurve	3.2.3.1
Enkelhet	3.2.3.2
Teststrategier	3.2.3.3
Refaktorering	3.2.3.4
Team-størrelse og samlokalisering	3.2.3.5
Brukermedvirkning	3.2.3.6
Dokumentasjon	3.2.3.7
Risikohåndtering	3.2.3.8

Tabell 3.2: Referansemetodikk oversikt

En målsetning med et moderniseringsprosjekt er testet og kjørbar kode som enkelt lar seg vedlikeholde videre. Hvis prosjektets størrelse skulle fordre en mer byråkratisk metodikk, kan en alternativ tilnærming være å bryte prosjektet ned i mindre deler som lar seg håndtere av lettere metodikker. Men dette introduserer andre utfordringer, og spesielt innenfor integrasjon av systemene som de nedbrutte prosjektene tar fram. Slik integrasjon ligger utenfor oppgavens skop, men generelt kan man si at muligheten for slik integrasjon er av de anvendelsesområder som moderniseringen etterstreber. Prosjekter av den typen jeg her omtaler vil lite trolig ha en høy grad av kritikalitet som setter menneskers liv i fare. Om så skulle vært tilfelle ville vi måttet vurdert ytterligere metodikkelementer, og krevd strengere kontroller og lavere toleranser. Hvis kritikaliteten er høyere enn normalt, kan vi også vurdere strengere kontroller og lavere toleranser innen de rammer som her presenteres, ved for eksempel en økning av dekningsgrad for tester av koden (se side 35), og holde team-størrelsen nede (se side 37). Objektteknologien som er vesentlig for kostkurven, kommer jeg tilbake til i kapittel 5, hvor jeg også ser nærmere på teknologien som en metodikk-forsterker.

3.2.3.1 Planlegg for endring—Utflatet kostkurve

En av de universelle antagelsene innenfor software engineering er at kostnaden med endring av et program øker eksponentielt over tid[13]. Tilsvarende finner vi for bygninger, hvor kostnad til vedlikehold over tid er høyere enn kostnaden til den initielle byggingen[18].

Hvis vi kan flate ut kostnadskurven for systemer, holder ikke lengre de gamle antagelsene om hvordan vi best skal utvikle programvare. I følge Beck[13] kan en utflatet kostnadskurve oppnås, men ikke uten videre. En nøkkelteknologi er objekter slik som Kristen Nygaard og Ole Johan Dahl beskrev på slutten av 1960-tallet. Med utgangspunkt i objekter, erfarte Beck en utflatet kurve når koden var enkel å modifisere. Og for at koden skulle være enkel å modifisere kreves flere faktorer som *enkelt design* uten overflødige

designelementer, *automatiserte tester* slik at vi har tiltro til at vi ikke kommer i skade for å endre eksisterende oppførsel til systemet. Til slutt; *Mye erfaring og mot til å modifisere designet* slik at når tiden er inne, tør vi gjøre endringer. Ambler[7] uttrykker tilsvarende syn ved at kostnadene til endring blir lavere med bruk av moderne (agile) teknikker, enn hva som har vært tradisjonen.

Gitt et slikt skifte i antagelsene vedrørende endringskostnader, muliggjør det en helt annen tilnærming til programvareutvikling. Like disiplinert som andre tilnærminger, men via andre innfallsvinkler. Så istedet for å være omstendelig med å gjøre store beslutninger tidlig i utviklingsløpet og kun små beslutninger senere, jobber vi heller ut fra en tilnærming til programvareutvikling som muliggjør hurtige beslutninger når de trengs. XP gjør et veddemål på at det er bedre å gjøre en enkel ting i dag og betale litt ekstra senere hvis det må endres, enn å gjøre en komplisert ting i dag som kanskje aldri skal brukes[13].

Ett mål med modernisering av eksisterende EIS er nettopp en utflatet kostkurve ved senere vedlikehold. Og som nevnt fordres objektteknologi, enkelt design, og automatiserte tester. Objektteknologien beskrives i seksjon 5.1, og vil være den hovedteknologien som re-designet system vil anvende. Enkelt design søker jeg å oppnå ved bruk av arkitektur som jeg mener generelt passer for EIS, og som i ettertid eventuelt kan justeres om behovende tilsier det. Slik refaktorering beskrives i seksjon 3.2.3.4, mens enkelt design i form av en referansearkitektur som kan bidra til å øke anvendelsesmulighetene omtales i kapittel 4. Automatiserte tester omtales under teststrategier i seksjon 3.2.3.3.

3.2.3.2 Enkelhet

En av XP's praksiser er å dreie mot det enkleste designet som støtter aktuell funksjonalitet ("the simplest thing that could possible work")[13]. Dette fordrer refaktorering for enkelt kode-design, som igjen fordrer det sikkerhetsnettet som automatiserte tester gir. I kapittel 4 vil jeg bruke en ferdig referansearkitektur til å forenkle oppstarten av re-design-arbeidet, samtidig som designet er enklest mulig men gir rom for aktuelle anvendelsesområder.

Høyere seremoni (strengere kontroll og lavere toleranse) i et prosjekt fordrer tyngre metodikk. I tilfelle re-design av et livs-kritisk system dukker opp, kan ekstra metodikkelementer legges til for å øke metodevekten, men dette gir økte kostnader som følge av ekstra eksplisitt innsats med feilreduksjon. Avhengig av seremonigraden, kan det være bedre i ta utgangspunkt i en lett metodikk, og prøve å strekke denne til å passe prosjektet, enn å starte med for tung metodikk.

En metodikk som XP legger stor vekt på refaktorering, klargjort for endringer, utflatet kostkurve slik at vi ikke bruker mye energi og ressurser i forkant for å hindre endringer sent i utviklingsløpet. Motivasjonen er enkel: Om framtiden viser at behovet ikke var til stede, har vi spart mye unødig arbeid. Dette er prinsipper som vi kan bruke for re-design også; vi foretar de grep som trengs innenfor en hensiktsmessig referansearkitektur og i henhold til denne metodikken, og senere behov for endringer og vedlikehold utsettes til det er grunnlag for å gjøre noe mer.

3.2.3.3 Teststrategier

Bruk av testdrevet utvikling er den beste måten for å produsere kode av høy kvalitet[14, 68]. Kode av høy kvalitet er et selvfølgelig mål for den re-designede

applikasjonen.

XP foreskriver at alle tester hele tiden (enhets-testing), inklusive brukerne som skriver prosa for funksjons-tester. Vi skal integrere og teste flere ganger om dagen—og helst i form av "kontinuerlig integrasjon" hvor kode automatisk hentes ut av et versjonsstyringssystem, kompilerer, enhetstester, integrerer, integrasjonstester, og deployerer produktet slik graden av ferdigstilling til enhver tid er[13].

Det er umulig å teste alt. Tester skrives for å hjelpe programmet til å virke og til å sørge for at det fortsetter å virke. Tester skrives for tilfeller som kan bryte sammen programmet, og i tillegg når det gir en verdi. Verdi har vi når en forventning til en test er forskjellig fra utfallet, som når en test virker som ikke skulle ha virket, og når testen ikke virker når man forventer det motsatte. Hver test er autonom, og skal ikke påvirke andre tester. Videre skal testene være automatiske, og gi hurtig svar på om alt virker eller ikke. Hurtig respons oppnår vi ved å kjøre kode, og slik respons kan vi ikke oppnå fra tegninger, beskrivelser, et cetera[13].

Det er som nevnt programmererne og brukerne som skriver tester. Programmereren skriver tester metode for metode, og for følgende omstendigheter:

- Hvis grensesnittet for en metode er uklar, skrives en test først.
- Hvis grensesnittet er klart, men implementasjonen kan være komplisert, skrives en test først.
- Hvis man oppdager muligheten for at koden som skrives kanskje ikke virker i sjeldne tilfeller, skrives en test for å kommunisere tilfellet.
- Hvis et problem eller feil senere oppdages, skrives en test for å isolere problemet. Slik dokumenteres at feilen har forekommet, og vi sikrer at samme feil ikke oppstår igjen[53].
- Hvis kode skal omskrives og man er usikker på utfallet, og det ikke finnes tester for aspektet, skrives en test først.

Kundens representanter skriver funksjonstester historie for historie. Senere implementeres historiene som automatiserte tester av utviklerne.

Tilsvarende framgangsmåte vil jeg anvende ved re-design. I tillegg trengs andre tester. Disse skal også være automatiserte og grunnleggende viktig for senere vedlikehold av systemet. En oversikt over hvilke test-typer som jeg mener er av særskilt betydning for re-design er:

- Hver enkelt forretningsregel fra det gamle systemet settes opp som en test i form av en autonom automatisert test. Hvis forretningsregelen er for omfattende, brytes denne ned i håndterbare størrelser med tilhørende tester, og hvor unionen dekker den opprinnelige regelen. For hver test skrives kode som implementerer denne delen av systemet. Enhets-testene skal til slutt ha en dekningsgrad som rommer hele den opprinnelige applikasjonen's forretningsregler. Refaktorering skjer på vanlig måte.
- Videre skriver vi tester som sikrer at de ulike arkitekturlagene virker etter hensikten. Inklusive tester som viser at databasen er tilgjengelig fra datalaget.

- Vi trenger til slutt integrasjons-tester som sikrer at hele applikasjonen virker og at lagene som allerede er testet hver for seg, også testes til å virke lagene i mellom—integrasjonstesting.
- Størst utfordring ligger i å utarbeide et godt brukervennlig, og lett vedlikeholdbart brukergrensesnitt. Ulike programmatiske deler fra aktuelle arkitektur-lag drives også her fram av tester, men den grafiske utformingen må overlates til eksperter på området. Denne delen vil også innbyrdes avhenge av de teknologivalg jeg kommer tilbake til i kapittel 5.

Om vi skal modernisere en eksisterende applikasjon eller lage en ny uten rot i en eksisterende, gir TDD et design av en kontrakt som applikasjonen skal oppfylle.

3.2.3.4 Refaktorering

Refaktorering er sentralt innen XP, TDD og andre metoder og teknikker hvor kode-design er viktig. Refaktorering er en teknikk for å restrukturere eksisterende kode i små steg uten å endre oppførselen utad, og hele tiden ha et system som virker. Forutsetningen for å kunne utføre refaktorering er at vi har tester som gjør oss trygge på at systemet også fungerer etter re-faktoreringen[31]. Her mener jeg det er like viktig å benytte et utviklingsmiljø som er tilrettelagt for refaktorering. Erfaringer som referansegruppen i Forsvaret har gjort, kan tyde på dette (se appendiks A).

Design inngår i daglige oppgaver gjennom refaktorering[13]. For re-design vil refaktorering ha korresponderende betydning. Tilsvarende som en målsetning om kode med god vedlikeholdbarhet gjennom enkelt design etter nyutvikling, refaktoreres koden og systemet under re-designen, med samme målsetning om god vedlikeholdbarhet.

3.2.3.5 Team-størrelse og samlokalisering

Prosjektenes størrelse er av vesentlig betydning. Stadish-gruppen har rapporter som sier mye om sannsynligheten for å lykkes eller ikke lykkes etter som prosjektets størrelse vokser. I "Chaos Report" fra 1994[73] anslås prosjekter over 10 millioner dollar til å ha en suksess-rate på 0%. Dette tilsier at større prosjekter bør splittes i flere små håndterbare prosjekter, hvor de hver for seg kan ha større sannsynlighet for å lykkes. Forutsetningen for å kunne splittes opp i mindre prosjekter, er at systemene som ut-går fra disse, enkelt må kunne integreres i ettertid. Problemet er at dette ikke alltid er like enkelt, og kan avstedkomme en høy ressursbruk ved komplisert integrasjon. En tjenesteorientert arkitektur (se seksjon 4.2.5), kan være en mulig løsning på slike problemer. Et prosjekts størrelse påvirker etter min mening også størrelsen på teamet. Jeg kan vanskelig forestille meg 4 personer på et 65 millioners prosjekt.

"Quantitative Software Management" (QSM) har gjort undersøkelser på hvordan team-størrelsen påvirker resultatet av et systemutviklingsprosjekt. De har bygd opp en metrikk-database med over 4000 utførte prosjekter. Resultater fra ulike undersøkelser hvor 100 000 linjer kode skulle lages viste[67]:

Team størrelse Når team-størrelsen varierte mellom 4 og 32 personer var tendensen klar; Team med fem utviklere og færre hadde i snitt 4 personer for å utføre jobben. Team med 20 utviklere og flere trengte i gjennomsnitt 32 personer til samme jobben.

Ytelse Team med gjennomsnitt på 4 personer brukte tilsammen $24,5$ personmåneder på jobben, mens team på rundt 32 personer brukte 178 personmåneder.

Timeplan Små team (rundt 4) brukte en kalenderuke mere enn større team (rundt 32).

Feil i koden Større team (rundt 32) innførte fem ganger så mye feil i koden som små team (rundt 4).

Når jeg tar metrikken "ytelse" og tallfester annerledes, gir dette en totaltid på $6,13$ måneder for 4 personer, og $5,56$ måneder for 32 personer. Om jeg forsøker å transformere dette i personutgifter (da er ikke ekstra utgifter som flere PC'er og annet utstyr, lokaler, et cetera vurdert), vil det gi—om vi lønnsfastsetter hodene til 1 million per år: 4 personer koster 2 041 667 kroner, mens 32 personer koster 14 833 333 kroner. Over 7 ganger så store kostnader med en marginal reduksjon av tid.

Med bakgrunn i at flere mennesker gir økte kommunikasjonsproblemer[48], mindre effektivitet per person, flere feil, og at økning i antall personer krever tyngre metodikker, er det ønskelig å lage teamene så små som praktisk mulig. Størrelsen bør ikke overstige 10 personer i følge [24]. Jeg mener det kan argumenteres for at normalen bør ligge enda lavere. For å støtte premissen billigere er det åpenbart at små team er billigst. For re-design spesielt vil det være lite trolig at store dyre team settes opp for å spare minimalt med tid.

Samlokalisering, i tillegg til små team, gir enklest og best kommunikasjon. Ansikt-til-ansikt kommunikasjon med hjelpemidler som tavle gir direkte tilbakemelding for alle de involverte[48, 13, 24]. Samlokalisering er også mest hensiktsmessig for at brukere som skriver tester skal få disse implementert som kjørbar kode raskest mulig, i form av programvare produsert i korte iterasjoner som umiddelbart kan utprøves. Hurtige tilbakemeldinger og kommunikasjon maksimeres gjennom at kunden er tilstede og kan ta nødvendige beslutninger. *Par-programmering*, en praksis fra XP, anser jeg ikke som den mest nødvendige praksisen i første omgang. Viktigst er det at utviklerne sitter i samme rom og kan kommunisere ansikt til ansikt. Etter som erfaring opparbeides for de andre praksisene, kan par-programmering komme inn på et senere tidspunkt, eller utprøves i korte sekvenser hvis spesielle vansker ved programmeringen skulle oppstå.

Konsekvensene av disse prinsippene kan sammenfattes til å bruke små samlokaliserte team, hvor antallet ikke økes men teamet forbedres hvis nødvendig[24]. Små team er tilstrekkelig for mindre prosjekter, og moderniseringsprosjekter er typisk mindre enn nyutviklingsprosjekter etter min erfaring. Og etter som små team lettere kan samlokaliseres, er dette også en naturlig konsekvens for prosjekter som re-designer. Og som tidligere nevnt, og vist fra metrikk-databasen; små team lager mindre feil i koden slik at kvaliteten av re-designet bedres, i tillegg til at det er mye billigere.

Et problem som kan oppstå når et team settes sammen med utviklere og bruker-representanter, er at gruppedynamikken kan lede til konflikter, som i sin tur svekker teamet. Dette er et område jeg er klar over, men allikevel velger jeg ikke å behandle det her.

3.2.3.6 Brukermedvirkning

Brukermedvirkning betraktes som en kritisk sukkess-faktor for et vellykket utviklingsprosjekt[74, 8], og dette finner jeg også som en av forutsetningene for at den agile metoden XP i det hele tatt skal kunne virke. Kontrasten er stor til tidligere praksiser og manglende brukermedvirkning, med store bestrebelser for å få dette til—spesielt innen den socio-tekniske skolen[10].

Modellmakt kan påvirke brukermedvirkningen[17], men kan oppheves av å utligne asymmetrien mellom partene (brukerne og systemutviklerne)[22]. Brukermedvirkning slik XP foreskriver vil etter min mening utviske modellmakt mellom utviklere og brukere, slik jeg mener gruppedynamikk i sterke team tenderer til, men kan gi seg utslag mellom teamets brukere og de brukerne de representerer.

Brukere og systemutviklere innehar ofte ulike oppfatninger av et problemområde[27]. Bedre løsninger fordrer at problemområdet oppfattes likt. Samlokalisering og integrering av et felles team mener jeg vil kunne bedre denne forståelsen.

En annen viktig faktor å være klar over er at brukerne ikke alltid vet hva de vil ha[20], hvilket påvirker vanskelighetsgraden av å utvikle systemer. Denne faktoren mener jeg har mindre relevans innen mitt fokus på re-design. Generelt vil jeg hevde at den evolusjonære tilnærmingen vi finner i agile metodikker, vil redusere vanskelighetsgraden gjennom korte iterasjoner og kjørbar kode.

Greenbaum og Kyng anbefaler at brukernes rolle skal være aktivt samarbeidende med utviklerne, og forutsetter en gjensidig læringsprosess[41]. Dette anser jeg som en en god begrunnelse for agile prosesser som XP, hvor nettopp slik aktiv deltagelse er et krav, og forholdene for en gjensidig læringsprosess er tilstede. På den annen side er min erfaring at brukermedvirkning på fulltid ikke er ønskelig blant brukerne (se appendiks A).

Flere har tatt til orde for bedre brukermedvirkning, og gode tilnærminger som "Priority WorkShop"[17] er foreslått. Bruker vi XP som representant for agile metodikker igjen, har XP tatt brukermedvirkning enda lengre ved at den forutsettes av metodikken, og uten at fagforeninger eller avtaleverk på-dytter slik medvirkning. Men simpelt hen fordi at det er det beste for utviklerne, sponsoren og brukerne som ønsker raskest, best og billigst framskaffelse av en applikasjon. Når kundens representant i form av en eller flere ekspertbrukere, sitter i samme rom som utviklerne, og har mandat til å ta avgjørelser, så er det fordi kostbar tid ikke skal gå tapt, men brukes på å ta fram det rette artefaktet[13]. Man kan innføre teknikker for å sikre at brukerne blir deltagere i et utviklingsprosjekt. Men om vi innefører en tilleggs teknikk må det ikke være for å tvinge igjennom brukermedvirkning, slik det tidligere var behov for, men snarere som et middel for å oppnå en raskere, bedre, billigere, og enklere prosess for å komme i mål. Problemet med innføring av en tilleggsteknikk er at det kan fordre merarbeid og økt metodevekt.

Virkningsfull prototyping krever at omgivelsene er lagt til rette for samarbeid mellom systemutviklere og brukere[29]. Innen agile retninger som XP og TDD, er slikt samarbeid en forutsetning. På denne måten er Chaos-rapportens[74] viktigste suksess-faktor—brukermedvirkning, en sentral og integrert del av prosessen.

Jeg mener vi kan se brukermedvirkning i et helt nytt lys, etter som XP, TDD og andre agile tilnærminger oppnår en sterkere stilling og større utbredelse.

Ofte er det argumentert mot teknologidrevne prosesser innenfor IT. Her har vi et eksempel på at en bivirkning fra en teknologidrevet prosess (eller drevet fram av teknologer) med innføring av en utradisjonell metodikk, får positive følger for brukermedvirkning. Og uten av vi må innføre egne tidkrevende teknikker, ekstra aktiviteter, eller forhøyet metodevekt for å oppnå god brukermedvirkning. Jeg ser her kontrasten mellom tidligere behov for å tvinge gjennom brukermedvirkning, og brukermedvirkningen som en forutsetning for å anvende metodikken.

Etter re-design av EIS er sluttført, vil livssyklus-håndteringen gå over i modus for vedlikehold. Hvordan et prosjekt termineres og overføres til linjeorganisasjonen, eller hva som skjer med et avsluttet prosjekt og forvaltningen av systemet i ulike virksomheter, skal jeg ikke her gå inn på. Men skal vi fortsatt være smidige, trengs en form for brukermedvirkning til å støtte den eller de utviklerne som forhåpentligvis fortsetter vedlikeholdsarbeidet, men trolig kan denne begrenses i omfang. I referansegruppen i Forsvaret (se appendiks A) er det gjort gode erfaringer med brukermedvirkning gjennom å holde hyppige møter og tilbakemeldinger gjennom et eget oppfølgingsverktøy mellom utviklere og brukergruppen, samtidig som det hele tiden er et kjørbart system hvor nye versjoner hurtig tilgjengeliggjøres (deployeres) etter endring.

3.2.3.7 Dokumentasjon

Vi har sett kritikken av overdreven bruk av dokumentasjon som driver i tradisjonell metodikk, og da spesielt med henblikk på fossefallsmodellen.

UML-diagrammer kan ta seg godt ut på papir men kan være svært uegnet når innholdet skal programmeres. Slike diagrammer kan bare verifiseres gjennom manuell granskning. Selv erfarne designere blir ofte overrasket når UML design-diagrammer skal manifesteres som programvare[35].

Fra en feltstudie[25] ble det funnet at kommunikasjonsbehovene til team ble dårlig ivaretatt gjennom skrevet dokumentasjon fordi den ikke klarte å gi nødvendig argumentasjon for å løse misforståelser omkring krav eller design-beslutninger.

Min kritikk av de dokumentdrevne prosessmodellene, betyr ikke at man ikke skal ha et minimum av dokumentasjon. Teoribyggingen som skjer gjennom programmering (design og implementering) av et system, vil foruten selve koden, også bæres som iboende kunnskap hos de som utvikler systemet[61]. For å kunne hjelpe nye mennesker ved videreutvikling, kan vi sette opp tre krav til dokumentasjonen[24]:

1. Beskriv metaforene til systemet.
2. Lag en tekstlig beskrivelse til hver hovedkomponent i systemet.
3. Lag tegninger av viktige interaksjoner mellom viktige komponenter til systemet.

Dette blir da tillegget til "ren kildekode", inkludert tester, med konsistente navnekonvensjoner som kan bygge en enhetlig teori om systemet.

Man kan argumentere mot at vi kan bygge en enhetlig teori til systemet, for vi vil forsatt være avhengig av den iboende kunnskapen til de som utvikler. Og om personene som deltok i re-design-prosjektet ikke deltar eller er tilgjengelige

ved senere vedlikeholdsarbeide, vil det ikke være mulig å overføre all iboende kunnskap videre, og deler av teoribyggingen vil gå tapt. Jeg mener vi kan redusere slik risiko ved et minimum av riktig dokumentasjon, i tillegg til kode. Et minimum av dokumentasjon gjør det i tillegg enklere og billigere å holde denne á jour (bedre).

3.2.3.8 Risikohåndtering

Risiko er et hovedproblem forbundet med programvareutvikling. Fossefallsmodellen foreskriver en lang spesifiseringsfase tidlig i utviklingsløpet for blant annet å takle risiko. Boehm tok en annen tilnærming ved å lage den risikodrevne "Spiralmodellen", som tok utgangspunkt i nettopp å redusere risiko gjennom flere, men lange sykler. XP tilhører mere den evolusjonære utviklingsmodellen[26], og har sine egne klare tilnærminger for å håndtere risiko. Der spiralmodellen håndterer risiko gjennom flere lange sykler, går XP den ekstreme veien og foreskriver enda flere og veldig korte sykler.

XP angir flere former for risiki, og håndterer disse gjennom[13]: For å hindre at timeplanen ryker krever XP *korte sykler*. For å hindre at prosjektet kanselleres, ber XP *kunden velge* den minste versjonen som gir mest verdi. For at systemet ikke skal "surne" krever XP god trimming gjennom *automatiserte tester* som kjøres ofte og minimum etter alle daglige endringer. For at feilraten skal være lav skrives *automatiserte tester* av både programmerere og kunder. XP krever at *kunden er en del av teamet* slik at ikke misforståtte løsninger lages. *Korte sykler* hindrer at løsninger gått ut på dato lages hvis virksomheten endres. Unødvendige egenskaper utvikles ikke ved at bare *oppgaver av høyeste prioritet* settes øverst på kartet. For å hindre at prosjektdeltagerne slutter, ønsker XP å *unngå frustrasjon* gjennom ansvar for eget arbeid og estimering, med tilbakemeldinger slik at estimeringen bedres etter hvert.

For re-design av et eksisterende system er risikoen i utgangspunktet redusert fordi vi vet mere om hva som skal utføres i forhold til et nytt utviklingsprosjekt. Videre er risiko redusert ved at prosjektet normalt vil være mindre enn hva tilsvarende nyutvikling ville fordret.

XP's korte sykler vil ytterligere redusere risiko også for for re-design. Et problem som kan inntreffe er at vi vet så mye om eksisterende system at vi fristes til å utføre for store biter om gangen, for raskere å avslutte en oppgave. Her kan de automatiske testene hjelpe oss til å holde syklene korte. Tilsvarende vil kundens prioritering av hvilke oppgaver som løses i hvilken rekkefølge, kunne bidra til at riktige deler med mest verdi for resultatet, utføres først.

Tester som hindrer både at systemet surner og holder feilraten ned vil også for re-design bidra til å redusere risiko. Testing er en fundamental aktivitet i XP og TDD, og vil med hensyn på risikoreduksjon være en integrert del i en prosess. Teststrategier for re-design er omtalt i seksjon 3.2.3.3.

Som for testing er brukermedvirkning avgjørende for XP. Ved tradisjonelle modeller for systemutvikling har man kanskje måtte ty til avtaleverk for å tvinge brukermedvirkning gjennom, eller det har vært innført særskilte aktiviteter for at brukermedvirkning skulle kunne passe inn i modellene. Uansett virkemiddel, har denne medvirkningen stort sett involvert tidlige faser og i liten grad innen konstruksjonen av kode. Dette mener jeg øker risikoen for å lage misforståtte løsninger. Løsningen fra XP er at kunden alltid er tilstede og kan tidlig gjøre

avskjæringer om løsningen, i form av kjørbar kode, blir feil. Forutsetningen er at kunderepresentantene vet tilstrekkelig mye om problemområdet og at løsningen er også er riktig for de de representerer.

Oppgaver med høyeste prioritet velges av kunden til enhver tid, og gitt at kunden velger riktig, unngår vi å lage funksjonalitet som ingen etterspør. For re-design innen oppgavens kontekst vil denne problemstillingen ikke gi misforståtte løsninger, men man kan fristes til å fokusere på nye anvendelsesområder før forretningsreglene som må videreføres fra det gamle systemet er dekket. Dette tror jeg også kan være relevant for nyutvikling; det er fristende å først gjøre de "kule" tingene. Dette anser jeg som en mulig kritikk av XP, men når det er sagt så er det universelt; har man frihet må man også ta ansvar. Men vi snakker ikke om full frihet ala "kod-og-fix", men frihet ved at det er raskt å produsere resultater. Gitt at man tar ansvar og er disiplinert, ferdigstilles forretningsreglene fra gammelt system før nye anvendelsesområder, og reduserer derved også denne risikoen.

Prosjektdeltagere, gitt at disse er dyktige, som slutter er svært uheldig for alle prosjekter uansett enten det dreier seg om nyutvikling eller re-design. En metodikk som gjør det artig å utvikle og som gir personene ansvar for eget arbeide er derfor relevant. Ved at alle fortsetter i prosjektet økes derved muligheten for teoribyggingen omkring systemet.

3.3 Metodikk-forsterker

Agile metodikker vinner terreng innen nyutvikling, og tilsvarende metodikk-retning har jeg argumentert for i forbindelse med modernisering av eksisterende EIS. Enten vi utvikler nytt system eller re-designer, fordrer det både en arkitektur og design av systemet, samt teknologi, rammeverk og verktøy for å oppnå målet om et kjørbart system.

For den referansemetodikken jeg har argumentert for, mener jeg at dimensjonene arkitektur og teknologi kan være med å gi en ytterligere forsterkning til metodikken, hvilket jeg vil omtale som metodikk-forsterkere. Av den grunn må også dimensjonene arkitektur og teknologi behandles for å gi det komplette bildet.

Før vi kan avgjøre om vi har en metodikk-forsterker, må den defineres. Jeg velger definere en metodikk-forsterker som følgende:

> *En metodikk-forsterker for en gitt metodikk har vi når andre dimensjoner forbedrer prosessen slik at den går raskere, eller produktet blir billigere eller bedre. Forbedring av én premiss skal ikke resultere i svekkelse av andre premisser.*
>
> Det skal med andre ord være en positiv korrelasjon mellom en metodikk og forsterkeren.

I de etterfølgende kapitlene om arkitektur og teknologi, skal jeg prøve å argumentere for de elementer som jeg mener virker forsterkende på metodikken med fokus på et re-design perspektiv.

Jeg har hittil vært innom tre konkrete forslag til metodikk-forsterkere:

Enhetstest-rammeverk Den første var et enhets-test rammeverk for TDD som jeg var innom i seksjon 3.2.2 (side 32). Dette rammeverket tilhører teknologi-dimensjonen, og behandles nærmere i seksjon 5.4.3.5.

Objektteknologi Den andre var objekter som en nøkkelteknologi for utflatet kostkurve, blant annet i kombinasjon med første forsterker, og ble nevnt i seksjon 3.2.3.1 (side 34). Objektteknologien tilhører naturligvis også teknologi-dimensjonen, og omtales i seksjon 5.1.

Referansearkitektur Tredje forsterker, etter min mening, var det jeg nevnte som en hensiktsmessig referansearkitektur for hurtig oppstart i forbindelse med utvikling og re-design av et enterprise informasjonssystem i seksjon 3.2.3.1 (side 35). Denne referansearkitekturen tilhører arkitekturdimensjonen, og beskrives i sin helhet under seksjon 4.4.

Vi kan i tillegg til metodikk-forsterkere gjøre bruk av ulike hjelpemidler, men disse anser jeg ikke som konkrete forsterkere for en metodikk, men snarere som bidragsytere for å kunne effektivisere utviklingsarbeidet på generell basis. Et eksempel som allerede er nevnt er en integrasjons-tjener. Dette er et hjelpemiddel som er uavhengig av metodikk, men kan kategoriseres som verktøystøtte for teknikker generelt. En del verktøystøtte blir behandlet i kapittel 5.

Dimensjonene arkitektur og teknologi omtales i de etterfølgende kapitlene.

Kapittel 4

Arkitektur og Design

> First Law of Distributed Object Design: "Don't distribute your objects!"
>
> — *Martin Fowler[32]*

Innen konteksten for enterprise informasjons systemer, er arkitekturen designet til systemet, og har tradisjonelt sett framkommet i en design-fase. Men design-begrepet kan favne andre aspekter enn hva begrepet arkitektur dekker. Dette kommer jeg tilbake til i seksjon 4.1.

Jeg vil i dette kapittelet ta for meg ulike arkitekturer og deres styrker og svakheter. Videre vil jeg vise at innen re-design som for nyutvikling, er arkitektur og design med tilhørende mønstre viktige byggeklosser og kunnskapsområder i seg selv, men i dette kapittelet vil jeg i tillegg prøve å se om arkitekturen, eller deler av arkitekturen, kan inngå som metodikk-forsterker til referansemetodikken som ble omtalt i forrige kapittel.

Målet med kapittelet er å gi en referansearkitektur og prøve om denne oppfyller kravene til en metodikk-forsterker. Videre skal denne referansearkitekturen gi grunnlag for videre arbeid i teknologidimensjonen. Kapittelet er delt inn i følgende seksjoner:

Design I seksjon 4.1 omtales begrepet design. Her forsøker jeg å definere og sette design-begrepet i en egnet kontekst, og som grunnlag for senere å gå inn i arkitekturbegrepet.

Arkitektur I seksjon 4.2 starter jeg med ulike definisjoner og beskrivelser vedrørende begrepet arkitektur. Deretter følger en evolusjon for noen sentrale arkitekturer, og toneangivende retninger. Viktige mønstre med relevans for oppgaven er beskrevet til slutt i denne seksjonen.

Mønstre I seksjon 4.3 er ulike mønstre beskrevet.

Referansearkitektur I seksjon 4.4 oppsummerer jeg arkitekturdimensjonen tilpasset denne oppgaven, som vil danne grunnlaget for å kunne vurdere referansearkitekturen som metodikkforsterker. Her foreslår jeg en referanse programvarearkitektur til bruk for re-design, og som jeg vil bruke til argumentasjon for, som en forsterker til referansemetodikken fra forrige

kapittel. I tillegg belyser jeg enkeltsider fra arkitektur-dimensjonen som også kan virke forsterkende på, eller til hjelp for, metodikk. Referansearkitekturen jeg foreslår, kan også ansees som veiledende i andre sammenhenger, som nyutvikling.

Metodikkforsterker? I seksjon 4.5 forsøker jeg å argumentere for om arkitektur metodikk-forsterkeren kan godkjennes eller må forkastes.

4.1 Design

En generell definisjon av design kan være som følger:

> "Design som prosess kan ta mange former avhengig av objektet som designes og individet eller individene som deltar. Innen konteksten for utøvende kunst, engineering, arkitektur og andre tilsvarende kreative aktiviteter, er design både et substantiv og et verb.
>
> Design innen verbets kontekst er prosessen av å frambringe og utvikle en plan for et estetisk og funksjonelt objekt, som vanligvis krever en anseelig mengde grunnarbeid, tankevirksomhet, modellering, iterativ justering og re-design.
>
> Som et substantiv, er design brukt både om den endelige planen (en tegning, modell, eller annen beskrivelse), eller resultatet av å følge den utarbeidde planen (det produserte objektet)."
>
> — *Wikipedia[83]*

Innen systemutvikling har de tradisjonelle prosessmodeller beskrevet design som en eksplisitt aktivitet eller fase, som fulgte etter analysen og som endte ut i systemdesign, modeller og arkitekturen til systemet—tidvis omtalt samlet som arkitekturen. Arkitektur og design behandles tidvis om hverandre i litteraturen, og abstraksjonsnivået eller detaljeringsgraden kan være ulik, men framgår (vanligvis) av sammenhengen.

På overordnet nivå—overordnet design, kan vi omtale design som å designe arkitekturen, og resultatet er designet til arkitekturen, eller arkitekturen. Når design benyttes som substantiv er derved design synonymt med arkitekturen på overordnet nivå. Med økende detaljeringsgrad—detaljert design, kan vi designe en domene-modell for å ende opp med en designet domene-modell—domene-modellen[1]. Slik kan vi videre beskrive design, og ende opp med design av ytterligere forfinet detaljeringsgrad. Ved kodedesign, eller kode som driver, rett til designet. Slik design av kode for systemet vil kunne omhandle ulike detaljeringsnivå.

Vi designer en arkitektur, vi designer lagdeling innen en arkitektur, vi designer komponenter og designer for gjenbruk av komponenter, vi designer tester (ofte mens vi skriver dem), og vi designer kode-strukturer og kode.

Som en aktivitet *designer* vi noe, og kan i ettertid forholde oss til *designet* av dette noe. Sammenligner vi dette med XP's syn på metaforer i forbindelse med arkitektur, er vi inne på idéen om å formidle kunnskapen om teorien til programmet[61]—gjennom designet og koden.

[1] Domene-modellen har også verdi i analysesammenheng og ikke bare i designsammenheng

En slags aristotelisk beskrivelse på design av en arkitektur med lagdeling finner jeg hos Beck[13]:

> "Designprosessen er å lage en struktur som organiserer logikken i systemet. God design organiserer logikken slik at endringer i en del av systemet ikke påvirker andre deler. God design sikrer at hver del av logikken er på kun ett sted. God design legger logikken i nærheten av de data som behandles. God design tillater utvidelser av systemet gjennom endringer på kun ett sted. God design er å lagdele så enkelt at det enten gir ekstra funksjonalitet eller en forklaring av hensikten. God design er enkelhet framfor kompleksitet."

Hvilket er nettopp hva vi ønsker å oppnå i vår arkitektur. Og med lagdelingen som mål for programvarearkitekturen, designer vi lagene slik at et lag som skiftes ut ikke berører andre lag. Ved å designe hvert lag med et veldefinert grensesnitt, for så å implementere det aktuelle laget oppnår vi denne fleksibiliteten. Arkitektur og design henger sammen og er gjennomgripende i alle deler av et utviklingsløp.

Design har vi også for brukergrensesnitt. Både i form av funksjonelt design slik en tenkt bruker vil benytte systemet og hvilke funksjoner som tilbys, og i retning av grafisk design slik systemet faktisk vil materialisere seg i form av skjermbilder.

I et programvareutviklingsprosjekt er utførende design en viktig faktor for å komme i mål. Tradisjonell designaktivitet følger en forutgående analyseaktivitet, enten vi ser på fossefallsmetoden og RUP, eller spiralmodellen. Innen den evolusjonære utviklingsmodellen og de agile metodikkene, har vi ikke denne eksplisitte fasen eller aktiviteten, men snarere en kontinuerlig prosess gjennom refaktorering, stadige endringer, iterasjoner og inkrementer. Cockburn tar opp problematikken med design av metodikker[24] gjennom å foreslå prinsipper for akkurat slik design, som vi så i seksjon 3.2.3.

Vi kan videre tenke på ulike *drivere* av design. Design kan drives fram av tekstlige spesifikasjoner og krav. Tester og kode kan være en annen driver for designet. En tredje driver kan være domenedrevet design, hvor vi har en sterkere objektorientert tilnærming for å gripe an problemområdet. Datadrevet design eller modellering er en annen variant med fokus på data-modellen.

Dette som en innledning og kontekstualisering for videre beskrivelser av arkitektur og mønstre tilhørende begge begreper. Angående mønstre for design, er konteksten design i form av kode, med vektlegging av objektorientering.

4.2 Arkitektur

Arkitektur assosieres gjerne med kunst og kreativitet, men begrepet arkitektur kommer fra bygnings-bransjen, og har eksistert i århundrer. Å definere arkitektur innen området systemutvikling blir en subjektiv betraktning. Jeg vil presentere tre ulike grupper av betraktninger, før jeg selv velger min definisjon.

Martin Fowler[32] definerer, eller rettere sagt beskriver forslag til, arkitektur som for det første; "det høyeste nivå av et system splittet opp i sine deler", og for det andre; "beslutninger som er vanskelig å endre". Han gir også en tredje tilnærming til en definisjonsbeskrivelse;

> "En subjektiv felles forståelse i form av de største komponentene for et system og hvordan disse interagerer."

Vi kan også se til IEEE's[49][2] definisjon av arkitektur:

> "Arkitektur har ulike meninger, avhengig av dens kontekstuelle bruk:
> (a) Strukturen av komponenter, deres inter-relasjonsskap, og prinsippene og retningslinjene som styrer deres design og utvikling over tid.
> (b) Organisasjonell struktur for et system eller komponent."

Felles for begge forståelsene er at arkitektur regnes som den mest bestandige delen av systemet.

En tredje tilnærming til arkitektur, finner vi innen lettvektsmetodikken XP, der arkitektur vektlegges, men hvor fokus først og fremst er på å beskrive gode *metaforer* for arkitekturen, slik at denne lett kan kommuniseres (hvilket ligger tett opp til Naur's[61] idé om å overlevere teorien til designet). Etter at arkitekturen er gitt en metafor, er neste steg å designe en arkitektur, og i henhold til XP gjøres det ved å ta samlingen av bruker-historier fra "Planleggingsspillet" (se seksjon 3.2.1), velge ut én som man antar tvinger en til å lage hele arkitekturen, og implementere denne som en første iterasjon. Deretter forenkles arkitekturen gjennom refaktorering så mye som mulig, for samtidig å virke etter hensikten. Arkitekturen endres i tråd med behovene som oppstår, men skal forsatt være så enkel som overhodet mulig[13]. Ved å følge XP's oppskrift på å lage en arkitektur for systemet under bygging, er ikke arkitekturen den mest bestandige delen av systemet lengre. XP hevder at *endringer* er det eneste bestandige.

Jeg velger for denne oppgaven å definere arkitektur med fokus på programvare, eller som system, for enterprise informasjonssystemer som:

> Et systems minimale struktur i form av laginndelinger som spiller sammen i et hele for å gi nødvendig virksomhetsstøtte i systemets problemområde.

Dette gir hovedsaklig en tradisjonell forståelse av arkitektur, men begrenset i omfang.

I de følgende underseksjoner har jeg beskrevet hvordan programvarearkitekturer har utviklet seg, og hvilke fordeler og ulemper disse har. En sammenstilling og oppsummering kommer til slutt i form av arkitektoniske lagdelinger.

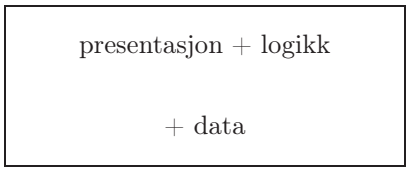
4.2.1 Monolittisk arkitektur

I denne arkitekturen, som mer er i retning av en maskinarkitektur, råder stormaskiner med tilkoblede terminaler som kan være ikke-grafiske eller grafiske (som X-terminaler).

All logikk, presentasjon, og data i form av filer manipuleres fra applikasjonen på stormaskinen[32].

Som programvarearkitektur må vi se logikk, presentasjon, og data samlet. Som en maskin-arkitektur kan vi finne én sentral maskin (tjener) med programvarearkitektur i flere lag. Se figur 4.1 for en skjematisk oversikt.

[2] IEEE—The Institute of Electical and Electronics Engineers



Figur 4.1: 1-lags maskin-arkitektur.

Fordeler

- Sentralisering sikrer enkel og rask deployering.
- Sentralisering forenkler administrasjon og drift.
- God verktøystøtte gir lavere utviklingstid, og enklere utvikling. Verktøystøtten kan være kostbar.

Ulemper

- Brukergrensesnittet kan være uegnet for Internet-bruk.
- Kan gi dårlig brukeropplevelse, spesielt ved karakterbaserte terminaler.
- Skalering kan kun skje gjennom økning i maskinkapasitet, hvilket kan gi begrensninger i antall samtidige brukere.
- Meget lavnivå og komplisert integrasjon direkte mot applikasjonens persisteringsmekanismer.
- Vanskelig vedlikehold av systemet ved at logikk, presentasjon, og data ofte ikke er adskilt.
- Svært høy TCO (Total Cost of Ownership) i dag, da dette er "legacy" systemer som stadig færre har kunnskap om.
- Lavstatus å jobbe med.

4.2.2 Klient-Tjener arkitekturer

Klient-Tjener arkitekturen er en nettverks programvare arkitektur, som som skiller klienten fra tjeneren, ved at programvaren på ulike klienter sender forespørsler til tjeneren som i tur gir responser tilbake[42].

Denne kategorien oppsto først som en 2-lags arkitektur hvor applikasjonslogikken var bundet opp i klientprogrammet[32, 42]. Senere fikk vi et tredje lag mellom klienten og tjeneren. En generalisering oppnår vi ved å partisjonere tjeneren i flere tjenester og lag, og derav den såkalte n-lags arkitekturen[42].

Klientene kan være tykke, tynne, eller rike. For tykke klienter som kjører på en PC eller arbeidsstasjon, ligger all logikk, eller deler av den, samt brukergrensesnittet, lokalt. For hver endring i klient-programvaren må denne tilgjengeliggjøres på en eller annen måte ut til de ulike klient-maskiner. Krav til endring i klient-programvaren skjer ofte som en følge av endringer på tjener-programvaren.

Vi kan ha tynne klienter, hvor vedlikehold og tilgjengeliggjøring kun skjer sentralt på en applikasjons-tjener og eventuelt en database-tjener.

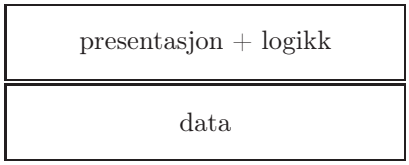
En mellomting mellom tykke og tynne klienter benevnes som rike klienter, hvor man bestreber seg på å ha et godt (rikt) grafisk brukergrensesnitt på klienten for en optimal brukeropplevelse, men da fortrinnsvis uten forretningslogikk.

4.2.2.1 2-lags arkitektur

En typisk 2-lags klient-tjener programvarearkitektur finner vi i systemer med tykke klienter hvor logikk og presentasjon ligger i programvaren lokalt[32], mens en sentral databasetjener stort sett ikke implementerer forretningslogikk men kun håndterer persistens[42].

Varianter finnes med ulik grad av logikk på klienten og tjeneren. Vedlikeholdsmessig er det mest ressurskrevende når logikken er spredt[32].

Innenfor denne arkitekturen finner vi oppgavens enterprise informasjonssystemer som jeg ønsker å modernisere. Se figur 4.2 for en skjematisk oversikt.



Figur 4.2: 2-lags arkitektur.

Fordeler

- God brukeropplevelse med rask skjermbilde-respons og mulighet for avanserte grafiske grensesnitt.
- God verktøystøtte gir lavere utviklingstid, og enklere utvikling.
- Skalering av database-tjener er enkelt gjennom klustring av flere maskiner.
- Krever normalt lite nettverkstrafikk for skjermbildeoppdatering.

Ulemper

- Brukergrensesnittet er i utgangspunktet uegnet for Internet-bruk.
- Fordelen med god verktøystøtte er også en ulempe i det at verktøyene ofte er sentriske mot bruk av SQL[3], og typiske applikasjoner er begrenset til å håndtere data inn og ut, samt oppdateringer—såkalt CRUD[4]-logikk. I tillegg kan støtten være kostbar.

  SQL som standard for relasjonelle databaser, skulle gi muligheten til å bytte leverandør om man ønsket, uten for høye flytte-kostnader. Men bruk av lagrede prosedyrer reduserer denne muligheten[32].

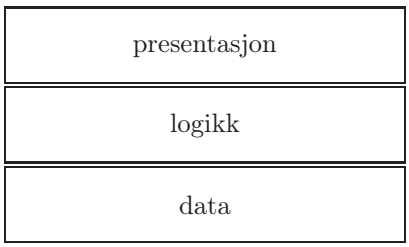
[3]SQL—Structured Query Language
[4]CRUD—Create, Retrive, Update, Delete

- Deployering/oppdatering må gjøres til alle klienter.
- Kun lavnivå integrasjon direkte mot databasen.
- Høy trafikk på nettverket av data, spesielt i forbindelse større mengder data som returneres fra en spørring, for sortering på klienten, gir begrensninger i antall samtidige brukere[42].
- Krevende administrasjon ved at alle klienter krever administrasjon.
- Vanskelig vedlikehold gjennom at endringer ett sted påvirker resten av systemet[32].
- Kostbar TCO grunnet nevnte administrasjon, deployering vedlikeholdsproblemer, og ofte høye lisenskostnader.
- Tiltagende lavstatus å jobbe med—utviklere ønsker en dreining over mot mere tidsriktige systemer.

4.2.2.2 3-lags arkitektur

Ved å innføre ett ekstra lag mellom klienten og tjeneren, oppnås en merkbar ytelsesforbedring. Klienten vil nå typisk være "tynn", og vi flytter logikken og håndtering av brukergrensesnittet over i det nye mellomliggende applikasjonslaget. Vi ønsker vanligvis at databasetjeneren også skal være "tynn" av hensyn til enklere vedlikehold ved at logikken ligger samlet ett sentralt sted. Applikasjonslaget og databaselaget kan enten være på samme maskin, eller de kan spres til ulike noder[42]. Om det siste er tilfelle er det ønskelig med høyest mulig båndbredde mellom nodene. Fordelen med separate noder kan være økning i prossessorkapasitet. Ulempen kan bli mer administrasjon og mindre kontroll, men i tilfeller med delte databaser er ikke dette til å unngå.

Se figur 4.3 for en skjematisk oversikt.



Figur 4.3: 3-lags arkitektur.

Fordeler

- Forbedret ytelse i forhold til 2 lag[42].
- Egnet for Internet og store intranett med mange brukere.

  Et godt designet 3-lags system, trenger bare å legge til et nytt presentasjonslag for "Web'en", og er med bruk av et objekt-orientert språk mindre knyttet til SQL[32].

- Kan skaleres opp for flere samtidige brukere ved at lagene distribueres til ulike maskiner[42].
- Sentraliseringen gir enklere deployering og administrasjon.
- Lavere TCO gjennom sentralisering og gode muligheter for skalering.
- Integrasjon kan skje på høyere (abstraksjons)nivå.
- *Kan* gi støtte for enklere testing, gitt teknologisk verktøy-støtte, til en metodikk.
- I tillegg til lavere TCO, et mål for sponsoren, ønsker utviklerne denne retningen, da den gir høyere status og mere interessant å arbeide med.

Ulemper

- Tynne klienter gir normalt en dårligere brukeropplevelse enn tykke klienter. Tynne klienter er i tillegg uegnet for nettverk med lav båndbredde.
- Distribuering av lagene kan føre til ekstra administrasjon i forbindelse med deployering, og øker kompleksiteten.
- Vanskeligere og mer tidkrevende å utvikle. Spesielt web brukergrensesnittet *kan* kreve mye ressurser.

4.2.2.3 n-lags arkitektur

> "Gjennom 90-tallet har utviklingen av tradisjonelle informasjonssystemer gått fra 2-lags klient-tjener arkitektur til 3-lags eller n-lags arkitektur"[42].

Med utgangspunkt i applikasjonslaget vi skilte ut for 3-lags arkitekturen, kan dette mellomvarelaget splittes opp i nye lag, hvor vi kan skille ut domenemodell og forretningslogikk i separate lag[42], og legge til et abstrahert lag for data-håndtering[50]. Man kan distribuere disse mellomvarelagene på separate noder for å lage en distribuert løsning[42], men spesielt på grunn av drastisk ytelses-svekkelse og kompliserende utvikling er ikke dette å anbefale. Grunnen belyses ytterligere i seksjon 4.2.4.

Spesialisering av lagene medfører at endringer i ett lag ikke trenger å få følger for alle andre lag, men for stor grad av spesialisering kan få følger for vedlikeholdbarheten. Mer om dette i seksjon 4.2.6.

4.2.3 Agent-arkitekturer

Agent-arkitekturer er en helt annen type programvarearkitektur, som benyttes ved agentbaserte systemer. Disse beskrives i [42, ss. 48-50]. Jeg vil ikke gå inn på disse, da de etter min menig ikke har relevans innen oppgavens omfang.

4.2.4 Distribuert objekt-arkitektur

Objekter har vi hatt lenge, og mange har ønsket å distribuere disse. CORBA er en slik teknologi[42]. Distribuerte objekter har mange flere fallgruver enn hva de fleste er klar over. Den største fallgruben er trolig treghet. Sammenlignet med et prosedyrekall utført internt i en maskin, vil et kall fra et objekt på en maskin, over et nettverk, til et annet objekt på en annen maskin være meget langsomt[32, [84]]. På den annen side har en viktig faktor for CORBA vært plattformnøytralitet og interoperabilitet. Denne faktoren vil jeg komme nærmere tilbake til i forbindelse med tjeneste-orientert arkitektur (i seksjon 4.2.5).

Et finmasket grensesnitt for et objekt er i henhold til gode OO prinsipper, hvor mange små deler kan kombineres og overstyres. Samtidig som det letter framtidige design-utvidelser. Dette virker godt for lokale objekter, men for distribuerte, må vi ty til grensesnitt av en granularitet som er mer grovkornet, for å minimalisere nettverkskall[32]. Her finner vi også grunnlaget for Façademønsteret (se seksjon 4.3) som også kan brukes i arkitekturer som ikke er distribuert, fordi det forenkler det totale systemets grensesnitt mot omverdenen.

Et argument for å bygge distribuerte systemer med forretningslogikk i form av distribuerte samspillende objekter spredt rundt på ulike noder i et nettverk, skulle være skalerbarhet. Prisen vi må betale er høy i form av ekstra kompleksitet, vesentlig lavere hastighet, og mer omfattende deployering[32]. Etter min mening viser dette at denne arkitekturen ikke skalerer, og derved virker delvis mot sin hensikt. En sentralisert applikasjon som kjører på én node kan i stedet skaleres ved for eksempel å klustre to maskiner. I de fleste tilfeller, og spesielt med fokus på enkel re-design av EIS, bør distribuerte objekter unngås. Dette viser seg også i praksis å ofte ikke være påkrevd.

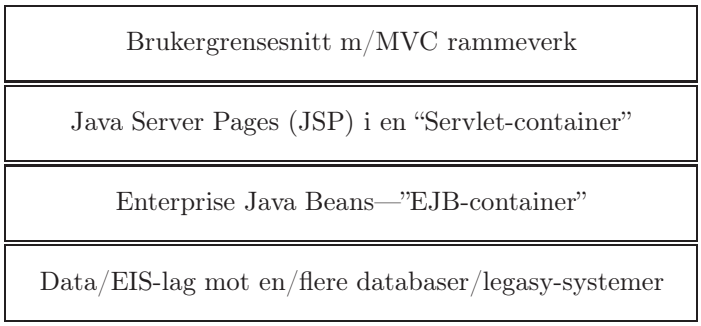
4.2.4.1 Klassisk J2EE arkitektur

Dette er arkitekturen som J2EE plattformen skulle dreie seg om. Den reflekterer et nesten universelt syn på J2EE som en distribuert plattform, hvor både sesjonsbønner og entitetsbønner, innen kategorien "Enterprise Java Beans" (EJB), er viktige bestandeler spredd utover ulike noder. Weblaget er vanligvis implementert i et MVC (Model View Controller—se seksjon 4.3.1.4) mønster og rammeverk, mens forretningsobjektene er tilstandsløse sesjons-bønner med "remote" grensesnitt. All datatilgang skjer via entitets-bønner som opererer mot et lag av en eller flere databaser og eventuelt legasy-systemer[53].

En enkel skjematisk framstilling av denne arkitekturen, i form av lagdeling og system-komponenter, er listet i figur 4.4.

Fordeler med den klassiske J2EE arkitekturen[53]:

- Veldefinert tjenestelag.
- Viktige enterprise-tjenester er inkludert.
- Ulike klient-typer støttes gjennom et felles mellomvarelag.
- Forretningslogikk i EJB-laget kan oppdateres uavhengig av weblaget.



Figur 4.4: Klassisk J2EE arkitektur.

Ulemper med den klassiske J2EE arkitekturen[53]:

- Krever store utviklingsressurser, spesielt fordi EJB'er er svært kompliserte.
- Forsaker objektorientering til fordel for distribuering. Når vi mister fordelene med objektorientering, er det å betrakte som et antimønster[21].
- Vanskelig å teste på grunn av den sterke avhengigheten til EJB-containeren.
- Ytelsen blir lav gjennom distribuerte objekter[32].

4.2.5 Tjeneste-orientert Arkitektur

I de siste tre til fem årene har det vært skrevet mye om tjeneste-orientert arkitektur. Store kommersielle leverandører ønsker å selge konseptet med tilhørende metode- og verktøy-støtte eller reklamerer med at produktene deres har støtte for denne arkitekturen. Andre ikke-kommersielle aktører og standardiseringsorganisasjoner som OASIS[5] og W3C[6], har etter min mening, et mere nyansert syn på anvendbarheten.

Begrepet "tjeneste-orientert arkitektur" er en oversettelse av det engelske begrepet "Service-Oriented Archetecture" (SOA). SOA som arkitektur består av en samling løst koblede tjenester, som i seg selv er en samling funksjonalitet. Tjenester i denne kontekst er lik komponenter, hvor begge har klart definerte grensesnitt, og en datamodell for utveksling av informasjon[37].

SOA muliggjør interoperabilitet gjennom at tjenestene utveksler informasjon. Tjenestene tilbyr fleksibilitet, smidighet og tilpasning[60].

Tjenester i denne sammenhengen henspeiler på tjenester som er tilgjengelige for enterprisen, og ikke interne tjenester i applikasjonene. I moderniseringen av enterprise informasjonssystemer, er dette av de nye anvendelsesområdene som jeg forsøker å gi et enklere bidrag til.

Fra bakgrunnen til oppgaven beskrives premissen "raskere", og med riktig orkestrering, eller deklarativ komposisjon av tjenester, skal vi kunne raskere understøtte de stadige endringer i virksomhetsprosessene, gitt at virksomheten

[5]OASIS—Organization for the Advancement of Structured Information Standards
[6]W3C—The World Wide Web Consortium

er på et slikt modenhetsnivå med hensyn på at den innehar gjenbrukbare standardiserte tjenester—og er i stand til å orkestrere disse.

SOA fokuserer på å lage både et begrep, teknologi, og prosess-rammeverk, med fokus på å hjelpe enterpriser å utvikle, inter-konnekte, og vedlikeholde enterprise applikasjoner og tjenester på en kraftfull og kost-effektiv måte. Målet er ikke nytt, men SOA forsøker å integrere applikasjoner og tjenester til å kjøre sømløst uten barriérer fra distribuerte systemer, applikasjonsintegrasjon, ulike plattformer og protokoller, og ulike tilgangs-enheter. Dette søkes oppnådd ved gjenbruk av tidligere bestrebelser gjort med hensyn på modulær programmering, gjenbruk av kode, og objektorientert programvare. SOA er designet slik at utviklernes mange og komplekse implementeringsutfordringer lettes, og samtidig gi økt potensiale for Internet[55]. I tillegg vil jeg hevde at "discovery"-prosessen er sentral i forhold til SOA. Denne prosessen muliggjør lagring av metadata om en tjeneste, slik at andre tjenester skal kunne oppdage og gjenbruke tjenesten.

I et SOA miljø har vi uavhengige tjenester fra ulike noder på et nettverk som er tilgjengelige på et standardisert sett[83]. Enklere sagt, skal vi ved hjelp av SOA og web tjenester (WS[7]), kunne bygge tjenester av tjenester.

VVV[8] blir stadig mer brukt for kommunikasjon mellom programmer. Her spiller webtjenester en viktig rolle ved å gi et programmatisk grensesnitt[81, 82]. Som oftest forbindes SOA med webtjenester og protokollene SOAP[9], WSDL[10], og UDDI[11], men SOA kan implementeres ved hjelp av andre tjeneste baserte teknologier. For eksempel kan vi lage SOA ved hjelp av andre teknologier, som kan være enklere i de tilfeller hvor full plattformuavhengighet ikke er nødvendig. Slik mulig løsning forfekter jeg senere i denne oppgaven, ved å legge på "remoting"-mekanismer som beskrevet i seksjon 5.4.3.6. Vi kan også bruke CORBA[42] for å implementere SOA, selv om vi ikke bruker distribuerte objekter. Like fullt kan vi ha en tjeneste-orientert arkitektur (SOA).

Som beskrevet i seksjon 4.2.6.2, prøver vi å unngå å holde tilstander i tjenestelaget, men overlater slik håndtering til presentasjonslaget. Webtjenester bør også være tilstandsløse, og eventuelle tilstandshåndteringer overlates til konsumenten av tjenestene. Eksempelvis kan vi gjøre bruk av BEPEL[12] til både å koreografere og tilstandshåndtere webtjenestene.

Med erfaringer fra referansegruppen i Forsvaret har jeg sett at en fornuftig lagdelt arkitektur med et veldefinert Façadelag/Tjenestelag, er et godt utgangspunktet også for SOA. Ved hjelp av et grovkornet Façadelag, som i utgangspunktet settes opp for å understøtte presentasjonslaget, kan vi velge hvilke tjenester som skal tilgjengeliggjøres i form av webtjenester. Referansearkitekturen som beskrives i seksjon 4.4 har en figur (4.5) som viser dette skjematisk. Utfordringen til virksomheten blir å opparbeide kunnen om å vite hvilke gjenbrukbare tjenester som faktisk behøves. Dette siste anser jeg som utfordrende, og vil kreve en kombinasjon av bred teknologisk og generell virksomhetssentrisk erfaring, men omtales ikke i nærmere i oppgaven.

[7] WS—Web Services (se definisjon side122)

[8] VVV—forkortelse for VerdensVeVen i.h.h.t. Norsk Språkråd. Samme som "World Wide Web" (WWW).

[9] SOAP—Simple Object Access Protocol (opprinnelig) eller Service Oriented Architecture Protocol (nyere oversettelse med sterkere SOA-fokus)

[10] WSDL—Web Services Description Language

[11] UDDI—Universal Description, Discovery and Integration

[12] BPEL—Business Process Execution Language

Design av en tjenesteorientert arkitektur kan by på utfordringer. Trenden, i følge Fowler[33] og Evans[28], blant "SOA-folkene" er å gjøre domene-modellen anemisk ved å legge alt av metoder/operasjoner inn i tjenestelaget. Dette bryter fundamentalt med objektorientert tankegang, hvor både data og operasjoner skal kapsles inn. Etter min mening oppnår vi best resultat ved å først bygge en objektorientert domene-modell, og deretter legge et tynt tjenestelag på toppen av denne. Siden kan utvalgte deler fra det tynne tjenestelaget "SOAfiseres". Mitt re-design vil legge opp til en slik framgangsmåte.

Adopsjon av SOA for gamle systemer som først burde re-designes, kan sammenlignes med et av oppjusteringsalternativene i seksjon 2.2; hvor en leverandør tilbyr lisensbaserte verktøy og pakker problemene med de gamle systemene bort, men forenkler integrasjonen innen en enterprise vid kontekst.

Fordeler med tjeneste-orientert arkitektur:

- *Fleksibilitet* er etter min mening den absolutt største fordelen med tjenesteorienterte arkitekturer.
- Høynivå integrasjon og samspillende systemer fra ulike plattformer som J2EE og .NET[13][2].
- Raskere understøttelse av stadige endringer i virksomhetsprosessene, gitt at gjenbrukbare tjenester og mekanismer for orkestrering og "discovery" eksisterer.
- Kan gjenbruke lokale tjenester i et veldefinert tjenestelag som er eksponert for dette[53].
- Deklarative beskrivelser slik at tjenester kan benyttes for å komponere tjenester av tjenester.
- Tjenester blir tilbudt gjennom grensesnitt, hvor implementasjonen skjules og implementeres etter behov.
- Vi kan bruke tjenester fra legasy applikasjoner hvis disse eksponerer sine tjenester som grensesnitt, og derved implementere disse med "native" API'er; eksempelvis[53] JDBC direkte mot databaser, eller JCA direkte mot legasy systemer. Legasy systemene trenger ikke å endres for slik tjenestegjøring.

Ulemper med tjeneste-orientert arkitektur:

- Kan gi store utviklingskostnader hvis lokale tjenester i en applikasjon som ikke trenger å distribueres blir distribuert.
- Ytelsen blir lavere med bruk av distribuerte tjenester enn ved bruk av lokale tjenester—en avveining opp mot fordelene ved flekibilitet og gjenbrukbarhet.
- Domene-modellen kan bli anemisk[33].
- Trolig meget kostbart, og må vurderes opp mot forventede senere innsparinger. Som i seg selv kan være svært vanskelig å fastslå.

[13].NET—Microsoft .NET Framework

4.2.6 Arkitektoniske lag i enterprise informasjonssystemer

Vi har sett ulik lagdeling for programvare fra ett til mange lag, med og uten lokale objekter, og objekter distribuert på ulike noder i et nettverk. I tillegg så vi tjenester i stedet for objekter distribuert på samme måte.

Jeg mener at et begrenset antall lag (men over to) og uten bruk av distribuerte objekter, vil gi redusert kompleksitet for et system, i motsetning til flere lag og distribuering av objekter. Dette er også erfart av en gruppe i Forsvaret med hensyn på enterprise informasjonssystemer, og tilsvarende erfaringer er uttrykt innenfor deler av konsulentbransjen jeg kjenner. For andre typer systemer og bruksområder vil vi måtte leve med en mere komplisert arkitektur, men dette ligger utenfor oppgaven. Tjenestene derimot, kan være distribuerte og gjenbrukbare fortrinnsvis støttet gjennom en hensiktsmessig arkitektur og lagdeling. Mellomvare og rammeverk som støtter denne trenden er allerede godt etablert. Slike rammeverk kommer jeg tilbake til i kapittel 5.

Målet med programvarearkitektur generelt er å få fram en lagdeling med løse knyttninger og sterke logiske avhengigheter. Videre at kommunikasjonen mot systemets lag normalt går kommer inn mot grensesnittet i det øverste laget og går videre nedover i lagene. Lagene må være utskiftbare slik at vi på et senere tidspunkt kan endre eller bytte ut teknologien eller implementasjonen i et lag[32, 40]. En god arkitektur for et system omtales ofte som at systemet er godt designet, og koblinger lagene i mellom kan være på ulike detaljeringsnivåer[32]. Lagdeling kan også gi ulemper. Et klassisk eksempel er når et persisterbart felt skal synes i brukergrensesnittet, hvilket fordrer oppdatering av alle mellomliggende lag. Ytelsen kan også reduseres ved for mange lag, men samtidig kan lagdeling i form av innkapsling av underliggende funksjoner, som eksempelvis transaksjonshåndtering, oppveie ytelsesreduksjonen gjennom å optimalisere laget som innkapsler slike underliggende funksjoner[32].

Lagdeling er en av de mest brukte teknikkene for å bryte ned et system i mindre og lettere håndterbare deler. Lagdeling av applikasjoner forenkler, gjennom at hvert lag har sitt fundamentale ansvarsområde[32, 53, 40]. Objektorientert programmering (OOP) gjør effektiv lagdeling enklere gjennom innkapsling. Videre støtte fra OOP oppnås ved å programmere mot grensesnitt, hvilket letter innføringen av nye lag i form av abstraksjonslag som presenterer enkle programgrensesnitt for sine klienter. Dette gjelder som regel for alle lagene; at hvert lag internt er delt opp i et tynt grensesnittlag og et implementeringslag. Igjen for å forenkle slik at implementasjonen kan endres uten å påvirke andre lag. Detter bryter med XP's syn på arkitektur, hvor elementer ikke skal innføres før behovet oppstår. Det kan i ettertid vise seg at ikke alle lag i var nødvendige, hvilket synliggjør unødvendig ressursbruk. Men jeg ønsker å imøtegå XP på dette området, og spesielt med tanke på re-design av enterprise informasjonsystemer. Jeg vil hevde at når denne ytre og indre lagdelingen er etablert på forhånd, og erfaringene tilsier at når lagene er tilpasset generelle EIS, er dette en god tilnærming.

For enterprise systemer har vi tre kjernelag, som best vedlikeholdes om de plasseres på en tjener-maskin[32], identisk med de vi så i 3-lags arkitekturen:

1. *Presentasjon*, omhandler interaksjonen mellom bruker og system. Det kan også være grensesnitt mot andre systemer. HTML er en god løsning for grensesnittet ut til brukeren.

2. *Domene logikk*, ofte omtalt som "forretnings logikk". Her utføres arbeidet som applikasjonen skal dekke mot det aktuelle domenet eller problem-området. Dette vil inkludere forretningsregler som systemet skal oppfylle.

3. *Data laget*, vil for de fleste enterprise applikasjoner være i form av en database for persistering av applikasjonens data.

En videre vanlig oppdeling er å splitte domene logikk laget i to[32]: Et *tjeneste lag*, og et underliggende lag for *domene modellen* hvor begge også er mønstre av samme navn (se seksjon 4.3.1). En annen betegnelse på tjenestelaget når dette er tynt, er Façadelaget. Dette skal gi et høynivå grovkornet grensesnitt over domene modellen[40]. Tilsvarende resultat fant vi for n-lags arkitekturen i seksjon 4.2.2.3. Tjenestelaget er det laget jeg vil legge til grunn for generelle grensesnitt ut mot andre systemer.

For å oppsumere vil jeg gi denne lagelingen en metafor:

> Et dokument delt inn i seksjoner, hvor hver seksjon beskriver sin del av en verden, og kun refererer til nærmeste etterfølgende seksjon.

Jeg vil nå presentere de lag som senere vil inngå i referansearkitekturen.

4.2.6.1 Domene-modell

Dette laget er utskilt fra domene logikk laget[32], med tilhørende arkitektur mønster av samme navn. Alternativ navngiving på dette laget kan være "domenelaget" eller "modelllaget".

Spesielt for domene-modellen er at den ikke kjenner til, eller er klient mot, andre lag. Domenemodellen er et objekt-hierarki med attributter og metoder for domenet som behandles (problemområdet)[32]. Domenemodellen kan brukes direkte av både façadelaget og presentasjonslaget, mens datalaget har egne programerings-grensesnitt (API[14]) for den fysiske persisteringen mot enten enkle filer eller databasesystemer. At domene-modellen er uavhengig av andre lag, og egne mekanismer sørger for persisteringen, gjør det enkelt å teste dette laget isolert.

For applikasjoner med ikke-triviell forretningslogikk, hvilket etter min mening er typisk for enterprise informasjonssystemer (selv om de ikke graderes ut mot den mest kompliserte enden av skalaen), kan domenemodellen hjelpe oss å forbedre gjenbruk av kode og gi et betraktelig forenklet vedlikehold[11].

Domene-modellen er ansvarlig for å representere konseptene, informasjonen og reglene for problem-området—altså både tilstand og oppførsel, selv om tekniske detaljer for lagring delegeres til infrastrukturen. Dette laget er kjernen for foretnings-programvaren[28]. Domene-modellen kombinerer data og prosesser[32], og det er dette, i form av forretningsregler og domenerepresentasjonen fra databaseskjemaet, vi tar med oss fra gammelt system for re-design.

I stedet for å la arkitekturen av våre systemer bli dominert av idéen om å kunne kjøpe deler fra ulike leverandører, må vi ta utgangspunkt i at ekte smidighet krever en homogen forretnings objekt modell som ikke kan kjøpes, men må designes innenfor enterprisen for å kunne reflektere de riktige forretningsbehovene[65]. Oppgavens re-design vil fullt ut støtte denne

[14] API— Application Programming Interface

tanken. Tilsvarende syn stemmer godt overens med Fowler og Evans oppfatning vedrørende anemiske domene-modeller som jeg nevnte under tjenesteorientert arkitektur (i seksjon 4.2.5).

4.2.6.2 Tjeneste (Façade) Laget

Nøkkelen til en sunn arkitektur er et veldefinert tjeneste lag[53], som eksponerer forretningslogikk til aktuelle klienter[32, 40]. Laget består typisk av ulike program-grensesnitt, med hver sin vel-definerte kontrakt. Laget er et egnet sted for å inkludere transaksjons-kontroll og sikkerhetsmekanismer[32]. Laget omtales også som "applikasjonslaget". Vi har følgende kjennetegn for et godt definert tjenestelag[53]:

Komplett for å eksponere alle operasjoner en klient trenger.

Enkelt med hensyn på den tekniske implementeringen.

Grensesnitt [15] defineres i stedet for direkte klasser i tråd med god objekt-orientert praksis.

Objekt-orientert med færrest mulig avhengigheter til spesielle grensesnitt eller utvidelser av klasser.

Presentasjonsteknologi uavhengig, slik at ulike teknologivalg kan brukes eller virke om hverandre.

Enkelt å skrive med fokus på maksimal produktivitet og reduserte kostnader.

Uavhengig av data aksess teknologien i underliggende lag gjennom løse koblinger via veldefinerte grensesnitt mot nevnte lag.

Transaksjonshåndtering slik at overliggende kallende lag, ikke skal bekymre seg om transaksjoner[32].

Horisontal skalerbarhet i den forstand at laget ikke hindrer klustring av applikasjons-tjenere.

Lett å teste og være gjennomgående testet. Forretnings-tjenestelaget (tjenestelaget) med sine forretnings objekter er den delen av applikasjonen som er viktigst å ha en gjennomgående testing av for å kunne utvikle kvalitets applikasjoner.

Tilstandsløst hvis mulig, og overlate til ovenforliggende lag å holde på tilstander. I forbindelse med skalering nevnt ovenfor, er tilstandsløse tjeneste lag lett skalerbare. Støtte for fjern-klienter ("remoting") forekles ved tilstandsløshet. Hvis tilstand holdes i domenelaget, hvilket er argumentert for i seksjon 4.2.6.1, anser jeg dog tilstand til å befinne seg i et underliggende lag eller som et lag på siden. Persisterte data er uansett i et underliggende lag.

[15] Grensesnitt—som i "interface" i programeringsspråk

Mange av kjennetegnene over argumenterer mot bruk av Enterprise Java Beans (EJB), og tilsvarende Microsoft's .NET Serviced Components. Riktignok gir sistnevnte mindre unødvendig programmering og kompleksitet ved deployering enn bruk av EJB[53].

Når tjenestelaget er tynt, gir det et grov-kornet grensesnitt til en fin-masket domenemodell[40]. Det fungerer da som en fasade over domene-modellen, og derav det alternative navnet; 'Façade'. Et slikt Façadelag vil jeg legge til grunn i re-design arbeidet. Et mønster omtalt som "Session Façade"[2] argumenterer for en slik fasade over forretningsobjektene for å unngå tett kobling mellom klienter og forretningslaget. Med et Façadelag på plass har vi lagt grunnen for enklere tjenesteorientering (seksjon 4.2.5), i tillegg til å være et veldefinert grensesnitt for ulike brukergrensesnitt. I dette laget kan vi også bygge inn ulike sikkerhetsmekanismer og andre enterprise tjenester som er tversgående("cross-cutting")[40]. Façade laget blir applikasjonens tilgangs-punkt for omverdenen.

Laget er svært viktig for å kunne realisere de ulike anvendelsesområdene vi trenger i dag, og gjøre det enkelt å innføre nye ved behov.

4.2.6.3 Presentasjons Laget

Presentasjonslaget kan være et brukergrensesnitt[32] eller en fasade for fjerntilgang ("remoting") som har alle operasjoner tilgjengelig fra Façadelaget[53].

En av de største endringer for enterprise systemer de siste årene er framveksten av web-baserte bruker-grensesnitt. Fordelene er enkel og rask deployering og ingen klient-programvare som skal installeres, felles grensesnitt-tilnærming, og aksess for alle med nettverkstilgang. Det eksisterer et godt utvalg av rammeverk som støtter enkel utvikling av slike grensesnitt for tynne klienter[32]. På den annen side har jeg erfaring fra referansegruppen i Forsvaret som indikerer at 50% eller mer av ressursene går med til å lage et godt grafisk presentasjonslag. En viktig poengtering er at den samlede ressursbruken er relativt lav allikevel om vi måler i person-måneder (se omtale av brukergrensesnitt i seksjon 5.3.3.1).

Mitt fokus på brukergrensesnitt for re-design er nettopp slike web-baserte grensesnitt, som er typisk for moderne enterprise informasjonssystemer. De vil kunne kjøres på lokale intranett eller være tilgjengelig på Internet når sikkehet for dette ivaretas.

Brukeropplevelsen blir ikke alltid like bra som for en rik eller tykk klient som er utviklet for en gitt PC eller arbeidsstasjon, men dette oppveies, etter mitt syn, av de nevnte fordeler, og avhenger selvsagt av behovet Det er mulig å bytte tynne klienter med rike klienter slik en av kravene for tjenestelaget var, men dette fordrer mekanismer og teknologi som sikrer fortsatt enkel deployering.

Skillet på rike klienter og typiske tynne web-klienter blir stadig mindre rent utseendemessig, og avhenger i stor grad av teknologien som benyttes, og hvor mye ressurser som legges ned i utformingen av klientene. Jeg mener at jo flere brukere og jo mere disse er avhengig av den aktuelle EIS, desto bedre grunn er det for å legge mere arbeid i å gjøre klienten så "rike" som mulig. Framveksten og utviklingen av rammeverk som tar hånd om mesteparten av det ekstra arbeidet som kreves, senker denne terskelen slik at det også for mindre systemer er regningssvarende å tilby rikere klienter.

Lagdeling av applikasjonen skal sikre en spesialisering av de oppgaver som en enterprise applikasjon som et minimum trenger. Presentasjonslaget's objekter

benytter grensesnitt som Façadelaget tilbyr. Presentasjonslaget kan gå direkte mot objektmodellen ved behov, etter å ha mottatt en referanse fra Façadelaget. Unntaket her er når vi tilbyr tjenestene utenfor applikasjonen via såkalte webtjenester (WS).

Som for andre lag er det hensiktsmessig med et tynnest (enklest) mulig lag for web-delen. Ofte blir web-applikasjoner utviklet med en web-del inneholdende både presentasjons og forretnings logikk, slik vi så for 2-lags klient-tjener arkitekturer i seksjon 4.2.2.1. Dette er spesielt tilfelle der hvor en mangler et veldefinert tjenestelag, og en bruker et MVC web rammeverk og view-komponenter som utfører tjenester som skulle vært skilt ut i egne lag. I slike tilfeller har utvikleren mulighet til å legge forretningslogikk i spesial-klasser som MVC-rammeverket bruker—såkalte action-klasser. Man oppnår i første omgang hurtig resultater, men på lengre sikt skaper dette problemer både for testbarhet og vedlikehold. Ytterligere problemer oppstår om vi skal bytte ut web grensesnittet eller skifte klienter-typer, eller det er behov for flere sameksisterende klient-typer[53].

4.2.6.4 Data Laget

Normalt vil Datalaget, som alternativt kan omtales "data aksess laget", forholde seg mot én enkelt database, men skopet kan senere utvides ved behov for å aksessere flere databaser og kanskje andre legasy systemer. Det er viktig å understreke at en slik utvidelse ikke foretas før behovet tilsier det, slik det poengteres i seksjon 3.2. For re-design-arbeidet forutsettes gjenbruk av databaseskjema fra kun én database. Konteksten her er systemarkitektur. Ulike teknologier kan benyttes for utvidelser av dette laget. Jeg går nærmere inn på noen eksempler i teknologi-kapitlets seksjon 5.3.1.1.

Vår objekt-modell vil ikke forholde seg direkte mot dette laget. Som nevnt (på side 58) brukes egne grensesnitt for slik kobling. Programvarearkitekturmessig er dette et lag med API'er mot databasen, som håndterer sømløs persistering av objektene fra domene modellen. Av den grunn er kjernen i datalaget mekaniskmer som følger *Data Mapper* mønsteret[32]. Teknologier for å understøtte dette laget og sikre enkelhet og vedlikeholdbarhet er beskrevet i seksjon 5.3.2.

4.3 Mønstre

Mønstre (eller "patterns" på engelsk) ble først beskrevet av bygnings-arkitekten Christopher Alexander:

> "Hvert mønster beskriver et problem som hender gang på gang i våre omgivelser, for så å beskrive kjernen for løsningen til problemet på en slik måte at løsningen kan gjenbrukes millioner av ganger uten å å måtte gjøre samme tingen om igjen"[38, [1]].

Mønstre finnes for ulike områder og på forskjellig detaljeringsnivå. Ved å kjenne og bruke mønstre innen aktuelle områder fremmes gjenbruk av løsninger på kjente problemer, i tillegg til å sikre en mer presis kommunikasjon mellom to parter[2]. Videre er mønstre viktige til å hurtig øke omfanget og effekten av vårt kognitive bibliotek[48].

Tilsvarende som forfattere gjenbruker mønstre som går igjen i suksessrike bøker og skuespill, gjenbruker gode designere og arkitekter mønstre som tidligere er brukt med hell. Mange mønstre finnes i objekt-orienterte systemer, hvor slike mønstre kan bidra til å løse spesifikke designproblemer ved å gjøre den objekt-orienterte designen mer fleksibel, elegant, og gjenbrukbar[38]. Andre typiske mønstre har vi for arkitektur-området, hvor jeg blant annet vektlegger mønstre for lagdeling og intern behandling av konkrete arkitektturlag.

Mønstrene kommer til anvendelse for objektteknologien, og virker i kombinasjon med applikasjonsrammeverk hvor mønstrene er implementert.

Til slutt omtales såkalte "antimønstre".

4.3.1 Arkitekturmønstre

Det eksisterer mange ulike arkitekturmønstre. Noen mønstre kan tilhøre både arkitektur og design, mens andre kan det være problematisk å kategorisere. Videre kan vi komme over designmønstre som like gjerne kunne vært kategorisert som arkitekturmønstre. J2EE-mønstre er nevnt fordi disse gir en kompletthet og et grunnlag for senere argumentasjon. MVC-mønsteret til Trygve Reenskaug[70], har jeg gruppert inn under arkitektur, men kunne like gjerne tilhørt designmønstre.

4.3.1.1 Generelle arkitekturmønstre

Fowler beskriver mønstre for arkitekturen til enterprise systemer, hvor disse er inndelt i ulike kategorier med tilhørende mønstre. Jeg har valgt de to kategoriene som omtaler lagdeling:

Domene Logikk omfatter mønstre for å organisere logikken innen et domene. I det objektorienterte paradigme, vil domene-modellen være et egnet sted for organiseringen, gitt at vi har kompleksitet i logikken som skal håndteres (domene-modellen ble også omtalt som et arkitekturlag i seksjon 4.2.6.1). Bruk av domene-modellen fordrer et annet mønster som må inkluderes for å kunne håndtere persistering mot relasjonelle databaser; "Data Mapper"[32].

Innenfor domene logikk, når denne bruker domene-modellen har vi mønsteret "Tjeneste lag", hvor en av to implementasjoner er i form av en fasade[32]. Denne fasaden er typisk når tjenestelaget er tynt og vi har en rik, ikke-anemisk, domene-modell.

Data Kilde Arkitektur Relasjons-databasen er den mest brukte persisteringsmekanismen i dag for informasjonssystemer (som vårt re-design tar utgangspunkt i), og representerer datastrukturen gjennom et skjema for relasjoner. For å kunne arbeide objektorientert, trenger vi ytterligere et skjema for objekter. Disse to skjemaene ønsker vi å overlate håndteringen mellom til et eget lag i arkitekturen, og her kommer mønsteret "Data Mapper" inn. Dette mønsteret flytter data mellom de to skjemaene og isolerer de fra hverandre. Mønsteret og tilhørende datalag kjenner ikke til domene-modellen i domenelaget. Et annet mye brukt navn på dette mønsteret er "Data Aksess Objekt", men som et J2EE-mønster er ikke domene-modellen framtredende i beskrivelsen[32].

4.3.1.2 J2EE Mønstre

En samling mønstre for Java 2 Enterprise Edition (J2EE) er beskrevet i [2]. Til tross for kritikk[53, 40, 76] mot disse mønstrene som går ut på at de fleste mønstrene er omgåelser av problemer forbundet med Enterprise JavaBeans (EJB), har Grady Booch og Martin Fowler kommentert disse mønstrene som et viktig bidrag innen utvikling av systemer for J2EE-plattformen. Men samtidig sier Fowler at vi ikke trenger Enterprise JavaBeans (EJB), men at vi kan lage et godt design ved hjelp av rene Java-objekter (POJO[16]) i stedet[32]. Jeg mener i tillegg at bruk av POJO krever alternative mekanismer som erstatter EJB-containeren. Dette kommer jeg tilbake til i form av lettvekter-container. Tilsvarende bruk av POJO er erfart i gruppen i Forsvaret gjennom de siste tre årene. I løpet av den tiden er ikke Enterprise JavaBeans anvendt eller funnet behov for innen informasjonssystemene. Utvikling, re-design, og vedlikehold utføres derved enklere.

Samlingen deler mønstrene inn i kategorier som representerer de tre typiske lagene som klassiske J2EE-systemer er bygd opp av, og som er identisk med 3-lags programvarearkitekturen: *Presentasjonslag*, *Forretningslag*, og *Integrasjonslag*.

Spesifikke arkitekturmønstre som ikke allerede er dekket, eller designmønstre som omtales i seksjon 4.3.2, gjør at jeg ikke omtaler andre spesifikke mønstre fra samlingen enn "Data Access Object" (DAO) og "Service locator".

Hovedformålet til DAO-mønsteret er å separere persistens-relaterte oppgaver fra generelle forretningsregler og arbeidsflyt[50, 2]. Vi finner tilsvarende i designmønsteret "Strategy", se 4.3.2. DAO regnes som et veletablert J2EE-mønster, og må kunne sies å gjelde uavhengig av teknologi-plattform. Forskjellen i de ulike beskrivelsene ligger i bruk av objekt-orientering og vektleggingen av domene-modellen.

"Service locator" kapsler inn tjenester og gjemmer implementeringsdetaljer, og gir derved en uniform tilgang til applikasjonens tjenester[2].

4.3.1.3 ORM-mønstre

Ambler omtaler knyttninger av objekter til relasjonsdatabaser i [7, 6], hvor flere mønstre relatert til tilordning mellom objekter og tabeller beskrives. Disse mønstrene kommer i tillegg til de tidligere omtalte arkitekturmønster i kategorien "Data Kilde Arkitektur" (se seksjon 4.3.1), som til sammenligning er på høyere abstraksjonsnivå.

Mange mønstre vil vi senere se, enten er inkludert i referansearkitekturen, eller implementeres i referanserammeverket som vil bli foreslått. Noen mønstre er ikke inneholdt andre steder, og medfører praktiske konsekvenser for vårt re-design:

Single Column Surrogate Keys Alle tabellene i basen gis en surrogat nøkkel for unik identifisering, slik at vi unngår unike nøkler som kan knyttes til problemområdet[6].

Dette mønsteret vil måtte implementeres manuelt ved behov.

[16] POJO—Plain Old Java Objects

Identity Field Ett hvert objekt må ha en unik identifikator. La denne tilsvare tabellens surrogatnøkkel[32].

Tilsvarende manuell utførelse som foregående mønster. Objekter er nytt for systemet som moderniseres, slik at dette steget blir obligatorisk.

Map Similar Types Sørg for at datatypen for et objekt's attributter ligger nærmest mulig datatypen til tabellens kolonner[6].

Representing Objects as Tables Re-design med utgangspunkt i at hver tabell tilordnes til en klasse[6]. Refaktorer om nødvendig på et senere tidspunkt. Dette utgangspunktet fordrer høy grad av normalisering i databaseskjemaet.

Databaseskjemaer av høy normaliseringsgrad vil, i tillegg til å hindre data inkonsistens ved at data kun lagres ett sted, ligge konseptuelt nærmere objekt-orienterte skjemaer. Årsaken ligger i et av de objektorienterte prinsipper om løse koblinger og sterk logisk sammenheng, som normalt gir høy normalisering av objekt-skjemaet[7, 5]. Generelt forenkles derfor knyttningen mellom objekter og tabeller når skjemaene på begge sider har en høy grad av normalisering. Dette kan gi utfordringer ved re-design hvis det opprinnelige skjemaet er på lav normalform. Idealet mener jeg bør ligge på BCNF[17] eller 4. normalform for relasjonsdatabasen. Grunnen til at jeg hevder dette, er at vi da kun tillater funksjonelle avhengigheter basert på supersettet til kandidatnøkkelen. Her mener jeg "Identity Field" med bruk av en uavhengig unik identifikator, dermed kan gi best effekt.

4.3.1.4 MVC

> "I created the Model-View-Controller pattern as an obvious solution to the general problem of giving users control over their information as seen from multiple perspectives."[70]
>
> — *Trygve Reenskaug*

Model View Controller (MVC), er et mønster som krediteres Trygve Reenskaug[70]. Essensen i mønsteret er å skille ut modellen, som kan sees på som dataene eller domene-modellen, og de ulike presentasjonene ("views"). Kontrolleren sørger for å holde view'et oppdatert fra modellen. På denne måten kan ulike typer grensesnitt legges til eller byttes ut uten at modellen endres. Effekten er både gjenbrukbarhet og fleksibilitet.

Fowler betegner separasjon av delene for presentasjon og modell, som en av de viktigste design-prinsipper for programvare[32]. Mønstret kan beskrives både som et arkitektur- og designmønster.

I Reenskaug's originale artikkel fra 1979, finner vi egentlig fire oppdelinger, hvor den siste (editor) er en spesialisering av kontrolleren. De ulike delene i MVC-mønsteret er[70]:

Models Modeller, helst i form av strukturete objekter, representerer kunnskap i et problem-område. Modell-nodene bør representere identifiserbare deler av et problem, og de bør være på det samme problem-nivå.

[17] BCNF—Boyce-Codd Normal Form

Views Et view er koblet mot hele eller deler av modellen, og representerer en visuell betrakning og et presentasjonsfilter. View'et mottar sine data gjennom spørringer mot modellen, og kan oppdatere modellen gjennom meldinger. View'et må således kjenne semantikken til modellens attributter.

Controllers En kontroller er linken mellom brukeren og systemet. Den leverer innhold til brukeren ved å arrangere for at riktig view presenterer seg på brukeren's skjerm. Den gir mulighet for brukeren til å velge fra menyer, utføre kommandoer og gi data inn til systemet. Ut-verdier fra brukeren oversettes til meldinger som overføres til aktuell(e) view.

Kontrolleren bør ikke supplere view'ene, hvilket betyr at view'ene er agnostisk til kontrolleren men ikke omvendt.

Editors En editor er en spesialutgave og en midlertidig forlengelse av en kontroller, som gir brukeren mulighet til å modifisere informasjon presentert via et view.

I forbindelse med web-applikasjoner er rammeverk basert på MVC-mønsteret meget utbredt. For enterprise informasjonssystemer med web-grensesnitt, vil jeg hevde at dette mønsteret er obligatorisk, og må av den grunn være en del av rammen for arkitekturen innen re-design aktiviteten.

For re-designet EIS kan vi bruke følgende beskrivelse for å eksemplifisere: *Modellen* er typisk domene-modellen. *View*-komponenten's oppgave er typisk å oversette de aktuelle delene av informasjon fra domene-modellen til HTML-sider. *Kontrolleren* mottar HTTP forespørsler fra brukeren, tolker og videresender til riktig del av domene-modellen (bruker Façadelaget), og viser resultatet fra modellen gjennom riktig utvalgt view-komponent ut til brukeren som HTTP svar, generert som HTML. En variant av view'et kan gi andre anvendelsesområder som portlets, eller alternativ visningsteknologi som XML.

Den interne lagdeling som dette mønstret gir innenfor et presentasjonslag, gjør det enklere å delegere oppgaver mellom grafisk designere som designer og arbeider med HTML-sider for view-komponentene, og systemutviklere som tar hånd om modellen, foruten de andre lagene i selve applikasjonen. Denne interne lagdeling skal vi i seksjon 5.3.3 se teknologisk støtte til i form av rammeverk (som tar seg av kontrolleren).

4.3.2 Design Mønstre

Vi har ulike mønstre for design, og en kjent samling med designmønstre kommer fra "The Gang of Four" (GoF)[38]. GoF har katalogisert viktige design mønstre, med standardiserte navn og definisjoner på brukte teknikker. Designmønstrene kan gi løsninger på kjente felles design problemer. De kan også oppfattes som omgåelser av problemer forbundet med `C++`[40, 76]. Denne oppfattelsen av omgåelser av problemer forbundet med `C++`, mener jeg ikke kan være tilfelle for alle mønstrene GoF beskriver, og spesielt ikke de mønstrene jeg lister opp nedenfor. De anvendes av mange, blant annet innen referansegruppen jeg omtaler fra Forsvaret (se appendiks A), ved bruk av språket Java gjennom implementasjoner i ulike Java-baserte applikasjonsrammeverk med stor utbredelse. Om påstandene om mønstrene som omgåelser av problemene

vedrørende C++ skulle delvis stemme, fremmer mønstrene om ikke annet presis kommunikasjon, og kunne brukes som "antimønstre" i aktuelle tilfeller.

En betrakning av de nevnte mønstre fra GoF, viser at de enten har nedslagsfelt på klasser eller objekter, hvilket henspeiler på at mønstrene er tiltenkt anvendelse innen objektorientering—objektorientert design.

4.3.2.1 GoF-mønstre

GoF har tre kategorier av design mønstre, hvorav jeg har plukket ut de konkrete mønstre som er aktuelle for oppgaven[38]:

1. Instansierings mønstre.

   Singleton Sikrer at en klasse bare har en instans, og gir et globalt punkt for aksess. Dette mønsteret er sentralt i enterprise systemer.

   Det er laget rammeverk (se seksjon 5.4.3.2) som gjør det svært enkelt å ta i bruk mønsteret.

2. Strukturelle mønstre.

   Composite Objekter komponeres i tre-strukturer for å representere hierarkier av hele/deler, slik det for øvrig alltid har eksistert innen objektorienteringen. En spesialisering av mønsteret finnes i *MVC* som beskrevet i seksjon 4.3.1.4.

   Façade mønsteret har vi sett eksempel på under lagdelingen av arkitektur i seksjon 4.2.6.2. Generelt skal det gi et felles høynivå grensesnitt til et sett av underliggende grensesnitt, og forenkle bruken av disse.

3. Oppførsels mønstre.

   Observer mønsteret's formål er å sørge for at objekter som endrer tilstand og har en-til-mange avhengighet til andre objekter, underretter og automatisk oppdaterer alle avhengige objekter.

   Mønsteret er også kjent som "publish subscribe", og kan anvendes innen tjenesteorientert arkitektur.

   En spesialisering av mønsteret finnes i *MVC*-mønsteret.

   Strategy Et objekt som representerer en algoritme. Eksempel er relasjonen mellom view og kontroller komponentene i *MVC*-mønsteret.

4.3.2.2 "Avhengighetsinnsprøytning"

> Hollywood-prinsippet: "Don't call us - we call You!"

Avhengighetsinnsprøytning[18] eller "Inversion of Control" (IoC), er et mønster hvor prinsippet er å separere konfigurasjon og bruk[34]. Avhengighetsinnsprøyting er ikke eneste måten å håndtere avhengigheter. Tradisjonelt har J2EE brukt "Service locator"[38, 2] som vi har vært inne på.

[18] Avhengighetsinnsprøytning er en norsk oversettelse jeg benytter for "Dependency Injection".

Kort fortalt reduserer mønsteret avhengigheter og sikre løse koblinger slik at programmering mot grensesnitt blir enklere, i henhold til god objektorientert praksis[34]. Dette er ytterligere beskrevet som programmeringsmodell i teknologi-kapittelets seksjon 5.1.1.

Et praktisk eksempel på *avhengighetsinnsprøytning* kan vi finne fra virksomheter som ønsker å redusere antall ansatte:

> Anta at en oppgave utført av en organisasjonsenhet på vegne av alle ansatte, forsatt skal utføres etter nedleggelse av denne enheten. Resultatet kan bli at hver enkelt må lære detaljene forbundet med oppgaven, og stadig re-lære etter som reglene endres, hvilket må gå på bekostning av den tid og krefter de egentlig skulle anvendt for sine egentlige oppgaver. Når en ny oppgave dukker opp som de ansatte har behov for eller blir pålagt å utføre, må også denne løses individuelt (og ofte ikke like bra) dersom vi ikke har *avhengighetsinnsprøytning* eller tilsvarende organiseringsformer.

4.3.3 AntiMønstre

Mange mønstre er omgåelser av problemer vi i utgangspunktet ikke skulle kommet opp i [53, 40]. Og like viktig som å kjenne og bruke gode mønstre, er det å kjenne å unngå å bruke mønstre som kan gi flere problemer enn de løser. Dette omtales som antimønstre[21]. To relevante antimønster for oppgaven tilhører henholdsvis arkitektur og design:

Vendor Lock In [21] er et antimønster som beskriver mulig leverandør-låsing når et programvareprosjekt gjør seg avhengig av et produkt fra én bestemt leverandør. Produktet kan være basert på åpne standarder, men prosjektet gjør bruk av spesifikke utvidelser. Problemer kan oppstå som en følge av oppgraderinger i leverandørens implementasjon ved at prosjektets programvare må endres, og som videre kan gi interoperabilitets problemer, forsinkelser, og fravær av forventet funksjonalitet.

Hvis en slik leverandør-låsing er uunngåelig, er en løsning som mønsteret beskriver, å innføre et isoleringslag som separerer den leverandøravhengige teknologien fra andre programvare-pakker.

Et kjent unntak for dette antimønstret er når én leverandørs kode utgjør majoriteten av kode som er nødvendig for en applikasjon. Men dette unntaket kan også lede til at vi egentlig ikke har noen arkitektur.

Funksjonell dekomponering [21] er et antimønster for design av programkode. Antimønsteret finnes typisk i "legasy" systemer som kodes i et objektorientert språk, men hvor det benyttes et "ikke-objektorientert" design.

Løsningen er å re-designe ved bruk av objektorienterte prinsipper, slik at fordelene ved objektorientering nyttegjøres.

I de ulike oppjusterings-alternativer for informasjonssystemer som kort ble omtalt i seksjon 2.2, kan flere av forslagene rammes av antimønsteret "Vendor Lock In". Effekten av å rammes kan være at senere vedlikehold og oppdateringer kan bli uforholdsmessig kostbare.

Til bruk i argumentasjonen for metodikkforsterker har jeg valgt å ta med mønsteret "Funksjonell dekomponering".

Vi vil alltid kunne oppleve grader av låsing, selv ved bruk av et uavhengig rammeverk, eller et rammeverk vi selv utvikler. Et uavhengig rammeverk til tross, når koden bruker rammeverket har vi en låsing. Men rammeverket kan til gjengjeld fremme uavhengighet for andre områder i en grad som oppveier dette. For eksempel kan rammeverket gjøre arkitekturlag eller komponenter enklere å skifte ut ved behov. Hvilket medfører at vi lettere kan gjøre endringer når behovet senere oppstår.

4.4 Referansearkitekturen

Til å støtte metodikken vi bruker gjennom å gi oppstarten av et re-design prosjekt høy produktivitet og en egnet arkitekturramme som forenkler videre re-design prosess, vil jeg sette opp en hensiktsmessig referansearkitektur. En drøfting om dette kan ansees som en metodikk-forsterker følger i neste seksjon.

Fordelen av en fornuftig stabel av erfaringsmessig gode lag er at vi senere kan sette opp en implementerbar samling av rammeverk som implementerer referansearkitekturen[53, 32, 40]. Dette bidrar til raskere oppstart for et utviklingsprosjekt (også av typen re-design). Jeg vil videre i oppgaven omtale en slik ferdig implementert referansearkitektur som "referanserammeverk". Referanserammeverk omtales under teknologi i det etterfølgende kapittelet.

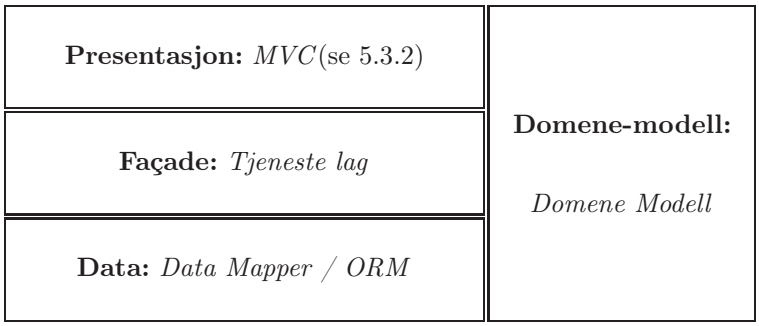
Som nevnt i referansemetodikken, har jeg gjort ett brudd med XP på arkitektur-området. Min tilnærming til effektiv re-design av EIS er å ta utgangspunkt i en referansearkitektur som jeg mener kan være generell for mange enterprise informasjonssystemer. Samtidig kan en slik referansearkitektur sees på som en obligatorisk brukerhistorie som tas fra planleggingsspillet, og som derved allerede rommer hele den initielle arkitekturen i form av et vertikalt snitt av systemet med hele arkitekturen ferdig refaktorert. Etter at alle forretningsregler er implementer, og nye anvendelsesområder er etablert slik kravene tilsier, skal systemet, og derved arkitekturen refaktoreres slik at vi sitter igjen med med et re-designet enkelt vedlikeholdbart system. Med hensyn på vedlikeholdbarhet; om vi har mange EIS som skal moderniseres, kan det være hensiktsmessig å ikke trimme ned den ytre strukturen i referansearkitekturen. Dette fordi gjenkjennbarheten mellom ulike EIS på arkitektur-planet derved bedres, og kan dermed gi grunnlag for enklere vedlikehold gjennom at arkitekturens iboende kunnskap gjenbrukes og er kjent av flere. Dette er en praktisk erfaring jeg har sett fra referansegruppen i Forsvaret (se appendiks A).

En egnet referansearkitektur for web-baserte enterprise informasjonssystemer, er utledet gjennom å kombinere fordeler fra kjente arkitekturer som passer nedslagsfeltet for oppgaven.

Lagdelingen som referansearkitekturen vil bestå av er beskrevet i seksjon 4.2.6, og kan oppsummeres med **navn på lag** og tilhørende *navn på mønstre* (fra seksjon 4.3.1) i figur 4.5.

Tar vi unionen av arkitekturlagenes fordeler, satt inn i konteksten av de fire premissene, gis følgende idealisering og krav:

Raskere Referansearkitekturen skal støtte referansemetodikken for re-design ved at utførelsen går raskere, og at senere vedlikehold og deployering kan



Figur 4.5: Referanse programvare-arkitektur.

skje hurtig og effektivt:

- Rask igangsetting og gjennomføring av av prosjektet.
- Raskt å endre og utplassere nye versjoner.
- God ytelse gjennom bruk av lag som støttes av enkle containere (se teknologi).
- Mulighet for enkel skalering ved behov gjennom tilrettelagt lagdeling.

Selve applikasjonen må i tillegg ha lav responstid for brukeren.

Bedre Referansearkitekturen skal legge grunnlaget for et bedre system, både for sluttbrukeren, virksomheten, og de som utvikler det:

- Veldefinert tjenestelag som sikrer gjenbrukbare tjenester.
- Forretningslogikk kan oppdateres uavhengig av presentasjonslaget.
- Tilstrekkelig spesialisering av de ulike lagene slik at endringer i ett lag ikke får følger for andre lag, samtidig som vi holder antall lag på et minimum.
- Fremme anvendelsesområder, som i tillegg til web-basert brukergrensesnitt, enkelt kan gi portlets i portaler, og integrasjon på høyere abstraksjonsnivå.
- Tilgjengelighet og egnethet for både intra- og Internet gjennom å være agnostisk til brukernes arbeidsstasjoner eller PC'er.
- Fremme god OO praksis.
- Gjøre det lettere å teste samtlige lag.
- Moderne systemer gir utviklernes arbeide høyere status, og er mere tilfredsstillende å jobbe med.

Billigere Re-design arbeidet kan bli billigere når en gjennomprøvd referansearkitektur legges til grunn. Spesielt videre vedlikehold har tradisjonelt vært kostbart, og var en viktig driver for å velge re-design.

- Vedlikeholdbarhet og utvidbarhet. Vedlikehold har tradisjonelt vært den desidert mest kostbare delen i livssyklusen for programvare.

Vedlikeholdbarhet og utvidbarhet er i stor grad avhengig av et rent design. Vi må forsikre oss om at hver komponent i applikasjonen har et veldefinert ansvar, og at vedlikehold ikke hindres av tett koblede komponenter.

- Lavere TCO (Total Cost of Ownership) for eierne.

Enklere Referansearkitekturen skal bidra til å forenkle re-design prosessen, og gjøre systemet enklere:

- Gjøre ting så enkelt som mulig. Redusert kompleksitet øker produktiviteten, hvilket ekskluderer distribuerte objekter og unødvendige og kompliserende arkitekturteknologi-komponenter som EJB.
- Enkel deployering og administrasjon gjennom sentralisering.
- Testing er en vital aktivitet gjennom hele syklusen. Enkel testing oppnås ved at alle lag kan testes uten avhengigheter til andre lag, og at "Vendor Lock In" unngås i størst mulig grad. Referansearkitekturen støtter design beslutninger som forenkler testing.
- Arkitekturen skal ikke dekke alle eventualiteter, men bestå av en stabel av fornuftige arkitektoniske lag som vi enkelt kan plugge ny funksjonalitet inn i, og fjerne funksjonalitet fra om behovet for slik funksjonalitet ikke lengre er til stede.
- Gjenbruk. Enterprise informasjonssystemer må passe inn i organisasjonens langsiktige strategi. Av den grunn må vi legge til rette for gjenbruk, slik at duplisering av kode minimeres både innen prosjekter og mellom disse. Gjenbruk av kode er vanligvis resultatet av god OO design.

Mange av disse premissene krever ytterligere støtte utover metodikk og arkitektur for å nå idealet. Denne støtten, i form av den tredje dimensjonen, kommer vi nærmere inn på i kapittel 5.

Til tross for at mange rammeverk støtter enkel bygging av grafiske grensesnitt[32], så er min erfaring at det i praksis krever mere tid og ressurser å lage gode web-baserte brukergrensesnitt, og spesielt når grensesnittene skal være rikest mulig til beste for brukerne. Dette til tross, jeg mener ressursbruken oppveies av fordelene med sentralisering, hurtig deployering, raskere endringer og tilgjengeliggjøring ut til brukeren, og en arkitektur som reder grunnen for nødvendige og nye anvendelsesområder.

Referansearkitekturen som er omtalt er utprøvd og brukes i referansegruppen i Forsvaret (se appendiks A) både for nyutvikling og re-design med godt resultat, og vil fortsatt benyttes som preferert referansearkitektur for flere enterprise informasjonssystemer i overskuelig framtid. Erfaringene har vist at referansearkitekturen også passer inn i deler av det SOA-arbeidet som i økende grad nå vektlegges av Forsvaret. Etter min mening gir en god velprøvd referansearkitektur tilsvarende støtte til normalt dyktige utviklings-team, som en tilstrekkelig metodikk gjør. Har vi en referansearkitektur som erfaringsmessig er god—trenger vi ikke å feile på dette området, men gjenbruker et godt "mønster".

4.5 Metodikk-forsterker?

Så til det første store spørsmål (det andre kommer i teknologi-dimensjonens seksjon 5.6): Er referansearkitekturen, eller deler av den, en forsterker for den angitte referansemetodikken, gitt i min definisjon av metodikk-forsterker i seksjon 3.3?

En forutsetning er at referansearkitekturen gir et positivt bidrag for å kunne hjelpe og være et hjelpemiddel. Dette mener jeg er vist gjennom å være en guide til å velge en erfaringsmessig riktig og god arkitektur for re-design av EIS. En ulempe er at den *kan* gi en begrensning i frihet og "låsing" som ikke ville vært til stede om vi sto helt fritt, slik ekte XP legger opp til. Dette har jeg argumentert for gjennom å vise til at den anbefalte arkitekturen erfaringsmessig er dekkende, og gir bidrag til enklere å øke anvendelsesområder og berede grunnen for nye.

Videre må en forsterker virke i prosessen som starter ved at metodikken (eller referansemetodikken) er i bruk. Forut for prosessen har vi et tidspunkt hvor tanken om mulig re-design oppstår, tiden T_{-n}. Fra T_{-n} har vi en tidsperiode med blant annet vurderinger og kost/nytte analyse, som skal gi svar på om et eksisterende system skal moderniseres. Denne tidsperioden, forut for når metodikken gjelder, velger jeg å kalle $T_{-n...-1}$. Gitt at prosessen i perioden $T_{-n...-1}$ konkluderer med oppstart av et re-design-prosjekt, kommer metodikken til anvendelse fra tiden T_0, og varer fram til systemet termineres på tiden T_n. Referansemetodikkens tid for anvendelse ligger dermed i tidsintervallet $T_{0...n}$. Tiden etter livsyklus-håndteringen, $T_{n+1...\infty}$, er dermed irrelevant i denne sammenhengen. Tidsperioden $T_{-n...-1}$ er følgelig problematisk for å kunne fastslå om metodikken forsterkes, fordi det som kunne angis som forsterker, trolig egentlig gjelder forut før metodikken kommer til anvendelse. Vi har altså at metodikken x anvendes i tidsperioden:

$$\forall x \mid x \geq T_0 \wedge x \leq T_n \wedge x \notin T_{-n...-1} \wedge T_{n+1...\infty})$$

Jeg vil nå gi en gjennomgang av de ulike elementene som referansemetodikken består av, og argumentasjon for om støtte eller forsterkning er til stede:

Planlegg for endring—Ut-flatet kost-kurve Vi hadde de tre faktorene; objektteknologi, lett modifiserbar kode gjennom rent design og automatiserte tester, og erfaring og mot til å modifisere. Rent design her henspeiler på kodedesignen hvor gode designmønstre kan hjelpe. Designmønster til tross, vi kan fortsatt skrive dårlig kode. Tilsvarende gjelder for objekter. Selv med designmønstre som er tiltenkt støtte innenfor objektorientering kan vi feile med koden i tråd med antimønsteret "Funksjonell dekomponering". Mulig støtte er derimot gitt. Automatiserte tester gis ikke av arkitekturen i seg selv, men den vil kunne legge grunnlag for at disse lettere kan settes opp som autonome, gjennom en hensiktsmessig arkitektur. Mot og erfaring ligger utenfor referansearkitekturens område.

Enkelhet Denne delen av referansemetodikken skulle være en god kandidat, etter som den forutsetter en referansearkitektur for hurtig oppstart. Spørsmålene er om den hurtige oppstarten gjelder i tiden $T_{0...n}$, og om forsterkning kan oppnås bare ved hjelp av en *beskrevet* arkitektur?

Med hensyn på tidsintervallet hvor metodikken anvendes, er jeg usikker på om jeg kan hevde at en gitt referansearkitektur for re-design-oppgaven,

er introdusert før T_0 eller ikke. Som en støtte til oppgavens metodikk gjennom at vi på forhånd vet hvordan mål-arkitekturen skal være for å gjennomføre re-design er svaret ja, men jeg vil uansett ikke hevde at vi har en reell forsterker i referansearkitekturen alene.

Teststrategier Riktig og hensiktsmessig lagdeling vil være et grunnlag for enklere testing, men også her vil jeg hevde at referansearkitekturen er mere i form av støtte enn som en forsterker. Men som nevnt ovenfor, gir ikke arkitekturen automatiserte tester, men hensiktsmessig lagdeling legger grunnlaget.

Refaktorering Denne teknikken berøres i liten grad av arkitekturen, men gir derimot et viktig bidrag tilbake *til* designet (og arkitekturen).

Team-størrelse og samlokalisering Uavhengig av arkitekturen.

Brukermedvirkning Uavhengig av arkitekturen.

Dokumentasjon Er i utgangspunktet uavhengig av arkitekturen. Men gode metaforer og en oversiktlig enkel arkitektur, forenkler overføringen av kunnskap, og kan bidra til mindre behov for dokumentasjon. Spesielt når flere EIS'er har samme grunnleggende arkitektur, vil denne være enklere å formidle.

Risikohåndtering En god og hensiktsmessig referansearkitektur vil etter min mening gi et reelt bidrag til å redusere risiko i et utviklingsprosjekt. Spørsmålet er om referansemetodikken anvendes på det tidspunkt hvor denne risikoreduseringen skjer. Ved å si at vi i forkant av re-design prosessen ($T_{-n...-1}$) beslutter bruk av den aktuelle arkitekturen er det ikke en forsterker, men om vi kan rettferdiggjøre at bidraget kommer etter at metodikken er kommet til anvendelse ($T_{0...n}$), er dette etter min menig en konkret forsterker for referansemetodikken. Jeg mener det er grunn til å hevde begge deler.

Oppsummering er gitt i tabellen 4.1 hvor jeg har valgt å ikke avgrense *metodikk-støtte* (Støtte) i tid. *Metodikk-forsterker* er forkortet til MF.

Metodikk-del	**Støtte**	**MF($T_{-n...-1}$)**	**MF($T_{0...n}$)**
Planlegg for endring...	Ja	Nei	Nei
Enkelhet	Ja	Ja	Kanskje
Teststrategier	Ja	Nei	Litt
Refaktorering	Nei	Nei	Nei
Team-størrelse...	Nei	Nei	Nei
Brukermedvirkning	Nei	Nei	Nei
Dokumentasjon	Litt	Nei	Nei
Risikohåndtering	Ja	Ja	Kanskje

Tabell 4.1: Arkitektur metodikkforsterker?

Basert på de foregående vurderinger, er jeg usikker på om jeg vil karakterisere referansemetodikken som en reell forsterker til referansemetodikken. Et argument *for*, er som nevnt at vi drar nytte av en bestemt lagdelingen i forkant,

slik som gode mønstre kan hjelpe å gi beste praksis for et gitt problem. Argumentet mot slik forutbestemt lagdeling, er at den også *kan* gjøre oss mindre fleksible. Dette mener jeg oppveies av fordelene.

Et annet argument *for*, er at lagdelingen skal være sentralisert, hvilket også kan bestemmes i forkant av re-design prosessen. Tilsvarende argument gjelder når jeg i referansearkitekturen velger å ekskludere EJB, som jeg anser som en miks av arkitektur og teknologi.

Et opplagt argument *mot* den lagdelingen jeg forfekter, er at bruken av en slik referansearkitektur i forbindelse med re-design av enterprise informasjonssystemer, delvis eller fullstendig viser seg feilslått. Med utgangspunkt i de gitte mønstre[32, 2, 38, 34, 70] som langt på vei støtter slik lagdeling, andres[53, 40] og egne erfaringer, anser jeg å stå på trygg grunn. Jeg ser likevel muligheten for problemer ved at ikke alle EIS'er er like og kan komme til å kreve andre ting fra arkitekturen som jeg ikke har dekket eller forutsett. Jeg mener til tross for det, at referansearkitekturen i de fleste tilfeller kan gi et fornuftig utgangspunkt.

En annen faktor har vi ved vurdering av arkitekturen generelt som et kunnskapsområde. Da kan vi vanskelig hevde at den ikke forsterker metodikken. Fordi; uansett hvor god metodikk vi anvender, og eventuelt ikke gjør bruk av, så kan det få svært uheldige konsekvenser om vi ikke tar inn over oss fordeler og ulemper vedrørende arkitekturen. En god metodikk brukt i et prosjekt som feiler fullstendig på arkitekturvalget, kan ikke lykkes slik jeg ser det. Men vi kan jo også vurdere slik kunnskap som en selvfølgelig forutsetning, og derved ikke anse arkitektur, som kunnskapsområde, som en forsterker på generell basis.

Jeg velger å utsette endelig konklusjon til teknologi-dimensjonen er behandlet. Foreløbig konklusjon fra min side er at arkitekturen uten hjelp fra teknologi absolutt er en støtte, men usikkert med hensyn på om den kan anses som en reel forsterker for referansemetodikken.

Forsterker eller ikke, referansearkitekturen er en forutsetning for at vi skal kunne sette opp et referanserammeverk. Om dette faktisk er en metodikkforsterker gjenstår å se i teknologi-dimensjonen i kapittel 5.

Kapittel 5

Teknologi

> "Do The Simplest Thing That Could Possibly Work"[9]
>
> — *Craig Larman*

Systemer som begynner å gå ut på dato, er kjørbare og brukes. Om vi ønsker å re-designe disse enterprise informasjonssystemene, må de kunne kjøre og brukes. Til det trenger vi teknologi. Uten teknologi blir metodikk og arkitektur en akademisk men viktig øvelse, som ikke gir virksomheten og brukerne den data-støtten de trenger—en av de reelle beveggrunner for systemutvikling. Tilsvarende som det gir mindre mening hvis vi ikke evner å forstå hva vi skal gjøre og hvordan vi skal komme dit.

Problemet med konkrete implementeringer av teknologien, er at bildet endres kontinuerlig, slik at det "beste" for kort tid siden (ett år?), ikke nødvendigvis er best i dag. Dermed kan teknologi-implementeringer bli en form for "ferskvare", hvor betydningen raskt svekkes. Av den grunn har jeg forsøkt å legge konkrete implementeringer til slutten av kapitlet, og omtaler teknologi i mere generelle termer først.

Mitt mål med teknologien sett fra et virksomhetsperspektiv med fokus på budsjetter, er at den er billig og leverandør-uavhengig (leverandør, lisens, og verktøy-uavhengighet). Jeg velger å unngå kommersielle produkter hvis teknologiunderstøttelse kan oppdrives gjennom fritt tilgjengelige produkter, og med bakgrunn i en undersøkelse utført av Stadish-Group[75], kan vi i mange tilfeller finne vel så gode produkter av typen "åpen kildekode". Etter min mening er det ikke avgjørende om produksjonsplattformen er av kommersiell art eller i form av åpen kildekode. Viktigere er det å utvikle for, og ha en kodebase som kan kjøres på den plattformen man velger, har, eller ønsker å investere i, og som enkelt og billig lar oss skifte om forholdene tilsier det. Her ligger mulige inntjeningspotensialer for enterprisene, hvis store deler av IT-budsjettene er bundet opp mot disse tre avhengighets-faktorene. Noen avhengigheter kommer vi ikke bort fra, men disse behøver ikke å være knyttet opp mot én leverandør og lisens-utgifter for å være funksjonelle og gode.

Jeg vil i dette kapittelet gå inn på de teknologier som jeg tidligere har referert til. Videre vil jeg vise hvordan teknologien kan inngå i den referansearkitekturen som allerede er foreslått.

Målet med kapittelet er både å gi forslag til teknologi i generelle termer som kommer til anvendelse for re-design og nyutvikling, og i form av et konkret forslag til referanserammeverk til utprøving om dette oppfyller kravene til en metodikk-forsterker. Kapittelet er delt inn i følgende seksjoner:

Objektorientering I seksjon 5.1 vil jeg gå inn på objektorienteringen, det paradigme jeg legger til grunn for re-design av oppgavens enterprise informasjonssystemer. Videre omtales noen støtteteknologier som kan forbedre objektorienteringen på visse områder.

Persistering I seksjon 5.2 beskriver jeg persisterings-muligheter i form av databaseteknologier. Oppgavens re-design og avgrensning medfører at det laveste laget i referansearkitekturen baserer seg på relasjonsdatabaseteknologien.

Mellomvare I seksjon 5.3 omtales mellomvare. Mellomvareteknologi er nødvendig for å understøtte den aktuelle lagdelingen vi ønsker for forbedret programvarearkitektur gjennom re-design, og er en forutsetning for å enkelt øke anvendelsesmulighetene gjennom webtjenester og høynivå integrasjon.

Åpen kildekode I seksjon 5.4 omtaler jeg et alternativ for teknologileveranser hvor vi ikke betaler for produktene, eller som medfører årlige lisenskostnader, nemlig åpen kildekode. Dette er et sterkt voksende segment, og siden oppstarten av min oppgaveskriving har vi attpåtil hatt et regjeringsskifte hvor IT-meldingen[30] ønsker å favorisere åpen kildekode.

Referanserammeverk I seksjon 5.5 vil jeg foreslå et referanserammeverk som implementerer referansearkitekturen.

Referanserammeverket jeg angir, kan også ansees som veiledende i andre sammenhenger, som nyutvikling.

Metodikk-forsterker? I seksjon 5.6 avslutter jeg med argumentasjon, tilsvarende som jeg gjorde for arkitektur, for om teknologi metodikk-forsterkeren kan godkjennes eller må forkastes.

5.1 Objekt Orientering

Programmeringsspråk har utviklet seg fra 1. generasjons språk i maskinkode. Forbedringer kom med 2. generasjons assembler språk og hullkort som representerte koden, før ulike tekstlige 3. generasjons programmeringsspråk kom innen genrer som funksjonelt orientert og prosedyreorientert. Norge sto for et gjennombrudd med objektorienteringen. Utviklingen gikk videre til ulike 4. generasjons språk, og man snakket etter hvert om 5. generasjons logikk orienterte språk, og kunstig intelligens. 3. generasjons språk (3GL) av typen objektorienterte, er igjen de dominerende. Vi ser også retninger hvor målet er å generere kjørbare systemer ut fra modeller, men den kjørbare koden er oftest 3GL.

Objektorienteringen kommer fra Norge. Forskerduoen Kristen Nygaard og Ole Johan Dahl, som begge kom fra Forsvarets Forskningsinstitutt til Norsk Regnesentral, regnes som opphavsmennene til objektorienteringen. Med

utgangspunkt i språket Algol, utviklet de på 60-tallet Simula og Simula67, som senere dannet grunnlaget for det mer kjente språket Smalltalk fra XEROX Park, som kom i sin første versjon i 1972. I 2002 fikk Kristen Nygaard og Ole Johan Dahl tildelt informatikkens nobelpris, A.M. Turing Award, for sitt arbeide.

Objektorientering (OO) er stadig et sentralt paradigme for både analyse (OOA), design (OOD), og programmering (OOP). Jeg vil påstå at det faktisk er enda mere sentralt i dag enn noen gang. Fordelen med det objektorienterte paradigme, er at det på en god måte kombinerer styrkene fra de data-sentriske og oppførsels-sentriske tilnærmingene. Allikevel er det ikke noe universalmiddel idet det fortsatt er lett å lage dårlige systemer og vanskelig å lage gode[66].

I tillegg til å kombinere tilnærmingene fra data-sentrisk og oppførsels-sentrisk oppførsel, slik at vi har objekter med både tilstand (attributter) og metoder/operasjoner (oppførsel), gir objektorienteringen støtte for å oppnå løse koblinger og sterk kohesjon, tilsvarende et av målene for lagdeling i arkitekturen. Sterk Kohesjon (logisk sammenheng) i en komponent har vi når den representerer én enkelt del av problemløsningen. Løs kobling mellom komponenter oppnår vi ved at detaljene til komponenten holdes skjult for omverdenen, og at komponenten bare kan nås via veldefinerte grensesnitt[72].

Innen perspektivet for enterprise informasjonssystemer, har vi behov for enterprise funksjonalitet som beskrives i seksjon 5.3. Om vi skal kunne oppnå slik funksjonalitet uten en mastodont av en komplisert og kostbar mellomvareplattform, trenger vi støtteteknologier som kan hjelpe der OO ikke er perfekt, slik som Aspekt Orientert Programmering (AOP) og "avhengighetsinnsprøytning" (som vi også har sett mønster for).

5.1.1 "Avhengighetsinnsprøytning"

Avhengighetsinnsprøytning, eller IoC, er en felles karakteristikk for lettvekts-containere[34], hvor hovedkontrollen for programmet flyttes ut til et rammeverk. Som teknologi er den basert på tidligere omtalt mønster av samme navn.

Mønsteret støtter løse koblinger, forenkler lagdeling, testing, vedlikehold, og senere utvidelser. Rammeverkene (eller lettvekts containerne) skiller ut avhengighetene i form av konfigurasjonsfiler. Et faresignal det er verdt å merke seg er om antall tekst-linjer til konfigurering blir anseelig, hvilket kan øke kompleksiteten med å forstå systemet i form av kodelinjer og konfigurering.

Avhengighetsinnsprøyting er en virkningsfull programmeringsmodell når vi programmerer mot grensesnitt og lar en 3. part sprøyte inn avhengigheter for oss[76]. Det maksimaliserer løse bindinger (som i god OO-ånd sier; løst koblede objekter/komponenter med sterk logisk sammenheng—"loose copling, strong cohesion"). Sett opp mot arkitektur og lagdeling, gir dette mønsteret implementert i teknologien i form av rammeverk, god støtte for våre design-valg. Vel-definerte grensesnitt forenkler utvikling og vedlikehold av applikasjoner på sikt ved at implementasjoner kan endres uten at komponenter og tjenester som er avhengig av disse (for eksempel brukergrensesnittet) påvirkes. Derimot vil selvsagt endringer i metodesignaturer i grensesnittene måtte føre til endringer der hvor avhengigheter er knyttet mot disse. Sistnevnte taler for å bruke litt ekstra tid på definering av grensesnittene slik at disse om mulig kan holdes mest mulig konstant.

Fordelen med IoC strategien er at den lar oss skille ut avhengigheter fullstendig fra applikasjonen og omgivelsene. Dette muliggjør enklere testing, og

uavhengighet til containeren eller mellomvaren. Man kan i tillegg tilpasse samme applikasjon til ulike behov uten å foreta endringer i koden (ala konfigurering av "halvfabrikata").

5.1.2 Aspekt Orientert Programmering (AOP)

Aspekt Orientert Programmering (AOP), er en teknologi som er egnet til å understøtte OO der hvor denne kommer til kort. AOP kan brukes til å programmere aspekter eller "cross-cutting concerns", som logging, sikkerhet, transaksjoner, og andre behov som må dekkes i for eksempel en enterprise applikasjon.

Et eksempel hvor behovet for AOP er tilstede, er når aspekter eller anliggender som sikkerhet eller logging, alltid skal inn som kode i klasser eller metoder. Normalt ville vi måttet duplisert kode rundt omkring, med følger for senere vedlikeholdbarhet. Innen enterprise applikasjoner er AOP sentralt hvis vi skal unngå å bruke en fullblods J2EE applikasjons-tjener som inkluderer alle enterprise funksjoner, men foretrekker å forenkle systemene ved kun å bruke web-containeren (servlet-containeren).

AOP gir mulighet for å modularisere anliggender som går på tvers innen systemet, slik som OO modulariserer felles anliggender[56]. Disse anliggendene er ofte de samme som en EJB-containeren innen J2EE skal løse.

Litt forenklet vil jeg hevde at der *avhengighetsinnsprøytning* etter beviste designvalg gjør det lett å utvide funksjonalitet i en komponent, programmere mot grensesnitt, og sikre løse koblinger—gir AOP oss ny funksjonalitet i alle komponenter uten at vi er klar over det (eller i alle fall synes det ikke i annen kode enn der aspektene designes og implementeres).

På grunn av at et aspekt ikke er synlig i koden, men legges inn gjennom såkalte "interseptorer", er det viktig å ikke bruke AOP for forretningslogikk som vi lett skal skjønne ved å lese koden[68]. Dette står i kontrast til [56] hvor foretningsregler anbefales implementert ved hjelp av AOP. Jeg mener at en forutsetning er at disse kan modulariseres ved hjelp av AOP, men uten å gå på bekostning av lesbarhet til koden.

5.2 Persistering

I et objektorientert paradigme, er det en selvfølgelig tanke at enterprise informasjonssystemet's problemområde-objekter persisteres i form av metodenes oppførsel og attributtenes tilstand. Med en objektorientert database som en forlengelse av applikasjonens objekter, kan domene-modellen enkelt og sømløst persisteres.

Persistering av data er en av de fundamentale konsepter innen applikasjonsutvikling [11]. Fra fillagring til ulike database håndterings-systemer, med 70-tallets nettverksdatabaser til 80-tallets relasjonsdatabaser, og videre til 90-tallets objektdatabaser. Mange mente på 90-tallet at relasjonsdatabasene i stor grad ville bli erstattet av objektorienterte databaser når utbredelsen og familiæriseringen av objektorientert programmering ble hovedretningen. Dette slo ikke til, og vi observerer fortsatt at relasjonsdatabaser dominerer med unntak av i noen få nisje-miljøer hvor objektorienterte databaser brukes. For øvrig stanset Standardiseringsorganisasjonen Object Data Management Group (ODMG) sitt

arbeide med spesifikasjoner av objektdatabaser i 2001[63], og vi har i dag ingen dominerende aktører som leverer standardiserte objektdatabaser med stor utbredelse.

I oppgavens perspektiv på re-design er det ikke avgjørende hvilket databasehåndterings-system vi bruker, men i avgrensningen til oppgaven har jeg sagt EIS i en to-lags programvarearkitektur som persisterer sine data i en relasjonsdatabase. Utgangspunktet er som nevnt, at systemet som skal redesignes allerede har sine data i en relasjonsdatabase, og følger SQL[1]-standarden ANSI-92[2]. Og dermed er typen gitt.

Persistering av data, eller datalagring, er ikke mellomvare. Databaser kjører i praksis på en egen dedikert database-tjener, som typisk videreføres i oppgavens re-design. Derimot vi håndteringen av objekter som persisteres ligge i mellomvaren. Den videre forlengelsen av område persistering vil av den grunn omtales videre i mellomvare-seksjonen, nærmere bestemt i seksjon 5.3.2. Mellomvare-seksjonen representerer den nye teknologien, og i praksis en tjenermaskin, som vi innfører.

5.3 Mellomvare

Mellomvareteknologi kan brukes til å integrere applikasjoner i en distribuert omgivelse. Videre tilbyr mellomvaren tjenester som moderne applikasjoner trenger. Et annet viktig formål med mellomvare er å sette applikasjoner designet for en n-lags arkitektur i produksjon på en sentral tjener-maskin, og tilby tynne web-klienter for brukertilgang. Enkel distribuering av tjenester vil komme som en følge av sentralisering, fornuftig arkitektur, og teknologi-støtte[42].

Mellomvaren skal gjøre det enklere for utviklerne å konsentrere seg om problemområdet og utviklingen av produktet, uten å hemmes for mye av teknisk infrastruktur.

Et eksempel på en viktig tjeneste som mellomvaren må støtte er transaksjonshåndtering. En transaksjon er en sekvens av operasjoner som utføres på dataene systemet behandler, og transformerer systemet fra en tilstand til en annen[42, 68, 50]. En transaksjon har fire såkalte "ACID"-egenskaper[42]:

Atomær hvilket betyr at enten utføres alle aksjonene i transaksjonen, eller så utføres ingen.

Konsistens "Consistency" er når transaksjonen tar systemet fra en korrekt tilstand til en annen.

Isolasjon av en transaksjon, er kravet om at transaksjonens resultater ikke avsløres til andre parallelle transaksjoner før den er avsluttet.

Bestandig "Durability" har vi når en transaksjon er vellykket, hvilket gir permanente forventede resultater som ikke kan være forandret på som en følge av en eventuell systemfeil.

Før innføringen av mellomvare, var det typisk bare databasehåndteringssystemet som håndterte transaksjoner etter inndata fra klientene. I dag er

[1] SQL—Structured Query Language
[2] ANSI-92—SQL-standard for Relasjonell databasespråk

mellomvaren viktigste håndterer, blant annet fordi mellomvaren kan håndtere flere databaser i samme transaksjon[42].

Vi har ulike tekniske valg av plattform for mellomvare. For enterprise informasjonssystemer med web-baserte brukergrensesnitt, klargjort for nye anvendelsesmuligheter som webtjenester og portlets, har vi i dag to dominerende retninger. Det er den proprietære *.NET* fra Microsoft, og den nylig helt åpne (GPL[3]), *Java/J2EE*. Disse to plattformene tilsammen dominerer dette segmentet. Dette gjenspeiler seg også i hvordan konsulent-bransjen organiserer seg; de fleste har fokus rundt begge disse to teknologiene. CORBA som plattform ble for for noen år siden spådd stor dominans, og vektlagt av forskermiljøer og enkelte kommersielle aktører. I ettertid har vi sett at CORBA aldri har klart å "ta av". En grunn kan være at distribuering av objekter er komplisert, dyrt, og ikke minst tregt, uten at det gir regnings-svarende effekter. Kanskje litt av samme grunn ser vi en kursendring innenfor Java/J2EE-verdenen, hvor en av teknologiene, Enterprise Java Beans, har fått flere konkurrenter. Spesielt i denne sammenhengen er "lettvekter container" rammeverk og "objekt relasjons mapping" rammeverk.

Jeg velger å fokusere den vidrer teknologiske implementeringen på Java/J2EE framfor .NET. Ikke fordi den ene nødvendigvis er bedre enn den andre—mange har ulike meninger her, men fordi vi har åpne spesifikasjoner som ofte bygger på åpne eller "de-facto" standarder, som forøvrig for en stor del har sitt utspring fra denne siden. I tillegg er Java/J2EE åpen kildekode. Java/J2EE gir derved bedre mulighet til å unngå leverandør-binding, faste lisens-kostnader, og verktøy-avhengighet—samtidig som det er en dominerende plattform. Man kan faktisk velge, gitt at man låser seg til Java som åpent språk og kjøremiljø, åpne kildekode produkter for alt man trenger innen utvikling eller framskaffelse, og drift av enterprise informasjons-systemer. Man har selvfølgelig valgfrihet til å betale masse penger for kommersielle produkter, akkurat som innen .NET, om man måtte ønske det.

5.3.1 Applikasjons-tjenere

De logiske (skjematisk) oppdelte n-lags-arkitekturene skiller ut domene- og forretningslogikk fra andre systemtjenester og brukergrensesnitt. N-lags arkitekturen forenkler utvikling og deployering, samt håndteringen av enterprise systemer, og ny mellomvareteknologi har økt interessen for n-lags arkitektur[42]. Tilsvarende har vi i referansearkitekturen.

Jo mer funksjonalitet vi kan legge på applikasjons-tjeneren, jo enklere blir det å distribuere, deployere, og administrere applikasjonen. Dette *kan* som nevnt gå på bekostning av systemets brukeropplevelse ved at vi da utvikler web-baserte tynne klienter, men som omtalt i seksjon 4.2.6.3 og hvilket jeg kommer tilbake til i seksjon 5.4.3.4, krever det "bare" ekstra ressurser.

5.3.1.1 J2EE

> "J2EE ble utviklet for å løse problemene rundt utvikling, deployering og håntering av skalerbare fler-lags systemløsninger"[42].

[3]GPL—GNU General Public License

Spesifikasjonen kommer som et industrisamarbeide mellom store aktører som IBM og Oracle, ledet av Sun Microsystems. Distribuerte løsninger var utgangspunktet for teknologien[53].

Hovedfordelene med J2EE (Java2 Enterprise Edition), er at det er en åpen spesifikasjon, og helt nylig også ble frigjort som åpen kildekode, og at systemene kan kjøres på de fleste maskiner og de fleste operativsystemer gjennom bruk av en Java Virtuell Maskin (JVM). Videre fremmer spesifikasjonen og rammeverk innen denne plattformtypen leverandøruavhengighet. En ulempe kan være at vi knyttes opp mot et bestemt, men åpent språk; Java. Til tross for avhengighet til ett språk, hvis behovet oppstår, kan tjenesteorientering gjennom webtjenester tilbys også til andre systemer skrevet i andre språk og på andre plattformer.

J2EE består av av mange ulike Javateknologier, hvorav mange er viktige innenfor konteksten til denne oppgaven (og ofte inkludert og forenklet i rammeverk jeg senere kommer tilbake til). Jeg har omtalt disse i appendiks B.1. En dyptgående beskrivelse av "Enterprise JavaBeans" (EJB), som av mange regnes som en kjerneteknologi innen J2EE, vil jeg ikke gjøre. Mitt fokus er å unngå bruken av denne teknologien. En kort omtale av EJB er også lagt til appendiks B.1.

5.3.1.2 Lettvekts-container rammeverk

Framveksten av denne typen containere har sitt utspring i at mange finner den opprinnelige J2EE-plattformen unødvendig omfattende, komplisert, og tildels uhensiktsmessig, og hindrer god bruk av objektorientering[53]. Tilsvarende type beveggrunn som vi så for forenkling av metodikker.

Teknologier disse ofte benytter er "avhengighetsinnsprøytning" og "AOP" slik vi så i seksjonene 5.1.1 og 5.1.2. Disse er sentrale for også å kunne sette sammen en plattform som også nyttiggjør og gjenbruker komponenter fra ulike prosjekter (og deres rammeverk) på et enkelt sett.

Vi har flere ulike lettvekts-containere på markedet, men så vidt jeg har brakt til erfaring, kommer alle fra åpen kildekode prosjekter.

I forhold til referansearkitekturen vil spesielt tjenestelaget og domene-modellen dekkes av en lettvekts container. Men vi kan videre se containeren som integratoren som syr sammen andre rammeverk og derved i sum representerer alle lagene.

5.3.2 Objekt relasjons tilordning (ORM) rammeverk

En standard Java-applikasjon som kommuniserer mot en SQL database sender SQL-setninger mot databasen via JDBC. Vi ønsker en mekanisme som frigjør oss fra slik lav-nivå aktivitet, og gjør det enkelt å forholde seg til objekter og som tilbyr lagring og framhenting av klassenes instanser[11].

Når vi benytter et objektorientert språk og persisterer dataene i en relasjonsdatabase, kan vi støte på problemer som stammer fra forskjeller i de to paradigmenes former for representasjon av virkeligheten (problemområdet); objekter og tabeller (relasjoner). Dette gir et gap mellom paradigmene som må tettes[32, 11]. Et eksempel kan være ulik granularitet. Om vi har et objekt a med noen atomære attributter, som i tillegg inneholder et annet objekt b med egne attributter, hvordan skal vi representere a-objektet i en tabell? Et annet problem er at objekter kan inngå i arv, mens relasjonsdatabaser ikke støtter arv

(med unntak av noen leverandør-spesifikke hybrider). Slike gap eller forskjeller i paradigmene, må tettes[11].

I komitée-arbeidet med spesifisering av J2EE teknologien, nærmere bestemt entitets-bønnene, er problemet med objekt-persistering blitt løst. Allikevel søker stadig flere å unngå denne teknologien[53, 40], på grunn av dens kompliserte spesifikasjon, lave ytelse, og ikke minst; dreining vekk fra objektorienteringen. Mønstre er blitt introdusert for dette området, men de bidrar ikke til annet enn å avhjelpe svakhetene med teknologien som brukes i følge [40, 76], hvilket er en indikator på at andre veier bør utforskes.

Alternative løsninger på objekt relasjons tilordningen (ORM[4]), i form av rammeverk som inkluderer ORM-mønstre, er kommet. Disse oppnår stadig bredere aksept. ORM mediaterer applikasjonens interaksjon med en relasjonsdatabase, og fristiller utvikleren til å konsentrere seg om forretnings problem-området[11] og domene-modellen. Slike rammeverk som ikke "invaderer oss" i form av avhengigheter (sees i koden når vi må "subklasse" containeren eller rammeverket), er med på å forenkle enhets-testing og fremmer enkelt vedlikehold.

I enkelhetens ånd, kan vi starte med å knytte objekter og tabeller en-til-en. Dette gjøres i ORM-rammeverkets tilhørende konfigurasjonsfiler, vanligvis i et XML-format. For re-design med datamodellen som en del av arven vi skal videreføre, er dette enda mer typisk. Underveis i utviklings-løpet kan vi refaktorere kode og objekt-modell om dette synes hensiktsmessig med hensyn på ytelse eller enkelhet i designet av objekt-modellen.

ORM plasserer seg i referansearkitekturens nederste lag.

5.3.3 MVC rammeverk

MVC rammeverk er ikke en typisk mellomvareteknologi, men jeg har valgt å sortere slike rammeverk her, og ytterligere snevre inn nedslagsfeltet til konsentrasjon rundt *Web* MVC rammeverk. Web-fokuset for enterprise informasjonssystemer som plasseres på en sentral tjener under kontroll av mellomvaren, gjør denne sorteringen naturlig. Web MVC rammeverk plasseres inn i arkitekturens presentasjonslag, og dekker tidligere omtalte MVC-mønstre.

Vi har tidligere sett på mønstre, hvorav ett av disse var MVC (se seksjon 4.3.1.4), som sammen med objektorienteringen har sitt opphav fra Norge.

Web MVC-rammeverk kan deles inn i to grupper, som angir ulik filosofi for virkemåte og tilnærming:

Anmodningsdrevne eller drevet av "Requests/Responds". Dette var kategorien som først ble vanlig, og har fortsatt størst utbredelse. Slike rammeverk har en tynn MVC innpakning over servlet'ens "request-response" arbeidsflyt[53]. De sender et skjema (hele eller deler av en web-side) til en URL[5], som er knyttet mot en "Controller" som håndterer henvendelsen videre. Det er lett å bytte ulike visningsteknologier, da disse er relativt uavhengige av rammeverket.

Hendelsesdrevne web-rammeverk, inngår også i kategorien komponent-baserte hendelsesdrevne rammeverk. Disse har skjema-komponenter som

[4] ORM—Objekt Relasjons Mapping
[5] URL—Universial Resource Locator

er knyttet mot ulike lytte-komponenter som starter utførelsen når gitte hendelser oppstår[53]. Her er en web-side å sammenligne med en komponent som kan holde tilstander og vet selv hvordan de skal presenteres.

Som for lettvekts-containere er også mye av dette markedet dominert av prosjekter fra åpen kildekode-verdenen. Dette gjelder særlig for de anmodningsdrevne rammeverkene.

For den hendelsesdrevne typen, har vi en variant i form av en spesifikasjon (JSR-127[6]) med navn "Java Server Faces" (JSF), hvor også kommersielle aktører er på banen med verktøy-støtte i tillegg til delaktighet med spesifikasjonsarbeidet[53]. Mulighet for hurtig brukergrensesnitt utvikling er et mål for JSF, men det gjenstår å se hvordan prinsipper om enkel kode som er lett å refaktorere og teste, uavhengigheter til verktøy og leverandører, og kun aktiv promotering av én visningsteknologi, JSP, vil vise seg som et godt valg.

Jeg velger å fokusere på den hendelsesdrevne typen rammeverk, fordi de etter min oppfatning gir en bedre og bredere objektorientert tilnærming, og gir bedre abstraksjon bort fra servlet-teknologien.

5.3.3.1 Brukergrensesnitt

Ved re-design av enterprise informasjonssystemer, er mitt utgangspunkt å implementere brukergrensesnittet som en tynn web-klient som kjører i en standardisert webleser—uten behov for proprietære avhengigheter på klientmaskinen. "Viewet" til den tynne klienten innenfor de hendelsesdrevne MVC-rammeverkene kan finnes i flere varianter. Noen benytter HTML[7] som visningsteknologi, mens andre kan benytte JSP. Fordelen med HTML, slik jeg ser det, er at profesjonsdeling mellom utviklere og grafisk designere blir enklere og mere kodenær.

Ajax[39] er en mulig ikke-proprietær teknologi som kan benyttes til å kjøre applikasjoner med en bedre brukeropplevelse enn hva som tradisjonelt har vært mulig ved en HTTP "request" mot en tjener og en HTTP "respons" tilbake som tegner opp hele web-siden på nytt. Ved hjelp av Ajax kan deler ar en web-side i en web-applikasjon tegnes opp, hvilket gir bedre dynamikk, mindre nettverksbelastning og dermed mindre ventetid. En slik tilnærming og teknologi-støtte fordrer at vi må tenke utenfor de normale baner ved design av brukergrensesnitt for web-applikasjoner, hvor vi er nærmere de tradisjonelle rike klientene i funksjonalitet.

Selv med bruk av gode rammeverk til å produsere selve systemet, går forholdsmessig mye tid til GUI-delen når enterprise informasjons-systemer utvikles eller re-designes. Samtlige prosjekter som er utført i referansegruppen i Forsvaret (se appendiks A) de siste tre årene ved bruk av tilsvarende arkitektur og rammeverk, er estimert til i overkant av 50% ressursbruk for GUI-delen. Samme erfaring fikk vi bekreftet fra et av de større konsulent-husene i Norge. En mer akseptabel fordeling på ressursbruk for presentasjonslaget burde etter min mening ligget rundt 25–30%. Forhåpentlig vil vi etter hvert oppleve bedre GUI-baserte "drag-and-drop" verktøy, som integrerer godt med resten av kodebasen vår uten å hindre enkel refaktorering og testing. Dette fordrer enda bedre

[6] `http://jcp.org/en/jsr/detail?id=127`
[7] HTML—HyperText Markup Language

rammeverk med verktøystøtte som ikke forkludrer koden. Microsoft's .NET har trolig bedre verktøystøtte på dette området idag, men gir de omtalte avhengighets-faktorene.

En sak å spesielt nevne om 50% av ressursene går med til presentasjonslaget. Hvis vi gjør et tanke-eksperiment med en tradisjonell metodikk hvor en 2-lags klient-tjener EIS skal utvikles, og angir den totale ressursbruken for prosjektet. La oss anta at kun 10% av ressursene går med til grafisk brukergrensesnitt, og ser bort fra ressurser til logikk i samme lag. Dette prosjektet gir jeg en total ressursbruk på 32 personer i 5 måneder, altså 160 person-måneder. Det er denne ressursbruken som 10% til grafisk brukergrensesnitt er en del av. Motsvarende, anta at prosjektet gjennomføres med minimal agile metodikk, og ved bruk av de forventede forsterkninger som vi indirekte oppnår fra arkitektur, og direkte fra teknologien. Og 4 personer bruker 6 måneder, altså 24 person-måneder. Da er 50% ressursbruk for presentasjonslaget svært lite i person-måneder!

Ressursbruken for å oppnå en god brukeropplevelse kan allikevel være et ankepunkt til forfektelser i oppgaven, men sett i et enterprise perspektiv med nye anvendelsesområder og enklere deployering og totalt vedlikehold, er det likevel en akseptabel pris å betale slik jeg ser det.

5.3.3.2 Validering

Ved omtale og argumentasjon rundt tjenestelaget, skulle validering og håndtering av forretningsreglene ligge der. I presentasjonslaget vil vi i tillegg ha behov for validering slik at brukeren kan få en fornuftig respons raskest mulig.

Slik validering bør ikke knyttes til web-laget selv om vi har en web-applikasjon[51]. Valideringsmekanismen bør være så generell at validering på façade-laget og domene-modellen er like naturlig. Da kan valideringsmekanismen gjenbrukes i eksempelvis en portlet, eller om vi skifter web MVC-rammeverk, på et senere tidspunkt.

5.3.4 Webtjenester og fjerntilgang

Verdensveven blir stadig mer utbredt for applikasjon til applikasjon kommunikasjon (P2P[8], B2B[9]). Det programmatiske grensesnittet som er tilgjengeliggjort refereres til som Web Services[81] (WS) eller webtjenester. Dette er for øvrig i tråd med Reenskaug's tanker i forbindelse med rolle modellering og vanskelighetene med å få gjennomslag for dette, men hvor han oppmuntres av arbeidet som gjøres innenfor webtjenester, hvor mennesker og komponenter er de sentrale aktørene, mens klasser overlates til spesialistene[70].

Teknologien er enkel; den skal flytte XML-dokumenter mellom tjenesteprosesser ved bruk av standard Internet protokoller. En stor forskjell i forhold til CORBA og EJB, er at disse tjenestene er tilstandsløse. Tilstandsløsheten forenkler interoperabilitet på tvers av ulike plattformer, operativsystemer et cetera.

XML's allsidighet, gjør denne teknologien attraktiv for å utveksle informasjon mellom programmer (interoperere) som kjører på forskjellige operativsystemer og er skrevet i ulike språk[81].

Webtjenester i sin enkleste form består av tre (fire) komponenter[80]:

[8] P2P—pear-to-pear
[9] B2B—business-to-business

Programvare Tjenesten er programvare som kan prosessere et XML-dokument som mottas gjennom en kombinasjon av transport og applikasjons protokoller (Simple Object Access Protocol (SOAP) er vanlig). Det eneste kravet er at tjenesten er i stand til å prosessere gitte veldefinerte XML dokumenter.

XML XML dokumentet er Webtjenestens hoveddel fordi det inneholder all applikasjons-spesifikk informasjon som tjenestens konsument sender til tjenesten for prosessering. Dokumentene som en webtjeneste kan behandle beskrives i XML skjemaet ved hjelp av Web Services Description Language (WSDL).

Adresse Adressen er en protokoll binding kombinert med en nettverksadresse som tjeneste-brukeren (spørreren) kan anvende til å aksessere tjenesten. Adressen/referansen identifiserer hvor tjenesten kan finnes med bruk av en bestemt protokoll (for eksempel TCP eller HTTP).

Konvolutt De tre nevnte komponentene kan kompletteres med en fjerde komponent, konvolutten, som pakker inn dokumentet og hvor ekstrainformasjon som ruting og sikkerhet kan legges til uten å endre det opprinnelige XML-dokumentet.

Ulike rammeverk kan forenkle implementering av webtjenester, og spesielt når vi har et godt definert façade-lag. Konkrete forslag omtales om litt.

5.3.4.1 SOAP

For webtjenester er "Simple Object Access Protocol" (SOAP) den mest kjente og brukte protokollen. Selv om *Objekt* inngår i protokoll-navnet, har den ikke noe å gjøre med objekt-aksess. SOAP er en lettvekts protokoll for utveksling av informasjon i et desentralisert og distribuert miljø[69]. Protokollen er XML-basert med tre deler: en konvolutt som beskriver innhold og semantikk, samt avsender og mottager, ett sett av regler som beskriver applikasjonsdefinerte datatyper, og en konvensjon for hvordan fjern-prosedyre kall og responser skal representeres.

W3C[10] er i ferd med å gjøre SOAP til en fullstendig åpen standard, og rydder opp i uklarheter som har vært i spesifikasjonen.

5.4 Åpen kildekode

Fri og åpen kildekode programvare (FOSS[11]) er programvare som gir rettigheter til å kjøre, kopiere, distribuere, studere, endre, og forbedre programvare ved behov[4]. Generelt vil jeg omtale FOSS og åpen kildekode synonymt, selv om det finnes programvare av kommersiell art som kjøpes og lisensieres, men som leveres i form av kildekode[12]. Generelt kan vi si at ingen programvare som kun lar seg kjøre på ett operativsystem, som for eksempel kun Microsoft's, kan betegnes som fri programvare.

[10] W3C—The World Wide Web Consortium. Finnes på VVV som `http://www.w3c.org`.
[11] FOSS—Free and OpenSource Software
[12] "JIRA" fra `www.atlassian.com` er et eksempel

På oppdrag fra det Amerikanske forsvarsdepartementet (DoD), utarbeidet MITRE[13] en rapport om departementets bruk av fri åpen kildekode programvare i 2002/2003[4]. Hovedkonklusjonen fra analysen viste at denne type programvare spilte en mer kritisk rolle enn antatt. Fri åpen kildekode programvare ble ansett som viktigst innenfor områdene *infrastruktur støtte*, *programvare utvikling*, *sikkerhet* og *forskning*. En av anbefalingene fra rapporten var å oppmuntre til bruk av fri åpen kildekode programvare på grunn av lavere kostnad, høyere sikkerhet, og i mange tilfeller høy grad av funksjonalitet og gode brukererfaringer. Dette gjør FOSS til gode kandidater både ved ervervelser og innen arkitektur for DoD systemer.

Fra det Kanadiske forsvarets forskningsavdeling kommer en rapport som konkluderer med at FOSS tilbyr konkrete muligheter for kostnadsreduksjoner, økt fleksibilitet, og teknologi innføring[71]. Som en konkret fordel nevnes støtten til åpne standarder med den følge at det vil bidra direkte til interoperabilitet og høyere sikkerhet gjennom å kunne se på kildekoden.

Stadish Group utførte en undersøkelse i 2004 på kvaliteten til noen åpen kildekode produkter, og stiller spørsmål til hvordan man kan bruke penger på kommersielle produkter med dårligere kvalitet. En sammenligning mellom "Apache" og Microsoft's "IIS" web-tjenere viste at bare 20% mente at Microsoft's produkt var mere pålitelig, mens over 39% gikk i "Apache"'s favør. Kun 9% mente at "IIS" hadde færre feil. For J2EE applikasjons-tjenere rangerte 52% kvaliteten til "JBoss" likt med IBM's "WebSphere", og 7% mente kvaliteten til "JBoss" var bedre. Den dyreste databasen på markedet, "Oracle", overbeviste heller ikke sammenlignet med "MySQL" hva angikk sikkerhet. Disse ble rangert likt. Operativsystemet "Red Hat Linux" ble av 73% vurdert å være mindre utsatt for ulike fiendtlige angrep enn "Microsoft Server"[75]. For øvrig brukes Apache web-tjener av opp i mot 50 millioner web-steder på verdensbasis (gjaldt året 2005) i følge Netcraft[14]. Dette ga en markedsandel på nesten 70%. Til sammenligning hadde Microsoft's Internet Information Services (IIS), på andreplass, rundt 20% markedsandel.

Med bakgrunn i tsunami-katastrofen 26. desember 2004, tok Jeffrey A. Kaplan initiativet til grunnleggelsen av "The Roadmap for Open ICT Ecosystems". Initiativet er støttet av myndighetspersoner fra 13 land og ulike kunnskaps- og teknologi-eksperter. Behovet for felles åpne standarder hadde aldri vært mer iøynefallende. Som rapporten beskriver er åpne standarder og åpen kildekode ikke det samme. Derimot kan implementasjoner av en åpen standard i form av åpen kildekode akselerere utbredelsen av standarder. Likeledes har disse to mange likhetstrekk i form av fellesskap-orientert, åpen samarbeidende prosess som ender ut i et fritt tilgjengelig produkt[44].

Innen åpen kildekode-verdenen dukker det stadig opp nye produkter, og ganske ofte forsvinner mange. Men noen får fotfeste. Her deltar et stort fellesskap med forslag, utvikling, og kodegranskning. Noen ender opp som defacto standarder, og med en utbredelse og bruk så stor at mange tror dette er kostbare kommersielle produkter. Videre kommer mange alternativer fra åpen kildekode som et mottrekk til tunge og komplekse produkter. Andre ganger kommer produkter som utforsker nye idéer, som senere blir en hovedretning. Resultatene er ofte svært gode fordi disse løsningene bare blir adoptert av mange

[13] MITRE—The MITRE Corporation

[14] Se `http://www.fcw.com/article90919-09-26-05-Print`

når de er pålitelige og leverer verdi[53]. Mange produkter oppstår som et resultat av at noen ser reelle behov for en generell løsning på generelle problemer, slik vi tidligere så for framveksten av mønstre.

Flere og flere, inklusive offentlige virksomheter har skjønt betydningen av åpen kildekode[4, 71, 30]. Etter siste regjeringsskifte, har også den Norske politiske kursen for programvare endret seg. Vi fikk nylig IT-meldingen som gir anbefaling av bruk av åpen kildekode i offentlig forvaltning[30].

Fordeler som muligheten til å studere kildekoden slik at vi virkelig kan vite hva programmene gjør er spesielt viktig når sikkerhet skal vurderes. Mange utbredte produkter finnes bare i form av åpen kildekode. Lavere kostnader og leverandøruavhengighet er en direkte konsekvens ved valg av åpen kildekode. Support er mulig å kjøpe for enkelte produkter, om man trenger noe utover det man kan få fra åpne brukergrupper, forumer, et cetera.

Ulemper Mye bra kan sies om gode åpen kildekode produkter. Men selvsagt eksisterer ulempene også. En slik ulempe er at vi ikke i utgangspunktet har en garantert support. Supporten er dugnadsdrevet og gratis (gratis er ikke en ulempe), men denne kan i økende grad kjøpes og garanteres for de som ønsker det.

En annen ulempe med åpen kildekode programvare, er hvis vi er avhengig av sikkerhetsgodkjenning eller sertifisering av programvare for at denne skal kunne kjøres på høygraderte nettverk. Dette gjelder virksomheter som omfattes av sikkerhetsloven. Nasjonal Sikkerhets-Myndighet (NSM) som etter sikkerhetsloven utøver funksjonen som nasjonal sikkerhetsmyndighet, og derverd godkjenner systemer som benyttes på slike nettverk. Godkjenning avhenger enten av at produktet er sikkerhetssertifisert og eksempelvis finnes på "Common Criteria"-listen[15], og innehar høyt nok evaluert sikkerhetsnivå (EAL). Eller at NSM er i stand til og innehar ressurser til selv å gjøre vurderinger omkring sikkerheten. Sertifisering er basert på standardiserte krav og metodikk som er internasjonalt anerkjent i henhold til "Common Criteria" og "Common Evaluation Methodology"[62]. Sikkerhetssertifisering er kostbart, og ligger typisk til store kommersielle leverandører å finne midler til, hvilket lett kan ekskludere åpen kildekode produkter, om ingen andre instanser finansierer slik sertifisering.

Andre ulemper kan gis, men ved at jeg tar utgangspunkt i de produktene som har oppnådd stor utbredelse, mener jeg mange ellers opplagte ulemper utgår.

Det er nødvendig å belyse at andre faktorer spiller inn; for eksempel om det gamle systemet persisterer sine data i en kommersiell relasjonsdatabase, og virksomheten har investert mye i opplæring og support for denne, eller om en mellomvare-løsning fra en kommersiell leverandør er installert på flere tjenere og kompetansen her er god. Vi har valgfrihet til å velge litt eller alt fra åpen kildekode. Mitt valg på en lettvekter container, er av typen åpen kildekode. Jeg har ikke funnet noen kommersiell aktør i dette segmentet, hvilket kan henge sammen med at fokus for disse aktørene er å selge dyre løsninger. Enkle og lette løsninger innehar trolig ikke samme inntjeningspotensial, men inkluderes i flere tilfeller allikevel inn som integrerte deler av den kommersielle løsningen.

[15] Common Criteria—`http://www.commoncriteriaportal.org`

5.4.1 Organisasjoner

> *"Imagine a world in which every single person is given free access to the sum of all human knowledge. That's what we're doing."*

Wikipedia's[83] opphavsmann Jimmy Wales skrev dette "manifestet", som like gjerne kunne vært skrevet om åpen og fri kildekode programvare.

Vi har flere organisasjoner som promoterer åpen programvare. To av disse er "Open Source Initiative"[16] (OSI) og "The Free Software Foundation"[17] (FSF). Begge jobber for å sikre brukere retten til å studere, kopiere, modifisere, og videredistribuere programvare. FSF er strengere i sin definisjon av frihet - fri programvare skal bare brukes i annen fri programvare, mens andre lisenser for åpen kildekode, tillater programvaren brukt i lukkede systemer.

En tredje organisasjon er "Free Software Foundation Europe"[18] som ble etablert i 2001 med vekt på de europeiske aspektene ved fri programvare. Spesielt kjent for kampen mot EU's "Software patent directive":

> *After years of struggle, the European Parliament finally rejected the software patent directive with 648 of 680 votes: A strong signal against patents on software logic, a sign of lost faith in the European Union and a clear request for the European Patent Office (EPO) to change its policy: the EPO must stop issuing software patents today.*

5.4.2 Lisens-typer

Åpen kildekode og fri programvare kan ha ulik innhold og politisk betydning. For å bruke åpen kildekode på lovlig vis er det viktig å være klar over forskjellige regimer for lisensiering. Spesielt hvis man skal utvikle produkter av kommersiell art, eller man skal bruke åpen kildekode eller fri programvare i nye systemer som ikke forblir åpne.

Vi finner hos OSI og FSF mange ulike lisenstyper. Her vil jeg nevne noen få som er mye brukt, og støttes av begge organisasjonene. Vi finner akronymet GNU[19] foran flere lisenstyper, som stammer fra Richard Stallmann's GNU-prosjekt og forkjempelse for fri programvare.

GNU GPL "GNU General Public License", er en fri programvarelisens som krever at kildekode alltid skal distribueres og re-distribueres. Programvare under denne lisensen kan bare brukes i fri programvare. Anbefales av FSF som standard lisensiering.

Se: `http://www.fsf.org/licensing/licenses/gpl.html`

GNU LGPL "GNU Lesser General Public License", er også en fri programvarelisens, men i motsetning til GPL tillater denne inkludering av programvare under denne lisensen i annen programvare som ikke er fri.

Se: `http://www.fsf.org/licensing/licenses/lgpl.html`

BSD Finnes i to varianter, den opprinnelige BSD lisensen, og en modifisert utgave hvor man er fritatt fra å oppgi

[16] `http://www.opensource.org`
[17] `http://www.fsf.org`
[18] `http://www.fsfeurope.org`
[19] http://www.gnu.org

> "This product includes software developed by the University of California, Berkeley and its contributors."

Se: `http://www.xfree86.org/3.3.6/COPYRIGHT2.html\#6`
og `http://www.xfree86.org/3.3.6/COPYRIGHT2.html\#5`

Apache License, Version 2.0 Er en fri lisens, men som ikke er kompatibel med GNU GPL, med den forskjell at Apache-lisensen har en patenteringsklausul. Lisensen er mye brukt, og brukes blant annet for Apache's webtjener, og andre produkter som har tilhold på utsiden av "Apache"-organisasjonen. Rammeverket og lettvekts-containeren "Spring" er underlagt nevnte lisens.

Se: `http://www.apache.org/licenses/LICENSE-2.0`

5.4.3 Teknologi-implementeringer

Jeg har fram til nå beskrevet teknologier relativt generelt. Jeg vil fokusere på tre typer rammeverk som tilsammen dekker den nødvendige implementering av den referansearkitekturen jeg utledet i seksjon 4.4 til bruk innen re-design ev EIS.

En forutsetning for rammeverkene er språk, og jeg har valgt Java ut fra tidligere argumentasjon. Språket gir nødvendig implementering av objektteknologien som forutsettes for en ut-flatet kostkurve, men valget av språk er generelt ikke avgjørende. Objektorientert språk er derimot essensielt.

Det første rammeverket er et applikasjons-rammeverk som implementerer en lettvekter container, og gir foruten forenklet J2EE utvikling, "avhengighetsinnsprøytning", og muligheter for aspekt-orientert programmering.

Rammeverk nummer to er en utvidelse av førstnevnte, og bidrar til å bygge bro mellom objekt-modellen og relasjonsdatabase-modellen, slik at det objektorienterte paradigme benyttes, samtidig som database-arven kan videreføres.

Til bruk for å implementere presentasjons-laget med MVC-mønster, kommer tredje rammeverk inn. Også dette må sees som en utvidelse av det første rammeverket.

Summen av disse tre rammeverkene dekker den referansearkitekturen jeg satte opp i forrige kapittel.

Ut fra den inndeling jeg tidligere i dette kapitlet har gjort for teknologi; objektorientering, persistering og mellomvare, bryter jeg denne rekkefølgen når jeg nå skal gi forslag til konkrete implementeringer. Det primære, foruten språket, er å implementere arkitekturen , deretter angir jeg utviklingsmiljø med testrammeverk, og muligheter for implementering av webtjenester til slutt. Konkrete forslag til database-håndterings systemer, er utelatt grunnet forutsatt gjenbruk.

5.4.3.1 Objektorientert språk

Objektorientert kode trenger et språk å skrives i. Og det er denne koden som kan kjøres, og gi liv til et re-designet EIS. Flere mulige språk med stor utbredelse er *C++*, og det litt nyere *Java* fra SUN—opprinnelig ble laget for å inngå i

forbrukerelektronikk. Et relativt nytt språk med økende utbredelse kommer fra Microsoft; *C#*.

Av årsaker begrunnet i seksjon 5.3 har jeg valgt Java som språk med endel tilhørende standarder. Java kan kjøres uten rekompilering på de fleste plattformer gjennom sin Java Virtuelle Maskin (JVM) liggende over operativsystemet. Kjørbar kode kan flyttes og gjenbrukes uten kompilering. Dette er på mange måter slik som Smalltalk kjører sine objekter i en Smalltalk Virtuell maskin (VM).

Det er mulig å lage rike grensesnitt gjennom bruk av Java's "Swing" bibliotek eller andre tredjeparts API'er, og tilgjengeliggjøre brukergrensesnittene ved hjelp av teknologier som "Applets" og "Java Web Start". Men Java's styrke ligger først og fremst innenfor den større delen av enterprise systemet som ikke programmerer brukergrensesnittet direkte. Til dette brukes eksempelvis HTML eller JSP, støttet av Java.

"Plain Old Java Objects" (POJO), er ofte omtalt som motvekt til de tyngre objektene som Enterprise JavaBeans representerer. POJO'er brukes i økende grad, og representerer ofte mindre eller ingen låsing mot omgivelsene. Dette vil i stor grad være med på å gi enklere kodedesign og testing.

5.4.3.2 Lettvekts-container

Lettvekts-container rammeverk (eller applikasjons-rammeverk) er etter min mening helt avgjørende for å kunne imøtekomme intensjonen om raskere, bedre, billigere, og enklere re-design (eller utvikling) av EIS:

Raskere Vi kommer raskere i mål med enklere system (unntaket er kanskje presentasjonslaget).

Bedre Sentralt for å implementere referansearkitekturen, og legge grunnlaget for enkelt å utøke anvendelsesområder ved behov. Andre gode rammeverk kan enkelt integreres.

Billigere Alle er åpen kildekode—billigere. Vi behøver kun en rimelig eller åpen kildekode servlet-container for å produksjonssette de fleste applikasjoner basert på slike rammeverk, som igjen fordrer mindre av maskinvaren.

Enklere Testing forenkles, hvilket motiverer for testing som en integrert del av design og koding. Mange gode mønstre er implementert, hvilket gjør det derved enkelt å ta disse i bruk. Rammeverket bidrar til enklere kode og færre kodelinjer. Færre linjer gir mindre kode som kan feile. Færre linjer med enkel kode letter vedlikehold og senere utvidelser, hvilket gir lavere livstidskostnader for systemet.

Enkelhet i forholdt til tradisjonelle J2EE alternativer, og løsere koblinger mellom objektene gjennom bruk av AOP og IoC, og kodens avhengighet til miljøet rundt seg blir minimal (i noen tilfeller ingen). Gjennom bruk av POJO'er og programmering mot grensesnitt rettes fokus på objektorientering igjen, en kontrast til hvordan J2EE med EJB har utviklet seg.

Av implementasjoner innen denne kategorien finner vi mange ulike rammeverk fra ulike prosjekter. Av disse vil jeg nevne tre alternativer som kunne vært benyttet for oppgaven:

Pico fra `http://www.picocontainer.org`. Denne finnes med ulike språkimplementeringer, inklusive Java og .NET plattformene. IoC-støtte er innebygd, og en annen container som er en utvidelse av denne, "NanoContainer", brukes for AOP.

Spring eller Springframework fra `http://www.springframework.org`.

HiveMind fra `http://hivemind.apache.org` er nok et alternativ, og har fått status i Apache-organisasjonen som et topp-nivå prosjekt.

Jeg har valgt Spring-rammeverket på grunn av størst utbredelse, og gode erfaringer fra referansegruppen i Forsvaret (se appendiks A), hvor rammeverket er fast inventar i de sammenhenger jeg refererer til. Flere kommersielle aktører har, eller er i gang med å implementere denne lettvekts-containeren eller rammeverket i sine produkter.

Hovedformålet med Spring, og andre lettvekts containere, er blant annet å gjøre J2EE enklere å bruke, og å fremme god objektorientert programmeringspraksis[36, 76, 54, 68]. Enterprise funksjonalitet slik som logging, sikkerhet, persistering av forretningsobjekter, er ikke inkludert i rammeverket, men integreres ved hjelp av IoC og AOP. Sist men ikke minst, slike rammeverk gjør testing lettere. Spring legger et abstraksjonslag over J2EE API'er som normalt er vanskelig å teste, slik at vi kan teste mot dette enkle abstraksjonslaget i stedet[53].

Modulære oppbygning og en internt konsistent arkitektur gjør at man kan velge hvilke deler en ønsker å benytte. En hovedfordel med Spring rammeverket er at det er sentrert rundt det å tilby organisering av forretnings objektene[53].

Spring er et lagdelt Java/J2EE applikasjonsrammeverk basert på kode publisert i [52] fra 2002. Utbredelsen er raskt voksende, og store selskaper og organisasjoner tar i bruk Spring innen stadig nye områder som bank, logistikk, universitets-systemer, forskning, "Grid computing", med flere[76].

Alle saker har flere sider, og jeg vil til slutt ta opp noen innvendinger forbundet med lettvekter-containere, hvorav Spring-rammeverket i dette tilfeller rammes:

DI Dependency injection ("avhengighetsinnsprøytning"). Spring har sine teknikker, både for deklarativ og programmatisk definering'er. Dette er ikke standard, og gir en form for låsing.

MVC-rammeverk Ut fra Spring's egen filosofi[51] om gjenbruk av andre rammeverk og ikke "finne opp hjulet" på nytt, skulle ikke et eksplisitt rammeverk for MVC vært inkludert, da det finnes flere andre gode på markedet som kan integreres. Min personlige mening er at presentasjonslaget hvilket gir systemer et ansikt utad, har vært en av driverne for nettopp dette. I mitt forslag til referanserammeverk har jeg holdt meg til den nevnte filosofien, og valgt et annet MVC-rammeverk.

XML Vi kan ende opp med svært mange mange linjer XML. Dette må tas i betrakning når vi teller kodelinjer. En annen side av det er at koden blir renere, kortere og bedre vedlikeholdbar - som kanskje snur dette punktet til en fordel?

Et slik uttrekk fra koden over i definisjoner, er i tråd med Fowler's syn på avhengighetsinnsprøytning[34]. XML definisjonene gir også en form for låsing, da det ikke eksisterer standarder for området.

Avhengighet (slik vi finner i leverandør-låsing) kan oppstå når vi forenkler vår kode ved å bruke abstraksjoner i form av API'er fra Spring, mens den egentlige teknologien endres. Dette er så langt ikke erfart som problem, da Spring er raskt ute med oppdaterte versjoner. Spring oppfordrer forøvrig til minst mulig avhengighet til API'ene. Dette er mulig i mange senarier for alle lagene med unntak av presentasjons-laget hvis vi benytter Spring MVC.

5.4.3.3 Objekt relasjons tilordning (ORM)

Innen kategorien objekt relasjons tilordning, foreligger både kommersielle og åpen kildekode rammeverk som kan konfigureres inn i en lettvekter container som Spring. Felles for disse er at de erstatter bruken av EJB entitets-bønner, slik at vi kan designe systemene i et objektorientert paradigme ved å forholde oss til objekter[53, 40]. Innen kategorien åpen kildekode kan flere nevnes:

JDO "Java Data Objects" er en spesifikasjon[20] og en konkret implementasjon er JPOX[21]. JPOX 1.1 er referanseimplementasjonen for JDO versjon 2.

Apache OJB fra `http://db.apache.org/ojb` støtter blant annet JDO-spesifikasjonen.

iBATIS SQL Maps fra `http://www.ibatis.com` er en enkel variant uten basis i spesifikasjoner som de ovenfor.

Hibernate fra `http://www.hibernate.org`. Dette er trolig den mest dominerende i dag, og legges til grunn for spesifiseringsarbeide av persisteringsmekanismen til EJB 3.0.

Jeg har valgt Hibernate både fordi det er det ORM rammeverket som har størst utbredelse innen Java/J2EE verdenen, og er godt integrert med Spring gjennom forenklede API'er. Referansegruppen i Forsvaret (se appendiks A) benytter samme rammeverk i mange enterprise informasjonssystemer med godt resultat. En konkret erfaring jeg ønsker å trekke fram er når vi utvikler eller re-designer, bytter vi mellom ulike databaser ved hjelp av én linje XML som endres, forutsatt at lagrede prosedyrer eller annen type låsing ikke er tilstede. I produksjons-systemet er det ofte en kommersiell database som benyttes, og som en aktiveres for den sentraliserte applikasjonen ved kun en mikroskopisk endring i en konfigurasjons-fil.

5.4.3.4 MVC-rammeverk (for web)

Web MVC-rammeverk i Java-verdenen finnes i et stort antall. Jeg har plukket ut et beskjedent utvalg, men som jeg mener er gode representanter for både utbredelse og ulike typer:

[20] `http://www.jcp.org/en/jsr/detail?id=12`
[21] `http://www.jpox.org`

Struts Struts[22], kom ut i 2001 som et av de første web MVC-rammeverk, og regnes som en *de facto* standard[68]. Sorterer inn under kategorien anmodningsdrevet.

WebWork WebWork[23] tilhører andre generasjons MVC-rammeverk, og er av kategorien anmodningsdrevet.

SpringMVC SpringMVC[24], sorterer også inn under kategorien anmodningsdrevet.

Tapestry Tapestry[25], kategoriseres som både hendelsesdrevet og komponentbasert. Til forskjell fra de anmodningsdrevne (se 5.3.3) rammeverkene er Tapestry mer i tråd med objekt-orientering med objekter, metoder og egenskaper, istedet for URL'er og spørreparametre[68].

Echo2 Echo2[26] er et rammeverk for å utvikle objektorienterte hendelsesdrevne web applikasjoner, tilsvarende Tapestry.

Det finnes kommersielle hjelpeverktøy som støtter visuell design og utvikling mot rammeverket. Kunnskap om HTML, HTTP, og JavaScript er da ikke nødvendig.

JSF "Java Server Faces"[27], er som Tapestry i kategorien komponent-basert og hendelsesdrevet. Dette er det eneste som innehar har en form for standardisering (JSR-252) gjennom Sun's Java Community Process (JCP).

Presentasjonslaget gir derved mange ulike valg for rammeverk. Om vi har tidligere preferanser, god erfaring, og kunnskap om et slikt rammeverk, kan trolig enkleste tilnærming være å inkludere det mest familiære i containeren.

Jeg har valgt Tapestry fordi det er enkelt, objektorientert, og lett integrerbart i Spring. Valget har derimot ingen avgjørende betydning for oppgaven. Valget av dette rammeverket er trolig det som kan diskuteres mest og lettest argumenteres mot, av de rammeverkene jeg har valgt. I referansegruppen i Forsvaret (se appendiks A) er både Struts, WebWork og SpringMVC benyttet.

Tapestry's fokuserer på fullstendig å skille Java kode og HTML kode[53]. En av Tapestry's styrker ligger dermed i at en grafisk designer kan lage HTML web-sider i gode HTML-verktøy, mens utviklerne tar hånd om Java-koden.

Dagens versjon bryter med prinsippet om "invadering" i koden ved at klassene som skrives må være abstrakte og subklasse en Tapestry-klasse. I "veien videre" for Tapestry, er målet avhengighetsinnsprøytning fra en IOC-container, og en slik versjon er under utvikling, og trolig klar høsten 2007.

Portlets Portlets har en standardisering fra JCP med betegnelsen "JSR-168"[28], en spesifikasjon som de største kommersielle aktører og Apache, med

[22] `http://struts.apache.org`
[23] `http://www.opensymphony.com/webwork`
[24] `http://www.springframework.org`
[25] `http://tapestry.apache.org`
[26] `http://www.nextapp.com/platform/echo2`
[27] `http://java.sun.com/javaee/javaserverfaces`
[28] `http://jcp.org/en/jsr/detail?id=168`

unntak av Microsoft, har blitt enige om. En nyere standardisering, for "Portlet 2.0", gjennom "JSR-286"[29] er på vei.

Formålet er å sørge for interoperabilitet mellom portlets og portaler ved å standardisere et sett API'er. Mange web MVC-rammeverk tilbyr god støtte til portlets, og derved vil behov for et slik anvendelsesområde enkelt kunne realiseres innenfor de allerede valgte rammeverk.

5.4.3.5 Utviklingsmiljø og test-rammeverk

Den siste obligatoriske programvaren som gjenstår for å kunne utføre re-design er nå et utviklingsmiljø hvor støtte til enkel refaktorering og testing inngår. Vi så generelt for agile metodikker behovet for gode tester som et av kriteriene for ut-flatet kostkurve. XP la stor vekt på testing, og en forutsetning for TDD er tilgjengeligheten for et enhets-test rammeverk. Agile programvare utviklere bruker ofte "xUnit" familien, og Java utviklere bruker da JUnit[30][7].

Med utgangspunkt i åpen kildekode er Eclipse som integrert utviklingsmiljø (IDE[31]) et godt valg som benyttes av en stor andel utviklere, deriblant i Forsvaret. Det nevnte JUnit test-rammeverket er integrert i Eclipse, sammen med god støtte for refaktorering.

5.4.3.6 Webtjenester og fjerntilgang

Webtjenester kan implementeres ved hjelp av ulike teknologier. Jeg har valgt ut noen mulig rammeverk, slik at det muliggjør de nye anvendelsesområder som vårt re-design skal kunne gi. Eksemplene er ikke fyldestgjørende, og vil ikke inngå i senere referanserammeverk, men kan inkluderes når behovet skulle oppstå. Alle de ulike variantene kan konfigureres inn i Spring-rammeverket, og dermed inngå i en lettvekts container. Forøvrig er alle utprøvd i referansegruppen i Forsvaret i ulik grad, og med hurtige resultater.

Rammeverk som gir SOAP-baserte tjenester:

JAX-WS JAX-WS[32] er, i følge Sun, en fundamental teknologi for utvikling av SOAP-baserte webtjenester. Standardisert gjennom JSR-224. Dette er etterfølgeren til JAX-RPC.

Axis Apache Axis[33] er en åpen kildekode implementasjon av Simple Object Access Protocol (SOAP) som W3C har som mål å standardisere. Axis er tilgjengelig i språkene C/C++ og Java.

XFire XFire[34] er en, etter min mening, enda enklere SOAP-variant.

Hessian Det finnes imidlertid andre og enklere teknologier som kan brukes når programmer skal interoperere med hverandre over et nettverk, og hvor vi ikke har krav om at vi skal være agnostisk til hvilke programmer eller andre tjenester som forbruker en tjeneste.

Hessian fra Caucho[35] er et rammeverk for binær dataoverføring. Implemen-

[29] http://jcp.org/en/jsr/detail?id=286
[30] http://www.junit.org
[31] IDE—Integrated Developing Environment
[32] java.sun.com/webservices/jaxws
[33] http://ws.apache.org/axis/
[34] http://xfire.codehaus.org
[35] http://www.caucho.com/hessian

tert som åpen kildekode i ulike språk som Java, C++, C#, Python, PHP, og Ruby. Igjen har vi et eksempel fra åpen kildekode verdenen, hvor alternativer til en standard hovedretning forenkles, og gir muligheter for raskere å komme til målet. Ulempen er at den ikke bygger på en standard.

Til integrasjon mellom systemer hvor vi har kontroll på teknologien de ulike systemene kjører på, har dette valget klare fordeler i form av ytelse og rask og enkel høynivå integrasjon. Ved å starte enkelt, kan vi senere endre om behovet krever tyngre og standardiserte tjenester.

5.5 Referanserammeverk

Med utgangspunkt i en enkel referansemetodikk, og en referansearkitektur med vekt på lagdeling og tilhørende mønstre, og nå til slutt teknologier som støtter lagdelingen og viktige designområder, er grunnen redet for å re-designe den type enterprise informasjonssystemer jeg tidligere har beskrevet.

Samlingen av rammeverk, velger jeg å omtale *referanserammeverk*, og i tråd med begrepet "template" fra [53]. Referanserammeverket skal gi hurtig igangsettelse for et re-design-prosjekt ved at vi sparer tid ved initieringen av prosjektet, og kan raskt komme i gang med å lage et vertikalt snitt av det nye re-designede systemet.

Andre varianter kan erstatte ett eller flere av rammeverkene—mye avhenger av smak og behag. Forutsetningen er at de gir tilsvarende enkel anvendelse og nedslagsfelt. Forslaget til referanserammeverk er reelle, og brukt for re-design av flere EIS i referansegruppen i Forsvaret (se appendiks A) med godt resultat. Tilsvarende godt resultat har vi oppnådd ved bruk av andre MVC-rammeverk som WebWork og SpringMVC.

En oppsummerering av mitt forslag setter rammeverkene inn i aktuelt arkitektur-lag, og henviser til teknologityper. Se tabell 5.1. Utviklingsmiljøet er inkludert til tross for at dette ikke tilhører lagdelingen, men angis da det er vesentlig for metodikk-støtten.

Arkitekturlag	**Teknologi**	**Rammeverk**
Presentasjon	MVC Java/ HTML	Tapestry
Façade	POJO	Spring
Domene-modell	POJO	Spring
Data	ORM	Hibernate
Alle	Refaktorering og Testing	Eclipse m/JUnit

Tabell 5.1: Referanserammeverk

5.6 Metodikk-forsterker?

Så til det andre og siste store spørsmål: Er hele eller deler av referanserammeverket, hvor referansearkitekturen er implisitt, en forsterker for den angitte referansemetodikken, gitt i min definisjon av metodikk-forsterker i seksjon 3.3?

Også her er forutsetningen at referanserammeverket gir et positivt bidrag for å kunne hjelpe og være et hjelpemiddel. Hjelpemiddel er et uomtvistelig faktum gjennom at arkitekturen vekkes opp fra papiret og kan leve gjennom kjørbar kode.

Tilsvarende som ved vurdering av arkitektur som forsterker for metodikken, gjelder de samme regler for tid også her. For å gjenta, anvendes metodikken x i tidsperioden:

$$\forall x \mid x \geq T_0 \wedge x \leq T_n \wedge x \notin T_{-n\ldots-1} \wedge T_{n+1\ldots\infty})$$

Men denne gangen skal vi se at det ikke kan være tvilsomt at tiden for metodikk og og anvendelse av teknologi faller utenfor.

Jeg vil nå gi en ny gjennomgang av de ulike elementene som referansemetodikken består av, og argumentasjon for om støtte eller forsterkning er til stede. Denne gang fra referanserammeverket, og implisitt referansearkitekturen, som ståsted:

Planlegg for endring—Ut-flatet kost-kurve Vi hadde de tre faktorene; objektteknologi, lett modifiserbar kode gjennom rent design og automatiserte tester, og erfaring og mot til å modifisere.

Rent design her henspeiler på kodedesignen hvor gode design-mønstre kan hjelpe. Design-mønster til tross, vi kan også denne gangen skrive dårlig kode til tross for mønstre. Tilsvarende gjelder for objekter. Selv med design-mønstre som er tiltenkt støtte innenfor objektorientering kan vi feile med koden i tråd med antimønsteret "Funksjonell dekomponering". Mulig støtte er derimot gitt.

Automatiserte tester forenkles nå gjennom et kjørbart rammeverk som anvendes innenfor metodikkens gyldighetstid. Likeledes gir utviklingsmiljøet direkte støtte for testene.

Teknologien *kan* gi støtte til at vi våger å refaktorere koden. Både med støtte til automatiserte tester, gjennom enkle rammeverk som gir store effekter med relativt få kodelinjer, og et utviklingsmiljø med god støtte til refaktorering. Alltid innenfor metodikkens gyldighetsområde i tid.

Enkelhet Denne gangen vil vi med sikkerhet bevege oss innfor det tidsintervall hvor referansemetodikken gjelder ($T_{0\ldots n}$). Ved bruk av et ferdig oppsatt referanserammeverk vil vi oppnå en enkel og hurtig oppstart med redesign prosjektet. Min erfaring er, gitt et referanserammeverk som her angitt eksisterende i form kode, at alt er klart i underkant av 1–2 timer (se appendiks A). Dermed er vi i stand til å deployere en tom applikasjon som vi enkelt kan kobles opp mot en kopi av databasen, skrive tester for aktuelle forretningsregler, og implementere disse.

Her mener jeg vi ser en reell forsterker til metodikken.

Teststrategier Med utgangspunkt i riktig og hensiktsmessig lagdeling, implementert i et referanserammeverk, vil automatiserte tester effektivt kunne settes opp gjennom foreslåtte test-rammeverk med støttet av containeren som forenkler testing.

Her vil teknologien gir den nødvendige forsterkning for TDD som angitt for metodikk. Denne konklusjonen var vel ventet gjennom kravet til

automatiserte tester i [14]. Men ytterligere støtte til enkel testing kom fra applikasjons-rammeveket. All testing vil gjelde i metodikkens tidsintervall.

Refaktorering Refaktorering av kode på et trygt sett, avhenger av en IDE som har støtte for refaktorering.

Denne støtten må erfares, og erfaring fra referansegruppen i Forvaret viser tilstrekkelig støtte for å kunne refaktorere fra det valgte utviklingsmiljøet. En slik IDE er avgjørende for effektiv refaktorering av kode, og derved systemet. Blant annet kan pakker flyttes og referanser oppdateres automatisk.

Her har vi en reell forsterker i gyldig tidsintervall.

Team-størrelse og samlokalisering Uavhengig av teknologien.

Brukermedvirkning Uavhengig av teknologien.

Dokumentasjon Teknologi i form av det språkvalget jeg har foretatt, vil kunne generere relativt nyttig dokumentasjon til bruk i visse sammenhenger. Dette er den såkalte "JavaDoc". Nytten avhenger av i hvilken grad vi er dyktige til å kommentere relevante deler av koden. Riktignok ikke som en forsterker, men snarere som teknologisk støtte til denne delen av metodikken.

Risikohåndtering Med utgangspunkt i en god og hensiktsmessig referansearkitektur, som er satt opp ved hjelp av rammeverk som dekker lagene, har vi to hovedutfall for risiko: Det ene er at referanserammeverket feiler for re-design oppgaven, og derved hadde vi en risiko. Den andre muligheten er at valget treffer godt og derved reduserer risiko for gjennomføringen.

Basert på egne erfaringer, og anbefalinger i [53, 40], så vil referanserammeverket gi redusert risiko for gjennomføringen ved både støtte til lagdelingen, som gir støtte for metodikken, og trygghet for at en hensiktsmessig ytre teknologisk ramme for prosjektet er etablert.

Her har vi en reell forsterker. Spørsmålet er om metodikkens tidsintervall er gyldig. Når jeg argumenterte i forrige kapittel om tilsvarende metodikkelement, hadde jeg ingen klar konklusjon. Mange av de samme argumentene kan nyttiggjøres igjen, men for teknologien vil jeg hevde at vi beveger oss i gyldig tidsintervall. Grunnen er at referanserammeverket er i praktisk anvendelse.

Oppsummering er gitt i tabellen 5.2 hvor jeg, tilsvarende som for arkitektur, har valgt å ikke avgrense *metodikk-støtte* (Støtte) i tid. *Metodikk-forsterker* er forkortet til MF, og alltid innenfor tidsintervallet for når metodikken gjelder.

Jeg ser fortsatt muligheten for problemer ved at ikke alle EIS'er er like og kan komme til å kreve teknologielementer som ikke allerede er dekket. Min erfaring er at det gitte referanserammeverk dekker mye, og kan i de fleste tilfeller enkelt utvides til å inkludere andre komponenter ved behov. En faktor, også denne gangen, er å betrakte teknologien generelt som et kunnskapsområde. Hensiktsmessig metodikk og riktig arkitektur kan ikke manifestere seg som et system alene. Et IT-system på papiret har ikke verdi for en virksomhet og de brukerne som trenger data-støtte før det kjørbare artefaktet benyttes. Valget av teknologi kan gi ulike utslag for både kostnader (helst billigere) og kvalitet

Metodikk-del	**Støtte**	**MF($\mathbf{T_{0...n}}$)**
Planlegg for endring...	Ja	Kanskje
Enkelhet	Ja	Ja
Teststrategier	Ja	Ja
Refaktorering	Ja	Ja
Team-størrelse...	Nei	Nei
Brukermedvirkning	Nei	Nei
Dokumentasjon	Ja	Nei
Risikohåndtering	Ja	Ja

Tabell 5.2: Teknologi metodikkforsterker?

og anvendelsesområder (bedre), og hvor effektivt et prosjekt kan gjennomføres (raskere). Kompleksitetsgraden av resultatet er en aktuell problemstilling, og jeg foreslår her, etter min oppfatning, en relativt enkel teknologi-tilnærming.

For livsyklus-håndteringen og ut-flatet kostkurve for vedlikehold, trenger vi kombinasjonen av alle tre dimensjoner. For spesielt videre vedlikehold vil jeg imidlertid vekte teknologidimensjonen opp i forhold til de to andre, grunnet mindre behov for metodikk som tidligere omtalt, og trolig en stabil arkitektur—men ikke nødvendigvis.

Basert på de foregående vurderinger, vil jeg denne gangen karakterisere referanserammeverket og tilhørende utviklingsmiljø, som en reell forsterker til referansemetodikken. Men jeg makter ikke å skille referanserammeverket fra referansearkitekturen av den grunn at sistnevnte er det faktiske grunnlaget for referanserammeverket. Muligens kan automatiserte tester og refaktorering gjennom bruk at et godt utviklingsmiljø være unntaket, men testing og refaktorering har jo som vist støtte fra rammeverkene som dekker arkitekturen, og derved trolig ikke gir forsterkningen alene.

Kapittel 6

Diskusjon og konklusjon

Utgangspunktet for denne oppgaven var effektiv modernisering av enterprise informasjonssystemer (EIS). Jeg valgte tre dimensjoner for å å løse oppgaven; *metodikk*, *arkitektur*, og *teknologi*. Dimensjonene mener jeg har sterke avhengigheter til hverandre, og jeg ønsket å se disse i et agile perspektiv. Ved å ta utgangspunkt i en enkel metodikk, har jeg forsøkt å gi forsterkning fra dimensjonene arkitektur og teknologi. Men re-design av et eksisterende system må i en dynamisk verden med stadige endringer og krav til reduserte kostnader, også legge grunnlaget for at videre endring og vedlikehold kan skje med en fortrinnsvis utflatet kostnadskurve. I tillegg må grunnlaget legges for nye anvendelsesområder som høynivå integrasjon og gjenbruk i en virksomhetsportal.

Jeg vil ta utgangspunkt i de mange spørsmålene jeg stilte for oppgaven i seksjon 2.3, og benytte de som diskusjonsgrunnlag før jeg går inn på konklusjonen i 6.1.

Metodevekt

Trenger vi en tung og besværlig metodikk *med mange støtte-artefakter, eller kan denne være agile og i form av metodikkelementer?*

En hovedkilde til problemer med fossefallsmodellen er i følge Boehm, dens vektlegging av dokumenter som kriterier for ferdigstillelse av krav og designfaser. Dette fungerer dårlig for interaktive sluttbrukerapplikasjoner. Dokumentdrevne standarder har påtvunget mange prosjekter å skrive detaljerte spesifikasjoner av dårlig forstått funksjonalitet, som etterfølges av design og implementering av løsning for feil problem[15]. Her trekker jeg paralleller til utgangspunktet for fossefallsmodellen som jeg mener er en statisk verden, hvor en går fra spesifikasjon—gitt at denne er riktig, til implementasjon i en forutsigbar faseorientert prosess. Mange av idéene til Taylor[77] om "Scinetific Management" gjør seg gjeldende i denne prosessen. I tillegg til å være svært dokumentsentrisk, er modellen etter mitt mening lite egnet for brukermedvirkning utover spesifikasjonsfasen, og analysefasen.

Boehm ser likhetstrekk mellom en disiplinert prosess og XP, og hevder de følger en spiral-filosofi; begge er både risiko og inkrementdrevet[16]. En stor forskjell, etter min mening, er at XP's sykler er ekstremt mye kortere, og ikke følger noen faser med mindre man hevder at fasene er ekstremt korte. Mitt hovedargument mot spiralmodellen er at den bruker mye tid på en slags

"spesifisering" av risiko, for deretter å være faseorientert. Kjørbare artefakter tidlig i prosjektet vil ikke typisk tas fram bortsett fra prototyper, og dette mener jeg kan være risikodrivende. Brukermedvirkning vil således også i denne modellen være unaturlig som en kontinuerlig del av prosjektet grunnet modellens faseorientering som kommer i store sykler.

I følge Fowler og Highsmith, er de agile metodene mer rettet mot *tilpasning* enn å følge en fastsatt plan, hvilket fremmer *endring* i motsetning til å motsette seg endring ved blindt å følge en plan. Erfaringer jeg selv har sett i referansegruppens arbeide, viser at metodikkelementene innen en agile kontekst gir god endringsvillighet, og dette er nødvendig fordi endringer kommer, også sent i utviklingsløpene (se appendiks A). En vesentlig forskjell mellom de tradisjonelle engineeringsdisiplinene som gjelder for konstruksjon av broer og skyskrapere—og systemutvikling av programvare, er at førstnevnte i mye større grad trenger planer og tegninger for å konstruere de *forutsigbare* artefaktene riktig første gangen.

Svakhetene, og motstanden fra utviklere, til dokumentsentriske og tunge prosessmodeller har gitt en framvekst av lettvekter metodikker og teknikker med fokus på hurtig tilbakemelding og utvikling av riktig system til rett tid, sterk brukermedvirkning, og hvor bi-artefakter tones kraftig ned. Unødvendig metodevekt vil etter min mening også virke forsinkende på prosessen med å ta fram et kjørbart artefakt.

Større grupper svekker god direkte kommunikasjon. Dette er godt underbygget av Hohmann, hvor han angir ulike potensielle kommunikasjons-stier etter som team-størrelsen vokser[48]. Eksempelvis gir et team på fire bare 28 mulige stier, mens et team på 11 gir hele $11,253$. Slik økning i antall kommunikasjons-stier mener jeg krever større metodevekt for å kunne takle utfordringene. En enklere variant er å holde team-størrelsen nede, for å hindre behovet for økt metodevekt. Dette gir i følge [67] færre feil og marginal økning i utviklingstiden. I tillegg til at små team er mye billigere.

Desto mindre kunnskap og faglig dyktighet et team innehar, jo mere nytte kan de dra ut av dokumentbaserte prosesser med mange detaljerte steg som må gjennomføres for om mulig komme i mål[46]. I tillegg til manglende kunnskap og dyktighet, gir fravær av brukermedvirkning størst risiko for prosjektet[74]. Agile metoder er orientert mot *mennesker* og mindre mot prosesser som skal hjelpe alle, uansett kunnskapsnivå. De agile prosessene skal fremme teamet bestående av gode utviklere og gode brukere med kompetanse om problem-området[35]. Dette er også vist i referansegruppen og deres bruk av metodikkelementer.

Agile metodikk er etter min mening bedre enn tunge metodikker når prosjektene ikke er for store. På den annen side mener jeg slik som Cockburn hevder[24]), at svært dyktige utviklingsteam ikke nødvendigvis trenger å følge en gitt metodikk, men at enkle metodikker kan være til hjelp. For metodikkens vedkommende, den iboende kunnskapen medfører trolig at de allikevel implisitt anvender en form for metodikk.

Re-design og nyutvikling

Metodikken, enten man re-designer eller utvikler nye systemer, kan kanskje være den samme—man har bare kortere og mindre prosjekter uten samme grad av problemløsning inn mot problemområdet ved re-design?

Jeg gikk ut fra en antagelse om at det som virker godt for utvikling av helt nye systemer, også kan være et godt utgangspunkt for re-design. Denne antagelsen mener jeg å finne støtte til fra erfaringer gjort i referansegruppen (se appendiks A).

En faktor ved nyutvikling er at brukerne ikke alltid vet hva de vil ha[20], hvilket påvirker vanskelighetsgraden og omfanget av å utvikle systemer. Når denne faktoren er redusert slik tilfellet er for re-design, mener jeg et prosjekt for re-design er enklere og av mindre omfang enn hva tilfellet er for nyutvikling.

Ved re-design slik jeg legger opp til, vil den eksisterende relasjondatabasen gjenbrukes, sammen med etablert forretningslogikk i form av forretningsregler. Dette utgjør sammen med medvirkning av de brukerne som kjenner eksisterende system godt, en forenklet spesifikasjon, og dermed færre aktiviteter.

Når Naur[61] argumenterer for at programmering (design og programmering) er teoribygging, kan dette sprike med re-design? Hva når vi tar utgangspunkt i gamle systemer med et eksisterende databaseskjema og forretningsregler, og kanskje bare brukernes erfaringer med det eksisterende systemet, og vi ikke får overført innebygd kunnskap fra de opprinnelige utviklerne? Har vi da mistet dette aspektet ved at programmering er teoribygging? Jeg mener at vi prøver å bygge teorien slik at den endelig sitter. Men idéelt sett skal de samme, eller et utvalg av de samme, utviklerne som utviklet det opprinnelige systemet, stå for moderniseringen sammen med et utvalg superbrukere. Derved vil den iboende kunnskapen enklere overføres.

Videreutvikling og utflatet kostkurve

Et re-designet system er ikke endelig, det er utgangspunktet for videre utvikling. Og om denne metodikken passer like godt for re-design som for nyutvikling, vil nytteverdien for vedlikehold økes ytterligere? Hvordan kan vi legge grunnlaget for videreutvikling gjennom realistiske rammer?

Re-design er et skritt på veien. Videre utvikling og endring kommer i de etterfølgende rundene. Og velger vi først en agile tilnærming, er det hensiktsmessig å videreføre samme tilnærming for systemets videre livssyklus.

Ett mål med modernisering av eksisterende EIS er nettopp en utflatet kostkurve ved senere vedlikehold. En av de universelle antagelsene innenfor software engineering er at kostnaden med endring av et program øker eksponentielt over tid[13]. Hvis vi kan flate ut kostnadskurven for systemer, holder ikke lengre de gamle antagelsene om hvordan vi best skal utvikle programvare. I følge Beck[13] kan en utflatet kostnadskurve oppnås, men ikke uten videre. En nøkkelteknologi er objekter slik som Kristen Nygaard og Ole Johan Dahl beskrev på slutten av 1960-tallet. Med utgangspunkt i objekter, erfarte Beck en utflatet kurve når koden var enkel å modifisere. Og for at koden skulle være enkel å modifisere kreves flere faktorer som *enkelt design* uten overflødige designelementer, *automatiserte tester* slik at vi har tiltro til at vi ikke kommer i skade for å endre eksisterende oppførsel til systemet. Til slutt; *Mye erfaring og mot til å modifisere designet* slik at når tiden er inne, tør vi gjøre endringer. Ambler[7] uttrykker tilsvarende syn ved at kostnadene til endring blir lavere med bruk av moderne (agile) teknikker, enn hva som har vært tradisjonen.

Objektteknologien er gjennomgående i både språk og rammeverk som jeg foreslår. Enkelt design søker jeg å oppnå ved bruk av referansearkitekturen

som jeg mener generelt passer for EIS. Automatiserte tester og støtte for refaktorering er på plass i utviklingsmiljøet. I tillegg til rammeverk som gir store effekter med relativt få kodelinjer. En kodeimplementert arkitektur med løst knyttede lag forenkler videre vedlikehold gjennom å tilrettelegge for refaktorering og automatiserte tester og derved reduseres kostnadene.

Men mot og erfaring er opp til dyktige utviklere å klare av. Jeg mener å kunne observere fra referansegruppen i Forsvaret (se appendiks A) at viderutvikling følger de samme baner som utvikling, og at kostnadskurven ikke øker eksponentielt.

Uavhengig arkitektur og nye anvendelsesområder

Skal arkitekturen være tilpasset en gitt leverandør's produkter eller en konsortium bestemt plattform, eller kan den designes til å være uavhengig og smidig? Og hvordan sikrer vi gjennom arkitekturen enkel høynivå integrasjon, videre tjeneste-bygging, og enkel utplassering, drift og forvaltning?

Framveksten av lettvekter-containere har sitt utspring i at mange finner den opprinnelige J2EE-plattformen unødvendig omfattende, komplisert, og tildels uhensiktsmessig, og hindrer god bruk av objektorientering[53]. Alle lettvekter-containere jeg har undersøkt kommer fra åpen kildekode prosjekter, og fordrer kun en Java-plattform for å benyttes. I forhold til referansearkitekturen vil spesielt tjenestelaget og domene-modellen dekkes av en lettvekts container. Men vi kan videre se containeren som integratoren som syr sammen andre rammeverk og derved i sum representerer alle lagene.

Jeg gjorde ett direkte brudd med XP på arkitektur-området. Min tilnærming til effektiv re-design av EIS er å ta utgangspunkt i en referansearkitektur som jeg mener kan være generell for mange enterprise informasjonssystemer. Samtidig kan en slik referansearkitektur sees på som en obligatorisk brukerhistorie som tas fra planleggingsspillet, og som derved allerede rommer hele den initielle arkitekturen i form av et vertikalt snitt av systemet med hele arkitekturen ferdig refaktorert.

I min arkitektur vektlegger jeg et tynt tjenestelag, façadelag, som gir et grovkornet grensesnitt til en fin-masket domenemodell[40]. Et mønster omtalt som "Session Façade"[2] argumenterer for en slik fasade over forretningsobjektene for å unngå tett kobling mellom klienter og forretningslaget. Med et Façadelag på plass har vi lagt grunnen for enklere tjenesteorientering, i tillegg til å være et veldefinert grensesnitt for ulike brukergrensesnitt. I dette laget kan vi også bygge inn ulike sikkerhetsmekanismer og andre enterprise tjenester som er tversgående("cross-cutting")[40]. Façade laget blir applikasjonens tilgangspunkt for omverdenen. Laget er svært viktig for å kunne realisere de ulike anvendelsesområdene vi trenger i dag, og gjøre det enkelt å innføre nye ved behov.

Design av en tjenesteorientert arkitektur kan by på utfordringer. Trenden, i følge Fowler[33] og Evans[28], blant "SOA-folkene" er å gjøre domene-modellen anemisk ved å legge alt av metoder/operasjoner inn i tjenestelaget. Dette bryter fundamentalt med objektorientert tankegang, hvor både data og operasjoner skal kapsles inn. Etter min mening oppnår vi best resultat ved å først bygge en objektorientert domene-modell, og deretter legge et tynt tjenestelag på toppen av denne. Siden kan utvalgte deler fra det tynne tjenestelaget "SOAfiseres".

Referansearkitekturen som er omtalt er utprøvd og brukes i referansegruppen i Forsvaret (se appendiks A) både for nyutvikling og re-design med godt resultat, og vil fortsatt benyttes som preferert referansearkitektur for flere enterprise informasjonssystemer i overskuelig framtid. Erfaringene har vist at referansearkitekturen også kan passe inn i deler av det SOA-arbeidet som i økende grad nå vektlegges av Forsvaret. Etter min mening gir en god velprøvd referansearkitektur tilsvarende støtte til normalt dyktige utviklings-team, som en tilstrekkelig metodikk gjør. Har vi en referansearkitektur som erfaringsmessig er god—trenger vi ikke å feile på dette området, men gjenbruker et godt "mønster". Denne arkitekturen sentraliserer applikasjonen slik at vi oppnår enkel utplassering, drift og forvaltning.

Teknologivalg

Hvilken teknologi *bør vi vektlegge når vi re-designer? Og hvilke rammeverk og verktøy implementerer teknologien?*

For det første er det objektorienterte paradigme valgt, støttet av to supplerende teknologier; avhengighetsinnsprøytning og aspektorientert programmering. Fordelen med det objektorienterte paradigme, er at det på en god måte kombinerer styrkene fra de data-sentriske og oppførsels-sentriske tilnærmingene. Allikevel er det ikke noe universalmiddel idet det fortsatt er lett å lage dårlige systemer og vanskelig å lage gode[66]. For å kunne implementere objektteknologien i kjørbare systemer, behøver vi programmeringsspråk. Språket Java er svært utbredt og har nylig blitt åpen kildekode. Systemer implementert i Java kan kjøres uten rekompilering på de fleste plattformer gjennom sin Java Virtuelle Maskin (JVM) liggende over operativ-systemet.

Avhengighetsinnsprøytning er en felles karakteristikk for lettvekts- containere [34], hvor hovedkontrollen for programmet flyttes ut til et rammeverk. Aspekt Orientert Programmering (AOP), er en teknologi som er egnet til å understøtte OO der hvor denne kommer til kort. AOP kan brukes til å programmere aspekter eller "cross-cutting concerns", som logging, sikkerhet, transaksjoner, og andre behov som må dekkes i for eksempel en enterprise applikasjon.

Begge de supplerende teknologiene er implementert i det viktigste rammeverket, applikasjonsrammeverk eller lettvekter container, hvor hovedformålet er blant annet å gjøre J2EE enklere å bruke, og å fremme god objektorientert programmerings-praksis[36, 76, 54, 68]. Enterprise funksjonalitet er ikke inkludert i rammeverket, men integreres ved hjelp av avhengighetsinnsprøytning og AOP. Sist men ikke minst, slike rammeverk gjør testing lettere ved å legge et abstraksjonslag over J2EE API'er som normalt er vanskelig å teste, slik at vi kan teste mot dette enkle abstraksjonslaget i stedet[53].

Jeg valgte Spring-rammeverket på grunn av størst utbredelse, og gode erfaringer fra referansegruppen i Forsvaret (se appendiks A), hvor rammeverket er fast inventar i de sammenhenger jeg refererer til.

For det andre trenger vi en teknologi som tetter gapet mellom det objektorienterte paradigme og relasjonsdatabasen, nemlig objekt-relasjon tilordningen (ORM). Når vi benytter et objektorientert språk og persisterer dataene i en relasjonsdatabase, kan vi støte på problemer som stammer fra forskjeller i de to paradigmenes former for representasjon av virkeligheten (problemområdet); objekter og tabeller (relasjoner). Dette gir et gap mellom paradigmene

som må tettes[32, 11]. I komitée-arbeidet med spesifisering av J2EE teknologien, nærmere bestemt entitets-bønnene, er problemet med objekt-persistering blitt løst. Allikevel søker stadig flere å unngå denne teknologien[53, 40], på grunn av dens kompliserte spesifikasjon, lave ytelse, og ikke minst; dreining vekk fra objektorienteringen. Alternative løsninger på objekt-relasjon tilordningen, i form av rammeverk som inkluderer ORM-mønstre, er kommet. Disse oppnår stadig bredere aksept. ORM mediaterer applikasjonens interaksjon med en relasjonsdatabase, og fristiller utvikleren til å konsentrere seg om forretnings problemområdet[11] og domene-modellen.

Jeg valgte Hibernate både fordi det er det ORM rammeverket som har størst utbredelse innen Java/J2EE verdenen, og er godt integrert med Spring gjennom forenklede API'er. Referansegruppen i Forsvaret (se appendiks A) benytter samme rammeverk i mange enterprise informasjonssystemer med godt resultat.

For det tredje er vi avhengig av et rammeverk som implementerer Trygve Reenskaug's MVC-mønster, slik at vi kan lage web-grensesnitt uavhengig av modellen. Senere kan vi gjenbruke modellen (og kontrolleren), og lage alternative view som for eksempel portlet hvis behovet dukker opp.

Web MVC-rammeverk, som jeg har fokusert på, kan deles inn i to grupper, som angir ulik filosofi for virkemåte og tilnærming, de anmodningsdrevne og hendelsesdrevne. Jeg har argumentert for det hendelsesdrevne grunnet større nærhet til objektorienteringen, og mindre teknisk fokus på servlet-teknologien. På den annen side benytter referansegruppen den anmodningsdrevne varianten med godt resultat.

Web MVC-rammeverk finnes det mange av fra åpen kildekode verdenen. Jeg har valgt Tapestry fordi det er enkelt, objektorientert, og lett integrerbart i Spring. Valget har derimot ingen avgjørende betydning for oppgaven. I referansegruppen i Forsvaret (se appendiks A) benyttes WebWork og SpringMVC.

Til slutt må vi ha et utviklingsmiljø som inkluderer automatiserte tester og støtter refaktorering av koden.

Her finnes mange valg, hvor jeg valgte det åpen kildekode utviklingsmiljøet Eclipse, som er mye brukt i Javaverdenen, og benyttes med godt resultat av referansegruppen (se appendiks A). Eclipse gir god støtte for refaktorering, i tillegg at det integrerer JUnit test-rammeverket.

Åpen kildekode

Kan åpen kildekode gi nødvendig støtte til effektivitet og leverandøruavhengighet, og samtidig redusere kostnader?

Flere og flere, inklusive offentlige virksomheter, har skjønt betydningen av åpen kildekode.

Etter siste regjeringsskifte, har også den Norske politiske kursen for programvare endret seg. Vi fikk nylig IT-meldingen som anbefaler bruk av åpen kildekode i offentlig forvaltning[30].

På oppdrag fra det Amerikanske forsvarsdepartementet (DoD), utarbeidet organisasjonen MITRE en rapport om departementets bruk av fri åpen kildekode programvare i 2002/2003. Hovedkonklusjonen fra analysen viste at denne type programvare spilte en mer kritisk rolle enn antatt[4].

Fra det Kanadiske forsvarets forskningsavdeling kommer en rapport som konkluderer med at FOSS tilbyr konkrete muligheter for kostnadsreduksjoner, økt fleksibilitet, og teknologi innføring[71].

Fordeler som muligheten til å studere kildekoden slik at vi virkelig kan vite hva programmene gjør er spesielt viktig når sikkerhet skal vurderes. Lavere kostnader og leverandøruavhengighet er en direkte konsekvens ved valg av åpen kildekode. Support er mulig å kjøpe for enkelte produkter, om man trenger noe utover det man kan få fra åpne brukergrupper og forumer.

En ulempe er at vi ikke i utgangspunktet har en garantert support. Supporten er dugnadsdrevet og gratis (gratis er ikke en ulempe), men denne kan i økende grad kjøpes og garanteres for de som ønsker det.

En annen ulempe med åpen kildekode programvare, er hvis vi er avhengig av sikkerhetsgodkjenning eller sertifisering av programvare for at denne skal kunne kjøres på høygraderte nettverk. Dette gjelder virksomheter som omfattes av sikkerhetsloven.

En ulempe kan være at produkter forsvinner, slik også kommersielle produkter kan, men det viktige her er å velge åpen kildekode produkter med stor utbredelse og derved redusere risikoen for at dette skjer.

I denne oppgaven har jeg kunnet velge både språk, rammeverk og utviklingsmiljø fra åpen kildekode verdenen. Erfaringene fra referansegruppen i Forsvaret (se appendiks A) er gode. Rammeverkene øker effektiviteten, gir leverandøruavhengighet, og har ikke gitt kostnader utover innkjøp av bøker og ett kurs i Spring-rammeverket.

Metodikkforsterker

Avslutningsvis og det mest sentrale spørsmålet; vil vi finne fordeler av å se disse tre dimensjonene samlet for å forsterke virkningen av en agile metodikk, og hvordan støtter dimensjonene hverandre?

Etter at jeg satte opp mitt forslag til de agile metodikkelementer som jeg ville benytte for re-design, definerte jeg en metodikkforsterker slik at jeg kunne bruke mine anbefalinger fra arkitektur og teknologi, og vurdere disse opp mot definisjonen av metodikkforsterker.

Jeg forsøkte å argumentere for referansearkitekturen som forsterker til metodikk-valget. Jeg viste at referansearkitekturen kunne være en guide til å velge en erfaringsmessig riktig og god arkitektur for re-design av EIS. En ulempe var at den kunne gi gi en begrensning i frihet og "låsing" som ikke ville vært til stede om vi sto helt fritt, slik for eksempel ekte XP legger opp til. Mitt argument var at den anbefalte arkitekturen erfaringsmessig var minimal og dekkende, og ga bidrag til enklere å øke anvendelsesområder og berede grunnen for nye. Samme resultat kunne jeg ha oppnådd ved å anvendt XP's tilnærming til arkitektur, men her mener jeg det er raskere å benytte min tilnærming, i tillegg til at det sikrer at en erfaringsmessig god arkitektur blir implementert.

Min konklusjon med hensyn på om referansearkitekturen var en forsterker for metodikken var ikke entydig, og delvis basert på at referansearkitekturen kan være bestemt før metodikken har kommet til anvendelse.

I teknologikapitlet satte jeg opp mitt forslag til et referanserammeverk som implementerte referansearkitekturen.

For metodikkelementet "Enkelhet": Ved bruk av et ferdig oppsatt referanserammeverk vil vi oppnå en enkel og hurtig oppstart med re-design prosjektet.

For metodikkelementet "Teststrategier": Med utgangspunkt i riktig og hensiktsmessig lagdeling, implementert i et referanserammeverk, vil automatiserte tester effektivt kunne settes opp gjennom foreslåtte test-rammeverk med støttet av containeren som forenkler testing. Her vil teknologien gir den nødvendige forsterkning for TDD som angitt for metodikk. Ytterligere støtte til enkel testing kom fra applikasjons-rammeveket.

For metodikkelementet "Refaktorering": Erfaring fra referansegruppen i Forvaret viser tilstrekkelig støtte for å kunne refaktorere fra det valgte utviklingsmiljøet. En IDE er avgjørende for effektiv refaktorering av kode, og derved systemet. Blant annet kan pakker flyttes og referanser oppdateres automatisk.

For metodikkelementet "Risikohåndtering": Basert på egne erfaringer, og anbefalinger i [53, 40], så vil referanserammeverket gi redusert risiko for gjennomføringen ved både støtte til lagdelingen, som gir støtte for metodikken, og trygghet for at en hensiktsmessig ytre teknologisk ramme for prosjektet er etablert.

Basert på de foregående vurderinger, karakteriserte jeg referanserammeverket og tilhørende utviklingsmiljø, som en reell forsterker til referansemetodikken. Men jeg kunne ikke skille referanserammeverket fra referansearkitekturen av den grunn at sistnevnte er det faktiske grunnlaget for referanserammeverket. Muligens kan automatiserte tester og refaktorering gjennom bruk at et godt utviklingsmiljø være unntaket, men testing og refaktorering har jo som vist støtte fra rammeverkene som dekker arkitekturen, og derved trolig ikke gir forsterkningen alene.

Referansegruppen har oppnådd gode resultater ved bruk av lette og smidige metodikkelementer i kombinasjon med den omtalte referansearkitektur implementert som et referanserammeverk. Jeg har selv gode erfaringer i forbindelse med de to siste CASE'ne hvor jeg deltok i arbeidet. Etter min mening er referanserammeverket svært sentralt for at metodevekten kan holdes på et tilstrekkelig og lavt nivå. Det kan argumenteres mot at det ikke er anvendt noen konkret metodikk i referansegruppen, men metodikkelementer er tilstede.

Hvis en metodikk skal anvendes for å ta fram et artefakt, i form av et kjørbart re-designet enterprise informasjonssystem, er det etter min mening avgjørende å inkludere dimensjonene arkitektur og teknologi. Uansett metodikkens velegnethet, gir den ikke kjørbare systemer anvendt isolert. Men dimensjonen arkitektur anser jeg ikke tilstrekkelig alene til å forsterke metodikken. Til det trenger vi dimensjonen teknologi i tillegg. Det er først etter at vi kombinerer disse to dimensjonene, at jeg sikkert vil hevde at vi oppnår en forsterkning til metodikken. Vi kan dermed ikke se de tre dimensjonene isolert.

Arkitekturen som kunnskapsområde er alene viktig for å støtte metodikken. Fordi; uansett hvor god metodikk vi anvender, og eventuelt ikke gjør bruk av, så kan det få svært uheldige konsekvenser om vi ikke tar inn over oss fordeler og ulemper vedrørende arkitekturen. En god metodikk brukt i et prosjekt som feiler på arkitekturvalget, vil trolig ikke lykkes slik jeg ser det.

Når vi re-designer enterprise informasjonssystemene, trenger de teknologi for å kunne kjøre og brukes. Uten teknologi blir metodikk og arkitektur en akademisk men viktig øvelse, som ikke gir virksomheten og brukerne den

data-støtten de trenger—en av de reelle beveggrunner for systemutvikling. Tilsvarende som det gir mindre mening hvis vi ikke evner å forstå hva vi skal gjøre og hvordan vi skal komme dit.

6.1 Konklusjon

Målet med denne oppgaven var å svare på hypotesen: *"Agile metodikk er velegnet for modernisering eller re-design av enterprise informasjonssystemer. Men agile metodikk må følges av arkitektur og teknologi som forsterker de viktige aspektene ved agile metodikk."*

Jeg mener at agile metodikk er velegnet for modernisering av EIS, men en enkel metodikk er avhengig av forsterkning fra dimensjonene arkitektur og teknologi for å gi nytte, slik jeg nettopp argumenterte for.

For det første er moderniseringsprosjekter mindre enn tilsvarende prosjekter med nyutvikling, og følgelig kreves mindre metodevekt. Mindre behov for metodevekt gjør enkle og smidige metodikker velegnet.

For det andre må moderne informasjonssystemer bygges i en arkitektur som gir mulighetrom for enklere å ta nye anvendelsesområder i bruk, og her holder ikke isolert bruk av en agile metodikk.

Og for det tredje må arkitekturen løftes opp fra papiret slik at den reelle nytten av arkitekturen kommer til anvendelse, og med en teknologi-implementering av arkitekturen mener jeg å finne at disse to dimensjonene gir forsterkning tilbake til en agile metodikk i form de foreslåtte metodikkelementer.

"Silver Bullet" Jeg kan ikke sikkert hevde at jeg her har en "Silver Bullet"[20], men ved å ta utgangspunkt i en enkel metodikk forsterket av de to dimensjonene arkitektur og teknologi, vil jeg hevde at kulen har fått en edlere valør. Fortsatt kan det finnes *kompleksitet*, men jeg mener jeg har kunnet bidra til en reduksjon. Behovet for *konformitet* vil alltid være tilstede, og jeg mener jeg har bidratt til at den lettere kan oppnås. *Endringer* vil komme, men vi er bedre forberedt og grunnlaget er lagt for en utflatet kostkurve. *Usynligheten* er gjort mindre for de brukerne som deltar i prosjektet gjennom den aktive medvirkningen, og et kjørbart artefakt som stadig utvikler seg i riktig retning. Om sponsoren undrer seg på status, er det bare å gå til nærmeste webleser, skrive inn URL'en til systemet og se.

Faglig sett er jeg stolt av å ha kunnet tatt turen gjennom tre store dimensjoner, som hver for seg kunne dekket en hovedoppgave. Jeg mener jeg har evnet å gå tilstrekkelig i dybden for hver av disse, i tillegg til å ha kombinert dimensjonene slik at de innbydes avhengigheter er godt belyst. Jeg mener jeg har evnet å holde abstraksjonen oppe på et rimelig nivå for diskusjon, samtidig som jeg har et detaljeringsnivå som kan gi en praktisk gevinst.

Tillegg A

Erfaringer

Forsvaret har generelt hatt en dreining over mot en IT-politikk med vektlegging av såkalt Nettverksbasert Forsvar (NbF[1]), og økt fokus på tjenesteorientert arkitektur (SOA). Et stort prosjekt, "Modernisering av kjernetjenester" (P8009), i størrelsesorden et tresifret antall millioner[2] ble vedtatt av Forsvarsdepartementer i år. Prosjektet skal forbedre de såkalte kjernetjenestene i Forsvarets IT-plattform. Her vil blant annet enterprise portal som integrerer ulike tjenester blant annet i form av portlets komme. Likeledes skal det gi nødvendig infrastruktur, deriblant applikasjons-tjenere og registertjenester for å understøtte det tekniske aspektet ved tjenesteorientering. Dette nevner jeg som eksempler på relevansen av de nye anvendelsesområder jeg beskriver i denne oppgaven.

Konkrete erfaringer som denne oppgaven referer til fra Forsvaret, stammer i all hovedsak fra en systemutviklingsgruppe som utfører oppdrag innen kategorien skreddersøm for kunder internt i Forsvaret. Organisatorisk tilhører gruppen Fellessystemer og ApplikasjonsTeknologi (FAT), FLO/IKT/BST[3]. Gruppen besto i 2004 av to systemutviklere, og økte til tre i 2005. I tillegg har eksterne konsulenter deltatt i noen prosjekter. Jeg refererer denne gruppen som referansegruppen eller gruppen. Andre oppdrag og prosjekter er utført av referansegruppen, uten at de er omtalt her.

Generelt for prosjektene

Siden våren 2004 har gruppen benyttet en minimal og smidig metodikk, anvendt den angitte referansearkitekturen og tilsvarende referanserammeverk som oppgitt, med unntak av andre web MVC-rammeverk (WebWork og SpringMVC). Tapestry er kun anvendt av en kollega fra en annen gruppe og meg selv siden høsten 2006, etter at jeg satte opp referanserammeverket i denne konfigurasjonen. Utviklingsmiljøet Eclipse er benyttet av alle, sammen med testrammeverket JUnit. Java er språket som har representert objektteknologien.

Oppfølgingssystemet JIRA[4] er benyttet for alle prosjekter etter mars 2006.

[1] NbF—Nettverksbasert Forsvar. En introduksjon til NbF finnes i [78, 60]

[2] Eksakt antall millioner er unntatt offentlighet

[3] Forsvarets logistikkorganisasjon/informasjons og kommunikasjonstjenester/beslutningsstøttetjeneter

[4] JIRA—"Bug Tracking, Issue Tracking, & Project Management" fra `http://www.atlassian.com/software/jira`

Her er ønsker, krav, feil, status, estimert tid, og løpende medgått tid registrert. Både utviklere og brukere registrerer oppgaver og feil, men bare utviklerne registrerer status og tid. Flere oppgaver og feil er ikke registrert med medgått tid, og gir av den grunn ikke eksakt bilde på ressursbruk.

Innledende grovspesifikasjon og tilbud, var krav til formell oppstart av prosjektene. Disse spesifikasjonene ble endret underveis etter som nye behov oppsto eller ble erfart. Dette har ikke sett ut til å gi nevneverdige utfordringer for utviklingsløpene. Prosjektene har levert på tid den funksjonaliteten som prosjektetenes deltagere hele tiden ble enige om. Generelt er både kundetilfredshet og brukertilfredshet god.

De ulike prosjektene er gjennomført uten bruk av en eksplisitt metodikk, men de nevnte metodikkelementer er anvendt som beskrevet nedenfor. Arkitektur og teknologi er benyttet uten tanke på at dimensjonene skulle forsterke en metodikk.

Under utviklingen har alle systemer vært tilgjengelig tidlig i prosjektløpet for super-brukerne via en webleser. Nye kjørbare versjoner ble testet og lag ut hver dag til tider, men også med flere dagers mellomrom i de tilfeller hvor ingen ny funksjonalitet var lagt til.

Generelt er det en del endringer sent i utviklingsløpet, uten at dette har sett ut til å gi særlige ekstrakostnader. Dette er ikke basert på målinger, men min subjektive oppfatning av tilstanden. I følge en av utviklerne anser han det som en prosess med kontinuerlig "forbedring" og utvidelse av systemene. FERDABALL er unntaket med en stor endring etter første versjon av re-design var avsluttet.

Alle systemer kjører på en sentral applikasjons-tjener, og nås fra en standard webleser. Normalt benyttes et sikkerhetsgradert begrenset nettverk når systemene er produksjonssatt.

Videre vedlikehold skjer på tilsvarende måte som utviklingen ellers er utført på. Her videreføres samme metodikkelementer, arkitekturvalg, og rammeverk.

Referansemetodikk

Jeg gir her en gjennomgang av de ulike elementene for referansemetodikken, og hvordan disse generelt er benyttet av referansegruppen. Tabellen A.1 gjengir de ulike metodikkelementene i referansemetodikken.

Metodikk-del	**Se seksjon**
Planlegg for endring—Utflatet kostkurve	3.2.3.1
Enkelhet	3.2.3.2
Teststrategier	3.2.3.3
Refaktorering	3.2.3.4
Team-størrelse og samlokalisering	3.2.3.5
Brukermedvirkning	3.2.3.6
Dokumentasjon	3.2.3.7
Risikohåndtering	3.2.3.8

Tabell A.1: Referansemetodikk oversikt

Planlegg for endring—Utflatet kostkurve Alle prosjekter har benyttet objektteknologi representert ved språket Java. Større refaktoreringer har skjedd ved tilfeller av vesentlige designendringer. Mindre refaktoreringer

skjer oftere. Automatiserte tester støtter refaktorering, men jeg er usikker på i hvor stor grad de faktisk bidrar. Refaktoreringsegenskapene til Eclipse gir ytterligere støtte.

Enkelhet Ved bruk av et ferdig oppsatt referanserammeverk har gruppen oppnådd en enkel og hurtig oppstart med prosjektene (både for nyutvikling og re-design). Nyutvikling og re-design oppleves i mindre grad som forskjellig.

Erfaringen er at et referanserammeverk som her angitt eksisterende i form kode, er oppsatt og klart for nytt prosjekt i underkant av 1–2 timer. Deretter er gruppen i stand til å deployere en tom kjørbar applikasjon med et vertikalt snitt av arkitekturen.

Teststrategier Spesielt innenfor lagdelingen fra det tynne tjenestelaget og nedover benyttes automatiserte tester. Gruppen har ingen sterkt påtvunget teststrategi som alltid følges, men tester skrives, men sjelden i forkant.

MVC-laget's view-komponent oppleves som vanskeligere å sette opp tester for, og krever mere ressurser. Praksis i gruppen viser at testdekningen for dette laget er lav i form av automatiserte tester, og derved testes dette laget i større grad manuelt av både utviklere og brukere i det til enhver tid kjørbare systemet.

Refaktorering Refaktorering av kode på et trygt sett, avhenger av en IDE som har støtte for refaktorering. Denne støtten må erfares, og erfaring fra referansegruppen viser tilstrekkelig støtte for å kunne refaktorere fra det valgte utviklingsmiljøet. En slik IDE virker greit for refaktorering av kode, og derved systemet. Blant annet kan pakker flyttes og referanser oppdateres automatisk. Rammeverkene er erfart til å gi ytterligere støtte ved at de gir store effekter med relativt få kodelinjer. Refaktorering har ikke medført problemer så langt jeg har kunnet observere.

Team-størrelse og samlokalisering Gruppen benytter et såkalt team-rom, organisert med stort "white-board", projektor, og hvor skrivebordene er satt sammen i en oval form slik at alle sitter vendt mot hverandre. Vanligvis foregår all utvikling og kundedeltagelse i dette rommet. Det er tre faste systemutviklere fra referansegruppen, til tider supplert med innleid hjelp. Antall samtidige utviklere på et prosjekt har aldri oversteget seks. Vanligvis har antallet ligget på to–tre utviklere.

Brukermedvirkning Noen kunder og superbrukere er stasjonert i Stavanger. Andre arbeider i Oslo-området, men på lokasjoner som er forskjellig fra den referansegruppen holder til i. Kun ett av prosjektene har kunde og superbrukere i samme bygning som utviklerene. Kontakten har foregått gjennom epost, telefon, oppfølgingssystemet, og arbeidsmøter. På arbeidsmøtene brukes det kjørbare systemet, så langt det er kommet, aktivt ved gjennomgangen. Selv for prosjektet med superbrukere som er lokalisert i samme bygning, har ikke brukerdeltagelsen vært anderledes. Superbrukerne har tilgang til det kjørbare systemet fra sin arbeidsplass for utprøving og testing.

Brukermedvirkning i form av en eller flere brukere som sitter sammen med utviklerne på fulltid har aldri kommet i stand. En årsak mener

jeg ligger i at oppdragsgiver eller brukerne ikke ser seg tjent med å sette av så mye tid. En annen årsak kan ligge i at ingen av prosjektene skulle resultere i systemer som på noen måte kunne true arbeidsplasser eller endre maktfaktorer, og dermed er "ufarlige". En tredje årsak kan være at gruppen ikke er flink nok til å oppfordre, eller ønsker tettere brukermedvirkning selv.

Uansett mengde brukermedvirkning, har ingen brukere selv uttrykt ønske om sterkere deltagelse.

Kompensasjon for mindre brukermedvirkning er hyppige møter, samtidig som det hele tiden er et tilgjengelig kjørbart system som brukerne prøver, tester, og gir tilbakemelding på. Enten i form av dirkete tilbakemelding, eller gjennom oppfølgingsverktøyet.

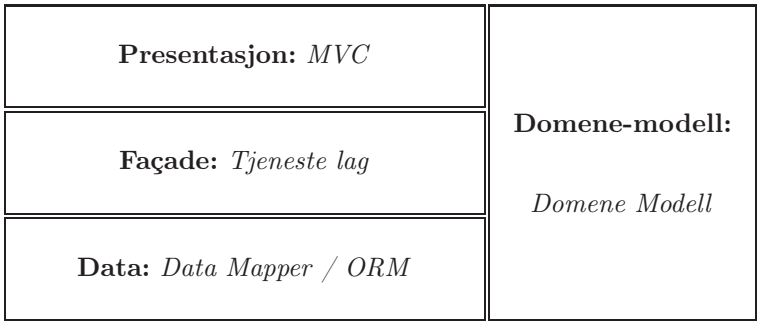
Dokumentasjon Gruppen har hovedsaklig dokumentert gjennom kommentarer i koden, og gjennom bruk av oppfølgingssystemet JIRA, etter at det ble tatt i bruk. Systemene som utvikles har ingen papirbasert brukerdokumentasjon, men gruppen etterstreber intuitive brukergrensesnitt utviklet i samarbeid med brukerne. For to av systemene finnes det en elektronisk brukerhjelp.

Det er etter tre år ikke erfart behov for ytterligere dokumentasjon.

Risikohåndtering Gruppen har ikke foretatt konkrete handlinger vedrørende risikoreduksjon. I ettertid kan man si at etter som alle prosjektene har vært vellykket, har risikoen vært lav.

Referansearkitektur

Referansearkitekturen som benyttes er identisk med den jeg foreslo i seksjon 4.4, og jeg gjengir denne i figur A.1.



Figur A.1: Referanse programvare-arkitektur.

Referanserammeverk

Referanserammeverket som er brukt, er nesten identisk med det jeg presenterte i seksjon 5.5. For ordens skyld gjengir jeg dette her, hvor jeg har byttet MVC-rammeverk til de to som er benyttet av referansegruppen. Se tabell A.2.

Arkitekturlag	Teknologi	Rammeverk
Presentasjon	MVC Java/ HTML	WebWork / SpringMVC
Façade	POJO	Spring
Domene-modell	POJO	Spring
Data	ORM	Hibernate
Alle	Refaktorering og Testing	Eclipse m/JUnit

Tabell A.2: Referanserammeverk

CASE—FERDABALL

Forsvarets ERfarings-DAtaBAse Lesson Learned (FERDABALL). Systemet samler erfaringer fra ulike militære operasjoner, påtegning av kommentarer, og sørger for riktig saksbehandlingsflyt rundt om i Forsvaret. Til slutt ender erfaringen opp som et godkjent erfaringsdokument. Initiell inndata til systemet er et Word-dokument hvor malen er hentet fra systemet, og brukes til å registrere erfaringer (gjerne på en frittstående bærbar maskin). Dokumentet lastes siden opp i systemet, og danner basis for videre saksgang. Opprinnelig var systemet utviklet av et eksternt konsulentselskap, men oppgaven med re-design ble siden gitt til referansegruppen.

Dette var det første prosjektet som anvendte oppgitte metodikkelementer, referansearkitektur, og referanserammeverk. Prosjektet ble utført i samarbeid med et eksternt konsulentselskap (ulikt det som først nyutviklet systemet).

Prosjektet fikk underveis uventede utfordringer når ekstra teknologistøtte skulle inkluderes. Problemet hadde sitt utspring i en proprietær teknologi fra Microsoft, som ble løst med å inkludere en komponent fra OpenOffice som konverterte filer fra Word filformat til XML. Utfordringen lå ikke i problemer med å utvide eksisterende rammeverk, men manifesterte seg ved en relativt høy andel kodelinjer for å løse uforutsette problemer når brukerne avvek fra malen. Jeg mener fortsatt at arkitekturen og referanserammeverket var riktig, men muligens ville en kommersiell konverterings-komponent vært bedre. Dette er ikke forsøkt.

Min største overraskelse når jeg fulgte dette prosjektet, var den høye ressursbruken for brukergrensesnittet. Jeg var vant til 4GL-verktøy som svært enkelt genererte brukergrensesnitt fra datamodellen støttet av noen manuelle justeringer. Nå utgjorde brukergrensesnittet over 50% av utviklingstiden.

Systemet har elektronisk brukerhjelp i form av integrerte HTML-sider i web-applikasjonen.

Prosjektet har gjennomgått to større revisjoner i form av to ekstra oppdrag, som følge av nye behov.

Oppsummering av prosjektet er presentert i tabell A.3 (CVS[5] er utgangspunktet for statistikk, og gir ikke nødvendigvis helt korrekt bilde av situasjonen med hensyn på kodelinjer.) .

[5]CVS—Concurrent Versions System. Finnes på VVV som `http://www.nongnu.org/cvs`.

Beskrivelse	Resultat
Antall kodelinjer	139.614
Prosjektstart	juni 2004
Vedlikehold fra	januar 2005
Siste endring	desember 2006
Totalt antall utviklere	12
Max. samtidige deltagere	6
Større endringer sent i utviklingsløpet?	Ja
Store ekstrakostnader ved sene endringer?	Tilsynelatende ikke

Tabell A.3: CASE—FERDABALL

CASE—FOBID

Forsvarets Oversikt over Bestemmelser, Instrukser og Direktiver (FOBID). Systemet håndterer alle dokumenter og gir muligheter for søk og gjenfinning av nåværende gjeldende dokument, eller gjeldende for et tidspunkt tilbake i tid.

Samme type prosjekt som det foregående, men med mindre grad av eksterne utviklingsressurser. Flere endringer og skifte av spesifikasjonene underveis, også sent i utviklingsløpet uten store problemer.

Én større revisjon er foretatt etter første runde utvikling.

Brukerhjelpen skrives av superbruker, og lastes opp i systemet gjennom systemets innebygde funksjonalitet for å laste opp og tilgjengeliggjøre informasjon.

Oppsummering av prosjektet er presentert i tabell A.4.

Beskrivelse	Resultat
Antall kodelinjer	70.266
Prosjektstart	juli 2005
Vedlikehold fra	november 2005
Siste endring	april 2007
Totalt antall utviklere	6
Max. samtidige deltagere	4
Større endringer sent i utviklingsløpet?	Noe
Store ekstrakostnader ved sene endringer?	Tilsynelatende ikke

Tabell A.4: CASE—FOBID

CASE—FODATEL

FOrsvarets DAta og TEleLinjer (FODATEL). Forsvaret informasjonssystem til bruk for bestilling og administrasjon av Forsvarets ulike sambandsmedier.

Dette var et Re-design av et eksisterende EIS, med delvis gjenbruk av databaseskjema og forretningsregler. Samme superbruker som var med første gang, gjorde trolig overføring av kunnskap bedre.

Én utvikler har gjennomført prosjektet, sammen med opp til 4 brukere, hvorav en har deltatt mest aktivt.

Systemet var opprinnelig utviklet i en 2-lags klient-tjener arkitektur. Det var planer om at alle med behov, skulle kunne bestille direkte via systemet,

men dette medførte at svært mange lokasjoner og tilhørende brukere måtte ha eksplisitt tilgang til den tykke klienten fra sin PC. Dette ble stadig utsatt, og elektroniske dokumenter ble i stedet skrevet ut, fylt inn, og fakset til ansvarlig enhet.

Dette systemet er helt uten brukerhjelp, men er laget slik at brukeren følger en flyt som styrer hva som skal gjøres.

Oppsummering av prosjektet er presentert i tabell A.5.

Beskrivelse	**Resultat**
Antall kodelinjer	43.662
Prosjektstart	desember 2005
Vedlikehold fra	juli 2006
Siste endring	april 2007
Totalt antall utviklere	1
Max. samtidige deltagere	2
Større endringer sent i utviklingsløpet?	Noe
Store ekstrakostnader ved sene endringer?	Tilsynelatende ikke

Tabell A.5: CASE—FODATEL

CASE—FILMDB

Forsvarets Mediesenters (FMS) filmdatabase, FILMDB. Her kan alle med tilgang søke fram aktuelle filmer og foreta bestillinger. Separat administrasjonsmodul for FMS.

Dette systemet er et re-design av et tidligere system hvor teknologien var gått ut på dato. Databasen måtte konverteres fra et filformat til en SQL-database. Dette systemet er tilgjengelig via en webleser på Internet, men krever pålogging. Oppsummering av prosjektet er presentert i tabell A.6.

Beskrivelse	**Resultat**
Antall kodelinjer	39.219
Prosjektstart	november 2005
Vedlikehold fra	oktober 2006
Siste endring	mars 2007
Totalt antall utviklere	3
Max. samtidige deltagere	3
Større endringer sent i utviklingsløpet?	Noe
Store ekstrakostnader ved sene endringer?	Tilsynelatende ikke

Tabell A.6: CASE—FILMDB

CASE-Rapporter

I 2006 ble alle PC-klientene i Forsvaret oppgradert. Som én av konsekvensene ble kjøre-miljøet for rapporter, skrevet for Oracle Reports, ikke oppgradert grunnet høye kostnader.

Behovet for rapportering var fortsatt til stede. En mulig løsning som ble vurdert, var å legge de eksisterende rapportene til en terminal-tjener, som kunne nås fra PC-klientene. Løsningen som ble valgt var den billigste. Den besto i å re-designe rapportløsningene med utgangspunkt i referansearkitekturen og referanserammeverket. Databaseskjema ble holdt uendret, og en alternativ åpen kildekode rapportgenerator[6] ble integrert inn i referanserammeverket.

En til to utviklere har deltatt, hvor jeg selv var med å satte opp utvidelser av referanserammeverket, samt en mal for parametervinduene. Alle rapportdefinisjoner er utført av en utvikler som tilhører en annen organisasjonsenhet enn referansegruppen.

Denne case'en er en samling av flere miniprosjekter. Ved oppstarten av første rapportprosjekt, manglet vi konkret teknologi for rapporteringen. Vi brukte omlag to personuker på det første miniprosjektet. For det andre miniprosjektet, gikk tiden ned med omlag 50%. De etterfølgende miniprosjektene har resultert i ytterligere reduksjon i tidsforbruk.

Miniprosjektene er ikke direkte sammenlignbare, grunnet ulike behov for antall rapporter, og kompleksiteten for å skrive om rapportdefinisjonen er forskjellig. Erfaring tilsier at kompleksiteten reduseres gjennom bedre læring av rapportrammeverket, samt praksis.

En sentral applikasjons-tjener kjører de ulike rapportapplikasjonene som nås via en standard webleser, hvor tilhørende skjermbilder og parametervinduer er tilgjengelige. Rapportene presenteres i samme webleser i ulike formater som HTML, PDF, og Microsoft Excel.

CASE—Adressekatalog

Et av prosjektene var en adressekatalog for Forsvaret. I første omgang satt opp tilsvarende forrige case. Denne skulle gi alle brukerne en alltid oppdatert PDF som kunne nås fra hvilken som helst PC i Forsvaret. I tillegg var det behov for søk etter spesifikke organisasjonsenheter med svar tilbake på tilhørende adresseropplysninger.

Etter hvert dukket også behovet opp for å tilby en webtjeneste mot systemet. Årsaken for behovet kom opp i forbindelse med er workshop Forsvaret hadde, og hvor det var ønskelig med en demonstrator som kunne tilby en webtjeneste mot et eksisterende system.

Behovet ble imøtekommet ved å inkludere et åpen kildekode SOAP-basert rammeverk[7] inn i referanserammeverket, og derved tilby samme tjeneste som allerede var tilgjengelig fra en webleser, i form at en standard webtjeneste (WS).

Oppsummeringer

Wayne Stevens[8], hadde følgende test på en metodikk når noen ønsket å innføre en uprøvd metodikk i metodikk-biblioteket: "Prøv den på et prosjekt og fortell etterpå hvordan det gikk"[24, ss. 144].

[6] JasperReports—`http://jasperforge.org/sf/projects/jasperreports`

[7] XFire—`http://xfire.codehaus.org`

[8] Wayne Stevens—designer av "IBM Consulting Group's Information Engineering methodology" på 90-tallet

Referansegruppen har oppnådd gode resultater ved bruk av lette og smidige metodikkelementer i kombinasjon med den omtalte referansearkitektur implementert som et referanserammeverk. Jeg har selv gode erfaringer i forbindelse med de to siste case'ne hvor jeg har deltatt. Etter min mening er referanserammeverket svært sentralt for at metodevekten kan holdes på et tilstrekkelig og lavt nivå. Det kan argumenteres mot at det ikke er anvendt noen konkret metodikk, men metodikkelementer er tilstede.

I tillegg til de nevnte metodikkelementer er prototyping benyttet, både som en tidlig "mock-up" i form av grafisk HTML grensesnitt, og i form av kjørbare prototyper som gjennom iterasjoner til slutt utgjør det ferdige systemet. Når "mock-up" er godkjent av brukerrepresentantene, benyttes HTML-koden i det videre arbeidet.

Generelt er det gode tilbakemeldinger på tilfredsheten med systemene fra oppdragsgivere og brukere.

Tillegg B

Definisjoner og Akronymer

Definisjoner og begreper som omtales i oppgaven men som ikke er behandlet i egne seksjoner er gitt i B.1. Akronymer er listet i B.2.

B.1 Definisjoner

System Jeg bruker følgende definisjon[72, ss. 24] på et system:

> Et system er en samling av innbyrdes relaterte komponenter som spiller sammen for å oppnå et formål.

Applikasjon Se system.

Programvare Se system.

Enterprise Organisasjon, bedrift, firma, virksomhet, et cetera.

Enterprise Informasjon System EIS, definerer jeg som viktige informasjonssystemer som forutsettes for å drive en enterprise. De viktigste og mest strategiske vil typisk variere fra enterprise til enterprise. Andre vil kunne være delvis like mellom ulike enterpriser. En forutsetning for denne kategori systemer er at de på en eller annen måte har nettverkstilknytning—de kan ikke være autonome. Trenden er også at de skal kunne være tilgjengelig over Internet, enten gjennom sikker pålogging og kryptering av trafikken, eller via "tunneling" over Internet mot bedriftens intranett. Eksempler på slike systemer er ledelsesinformasjon-systemer, kunnskaps-sytemer, erfaring-systemer, salgs og kundeoppfølging-systemer, logistikk-systemer, personell-systemer, økonomi-systemer, lønn-systemer, arkiv-systemer. Viktige attributter for denne type systemer er sikkerhet, autentisering, autorisering, transaksjonser.

En annen definisjon på EIS er i form av informasjonssystemer for en enterprise, som vanligvis involverer datasentriske, hvor mange samtidige brukere, mange ulike skjermbilder for ulike behov, og hvor det ofte er behov for integrasjon med andre systemer[32, ss 3-4]. EIS er av avgjørende betydning for virksomhetens kjerne-områder[50, ss. 367].

Eksempler på systemer som ikke omfattes; tekstbehandlere, styringssystemer, spill, og operativsystemer.

Legasy systemer er gamle programvaresystemer som har nytteverdi for organisasjonen, men som henger etter rent teknologisk.

Komponent En komponent er en programvaredel som er utenfor opphavsmennenes kontroll i det den brukes lokalt i en programvare et annet sted[34].

Tjeneste Tjeneste ligner en komponent, men med den forskjell at her finnes ikke tjenesten lokalt i en programvare, men er tilgjengelig som en ekstern komponent med aksess fra utsiden. utsiden.

J2EE Java 2 Platform Enterprise Edition (J2EE) med de sentrale tjenestene for oppgaven beskrives først. Fra og med EJB-tjenesten, som erstattes av en lettvekter-container, er tjenestene ikke direkte relevante for oppgaven.

- **JDBC** "Java DataBase Connectivity" er et Java-grensesnitt mot relasjonsdatabaser[42]. JDBC, riktignok anvendt gjennom egne rammeverk, er den grunnleggende Javateknologien mot den videreførte arven i oppgaven.
- **JMS** "Java Messages Service" gir asynkron kommunikasjon via meldingsutveksling[42].
- **Servlets** "Java Servlets" er komponenter som utplasseres på en webserver[42]. og anvendes hovedsaklig av web rammeverkene, eller genereres fra JSP'ene. Innen et rammeverk som gjør bruk av MVC-patternet, kan en servlet være controller-komponenten.
- **JSP** "Java Server Pages" er en overbygning over Servlets, som sin tur genererer Servlets for dynamisk generering av websider[42]. Innen et rammeverk som gjør bruk av MVC-mønsteret, kan en JSP være view-komponenten.
- **JTA** "Java Transaction API" er en viktig komponent innenfor J2EE-stakken, i form av en transaksjonstjeneste[42] som brukes blant annet til orkestrering mot flere samtidige databaser. JTA kommer ikke til anvendelse hvis systemet kun skal forholde seg til én database, hvilket er tilfelle for oppgaven.
- **JCA** "Java Connection Architecture" en en annen viktig komponent innefor J2EE-stakken. Teknologien muliggjør en utvidelse av Data Aksess Laget 4.2.6.4, til et Enterprise Informasjons System (EIS) lag, som i tillegg til en eller flere databaser, også inkluderer tilgang mot Legasy systemer. JCA muliggjør integrasjon mot ERP[1] systemer og andre backend systemer slik spesifikasjonen opprinnelig var tenkt[53].
- **XML** "eXtensible Markup Language" anvendes av J2EE-systemer, eller frittstående Java-systemer og rammeverk til konfigurasjon. Innen Java-verdenen er XML en Java XML API mot en XML-tolker[42].
- **JavaMail** er Java API for elektronisk post[42].
- **JAF** "JavaBeans Activation Framework" gir automatisk aktivering av Java-komponenter for å håndtere ulike objekter[42].

[1] ERP—Enterprise Resource Programs

J2SE "Java2 Platform Standard Edition" inkluderer kjernespråk-tjenestene[42] og gir oss også JVM[2] som gjør J2EE applikasjonen maskin- og operativsystem uavhengig.

EJB "Enterprise JavaBeans" som er en arkitektur for å bygge gjenbrukbare tjenerkomponenter[42]. I stedet for disse bønnene, fokuserer lettvekter containere på POJO'er. De enterprise tjenester som EJB-containeren gir, kan i stedet gis av lettvekt-containeren gjennom å "sprøyte inn" nødvendig støtte ved behov.

JNDI "Java Naming and Directory Interface" som lokaliserer ressurser over et nettverk[42]. Dette er viktig for distribuerte objekter, men er overflødig når objektene kjøres lokalt i en lettvekts container.

RMI-IIOP "Java remote Method Invocation over the Internet Inter ORB-Protocol" er fjernmetode-kall mellom virtuelle Javamaskiner (JVM)[42].

Java IDL "Java Interface Definition Language" er en Java CORBA ORB som implementerer et subsett av CORBA-spesifikasjonen[42].

Andre tjenester for virksomheten er slike som lastbalansering, mellomlagring, "failover", som er skilt ut av spesifikasjonen og overlatt til mellomvareleverandørene å tilby[42]. Dette passer lettvekts-containere bra, ved at de kan inkludere slike tjenester ved behov.

MDA OMG's[3] Modell Drevet Arkitektur(MDA[4]), skal gi en åpen og selgernøytral tilnærming til de utfordringer som er knyttet til endringer innen forretningsområder og teknologi.

Med utgangspunkt i OMG's etablerte standarder skiller MDA forretnings- og applikasjons-funksjoner fra den underliggende plattform-teknologien. Plattformuavhengige modeller for en applikasjon bygges ved hjelp av UML (se B.2).

B.2 Akronymer

I tillegg til fotnoter for mange akronymer og forkortelser, vil jeg her samle alle for eventuelle referanser og kompletthet.

UML "Unified Modelling Language" fra OMG[64].

XML *eXtensible Markup Language*, er en kryss-plattform, utvidbar, og tekstbasert standard for å representere data. Man kan definere egne "tagger" for data'ene i spesielle XML-dokumenter, og deretter utveksle dokumentene med forvissning om at motparten skjønner semantikken. XML "stylesheets" brukes for å håndtere visningen av data'ene.

HTML "HyperText Markup Language" er verdensvevens dominerende språk for visning ("rendering") av web-sider.

[2] JVM—Java Virtuell maskin
[3] Object Management Group[64], mest kjent for UML-spesifikasjonene
[4] MDA—"Model Driven Architecture". Finnes på VVV som `http://www.omg.org/mda`

VVV forkortelse for VerdensVeVen i.h.h.t. Norsk Språkråd. Samme som "World Wide Web" (WWW).

URL "Universial Resource Locator", ofte forbundet med en adresse på VVV.

WS Web Tjenester ("Web Services").

SOAP "Simple Object Access Protocol" eller "Service Oriented Architecture Protocol"

WSDL "Web Services Description Language"

UDDI "Universal Description, Discovery and Integration"

BPEL "Business Process Execution Language"

J2SE "Java2 Standard Edition".

J2EE "Java2 Enterprise Edition".

IDE Integrert utviklingsmiljø ("Integrated Development Environment").

TDD Test Drevet Utvikling ("Test Driven Development").

TFD Test Først Utvikling ("Test First Development").

RAD Hurtig applikasjons utvikling ("Rapid Application Development").

CASE "Computer Aided Systems Engineering" eller verktøy for 4. generasjons språk (4GL).

SQL "Structured Query Language", et standard språk for relasjonsdatabaser som gir mulighet for spørring, oppdatering, endring, og vedlikehold av innhold og metadata i databasen.

RUP "Rational Unified Proses" - kommersialisering av UP. Eies nå av IBM.

UP "Unified Proses".

CMM "Capability Maturity Model".

API Applikasjon Programmerings grensesnitt ("Application Programming Interface".

ASP Tjenestetilbyder ("Application Service Provider").

TCO "Total Cost of Ownership". Totale kostnader for eierskap. I oppgavens kontekst gjelder dette eierskapet for systemer.

Bibliografi

[1] Christopher Alexander, Sara Ishakawa og Murray Silverstein. *A Pattern Language*. Oxford University Press, New York, 1977.

[2] Deepak Alur, John Crupi og Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 2001. ISBN 0-13-064884-1.

[3] Budde et al. *Prototyping - An Approach to Evolutionary Systems Development*, kapittel 1 og 4, side 6–9 og 33–48. Springer Verlag, 1991.

[4] Terry Bollinger et al. Use of Free and Open Source Software (FOSS) in the U.S. Department of Defense. Rapport, The MITRE Corporation, Januar 2003. Versjon 1.2.04. Finnes på VVV som `www.mitre.org/work/sepo/library/SoftwareEngineering/OpenSourceSoftware/%dodfoss.pdf`.

[5] Scott W. Ambler. Data modeling 101.

[6] Scott W. Ambler. Mapping objects to relational databases.

[7] Scott Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. John Wiley & Sons, Inc., New York, NY, USA, 2003. ISBN 0471202835.

[8] Niels Erik Andersen, Finn Kensing, Monika Lassen, Jette Lundin, Lars Mathiassen, Andreas Munk-Madsen og Pål Sørgaard. *Professionel Systemudvikling - Erfaringer, muligheder og handlinger*. Teknisk forlag as, Århus, 1986.

[9] Aritma.com. The Simplest Thing that Could Possibly Work. Finnes på VVV som `http://www.artima.com/intv/simplest.html`.

[10] Jørgen Bandsler. Systems development research in scandinavia. *Scandinavian Journal of Information Systems*, 0(1):3–20, 1989.

[11] Christian Bauer og Gavin King. *Hibernate in Action*. Manning Publications Co, 2005.

[12] Kent Beck, Alistair Cockburn og Martin Fowler et al. Manifesto for agile software development. Finnes på VVV som `http://www.agilemanifesto.org/`.

[13] Kent Beck. *extreme Programming explained*. Addison-Wesley, 2003.

[14] Kent Beck. *Test-Driven Development By Example*. Addison-Wesley, 2004.

[15] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer: Innovative Technology for Computer professionals*, 0(6):61–72, 1988.

[16] Barry Boehm og Richard Turner. Balancing Agility and Discipline: Evaluating and Integrating Agile and Plan-Driven Methods. I *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, side 718–719, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.

[17] Kristin Braa. Priority Workshops: Springboard for User partisipation in Redesign Acivities. I *Proceedings of the Conferance on organizational Computing Systems COOCS 1995*, side 246–255, 1995.

[18] Stewart Brand og Penguin U. S. A. Paper. *How Buildings Learn: What Happens After They're Built*. Penguin Books, October 1995. ISBN 0140139966.

[19] Tone Bratteteig. *Making change. Dealing with relations between design and use*. Doktorgradsoppgave, University of Oslo, Department of Informatics, Faculty of Mathematics and Natural Sciences, Norway, 9 2003.

[20] Frederick P. Brooks Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, april 1987.

[21] William J. Brown, Raphael C. Malveau, III Hays W. McCormick og Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998. ISBN 0-471-19713-0.

[22] S. Bråten. *Dialogens vilkår i datasamfunnet*, kapittel Asymmetrisk samtale og selvstendig syn: Opphevelse av modellmonopol, side 165–183. Universitetsforlaget, 1983.

[23] C. Ciborra. *Teams, Markets and Systems*. Cambridge University Press, Cambridge, 1998.

[24] Alistair Cockburn. *Agile Software Development*. Addison-Wesley, 2002.

[25] Bill Curtis, Herb Krasner og Neil Iscoe. A field study of the software design process for large systems. *Commun. ACM*, 31(11):1268–1287, 1988. ISSN 0001-0782.

[26] Pelle Ehn. *Evlusjonær systemutviklingsmodell*. ukjent, 1990.

[27] Pelle Ehn. *Participatory Design: Principles and Practices*, kapittel Ch. 4: Scandinavian Design: On Participation and Skill, side 41–47. Lawrence Erlbaum Associates, 1993.

[28] Eric Evans. Domain driven design: Tackling complexity in the heart of business software, august 09 2002.

[29] C. Floyd, F.-M. reisin og G. Schmidt. STEPS to software development with users. I C. Ghezzi og J. A. McDermid, redaktører, *ESEC'89 2nd European Software Engineering Conference*, University of Warwick, Coventry, United Kingdom, september 1989. Springer.

[30] Fornyings- og Administrasjonsdepartementet. St.meld. nr. 17, Eit informasjonssamfunn for alle, 12 2006. Finnes på VVV som `http://www.odin.dep.no/fad/norsk/dok/regpubl/stmeld/071001-040005/dok-b%n.html`.

[31] Martin Fowler, Kent Beck, John Brant, William Opdyke og Don Roberts. *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley, 1999.

[32] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[33] Martin Fowler. AnemicDomainModel. Finnes på VVV som `http://www.martinfowler.com/bliki/AnemicDomainModel.html`.

[34] Martin Fowler. Inversion of control containers and the dependency injection pattern. Finnes på VVV som `http://martinfowler.com/articles/injection.html`.

[35] Martin Fowler. The new methodology. Finnes på VVV som `http://www.martinfowler.com/articles/newMethodology.html`.

[36] Spring Framework. Spring framework. Finnes på VVV som `http://www.springframework.org/`.

[37] Tommy Gagnes, Anders Eggen, Ole-Erik Hedenstad, Rolf Rasmussen og Geir Sletten. Operative beslutningsstøttesystemer - fremtid NbF. Rapport 03584, Forsvarets Forskningsinstitutt, 2005.

[38] Erich Gamma, Richard Helm, Ralph Johnson og John Vlissides. *Design Patterns*. Addison-Wesley Professional; 1st edition (January 15, 1995), 1995.

[39] Jesse James Garrett. Ajax: A New Approach to Web Applications. Finnes på VVV som `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[40] Bruce A. Tate & Justin Gehtland. *Better, Faster, Lighter Java*. O'Reilly Media, Inc., 2004.

[41] Joan Greenbaum og Morten Kyng. *Design at Work: Cooperative Design of Computer Systems*, kapittel Introduction: Situated Design. Lawrence Erlbaum Associates, 1991.

[42] Hilde Hafnor, Brian Elveseter, Kurt A Veum og Kjell Viken. Arkitekturer for kommando og kontroll informasjonssystemer. Rapport 04582, Forsvarets Forskningsinstitutt, 2000.

[43] Martin Hancox og Ray Hackney. IT outsourcing: frameworks for conceptualizing practice and perception. *Inf. Syst. J*, 10(3):217–238, 2000.

[44] The Roadmap is a project of the Berkman Center for Internet & Society at Harvard Law School. The roadmap for open ict ecosystems: a user-friendly guide for policymakers and technologists offerings tools for understanding, creating, and sustaining open information and communication technologies ecosystems. Finnes på VVV som `http://cyber.law.harvard.edu/epolicy`, 2005.

[45] Berit Haugen, Arve Klev, Andreas Nergaard, Lene Synnestvedt og Lars Aarvik. MEDAKIS-prosjektet, Historien. Studentrapport for kurset IN364, Juni 1999. Tilgjengelig fra informatikk-biblioteket, Universitetet i Oslo.

[46] Jim Highsmith. What is agile software development. Rapport, CrossTalk, Oktober 2002. Finnes på VVV som `http://www.stsc.hill.af.mil/crosstalk/2002/10/highsmith.html`.

[47] Jim Highsmith. Objections to Agile Development. Finnes på VVV som `www.cutter.com/research/freestuff/AgileObjections.pdf`.

[48] Luke Hohmann. *Journey of the Software Professional*. Prentice Hall, 1997.

[49] IEEE Std 610.12-1990. IEEE standard glossary of software engineering terminology. Rapport, The Institute of Electical and Electronics Engineers, Inc., 1990.

[50] Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg og Colin Sampaleanu. *Professional Java Development with the Spring Framework*. Wiley Publishing, Inc., 2005.

[51] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack og Thierry Templier. Spring - java/j2ee application framework reference documentation. Finnes på VVV som `http://static.springframework.org/spring/docs/current/reference`.

[52] Rod Johnson. *expert one-on-one: J2EE Design and Development*. Wrox Press, 2002.

[53] Rod Johnson. *J2EE Development without EJB*. Wiley Publishing, Inc., 2004.

[54] Rod Johnson. Introduction to the spring framework. Finnes på VVV som `http://www.theserverside.com/articles/article.tss?l=SpringFramework`.

[55] Rob Kidd. Soa blueprints: An executive summary. Finnes på VVV som `http://www.sageza.com/available/Snapshot/SN7-27-04SOABlueprints.pdf`.

[56] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003. ISBN 1930110936.

[57] Lars Mathiassen og Thomas Seewaldt. Mixed approaches, mai 17 1999 (1995).

[58] Lars Mathiassen, Thomas Seewalt og Jan Stage. Prototyping and specifying: Principles and practices of a mixed approach. *Scandinavian Journal of Information Systems*, 7(1):55–72, 1995.

[59] Lars Mathiassen. The principle of limited reduction, mai 17 1999 (1992).

[60] NATO Command, Control and Consultancy Agency (NC3A). NATO Network Enabled Capability Feasibility Study, 2005. Bak studien står 12 NATO nasjoner; Belgia, Canada, Danmark, Tyskland, Frankrike, Italia, Nederland, Norge, Spania, Tyrkia, Storbritannia and USA. Studien gikk over 18 måneder og ble godkjent av nasjonene desember 2005. Finnes på VVV som `http://www.teleplan.no/artikler/NNEC_FS.pdf`.

[61] Peter Naur. Programming as theory building. *Programming as Theory Building, reprinted i Computing: A Human Activity (1992)*, 1985.

[62] NSM. Nasjonal sikkerhetsmyndighet. Finnes på VVV som `http://www.nsm.stat.no`.

[63] ODMG. Object Data Management Group. Finnes på VVV som `http://www.odmg.org`.

[64] OMG.org. Object Management Group. Finnes på VVV som `http://www.omg.org`.

[65] Richard Pawson. *Naked objects*. Doktorgradsoppgave, University of Dublin, Trinity College, Dublin, June 2004.

[66] Trygve Reenskaug with P. Wold og O.A. Lehne. Working with objects. the ooram software engineering method, 2001.

[67] Quantitative Software Management (QSM). Quantitative software management (qsm). Finnes på VVV som `http://www.qsm.com/risk_02.html`.

[68] Matt Raible. *Spring Live*. www.sourcebeat.com, 2005.

[69] W3C Recommendation. Soap version 1.2 part 1: Messaging framework (second edition). Finnes på VVV som `http://www.w3.org/TR/soap12-part1`.

[70] Trygve M. H. Reenskaug. MVC. Han skrev en artikken som finnes på VVV som `http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html`, 1978.

[71] Defence Research og Development Canada. Free and open source software: A viable cost-saving opportunity for the department of national defence and the canadian forces. Rapport 17, issues - In Defence Science & Technology, Juli 2004. Finnes på VVV som `www.drdc-rddc.gc.ca/publications/issues/issues17_e.asp`.

[72] Ian Sommerville. *Software Engineering (5th edition)*. Addison Wesley, 1995.

[73] Standish-gruppen. The chaos report. Rapport, Standish-gruppen, 1994.

[74] Standish-gruppen. Extreme chaos report. Rapport, Standish-gruppen, 2001.

[75] Standish-gruppen. First quarter 2004, spotlight on open source. Rapport, Standish-gruppen, 2004.

[76] Bruce A. Tate og Justin Gehtland. *Spring: A Developer's Notebook*. O'Reilly Media, Inc., 2005.

[77] Frederick Winslow Taylor. *The Principles of Scientific Management*. Nr taylor1911 i History of Economic Thought Books. McMaster University Archive for the History of Economic Thought, 1911. available at http://ideas.repec.org/b/hay/hetboo/taylor1911.html.

[78] Inge Tjøstheim, Bjørn Innset, Johan Haarberg og Harald Håvoll. Introduksjon til nettverksbasert forsvar, 2 2001. Finnes på VVV som `http://www.mil.no/multimedia/archive/00010/Introduksjon_til_NbF_10138a.%pdf`.

[79] W. M. Ulrich. The evolutionary growth of software engineering and the decade ahead. *American Programmer*, side 14–20, October 1990.

[80] Werner Vogel. All things distributed, en weblog om web services. Finnes på VVV som `weblogs.cs.cornell.edu/AllThingsDistributed/archives/000343.html`.

[81] W3C. Web services activity. Finnes på VVV som `http://www.w3c.org/2002/ws`.

[82] W3C. Web services architecure. Finnes på VVV som `http://www.w3.org/TR/ws-arch`.

[83] WIKIPEDIA. The Free Encyclopedia. Finnes på VVV som `http://www.wikipedia.org`.

[84] Ann Wollrath, Geoff Wyant, Jim Waldo og Sam Kendall. A note on distributed computing, mars 26 1994.

[85] E. Yourdon. RE-3, part 1: Re-engineering, restructuring, and reverse engineering. *American Programmer*, 2(4):3–10, April 1989.