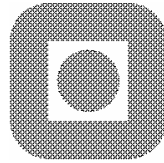


NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET
FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG
ELEKTROTEKNIKK



HOVEDOPPGAVE

Kandidatenes navn: Øivind Mollan og Ole Johan Lefstad

Fag: Systemutvikling

Oppgavens tittel (norsk):

”En undersøkelse av Ekstremprogrammering for å se på effekten i et utviklingsprosjekt”

Oppgavens tittel (engelsk):

”A study of Extreme Programming to examine the effects in a development project”

Oppgavens tekst:

Bruk Ekstremprogrammering i et utviklingsprosjekt. Utfør GQM i starten av prosjektet – sammen med personell fra IDI. Data – både kvantitative og kvalitative – skal samles underveis i utviklingsprosjektet. I tillegg skal utviklerne logge det som skjer gjennom prosjektet. Når utviklingen er ferdig, skal dataene analyseres. Loggen skal brukes til å sammenlikne resultater slik som produktivitet, brukbarhet og kvalitet på kode.

Oppgaven gitt: 21.1.2002

Besvarelsen leveres innen: 17.6.2002

Besvarelsen levert: 14.6.2002

Utført ved: IDI/SINTEF

Veileder: Tor Stålhane

Trondheim,

Faglærer

Sammendrag

Ekstremprogrammering (XP) er en lettvektsmetodikk som har vært mye omtalt i det siste. Systemutviklingsmetoden skiller seg fra andre metoder ved for eksempel at all produksjonskode skrives av utviklere i par, at kravspesifikasjon erstattes med korte historier og muntlig kommunikasjon og at den bruker lite dokumentasjon. I tillegg er metoden fleksibel fordi den tar høyde for at kunden kan endre krav underveis.

Vi ønsket å finne ut om det ble noen forskjell på XP og ”tradisjonelle” utviklingsmetoder i forhold til produktivitet og produktkvalitet. Produktkvalitet vil her si brukbarhet og kvalitet på kode. I tillegg ønsket vi å finne en beskrivelse av *vår* måte å utføre XP på i et utviklingsprosjektet.

Vi har gjennomført et case-studie der vi har implementert en elektronisk prosessguide ved hjelp av XP. Systemet skal være en del av Intranettet hos Bravida Geomatikk. Vi har sammenliknet produktivitet og produktkvalitet med det en utvikler hos Sintef Tele og Data hadde gjort på forhånd. Utvikleren hos Sintef Tele og Data utviklet ved hjelp av prototyping. Våre resultater antyder at XP er mer ressurskrevende enn prototyping og at kvaliteten på koden blir bedre.

Forord

Denne diplomoppgaven ble utført ved Norges Teknisk-Naturvitenskapelige Universitet i perioden fra januar til juni 2002. Oppgaven ble gjennomført som et samarbeid mellom NTNU og Sintef Tele og Data. Målgruppen for denne rapporten er prosjektledere og systemutviklere som vurderer å benytte XP i et utviklingsprosjekt. Ordforklaringer finnes på side 61.

Vi ønsker å takke veileder Tor Stålhane og Nils Brede Moe ved Sintef Tele og Data for verdifull tilbakemelding. Vi vil også takke student Håvard Julsrud Hauge for samarbeidet underveis.

Trondheim, 14. juni 2002

Øivind Mollan

Ole Johan Lefstad

Innholdsfortegnelse

1	INNLEDNING	1
1.1	Bakgrunn.....	1
1.2	Oversikt over rapporten	1
2	FORSKNINGSMETODE	3
2.1	Forstudie	3
2.2	Utviklingsprosjekt med XP	3
2.3	Analyse av resultat	3
3	XP – STATE-OF-THE-ART	4
3.1	Innledning	4
3.2	XP sammenliknet med andre metoder	4
3.3	Kostnadsutviklingen i et prosjekt	5
3.4	Fire sentrale variabler	6
3.5	Fire sentrale verdier	6
3.6	12 praksiser	8
3.6.1	Koblinger mellom praksisene.....	10
3.7	Innredning av XP-kontor	11
3.8	Ansvarsområder	12
3.9	XP er ikke løsningen på alle problemer	12
3.10	Fordeler og ulemper med XP	13
4	HOVEDDEL	15
4.1	Bakgrunn for prosjektet	15
4.1.1	Hva var laget på forhånd?	15
4.1.2	Beskrivelse av utviklingen	15
4.2	Hva har vi gjort?	16
4.2.1	Beskrivelse av systemet	16
4.2.1.1	Utgivelse 1.....	16
4.2.1.2	Utgivelse 2.....	18
4.2.2	Beskrivelse av utviklingen	20

4.3	Eget bidrag til de 12 praksisene	29
4.3.1	Bakgrunn	29
4.3.2	Vår måte å utføre XP på	31
4.4	Resultater	38
4.4.1	Vurdering av det som var laget på forhånd	38
4.4.1.1	Produktivitet	38
4.4.1.2	Vurdering av kode og arkitektur	38
4.4.1.3	Vurdering av brukbarheten til systemet	39
4.4.2	Vurdering av det vi har laget	40
4.4.2.1	Produktivitet	40
4.4.2.2	Vurdering av kode og arkitektur	41
4.4.2.3	Vurdering av brukbarheten til systemet	42
4.4.2.4	Testing	43
4.4.3	Sammenlikning av det som var laget på forhånd og vår del	44
4.4.3.1	Produktivitet	44
4.4.3.2	Kode og arkitektur	45
4.4.3.3	Brukbarhet	45
4.4.4	GQM-planen	45
4.4.4.1	Bakgrunn	45
4.4.4.2	Interessante avvik	46
4.4.4.3	Samvariasjoner	51
4.5	Egne erfaringer	51
4.5.1	Egne erfaringer med XP	51
4.5.2	Andre momenter vi ser med XP	52
5	DISKUSJON	53
5.1	Diskusjon av framgangsmåte	53
5.1.1	Bruke XP i hele vår del av prosjektet	53
5.1.2	Halve vår del av prosjektet med XP som utviklingsmetode, halve med en ”tradisjonell” utviklingsmetode	53
5.1.3	Vurdering	53
5.2	Diskusjon av erfaringsbasert forbedringssystem	54
5.3	Diskusjon av resultat	54
5.3.1	Produktivitet	54
5.3.2	Kvalitet på koden	55
5.3.3	Usikkerhet knyttet til GQM-planen	55
5.4	Hvordan sammenfaller resultatene med liknende studier?	55
5.4.1	”Ekstrem Programmering, Praktiske erfaringer” [21]	55
5.4.2	”Strengthening the case for pair programming” [22]	56
5.5	Evaluering av eget arbeid	56
6	KONKLUSJON	57

7	VIDERE ARBEID	58
8	REFERANSER.....	59
9	ORDFORKLARINGER.....	61
10	VEDLEGG	63

Figurliste

Figur 1. Utviklingsmetoder har gradvis gått over til kortere sykluser.	4
Figur 2. Kostnadene forbundet med endringer øker eksponentielt over tid, for eksempel ved bruk av vannfallsmodellen.	5
Figur 3. Kostnadene forbundet med endringer trenger ikke å øke dramatisk over tid.	5
Figur 4. Graden av effektiv kommunikasjon minsker med mindre menneskelig kontakt.	7
Figur 5. Praksisene påvirker hverandre.	10
Figur 6. Oppsummering av grunnlaget for XP fra verdier (innerst) til praksiser (ytterst)	11
Figur 7. Oversikt over prosjekter.	15
Figur 8. Et skjermbilde av første del vises til venstre.	16
Figur 9. Utgivelse 1 er en elektronisk prosessguide.	17
Figur 10. Detaljert informasjon om prosessen "Oppfølging".	18
Figur 11. Utgivelse 2. Brukeren kan selv tilpasse sin prosessmodell.	19
Figur 12. De tolv praksisene på veggen.	20
Figur 13. Utviklerne par-programmerer på en datamaskin.	21
Figur 14. Historier på tavle.	21
Figur 15. En modell av vår erfaringsbaserte forbedringssystem.	22
Figur 16. Erfaringer og forslag til endringer på praksisene.	23
Figur 17. Vi oppsummerer siste iterasjon ved hjelp av KJ.	24
Figur 18. KJ etter iterasjon 1.	25
Figur 19. Utdrag fra dagbok.	26
Figur 20. JUnit med "the green bar".	27
Figur 21. Eksempel på en test som feiler.	28
Figur 22. Oversikt over våre forslag til endringer av de 12 praksisbeskrivelsene.	29
Figur 23. Malen vi laget for å fylle ut historiekort.	32
Figur 24. Et eksempel på en historie som er brutt ned til oppgaver.	33
Figur 25. Et eksempel på en funksjonell test.	36
Figur 26. Detaljer om et prosjekt kan vises eller skjules ved å trykke på "+" eller "-".	39
Figur 27. Utvikleren hos Sintef mente layouten kunne endres noe i dette skjermbildet.	42
Figur 28. Del 1 (det som var laget på forhånd) inneholdt mer HTML- og JSP-kode enn del 2 (det vi laget), men mindre Java- og JavaScript-kode.	44
Figur 29. Avvik i produktiviteten i forhold til det vi trodde på forhånd.	46
Figur 30. Slik fungerte ukes-iterasjoner.	47
Figur 31. Mange forslag til endringer på prosessen i begynnelsen av prosjektet og færre mot slutten.	48
Figur 32. Forslag ble gjennomført gjennom hele utviklingsperioden.	48
Figur 33. Det ble kun en endring på funksjonaliteten gjennom hele utviklingsperioden.	49
Figur 34. Ingen historier endret prioritering.	49
Figur 35. Kunden hadde permisjon fra jobb fra iterasjon 3 til 6.	50
Figur 36. En god gjennomføring av "The planning game" er avhengig av at kunden er tilstede.	51
Figur 37. Kunden hadde permisjon fra arbeid fra iterasjon 3 til 6.	51

Tabelliste

Tabell 1. "The planning game"	31
Tabell 2. Metafor.....	34
Tabell 3. Kodestandard.	34
Tabell 4. Enkel design.....	35
Tabell 5. Testing.....	36
Tabell 6. Par-programmering.....	37
Tabell 7. Antall linjer kode produsert av utvikleren hos Sintef.	38
Tabell 8. Antall linjer kode produsert per utgivelse.....	40
Tabell 9. Totalt antall timeverk fordelt på aktiviteter.	40
Tabell 10. Prosentvis fordeling av timeverk på aktiviteter.	40
Tabell 11. Størrelsesforholdet mellom testkode og produksjonskode ble 1:6.	43
Tabell 12. Størrelsesforholdet ble 2:5 mellom antall tester og antall metoder som ble implementert.....	43
Tabell 13. Antall linjer kode fordelt på programmeringsspråk og utgivelser.	44
Tabell 14. Produktivitet i del 1 og 2.....	54

1 Innledning

1.1 Bakgrunn

Mange IT-bedrifter har opplevd prosjekter som ikke ble avsluttet innenfor tids- og kostnadsrammer. Mange har også erfart at det produktet de leverte ikke var det kunden egentlig ville ha, eller de har opplevd at store deler av arbeidet måtte gjøres om igjen fordi de ikke hadde forstått kundens krav riktig. Disse problemene gjør det lønnsomt for bedrifter å drive forbedringsarbeid [1].

Denne diplomoppgaven er en del av forskningsprosjektet PROFIT. PROFIT står for ”PROcess improvement For the IT industry” og er et samarbeid mellom NTNU, Universitetet i Oslo og SINTEF. 15 bedrifter deltar i prosjektet. PROFIT har fokus på utvikling av metoder og retningslinjer for prosessforbedring og bygger på SPIQ-prosjektet [1].

Hovedområder for PROFIT er:

1. Prosessforbedring ved usikkerhet og endringer
2. Målinger og gjenbruk av erfaringer og kunnskap
3. Prosessforbedring gjennom ny teknologi

I denne diplomoppgaven har vi jobbet innenfor alle hovedområdene til PROFIT.

Utgangspunktet for oppgaven var å se på effekten ved bruk av Ekstremprogrammering i et utviklingsprosjekt. Utviklingsmetoden XP passer godt der det er uklare krav eller der det skjer raske endringer på kravene underveis i prosjektet (område 1). Et delmål med oppgaven var å finne en beskrivelse av vår måte å utføre XP på i dette prosjektet (område 2). I tillegg er XP en ny teknologi (område 3).

Vi syntes XP hørt interessant ut og hadde lyst til å prøve metoden i et utviklingsprosjekt. Forsker Nils Brede Moe ved Sintef tilbød oss å delta i et prosjekt der vi skulle utvikle en del av et system ved hjelp av XP. Han skulle bistå med kontor, maskiner og ekspertise. Første del av systemet var allerede laget av en utvikler ved Sintef. Denne delen ble utviklet med prototyping som utviklingsmetode og vi skulle sammenlikne den med vår del når den ble ferdig implementert. Dette systemet skulle brukes av Bravida Geomatikk og en prototyp av første del var allerede installert. Nils Brede Moe fikk oppgaven med å være kunde for oss under utviklingen. For at vår del skulle utvikles uavhengig av det som allerede var laget, fikk vi ikke tilgang til den eksisterende koden. Vi fikk kun se databasen og noen få skjermbilder. Moe la vekt på de ikke-funksjonelle kravene modifiserbarhet og vedlikeholdbarhet.

1.2 Oversikt over rapporten

Denne rapporten består av ti kapitler.

Kapittel 1, *Innledning*.

Kapittel 2, *Forskningsmetode*. Beskrivelse av prosjektets tre deler: forstudie, utviklingsprosjekt med XP og analyse av resultat.

Kapittel 3, *XP – state-of-the-art*. Her beskrives utviklingsmetoden XP.

Kapittel 4, *Hoveddel*. Kapitlet omhandler bakgrunn for prosjektet og hva vi har gjort. Her presenteres også vårt bidrag til XP, hvilke resultater vi har kommet fram til og de erfaringene vi har fått.

Kapittel 5, *Diskusjon*. Her diskuteres framgangsmåtene vi vurderte, måten vi tilpasset XP på og resultatene. Resultatene våre sammenliknes med resultatene fra liknende studier. Vi diskuterer hva som var bra med måten vi løste oppgaven på og hva kunne vi gjort annerledes? Kapitlet sier også noe om hvordan XP og GQM fungerte sammen.

Kapittel 6, *Konklusjon*.

Kapittel 7, *Videre arbeid*.

Kapittel 8, *Referanser*.

Kapittel 9, *Ordforklaringer*.

Kapittel 10, *Vedlegg*. Programkode, dagbok, praksisbeskrivelser fra iterasjon 1 til 6 og Powerpoint-presentasjoner finnes kun på CD.

2 Forskningsmetode

Diplomoppgaven bestod av disse fasene:

- Forstudie
- Utviklingsprosjekt
- Analyse av resultat

2.1 Forstudie

I denne fasen skaffet vi oss kunnskap om XP og forberedte det videre arbeidet. Vi leste bøker og artikler om XP. I løpet av forstudiet deltok vi på to kurs, et i Trondheim og et i Oslo. Begge kursene var i regi av Sintef og hadde en praktisk vinkling mot XP. I tillegg lærte vi oss å bruke testverktøyet JUnit [2].

Vi laget en risikoanalyse for å avdekke potensielle farer underveis i prosjektet. Hensikten var å finne forebyggende tiltak og krisetiltak. Vi benyttet en grovanalyse [3] for å kartlegge farene. Se vedlegg F.

Vi definerte en GQM-plan (vedlegg A) sammen med Tor Stålhane, Nils Brede Moe og Håvard Julsrud Hauge. GQM er en metode som blir brukt når en skal ta i bruk målingsbasert prosessforbedring. Metoden er målorientert og inkluderer et sett av spørsmål og tilhørende metrikker. Se [1] for en beskrivelse av hvordan en bruker GQM. Vi brukte denne metoden for å se på effekten av XP i et prosjekt. Til dette arbeidet benyttet vi GQM-verktøyet [4] vi hadde laget i SIF8094 Fordypningsemne Systemutvikling. På denne måten hadde alle involverte tilgang til GQM-planen på Internett under hele prosjektet.

2.2 Utviklingsprosjekt med XP

Hoveddelen besto i å utvikle en elektronisk prosessguide. Prosessguiden skulle bli en del av Intranettet til Bravida Geomatikk og utvikles ved hjelp av XP. Gjennom hele utviklingsperioden samlet vi data til å svare på GQM-planen og tolket dem i feedbackmøter etter hver iterasjon (vedlegg C). Vi ga ut to utgivelser av systemet på de syv iterasjonene prosjektet varte. Utgivelsene beskrives i kapittel 4.2.1.

Vi deltok på et kurs med Agile Insight [5] som fokuserte på ”The planning game”. Vi var også med på en ”workshop” sammen med andre diplomstudenter som hadde XP som utgangspunkt for sin oppgave.

2.3 Analyse av resultat

I denne fasen av prosjektet analyserte vi data fra hele utviklingsperioden og prøvde å svare på GQM-planen. I tillegg analyserte vi første del av Intranettet – laget av Sintef – og fikk utvikleren hos Sintef Tele og Data til å vurdere vårt arbeid.

3 XP – state-of-the-art

3.1 Innledning

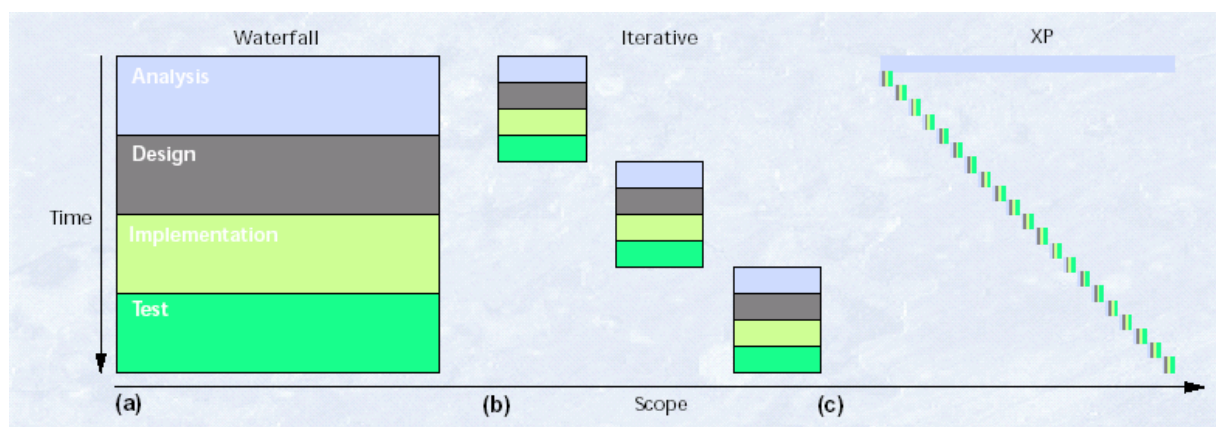
Mange systemutviklingsprosjekter har feilet på grunn av dårlig planlegging og at de ikke har vært mottakelige for endringer underveis. Ekstremprogrammering er en systemutviklingsmetode som blant annet forsøker å løse disse problemene.

Ekstremprogrammering består av kjente og utprøvde teknikker. Nyskapningen består i å samle alle teknikkene, sikre at de blir utført og sikre at de støtter hverandre. Hvorfor *ekstrem* i navnet? Grunnen er at teknikkene praktiseres ekstremt. Et eksempel er at det er vanlig å kvalitetssikre koden ved at en annen programmerer leser gjennom den. I ekstremprogrammering har den som programmerer hele tiden en annen programmerer ved siden av seg som leser koden.

XP er en kontrollert og dokumentert metode for små og mellomstore utviklingsgrupper. Den er laget for prosjekter som kan bestå av grupper på to til ti programmerere. XP passer godt der det er uklare krav eller der det skjer raske endringer av kravene underveis i prosjektet.

3.2 XP sammenliknet med andre metoder

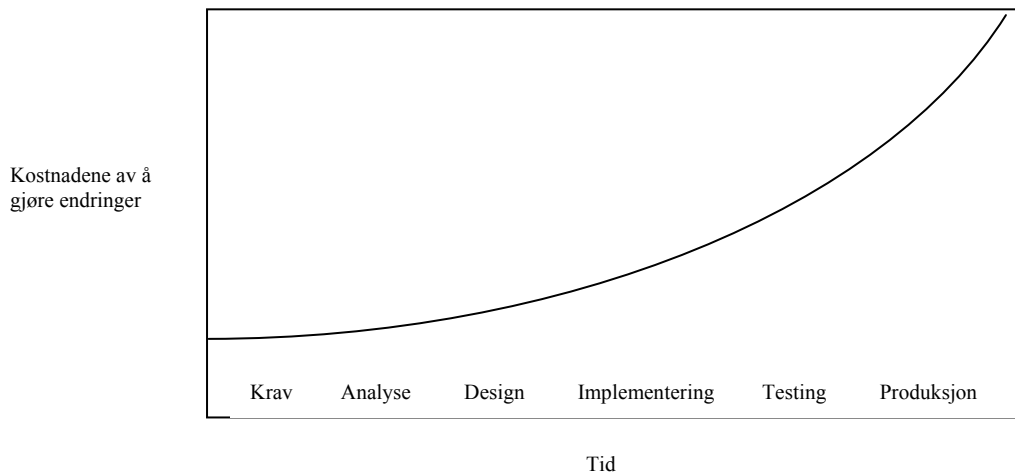
Den tradisjonelle vannfallsmodellen (se figur 1.a) gir ikke rom for endringer underveis i prosjektet. Kunden får gjerne bare *en* mulighet til å fortelle utviklerne hva han vil ha gjennom en kravspesifikasjon. Utviklerne designer systemet, programmerer og tester det. Deretter blir systemet levert med de egenskapene som var spesifisert i kravspesifikasjonen. Vannfallsmodellen tar ikke høyde for at kunden i mange tilfeller er usikker på hva han vil ha i begynnelsen av prosjektet. For at systemet skal tilpasse seg endringer underveis i prosjektet har iterative sykluser blitt innført (se figur 1.b). XP har gått et skritt videre. Forskning har vist at kostnadene for å endre programvare øker dramatisk over tid ved lange sykluser [6]. Derfor har XP innført ekstremt korte sykluser (se figur 1.c). I hver syklus gjennomføres fasene planlegging, design, koding og testing. XP blir på denne måten mottakelig for endringer fra kunden gjennom hele prosjektet.



Figur 1. Utviklingsmetoder har gradvis gått over til kortere sykluser. Vannfallsmodellen (a) har kun en syklus, Spiralmodellen (b) lange sykluser og XP (c) ekstremt korte sykluser. [6]

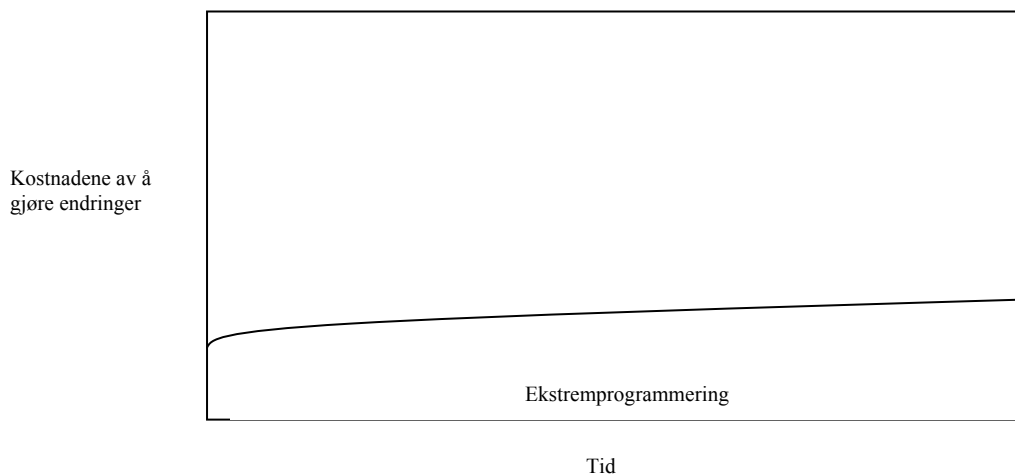
3.3 Kostnadsutviklingen i et prosjekt

Alt i programvare endres; krav, design, bransje, teknologi og størrelsen på utviklingsteamet. En vanlig antakelse i programvareutvikling er at kostnadene ved å endre programvare øker eksponentielt over tid. Se figur 2.



Figur 2. Kostnadene forbundet med endringer øker eksponentielt over tid, for eksempel ved bruk av vannfallsmodellen. [7]

XP ser ikke på endringer av programvare som et problem. Problemet er å takle de endringene som kommer, spesielt sent i prosjektet. Dersom en legger forholdene til rette, trenger ikke kostnadene øke eksponentielt over tid. Se figur 3.



Figur 3. Kostnadene forbundet med endringer trenger ikke å øke dramatisk over tid. [7]

Dette er hovedforutsetningen for XP. Dersom kostnadene av å gjøre endringer øker sakte med tiden, vil en opptre annerledes enn om kostnadene øker eksponentielt. En kan da utsette viktige avgjørelser til sent i prosessen for å ha en større mulighet for at avgjørelsene blir riktige. En kan implementere bare det en må. Dermed lages ikke kode som en senere ikke vil få bruk for. Utviklerne vil endre designet bare dersom det forenkler den eksisterende koden

eller gjør den neste kodebiten enklere. En bratt kostnadskurve for endringer vil gjøre XP umulig å gjennomføre.

Å holde kostnadene ved endringer nede hender ikke av seg selv. I tillegg til korte sykluser, finnes teknologier og praksiser som holder programvaren modifiserbar. Praksisene i XP er retningslinjer i et utviklingsprosjekt. Objektorientering er en anbefalt teknikk. Å sende meldinger mellom objekter er en effektiv måte å bygge inn muligheter for endringer. Det er ikke dermed sagt at en må ha objekter for å ha fleksibilitet i koden, men erfaringen [7] er at kostnadene ved endringer øker brattere uten objekter enn med objekter.

Andre faktorer gjør også at koden blir enklere å modifisere:

- En enkel design. Koden inneholder ingen ekstra designelementer som en *tror* en kan få bruk for i framtiden.
- Bruk av automatiserte tester. Denne typen testing avslører dersom en endrer den eksisterende oppførselen til systemet.
- Trening i å endre systemet ved hjelp av refaktorisering, som er å forbedre kode uten å endre funksjonalitet.

I stedet for å være forsiktig med å ta store avgjørelser tidlig og få avgjørelser senere, kan en i XP ta avgjørelsene raskt, men støtte hver avgjørelse med automatiserte tester.

3.4 Fire sentrale variabler

I XP er fire kontrollvariabler viktige [7]:

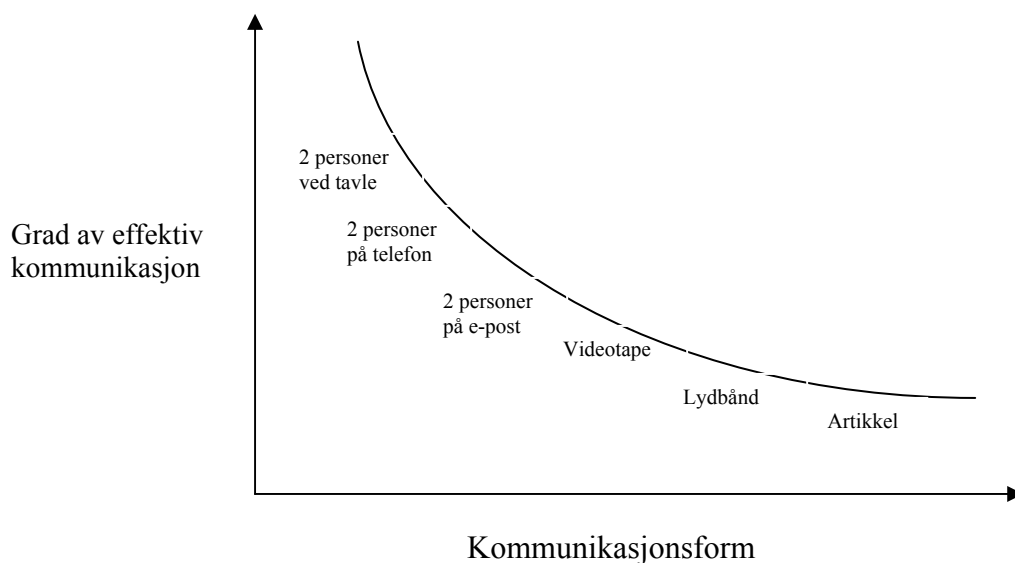
- Kostnader
- Tid
- Kvalitet
- Omfang

Kunden vil gjerne ha kontroll over alle disse variablene. XP sier at kunden kun skal ha muligheten til å velge tre av disse. Utviklerne kontrollerer den siste variabelen. Hensikten er å unngå dårlig kvalitet på produktet. Kunden vil gjerne ha et produkt med mye funksjonalitet til lav pris på kort tid. Dette er vanskelig å få til i praksis og fører gjerne til stressede utviklere og stressede utviklere er ingen suksessfaktor for god kvalitet. Ved at utviklerne bestemmer *en* variabel, får kunden forståelse for at han eksempelvis må kutte omfanget eller sette inn ekstra ressurser for å levere til avtalt tid.

3.5 Fire sentrale verdier

Ekstremprogrammering baserer utviklingsprosessen på verdiene kommunikasjon, enkelhet, tilbakemelding og mot [7][8]. Disse fire verdiene er sentrale når en skal vurdere om en utviklingsprosess er ekstrem.

Den første verdien er *kommunikasjon*. Kommunikasjon er viktig i alle utviklingsmetoder, også XP. Dårlig kommunikasjon kan føre til feil beslutninger og lite framdrift i prosjektet. XP sikrer god kommunikasjon ved å ta i bruk praksiser som ikke kan utføres uten muntlig kommunikasjon. Praksiser som par-programmering, testing og ”The planning game” stimulerer til god kommunikasjon mellom utviklere, kunder og ledelse. Dette er viktig fordi XP som metode inneholder minimalt med formell dokumentasjon.



Figur 4. Graden av effektiv kommunikasjon minsker med mindre menneskelig kontakt. Ifølge [9] er to personer ved tavle den mest effektive måten å kommunisere på.

Hensikten med *enkelhet* er å implementere kun det som er nødvendig for å få ting til å fungere. XP sier at det er bedre å lage noe enkelt i dag og betale litt for det i morgen, enn å lage noe komplisert i dag som ender opp med å ikke bli brukt. Dette er viktig for å holde veksten i kostnadskurven nede. Enkelhet og kommunikasjon henger tett sammen. Mer kommunikasjon fører til at en ser klarere hva som må og hva som ikke må gjøres.

Den tredje verdien er *tilbakemelding*. Både programmerere og kunder skal få rask tilbakemelding på utviklingen i prosjektet. Programmererne skriver enhetstester på logikken i systemet og får tilbakemelding hver gang disse testene kjøres. Kundene skriver nye historier og får tilbakemelding i form av estimater på disse. Konkret tilbakemelding fører til enklere kommunikasjon.

Mot oppmuntrer programmerere og kunder til å prøve ut nye ideer og ta sjanser – gjerne til å gå nye veier i utviklingsprosessen. For å prøve dette må de tre første verdiene kommunikasjon, tilbakemelding og enkelhet være på plass.

Disse fire verdiene er XP sine suksesskriterier, men de er lite konkrete. Ut fra verdiene oppstår noen fundamentale prinsipper:

- Rask feedback
- Anta at ting kan gjøres enkelt
- Inkrementelle endringer
- Ønske endringer velkommen
- Kvalitetsarbeid

Sammen med basisaktivitetene koding, testing, lytting og design, danner dette grunnlaget for XP som utviklingsmetode. Hvordan utviklingsmetoden utføres i praksis, beskrives i neste kapittel.

3.6 12 praksiser

Ekstremprogrammering består av 12 praksiser som skal fungere som retningslinjer i et utviklingsprosjekt – de er ikke regler. Praksisene er kjente, men her praktiseres de ekstremt. Praksisene er beskrevet ut i fra [7][10][11][12].

1. ”The planning game”

Hovedideen bak ”The planning game” er å lage en grovskisse raskt i starten av hver iterasjon og forbedre den etter hvert som ting blir klarere. Viktig for denne fasen er å la kunden bestemme forretningsavgjørelsene og la utviklerne ta de tekniske. Programvareutvikling er alltid en dragkamp mellom det mulige og det ønskelige.

Utviklerne bestemmer:

- Estimerer på hvor lang tid det tar å utvikle en historie
- Konsekvenser av å velge teknologier
- ”Team” organisering
- Risikofaktorer for hver historie
- I hvilken rekkefølge historiene skal utvikles innen en iterasjon

Kunden bestemmer:

- Omfanget på historiene og hvor mye funksjonalitet han krever for hver utgivelse
- Dato for utgivelse
- Prioritet på historiene

2. Korte utgivelser

Utgivelser skal være så korte som mulig og inneholde mest mulig verdi for kunden. Det er enklere å planlegge en til to måneder fram i tid, enn seks måneder eller et år om gangen. En trenger ikke å skrinlegge XP dersom det ikke er mulig å ha hyppige utgivelser. En mulighet er å lage midlertidige utgivelser som bare er tilgjengelig internt.

3. Metafor

En metafor skal hjelpe alle i prosjektet til å forstå systemet som helhet. Metaforen erstatter en del av arkitekturen. Ved å bruke en god metafor vil en få en arkitektur som er enkel å kommunisere og utvide.

4. Enkel design

Hovedideen bak enkel design er å lage det enkleste design som kan fungere. En skal lage det en trenger *når* en trenger det. En enkel design er en design som:

- Kjørere alle testene
- Ikke har duplisert logikk
- Viser hensikten med koden
- Har færrest mulig klasser og metoder

5. Testing

Utviklerne skriver enhetstester og kunden skriver funksjonelle tester. Utviklerne skriver enhetstestene før de skriver produksjonskode. Dette kalles ”test-først” programmering. Kunden skriver de funksjonelle testene etter å ha definert historiene. Resultatet er et program som blir sikrere over tid og som blir mer i stand til å ta i mot endringer. En *må* ikke skrive tester for trivielle metoder. Kunden og utviklerne må bli enige om hva som eventuelt er trivielle metoder.

6. Kontinuerlig integrasjon

Hensikten med kontinuerlig integrasjon er å integrere koden ofte slik at feilene som oppdages ved enhetstesting blir lett sporbare. Koden testes og integreres etter noen timer – maks en dag. En måte å gjøre dette på er å ha en datamaskin dedikert til integrasjon. Når datamaskinen blir ledig, kan et par integrere sin kode med resten av systemet og samtidig kjøre alle de eksisterende testene for å sjekke om deres kode førte til forandringer i systemets oppførsel.

7. Par-programmering

All produksjonskode skrives av to utviklere på én maskin.

Det er to roller for hvert par:

- 1) Den som har tastatur og mus tenker kortsiktig. Det vil si å fokusere på en metode av gangen.
- 2) Den andre tenker mer langsiktig og strategisk:
 - Sjekker feil
 - Vurderer ulike strategier og alternativer
 - Slår opp i kilder som for eksempel bøker, nettsider og diskusjonsgrupper.
 - Sjekker at man holder seg til kodenstandard.

Før utviklere par-programmerer produksjonskode, lager de ofte en ”spike”. En ”spike” er et enkelt program for å utforske en potensiell løsning på et problem. Når en utvikler har skrevet en ”spike” for en historie, finner han en partner til å par-programmere med. På denne måten kan han skifte partner.

8. Felles eierskap

Alle utviklerne i et team har lov til å gjøre endringer på en hver del av koden for å forbedre den. Alle eier koden og alle tar ansvar for hele systemet.

9. Refaktorisering

Refaktorisering er en teknikk for å forbedre kode uten å endre funksjonalitet. Denne teknikken er aktuell når nye egenskaper skal legges til systemet eller når en skal rydde i koden. Utvikleren ser om det er hensiktsmessig å endre det eksisterende programmet for at det skal bli lettere å legge til ny funksjonalitet. Etter å ha lagt til ny funksjonalitet, leter utvikleren etter muligheter for å gjøre programmet enklere. Alle de eksisterende testene må kjøre etterpå.

10. 40-timers uke

40-timers uke er et prinsipp for å unngå overarbeidede ansatte. Ingen kan jobbe 60-timers uker over lengre tid og fremdeles være kreative og fornøyde. Overtid er et symptom på et alvorlig problem i prosjektet. XP-regelen er enkel – du kan ikke arbeide to uker med overtid på rad.

11. Kunde tilstede

Et XP-team trenger å ha en kunde tilgjengelig for å

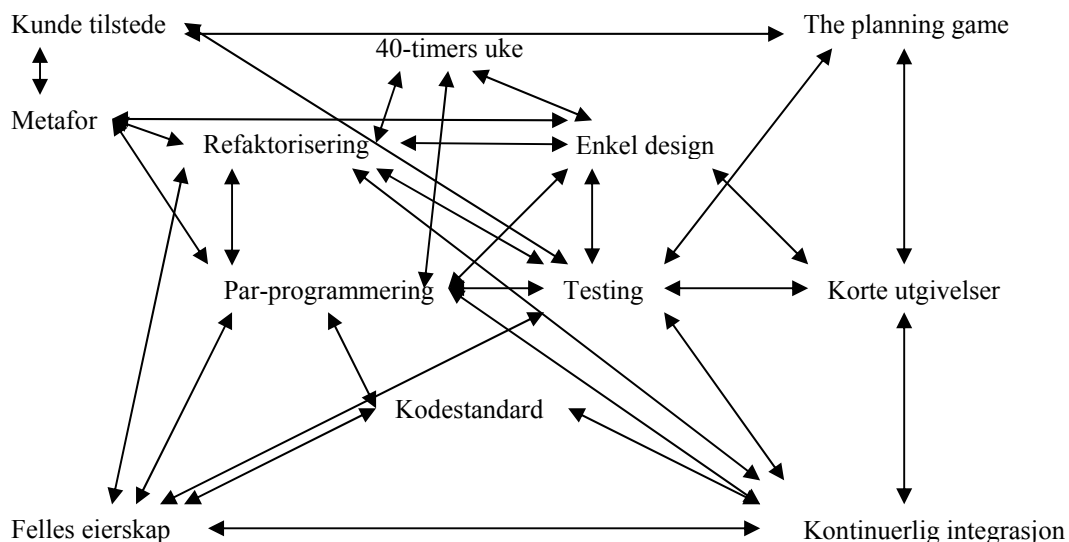
- oppklare historier
- avgjøre forretningsspørsmål
- få tilbakemeldinger
- unngå misforståelser
- svare på spørsmål
- bestemme prioriteringer på et lavere nivå

Det er viktig å ha en kunde som faktisk kommer til å bruke systemet i ettertid og som er tilstede. Kunden må finne ut hva som gir han mest verdi – at han er tilgjengelig for å gi bedre programvare eller at han er i vanlig arbeid.

12. Kodestandard

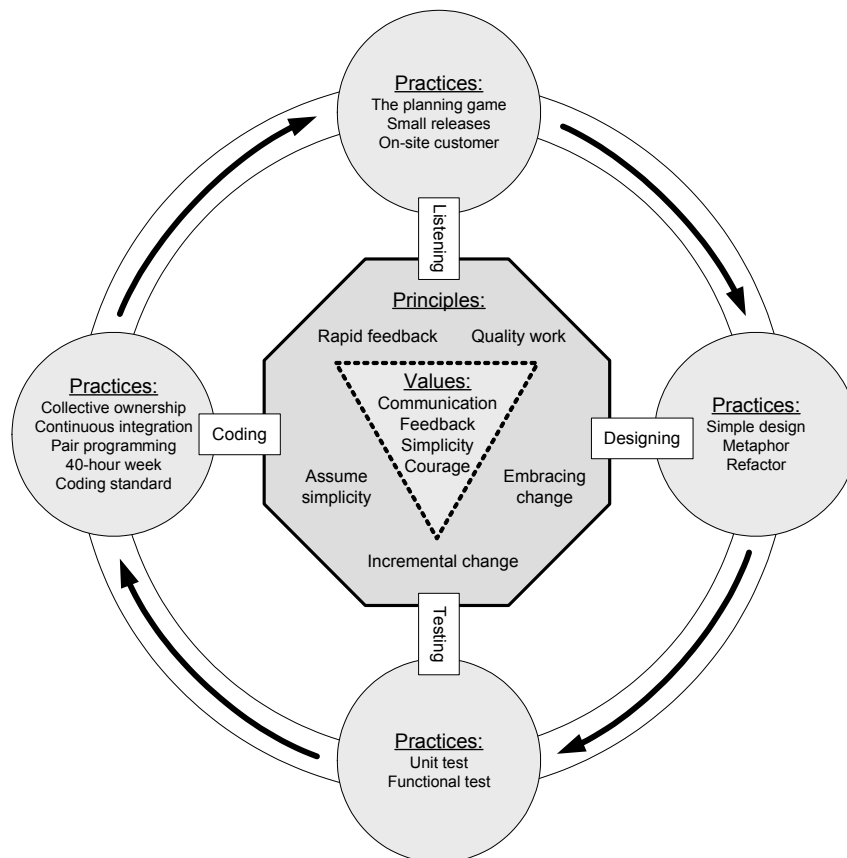
En kodestandard gjør det enklere for utviklerne å refaktorisere, skifte par og programmere raskere. Et av målene med kodestandarden er at ingen skal kunne gjenkjenne hvem som har skrevet hva. Standarden skal først og fremst forsterke kommunikasjonen blant utviklerne og må bli tatt i bruk frivillig.

3.6.1 Koblinger mellom praksisene



Figur 5. Praksisene påvirker hverandre [13].

Praksisene samhandler og styrker hverandre. Svakheten ved en praksis oppveies av styrken til en annen. Dette er viktig å være klar over siden mange av praksisene har blitt utelatt og erstattet i utviklingsprosjekter på grunn av deres svakheter. Eksempelvis blir det ikke så vanskelig å skrive tester hvis systemet har en enkel design. Misforståelser i ”The planning game” får minimale konsekvenser på grunn av korte iterasjoner i hver utgivelse og konstant feedback ved at kunden er tilstede. Se figur 5.



Figur 6. Oppsummering av grunnlaget for XP fra verdier (innerst) til praksiser (ytterst) [14].

Figur 6 viser en oppsummering av XP. Innerst i figuren ligger de fire verdiene. Ut i fra disse verdiene oppstår noen prinsipper. Sammen med basisaktivitetene koding, testing, lytting og design danner dette grunnlaget for de 12 praksisene. Pilene viser en mulig arbeidssyklus. Praksisen testing er her delt i enhetstesting og funksjonell testing.

3.7 Innredning av XP-kontor

Utvikling ved hjelp av XP bør foregå på et kontor eller kontorlandskap som er innredet på XP-vis. Det vil si at det ikke må være for trangt og at datamaskinene må være plassert slik at det er rom for par-programmering. Utviklerne bør sitte nær hverandre slik at kommunikasjonen kan gå uten vanskeligheter. Eksempelvis er det vanskelig å utføre XP hvis utviklerne sitter i to forskjellige etasjer. De 12 praksisene og historiene bør henges opp på veggen. Å trives på jobben er viktig. Dette omtales som ”Fun-factor” og innebærer blant annet at mat og drikke er lett tilgjengelig.

3.8 Ansvarsområder

I de fleste utviklingsprosjekter finnes én person med ansvar for forretning og flere personer med ansvar for utvikling. Ifølge XP skal begge gruppene dele på ansvarsområdet. "If either Business or Development gains too much power, the project suffers" [7]. Hvis personer med ansvar for forretningen får for stor gjennomslagskraft, vil alle de fire variablene (kostnader, tid, kvalitet og omfang) kontrolleres av forretningsfolkene. Det motsatte kan føre til at alt kontrolleres av utviklerne.

Løsningen er derfor å dele opp ansvarsområdet mellom kunder og utviklere. Kunder bør velge dato eller omfang for en utgivelse samt hvilke egenskaper som skal prioriteres. Utviklere estimerer hvor lang tid det tar å implementere egenskapene. De bør også få velge hvilken utviklingsmetode systemet skal utvikles ved hjelp av og vurdere konsekvensene av å velge ulike tekniske alternativer. Kunden tar avgjørelsen om valg av teknologi på grunnlag av denne vurderingen.

3.9 XP er ikke løsningen på alle problemer

XP vil ikke fungere for alle typer prosjekter og miljø. Det største hinderet for suksess er forretningskultur. Enhver ledelse som prøver å detaljstyre prosjekter, vil få problemer med et team som insisterer på å ta styringen selv. En variant av dette er dersom kunden insisterer på en komplett spesifisering, analyse eller design før en begynner å programmere.

En annen kultur som ikke passer XP, er der en må jobbe overtid for å vise innsatsvilje. Det er ikke mulig å utføre XP på en skikkelig måte dersom en er sliten, i henhold til praksisen 40-timers uke.

Størrelsen på utviklingsteamet er også viktig. Det kan være vanskelig å utføre XP-prosjekter med svært mange programmerere. Det største problemet er ofte integreringsdelen. Å integrere all kode på en maskin kan raskt bli en flaskehals. Store utviklingsteam blir også et problem fordi XP baserer seg på muntlig kommunikasjon og minimalt med dokumentasjon.

En annen barriere er et miljø der det tar lang tid å få feedback. Dette kan eksempelvis være et miljø der kunden er utilgjengelig over lengre perioder.

De fysiske forutsetningene må være på plass. Det må være enkelt for personer i et team å slå seg sammen for å par-programmere.

3.10 Fordeler og ulemper med XP

Nedenfor er det listet opp fordeler og ulemper med XP som utviklingsmetode [15].

Fordeler:

- *En kommer raskt i gang med utviklingen.* Et team lager raskt en grovskisse i starten og forbedrer den etter hvert som ting blir klarere.
- *Stor tilpasningsevne i forhold til skiftende krav.* Korte iterasjoner og praksisene par-programmering, testing, kontinuerlig integrasjon, felles eierskap og refaktorisering støtter endringer underveis i prosjektet.
- *Rask muntlig kommunikasjon.* Informasjon spres i et team ved at en skifter programmeringspartner. Par-programmering er effektivt til opplæring. En oppnår også rask feedback fra kunden ved å ha han tilstede.
- *God kvalitet gjennom*
 - *Par-programmering*
 - *Testing*
 - *Kontinuerlig integrasjon*All produksjonskode skrives i par og det skal eksistere tester for metoder som ikke er trivielle. Kontinuerlig integrasjon avdekker raskt følgefeil i systemet.
- *Finner ut hvor raskt en jobber.* Gjennom estimering av arbeidsoppgavene vil prosjektdeltakerne finne ut hvor mye arbeid de klarer å fullføre i løpet av en dag. Prosjektlederen vil ha nytte av slik informasjon for å finne ut hvor mye tid teamet trenger for å fullføre prosjektet.
- *Felles eierskap.* Kollektivt eierskap av programkoden reduserer avhengigheten av enkeltpersoner. Dette reduserer problemet med at den som skrev koden er opptatt eller har sluttet i bedriften.
- *Oppdatert dokumentasjon.* I XP genereres dokumentasjon til slutt. Dette gjør at dokumentasjonen som finnes er riktig.

Ulemper:

- *Lite på papir.* Kunden kan føle frustrasjon over mangel på dokumentasjon.
- *Par-programmering passer ikke for alle.* Par-programmering er en intens sosial aktivitet som ikke alle føler seg komfortable med.
- *Krever at alle deltar.* Enten deltar alle eller så må en finne en alternativ utviklingsmetode.

- *Krever en modig kunde.* Kunden må være villig til å ta en del av risikoen for prosjektet. Dette baseres en av de fire verdiene i XP på. Gevinsten ved å ta på seg denne risikoen kan bli et bedre produkt på kortere tid.
- *Passer ikke for store prosjekter.* Det er anbefalt med grupper på to til ti programmerere.

4 Hoveddel

Dette kapitlet omhandler bakgrunn for prosjektet og hva vi har gjort. Her presenteres også vårt bidrag til XP, hvilke resultater vi har kommet fram til og de erfaringene vi har fått. Deretter diskuteres framgangsmåtene vi vurderte og resultatene. Til slutt sammenliknes resultatene med resultatene fra liknende studier.

4.1 Bakgrunn for prosjektet

4.1.1 Hva var laget på forhånd?

Det fantes allerede en del av systemet som støtter prosjektoppfølgning internt i Bravida Geomatikk. En bruker kan for eksempel legge inn et nytt prosjekt og endre egenskaper i prosjektet. Systemet kan også sortere prosjekter etter ulike kriterier og vise forskjellig detaljeringsgrad av prosjektinformasjon. Se figur 7 og figur 8.

Status	Avd	Navn	Nummer	Fremdrift	Kunde	Team	Kommentar	Leveringsdato	Inntekter (1000kr)	Kat	Plan	Superoffice	Regnskap
+ Prospect	A2	MittProsj		1	2 % EnEllerAnnen	Per Persson, Pål Paulsen	Et veldig bra prosjekt	01.05.02	100				
+ Underveis - i rute	A2	GeoWeb Steinkjer - utvikling	MM041	90 %	null	HJL, AØ, TI	Utvikling / Vedlikehold	null	0				
+ Underveis - i rute	B1	GeoWeb Steinkjer - Stedfester	MM075	10 %	null	JSL, JAM	null	15.01.02	0				
+ Prospect	B1	testpro	kjh124	0 %	null	test	null	null	0				
+ Avsluttet - avbrutt	B2	EtDårligEtt	ZZZ1	35 %	ZZZ...ZZZ	i alle fall ikke meg	test	01.01.01	3				

Figur 7. Oversikt over prosjekter. Bravida Geomatikk ønsket ikke at vi skulle vise ekte prosjekter. Derfor er prosjektene fiktive.

4.1.2 Beskrivelse av utviklingen

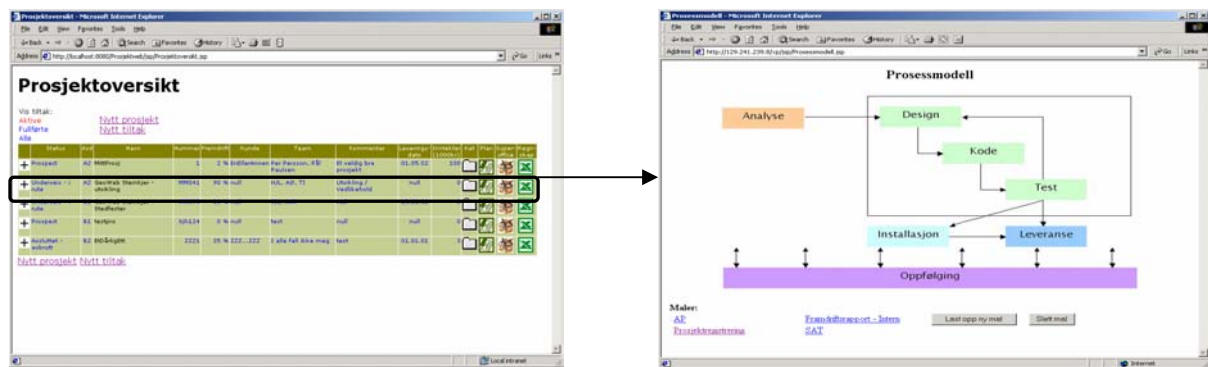
Bravida Geomatikk og Sintef Tele og Data ble enige om en kravspesifikasjon for systemet på et møte hos Bravida Geomatikk høsten 2001. Bravida Geomatikk ønsket at en skulle bruke JSP, siden det var språket de hadde erfaring med og selv kunne vedlikeholde. En utvikler hos Sintef laget deretter en prototyp i HTML-kode som demonstrerte den viktigste funksjonaliteten. Han tok utgangspunkt i de valgte verktøyene, kravspesifikasjonen og prototypen og begynte utviklingen. Etter at han hadde laget ferdig en ny prototyp, ble det

holdt en demonstrasjon hos Bravida Geomatikk. De kom med forslag til forbedringer av brukevennligheten som senere ble implementert. Testingen av systemet besto i å prøve ut funksjonaliteten fra en brukers synspunkt. Deretter ble prototypen installert. Bravida Geomatikk rapporterte i ettertid at systemet låste seg slik at det var umulig å bruke det. Utvikleren fant og rettet opp en del feil under feilsøking, men det viste seg senere at det var serveren deres som var hovedårsaken til at systemet feilet.

4.2 Hva har vi gjort?

4.2.1 Beskrivelse av systemet

Systemet består av del 1 som var laget på forhånd og del 2 som vi laget. Del 1 er en prosjektoversikt og del 2 er en prosessguide som består av to utgivelser.



Figur 8. Et skjermbilde av første del vises til venstre. Ved å velge et prosjekt i prosjektoversikten, kommer en til det vi har laget – en prosessguide.

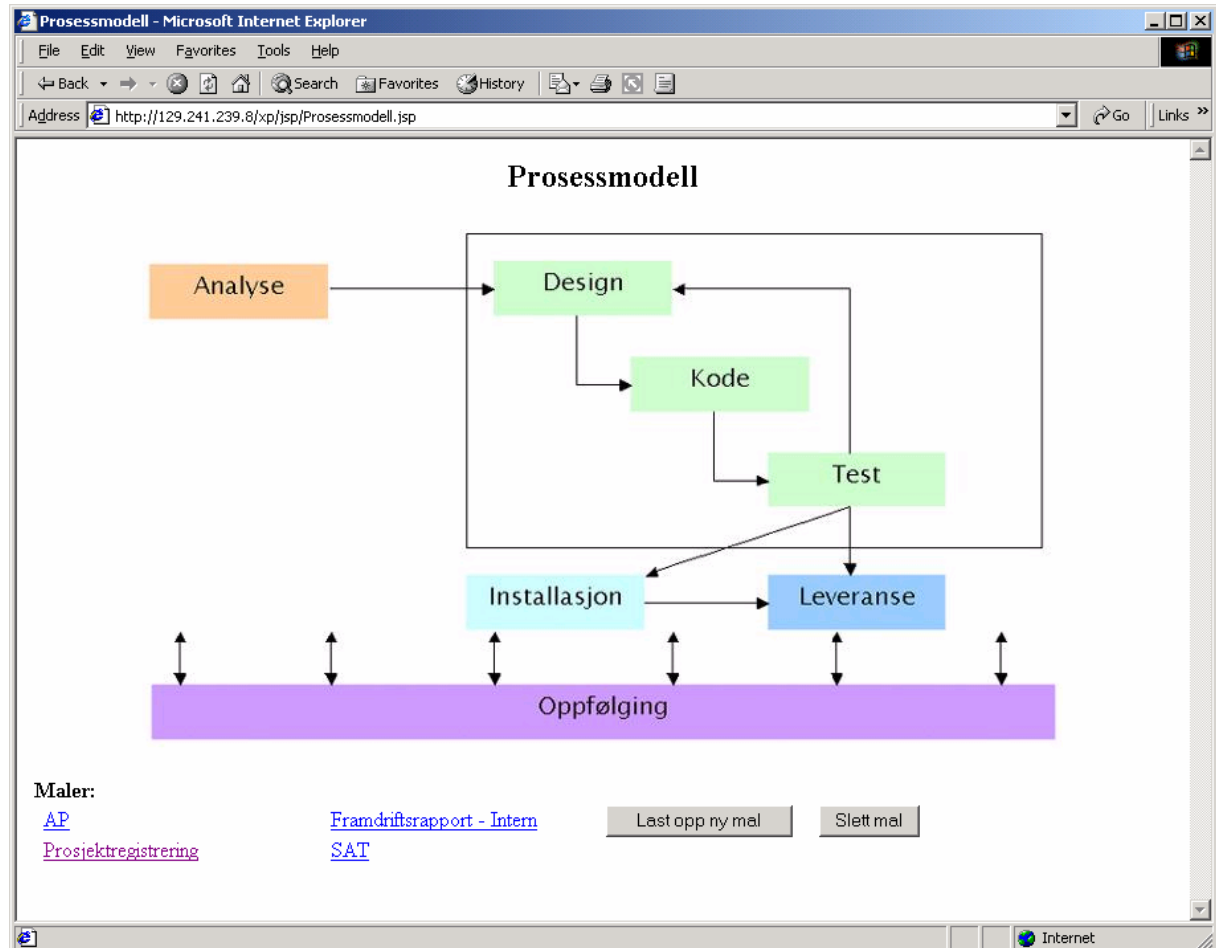
4.2.1.1 Utgivelse 1

Systemet er en elektronisk prosessguide. En bruker kan få detaljert informasjon om en prosess ved å klikke på denne prosessen i skjermbildet. Brukeren kan også laste opp, lese og slette maler. Se figur 9 og 10. Vedlegg L viser en arbeidsskisse av arkitekturen til systemet.

Disse historiene ble implementert i utgivelse 1:

1. Lage prosessmodell. Det skal være mulig å trykke på elementer i modellen.
2. Mulighet for å legge inn maler.
3. Mulighet for å vise maler.
4. Presentere nivå 2 med prosessbeskrivelse. Prosessbeskrivelsen skal være en egen fil som legges inn.
5. Navigasjon i prosessbeskrivelse. Hardkodes for å teste prinsipp.
6. Minimodell. Minimodellen skal vises i nederste venstre ramme. I tillegg skal en link tilbake til prosessmodellen lages.
7. Mulighet for å slette filer.
8. Kontroll på brukerinput. Kontroll ved brukerinput og opplasting. Dette omfatter også funksjonalitet for å unngå duplikate filer ved opplasting.
9. Maks filstørrelse i konfigurasjonsfil. Det skal være mulig å bestemme maks filstørrelse for opplasting i konfigurasjonsfil.

10. Automatiske prosessmodeller. Vurderer muligheter for å lage automatisk genererte prosessmodeller. Resultatet av historien er mulige løsninger og estimater.
11. Installere programvare på bærbar PC for presentasjon hos Bravida Geomatikk. Installere Windows 2000, Apache, Tomcat, JSDK, JDK, Textpad, JUnit, Microsoft Office 2000 og den elektroniske prosessguiden.



Figur 9. Utgivelse 1 er en elektronisk prosessguide. Ved å trykke på en prosess, får en opp detaljert informasjon om prosessen.

The screenshot shows a web browser window titled "New Page 2 - Microsoft Internet Explorer". The address bar shows the URL: http://129.241.239.8/xp/html/frame_oppfolging.html. The page content is as follows:

Formål
 Roller og Ansvar
 Input dokumenter
 Arbeidsflyt
 1. Opprette milepøler
 2. Opprette arbeidspakker
 3. Allokere ressurser
 4. Planlegg prosjektmøter
 5. Registrer prosjektet på Intranettet
 Produserte dokumenter

Bravida Prosess beskrivelse

Prosess navn: Prosjektoppfølgning og planlegging

Prosess eier: QA

Formål

For å kunne følge opp et prosjekt må alle prosjekter ha en prosjektplan. En prosjektplan skal inneholde følgende:

- En eller flere milepøler
- Arbeidspakker
- Ressurser
- Møteplan for prosjektmøter

Roller og ansvar:

Rolle	ansvar
Produktutviklingsjef/Prosjektleder	Planlegge prosjektet og allokere ressurser
Prosjekt deltaker	Gi estimater og oppgi tilgjengelige ressurser
QA	Oppdatere prosessbeskrivelse

Input dokumenter

- Prosjektramme/mandat
- Ressursliste (oversikt over tilgjengelige ressurser)

Arbeidsflyt:

1. Opprette milepøler

Prosessmodell

[TILBAKE](#)

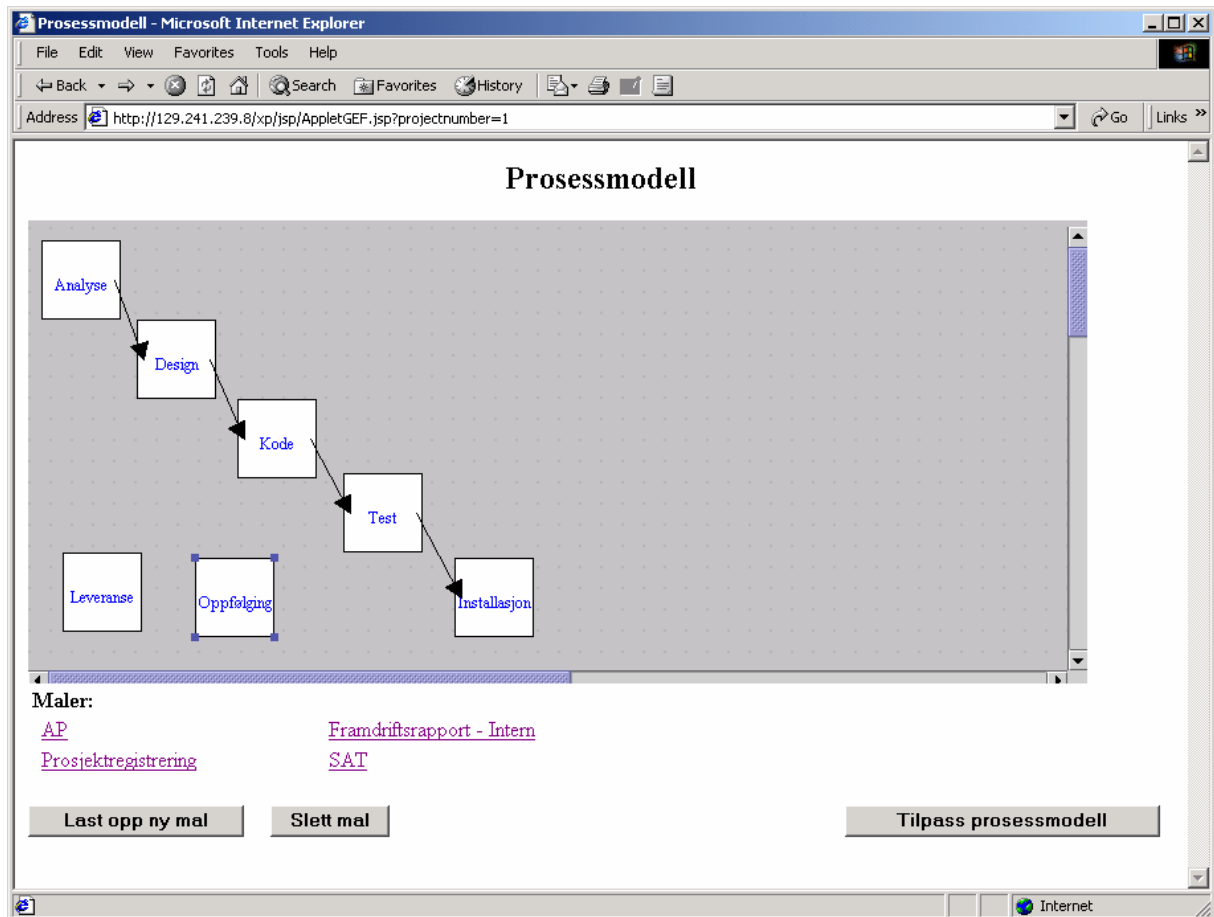
Figur 10. Detaljert informasjon om prosessen ”Oppfølging”. Linkene øverst til venstre er koblet til dokumentet som vises til høyre. Nederst til venstre vises en klikkbar minimodell.

4.2.1.2 Utgivelse 2

I tillegg til funksjonaliteten i utgivelse 1, støtter utgivelse 2 en dynamisk prosessmodell. Dette betyr at brukeren kan legge til prosesser i systemet som tegnes automatisk i en Applet. Prosessene kan så flyttes rundt i Appleten av brukeren. Prosessene kan kobles til hverandre ved hjelp av avhengigheter som legges til eller slettes etter eget ønske. Denne utgivelsen har i tillegg bedre støtte for kontroll av brukerinntut. Se figur 11.

Disse historiene ble implementert i utgivelse 2:

1. Lage prosessmodell ved hjelp av Applet. Bruk Applet å få det til å fungere slik det gjør i utgivelse 1.
2. Velge prosesser fra liste. En bruker skal kunne velge prosesser og få de tegnet i en Applet.
3. Legge til prosess. Brukeren skal kunne legge til en prosess.
4. Avhengigheter i Applet. Når en bruker velger prosesser fra liste, skal han kunne sette avhengigheter mellom dem, det vil si piler mellom boksene.
5. IP-adresse i konfigurasjonsfil. IP-adresse skal lagres i konfigurasjonsfil. Det må lages kode for å hente opp denne informasjonen.
6. Støtte for prosjektnummer. Dette gjelder hele prosjektet og er med tanke på integrering med det som er laget på forhånd (del 1).
7. Respons ved opplasting av for stor fil.



Figur 11. Utgivelse 2. Brukeren kan selv tilpasse sin prosessmodell.

4.2.2 Beskrivelse av utviklingen

Vi begynte prosjektet med å innrede kontoret på XP-vis. De 12 praksisene med beskrivelser ble hengt opp på veggen.



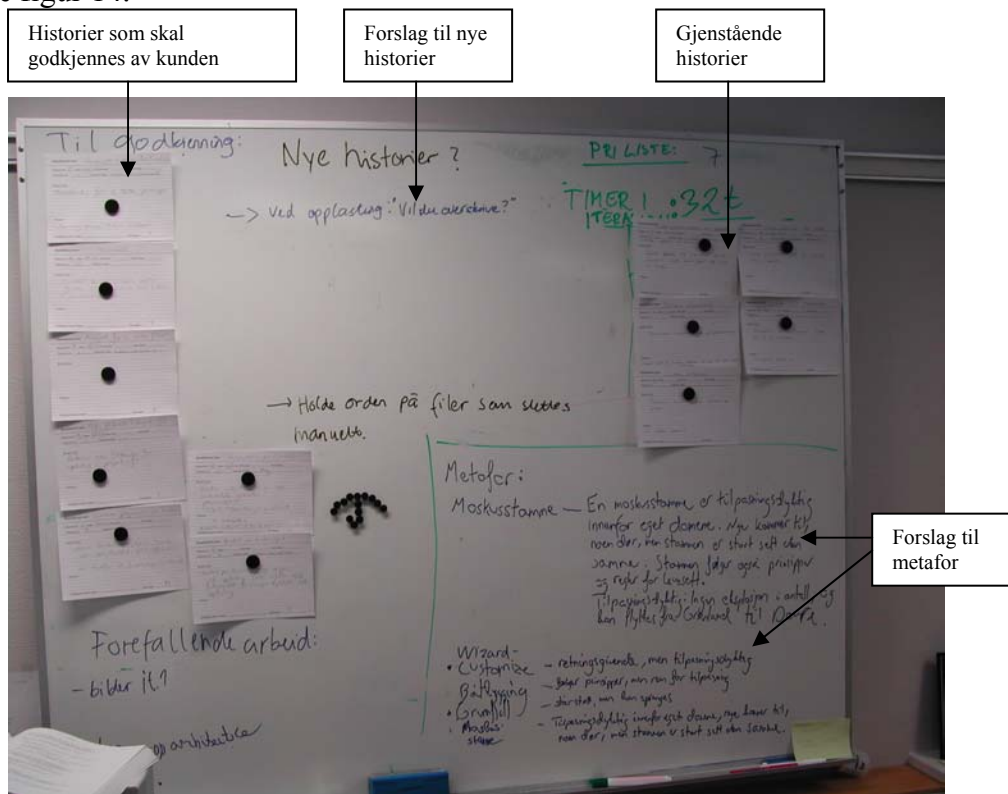
Figur 12. De tolv praksisene på veggen.

I tillegg ble tidsplan (vedlegg G), kodestandard (vedlegg E) og arkitektur (vedlegg L) hengt opp på veggen. Ei stor tavle ble brukt til å feste historiekort, skissere skjermbilder og skrive opp spørsmål til kunden. Datamaskinene ble plassert på en slik måte at de var tilpasset par-programmering. Se figur 13. Det var ellers god plass i rommet slik at både kundemøter og feedbackmøter kunne holdes på XP-kontoret.



Figur 13. Utviklerne par-programmerer på datamaskin.

Vi laget en mal for historiekort. På hver mal skrev kunden navn, dato, risiko, prioritering og en tekstlig beskrivelse av historien. Vi estimerte historien og brøt den ned i oppgaver som ble ført på baksiden av historiekortet. Historiekortene ble benyttet under hele prosjektet og festet på tavle. Se figur 14.

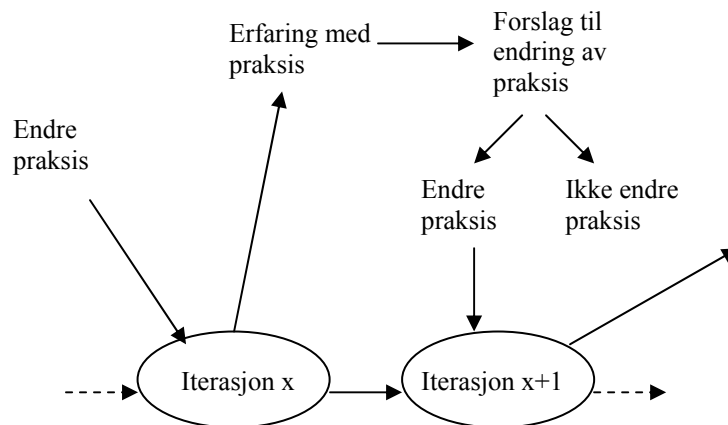


Figur 14. Historier på tavle. Fullførte historier kan sees til venstre i bildet og nye historier henger øverst til høyre.

Vi definerte lengden på en iterasjon til å være en uke. I begynnelsen av hver iterasjon hadde vi et planleggingsmøte sammen med kunden. Kunden foreslo ny funksjonalitet som ble skrevet ned på historiekort. Vi brøt deretter hver historie ned til oppgaver. Hver oppgave ble så estimert og summen av disse estimatene ble ført opp som antatt tidsbruk for en historie. Estimaten ble presentert for kunden som prioriterte blant historiene.

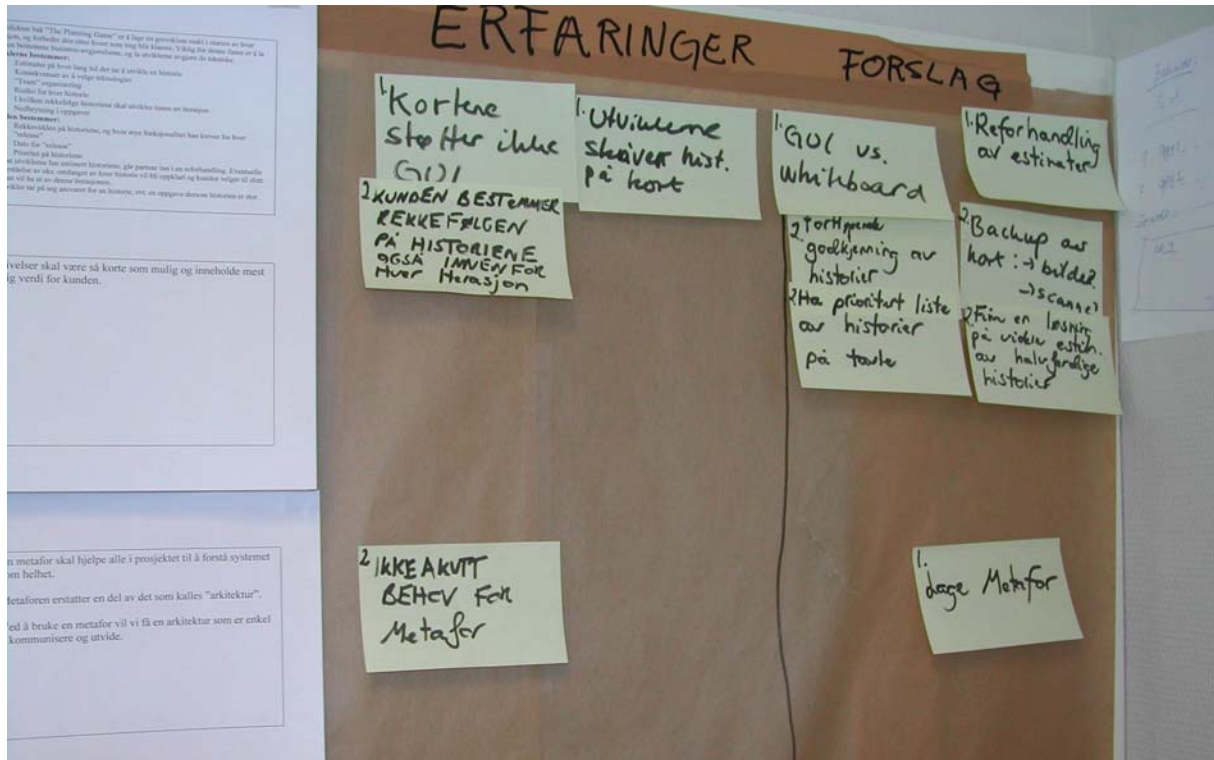
Syv iterasjoner ble gjennomført i perioden 20.2 - 19.4 og vi laget to utgivelser – den første etter iterasjon tre og den andre utgivelsen etter iterasjon syv.

De 12 praksisene var hele tiden veiledende i utviklingsprosessen. Noen praksiser ble mer benyttet enn andre. De to praksisene kontinuerlig integrasjon og metafor hadde liten nytteverdi for oss siden vi bare var to utviklere. Det var også vanskelig å finne en god metafor for elektronisk prosessguide. De fleste praksisene ble tilpasset underveis i prosjektet ved hjelp av et erfaringsbasert forbedringssystem. Forbedringssystemet besto av en beskrivelse av praksisene og gule lapper som erfaringer eller forslag til tilpasninger. Denne måten å tilpasse XP på ble til gjennom diskusjon mellom Nils Brede Moe, Håvard Julsrud Hauge og oss.



Figur 15. En modell av vår erfaringsbaserte forbedringssystem. I løpet av en iterasjon gjør vi oss erfaringer som kan bli til forslag til endringer av praksisene. Deretter kan praksisene bli endret til neste iterasjon.

I hver iterasjon kom vi opp med noen erfaringer og forslag til endringer. Se figur 16. Vi endret mest på praksisene i de første tre iterasjonene. Deretter ble det færre endringer og vi kom etter hvert fram til vår måte å utføre XP på. Artikkel [16] beskriver vår måte å tilpasse de 12 praksisene på som et eksempel på et erfaringsbasert forbedringssystem.



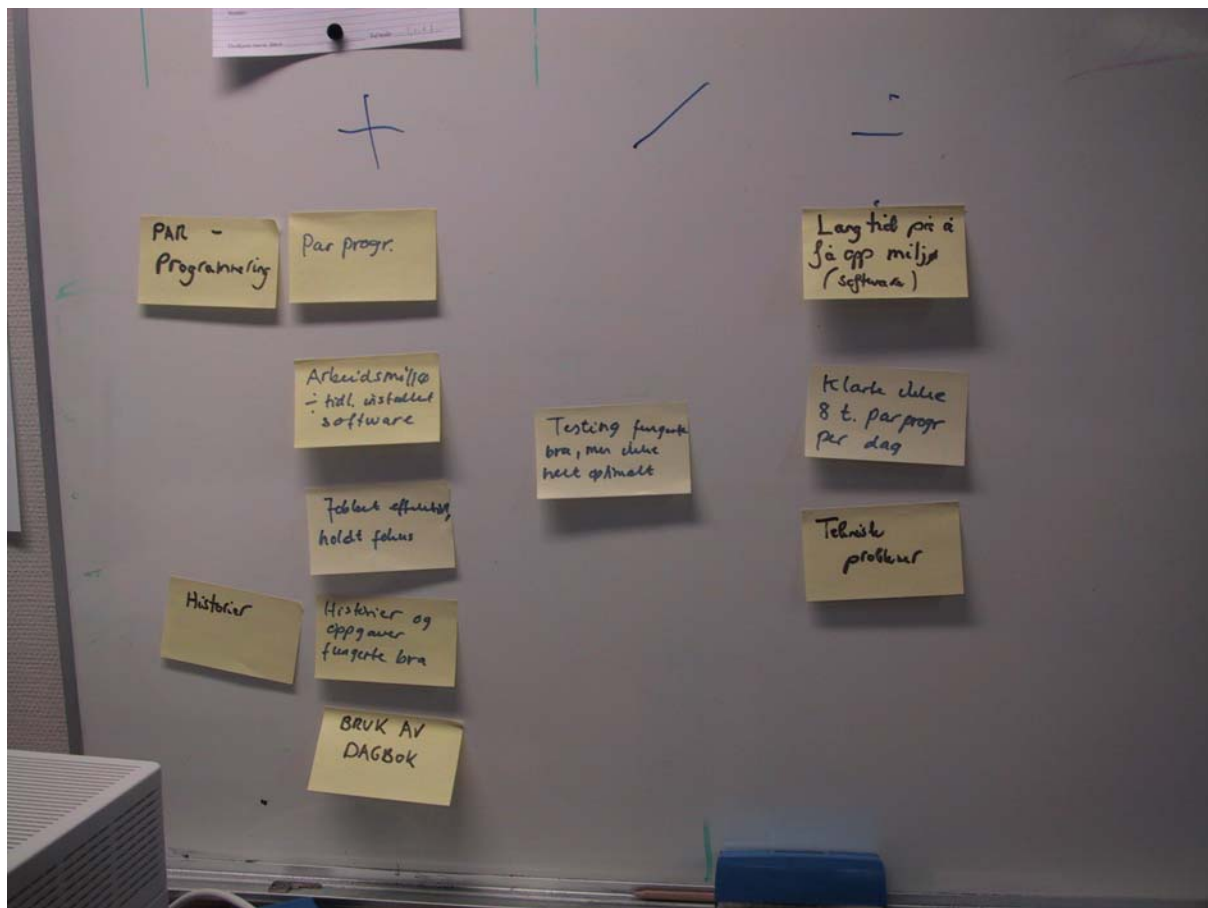
Figur 16. Erfaringer og forslag til endringer på praksisene. Tallet på hver lapp forteller hvilken iterasjon erfaringen eller forslaget kommer fra.

Hver iterasjon ble analysert i et feedbackmøte. Feedbackmøtet bestod av KJ [1], analyse av data samlet inn ved hjelp av målinger, gjennomgang av praksisene og diskusjon rundt gjennomføringen av siste iterasjon. Se vedlegg J.



Figur 17. Vi oppsummerer siste iterasjon ved hjelp av KJ.

Ved å bruke KJ fikk vi frem det som hadde gått bra og det som ikke hadde gått så bra i siste iterasjon (figur 18). Momentene som vi kom opp med ble diskutert i forhold til de 12 praksisene.



Figur 18. KJ etter iterasjon 1. Vi var fornøyde med par-programmeringen og bruken av historier, men misfornøyde med at det tok så lang tid å få opp programvaremiljøet.

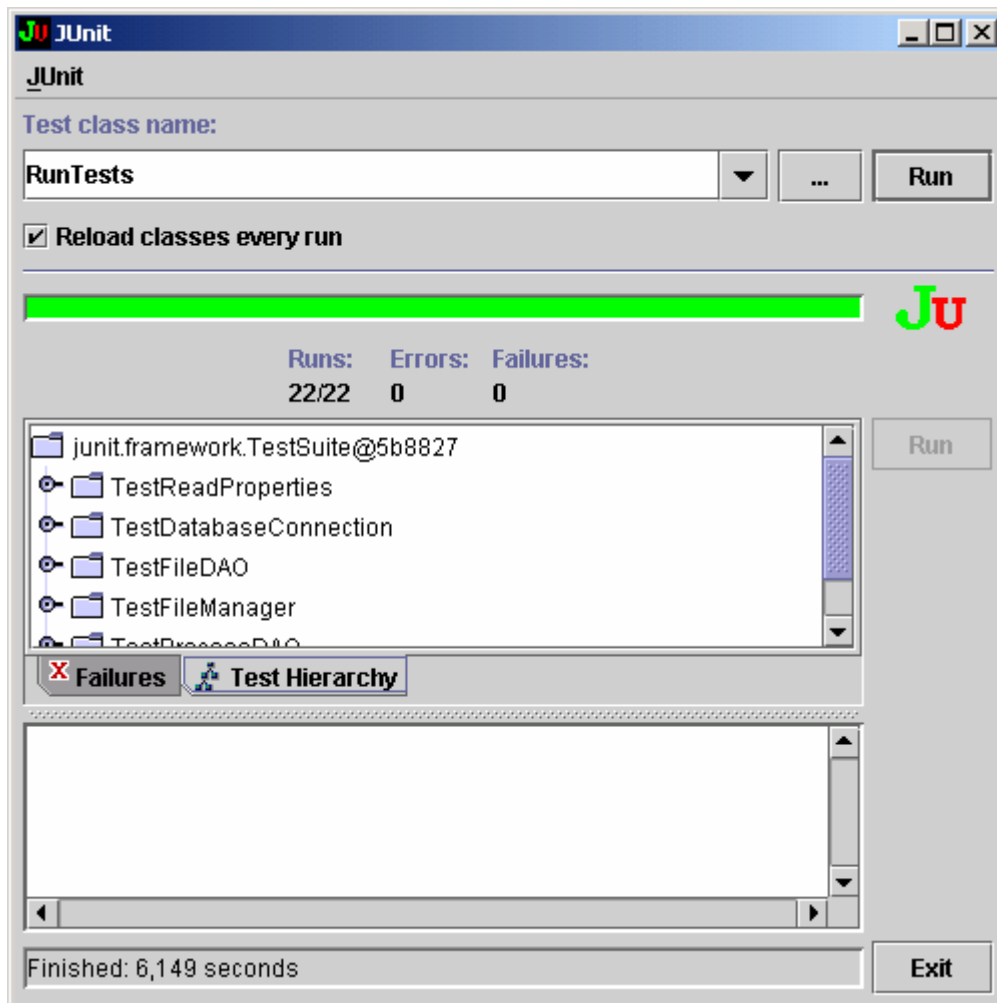
Kunden og Håvard Julsrud Hauge deltok sammen med oss på feedbackmøtene. I tillegg deltok veilederen på noen av dem. Hauge skrev diplomoppgave om prosessforbedring og kvalitetssikring i XP. Han skulle studere i hvilken grad vi klarte å bruke de erfaringene vi hadde samlet inn underveis i prosjektet.

Dagbok ble innført for å logge hendelser og timeverk (vedlegg O på CD). Vi noterte i dagboka på slutten av hver dag. Vi registrerte blant annet antall timeverk med par-programmering, refaktorisering, ”spikeing” og kundemøte. Vi noterte også antall timeverk brukt på de historiene vi fullførte og skrev begge to en beskrivelse av denne dagen. Dagboka ble brukt flittig gjennom hele prosjektet og var nyttig til å forberede feedbackmøtetene etter hver iterasjon. Se figur 19 for et eksempel på bruk av dagbok.

Dato:	5.3.2002, tirsdag
Antall timeverk refaktorisering:	0
Antall timeverk par-programmering:	8
Antall timeverk som angår Sintef-prosjektet:	14
Antall endringer i funksjonalitet på en historie:	0
Antall historier som endret prioritering:	0
Antall timeverk brukt per historie:	Historie #7: 7 (Estimat: 6,5) Historie #9: 1 (Estimat: 1,5)
Registrering av antall timeverk:	Parprogr: 8 Spikes: 4 Diskusjon med kunde: 2 Forberedelse til feedbackmøte: 1
Tekstlig beskrivelse av denne dagen:	
<ul style="list-style-type: none"> • ØM 	Vi startet dagen med en diskusjon med kunden. Vi hadde ferdigstilt alle historiene og foreslo nye historier. Disse estimerte vi og kunden prioriterte en av historiene som han ville at vi skulle fullføre innen iterasjon 2. Vi fullførte imidlertid 2 historier, siden vi fikk tid til overs på slutten av dagen. Estimatenes våre har blitt bedre på de siste historiene.
<ul style="list-style-type: none"> • OJ 	På slutten av dagen startet vi forberedelser til feedbackmøte neste dag. Par-programmering er fortsatt veldig effektivt. Når det oppstår feil, diskuterer vi oss frem til løsning på problemet på veldig kort tid og den fungerer svært ofte.

Figur 19. Utdrag fra dagbok. Denne dagen par-programmerte vi 8 timeverk og fullførte to historier.

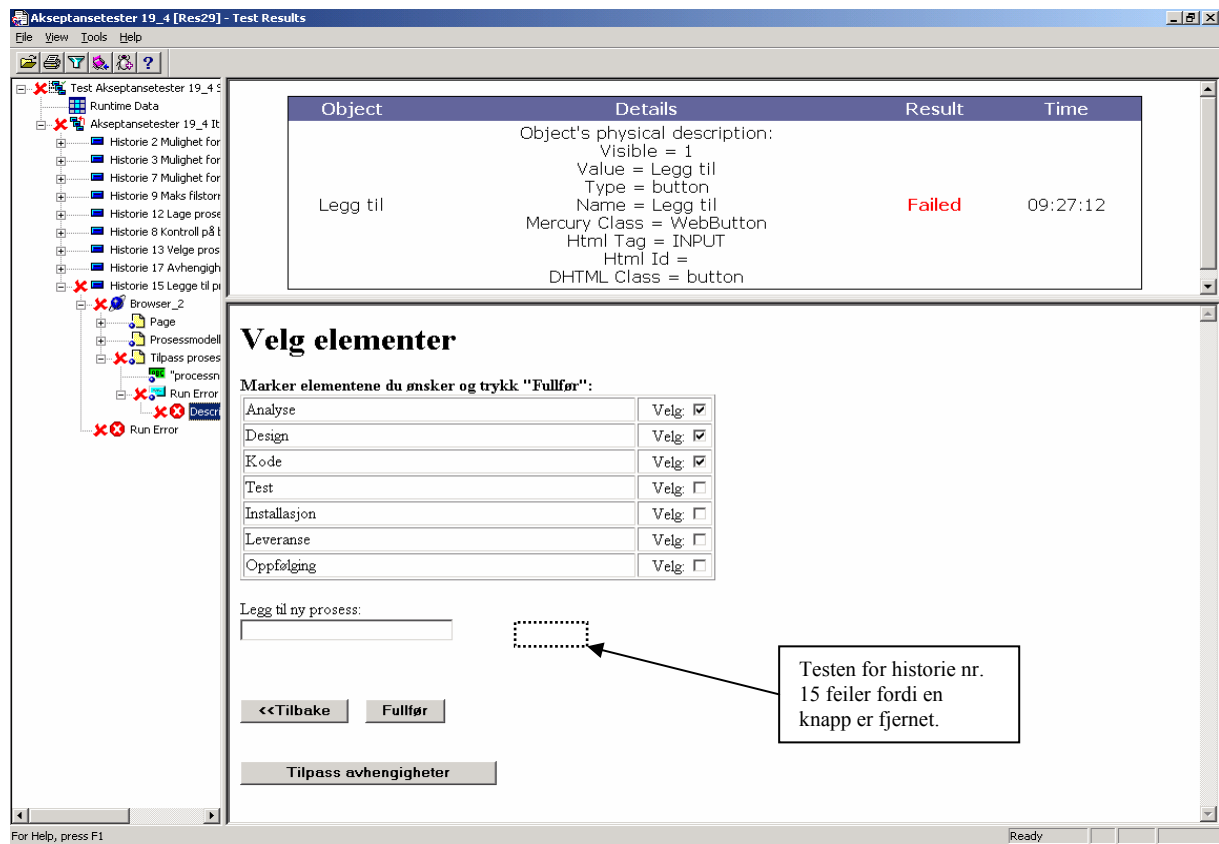
Vi tok i bruk ”test-først” prinsippet fra XP. Det vil si at vi begynte å skrive tester før vi laget koden. Vi brukte verktøyet JUnit til å teste Java-kode (figur 20). Dette verktøyet inneholder et bibliotek med funksjoner som er nyttig å bruke ved testing. I XP skal det skrives tester for all kode som ikke er trivielle. Hvis testene ikke feiler, vises ”the green bar”. Dette viser at alle testene kjører. Oppstår det avbrudd i programmeringen, skal en alltid starte opp igjen med å kjøre alle testene på nytt. Dersom ”the green bar” vises, vet utviklerne at koden fortsatt fungerer som den skal. Dette gir trygghet og selvtillit. Alle testene vi laget finnes i vedlegg N på CD.



Figur 20. JUnit med ”the green bar”.

Vi brukte test-verktøyet Astra QuickTest [17] til å teste funksjonaliteten i systemet. Vi hadde en del problemer med dette verktøyet. Blant annet krasjet det flere ganger under kjøring og påvirket også Windows-miljøet. De gangene vi fikk verktøyet til å fungere, var det nyttig. Astra QuickTest fungerer slik at hvert museklikk på hvert enkelt skjermbilde blir registrert og lagret i en eller flere tester. Det virker på samme måte som en makro. Vi skrev en test for hver historie som kunne automatiseres. En av våre tester var å laste opp et dokument og en annen var å slette det samme dokumentet. Ved å kjøre disse testene oppdaget vi om vi hadde endret på brukergrensesnittet eller om funksjonaliteten ikke lenger var som den skulle.

Nedenfor er et eksempel på en test som feiler. ”Legg til”-knappen under ”Tilpass prosessmodell” er fjernet. Dette medfører at testen stopper opp og forteller hvor det ble kjørefeil. Se figur 21.



Figur 21. Eksempel på en test som feiler.

Vi hadde et avsluttende feedbackmøte etter syvende iterasjon. På dette feedbackmøtet deltok kunden, veilederen og tre personer fra Sintef som jobbet med prosessforbedring i tillegg til oss. Vi startet møtet med å demonstrere utgivelse 1 og 2. Deretter diskuterte vi XP som utviklingsmetode og utvekslet erfaringer. Etterpå gjennomførte vi en KJ på siste iterasjon og en overordnet KJ for alle iterasjonene.

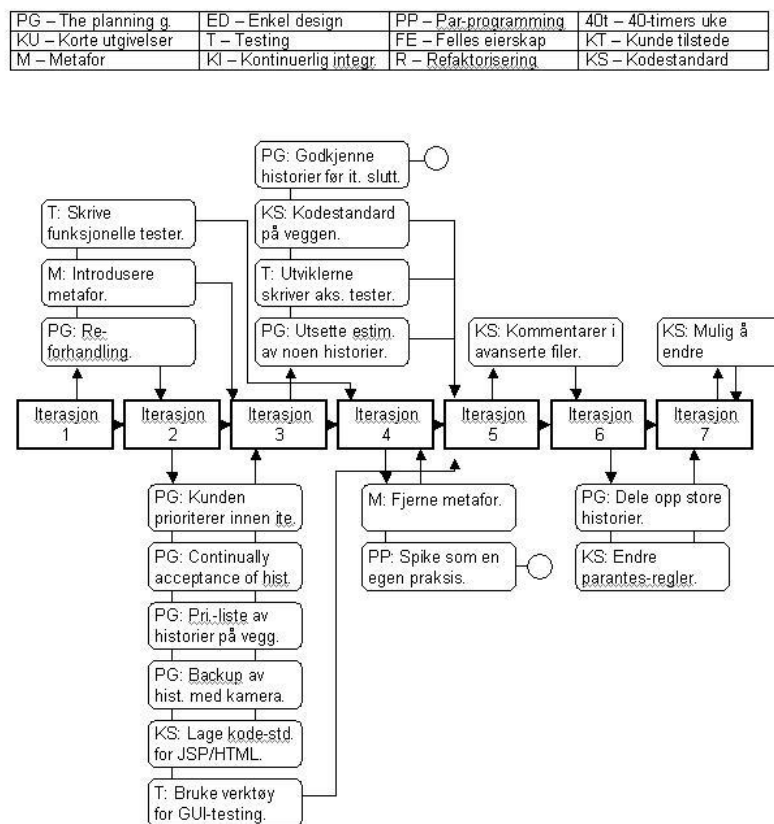
4.3 Eget bidrag til de 12 praksisene

4.3.1 Bakgrunn

Et delmål for oppgaven var å bidra med noe selv til de 12 praksisene i XP. Vi hengte de 12 praksisene på veggen og skrev erfaringer og forslag til endringer på gule lapper. Vi endret på de 12 praksisene gjennom hele utviklingsprosessen. Vi kom med forslag til endringer både i løpet av en iterasjon og i feedbackmøter og endret praksisene mellom iterasjonene. Vi satt til slutt igjen med vår måte å utføre XP på i dette prosjektet (vedlegg D).

De 12 praksisene var veiledende for utviklingsprosessen. I løpet av hver iterasjon fikk vi nye erfaringer på en eller flere praksiser. Noen erfaringer ble til forslag til å endre praksisbeskrivelsen. To forslag ble ikke gjennomført: Godkjenning av historier senest ved feedbackmøte og ”spikeing” som en egen praksis. Førstnevnte ble forsøkt innført. Kunden hadde permisjon og de gangene han var innom kontoret, hadde han det travelt. ”Spiking” som egen praksis var et forslag fra kunden. Vi følte at det ikke var behov for en ny praksisbeskrivelse fordi ”spikeing” fungerte bra, slik vi brukte den.

I feedbackmøtene gikk vi gjennom praksisene og diskuterte de erfaringene og forslagene som hadde blitt skrevet opp. Det ble skrevet ned flest forslag i de tre første iterasjonene. Deretter stabiliserte utviklingsprosessen seg og det kom opp færre forslag mot slutten av prosjektet. Noen forslag ble gjennomført direkte, mens andre forslag ble gjennomført ved senere iterasjoner. Se figur 22.



Figur 22. Oversikt over våre forslag til endringer av de 12 praksisbeskrivelsene. Pil ut av en iterasjon betyr at forslaget ble foreslått i denne iterasjonen og pil inn betyr at forslaget ble gjennomført i denne iterasjonen. Sirkel betyr at forslaget ikke ble gjennomført [18].

Praksisene korte utgivelser, kontinuerlig integrasjon, felles eierskap, refaktorisering, 40-timers uke og kunde tilstede ble ikke endret. Vi fikk lite erfaring med praksisen korte utgivelser siden vi bare hadde to utgivelser i prosjektet. Kontinuerlig integrasjon og felles eierskap ble ikke aktuelle da vi kun var to utviklere og laget all produksjonskode ved par-programmering. Refaktorisering og 40-timers uke fungerte meget bra og vi trengte ikke å endre på disse praksisene. Vi hadde heller ikke noe å utsette på praksisen kunde tilstede i vårt prosjekt, selv om kunden ikke var tilgjengelig i deler av prosjektet. Årsaken til at kunden var lite tilgjengelig, var at han hadde permisjon fra jobb midtveis i prosjektet.

4.3.2 Vår måte å utføre XP på

Vi endret på praksisene ”The planning game”, metafor, enkel design, testing, par-programmering og kodestandard. Endringer i praksisene er uthevet i kursiv. Fullstendig beskrivelser av alle praksisene finnes i vedlegg D.

”The planning game”

Utgangspunkt:

Etter iterasjon 7:

<p>Hovedideen bak ”The Planning Game” er å lage en grovskisse raskt i starten av hver iterasjon og forbedre den etter hvert som ting blir klarere. Viktig for denne fasen er å la kunden bestemme forretningsavgjørelsene og la utviklerne avgjøre de tekniske.</p> <p>Utviklerne bestemmer:</p> <ul style="list-style-type: none"> • Estimerer på hvor lang tid det tar å utvikle en historie • Konsekvenser av å velge teknologier • ”Team” organisering • Risiko for hver historie • I hvilken rekkefølge historiene skal utvikles innen en iterasjon <p>Kunden bestemmer:</p> <ul style="list-style-type: none"> • Omfanget av historiene, og hvor mye funksjonalitet han krever for hver utgivelse • Dato for utgivelse • Prioritet på historiene 	<p>Hovedideen bak ”The Planning Game” er å lage en grovskisse raskt i starten av hver iterasjon og forbedre den etter hvert som ting blir klarere. Viktig for denne fasen er å la kunden bestemme forretningsavgjørelsene og la utviklerne avgjøre de tekniske.</p> <p>Utviklerne bestemmer:</p> <ul style="list-style-type: none"> • Estimerer på hvor lang tid det tar å utvikle en historie • Konsekvenser av å velge teknologier • ”Team” organisering • Risiko for hver historie • <i>Re-estimerer på halvferdige historier pga. ny iterasjon.</i> • <i>Nedbrytning i oppgaver</i> <p>Kunden bestemmer:</p> <ul style="list-style-type: none"> • Omfanget av historiene, og hvor mye funksjonalitet han krever for hver utgivelse • Dato for utgivelse • Prioritet på historiene(<i>som vises på tavle</i>) • <i>I hvilken rekkefølge historiene skal utvikles innen en iterasjon</i> <p><i>Etter at utviklerne har estimert historiene, går partene inn i en reforhandling. Det er mulig å avvente estimering av noen historier for å minske usikkerhet. En utvikler tar på seg ansvaret for en historie, evt. en oppgave dersom historien er stor. Historier større enn en halv iterasjon bør unngås. Utviklerne skal også ta backup av historie vha. digitalt kamera. GUI av applikasjonen bør tegnes på whiteboard.</i></p>
--	--

Tabell 1. ”The planning game”.

”The planning game” ble endret på flere områder. I begynnelsen av utviklingsprosessen skulle kunden prioritere historier og vi bestemme i hvilken rekkefølge disse historiene skulle utvikles i innen en iterasjon. Kunden ønsket derimot å prioritere også innen en iterasjon fordi han visste det var en mulighet for at vi ikke klarte å fullføre alle historiene til de estimatene som var gitt. Slik fikk kunden sterkere styring på prosjektet.

For å unngå misforståelser, ble et reforhandlingsmøte innført i ”The planning game”. I reforhandlingsmøtet ble estimatene på historiene diskutert på nytt. Hvis estimatet på en historie ikke samsvarte med det kunden trodde på forhånd, ble omfanget av historien diskutert på nytt. På den måten kunne en finne ut om utviklerne hadde feiltolket størrelsen på historien.

Store historier, som gikk over en hel iterasjon, førte til at vi ikke fikk vist framgangen i prosjektet til kunden. De var også vanskelige å estimere. Vi valgte derfor å unngå historier som var større enn en halv iterasjon ved å dele disse historiene opp i mindre historier. Dette gjorde at det ble lettere å estimere.

Vi fant ut at det var best å avvente estimering av noen historier for å minske usikkerhet. Dette fordi det var vanskelig å estimere historier ved innføring av ny teknologi, som for eksempel koblingen mellom Java Applet og database. Vi ventet med å estimere til den første historien innenfor den nye teknologien ble fullført. Dette gjorde at vi fikk et bedre grunnlag for å estimere de resterende historiene.

Sammen med en annen XP-gruppe [14] laget vi en mal for historiekort som ble brukt gjennom hele utviklingsprosessen. Se figur 23.

Kundehistorie kort: <i>Legge til prosess</i>	
Historie #: <i>15</i>	Dato: <i>12.3.02</i> Prioritet: _____ Ansvarlig: _____
Teknisk est.: <i>8</i>	Teknisk risiko: <i>Middels, kognitiv i skjevbilde - ikke over skiver prosesser etc.</i>
Beskrivelse:	
<i>Brukeren skulle kunne legge til en ny prosess</i>	
Notater:	
<i>Se notater historie 14.</i>	
Godkjent (navn, dato): _____	Tid brukt: <i>1,5</i>

Figur 23. Malen vi laget for å fylle ut historiekort. Vi brukte 1,5 timeverk for å fullføre denne historien, mens vi estimerte den til 8 timeverk. Teknisk risiko ble vurdert som middels.

Historiekortet inneholder tittel, historienummer, dato, prioritet, ansvarlig, estimat, risiko beskrivelse, notater, godkjent og tid brukt. Risikofeltet fylles ut som enten høy, middels eller lav. Risikoen gjør kunden oppmerksom på usikkerheten som er forbundet med å implementere historien, for eksempel ny teknologi. Vi brukte ikke prioritetsfeltet på historiene, men laget isteden en prioritert liste av historier på tavle.

Metafor

Utgangspunkt:

Etter iterasjon 7:

En metafor skal hjelpe alle i prosjektet til å forstå systemet som helhet.	En metafor skal hjelpe alle i prosjektet til å forstå systemet som helhet.
Metaforen erstatter en del av det som kalles ”arkitektur”.	<i>Vi har ikke behov for en metafor siden prosjektet består av bare to utviklere som har jobbet sammen i flere prosjekter.</i>
Ved å bruke en metafor vil vi få en arkitektur som er enkel å kommunisere og utvide.	

Tabell 2. Metafor.

En metafor i XP skal hjelpe deltakerne i prosjektet til å forstå systemet som helhet. Vi var bare to utviklere i dette prosjektet og fant derfor ut at det ikke hadde noen hensikt å bruke denne praksisen. Det var også vanskelig å finne en god metafor for systemet.

Kodestandard

Utgangspunkt:

Etter iterasjon 7:

En kodestandard gjør det enklere for utviklerne å refaktorisere, skifte par og programmere raskere.	En kodestandard gjør det enklere for utviklerne å refaktorisere, skifte par og programmere raskere.
Målet med kodestandarden er at ingen skal kunne gjenkjenne hvem som har skrevet hva.	Målet med kodestandarden er at ingen skal kunne gjenkjenne hvem som har skrevet hva.
Standarden skal forsterke kommunikasjonen mellom utviklerne.	Standarden skal forsterke kommunikasjonen mellom utviklerne. <i>Kodestandarden kan endres underveis i prosjektet.</i> <i>Det skal eksistere en kodestandard lett tilgjengelig for Java, JSP og HTML.</i> <i>Det skal skrives utfyllende kommentarer i koden i de viktige Manager- og DAO-filene. (Se vedlegg L)</i>

Tabell 3. Kodestandard.

Vi laget kodestandard for både Java, JSP og HTML. Vi hadde god erfaring med å ha kodestandarden lett synlig på veggen. Vårt bidrag til denne praksisen ble at det skal være mulig å endre på kodestandarden underveis i prosjektet. En endring av kodestandarden kan medføre at mange filer må endres gjennom refaktorisering. Vi tror likevel at en fastlåst kodestandard ved prosjektets start kan være uheldig. Kodestandarden bør, som resten av XP, være mottakelig for endringer.

Enkel design

Utgangspunkt:

Etter iterasjon 7:

Hovedideen bak enkel design er å lage det enkleste design som kan fungere. En enkel design er en design som: 1. Kjører alle testene 2. Har ingen duplikat logikk 3. Viser hensikten med koden 4. Har minst mulig klasser og metoder	Hovedideen bak enkel design er å lage det enkleste design som kan fungere. En enkel design er en design som: 1. Kjører alle testene 2. Har ingen duplikat logikk 3. Viser hensikten med koden 4. Har minst mulig klasser og metoder. <i>Utviklerne ser hele iterasjonen under ett, samt velger løsning ut i fra egen erfaring.</i>
--	--

Tabell 4. Enkel design.

Praksisen enkel design ble tilpasset tidlig i utviklingsprosessen. Vi hadde ikke noe å tilføye på de tre første punktene i tabell 4 ovenfor, men endret punkt 4. XP-teori sier at en skal lage det enkleste design som kan fungere med minst mulig klasser og metoder. Vi valgte å se lenger fram enn en historie. På den måten laget vi ikke det enkleste som kunne fungere, men tok også høyde for de resterende historiene vi hadde fått fra kunden. Eksempelvis laget vi en arkitektur som inneholdt JavaBeans, siden vi visste at vi ville få nytte av dette i andre historier. Vi følte det ikke var noen hensikt i å lage en arkitektur som vi visste vi måtte endre på. Vi tror at vi unngikk en del refaktorisering fordi vi laget JavaBeans.

I XP lages gjerne alt helt fra begynnelsen av. Da har en full kontroll over koden. For å spare tid tok vi utgangspunkt i to mindre komponenter som vi hadde laget i tidligere prosjekter. Disse komponentene tilbyr et sett av funksjoner mot database og det å lese informasjon fra en konfigurasjonsfil. Vi har hatt god erfaring med å bruke de, men de kunne feile i et nytt prosjekt. Så lenge vi lager tester for disse komponentene, ser vi ingen ulemper med å bruke ferdiglagede komponenter vi kjenner godt fra før. Det har ingen vits å finne opp kruttet på nytt, heller ikke i XP.

Kunden ga tidlig beskjed om at han ville ha et system som var modifiserbart. Vi tok derfor utgangspunkt i en arkitektur vi kjente fra tidligere. I faget SIF 8056 Programvarearkitektur så vi nærmere på systemet eCourse [19]. Dette systemet brukte mønsteret "Modell View Controller" (MVC) som gjør det enklere å gjøre systemet modifiserbart. Vi hentet ideer fra dette prosjektet i tillegg til andre prosjekter både fra Ingeniørhøgskolen og fra praksis i bedrifter.

Vi mener at punkt 2 og 4 (tabell 4) i enkelte tilfeller kan være selvmotsigende ved bruk av MVC. Ved å lage minst mulig klasser og metoder i "Controller", må en gjerne lage mer funksjonalitet i "View". Mye funksjonalitet i "View" kan føre til duplikat logikk. Vi laget "JavaBeans" og "Data Access Objects" [19] slik at vi unngikk duplikat logikk i "View".

Testing

Utgangspunkt:

Etter iterasjon 7:

<p>Utviklerne skriver:</p> <ul style="list-style-type: none"> • Enhetstester <p>Kunden skriver:</p> <ul style="list-style-type: none"> • Funksjonelle tester <p>Uviklerne skriver enhetstester før de skriver produksjonskode.</p> <p>Kunden skriver funksjonelle tester etter å ha definert historiene.</p>	<p>Utviklerne skriver:</p> <ul style="list-style-type: none"> • Enhetstester <p>Kunden skriver:</p> <ul style="list-style-type: none"> • Funksjonelle tester <p>Uviklerne skriver enhetstester før de skriver produksjonskode.</p> <p><i>Utviklerne skriver funksjonelle tester knyttet til hver historie. Kunden godkjenner de funksjonelle testene før han godkjenner historiene.</i></p> <p><i>Verktøy benyttes for testing av GUI.</i></p>
--	--

Tabell 5. Testing.

Vårt bidrag til praksisen testing var å endre ansvarsområdet for funksjonelle tester. I XP er det først og fremst kunden og ikke utviklerne sitt ansvar å skrive funksjonelle tester. Imidlertid er dette ansvarsområdet ikke helt klart definert i metoden. Kunden gjorde en avveining og lot oss skrive de funksjonelle testene fordi han hadde lite tid. Derfor ble det vi som skrev de funksjonelle testene knyttet til hver historie. Vi laget en mal for dette:

Historie nr.	8, Kontroll på brukerinput			
Iterasjon nr.	3 (Utgivelse 2)			
Kritisk/Ikke kritisk	Ikke kritisk			
Beskrivelse av testen:	Det skal ikke være mulig å laste opp filer uten at feltene er fylt ut. Det skal også gis respons etter opplasting.			
#	Kommando	Hendelse	Data inn	Data ut
1	Gi filen en beskrivelse og laste den opp.	Fil og beskrivelse lastes opp og lagres.	Gyldig fil og tekstlig beskrivelse.	Respons på at opplastingen var vellykket.
2	Laste opp fil uten beskrivelse.	Feilmelding.	Fil.	Respons på at et felt står tomt.
3	Last opp kun beskrivelse.	Feilmelding.	Beskrivelse.	Respons på at et felt står tomt.

Figur 25. Et eksempel på en funksjonell test. Malen ble laget med utgangspunkt i [20].

Hver test inneholder en beskrivelse i tillegg til definerte hendelser som skal inntreffe når testen kjøres. Dersom hendelsene 1, 2 og 3 i dette eksempelet går feilfritt gjennom testen blir historien godkjent. Kritisk/Ikke kritisk bestemmer hvor alvorlig det er for systemet om testen

feiler. Kritisk betyr at systemet ikke vil fungere uten at denne testen kjører. Ikke kritisk vil si at systemet ikke er avhengig av om denne testen kjører for å være operativt.

Kunden godkjente de funksjonelle testene før han godkjente historiene. Vi brukte testverktøyet Astra Quicktest til å lage disse testene.

Par-programmering

Utgangspunkt:

Etter iterasjon 7:

All produksjonskode skrives av to utviklere på en maskin. Det er to roller for hvert par. 1) Den som har tastatur og mus tenker på den beste måten å programmere denne metoden akkurat nå. 2) Den andre tenker mer langsiktig og strategisk. Sjekker feil, vurderer ulike strategier og alternativer. Slår opp i kilder. Sjekker at man holder seg til kodestandard.	All produksjonskode skrives av to utviklere på en maskin. Det er to roller for hvert par. 1) Den som har tastatur og mus tenker på den beste måten å programmere denne metoden akkurat nå. 2) Den andre tenker mer langsiktig og strategisk. Sjekker feil, vurderer ulike strategier og alternativer. Slår opp i kilder. Sjekker at man holder seg til kodestandard. <i>Den som har minst erfaring med domenet sitter med tastaturet.</i>
---	---

Tabell 6. Par-programmering.

Praksisen par-programmering fungerte meget godt. Vi beholdt derfor de to rollene som var fastsatt for hvert par. Den som bruker tastaturet tenker i hovedsak kortsiktig, mens den som sitter ved siden av tenker langsiktig. Vårt bidrag er at den som har minst erfaring med et emne eller en teknologi bruker tastaturet. Dette er viktig av to grunner:

- **Opplæring.** Ved å ha en person med erfaring på emnet ved sin side, vil opplæringen skje raskt. I motsatt tilfelle kan den med erfaring programmere uten at sidekameraten har mulighet til å følge med. Han blir passivisert.
- **Feilfinning.** Vår erfaring er at det er mye enklere å oppdage feil når en ser på. Derfor kan det være hensiktsmessig å la den med erfaring sitte ved siden av.

4.4 Resultater

Vi vurderte det som var laget på forhånd, mens utvikleren hos Sintef Tele og Data vurderte vårt arbeid. Dette ble gjort for å sammenlikne produktivitet og produktkvalitet i de to delene.

4.4.1 Vurdering av det som var laget på forhånd

4.4.1.1 Produktivitet

Utvikleren hos Sintef Tele og Data brukte 168 timeverk på dette prosjektet og produserte omtrent like mye HTML-kode som JSP- og Java-kode. Han skrev 12 kodelinjer med JavaScript. Se tabell 7.

Språk	Antall linjer kode
HTML	381
JSP-kode	434
Java-kode	491
JavaScript	12

Tabell 7. Antall linjer kode produsert av utvikleren hos Sintef.

4.4.1.2 Vurdering av kode og arkitektur

Observasjoner på arkitektur:

- Det er direkte kobling mellom JSP og JavaBeans. Java Servlets benyttes ikke.
- Det finnes en databasekontakt-fil. Den brukes direkte av JSP-filene ved spørringer mot databasen.
- Objektorientering brukes i svært liten grad. Sintef-utvikleren verken lagrer eller bruker egenskapene i objektene.
- JSP-filene er store.
- Alle JSP- og Java-filene er bygd opp på samme måte og koden følger en kodestandard.
- Det er ikke laget konfigurasjonsfil. Egenskapene til databasekilden er hardkodet.
- JSP-filene kaller seg selv slik at en unngår mange filer.
- Det logges ikke til fil.

Observasjoner på kode:

- En fjerdedel av Java-funksjonene brukes ikke. Dette gjelder stort sett get()- og set()-metoder.
- Det finnes en del duplikat kode i Java-filene og i JSP-filene.
- En del kode er kommentert ut.

Kommentarer:

- Få lag med Java ned til databasenivå kan gi høy ytelse. Se skisse av arkitekturen i vedlegg L. Derimot kan mye JSP senke ytelsen på grunn av at JSP kompiles og tolkes under kjøring.
- Koden følger kodestandard og er ryddig og konsekvent.

- Mye av koden ligger i JSP-filene. Det betyr at lite av koden er kompillerbar og kan testes før kjøring. En slik kode kan være mindre modifiserbar og vedlikeholdbar.
- Det kan bli vanskelig å finne eventuelle feil hos kunden uten logging av feilmeldinger.
- Mangel på konfigurasjonsfil gjør at en kanskje må gjøre endringer i koden ved installering hos kunden.
- Kode som ikke er objektorientert kan være mindre modifiserbar og vedlikeholdbar.
- Det er større mulighet for at systemet feiler dersom kontroll på brukerinput mangler.
- Det kan være vanskelig å forstå og vedlikeholde kode der en del av den ikke er i bruk.

4.4.1.3 Vurdering av brukbarheten til systemet

- Det er enkelt for en bruker å endre informasjon om et prosjekt i verktøyet. Dersom brukeren trykker på en egenskap i et prosjekt, blir han sendt videre til en ”rediger”-side. Markøren blir plassert på den valgte egenskapen som også får en annen farge enn de andre egenskapene.
- Detaljer om et prosjekt kan vises eller skjules ved å trykke på henholdsvis ”+” eller ”-” foran et prosjekt. Se figur 26. Dette gjør prosjektoversikten oversiktlig og informativ.

The screenshot shows a web browser window titled "Prosjektoversikt - Microsoft Internet Explorer". The address bar shows "http://localhost:8080/Prosjektweb/jsp/Prosjektoversikt.jsp?visTiltak=Alle&l=+". The main content area is titled "Prosjektoversikt" and contains a table of projects. The table has columns for Status, Avd, Navn, Nummer, Fremdrift, Kunde, Team, Kommentar, Leverings-dato, Inntekter (1000kr), Kat, Plan, Super-office, and Regn-skap. The first row is expanded, showing details for "MittProsj" with status "Prospect". Below the main table, there are links for "Nytt tiltak" and "Nytt prosjekt".

Status	Avd	Navn	Nummer	Fremdrift	Kunde	Team	Kommentar	Leverings-dato	Inntekter (1000kr)	Kat	Plan	Super-office	Regn-skap
Prospect	A2	MittProsj	1	2 %	EnEllerAnnen	Per Persson, Pål Paulsen	Et veldig bra prosjekt	01.05.02	100				
		Tiltak		Status	Beskrivelse	Ansvar	Kommentar	Frist	Avhengigheter				
		test		Fullført	null	null	null	null					
		Hjelpe Per		Fullført	Få fart på Per...	Pål Paulsen	Per husker aldri noe	10.01.02					
		Oppstart		Aktiv	Komme raskere i gang	Per Persson	kanskje ikke	15.01.02	tja				
+ Undervels - i rute	A2	GeoWeb Steinkjer - utvikling	MM041	90 %	null	HJL, AØ, TI	Utvikling / Vedlikehold	null	0				
+ Undervels - i rute	B1	GeoWeb Steinkjer - Stedfester	MM075	10 %	null	JSL, JAM	null	15.01.02	0				
+ Prospect	B1	testpro	kjh124	0 %	null	test	null	null	0				
+ Avsluttet - avbrutt	B2	ETDårligEtt	2221	35 %	222...222	I alle fall ikke meg	test	01.01.01	3				

Figur 26. Detaljer om et prosjekt kan vises eller skjules ved å trykke på ”+” eller ”-”.

- Hovedsiden har en god layout med en god kombinasjon av farger.
- Feilmeldinger skrives til skjerm.
- Brukeren av systemet får ingen respons ved endringer. Dette kan gjøre brukeren frustrert.*
- Det finnes ingen kontroll på klientsiden som gir brukeren beskjed om at han prøver å legge inn for lang tekst i et felt, om han mangler tekst i et felt etc. Brukeren blir heller ikke advart på forhånd. Han kan dermed få systemet til å feile ved å skrive inn ”feil” inndata.*
- Koblingen til Excel fungerer ikke.*

* Dette ble ikke prioritert med tiden som var til rådighet.

4.4.2 Vurdering av det vi har laget

4.4.2.1 Produktivitet

Tabell 8 viser antall linjer kode vi produserte i de to utgivelsene fordelt på programmeringsspråk. Utgivelse 1 var på 182 timeverk og utgivelse 2 var på 242,5 timeverk (vedlegg I). Det ble produsert mye mer Java-kode i utgivelse 2 enn i utgivelse 1. Kunden skiftet kurs etter utgivelse 1 og ønsket en dynamisk prosessmodell. Prosessmodellen ble derfor endret fra et statisk JPG-bilde til en Java Applet. Sammen med en del konfigurering og installering i utgivelse 1, forklarer dette økningen i antall linjer Java-kode.

Språk	Antall kodelinjer i utgivelse 1	Antall kodelinjer i utgivelse 2	Sum antall kodelinjer
HTML	178	180	358
JSP	39	108	147
Java	489	1255	1744
JavaScript	39	59	98

Tabell 8. Antall linjer kode produsert per utgivelse.

Tabell 9 viser totalt antall timeverk fordelt på aktivitetene i andre del av prosjektet (utgivelse 1 og 2). Totalt ble det arbeidet 424,5 timeverk i del 2. En ser av tabellen at det ble en del konfigurering og installering i de to første iterasjonene og ingenting senere i prosjektet. Kundemøtene var korte gjennom hele prosjektet.

Iterasjon nr.	Parprogrammering	"Spikeing"	Konfigurering/Installering	Kundemøte	Diverse
1	16,5	17	31	2,5	5
2	26	13,5	17	2,5	0
3	36	9	0	1	4
4	31	27	0	1,5	0
5	25,5	11,5	0	0	5
6	29	32	0	2	0
7	43	30,5	0	2,5	3
Sum	207	140,5	48	12	17

Tabell 9. Totalt antall timeverk fordelt på aktiviteter.

Tabell 10 viser en prosentvis fordeling av timeverk på aktiviteter. Vi trodde på forhånd at vi skulle bruke halvparten av tiden på par-programmering og mellom 1/3 og 1/4 av tiden på "spikeing" (vedlegg P på CD). Dette viste seg å stemme bra.

Aktivitet	Prosent av total
Idealtimer parprogrammering	48,8 %
Andel spikeing	33,1 %
Konfigurering/Installering	11,3 %
Kundemøte	2,8 %
Diverse	4,0 %

Tabell 10. Prosentvis fordeling av timeverk på aktiviteter.

4.4.2.2 Vurdering av kode og arkitektur

Utvikleren fra Sintef Tele og Data vurderte vår kode og arkitektur ved å svare på spørsmålene nedenfor.

1. Hvor enkelt er det å lese og forstå koden?

(Ta utgangspunkt i funksjonaliteten som kontrollerer sletting av fil; DeleteFiles.jsp, DeleteFileServlet.java, FileManager.java, FileDAO.java, FileObject.java, DatabaseConnection.java)

Svar:

"Koden er lett å lese og forstå. Det kunne vært litt flere kommentarer. Det tar noe tid å se på all funksjonaliteten siden en må gå igjennom flere filer. Flere filer gir ofte lettere lesbar kode og kan ikke sies å være særlig negativt, men å følge alt for mange filer kan bli litt tungt. Det er mange filer i forhold til størrelsen på programmet"

I tillegg fikk vi disse kommentarene:

Hva var bra?

- *"Holder samme kodestandard"*
- *"Alle variable på samme språk"*

Hva var ikke bra?

- *"Lange kodelinjer gjør koden uoversiktlig pga. horisontal scrolling"*
- *"Mange nivå og store innrykk gir lange kodelinjer og horisontal scrolling"*
- *"Uheldig med både HTML og Java på samme linje (I alle fall hvis der dreier seg om mer enn en variabel)"*
- *"Kommentarer er stort sett på engelsk, men litt er på norsk"*

2. Hvor enkelt er det å legge inn funksjonalitet som, for eksempel, å slette en prosess?

Svar:

"Krever noe tid for å sette seg inn i lignende kode (eks. kode for å legge til en prosess). Etter det burde det være greit å legge til ønsket funksjonalitet. En svakhet kan være at en ikke kan legge alt inn i en (to) fil(er), men må oppdatere flere for å få lagt til en enkelt ting."

3. Hva er din vurdering av arkitekturen i del 2 (det vi har laget) sammenliknet med del 1 (det du laget)? Eksempel: Objektorientering, modifiserbarhet, vedlikeholdbarhet, ytelse, testbarhet.

Svar (Se arbeidsskisse av arkitekturen i vedlegg L):

"Del 2 bruker flere objekter. Dette fører til at en må forholde seg til flere filer på alle nivå:

- *En må jobbe med flere filer kode, for å lese og for å sette seg inn i (legge til) en funksjonalitet --> Utvikling går tregere*
- *Flere nivåer kan gjøre programmet litt tregere, men dette vil antagelig ikke ha så stor innvirkning*
- *Mindre objekter kan gi mindre og enklere tester*
- *Raskere å lage ferdig hvert enkelt objekt*

- *Flere objekter gjør det lettere for flere personer å jobbe på den samme koden*
- *Hvert enkelt objekt er lettere å få oversikt over*

4.4.2.3 Vurdering av brukbarheten til systemet

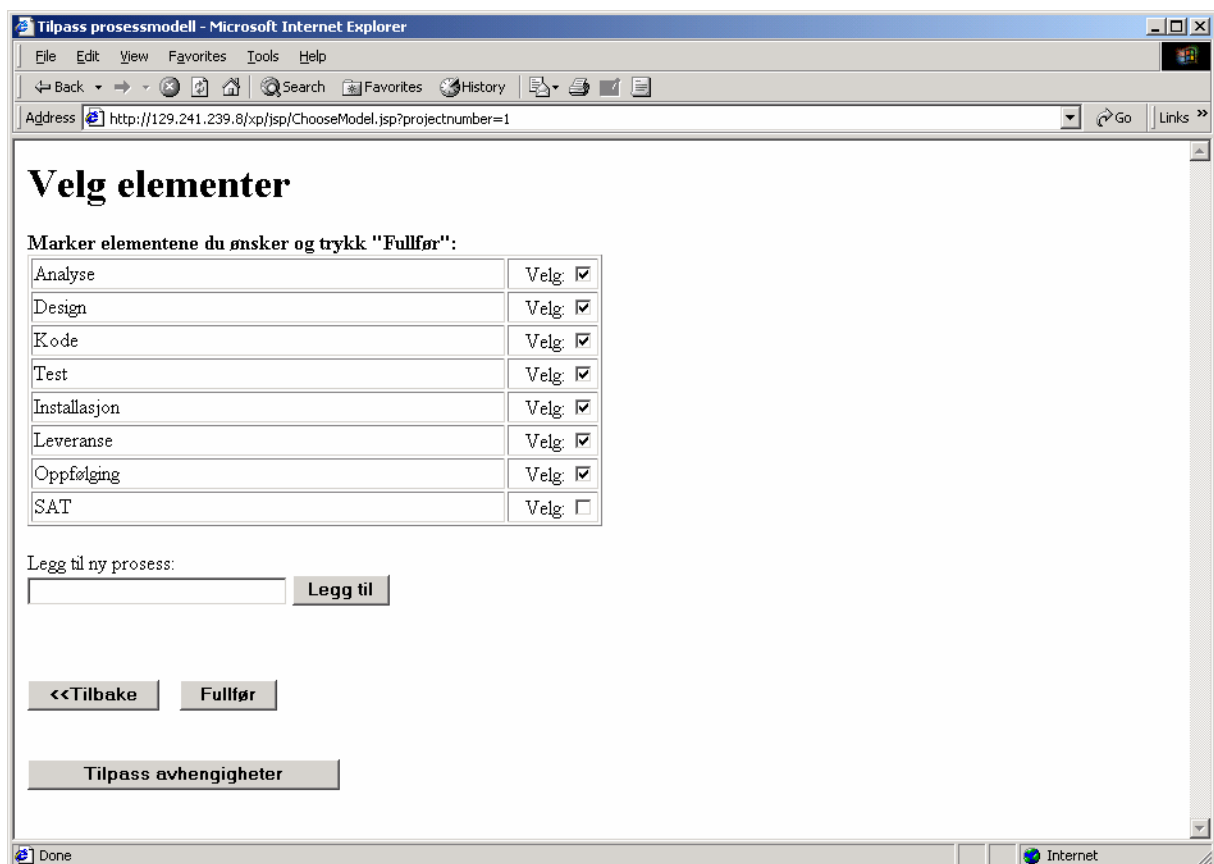
Utvikleren fra Sintef Tele og Data testet våre utgivelser. Han hadde følgende kommentarer:

Utgivelse 1

- ”Gir en grei oversikt”
- ”Greie farger”
- ”Bra at en kan gå direkte inn i maler ved å trykke på linker”

Utgivelse 2

- ”Fint at en kan dra prosessene rundt som en vil, men det er vanskelig å markere prosessene innimellom”
- ”Bra at brukeren blir tvunget til å skrive inn alt som er nødvendig, eksempelvis ved opplasting av filer”
- ”Funksjonaliteten er ganske likt windows-standard med eksempelvis <<Tilbake>>-knapper”
- ”Savner et standardoppsett på prosessmodellen som i utgivelse 1, spesielt når en laster opp siden på nytt”
- ”Høyre museklikkfunksjonalitet er litt uklar”
- ”Fullfør-knapp under <<Tilpass prosessmodell>> bør kanskje flyttes”



Figur 27. Utvikleren hos Sintef mente layouten kunne endres noe i dette skjermbildet. ”Fullfør-knappen” kunne vært flyttet til høyre for ”Legg til-knappen”.

Sintef-utvikleren konkluderte med:

”Utgivelse 1 er mer oversiktelig enn utgivelse 2. Utgivelse 2 gir brukeren mer frihet, men er mer rotete”.

4.4.2.4 Testing

Det blir en del testkode på grunn av ”test-først” prinsippet i XP (se kapittel 3.6).

Antall linjer testkode	293
Antall linjer produksjonskode	1744
Størrelsesforhold	1:6

Tabell 11. Størrelsesforholdet mellom testkode og produksjonskode ble 1:6. Dette gjelder kun Java-kode. Astra Quicktest ble brukt til å teste HTML, JSP og JavaScript.

Antall tester i JUnit (Java)	25
Antall metoder i kode	61
Størrelsesforhold	2:5

Tabell 12. Størrelsesforholdet ble 2:5 mellom antall tester og antall metoder som ble implementert. Dette gjelder kun Java-kode. Filer som omhandler testing og Java Servlet-filer er ikke tatt med i beregningen.

Det ble skrevet 25 enhetstester i JUnit til de 61 metodene som ble implementert (se tabell 12). Vi unnlot å skrive tester på get()- og set()-metoder og skrev bare tester for metoder som ikke var trivielle, jamfør praksisen testing. Eksempel på en enkel get()-metode fra ”ReadProperties.java” (Vedlegg N på CD):

```
public static String getDriver() {
    return dbDriver;
} //getDriver()
```

Det ble kun laget 9 automatiske tester i Astra Quicktest, mens 18 historier ble fullført. Grunnen til at det ikke ble laget flere funksjonelle tester var at ikke alle historiene kunne testes i dette verktøyet. Eksempelvis kunne ikke editering i konfigurasjonsfilen gjøres fordi Astra Quicktest bare tester gjennom en nettleser. Vi skrev likevel akseptansetester for alle historiene. Se vedlegg H.

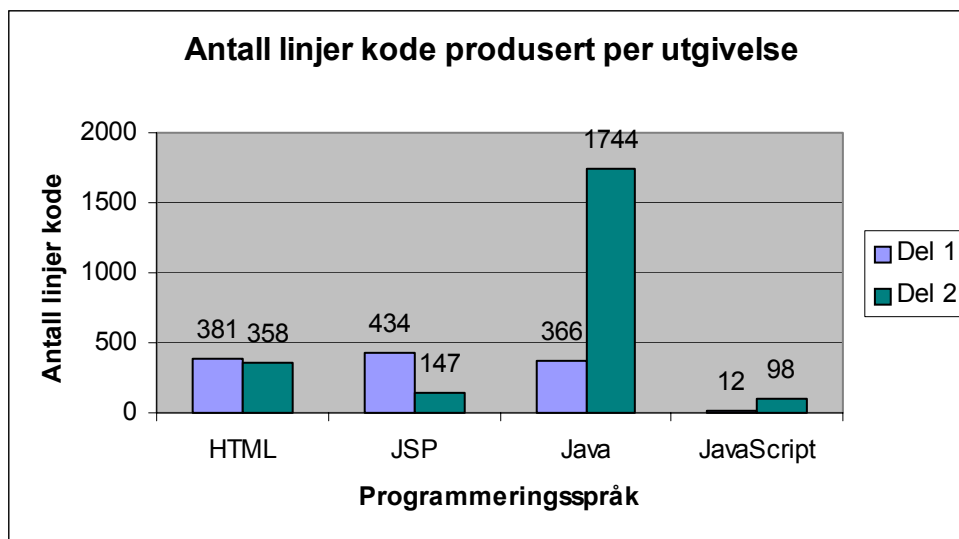
4.4.3 Sammenlikning av det som var laget på forhånd og vår del

4.4.3.1 Produktivitet

Tabell 13 og figur 28 gir en oversikt over antall linjer kode per utgivelse fordelt på programmeringsspråk. Del 1 var på 168 timeverk og ble laget av en utvikler hos Sintef. Del 2 besto av utgivelsene 1 og 2 og var på til sammen 424,5 timeverk. Kodelinjene i del 1 og del 2 var sammenliknbare. Vi hadde ganske lik kodenstandard for Java. JSP- og HTML-kode på samme linje ble telt som to linjer.

Språk	Del 1	Del 2
HTML	381	358
JSP	434	147
Java	366	1744
JavaScript	12	98
Sum	1193	2347

Tabell 13. Antall linjer kode fordelt på programmeringsspråk og utgivelser.



Figur 28. Del 1 (det som var laget på forhånd) inneholdt mer HTML- og JSP-kode enn del 2 (det vi laget), men mindre Java- og JavaScript-kode.

Dersom vi slår sammen antall kodelinjer i programmeringsspråkene HTML, JSP, Java og JavaScript, hadde del 1 en produktivitet på 7,1 linjer kode per timeverk. Til sammenlikning ble det produsert 5,5 linjer kode per timeverk i del 2. Forskjellen blir mindre hvis en tar i betraktning at det fantes en del duplikat kode i del 1 (se diskusjon av resultat). Dette tyder på at produktiviteten ble noe lavere ved å bruke XP, i beste fall like bra som ved prototyping. Se diskusjon av resultater i kapittel 5.3.

4.4.3.2 Kode og arkitektur

Figur 28 viser at del 1 la en stor del av funksjonaliteten i HTML- og JSP-filene, mens del 2 (utgivelse 1 og 2) la en større del i Java-filene. Vedlegg L, som viser skisser av to arkitekturer, illustrerer dette poenget. Skissene viser også at del 1 brukte færre klasser og lag i arkitekturen enn del 2. Få lag ned til databasenivå kan gi høy ytelse, mens mye JSP kan igjen senke ytelsen fordi JSP kompiles og tolkes under kjøring. Utvikleren hos Sintef poengterte at mange lag fører til at en utvikler må forholde seg til mange filer på flere nivå.

Bruken av objektorientering er noe av det som skilte del 1 og del 2 fra hverandre. Del 1 brukte objektorientering i svært liten grad, mens del 2 brukte objektorientering fullt ut. XP synes å tvinge fram en objektorientert arkitektur på grunn av den utstrakte bruken av automatiserte enhetstester. En får lettere testet mer kode ved hjelp av enhetstesting dersom mye av funksjonaliteten ligger i Java-filer sammenliknet med HTML- og JSP-filer. Objektorientert kode kan også være enklere å modifisere og vedlikeholde.

Del 1 hadde lite JavaScript-kode sammenliknet med del 2. Dette kan forklares med at del 1 ikke hadde noen kontroll på brukerinnt. Kritikken utvikleren hos Sintef ga vår kode gikk mest på kodestandarden. Eksempelvis mente han at det var uheldig med både HTML og Java på samme linje.

Del 1 hadde noe duplikat kode og en fjerdedel av Java-koden ble ikke brukt. Refaktorisering og bruken av objektorientering kan å ha fått bukt med dette problemet i vår del av prosjektet. Dette indikerer at kvaliteten på koden ble bedre ved bruk av XP.

4.4.3.3 Brukbarhet

Funksjonaliteten på det vi laget og det som ble laget på forhånd er forskjellig. En kan derfor ikke uten videre sammenlikne brukbarheten. Det som *er* felles for begge systemene, er at brukeren kan endre informasjon i tekstfelt. I motsetning til del 2, får brukerne av del 1 ingen respons når de gjør endringer. Det finnes heller ingen kontroll av brukerinnt, noe som kan få systemet til å feile ved ”feil” inndata. Dette kan gjøre brukeren av systemet frustrert. Det må presiseres at kontroll på brukerinnt og respons til brukeren ikke ble prioritert med den tiden som var til rådighet.

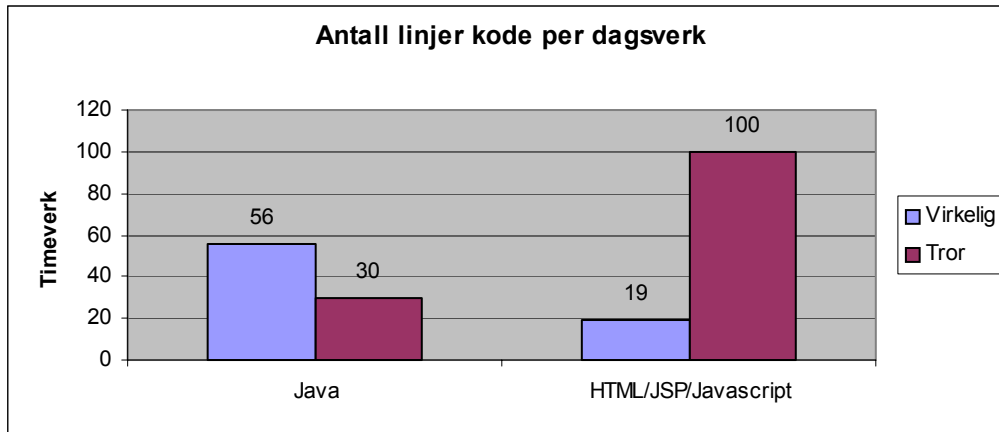
4.4.4 GQM-planen

4.4.4.1 Bakgrunn

Vi definerte GQM-målet slik: ”Analyser XP-prosessen for å forstå effekten i et utviklingsprosjekt sett fra utviklerne i følgende omgivelser: Intranettløsning hos Bravida”. Vi definerte til sammen 12 spørsmål og 31 metrikker i GQM-planen. Interessante avvik fra hva vi trodde på forhånd og samvariasjoner mellom de 12 praksisene er beskrevet nedenfor. Resten av resultatene fra målingene med kommentarer finnes i vedlegg C.

4.4.4.2 Interessante avvik

Hvor stor er produktiviteten? (Q2)



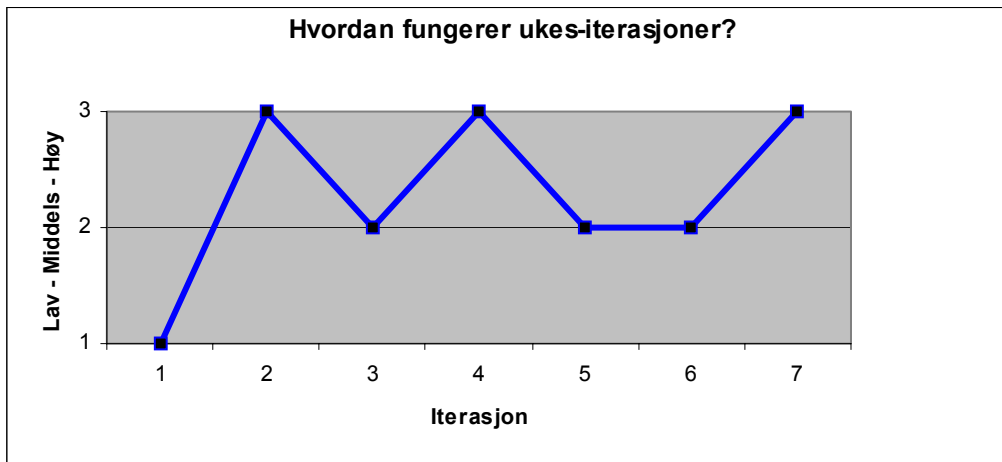
Figur 29. Avvik i produktiviteten i forhold til det vi trodde på forhånd.

Vi implementerte mer Java-kode og mindre HTML-, JSP-, og JavaScript-kode per dagsverk enn det vi trodde på forhånd. Helt fra begynnelsen av la vi vekt på å implementere mest mulig funksjonalitet i Java. Grunnen til det var at Java er et objektorientert språk som gjør det enklere å lage enhetstester. JSP og HTML kompiles under kjøring og er tregere enn Java-kode.

Vi implementerte funksjoner i Java som tilbød operasjoner mot database, filer og objekter for å unngå mye og duplisert kode i JSP. Derfor ble JSP-koden lettere å lese. JSP er et imperativt språk som lett kan bli uoversiktlig når det er mye kode.

I andre utgivelse skiftet kunden kurs og ønsket at en del av brukergrensesnittet skulle være dynamisk. Denne løsningen ble implementert som en Java applet. Derfor ble det stor forskjell på antall linjer Java-kode implementert og det vi antok på forhånd. Dette medførte også at det ble mindre HTML- og JSP-kode.

Hvordan fungerer ukes-iterasjoner? (Q3)



Figur 30. Slik fungerte ukes-iterasjoner.

Vi trodde på forhånd at iterasjoner med lengde på bare en uke stort sett skulle fungere bra (nivå 3 i figur 30). Dette ble ikke tilfellet selv om bare en iterasjon ble evaluert til "Lav".

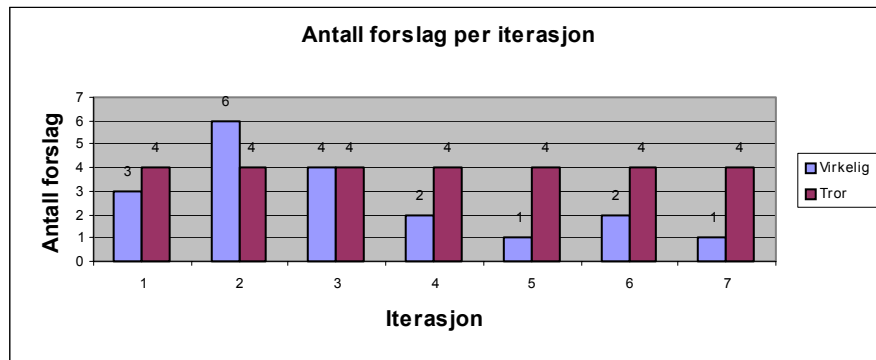
Hvorfor fungerte noen iterasjoner bra?

- Vi fikk rask tilbakemelding etter hvert feedbackmøtet. Dette gjorde at vi kunne forbedre prosessen.
- Vi fikk mye data ut av en kort utviklingsperiode.

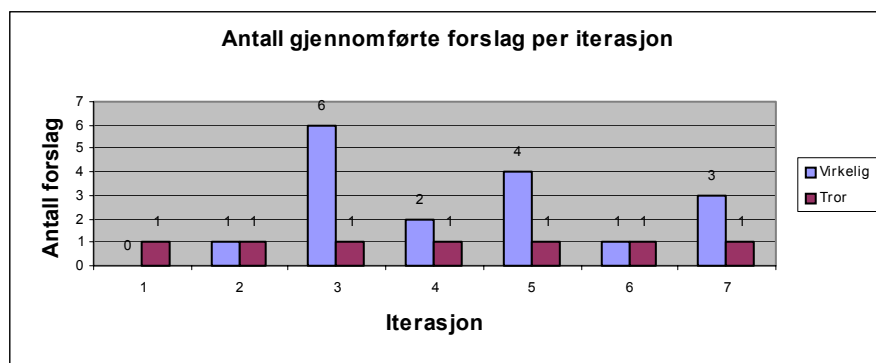
Hvorfor fungerte andre iterasjoner mindre bra?

- Iterasjoner uten et skikkelig "The planning game" med kunden fungerte mindre bra. Dette førte noen ganger til misforståelser rundt historienes omfang, prioritering etc.
- Store historier, som gikk over en hel iterasjon, førte til at vi ikke fikk vist framgangen i prosjektet til kunden.

Hvor lett er det å overføre erfaringer fra tidligere iterasjoner? (Q5)



Figur 31. Mange forslag til endringer på prosessen i begynnelsen av prosjektet og færre mot slutten.



Figur 32. Forslag ble gjennomført gjennom hele utviklingsperioden.

Vi trodde at vi skulle komme med mange forslag til endringer på prosessen, men gjennomføre ganske få. Vi endte opp med færre forslag enn antatt, men hele 17 av 19 forslag ble gjennomført. Kun tre av forslagene som ble gjennomført kom fra iterasjoner tidligere enn siste iterasjon.

Hvorfor ble det slik?

- Det gikk greit å overføre erfaringer pga. de ukentlige feedbackmøtene. På hvert feedbackmøte gikk vi igjennom alle praksisene.
- Vi foreslo og dokumenterte alle endringene fortløpende under utviklingen når vi oppdaget at det fantes rom for forbedringer.
- Vi innførte forslag til endringer så raskt som mulig. De fleste forslagene ble gjennomført i samme iterasjon som de ble foreslått i eller neste iterasjon. Dette er grunnen til at så få forslag kom fra før forrige iterasjon.

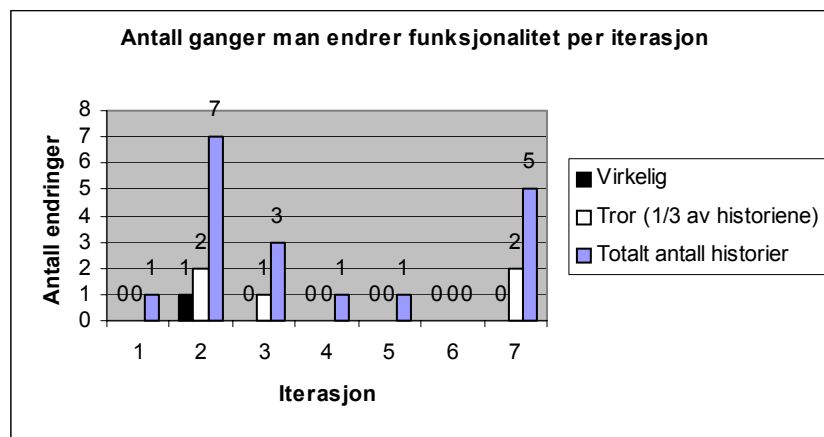
Håvard Julsrud Hauge skulle finne ut i hvilken grad vi klarte å bruke de erfaringene vi hadde opparbeidet oss underveis i prosjektet. På lik linje med oss, fant han ut at det var lettere å innføre endringer enn det han trodde på forhånd. Hauge mente at grunnen var at utviklerne *selv* foreslo endringer for å tilpasse praksisene [18].

Disse praksisene ble endret flest ganger:

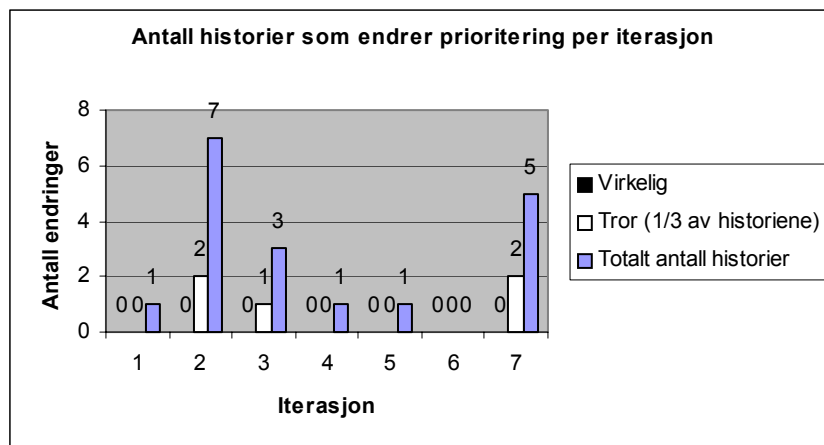
- "The planning game"
- Kodestandard
- Enkel design
- Testing

"The planning game" ble endret fordi praksisen er nært knyttet til forholdet mellom utviklere og kunde. Vi tror derfor at denne praksisen må tilpasses hvert enkelt prosjekt.

Hvor ofte forandrer kunden mening? Større endringer på historier eller prioriteringer? (QA)



Figur 33. Det ble kun en endring på funksjonaliteten gjennom hele utviklingsperioden.



Figur 34. Ingen historier endret prioritering.

Vi endte opp med ingen prioriteringsendringer og kun en endring av funksjonaliteten, som var en liten endring på brukergrensesnittet.

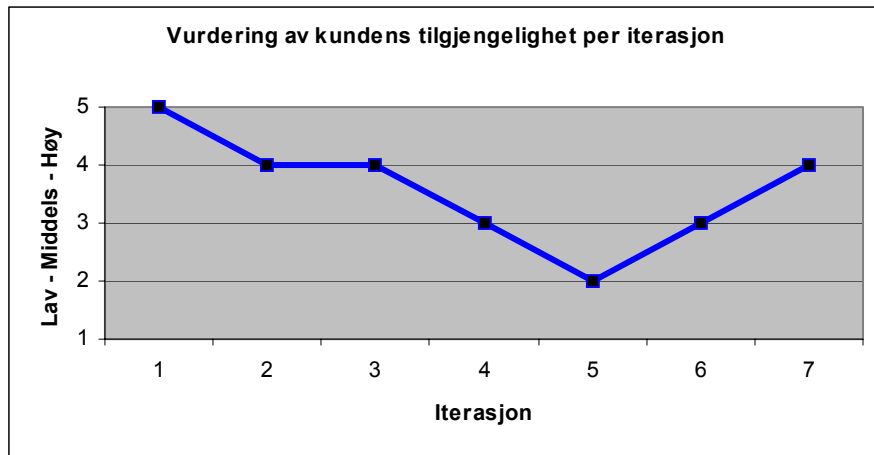
Hvorfor ble det så få endringer?

- Praksisene "The planning game" og kunde tilstede gjorde sitt til at det ble kun en endring på funksjonaliteten. Reforhandlingsmøtet i "The planning game" må spesielt fremheves

fordi det var en effektiv måte å oppdage misforståelser på. Vi hadde en god dialog med kunden og vi forstod hverandre.

- Kunden hadde datateknisk bakgrunn på lik linje med utviklerne.
- Kunden var bestemt på hvilke historier som var viktigst til enhver tid.

Kundetilgjengelighet (QB)

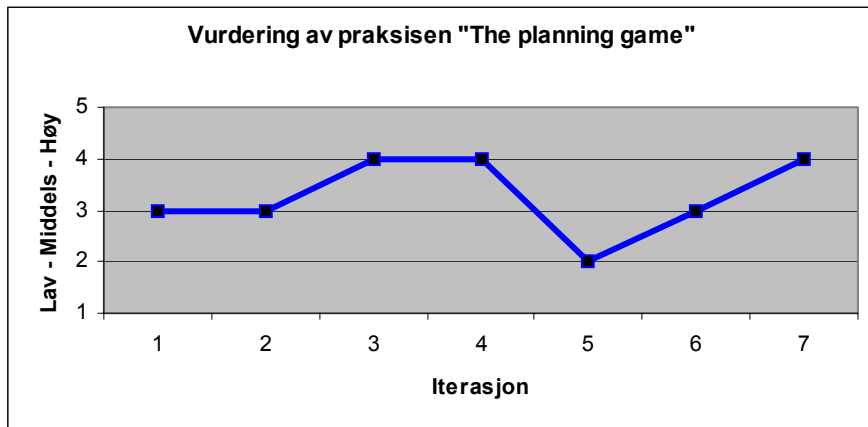


Figur 35. Kunden hadde permisjon fra jobb fra iterasjon 3 til 6.

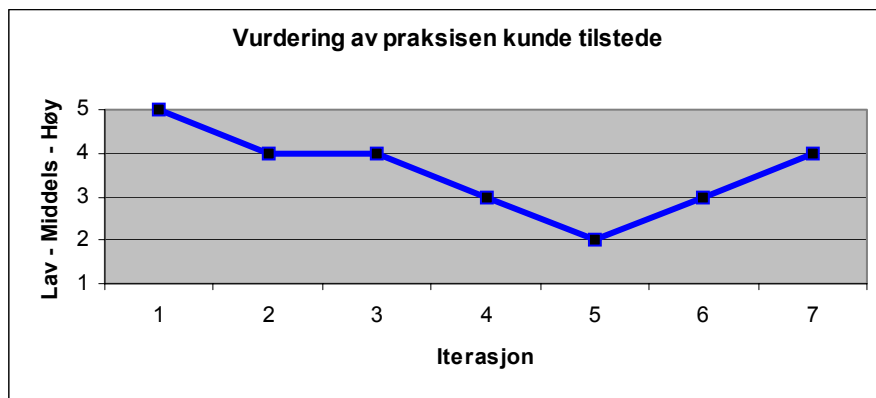
Vi trodde at kundens tilgjengelighet ville bli høy gjennom hele prosjektet. Dette ble ikke tilfellet, fordi kunden hadde permisjon fra jobb midtveis i prosjektet. Han var likevel innom kontoret vårt 1-2 ganger per iterasjon i denne perioden. Han var dessuten tilgjengelig på telefon.

4.4.4.3 Samvariasjoner

Vi fant bare samvariasjon mellom praksisene "The planning game" og kunde tilstede. Dette tyder på at en god gjennomføring av "The planning game" er avhengig av at kunden var tilstede. Blant annet skal kunden i "The planning game" prioritere historier og vurdere estimatene til utviklerne. Dette fører til at en unngår misforståelser mellom kunden og utviklerne.



Figur 36. En god gjennomføring av "The planning game" er avhengig av at kunden er tilstede.



Figur 37. Kunden hadde permisjon fra arbeid fra iterasjon 3 til 6.

4.5 Egne erfaringer

4.5.1 Egne erfaringer med XP

I løpet av prosjektet fikk vi en del erfaringer med XP:

- Det gikk bra å endre kurs underveis. Kunden ville at vi skulle ta i bruk ny teknologi i utgivelse to, slik at vi måtte kaste en del av det vi hadde gjort i utgivelse en. Dette forsinket prosjektet, men ga rom for nye løsninger.

- Det fungerte bra å lage en arkitektur tidlig i utviklingsperioden. XP er ”bunn-opp” programmering. Vi tror det er hensiktsmessig å lage en grovskisse for hele prosjektet på forhånd. Dette gjelder også til en viss grad arkitektur. Dette blir en slags middelvei mellom ”bunn-opp” og ”topp-ned”.
- Mønster kan med fordel brukes i XP. Vi brukte MVC [19], og la mye funksjonalitet i kontrolleren. I tillegg brukte vi mønsteret ”Data Access Object” [19] for å abstrahere bort databasen.
- Vi hadde god erfaring med par-programmering. Vi syntes par-programmering var en morsom måte å utvikle programvare på. Det var en effektiv praksis som gjorde at vi følte at det var fremgang i prosjektet. Par-programmering var dessuten en god måte å lære av hverandre, fordi vi diskuterte oss fram til løsninger.

4.5.2 Andre momenter vi ser med XP

Gjennom diskusjoner med veileder, studenter og ansatte ved Sintef Tele og Data ser vi også andre momenter ved XP som bør nevnes:

- Opplæring. Ved å bytte partner ved par-programmering vil prosjektdeltakerne lære seg å kjenne store deler av systemet. På den måten minskes risikoen hvis en i utviklingsteamet slutter eller er utilgjengelig.
- Pris på prosjekter og XP. Kunder kan være skeptiske til at de skal betale time for time i XP-prosjekter. Det kan være mulig å innføre fast pris også i XP-prosjekter. Eksempelvis kan en bli enig om en fast pris for prosjektet, men dersom prosjektet går over tiden, får utviklerne halv timespris i resten av utviklingstiden. Dersom de blir ferdige før tiden, deles overskuddet som en bonus.
- Hyppige partnerbytter. Det kan bli mye ”overhead” med hyppige partnerbytter i forbindelse med par-programmering. Det vil si at det kan ta tid å tilpasse seg en ny partner og bli effektiv.
- Kunden må forbli kunde og ikke en del av prosjektet. Det bør være én kunde som fungerer som et grensesnitt mot sin bedrift. Bedriften må snakke internt om endringer underveis.
- Felles eierskap. I denne praksisen ligger det at alle er ansvarlig for koden. Dette har en tendens til å ende med at ingen er ansvarlig for koden.
- XP må tilpasses hvert enkelt prosjekt og miljø. Dette har vi gjennomført.

5 Diskusjon

5.1 Diskusjon av framgangsmåte

Vi vurderte i starten to måter å gå fram på i dette prosjektet.

5.1.1 Bruke XP i hele vår del av prosjektet.

Deretter sammenlikne resultatet med det som utvikleren hos Sintef laget.

Fordeler:

En får god kjennskap til XP ved å bruke utviklingsmetoden gjennom et helt prosjekt. Det blir nok iterasjoner til at en rekker å få en del erfaringer med denne måten å utvikle programvare på. En får også tid til å endre prosessen ved å gjøre *lokale* tilpasninger og se konsekvensene av endringene.

Ulemper:

Det vil finnes en del feilkilder. De to delene av prosjektet blir utviklet av forskjellige personer med ulikt kunnskapsnivå. Utvikleren fra Sintef har bakgrunn som sivilingeniør fra NTNU. Han laget programvaren alene, mens vi er to om oppgaven. Arbeidsoppgavene i hans del og vår del vil være forskjellige.

5.1.2 Halve vår del av prosjektet med XP som utviklingsmetode, halve med en "tradisjonell" utviklingsmetode

Fordeler:

Denne framgangsmåten fører til bortfall av feilkilder som for eksempel kunnskapsnivået mellom utvikleren hos Sintef og oss.

Ulemper:

Det blir en kort utviklingsperiode med XP og dermed mindre relevant erfaring og færre målinger. Vi vil kanskje ikke rekke å gjøre lokale tilpasninger. Det blir dobbelt opp med planlegging og analyse, eksempelvis praksisen "The planning game" og kravspesifikasjon. Det vil også oppstå noen feilkilder ved denne måten å gå fram på. Rekkefølgen mellom XP og den "tradisjonelle" utviklingsmetoden vil være en faktor en må ta hensyn til. Vår erfaring er at en må bruke en del tid på å sette seg inn i problemstillinger og sette opp utviklingsmiljø før en går i gang med et prosjekt. Dette vil ikke være nødvendig i siste del.

5.1.3 Vurdering

Vi mente det var viktig at vi fikk best mulig kjennskap til å utvikle ved hjelp av XP med den tiden vi hadde til rådighet. Vi ønsket å tilpasse utviklingsmetoden etter våre egne erfaringer og dette måtte skje over en lengre periode. Dette resulterte i at vi valgte det første alternativet. En svakhet med begge framgangsmåtene er at sammenlikningsgrunnlaget er noe tynt, siden det er kun to korte prosjekter som er sammenliknet med hverandre.

5.2 Diskusjon av erfaringsbasert forbedringssystem

XP må alltid tilpasses hvert enkelt prosjekt. En måte å gjøre dette på er å bli enige på forhånd om hvordan XP skal utføres. Vi tror det er bedre å gjøre tilpasninger underveis. Vi har laget et erfaringsbasert forbedringssystem som gjør det enkelt å gjøre tilpasninger underveis i utviklingen.

Forbedringssystemet vi har kommet fram til er en oversiktelig måte å tilpasse XP-prosjekter på. Ved å bruke dette systemet vil praksisene alltid være oppdaterte og henge lett synlig på vegg. Erfaringsdelen innbyr til idémyldring som kan føre til flere forslag til å endre prosessen. Systemet kan gjenbrukes i andre prosjekter for å optimalisere prosessen. En annen fordel med dette systemet er at programmerere kan evaluere erfaringene sammen med kunden eller andre interessenter.

Det tar noe tid å bruke dette systemet. Vi har brukt til sammen 1 – 2 timeverk per iterasjon for å tilpasse vår prosess. Det vil kunne oppstå problemer med mange utviklere. En trenger kanskje fellesmøter for å bli enige om endringer av prosessen og forbedringssystemet vil ikke være like tilgjengelig for alle.

5.3 Diskusjon av resultat

Det vil alltid være usikkerhet knyttet til målinger. Målingene og resultatene vi kom frem til i forrige kapittel, ble basert på et lite prosjekt på syv iterasjoner. Vi sammenliknet del 1 og del 2 på produktkvalitet og produktivitet.

5.3.1 Produktivitet

Utvikleren hos Sintef Tele og Data produserte 7,1 linjer kode per timeverk i del 1 og vi produserte 5,5. Se tabell 14. Dette resultatet indikerer at produktiviteten ble lavere ved å bruke XP i vårt prosjekt.

	Antall linjer kode per timeverk
Del 1	7,1
Del 2	5,5

Tabell 14. Produktivitet i del 1 og 2.

Det er knyttet noe usikkerhet til disse resultatene:

- Sintef-utvikleren har laget noe duplikat kode både i Java- og JSP-filene
- De to systemene har ulik kompleksitet og teknologi
- Utviklerne har forskjellig kunnskap og erfaring
- Del 1 ble utviklet av en person, mens del 2 ble utviklet av to personer

- Utvikleren hos Sintef skrev mer JSP- og HTML kode enn det vi gjorde. Det motsatte var tilfellet med Java. Hastigheten på å utvikle ved JSP/HTML i forhold til Java kan være ulik.

5.3.2 Kvalitet på koden

Vi vurderte kvaliteten på koden til utvikleren hos Sintef Tele og Data og han vurderte kvaliteten på vår kode. For å få en objektiv vurdering burde en tredje person ha vurdert kvaliteten på koden for begge delene.

I motsetning til i del 1 inneholder ikke del 2 noe duplikat kode eller kode som ikke ble brukt. Dette indikerer at vi fikk bedre kvalitet på koden ved å bruke XP, blant annet ved bruk av praksisene par-programmering og kodenstandard. Derimot var fokus for Sintef-utvikleren å få opp en versjon så raskt som mulig.

Del 1 brukte objektorientering i liten grad mens del 2 brukte objektorientering fullt ut. Vi mener at bruken av enhetstester i XP tvang fram objektorientering, fordi det er mye enklere å lage enhetstester på en objektorientert arkitektur. Utvikleren hos Sintef Tele og Data var en tidligere C og C++ programmerer. Dette kan også være årsaken til forskjellen i bruk av objektorientering.

5.3.3 Usikkerhet knyttet til GQM-planen

Våre målinger tilknyttet timeforbruk ble rundet av til nærmeste halvtime. Dette kan ha medført små unøyaktigheter i metrikkene. I mange av vurderingene ligger det til grunn skjønsmessige vurderinger.

5.4 Hvordan sammenfaller resultatene med liknende studier?

5.4.1 "Ekstrem Programmering, Praktiske erfaringer" [21]

"Prosjektet ble kjørt som et internt prosjekt i Proxycom, og bestod i å utvikle et system der funksjonene skulle realiseres både som Web- og Windows-løsninger. Prosjektgruppen bestod av 9 personer, derav en kunde. Hensikten med prøveprosjektet var å finne ut om XP-metoden var bedre enn tradisjonelle metoder, spesielt med hensyn på tidsforbruk og kvalitet."

Positive erfaringer med XP:

- Hyppige og korte prosjektmøter er nyttige
- XP gir god kundemedvirkning
- Felleseie av programkode er fordelaktig
- Par-programmering er effektiv for opplæring og kvalitet

Negative erfaringer:

- Tidsforbruket er høyere enn tradisjonelle metoder
- Vanskelig å danne par når deltakerne jobber deltid.

Prosjektet gjennomførte tre iterasjoner og brukte totalt 496 timeverk. Trond Johansen hos Proxycom konkluderer med at tidsforbruket med XP som utviklingsmetode var omtrent 50 %

høyere enn med tradisjonelle metoder og at kvaliteten på koden ble bedre eller like god. Dette sammenfaller med våre resultater som antyder at XP er mer ressurskrevende og at kvaliteten på koden blir bedre.

5.4.2 "Strengthening the case for pair programming" [22]

I 1999 deltok 41 datastudenter i et forsøk ved universitetet i Utah. Hensikten var å undersøke gyldigheten av resultatene industrien hadde hatt med par-programmering. I løpet av forsøket fullførte studentene fire oppgaver over en periode på seks uker.

Resultatene viste at de som jobbet i par produserte bedre produkter på alle oppgavene enn de som jobbet alene. De som par-programmerte brukte 60 % flere timeverk på den første oppgaven. Etter en tilpasningsperiode falt disse 60 % til 15 %. Dersom de som jobbet alene skulle forbedret sin kode på nivå med de som jobbet i par, ville de til sammen brukt mer tid enn de som jobbet i par. Bedre produktkvalitet og større ressursbruk samsvarer bra med resultatene våre. I tillegg samsvarer det at produktiviteten økte etter en tilpasningsperiode. I utgivelse 1 var produktiviteten vår 42 % lavere enn Sintef-utvikleren og i utgivelse 2 var produktiviteten 15 % lavere. En usikkerhet knyttet til sammenlikningen av resultatene, er at studentene ikke brukte XP fullt ut.

5.5 Evaluering av eget arbeid

Hva var bra med måten vi løste oppgaven på?

Det var bra at vi brukte XP i et reelt prosjekt. Løsningen vi skulle lage var en del av et system som kunden hadde bruk for. Det som ikke var reelt, var at kunden ikke betalte utviklerne i dette prosjektet. Han betalte kun med egen innsats.

Hva kunne vært gjort annerledes?

I iterasjon en og to brukte vi til sammen 48 timeverk på konfigurering og installering av programvaremiljøet. Dette innebar å få webserver, database, editorer og samarbeidsverktøy til å fungere. Vi kunne ha gjort dette i en iterasjon null, slik at vi kunne gått rett på første historie etter "The planning game" i iterasjon en.

Hvordan fungerte XP og GQM sammen?

Det fungerte godt å bruke GQM for å måle data i prosjektet. Ved hjelp av GQM strukturerte vi informasjon om vår egen utviklingsprosess og samlet inn målinger til metrikkene hver dag. GQM feedback-møtetene med KJ var nyttige siden vi analyserte måledata og kunne foreslå forbedringstiltak underveis i prosjektet.

6 Konklusjon

Vi har brukt XP som metode i et utviklingsprosjekt. Våre resultater antyder at XP er mer ressurskrevende enn prototyping og at kvaliteten på koden blir bedre. Timeforbruket var 29 % høyere enn ved prototyping. I motsetning til vår kode, hadde utvikleren hos Sintef noe duplikat kode og kode som ikke ble brukt. I tillegg var ikke koden konsekvent med tanke på objektorientering. Vi tror at praksisene testing og refaktorisering har gjort at vi har unngått disse problemene.

Vi har tilpasset XP som utviklingsmetode og funnet vår måte å utføre XP på i dette prosjektet. Vi har laget et erfaringsbasert forbedringssystem for å tilpasse XP-prosjekter. Både våre tilpasninger til praksisene og forbedringssystemet kan være nyttige inndata i liknende prosjekter.

Målingene fra GQM-planen antyder at det er samvariasjon mellom praksisene ”The planning game” og kunde tilstede. Vi tror at en god gjennomføring av ”The planning game” er avhengig av at kunden er tilstede.

Som helhet fremstår ekstremprogrammering som noe nytt med sitt syn på programmering som en menneskelig aktivitet og sterke fokus på koden som skrives. Ved å samtidig forenkle planleggingen og kommunikasjonen mellom prosjektdeltakere, kan en få et prosjekt med bedre produktkvalitet, men med et noe større ressursbruk

7 Videre arbeid

Det hadde vært interessant å prøve et liknende prosjekt med den andre framgangsmåten som vi skisserte i kapittel 5.1, Halve del 2 av prosjektet med XP som utviklingsmetode, halve med en "tradisjonell" utviklingsmetode.

Det kunne også vært interessant med et prosjekt der en bruker XP over Microsoft NetMeeting. En utfordring ville da være å par-programmere over nett.

I denne oppgaven har vi sammenliknet XP med prototyping. Framtidige case-studier kan sammenlikne XP med andre "tradisjonelle" utviklingsmetoder.

8 Referanser

- [1] Dybå T., Wedde K, J., Stålhane T., Moe N. B., Conradi R., Dingsøy T., Sjøberg D., Jørgensen M. *SPIQ - Software Process Improvement for better Quality* Metodehåndbok, versjon 3.0, 14. januar 2000
<http://www.Geomatikk.com/spiq/>
- [2] JUnit
<http://www.junit.org/index.htm>
- [3] Rausand M., *Risikoanalyse, Veiledning til NS 5814*. Tapir forlag, 1991.
- [4] Mollan Ø., Juvik T, Lefstad O. J., *The GQM tool*. NTNU, IDI, høstprosjekt 2001
- [5] Agile Insight
palm@tihlde.org
rolvinge@stud.ntnu.no
- [6] Beck K., *Embracing change with Extreme Programming*. Computer , Volume: 32 Issue: 10 , Oct. 1999, Page(s): 70 –77
- [7] Beck K., *Extreme Programming Explained*. USA, Addison-Wesley, 2000
- [8] Xprogramming.com – *an Extreme Programming Resource*
<http://www.xprogramming.com/>
- [9] Cockburn A., *Selecting a project's methodology*. IEEE Software, Volume: 17 Issue 4, July-Aug. 2000, Page(s): 64-71
- [10] Highsmith J. *Extreme programming*. E-business application delivery, vol. XII, no. 2. Cutter consortium, februar 2000.
- [11] Beck K., Fowler M., *Planning Extreme Programming*. USA, Addison-Wesley, 2001
- [12] Auer K, Miller R., *Extreme Programming Applied*. USA, Addison-Wesley, 2002.
- [13] Thomas Flemming, XP kurs, 29.1.2002
KPNQwest
- [14] Stornes S., Sandnes J. B. *Utvikling av en prototyp av en universell profilløsning ved bruk av en personlig sekretær*. NTNU, IDI, diplomoppgave 2002
- [15] Christensen T., *Introduction to Extreme Programming (XP)*.

Kursmateriell.

<http://www.nordija.com>

- [16] Dingsøy T., Hanssen G. K. *Extending Agile Methods: Postmortem Reviews as Extended Feedback*. Submitted to Learning Software Organizations Workshop. Chicago, USA, 2002.
- [17] Astra QuickTest
<http://www.mercuryinteractive.no>
- [18] Hauge H. J. *How to Combine Software Process Improvement and Quality Assurance with Extreme Programming*.
NTNU, IDI, diplomoppgave 2002
- [19] Fløysand C., Frisk C., Indal E. *eCourse: support for software architecture teaching*.
<http://www.idi.ntnu.no/~letizia/swarchi/eCourse.html>
- [20] L. Crispin, T. House, C. Wade. *The Need for Speed: Automating Acceptance Testing in an Extreme Programming Environment*.
2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), May 2001.
<http://www.xp2001.org/xp2001/conference/papers/Chapter23-Crispin+alii.pdf>
- [21] Johansen T., *Ekstrem Programmering, Praktiske erfaringer*.
Proxycom AS, versjon 1.0, februar 2002
- [22] Williams L., Kessler R. R., Cunningham W., Jeffries R. *Strengthening the case for pair programming*.
IEEE Software, Volume: 17 Issue: 4 , July-Aug. 2000, Page(s): 19 –25
- [23] Extreme Programming FAQ
<http://www.jera.com/techinfo/xpfaq.html>
- [24] Extreme Programming: A gentle introduction
<http://www.extremeprogramming.org>

Web-referansene var gyldige da denne rapporten gikk i trykken.

9 Ordforklaringer

Bruker	En som benytter levert programvare, for eksempel en firmaansatt eller en privatperson.
Erfaringer	Kan være rå, innsamlede måledata (som antall feil eller kostnader) eller prosjektoppsummeringer ("post-mortems").
Feedbackmøte	Avholdes i gitte intervaller under et prosjekt. Formålet er å grovanalysere, presentere og tolke siste periodes måledata og andre resultater. Andre formål er å opprettholde motivasjon og eventuelt å justere metrikker og forbedringstiltak.
GQM	Goal Question Metric. Metode for å definere minimale og relevante metrikker ved et gitt forbedringstiltak, som ledd i å etablere kvantitative suksesskriterier. En starter med en konkret målsetting til forbedringstiltaket. Deretter identifiseres relevante spørsmål for å finne ut om målsetningen er nådd. Basert på spørsmålene kan en foreslå måledata og tilhørende metrikker, slik at spørsmålene besvares.
KJ	KJ er en idémyldringsmetode der ideer skrives ned på selvklebende lapper som festes på en tavle i tilfeldig orden. Deretter grupperes lappene for å finne underliggende relasjoner som binder gruppene sammen.
Kunde	Den som betaler for levert programvare, for eksempel et firma eller distributør.
Kvalitet	"Helheten av egenskaper en enhet har, og som vedrører dens evne til å tilfredsstillte uttalte eller underforståtte krav" (NS-ISO-8402 – Quality Vocabulary, 1996. I SPIQ: "Egenskaper ved produkter og tjenester som gir fornøyde kunde og brukere over hele produktets/tjenestens levetid – dvs. kunden og brukeren har alltid rett!")
Metrikk	Samling av egenskaper (attributter) og deres definisjoner som til sammen karakteriserer et programvareprodukt og/eller en programvareprosess, sett i forhold til et formål og en konkret omgivelse. For hver egenskap angis navn, måleskala og retningslinjer for datainnsamling- og analyse. Metrikker kan være direkte eller indirekte, objektive eller subjektive.
"Spike"	En "spike" er et enkelt program for å utforske en potensiell løsning på et problem. Programmet adresserer kun problemet en undersøker og er ikke en del av produksjonskoden. "Spiking" omfatter også å lete etter løsninger i diskusjonsgrupper, bøker etc.
"Tradisjonelle" utviklingsmetoder	Dette er metoder som for eksempel vannfallsmodellen og prototyping som har henholdsvis ingen eller lange sykluser. Metodene forutsetter en

kravspesifikasjon. Kunden har få eller ingen muligheter til å gjøre endringer på krav underveis i prosjektet.

Refaktorisere Refaktorisere er en teknikk programvareutviklere bruker for å forbedre kode uten å endre funksjonalitet. Et eksempel kan være å gjøre koden mer lesbar, mer modifiserbar etc.

XP Ekstremprogrammering

10 Vedlegg

- A GQM-plan
- B Oppsummering av data i tabellform
- C Oppsummering av data med figurer og forklaringer
- D Praksisbeskrivelser – utgangspunkt og etter iterasjon 7
- E Kodestandarder
- F Risikoanalyse
- G Tidsplan
- H Akseptansetester
- I Utgivelsesoppsummeringer
- J Rapporter fra feedbackmøter
- K Ukesoppsummeringer
- L Arbeidskisse av arkitektur i del 1 og del 2
- M Opprinnelig oppgavetekst

CD:

- N Kode
- O Dagbok
- P Presentasjoner
- Q Praksisbeskrivelser
- R Historiene som ble fullført