Endrings- og konfigurasjonsstyring

Notater til forelesning INF1050, Universitet i Oslo
21.april 2009
Hans Christian Benestad
Simula Research Laboratory

1. Motivasjon

Kurset har så langt formidlet anerkjente prinsipper for å bringe brukeres og andre interessenters *behov* (uttrykt som funksjonelle og ikke-funksjonelle *krav*) over til et programvaresystem som tilfredsstiller disse behovene. Den typiske hverdagen for dagens systemutviklere karakteriseres imidlertid i liten grad av blanke ark (eller blanke editorvinduer). Minst 50% av all ”programvareutvikling” globalt er *evolusjon*, dvs. *vedlikehold* og *videreutvikling* av eksisterende programvare [1]. Denne andelen vil sannsynligvis bare øke, i takt med at programvare blir allstedsnærværende.

Endringshåndtering og versjonskontroll er en systemutviklingsdisiplin som beskriver hvordan programvareevolusjon kan styres på en effektiv måte. Noen overordnete mål for disiplinen er:

- At utviklingsorganisasjonen gjennomfører systemendringer i takt med interessentenes kontinuerlig endrende behov.
- At programvaren kan eksistere i ulike versjoner og varianter uten at dette har negative konsekvenser for kostnader eller kvalitet på leveranser.

Endringshåndtering og konfigurasjonsstyring kan sees på som en egen disiplin, men det er mange sammenhenger med tidligere formidlet stoff. For eksempel:
- Både kravhåndtering og test utføres som en integrert del av endringshåndteringen.
- Initiell arkitektur og design påvirker enkelheten i å videreutvikle programvaren. Et viktig ikke-funksjonelt krav i designfasen er nettopp *endringsbarhet*.

2. Bakgrunn - programvareevolusjon

For 30 år siden formulerte Manny Lehma sin første ”law of software evolution” [2], basert på et langvarig studium av IBM OS/360.

> “*Software systems must be continually changed else they become progressively less satisfactory in use”*

Litt forenklet kan Swanson’s ”dimensions of software maintenance” [3] fungere som forklaringer på det kontinuerlige behovet for endringer:

1. *Korrektive endringer* er nødvendig for å rette opp feil som ubønnhørlig vil bli gjort i utviklingsprosessen
2. *Adaptive endringer* er nødvendig for å tilpasse systemet til endrede kjøreomgivelser (f.eks. støtte en ny browser, operativsystem eller database)

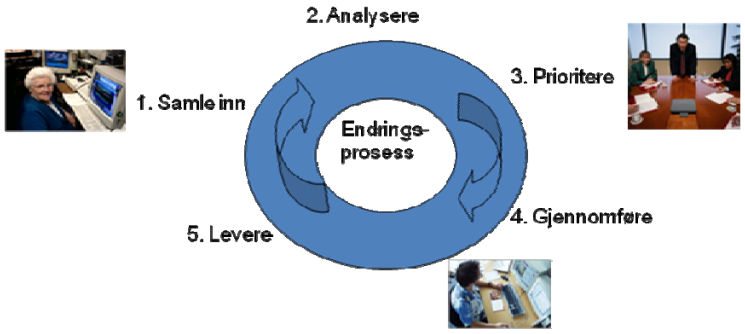
3. *Forbedrende (enhancive) endringer* er nødvendig for å tilfredsstille nye krav fra programvarens interessenter (stakeholders)

Effektiv håndtering av endringer blir stadig mer kritisk. Dette skjer i takt med følgende trender i dagens systemutvikling:

- Siden programvare er blitt allstedsnærværende, så vil tradisjonell nyutvikling bli stadig sjeldere. Typisk vil nye prosjekter måtte håndtere og gjerne fornye og videreutvikle en *programvarearv* (legacy systems).
- Programvare utvikles i økende grad som deler i et allerede større system av løst koplete komponenter – alt skal fungere sammen. Såkalt *nyutvikling* må skje på premissene av dette større systemet.
- Populære, smidige utviklingsprosesser beskriver at all utvikling i prinsippet er evolusjon, og bør organiseres rundt endringer [4].

3. Endringsprosessen

Figur 1 illustrerer en generisk endringsprosess.



Figur 1. Endringsprosessen

Samle inn. Aktiviteten henspeiler på å kommunisere med systemets interessenter for å samle inn endringsønsker og nye behov som måtte oppstå. Mer om dette er beskrevet i ”Kravhåndtering og evolusjon” i seksjon 5.

Analysere. Et endringskrav fra en bruker må analyseres av utviklingsgruppa. Kanskje har brukeren misforstått, og det er hjelpefunksjonen som har forbedringspotensiale i stedet for programvaren? Kanskje strider forslaget mot gjeldende produktstrategi? Eller kanskje det rett og slett er et kjempegodt forslag. På dette stadiet bør utviklere med kjennskap til berørt funksjonalitet analysere hvorvidt endringen påvirker andre funksjoner eller kvaliteter, og skissere hvordan eventuelle sideeffekter kan håndteres. Analysen munner gjerne ut i et estimat for endringsoppgaven.

Prioritere. Det er prosjektleders ansvar å sørge for fornuftige prioriteringer mellom foreslåtte endringsoppgaver. Kriteriene kan variere. I smidige metoder anbefaler man å prioritere etter ”business value”. Andre kriterier kan fokusere på *risikofaktorer* eller helt pragmatiske hensyn som ”Vi er nødt til å blidgjøre kunde Y”.

Gjennomføre. Her utfører utvikleren fysiske endringer i programvarens komponenter. Konfigurasjons- og versjonskontroll er sentralt i denne aktiviteten.

Levere. En samling endringer vil inngå i en ny *release* av et programvaresystem. Det er viktig å ha gode rutiner for å teste (spesielt regresjonsteste) slike relaser, og ellers sørge for at installasjon/idriftssetting av ny versjon skjer smertefritt og automatisk som mulig, uten fare for tap av eksisterende data.

Alle disse aktivitetene er kritiske. Det er nok å bomme på en av dem for å sette programvarens troverdighet og levedyktighet i fare.

4. Formalisme og verktøy for endringshåndtering

Endringsprosessen som beskrevet over kan eksistere uformelt, det vil si at det er en ikke-nedskrevet enighet i prosjektgruppa om at det er slik prosessen fungererer. Kanskje mottas endringsforespørsler på telefon eller email, prosjektleder samler dem i et regneark, og fordeler oppgaver i standup-møter. En slik prosess vil fungere i enkle tilfeller. Mer formalisme (for eksempel nedskrevne rutiner) og bedre verktøy er nødvending med:

- Større utviklingsgruppe
- Geografisk distribuert utviklingsgruppe
- Flere kunder og brukere
- Flere varianter av produktet
- Programvaren er del av et større system f.eks. et air traffic control system eller inkapslet programvare i mobiltelefoner
- Sikkerhetskritisk eller forretningskritisk programvare

5. Knytninger mot andre disipliner

Kravhåndtering og evolusjon

Kurset har tidligere behandlet disiplinen *kravhåndtering*. Kravhåndtering er også en essensiell del av videreutikling. Viktige momenter ved kravhåndtering under initiell utvikling så vel som under videreutvikling er:

- Brukerne må motiveres til å formulere krav, særlig de grunnleggende som avgjør systemets suksess
- Brukere må ha en velfungerende kommunikasjonskanal inn mot utviklergruppa.
- Utviklergruppa må analysere endringskravet i lys av gjeldende produktstrategi (som bør jevnlig gjennomgås)

Smidige metoder foreslår at praksisen *on-site customer* sikrer flyten av krav mellom brukere og utviklere. En problemstilling er at denne on-site customer har en tendens til å bli påvirket av utviklermiljøet, og derfor etterhvert slutter å tenke på systemet fra den typiske brukers premisser. For kompliserte systemer er det også litt naivt å forutsette enighet om krav i et stort brukermiljø, og at en eller noen få personer kan inneha all nødvending kunnskap knyttet til nye krav.

Arktitektur og evolusjon

Som nevnt innledningsvis er det normalt et viktig ikke-funksjonelt krav at programvaren skal være enkel å endre. Anerkjente design-prinsipper som lagdeling, innkapsling, veldefinerte grensesnitt og"design patterns" [5] er ment å bidra til enklere endringer. I praksis er det lettere sagt enn gjort å forberede et system for evolusjonsfasen:

- Endringsbarheten er avhengige av hvilke konkrete endringer som forekommer. Disse kan være vanskelig å forutsi under initiell utvikling (men man kan ihverfall prøve).
- Ulike programmerere oppfatter ikke nødvendigvis endringsbarhet på samme måte. Hvis man bruker avanserte "design patterns" må man være ganske sikker på at framtidige vedlikeholdere har kunnskapen som skal til for å forstå og utnytte disse [6]. Hvis man ikke klarer å opprettholde velmente, men avanserte designprinsipper, vil man få problemer. KISS (keep it simple stupid) er et pragmatisk og godt designprinsipp.

Test og evolusjon

Iterative systemutviklingsmetoder anbefaler hyppige leveranser, for eksempel månedlig. Hvis testing for disse hyppige leveransene foregår helt manuelt og etter innfallsmetoden vil man sannsynligvis oppnå enten lav kvalitet eller et uakseptabelt høyt tidsforbruk. For å understøtte hyppige leveranser kan automatiserte rutiner innarbeides. Verktøy som JUnit og Fitness er mye brukt i industrien for å automatisere testing. Selv om disse verktøyene er nyttige, så er det mange problemstillinger rundt testing som ikke løses av disse, slik som å finne et godt nok sett av testtilfeller og testdata.

Automatiserte tester vil måtte endres etterhvert som programvaren videreutvikles. Det betyr at test-suiter (altså samlingen av testtilfeller – eksekverbare ellller ikke) også bør legges under versjonskontroll.

Kontraksformer og evolusjon

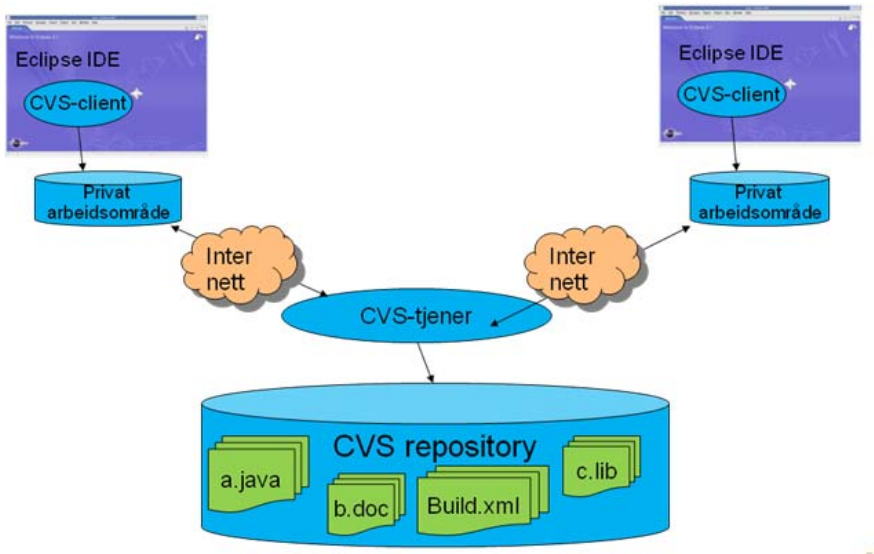
Dersom man utvikler et system for en bestemt kunde så er det utfordrende å etablere kontraktsforhold som ivaretar begge partenes interesser. De fleste kontraksformer forutsetter et relativt veldefinert omfang, samtidig er omfanget av videreutvikling og vedlikehold ofte vanskelig å forutsi.

I praksis løses dette ofte med en kombinasjon av fastpris og timepris, hvor partene løpende avklarer hvor aktivitetene tilhører. Slike kommersielle avklaringer må altså intergreres i endringsprosessen. Uansett kontraktsforhold er nødvending at samarbeidsklimaet er preget av gjensidig tillit.

Faren for *lock-in* (at man blir bundet til en leverandør eller et system) er en viktig problemstilling for kunde/oppdragsgiver. Miljøet som i utgangspunktet utviklet et system vil ofte peke seg ut som den naturlige leverandøren også for videreutvikling. Det er imidlertid gunstig å opprettholde en reell konkurransesituasjon, også for vedlikeholdsfasen. For å muliggjøre dette er det lurt allerede under utviklingsfasen å sette krav til *vedlikeholdbarhet* av systemet, slik at man ikke låser seg til leverandører eller enkeltpersoner.

6. Versjonskontrollverktøy

Versjonskontrollverktøy – arkitektur



Figur 2. Eksempel på arkitektur for versjonskontrollverktøy

Så godt som alle dagens versjonskontrollverktøy har en klient-tjener arkitektur i henhold til eksemplet i figur 2. Hele versjonshistorikken ligger lagret i et *repository*. Dataene i repository lagres gjerne på flate filer eller i en SQL-database, men all opphenting og lagring av versjoner og tilhørende meta-informasjon gjøres via en versjonskontrolltjener. Denne tjeneren tilbyr et sett funksjoner/kommandoer som kan nås fra versjonskontroll-klienter. Disse klientene kommuniserer med tjeneren over nettet, enten over et internt bedriftsnett, eller over internett.

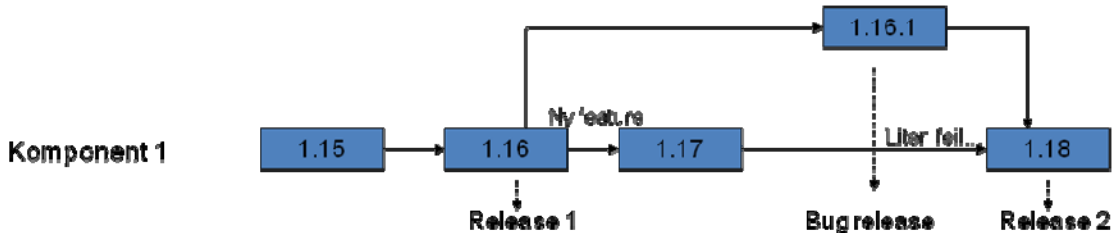
Versjonskontollklientene er ofte integrert i IDE'er (integrated development environment) slik som Eclipse. Alternativt brukes kommandolinjebaserte klienter, eller klienter intergrert med operativsystemets filutforsker.

En grunnleggende funksjon som gjøres fra VCS-klienten er å opprette et privat arbeidsområde. Dette vil typisk inneholde en bestemt versjon av hver komponent i systemet. Tegningen viser hvordan privat arbeidsområde er adskilt fra det sentrale repository. Det er viktig å forstå forskjellen på disse områdene.

7. Grunnleggende om konfigurasjons- og versjonsstyring

Versjoner og versjonstrær

Hver komponent som er underlagt versjonskontroll vil ha et versjonstre.



Figur 3. Komponentversjoner av en komponent i et versjonstre

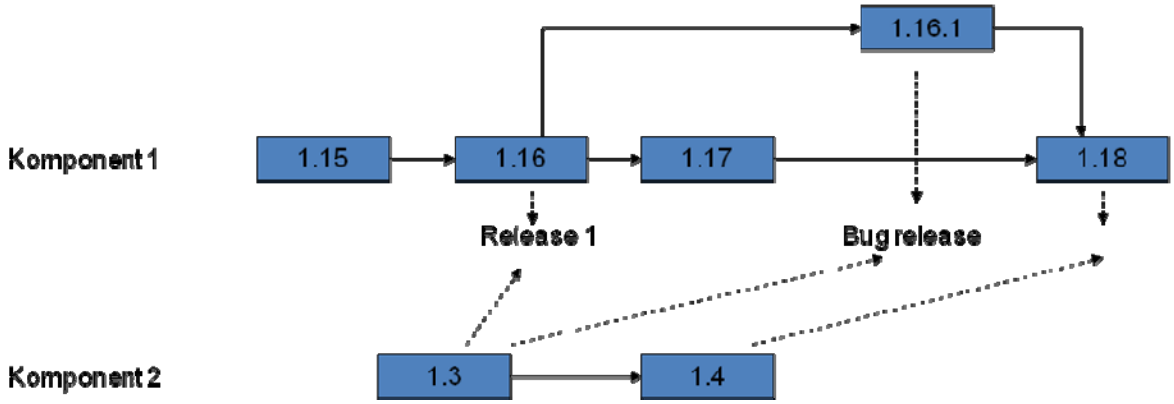
Når en utvikler har gjort en lokal endring på en fil, ønsker han/hun å gjøre endringen synlig for andre i prosjektet. Dette gjøres ved å utføre en *check-in*, eller *commit*, mot repository. For hver check-in/commit lagres en ny versjon av komponenten.

Vanligvis vil ”treet” bare være en rett linje (*mainline*, *trunk*), men noen ganger kan forgreininger være nødvendige. Eksempelvis, gitt eksemplet over

- Versjon 1.16 av ”Komponent 1” var del av første release (Release 1) av produktet
- Utviklingsgruppa jobber så videre mot Release 2, men samtidig blir det meldt inn en feilmeldig som viser seg å berøre Komponent 1.
- For å kunne gjøre parallell utvikling av ny funksjonalitet og rette feilen, lages det en *branch.* Feilen rettes i versjon 1.16.1
- Siden det haster å slippe en feilrettingsrelease lages det nå en ny versjon av produktet, hvor versjon 1.16.1 av Komponent 1 inngår
- Man ønsker selvsagt ikke at feilen skal gjenoppstå i Relase 2. Endringen i 1.16.1 må derfor flettes med 1.18

Konfigurasjon og Release

I en *konfigurasjon* inngår nøyaktig én versjon av hver komponent. Noen konfigurasjoner pekes ut til å slippes til brukermiljøene, de utgjør altså *release*’er.



Figur 4

Figur 4 viser at i Release 1 så inngikk versjon 1.16 av Komponent 1 og versjon 1.3 av Komponent 2. Versjonskontrollverktøy vil huske denne informasjonen, slik at man på et

senere tidspunkt kan *gjenskape* denne konfigurasjonen/releasen. Som det framgår av eksemplet var det nødvendig å gjenskape Release 1 for å kunne utføre en feilretting, og slippe en ”Bug release”. Versjonskontrollverktøyet holder så orden på hvilke versjoner som inngikk i ”Bug release”, for potensielt å gjøre endringer med utgangspunkt i denne.

Et litt større system inneholder tusenvis av komponenter. Man kunne sikkert klart seg med manuelle prosedyrer i eksemplet over, men for relle systemer er versjonskontrollverktøy en absolutt nødvendighet. Det er vel kanskje det eneste standpunkt angående systemutviklingsverktøy som det er universell enighet om!

Varianter og versjoner

Begrepet *versjon* brukes også om hele systemet, og ikke bare om en versjon av en enkeltkomponent. En versjon av et system vil da tilsynelatende være det samme som en *release*, men det kan være en forskjell:

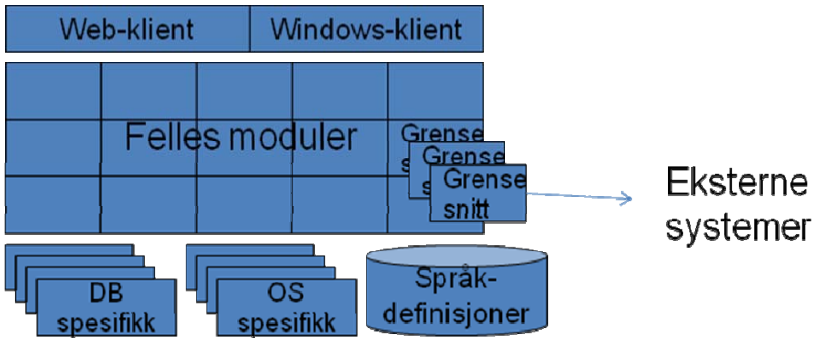
Et system kan også eksistere i ulike *varianter*. Mens to versjoner følger etter hverandre i tid (den ene er en videreutvikling av den andre), så vil to *varianter* inneholde ekvivalent funksjonalitet, men vil være tilpasset ulike *miljøer*. ”Miljø” i denne sammenhengen kan være både teknisk miljø (altså Linux vs. Windows eller MySQL vs. Oracle RDBMS) og brukermiljø (altså norsk vs. engelsk eller prøveversjon vs. full versjon). En konfigurasjon og en release er derfor en gitt variant av en gitt versjon. I praksis er begrepsbruken enda mer forvirrende og flertydig, så her må et prosjekt bare enes om begrepsbruk☺.

Krav om mange varianter setter ekstra store krav til et gjennomtenkt opplegg for konfigurasjonsstyring i et prosjekt. Selv om versjonskontrollverktøy teknisk sett ikke har noen problemer med å håndtere parallelle varianter av kildefiler, så vil utstrakt bruk av parallelle varianter skape trøbbel (blant annet fordi mye kode da vil dupliseres og dermed medføre dobbelt vedlikehold).

Krav til å tilby programvaren i ulike varianter er et viktig ikke-funksjonelt krav som kan ha stor betydning design av systemet. Mulige strategier for å håndtere varianter er følgende:

- Dynamisk, kjøretidskonfigurasjon. Dette betyr at programmet er laget for å la seg konfigurere i kjøretid, f.eks. ved at språkspesifikke tekster leses inn i minne ved oppstart, og programmet benytter seg siden av disse tekstene. Et annet eksempel er at forskjellen på ”Enterprise edition” og ”gratisversjon” kun er basert på en informasjonsbit, lagret på et eller annet lurt vis. Bruk av plattformuavhengig teknologi, slik som java, er også en måte å tilpasse seg omgivelsene i kjøretid.
- Statisk, design-tids-konfigurasjon. Dersom programvaren benytter seg av spesialiteter i forskjellig operativsystemer, så kan det være nødvendig å gjøre variantering på kildekodenivå. Det er i så fall viktig å lage arkitekturen slik at OS-spesifikk kode holdes i atskilte, veldefinerte moduler. Et godt designet byggesystem (vi kommer til dette) kan så plukke ut den rette varianten (og versjonen) når en spesifikk release skal lages.

En generisk god arkitektur, basert på 3-lags arkitekturmodellen, som muliggjør variantering er:



Figur 5. En arkitektur som muliggjør varianter

Oppsummering av vanlige versjonskontroll-operasjoner

Etablering av arbeidsområde (workspace/view). For å separere hver enkelts pågående kodeendringer fra andres, etableres private arbeidsområder. Dette arbeidsområde inneholder en lokal kopi av alle filer. Her kan utvikler boltre seg med utvikling, bygging og egen test uten at det har konsekvenser for andre. Typisk vil et versjonskontrollverktøy ha funksjoner for å hente ut, fra en sentral database (repository), en gitt konfigurasjon (som kan spesifiseres med en verktøyavhengig syntaks).

Innsjekk (Commit/checkin). Når utvikler har endret, bygget og testet vil han ønske å publisere endringene slik at andre utviklere får tilgang til dem. Det gjøres ved å sjekke inn endringene, som betyr at endringene lagres i repository. For at andre skal få tilgang til disse endringene, må de enten gjøre en eksplisitt kommando, *update*, på sitt arbeidsområde, eller de blir automatisk tilgjengelige gjennom såkalte dynamiske views (finnes i f.eks. IBM Rational Clearcase)

Håndtering av konflikter. Når man bruker verktøy uten låsing ved utsjekk så er det ingen kontroll på at flere utviklere gjør endringer samtidig på samme fil, i sine respektive arbeidsområder. Det vil først person 2 merke når han/hun prøver å sjekke inn sine endringer etter at person 1 har sjekket inn sine. Da vil verktøyet prøve å flette endringene så gått det kan, og varsle om eventuelle konflikter (som må løses opp i manuelt). En annen modell er at systemet sperrer for parallell oppdatering. Dette er den tradisjonelle tilnærmingen, men den ”optimistiske” tilnærmingen har vist seg å fungere overraskende bra i populære verktøy slik som CVSog Subversion.

Utsjekk (checkout). I pessimistiske versjonskontollsystemer så betyr utsjekk at filen gjøres skrivbar og låses for oppdatering av andre brukere. Denne kommandoen finnes ikke i ”optimistiske” verktøy (men Subversion har en lignende lock-funksjon)

Forgrening (branch). Dersom det er behov for at en komponent skal kunne eksistere i to parallelle varianter kan man utføre *forgrening*. Forgrening gjøres altså individuelt per komponent, men noen verktøy har støtte for å spesifisere at komponenter skal automatisk forgrenes ved utsjekk. Dette er praktisk dersom man jobber mot en bestemt feilrettings-release med feilrettinger i mange komponenter.

Fletting (merging). Dersom man har gjort en forgrening vil man oftest, på et eller annet senere tidspunkt, ønske å flette grenen tilbake til *mainline*. En typisk situasjon er at man skal slippe en videreutviklet release, men der man også ønsker å ta med feilrettingsendringer gjort i en forgrening. Med *3-veis merge* sammenlignes siste versjon i hver branch med versjonen ved forgreningspunktet (felles ”anchestor”). Denne algoritmen vil ofte kunne ordne opp i parallelle endringer i to brancher av en fil, men noen ganger er det nødvendig å løse opp i konflikter manuelt.

Labeling. Denne operasjonen tilordner et navn til alle komponentversjoner i en gitt konfigurasjon (typisk den utvikleren har i sitt arbeidsområde). Et typisk eksempel er at siste versjon i Mainline blir navngitt til ReleaseX, i det man har gjort ferdig alle kodeendringene for ReleaseX. Da vil det senere være enkelt å hente ut akkurat de komponentversjonene som inngikk i ReleaseX.

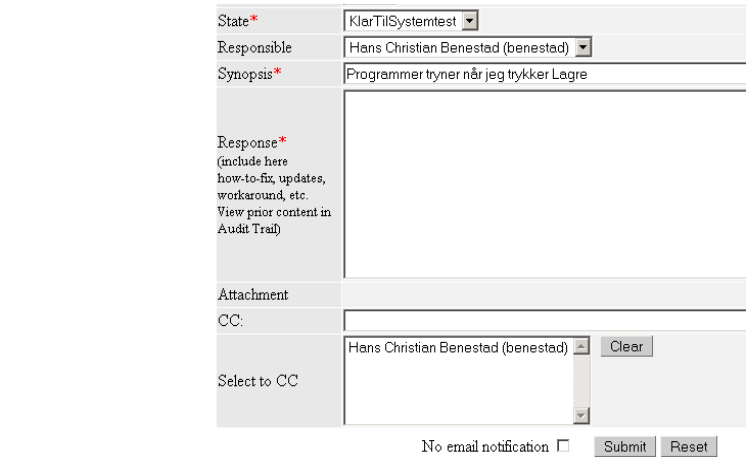
Bygging. Programsystemer består av enkeltkomponenter i en avhengighetsgraf. For eksempel: En Java kildefil kompileres til en klassefil. Klassefiler samles i biblioteker (.jar). Dersom en java-fil endres må man derfor sørge for at filen blir kompilert og inkludert i et oppdatert bibliotek. Såkalte ”byggeregler” kan defineres i verktøy som Ant og Make, og sørge for at disse stegene automatisk blir gjennomført. For større systemer er det en nødvendighet at byggeregler er definert, og at man ikke er avhengig av å gjennomføre mange manuelle byggetrinn. Byggeregler vil endre seg over tid, og det er derfor lurt å versjonskontrollere byggereglene – dette muliggjøre automatisk bygging også av tidligere versjoner. Merk at for å få til repeterbar bygging av et system er det nødvendig å versjonskontrollere alle komponenter som påvirker det endelige resultatet, inkludert standardbiblioteker, generatorverktøy, og dokumentasjon.

8. Endringshånderingsverktøy

En annen type verktøy, endringshåndteringsverktøy (bug tracker/issue tracker/ticket system) brukes for å holde orden på endringsforespørsler, og deres avklaringer, estimater, status, ansvarlig og evt. andre brukerdefinerte attributter, se figur 6. For prosjektledere fungerer disse verktøyene som prosjektstyringsverktøy for planlegging og oppfølging av løpende utviklingsoppgaver. For utviklerne fungerer de som en to-do liste, med beskrivelser av forestående arbeidsoppgaver. For eksterne interessenter fungerer de som en postboks der man kan sende endringsønsker.

Det er naturlig med en tett integrasjon mellom versjonskontrollverktøy og endringshåndteringsverktøy. For eksempel er det ofte nyttig å vite hvilken ”logisk endring” som ga opphav til konkrete kodeendringer som man ser i versjonskontrollverktøyet. Med slike koplinger er det lettere å forstå hvorfor kodeendringene ble gjort. Mange verktøy har åpninger for slik sammenkopling.

Noen IDE’er, slik som Eclipse, gjør det mulig å integrere versjonskontroll og endringshåndtering direkte i utviklingsmiljøet.



Figur 5. Et endringshåndteringsverkøy

Noen typiske scenerier ved bruk av et slikt system er:

Bruker:	Opprette ny endringsforespørsel
Bruker, Utvikler:	Sjekke status og historikk på en gitt endringsforespørsel
Bruker, Utvikler:	Finne ”mine endringer”
Prosjektleder:	Finne antall gjenstående endringsønsker for neste release
Utvikler:	Angi at en endring er ferdig implementert

… og mange andre varianter, knyttet til registrering, søk og oppdatering av endringer.

9. Oppsummering

En enighet om endringsprosess (formalisert eller ikke) gjør livet enklere for prosjektet og utviklerne og understøtter effektiv videreutvikling og kvalitet i leveranser. Grad av formalisme og bruk av verktøy må tilpasses prosjektets behov. Gode verktøy (kommersielle og open source) finnes både for håndtering av endringsønsker og for versjonskontroll. Verktøyene har et felles sett av kjernefunksjoner (beskrevet i dette dokumentet), men ulike verktøy har ulik avveining av enkelhet mot støtte for mer avanserte behov.

Referanser

[1] A. April and A. Abran, *Software Maintenance Management*. New Jersey: John Wiley & Sons Inc., 2008.

[2] M. M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proceedings of the IEEE*, vol. 68, pp. 1060-1076, 1980.

[3] E. B. Swanson, "The Dimensions of Maintenance," in *2nd International Conference on Software Engineering*, San Francisco, California, United States, 1976, pp. 492-497.

[4] K. Beck, "Embracing Change with Extreme Programming," *Computer,* vol. 32, pp. 70-77, 1999.
[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.
[6] E. Arisholm and D. I. K. Sjøberg, "Evaluating the Effect of a Delegated Versus Centralized Control Style on the Maintainability of Object-Oriented Software," *IEEE Transactions on Software Engineering,* vol. 30, pp. 521-534, 2004.