Förändringskontroll i XP-team

Love Johansson (d00lj),
Joakim Persson (d00jp)

21 februari 2005

Sammanfattning

Under sju veckor har vi agerat coacher åt en grupp relativt oerfarna programmerare i en större projektkurs vid LTH. Under kurserns gång har vi sedan studerat hur väl XP-metodiken stödjer Software Configuration Management-tekniker. Vi har baserat undersökningen på tidigare forskning inom området, och utvidgar nu dessa med en praktisk observationsstudie. Vi har funnit att de flesta SCM-tekniker kan tillämpas naturligt i ett litet XP-projekt, även om det inte formellt ingår i XP-metodiken. Andra tekniker skalas inte ner enkelt till projekt av den typen som XP stödjer, eller den förändringsmetodiken som XP är, utan är mer tillämpbara på projekt av mer traditionell karaktär.

Innehåll

1 Inledning **2**

2 Bakgrund **3**
2.1 Metod 3

3 Teori **4**
3.1 XP-practices i projektet 4
3.2 CM-practices i XP-kontext 4

4 Substans **6**
4.1 Incremental refactoring 6
4.2 Impact analyse refactorings 6
4.3 Use copy-merge work model 7
4.4 Impact analyse stories as part of planning game 7
4.5 Physical audit in the release process 8
4.6 Define configuration items and their structures 8
4.7 Trace changes to stories 9
4.8 Write proper commit comments 9
4.9 Automate and optimize the release process 10
4.10 Use a version control tool 10
4.11 Use a build tool 11
4.12 Keep the repository clean 11

5 Slutsatser **12**

6 Framåtblickar **14**

7 Referenslista **15**
7.1 Publicerade källor 15
7.2 Elektroniska källor 15

1 Inledning

Denna djupstudie i kursen ”Coaching av programvaruteam” handlar om konfigurationshantering för Extreme Programming (XP)-team. Till skillnad från traditionell ”Software configuration management” (SCM), innehåller inte XP-metodiken explicita practices om konfigurationshantering, utan förutsätter att detta sköts av utvecklarna på det sätt de finner lämpligt. Detta kan vara dumt i grupper av oerfarna utvecklare, vilket gjort att stödpractices för konfigurationshantering har tagits fram {Asklund, Bendix, Ekman}. Därför har vi under ett XP-projekt gjort undersökningar om det behövs mer explicit vägledning för oerfarna utvecklare, eller om XP på egen hand medför god sed gällande konfigurationshantering.

En intressant skillnad på ytan mellan XP och traditionellt styrda projekt är att rollen som ”configuration manager” helt saknas i XP. Istället är det upp till utvecklarna att själva ta ansvar för denna del av programvaruprocessen. Dessutom var kundens roll i detta projekt radikalt olikt kundens roll i ett mer traditionellt projekt, då kunden här var betydligt mer närvarande och engagerad i projektet.

2 Bakgrund

Konfigurationshantering är ett omfattande område. Klassiskt ses förändringar som något som bör spåras och kontrolleras noga av särskilt utnämnda konfigurationshanteringsexperter. I större projekt finns det alltså en roll som innebär att personen eller personerna har ansvar för att källkod, dokumentation, manualer och releaser ständigt har rätt innehåll. I kontrast till detta ser XP förändringar och konfigurationshantering som något nödvändigt och bra, men som måste hanteras av utvecklarna själva för att kunna användas praktiskt. Genom specifika XP-practices som ”test first”, ”simple design” och ”frequent releases” sätts fokus på att produkten som utvecklas hela tiden måste vara testad och körbar. De kända metodböckerna och artiklarna om XP såsom {Beck} och {Jeffries, Hendrickson, Anderson} lämnar dock detta område till läsarens fantasi. Ett undantag till detta finns skrivet vid institutionen för datavetenskap vid LTH, nämligen Asklund, Bendix, Ekman.

Projektet som vårt arbete baserats på är projektdelen av kursen ”Programvaruutveckling i grupp”, som är obligatorisk för studenter som läser det andra året på civilingenjörsutbildningen i datateknik vid LTH. Projektet, som baseras på XP-metodik, går ut på att i grupp om 8-10 personer konstruera ett tidmätningssystem för Enduro-tävlingar. Samtliga XP-practices ska följas så långt det är möjligt. Projektet, som av naturliga skäl[1] är ganska komprimerat, utförs under 6 iterationer, bestående av ett planeringsmöte (2 timmar), individuell spike-tid (4 timmar per gruppmedlem) samt en långlaboration på 8 timmar som motsvarar en arbetsvecka.

2.1 Metod

Vår studie i detta ämne har utförts som en naturalistisk observation. Vi har studerat hur en grupp mindre erfarna studenter hanterat konfigurationshanteringsfrågor i ett programvaruprojekt baserat på XP-metodiken. Fördelen med denna metod jämfört med t.ex. kvantitativa metoder (som enkäter) eller kvalitativa metoder (som studie av källkod, dokumentation eller intervjuer med utvecklarna) är att vi ser hur utvecklarna reagerar på förändringar i projektet under tiden de inträffar. Vi har dessutom inte explicit sagt till utvecklarna att vi studerar dem, däremot har vi på ett subtilt sätt uttryckt vårt intresse för konfigurationshantering samt låtit lappar med CM-practices ligga tillgängliga i datorsalen. Vi tror också att intervjuer skulle ge för allmängiltiga resultat och tillrättalagda svar.

[1] Projektet genomförs under 8 veckor i andra årskursen för studerande vid civilingenjörsprogrammet i datateknik

3 Teori

3.1 XP-practices i projektet

För att närmare precisera vilka XP-practices som vi studerat följer här en kort beskrivning av de tolv XP-practices som dominerar marknaden[2].

- Pair programming
- Refactor mercilessly
- Test first
- Simple design
- Customer on-site
- Frequent releases
- Metaphors
- Continuous integration
- Collective code ownership
- Planning game
- Coding standards
- 40-hour week

3.2 CM-practices i XP-kontext

Som komplement till de tolv XP-practices som implementeras i projektet, finns det tolv så kallade ”CM-subpractices”, som presenterats av {Asklund, Bendix, Ekman}. Dessa har vi försökt utnyttja i projektet, för att fylla kunskapsluckan som vi antog fanns bland studenterna. Vi har inte tvingat några av dessa practices på studenterna, utan observerat om och hur man använt sig av dessa. Beskrivningar av practices har funnits tillgängliga för referens under varje långlaboration i form av lappar.

En kort beskrivning av dessa practices följer:

- Incremental refactoring – Bryt ner stora refaktoriseringar i mindre steg, se till att alla enhetstester hela tiden går igenom. Gör commit efter varje steg.

[2]För mer detaljerad information om XP-practices, läs t.ex. {Beck}

- Impact analyse refactorings – Analysera möjliga mergekonflikter med andra stories som konsekvens av refaktoriseringen.
- Use copy-merge work model – Låt varje utvecklare ha en egen workspace med en lokal kopia av repositoryt, där de kan integrera och testa sin kod innan de committar.
- Impact analyse stories as part of planning game – Analysera olika stories och ta med eventuella mergekonflikter i tidsuppskattningen för varje story.
- Physical audit in release process – Granska fysiskt samtliga releaser, både formella (till kund) och informella (interna). Bör automatiseras om XP-practicen ”Frequent releases” ska följas fullt ut, t.ex. med hjälp av en checklista.
- Define configuration items and their structures – Definiera konfigurationsenheter och deras struktur tidigt i projektet.
- Trace changes to stories – Låt varje förändring vara spårbar till en särskild användarstory.
- Write proper commit comments – Beskriv orsaken till varje förändring i systemet på en hög nivå vid varje commit.
- Automate and optimize the release process – Gör det möjligt att automatgenerera en release, t.ex. kompilera (med rätt versioner av varje modul), packa, acceptanstesta och skicka releasen.
- Use a version control tool – Använd ett versionshanteringsverktyg för att lagra de gemensamma konfigurationsenheterna och låt alla utvecklare synkronisera sitt arbete mot detta.
- Use a build tool – Använd ett verktyg för att automatiskt kompilera och bygga den exekverbara mjukvaran, och även elektronisk dokumentation (t.ex. JavaDoc).
- Keep the repository clean – All incheckad kod ska alltid fungera och uttjänt kod ska inte synas.

4 Substans

I detta kapitel beskriver vi våra insamlade data. Vi har valt att dela upp det i ett stycke för varje CM-subpractice, för att enklare kunna binda samma detta med vår teori. Dessutom har vi, då detta skrivits, endast gått igenom tre av sex iterationer. Med tanke på de förändringar i gruppens beteende vi observerat, är det troligt att resultatet i slutändan kan skilja sig från nedanstående.

4.1 Incremental refactoring

Denna practice säger alltså att projektmedlemmerna, då de refaktoriserar sin kod efter att ha avslutat en story eller task, ska göra refaktoriseringen i små steg. Detta hänger ihop med XP-practices som ”test first”, ”refactoring” och ”continuous integration”, det vill säga varje förändringskvanta bör vara litet för att undvika problem.

Projektmedlemmarna har istället börjat med att göra större refaktoriseringar, men har på sistone delat upp detta arbete. En motsträvande kraft är att man upplever det som bortkastad (improduktiv) tid i jämförelse med att implementera nya stories. I efterhand har dock designen blivit sådan attmånga uttryckt sitt missnöje över den ”grötiga” koden, och därmed insett vikten av stegvis refaktorisering. På grund av stora mergekonflikter och en acceptans av nyttan med små steg i utvecklingen har gruppen själv anammat denna practice. Detta har inte ställt till några större problem, tvärtom har arbetet gått smidigare efter att storleken på refaktoriseringarna minskat.

4.2 Impact analyse refactorings

Enligt denna practice bör en kort analys genomföras före refaktoriseringar av systemet, för att se om det påverkar nyutvecklade stories och genererar stora mergekonflikter. Detta har muntligt diskuterats i gruppen, t.ex. på planeringsmöten inför större enskilda refaktoriseringar, men det finns ingen uppföljning av detta, även om de som ansvarat för en större refaktorisering gjort systemet bättre enligt deras mening, har andra i gruppen inte upplevt det så. Den breda översikten av systemet har inte förankrats hos alla gruppmedlemmar.

Trots bristen på överblick, har vi haft nytta av att från början följa denna practice. Som resultat har vi sluppit stora mergekonflikter under projekets gång. Om man motsatsvis antagit att vi inte följt denna subpractice, hade vi med all säkerhet råkat ut för fler konflikter. Detta baserar vi på subjektiva upplevelser av andra gruppers problem i samma kurs. Genom att undvika big-bang-refaktoriseringar har alltså mycket vunnits.

4.3 Use copy-merge work model

Copy-merge-modellen stöds i och med användandet av CVS under projektets gång, och är vital för att t.ex. ”continuous integration” skall fungera. Gruppen uppskattar mycket att arbeta på detta sätt, och i och med att man håller practicen ”continuous integration” högt, har denna CM-practice överlag fungerat smidigt. Däremot är förståelsen för denna process inte så utvecklad som den skulle kunna vara. Gruppmedlemmarna spårar inte själva exakt vad i systemet som förändras när de integrerar sina delar med andras, trots milda påpekanden från coacherna. Istället gör man en update på hela systemet utan att få veta vad som ändrats, och på motsvarande sätt gör man en commit för allt man arbetat med utan att själva se vilka filer som läggs till eller förändras i repositoryt. Trots detta har modellen fungerat tillfredsställande, eftersom update/commit-cyklerna är korta och större mergekonflikter helt har kunnat undvikas.

En intressant observation är att en ”update” upplevs som en del av det kontinuerliga arbetet, medan ”commit” reserveras för särskilda tillfällen. Det har blivit bättre på sistone, men det är ändå så att ”continuous integration” görs på ett konservativt sätt. Med det menar vi att man uppdaterar sitt repository med de senaste ändringarna ofta, men sällan bidrar med incheckningar. Det kan ha med programmeringsvana att göra, det vill säga att medlemmarna inte skriver kod som fristående kan exekveras, utan tenderar att åtminstone till en början skriva större metoder och stora saker på en gång.

Som konsekvens ger många ”updates” inget resultat på det lokala repositoryt, men när väl någon gör en större ”commit” kan mycket behöva kontrolleras hos alla par. Under projektets gång har gruppens medvetenhet ökat, och nu upplevs inte CVS som ett hinder för utvecklingen. Det är trots allt första gången som gruppen stöter på ett versionshanteringsverktyg. En viss inlärningströskel är därför att vänta. Uppskattningsvis görs fortfarande cirka fem gånger fler ”updates” jämfört med incheckningar.

Det som fungerat från första början är denna subpractice i kombination med ”test first”, det vill säga gruppen kör alltid alla enhetstester vid varje update och före varje commit. Dessutom uppdaterar de alltid hela projektet före incheckning. Det kan sägas vara en sorts manuell variant av ”long transactions”.

4.4 Impact analyse stories as part of planning game

Under planning game genomfördes tidsuppskattningar av varje story, för att på ett ungefär kunna räkna ut hur mycket arbete som skulle utföras för varje iteration. Enligt denna practice ska man också ta hänsyn till vilka förändringar i existerande kod som måste genomföras för att nya stories ska kunna implementeras. Detta har visat sig vara en väldigt naturlig del av gruppens arbete, då vi oftast dragit över på tiden vid planeringsmöten just

på grund av diskussioner av denna typ. Det upplevs som viktigt av gruppmedlemmarna att man inte ”tar sönder” systemet när man stoppar in ny funktionalitet.

Diskussionerna har ofta polariserats av två starka viljor -Ű dels ”simple design”-viljan, som vill bygga ut existerande datastrukturer och finslipa gamla stories för att få nya att fungera, dels ”refactor”-viljan som vill göra mer genomgripande förändringar av systemet för att enklare kunna lägga till mer funktionalitet i efterhand. Efter tre iterationer uppnådde vi trots allt en sorts konsensus, där vi valde en mer XP-anpassad lösning i ”simple design”-andan. Vid diskussioner har gruppen också sett att detta faktiskt inte tar längre tid om en omdesign i framtiden skulle krävas. Gruppens hastighet (”velocity” med XP-terminologi) har inte heller blivit lidande av att vi valt denna lösning. På grund av viljan att inte ”förstöra” systemet har mergekonflikter med anledning av nya stories kunnat undvikas, genom att denna subpractice använts. Intressant här är att det fallit sig naturligt under XP att arbeta på detta sättet.

Under iterationernas lopp görs löpande kontroller av gruppmedlemmarna själva, för att undvika för mycket parallell utveckling. Det är alltså en ”continuous impact analysis” som genomförs. Detta är möjligt tack vare mycket kommunikation mellan gruppmedlemmarna, genom tillämpande av parprogrammering och genom täta byten av par.

4.5 Physical audit in the release process

Under de tre iterationer som fortlöpit, har två releaser gjorts. Under den första releasen hade gruppen utvecklat en automatiserad releaseprocess, men insåg på tok för sent att en manuell koll av systemet hade varit önskvärd. Detta bland annat på grund av bristfälligt användarinterface[3]. Snart därefter insåg teamet vikten av en manuell kontroll av varje release, för att se att kundens acceptanstester manuellt kunde köras igenom.

4.6 Define configuration items and their structures

Denna subpractice handlar om vad som egentligen utgör en konfigurationsenhet, och hur den bör struktureras och hanteras i repositoryt, samt att göra detta tidigt i projektet. Detta strider dock till viss del mot XP-metodiken, där denna definition endast bör göras om det är till gagn för utvecklarna i deras fortsatta arbete, och att man ska kunna förändra det under arbetets gång. I vårt fall har vi inte haft någon större diskussion om vad som bör ingå i systemet och vad som bör vara en del av utvecklarnas ”privata” domäner. Det som finns är en åtskillnad mellan utvecklarnas ”spikes”, produktionskoden, teknisk dokumentation samt manual. Det fanns lite färdig struktur vid ”iteration 0” av systemet,

[3]Programmet kunde startas, men gjorde inget vettigt

men detta har helt ignorerats vid fortsatt utveckling – det har helt enkelt inte känts som något som behövts under processens gång.

Det som har hänt är att vi har tagit bort ett antal klasser som fanns i repositoryt från början. Detta har dock inte ställt till så stora problem eftersom Eclipse är väldigt ”snällt” och döljer raderade filer för användaren. Filerna ligger ju kvar rent fysiskt i CVS:en. Tänkbara problem är om man återskapar klasser som en gång raderats. Det har vi inte någon plan för.

4.7 Trace changes to stories

För att ha nytta av förändringskontroll i XP, bör ovanstående subpractice vara viktig, då kunden kanske väljer att stryka vissa stories helt ur systemet. Då kan det vara viktigt att se exakt vilka förändringar som hör till denna story. Men detta har inte varit populärt hos utvecklarna – det finns sporadiska kommentarer som relaterar till enskilda stories, men generellt sett är det väldigt svårt att se t.ex. vilka klasser som hänger ihop med en viss story. Dock har det på senare tid blivit så att incheckad kod kommenteras med ”hör ihop med story 13.1” och dylikt. Vi kommer att ha fokus på denna subpractice under kommande iterationer.

4.8 Write proper commit comments

Detta är en förutsättning för att ovanstående subpractice ska kunna implementeras, och kommentarerna till ”Trace changes to stories” kan direkt överföras till denna subpractice. Gruppens kommenterarer är generellt sett bedrövliga, men förståeliga om man tänker på subpracticen ”Use copy-merge work model”, där teamet inte alltid haft full koll på vilken kod de checkar in i systemet. Kommentarer som ”fixat lite” eller ”refaktoriserat” är vanliga. Kanske har teamet inte orkat skriva tillräckligt mycket kommentarer, eftersom man gjort update/commit ofta.

I vår studie har vi inte behövt backa eller spåra förändringar särskilt ofta. Istället har detta ersatts med verbal kommunikation. Kanske kan man se en brist i att verbal kommunikation behövs oftare än om vi haft en klar struktur för commit comments, dessutom hade det varit en brist om de personer som arbetat med en viss story varit borta vid tillfället de behövts. Detta är en nackdel ur XP-synvinkel av den anledning att det motarbetar ”Collective code ownership”.

En intressant skillnad mellan olika versionshanteringssystem är sättet som commit comments kan användas för att styra utvecklingen. Till exempel kräver verktyget ”ClearCa-

se”[4] att man skriver commitkommentarer vid utcheckning, och sedan bekräftar detta vid incheckning. Vi tror detta hade gett mer genomtänkta och disciplinerade kommentarer, eftersom man ofta är mer motiverad och fokuserad på sin uppgift innan man börjat med den. Risken finns alltid att man är mentalt uttröttad efter att ha gjort klart en avancerad story, och då inte har styrka kvar att formulera en bra incheckningskommentar.

4.9 Automate and optimize the release process

Som vi sett i subpracticen ”Physical audit in the release process”, har teamet på ett tidigt stadium förstått vikten av automatiserade releaser. Det har gjorts omfattande arbete för att automatisera och optimera detta, och efter tre iterationer har vi nu en smidig, automatiserad releaseprocess samt automatiserade acceptanstester som testar på det program som ingår i releasepaketet. Man kan ifrågasätta om man verkligen tjänar tid totalt sett på att implementera detta i ett avgränsat projekt. Vi har dock märkt att utvecklarna känner sig tryggare med en automatiserad process. Det upplevs som mentalt tillfredsställande att ofta kunna släppa en färdigpackad produkt, även om det bara är för internt bruk. Vinsten kan alltså bestå av bättre förtroende för systemet hos utvecklarna snarare än mindre totaltid för projektet. En bieffekt av detta kan dock vara högre effektivitet, vilket i så fall kan ge en tidsvinst.

4.10 Use a version control tool

Under projektet har Eclipse tillsammans med CVS använts för att spåra versioner och integrera systemet. Detta har varit en förutsättning för de flesta andra subpractices, och projektmedlemmarna har utan problem anpassat sig till detta sätt att arbeta. Det skulle kunna önskas lite mer förståelse för vad som egentligen sker under varje enskild synkronisering, men överlag fungerar det bra som en sorts transparent miljö, där alla vet hur den reagerar. Vi har använt CVS mer som ett mergeverktyg än ett versionshanteringsverktyg, av anledningen att XP implicit förutsätter en lättviktig syn på konfigurationshantering. Såväl branches som varianter lämpar sig inte väl för ett projekt som är både avgränsat i tiden, genomförs av studenter utan vana av såväl XP som SCM, och för en produkt vars releaseprocess inte underlättas av dessa finesser. Branches och varianter är inte kompatibla med ”Simple design”-paradigmet.

Att ha en djupare förståelse för vad ett verktyg verkligen gör är viktigt för att kunna lösa problem som verktyget orsakar. Om en mergekonflikt uppstår utan synbar anledning, är det bra att veta vad verktyget försöker göra vid en merge, hur kommunikationen med repositoryt ser ut, och så vidare. Som exempel kan nämnas att en gruppmedlem

[4] Se http://www-306.ibm.com/software/awdtools/clearcase/

ställer frågan ”Har någon ändrat i den här klassen?” till hela teamet, och att sedan övriga gruppmedlemmar inte kan svara på frågan.

4.11 Use a build tool

Som releaseverktyg används scriptspråket ”Ant”. Scriptet har förfinats mer och mer under iterationernas lopp, och innehåller nu nästan allt som krävs för att få en fungerande release. Detta har teamet själva konstaterat, efter de inledande erfarenheterna med att försöka skapa en release utan fungerande huvudprogram. Scriptet använder även javac för att producera exekverbara program. Ant utnyttjar XML-taggar för att definiera vad byggprocessen innebär, och flera i teamet har vidareutvecklat detta.

4.12 Keep the repository clean

Efter ett tag fylls repositoryt med uttjänta moduler, tester och klasser. Refaktoriseringar som görs har ännu inte tagit hänsyn till detta Ű man verkar vara alltför försiktig med att ta bort gamla konfigurationsenheter som inte längre används. Men däremot har teamet en hög etisk standard, och det är implicit förbjudet att lägga in kod som inte fungerar i det gemensamma repositoryt. Detta har kanske hänt en eller två gånger under projektets gång, men har inte ställt till några problem som tagit längre än fem minuter att lösa. En tänkbar konsekvens av ”Keep the repostory clean” kan vara att teamet inte checkar in kod lika ofta, som vi nämnt under ”Use copy-merge work model”.

5 Slutsatser

Det som varit intressant med vår studie är just att se hur teamet instinktivt (med minimal vägledning) reagerat då de utsatts för förändringar och nya utmaningar, som till exempel releaser och att arbeta tillsammans i stora grupper. Vi har sett att XP stimulerar till att arbeta på ett visst sätt, som överensstämmer med flera (dock inte alla) av de beskrivna CM-subpractices. Det visar sig alltså att gruppen, bara genom att följa processen, automatiskt lär sig god sed vid konfigurationshantering, även om de inte är medvetna om det. Detta kan användas som försvar av XP-metodiken då den kritiseras för brist på översikt, plan och dokumentation. Följande lista sammanfattar stödet som vi funnit mellan XP och SCM-subpractices:

- Incremental refactoring – Stöds delvis av XP. Det finns en tendens mot större refaktoriseringar i vårt team, men medlemmarna har självmant gått mot mindre refaktoriseringar.
- Impact analyse refactorings – Stöds av XP. ”Planning game”, ”Simple design” och ”Refactor mercilessly” ger denna subpractice som naturligt följd.
- Use copy-merge work model – Förutsättning för att ”continuous integration” ska fungera. Underlättar även ”collective code ownership” rent praktiskt.
- Impact analyse stories as part of planning game – Stöds av XP, av samma skäl som ”impact analyse refactorings”.
- Physical audit in release process – Stöds av XP genom ”Frequent releases”.
- Define configuration items and their structures – Här finns inget stöd i XP, snarare tvärtom. Att från början definiera vilka konfigurationsenheter som ska existera i systemet strider mot XP-visioner som ”do the simplest thing that could possibly work” och ”embrace change”[5]. Om projektet är omfattande eller om man arbetar i en formell miljö, kan det vara nödvändigt att introducera denna subpractice vid sidan av XP.
- Trace changes to stories – Stöds inte explicit av XP, men då förändringar ska vara levande och genomsyra hela teamets arbete kan det vara bra att få medlemmarna att spåra förändringarna genom denna metod. Enligt vår observation är det inte något som kommer naturligt.
- Write proper commit comments – På samma sätt som ”trace changes to stories” stödjs inte heller denna practice av XP. Vi uppfattar att kommunikationen sker på ett annat sätt, genom källkod och muntlig kommunikation.

[5] Se {Beck} eller {Jeffries, Hendrickson, Anderson}

- Automate and optimize the release process – Stöds av ”Frequent releases”, dessutom skänker den trygghet åt utvecklarna, något som också är en viktig faktor i XP.
- Use a version control tool – Stödjs inte explicit av XP. Denna subpractice har än så länge mest använts ur ett mergeperspektiv och som ett sätt att enklare bedriva parallell utveckling.
- Use a build tool – Stöds av ”frequent releases” på precis samma sätt som ”automate and optimize the release process”.
- Keep the repository clean – Stöds naturligt av ”Simple design”.

Något som våra resultat alltså visar är att alla SCM-subpractices inte följs på ett naturligt sätt. Ytterligare en sak som förvärrar situationen är att de subpractices som skulle underlätta teamets refaktoriseringsmöjligheter inte lika naturligt implementerats. Det kan leda till att teamets hastighet i framtida iterationer minskar, då systemet inte längre är lika lättarbetat. En tänkbar lösning på detta kan man söka bland olika practices som handlar om refactoring, det kan vara så att ett oerfaret team måste utsättas för en Grand Refactoring Day (se referenslista) för att se möjligheterna med SCM.

På grund av projektets omfattning, blev många intressanta SCM-problem eller möjligheter aldrig testade. Även om man försöker konstruera situationer då dessa kan användas (som exempel, att kunden begär rättningar från den första tidiga releasen sent i projektet), är risken att andra delar av kursen som inte behandlar SCM blir lidande. Det hade varit bra att ha läst en kurs i SCM innan man påbörjar ett projekt av den här typen, men å andra sidan kan en SCM-kurs i det här stadiet av utbildningen inte ge lika mycket som efter att man genomfört ett såhär pass stort projekt.

6 Framåtblickar

Denna studie skulle behöva göras på flera team, och även med erfarna programmerare. Dessutom är det svårt att dra några direkta slutsatser efter endast tre iterationer. Men med tanke på debatten kring XP och SCM finns det åtminstone ett par studier till som borde göras, främst med fokus på SCM och refaktorisering. Förslag på dessa följer:

- En studie på två grupper, där den ena strikt använder sig av SCM-subpractices, medan den andra observerar hur eller om medlemmarna på eget initiativ utnyttjar sig av SCM-subpractices.
- En studie som grundas på ”Antipatterns” till SCM-subpractices för att se om det över huvud taget blir någon skillnad.
- Ett större projekt med mer avancerade SCM-specifika problem, såsom flera kunder (implicerar flera branches), flera varianter för olika plattformar, att man skrotar ett utvecklingsspår, med mera.
- Ett ”maintenance”-projekt, där man tar över till exempel föregående års ”färdiga” projekt och underhåller det (utan att lägga till för mycket funktionalitet), samtidigt som kunden önskar förbättringar, felrättningar, felrättningar på äldre versioner, och så vidare.

7 Referenslista

7.1 Publicerade källor

- Ulf Asklund, Lars Bendix, Torbjörn Ekman: Software Configuration Management Practices for eXtreme Programming Teams, in proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques - NWPER'2004, Turku, Finland, August 17-19, 2004
- Wayne A Babich: Software Configuration Management - Coordination for Team Productivity, Addison-Wesley Publishing Company, Inc., 1986. (chapters 1, 2, 3 and 5)
- Kent Beck: Extreme Programming Explained: Embrace Change (2nd Edition), Addison-Wesley Professional, 2004
- Bohner and Arnold: An Introduction to Software Change Impact Analysis, in Software Impact Analysis, IEEE, 1996
- Brooks: The Mythical Man-Month, Addison-Wesley Professional, 1995
- M. A. Daniels: Principles of Configuration Management, Advanced Applications Consultants, Inc., 1985
- Ron Jeffries, Chet Hendrickson, Ann Anderson: XP Installed, Addison-Wesley Professional, 2000

7.2 Elektroniska källor

- `http://www.c2.com/cgi/wiki?ExtremeProgrammingRoadmap`, 2005-02-21
- `http://www.c2.com/cgi/wiki?RefactorMercilessly`, 2005-02-21
- `http://www.c2.com/cgi/wiki?GrandRefactorigDay`, 2005-02-21