Coaching av programvaruteam EDA270, djupstudie:

Praktisk SCM användning i XP-projekt

Martin Malek
Anders Hellström

Lunds Tekniska Högskola

22 februari 2005

Version 1.0

Sammanfattning

Som utgångspunkt för artikeln används tolv *Software Configuration Management, SCM,* arbetssätt som utvecklats från traditionella *SCM* principer och anpassats för eXtreme Programming, *XP*, projekt. Artikeln har som mål att minska klyftan mellan teori och praktik för användningen av *SCM* genom att ta upp praktiska erfarenheter som samlats in under en projektkurs ämnad för andraårsstudenter vid Lunds Tekniska Högskola.

Innehållsförteckning

Inledning

Syftet med denna artikel är att minska klyftan mellan teori och praktik för användningen av *Software Configuration Management*, *SCM*, i *eXtreme Programming*-projekt, *XP*. Som utgångspunkt har vi tolv *SCM* arbetssätt som utvecklats från traditionella *SCM* principer och anpassats för *XP*-projekt och ett mjukvaruteam på vilka dessa principer skall appliceras. De tolv arbetssätten återfinns i artikel [3] och består av följande:

- Inkrementell refaktorisering
- Påverkansanalys av refaktoriseringar
- Användning av *copy-merge* modellen
- Påverkansanalys av *stories* som en del av *planning game*
- Automatiserade *release*-processer
- Granskningar som en del av *release*-processen
- Definiera konfigurationsenheter
- Spårbarhet genom *stories*
- Skriva ordentliga incheckningskommentarer
- Använda versionshanteringsverktyg
- Använda byggverktyg
- Hålla rent i *repositoryt*

Syftet var att genom ökad förståelse för *SCM* öka produktiviteten i projektet. Som ett verktyg för att förmedla *SCM* delades artikeln [3] ut till alla teammedlemmer inför första iterationen.

Denna artikel börjar med en kort bakgrund följt av de speciella förutsättningar som gällde för projektet. Vidare görs en mappning mellan teorin och uppmaningarna i de tolv *SCM* arbetssätten mot våra faktiska erfarenheter.

Bakgrund

För ett par år sedan startade en ny kurs vid LTH i vilken man lär ut *XP* [1] till studenter som går andra året på Civilingenjörsprogrammet i Datateknik. Kursen är indelad i två delar, en teoretisk samt en praktisk. I den teoretiska delen undervisas studenterna under ett antal föreläsningar i de olika *XP-practices* som finns och i den praktiska delen får de användning för sina teoretiska kunskaper genom grupparbete i form av ett mjukvaruprojekt. För mer information om hur kursen är utformad finns det en artikel [2] skriven av kursens grundare.

Även om *XP* proklamerar saker som kollektivt kodägande, kontinuerliga integrationer och kontinuerliga releaser så ges inga direkta exempel på hur dessa praktiker skall efterlevas rent praktiskt. Vid LTH har man utarbetat ett arbetssätt på tolv punkter som är baserade på traditionella *SCM* praktiker men vinklade för

att passa in i *XP*-projekt. Dessa finns att tillgå i artikeln [3]. Ytterligare information om *SCM* i *XP*-projekt kan hittas i artikel [6].

Fastän dessa artiklar lyfter fram hur *SCM* kan appliceras på *XP*-projekt och ger konkreta förslag på vilka arbetsmoment som bör ingå så täcker de inte det praktiska arbetet med att få oerfarna programmerare att ta till sig och börja använda dessa arbetsmetoder och de hinder som kan dyka upp på vägen.

Förutsättningar

Eftersom teoridelen av kursen inte behandlade *SCM* i någon större utsträckning fann vi det lämpligt att förse vårt team med artikel [3] redan inför den första iterationen. Teammedlemmarna instruerades om att läsa igenom och tillgodogöra sig artikeln samt att ha den som referens under kursens gång.

Även om studenterna presenterades för *SCM* genom en tvåtimmars föreläsning och en tvåtimmars laboration under kursens teoretiska del, så anser vi att det varit lämpligt att utöver aktuell kurslitteratur även inkludera artikel [3] då denna specifikt behandlar *SCM* ur ett *XP*-perspektiv.

De resultat som diskuteras i denna artikel är kanske inte direkt vägvisande för ett professionellt *XP*-projekt. Det finns ett antal parametrar som måste tas i beaktning.

Antal personer: 8 utvecklare samt 2 coacher.

Nedlagd tid: Teamet jobbar 8 timmar per vecka i 6 veckor. Dessutom tillkommer *spike*-tid på 4 timmar per person och vecka. *Spike*-tiden får inte användas till att producera ny funktionalitet utan skall användas till att lära sig nya verktyg/arbetsmetoder samt läsa in sig på kommande uppgifter.

Inlärningströskel: Till skillnad från ett professionellt arbetslag så behövde teamet varje vecka lära sig nya verktyg och arbetsmetoder. Detta medförde att ett gediget arbetssätt inte infann sig direkt. Förutom *XP*-praktikerna introducerades även *SCM*. Vår uppfattning är att inlärningströskeln höjdes ytterligare på grund av detta.

Erfarenhet: Majoriteten av teamet hade inte mer programmeringsvana än grundläggande universitetskurser.

Övrigt: *XP*-praktiken, customer on site, kunde inte efterlevas som den är definierad då kunden var kund för flera team och inte fanns på plats hela tiden.

Istället för en kommersiellt fungerande mjukvara så var målet med kursen/projektet att lära ut *XP* som ett arbetssätt.

Införandet av SCM praktiker i projektet

Varje stycke börjar med en kort beskrivningen av praktiken och vad dess syfte är. Därefter följer våra erfarenheter av just den praktiken. Vi tar upp hur vi hanterat den och vilka positiva samt negativa erfarenheter vi har av den.

Inkrementell refaktorisering

Inkrementell refaktorisering uppnås genom att man delar upp de planerade refaktoriseringarna i flera mindre delar. Meningen är att skapa trygghet i arbetet med refaktoriseringen. Genom uppdelningen ökar chansen att snabbt hitta de fel man eventuellt råkar göra och det blir enklare att stämma av att programmet fortfarande fungerar.

I vårt team gjorde vi kontinuerliga refaktoriseringar under långlabbarna men också som *spikes*. Det tydligaste exemplet på användning av inkrementella refaktoriseringar är de fall då flera teammedlemmar parallellt gjorde refaktoriseringar som spikes. Det krävdes då att de samordnade och fördelade sitt arbete.

Även om denna praktik hör till de viktigare så var den en av de svåraste att efterfölja. Att refaktorisera kräver att man har en överblick av hela koden och som utvecklare har man översikt över den del man själv implementerat. Att sätta sig in i andras kod inför en refaktorisering kan verka skrämmande. Därför kan det hända att teammedlemmarna drar sig för refaktoriseringar.

Påverkansanalys av refaktoriseringar

En påverkansanalys av refaktoriseringar ämnar att finna de faktorer som kan orsaka problem, oftast *merge* problem, som en refaktorisering kan innebära.

För att kunna arbeta parallellt med refaktoriseringar så krävs det kommunikation. Denna kommunikation brukade i vårt fall resultera i en fördelning av arbetsuppgifterna. Teamet gjorde inga direkta påverkansanalyser av refaktoriseringar utan gjorde oftast bara en fördelning av arbetet. Arbetet fördelades så att en, eller max två personer, exklusivt arbetade med just vissa delar av koden. På så vis minimerades risken för att *merge*-konflikter skulle uppstå. Detta arbetssätt kanske inte är det som förespråkas i artikeln [3], men det fungerade för oss.

Användning av copy-merge modellen

Copy-merge arbetsmodellen [4] innebär, eller brukar åtminstone göra det, att varje utvecklarpar har ett eget så kallat *workspace*. I detta *workspace* har de tillgång till en komplett och uppdaterad kopia av allt material som ingår i projektet. Det är i *workspacet* som arbetet utförs. När arbetet är färdigt uppdateras *workspacet* med den senaste versionen av materialet från *repositoryt*. Efter alla enhetstester gått igenom förs den lokala kopian in i *repositoryt*. Eclipse, som är utvecklingsmiljön som används i kursen, använder via ett *CVS*-plugin c*opy-merge* modellen [4] då *CVS* används i så kallad *client-server* läge men stöder i detta läge inte så kallade *long transactions* [4].

Eftersom Eclipse använder *workspace* innebär det att teamet per automatik använder *copy-merge* modellen. Under detta faller också flera andra *SCM* tekniker in, bland annat användning av *brancher*. Att skapa en branch och jobba i den är enkelt och utvecklarna klarade det efter en kort introduktion. Jobbet låg i att genomföra en merge. Eclipse tillhandahåller en guide som delar upp de olika stegen i en *merge*. Beroende på hur omfattande ändringarna i koden var, resulterade de i *merge*-konflikter av varierande svårighetsgrad. Lyckligtvis erbjuder Eclipse även ett synkroniseringsverktyg för att manuellt göra en *merge*. Teamet fick nytta av detta verktyg vid ett flertal tillfällen. De *merges* som utfördes tog relativt mycket tid och resurser eftersom de krävde att minst två utvecklarpar jobbade tillsammans. Teamet lyckades genomföra alla *merges* utan att funktionalitet gick förlorad eller att andra problem uppstod.

En förbättring hade varit att låta en medlem ur varje par ingå i ett *merge*-par. Detta hade både minskat resurstillgången, då det andra paret hade kunnat jobba vidare med en annan story, och tidsåtgången då arbetet förmodligen blivit effektivare, eftersom färre viljor varit inblandade.

Vår uppfattning är att synkroniseringsverktyget alltid skall användas för att hålla sig uppdaterad med repositoryt. Dels därför att det är bra ur ett pedagogiskt perspektiv eftersom teammedlemmarna aktivt får ta beslut om den kod som skall ersättas, något som leder till önkad förståelse för koden, samt att risken för en oönskade merge-konflikter minskar.

Det hade varit önskvärt att i *CVS* kunna använda *long-transactions* då det stöder *XP*-praktiken om kollektivt kodägande. I praktiken så åstadkoms denna funktionalitet genom att användaren själv utför en *Refresh* och *Update from repository* av sitt *workspace*.

Påverkansanalys av stories som en del av planning game

Genom att göra en påverkansanalys och ta hänsyn till kostnaden för en *merge* konflikt av varje *story* som en del av *planning game* kan man göra noggrannare planeringar och kostnadsuppskattningar. Skillnaden mellan påverkansanalys och arbetsfördelning är den att i en påverkansanalys går man igenom de beroenden som finns mellan olika *stories* och bestämmer implementationsordningen så att man har så få beroenden som möjligt.

Under planeringsmötena i kursen fick teamet i uppgift att göra kostnadsuppskattningar av aktuella *stories*. Dessa uppskattningar låg senare till grund för kundens prioriteringar. I samband med att kunden prioriterade en *story* gjordes även en påverkansanalys på vilka andra s*tories* som kunde implementeras parallellt utan att risken för *merge* konflikter ökade. På så vis fick kunden maximal utdelning av produktivitet kontra kostnad.

En vanlig källa till *merge*-konflikter var att implementationsordningen på aktuella *stories* ändrades. I dessa fall användes synkroniseringsverktyget för att lösa dessa konflikter.

Automatiserade release-processer

För att underlätta arbetet och minimera risken av att mänskliga fel ska försvåra, försena eller rent av omöjliggöra en release så bör *release*-processen automatiseras. Om inte annat så för att det går snabbt att göra en release om processen är automatiserad.

Utan att teamet tidigare hade provat att göra en release, togs beslutet att processen skulle automatiseras. Motiveringen var att underlätta och minimera risken för att fel skulle uppstå. Tyvärr tog det lång tid innan vi fick det att fungera och resultatet blev att onödigt mycket resurser gick åt att lösa ett till synes enkelt problem. Felet låg i hur byggverktyget *Ant* genererade manifestfiler till den exekverbara jar-filen.

Trots att det tog onödigt mycket resurser att få *Ant*-scriptet att fungera så värderades resultatet mycket högt. Det gick snabbt och problemfritt att göra en *release* och arbetet kunde fokuseras på att verfiera och validera den. Det måste påpekas att *Ant*-scriptet kontinuerligt utvecklades under projektets gång, allteftersom nya entiteter skulle ingå i *releasen*.

Använda byggverkyg

Syftet med att använda ett byggverktyg är inte bara att kunna kompilera källkoden utan det ska även användas till att generera dokumentation, så som till exempel Java doc.

Åter kunde teamet dra nytta av funktionaliteten i Eclipse som via en *Ant*-plugin ger möjlighet att kompilera källkoden, lägga till Java doc och generera dokumentation på ett automatiserat och enkelt sätt. En av förutsättningarna för att lyckas i projektet är just tillgången på kraftfulla verktyg. *Spike*-tiden som investerades i att lära gruppen utveckla automatiserade releaser var en av de *spikar* som gav mest utbyte i form av sparad tid.

Granskningar som en del av release-processen

Validering och verifiering av materialet som ingår i en release är en förutsättning för att lyckas göra en korrekt release. Att kunden har möjlighet att testa det som produceras är en av de viktigaste förutsättningarna för att kunden skall känna sig delaktig i utvecklingsprocessen och på så vis garantera projektets fortsätta existens. Får kunden något som inte fungerar så är ju risken större att han slutar betala och därmed står teamet utan arbete.

Innan en release skickades till kunden så fick ett par i uppdrag att validera och verifiera den genom att packa upp och exekvera programmet. Flera gånger lyckades vi på detta sätt förhindra att skicka material till kunden som inte fungerade eller innehöll fel. Genom vårt sätt att arbeta, alltså att utveckla mjukvara enligt metodiken *XP*, gick det snabbt att korrigera felen och göra en ny release. Snabbheten ligger framför allt i de korta beslutsvägarna. *Change contol board*, *CCB*, utgörs ju av teamet och kunden.

Genom att validera och verifiera *releasen* innan den skickades till kunden kunde vi avhjälpa enklare fel såsom att programmet inte gick att köra och att alla filer inte skickades med.

Definiera konfigurationsenheter

Eftersom många konfigurationshanteringsverktyg inte klarar att hantera att filer flyttas eller döps om så krävs det att man i ett så tidigt skede som möjligt gör upp en struktur och bestämmer vad som skall vara konfigurationsenheter. Skulle man i efterhand till exempel döpa om en fil så går all versionshistorik om just den filen förlorad. Att verktygen sätter sådana här begränsningar är inte bra, framförallt inte ur ett *XP*-perspektiv där förändringar uppmuntras. En avvägning mellan flexibilitet och spårbarhet måste göras.

Under iteration noll gjorde teamets coacher upp en plan för vad som skall räknas som konfigurationsenheter. I den planen försökte vi förutsäga, bland annat baserat på tidigare erfarenhet, vad som skulle komma räknas som konfigurationsenheter. Denna plan presenterades under det första planeringsmötet. Att följa den förutbestämda planen var ibland svårt. Sådant som inte var tänkt att räknas som konfigurationsenheter letade sig emellanåt in i repositoryt. Detta berodde ofta på att vid incheckning så gjordes ingen detaljerad kontroll över de filer som skulle ligga under versionskontroll. Det rörde sig främst om filer med resultat från tävlingar som vårt program skapat. Det är tyvärr enkelt att lägga till nya sådana filer med Eclipse, men också enkelt att ta bort dem ur konfigurationsverktyget. En lösning var att använda *cvsignore* som listar alla filer som skall ignoreras av konfigurationshanteringsverktyget. Vi upplevde det dock aldrig att vi inte hade kontroll över vad som hanterades av konfigurationshanteringsverktyget.

Använda konfigurationshanteringsverktyg

Med hjälp av ett konfigurationshanteringsverktyg lagras alla konfigurationsenheterna. Verktyget tillåter alla utvecklare att hålla sig synkroniserade med repositoryt. Det är viktigt att allt som är incheckat i repositoryt fungerar och kompilerar eftersom alla utvecklare jobbar mot detta. Syftet med det är att skapa tilltro och ge utvecklarna mod att våga ändra koden. Konfigurationshantering möjliggör parallellt arbete samt möjlighet att backa tillbaka om det skulle visa sig att någonting inte fungerar.

I projektet användes konfigurationshanteringsverktyget *Concurrent Versioning System*, *CVS*, genom en plugin direkt i Eclipse. Genom att teamet under teoridelen av kursen fått möjlighet att bekanta sig med *CVS* så fanns goda förutsättningar för att kunna använda verktyget på ett korrekt och effektivt sätt. Tack vare detta kunde coacherna introducera teamet i *CVS* lite mer avancerade funktionalitet så som branch-, synkronisering- och spårningsverktyg. Vissa begränsningar finns dock i *CVS*, dessa rör bland annat möjligheten att flytta och döpa om filer. Det faktum att det aktuella projektet inte är särskillt omfattande och inte bedrivs under en längre tid medför därmed inte några problem.

Håll rent i repositoryt

Ett *repository* som innehåller fel förstör jobbet för hela teamet eftersom alla gör sina uppdateringar från detta. Ett rent och fungerande repository är ett måste för ett fungerande projekt.

Ett vanligt förekommande fel är halvfärdig kod, till exempel kod som inte kompilerar eller kod som gör att testfallen inte går igenom, i *repositoryt*. Ett rent *repository* var i vårt team en mognadsprocess. Inte förrän teamet fick uppleva det arbete som krävs för att återställa ett felaktigt *repository* förstod de vidden av att hålla det rent och snyggt. Ett sätt att snabbt lära ut en bra incheckningsmetodik är att som coach aktivt ta del i varje programmeringpars första incheckningar. Helt enkelt att gå igenom de kriterier som måste vara uppfyllda innan incheckning: hela storyn implementerad, java-dokumentation samt relevanta tester för *storyn*. Annars är det lätt att något av dessa steg hoppas över i ivern att få tillgodoräkna sig en *story*.

Desvärre så var ett rent repository svårt att uppnå. Så länge en coach var närvarande vid incheckning fungerade praktiken bra. Ett sätt att minska felen i repositoryt skulle kunna vara att tillhandahålla en checklista för de steg som måste göras innan något får checkas in. Detta kan kombineras med ett incheckningstoken i form av en maskot. Endast det par som har token får göra incheckningar.

Skriva ordentliga incheckningskommentarer

En bra incheckningskommentar beskriver anledningen till den ändring som gjorts i koden samt en kort beskrivning av vad det är som ändrats.

En granskning av teamets incheckningskommentarer visade att en klar majoritet av kommentarerna beskrev den ändring som hade gjorts. Desvärre saknades ofta en utförligare beskrivning som i större detalj beskrev de genomförda ändringarna. Kommentarer som ”Ändrade metod X” borde istället vara formulerade som ”Ändrade metod X till att omfatta händelse Y”. En ytterligare förbättring av kommentaren och något som skulle öka spårbarheten genom *stories* skulle kunna ha följande utseende, ”Story A. Ändrade metod X till att omfatta händelse Y”.

Ett exempel på en dålig incheckningskommentar är: ”Fixade lite i javadocen”

Ett exempel på en bra incheckningskommentar är: ”Refactored and added stub for actions needed on window closing in client GUI”

Spårbarhet genom stories

Varje ändring bör kunna spåras [5] till den *story* och dess deluppgifter som den berör. Detta leder till att man kan spåra när en viss funktionalitet introducerades i koden och på så sätt underlätta buggletning.

Denna *SCM* metod får man på köpet om man är noggrann med att skriva ordentliga incheckningskommentarer. Möjligen hade man fått större utbyte av detta arbetssätt om projektet varat längre. Storleken på nuvarande projekt samt att utvecklarna hela tiden är i kontakt med varandra gör att man oftast vet var och när eventuella buggar kan ha introducerats.

Slutsatser

Även om syftet var att introducera och på ett fördjupat sätt arbeta med *SCM* i projektet så stod det snabbt klart att detta var svårt. Att teamet saknade tidigare erfarenheter av mjukvaruutveckling i grupp samt tidigare erfarenhet av *XP* gjorde att det blev en högre tröskel att komma över innan *SCM* arbetet började fungera. Anledningen till detta var att mycket av coachernas tid gick åt till att hjälpa teamet i det grundläggande arbetet. Istället formades arbetet så att *SCM* praktikerna introducerades efterhand. Om det ursprungliga målet var att snabbt presentera och öka förståelsen för *SCM* så blev det nya målet att succesivt öka studenternas förståelse för de samma.

Vi upplevde inte *SCM*-praktikerna som svåra att förstå eller praktisera, problematiken låg snarare i att det var så mycket nytt att lära och därför blev i

vissa fall praktiserandet lidande. Som exempel kan ges att inte *repositoryt* alltid var rent och att incheckningskommentarerna inte höll önskad kvalitet.

Inledningsvis hade inte alla *SCM*-praktikerna lika stor vikt. Detta kan exemplifieras genom att granskningar som en del av *release*-processen inte var nödvändig förrän en *release* gjorts. För att sänka inlärningströskeln för teamet utan att för den delen tumma på kunskapen bör man i början fokusera på ett fåtal grundläggande *SCM*-praktiker, som till exempel hålla rent i *repositoryt,* använda versionshanteringsverktyg, påverkansanalys av *stories* som en del av *planning game* samt skriva ordentliga incheckningskommentarer. Övriga praktiker bör introduceras först då behovet för dem uppkommer på ett naturligt sätt.

Utfallet av vårt arbete, trots de rådande förutsättningarna, är att vi lyckats öka förståelsen i teamet för vad *SCM* är och hur man arbetar med det på ett effektivt sätt. Ett av de mest lyckosamma exemplen är att teamet använde brancher för att parallellt kunna arbeta. Ett annat exempel är att de genom god kännedom om vilka konfigurationsenheter som skulle användas lyckades att inte lägga in felaktiga filer i repositoryt. Att teamet hade tillgång till en, i stort sett för ändamålet komplett, utvecklingsmiljö är en annan faktor till det lyckosamma utfallet.

Skulle projektet drivits under en längre tid än sju veckor eller att projektet skulle utmynnat i en produkt som sålts på den öppna marknaden så skulle arbetet med att skriva bra incheckningskommentarer för att öka spårbarheten behövt förbättras, särskilt om projektet skulle behöva underhållas i framtiden. Att i det fallet kunna spåra förändringar i koden direkt till en *story* skulle varit en absolut nödvändighet.

Ursprungligen hade vi tänkt att vi skulle göra en jämförelse mellan vårt team som i förväg hade fått tillgodogöra sig information om hur *SCM* bör användas och ett team som inte fått samma information. Dock så insåg vi att denna jämförelse skulle vara intetsägande då korrelationen mellan våra urvalsgrupper var alldeles för liten. Anledningen var att den inbördes kunskapsfördelningen i grupperna var så pass stor samt att den sparsamma förekomsten av de mätvärden som vi var intresserade av, antalet *merge*-konflikter, antalet fel i repositoryt, antalet brancher samt antalet *merges* inte skulle ge något kvalitativt resultat.

Slutligen vill vi påpeka att utökad *SCM*-undervisning leder otvivelaktigt till ett bättre arbetssätt och sparar tid som kan användas till att implementera fler *stories* samt göra produkten stabilare. Även om inte alla praktiker följdes till punkt och pricka under projektet så är vi övertygade om att teammedlemmarna lärt sig fördelarna med ett strukturerat arbetssätt samt att de garanterat kommer närma sig framtida projekt på ett mer analytiskt sätt och lägga ner tid på att etablera en gedigen utgångsstruktur.

Referenser

1. K. Beck ”Extreme Programming Explained: Embrace Change”, Addison-Wesley Publishing Company, 2000.
2. G. Hedin, L. Bendix & B. Magnusson, “Teaching Extreme Programming to Large Groups of Students”, *Journal of Systems and Software*, 2003.
3. U. Asklund, L.Bendix, T. Ekman, “Software Configuration Management Practices for eXtreme Programming Teams”, Department of Computer Science, Lund Institute of Technology.
4. Peter H. Feiler: Configuration Management Models in Commercial Environments, Technical Report SEI-91-TR-7, Software Engineering Institute, 1991.
5. Bohner and Arnold: An Introduction to Software Change Impact Analysis, in Software Impact Analysis, IEEE, 1996.
6. U. Asklund, L.Bendix, T. Ekman, ”Configuration Management for eXtreme Programming”, Department of Computer Science, Lund Institute of Technology.