GIT som alternativ till CVS/SVN i agila utvecklingsmiljöer

Kristofer Jacobson, Patrick Ivarsson

Abstract

En studie om versionshanteringssystemet Git och om möjligheten att använda det som alternativ till CVS/SVN. Studien baseras på ett experiment utfört på ett programvaruprojekt. Projektet är en kombination av två kurser som ges av Lunds Teknologiska Högskola. Frågan som ska besvaras är om Git är enkelt nog och har tillräcklig funktionalitet för att ersätta CVS/SVN som är kursledningens val av system.

1 Introduction

Open Source projekt är något som med åren har blivit väldigt populärt. För många kommerciella program finns idag open source alternativ. Webbläsare, ordbehandlare och programmeringsspråk är alla exempel på detta. Ett exempel på detta är Microsoft Internet Explorer som år 2002 användes av 85,8% av alla internetbesökare [1] men som idag har fått lämna plats åt bland annat Mozilla Firefox och Google Chrome som båda är open source. Firefox och Chrome står nu tillsammans för 72,4% av alla internetbesök och det ökar för varje månad.

För att kunna arbeta effektivt i ett open source projekt behövs en typ av versionshanteringssystem (VCS) som organiserar och för historik över projektfilerna. Vilket VCS är då vanligast inom open source. Microsoft har undersökt frågan i en enkät som över 1000 utvecklare besvarat [2]. Undersökningen visade att år 2011 var Git det mest populära verktyget oavsett vilken plattform som användes. Samma undersökning hade genomförts året innan och visade att Gits popularitet hade ökat kraftigt. Vad detta beror på är vad vi vill ta reda på i den här djupstudien.

När man läser till Civilingenjör i Datateknik på Lunds Tekniska Högskola går man en kurs som heter Programvaruutveckling i Grupp (PVG) där ett mjukvaruprojekt utvecklas av en grupp med 8-10 utvecklare under en sju veckors period. Som VCS har de kursansvariga valt att använda systemet Apache Subversion (SVN). Gruppen leds av två coacher som hjälper gruppen att komma framåt och utvecklas. I vår roll som coacher väljer vi att istället använda oss av Git som VCS och se hur bra det fungerar. Våra tidigare erfarenheter av VCS är med SVN så vår studie kommer gå ut på att först lära oss Git och sen försöka lära ut det till gruppen. Kan Git göra samma jobb som SVN lika bra eller kanske bättre?

Studien är strukturerad med en kort förklaring av hur vår situation ser ut först. En beskrivning av hur gruppens kurs samspelar med coachernas kurs. Därefter följer en kort beskrivning av vad ett VCS bidrar med samt en översiktlig beskrivning av de system som behandlas i studien. Efter det går vi igenom hur studien har genomförts för att sen avslutas med en

resultat- och slutsatsdel. För att lättare följa med och förstå texten i studien finns det i Sektion 6 en terminologitabell där begrepp förklaras kortfattat.

2 BAKGRUND

Här följer en beskrivning över hur de två kurserna samspelar samt lite mer ingående hur de olika VCS fungerar och skillnader mellan dem.

2.1 VÅR SITUATION

Projektet som vi ska utföra vår studie på är ett delat projekt mellan två kurser. Den ena kursen går ut på att utveckla ett mjukvaruprojekt över en sju veckor lång period. Den andra kursen går ut på att gå in i en ledarroll och coacha en av grupperna under deras projekt. Projektet ska simulera ett verkligt utvecklingsprojekt med en kund och Extreme Programming (XP) [4] som utvecklingsmetod. Gruppen kommer att använda sig utav utvecklingsverktyget Eclipse och programmeringen sker i Java. Gruppen har innan projektets början fått lära sig XP och SVN genom föreläsningar och laborationer. För att testa koden under laborationerna kommer JUnit att användas som verktyg.

2.2 VERSIONSHANTERING

I mjukvaruprojekt behöver utvecklarna dela filer och dokument med varandra. I ett mindre projekt med ett fåtal utvecklare går det att komma undan med att skicka filer via email och att tillsammans sitta och sammanfoga dokument. Detta kan med tiden bli mycket tidskrävande och komplicerat med många versioner av samma dokument i omlopp. Det är i ett sådant läge som ett VCS kan användas och förenkla arbetet. Ett VCS tillåter att ett godtyckligt antal utvecklare jobbar med samma projekt och delar alla projektfiler på en plats, även kallat ett repositorie (hädanefter refererat till som repo). När en fil ändras sparas informationen om vad som har ändrats samt att filen får ett nytt versionsnummer. På detta vis håller systemet reda på vilken som är den senaste versionen av en fil samt möjliggör för utvecklarna att återställa en tidigare version av filen.

En ytterligare funktion i VCS är möjligheten att slå samman två versioner av samma fil, även kallat merge. Om flera utvecklare arbetar i samma version av en fil samtidigt och försöker spara den i det gemensamma repot kommer en sammanslagning att behöva göras. Vid enklare ändringar i filen kan ett VCS bidra med att automatiskt slå samman filerna. Däremot om ändringarna är komplicerade eller på samma ställe i filen markerar systemet konflikten och ber utvecklaren att lösa konflikten.

I projekt kan det ibland finnas behov av att utifrån en punkt utveckla systemet parallellt men åt olika håll. Med ett VCS kan man då skapa en förgrening av programmet, även kallad branch. Ett exempel på detta är om utvecklargruppen förbereder för en release av mjukvaran. Då behövs en stabil version av mjukvaran där funktionaliteten är helt färdigställd. För att inte hela projektet ska stanna upp för detta kan man välja att göra en branch när projektet är i ett stabilt läge. Releasearbetarna kan då arbeta på den nya branchen utan risk för att ny, halvfärdig kod följer med i releasen. De andra utvecklarna kan fortsätta att arbeta som tidigare och vid ett senare tillfälle kan man välja att återigen slå ihop branchen med huvudprojektet.

2.3 CVS/SVN

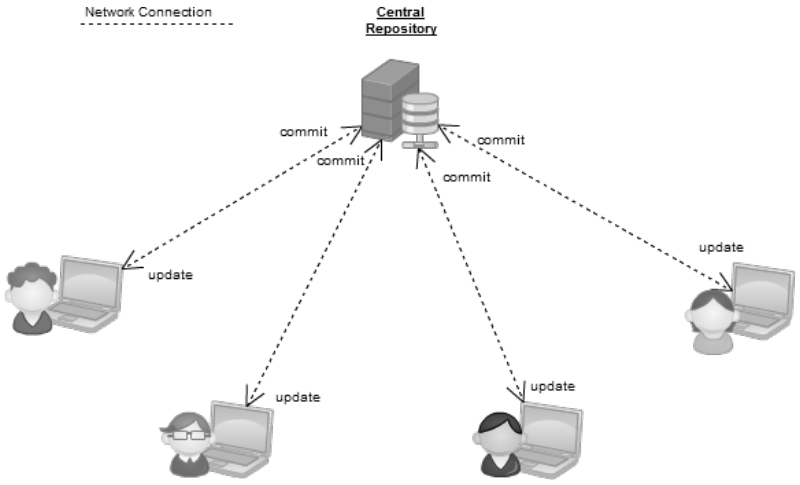

Fig. 1: Visar upplägget i ett centraliserat versionshanteringssytem. Alla användare kopplar upp sig mot ett gemensamt repo

Datavetenskapsinstitutionen har valt att lära ut ett VCS kallat "Concurrent Versions System" (CVS) till PVG-projektet. CVS lanserades år 1990 och är ett centraliserat VCS (CVCS). Att det är centraliserat innebär att det har ett centralt repo som alla utvecklare kopplar upp sig mot (se Fig. 1). I repot finns alla filer samt information om alla ändringar som gjorts. När en utvecklare checkar ut projektet skapas en lokal kopia av det på utveklarens dator. Därefter görs alla ändringar lokalt och när koden är klar laddar utvecklaren upp den i repot och löser eventuella merge-konflikter. Skulle ny kod bli uppladdat på repot går det att göra en uppdatering och på så sätt få de senaste ändringarna i sitt lokala repo. Allt detta kan göras via ett inbyggt användargränssnitt(GUI) i Eclipse.

CVS utvecklas inte längre men det finns många CVS-kloner som har samma funktionalitet som CVS. Klonerna är främst utvecklade för att lösa buggar och lägga till funktioner som saknades i den sista släppta versionen av CVS. Subversion (SVN) är en av dessa kloner och är systemet som används under PVG-projekten. Funktionerna som finns i CVS finns även i SVN och fungerar analogt. Även för SVN finns det GUI att installera till Eclipse för att grafiskt hantera projektfilerna.

2.4 GIT

För att hantera källkoden till Linuxkärnan valde Linus Torvalds att utveckla sitt eget system [5]. Tidigare hade ett system kallat BitKeeper används men när projektets gratislicens gick ut behövdes ett nytt gratis VCS. Enligt Torvalds kunde inget av de dåvarande gratisalternativen leverera den funktionalitet som behövdes så han valde att istället utveckla sitt eget. Det beslutet resulterade i Git.

Till skillnad från CVS och SVN är Git ett decentraliserat VCS(DVCS). Detta innebär att det inte finns ett gemensamt repo som alla utvecklare använder sig av. Istället delar utvecklarna filerna mellan sig (se Fig. 2). Alla utvecklare har ett eget lokalt repo med alla filer och all filhistorik och när man behöver den senaste versionen av koden hämtar man den

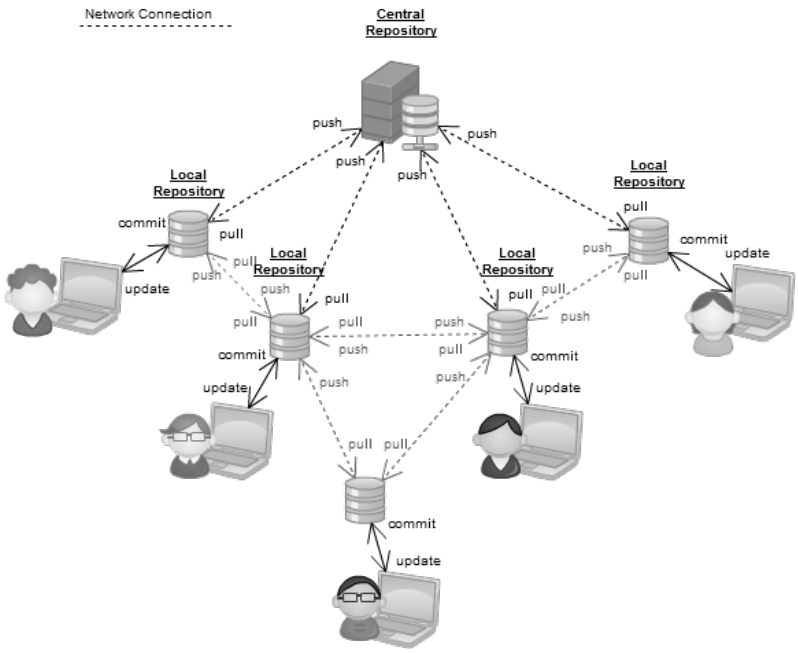

Fig. 2: Visar upplägget i ett decentraliserat versionshanteringssytem. Alla användare kan koppla upp sig mot ett centralt repo men kan också dela filer mellan varandra

från de andra utvecklarna. När en utvecklare själv har gjort ändringar kan han trycka ut den nya koden till de andra utvecklarna. Git kan även användas som ett centraliserat VCS genom att alla hämtar från och trycker upp uppdateringar på en server. En av fördelarna med ett decentraliserat system är att två personer kan arbeta med en del i projektet och dela experimentell kod mellan sig utan att det påverkar resten av teamet.

Arbetsflödet i Git påminner mycket om det som används i CVS/SVN. Man arbetar med det lokala repot på samma vis som man arbetar med ett repo i CVS/SVN. Varje gång man gjort ändringar i koden gör man en commit till det lokala repot. När koden är redo att skickas ut till de andra utvecklarna trycker man ut koden till dem, vilket kallas att göra en push. Genom att commita till sitt eget repo får man även lokalt en historik som är enkel att navigera i då man alltid kan återställa filerna till en tidigare commit.

Git kan användas både genom terminaler och genom grafiska GUI. Det finns både plugins och fristående program för att hantera sitt repo. Med installationen av Git medföljer även två simpla GUI som kan användas för att skapa och hantera branchar. Till Eclipse finns pluginet EGit vilket medför att man kan hantera sitt repo inifrån Eclipse.

3 Utförande

3.1 Upplägg

Vi lade upp arbete så att vi inte skulle lägga för mycket fokus på att använda Git som VCS då det är mycket ny information. Vårt team som vi skulle utbilda har inte varit i någon kontakt med Git innan men har inte heller några djuprotade arbetsätt med något annat versionhanteringssystem. Vi har delat upp upplägget för gruppen i 3 faser. Första fasen är

rent teoretisk och vi försöker hålla den på så grundläggande nivå som möjligt. Vi måste ständigt hålla i åtanke att för de flesta i gruppen som vi introducerar Git till har aldrig arbetat med något projekt innan av denna storlek. Vilket innebär att de har fullt upp med mycket annat och ger vi dem för mycket teori så kommer den ändå inte befästas. Gruppen har redan en god insikt till vikten av att kunna arbeta parallellt och vill lära sig mer om det.

Då vi också var nya till Git bara någon månad tidigare så förstår vi deras situation och har lättare för att skriva ner en kort guide på de mest nödvändigaste kommandona för att kunna arbeta med vårt projekt [Appendix A]. Denna guide delar vi ut på första planeringsmötet så alla kan direkt komma igång med att klona ner och arbeta med koden på första långlabben. Vi tilldelar även en i gruppen att läsa på mer teori och även testa lite själv, på så vis har de en VCS expert inom gruppen då det kan kännas lättare att prata med varandra.

Fas två uppstår vid första labben och för det flesta är därmed deras mer teoretiska lärande färdigt. Här efter kommer de uteslutande lära sig av vad de ser och gör. Det vill säga att de ser vilka fördelar Git har och i vilka situationer man använder sig av olika funktioner. I fas två är gruppen fortfarande väldigt outbildad vilket leder till att vi coacher går runt och hjälper dem vid mer komplicerade kommandon och händelser. Till att börja med så behöver de inte lära sig dessa alls utan bara kunna det som gavs till dem i fas ett. Allt efter som så får de själva skriva in kommandona och vi står bakom och instruerar och bara kontrollerar så att allt görs rätt. Vissa i gruppen som inte vill lära sig så mycket om Git kommer stanna i denna fas.

Efter några veckor av labbande kommer fler och fler i gruppen att ta till sig arbets sättet och inte tillkalla oss då de redan vet vilka kommandon som ska skrivas. De är då som fas tre inleds med mycket mer självgående användande utav Git. De vet själva vad som ska göras och kommer själva med förslag när de vill brancha för att till exempel arbeta i tätt samarbete med en annan grupp för att uppnå ett bättre resultat snabbare. För sådana strukturella beslut som att det går snabbare att två grupper branchar ut och arbetar tillsammans godkänner vi det först, men inte för att de inte tekniskt klarar det utan att vi fortfarande kan utvärdera om det finns någon vinning i att göra det.

Det är för det mesta väldigt säkert att låta grupperna själva experimentera själva så fort de har lärt sig grunderna eftersom Git är väldigt robust och man kan nästan alltid återskapa det mesta.

Detta speciellt då vi har minst fem stycken lokala repon med komplett historik samt en server. Följer de även den utdelade guiden och commitar ofta upp till sina egna lokala repon så kan man återställa nästan alla fel.

3.2 PLANERINGSMÖTEN

I slutet av varje långlabb ska studenterna skriva ett par meningar om den specifika XP practice som de har fokuserat på. Vi har även lagt till att de också ska skriva om saker och ting som går bra och mindre bra med gruppen och vårt arbete, där vi belyst att om det är några problem med Git så ska de gärna ta upp dem här då vi coacher lättare kan ta tag och åtgärda

dem. Dessa punkter tas sedan upp på våra schemalagda planeringsmöten för diskussion och reflektion. Vi kommer här med förslag till lösningar på hur vi ska åtgärda dessa och om någon i gruppen har något sätt. Här ser vi om vi till exempel skulle behöva skriva en ytterligare guide för att sätta in gruppen i de lite svårare funktionerna eller om de kan lära sig dem som beskrivet ovan i 3.1

I slutet av planeringsmötena delar vi ut spikes till studenterna och om det har varit något som flera har uppfattat som svårt eller problematiskt inom Git så sprider vi kunskapen genom att någon gör ett grundligare undersökning varför det blev så och hur vi ska göra i fortsättningen. För det absolut mesta så vet vi coacher redan detta men vi trycker hårt på att teamet ska besitta all kunskap själva och inte vara beroende utav oss coacher. Det kan även förekomma flera tillfällen då gruppen tycker att det är lättare eller smidigare att fråga varandra istället för oss coacher.

Det är viktigt att vi alla i gruppen också ska kunna ha tillgång till koden hemma för att kunna göra eventuella spikes så som till exempel kod granskning eller undersöka förslag till hur det är bäst att vidare utveckla en viss gren av programmet. För att detta ska fungera så har vi skrivit ihop en kort guide på den gemensamma trac-hemsidan. Där har vi också lagt upp ett shell-script som konfigurerar Git till deras inställningar utan att studenterna själva behöver skriva några kommandon.

Under de senare veckorna då gruppen arbetar mer självständigt med Git så skapar vi brancher även för de spikes som kräver något ur repot men även ska uppdatera repot till en ny version. Ledningen av kursen har beslutat att det får inte skrivas någon kod som trycks upp i repot mellan långlabbarna men gruppen får till exempel uppdatera JavaDoc eller teknisk dokumentation Det är viktigt för oss att dessa spikes inte blir för många och att de inte jobbar direkt på vår huvudbranch (master) utan att vi innan långlabben börjar kan merga ihop brancherna. Dessa merges ska alltid gå automatiskt då det inte ska vara flera brancher med samma ändringar. Till exempel så kan vi en vecka ha en spike med JavaDoc uppdateringar och en spike med att uppdatera manualen. Dessa två blir brancher för att kunna arbeta hemifrån utan att störa folk som vill ladda ner koden och få en halvt färdig skriven JavaDoc eller manual. Direkt då långlabben startar tar man en utav brancherna och mergar ihop den med master. Detta blir i git endast en fast forward och inga merge konflikter då det är bara en person som har ändrat. Sedan går vi till den andra spiken och gör det samma. Git löser denna merge-konflikten automatiskt då den tekniska dokumentationen inte har gjort några ändringar i några Java-filer och JavaDocen inte ska ha gjort några ändringar i den tekniska dokumentationen.

Ledningen för kursen har även godkänt att större refaktoriseringar utav hela programmet arkitekt kan vara bra att göra hemma då det förövrigt är produktionsstop då det annars skulle bli väldigt stora och svåra merge konflikter. När vi har dessa spikes är det viktigt att den spikegruppen är den ända som arbetar med koden och att alla har en god förståelse på vilka ändringar som kommer att göras och hur systemet kommer att se ut efteråt. Det är viktigt att lägga detta arbete på en egen branch då arbetet kan göras stegvis med flera commits under tiden. Att göra refaktoriseringar stegvis är inte bara bra för att andra får lättare att in-

tegrera sin kod utan att man lättare själv kan gå tillbaka om något inte blir bra. En annan mycket viktig aspekt på varför refaktoriseringen inte får ske på huvudbranchen är om den inte skulle lyckas bli klar så ska det inte vara några problem och alla kan bara arbeta på som vanligt. Skulle refaktoriseringen sedan bli klar någon timme in under labben så kan man fortfarande merga ihop den då. Det medför dock att man måste vara noggrann med den funktionaliteten som har tillkommit under tiden.

3.3 GIT KONFIGURATION

Git är mycket fritt och kan användas på många olika sätt där ingen är bunden att ställa sig i en hierarki. Vi har dock valt att arbeta med Git på ett lättförståligt sätt där alla hämtar(pull) det senaste från en server. På detta vis löser vi det krångliga som kan uppstå med att hämta från flera håll. Alternativet skulle vara att när en grupp har blivit klar med en story så hämtar alla ner hans version och mergar ihop med sitt eget. Här blir det dock lite rörigt i vår situation då vi enligt XP anda gör par byten och det slutar snabbt med att man inte har någon aning om vilken dator och inloggning man sitter på. Ett annat vanligt sätt att arbeta på är att ha en så kallad Gatekeeper [11] som alla hämtar ifrån. När ett par av utvecklare är färdiga med en story så säger man till Gate keepern som då hämtar från deras, kontrollerar ändringarna och kanske testkör programmet. När han är nöjd säger han till alla att det finns en ny version hos honom som alla ska hämta. Detta sättet är för visso mer säkert och röd kod sprider sig inte när någon har gjort fel. Dock så tycker vi coacher att detta inte känns så agilt och den snabbheten som är så viktig går lite förlorad. Det betyder även att teamet är beroende av denna enda “bättre vetande” personen som skall granska alla arbeten, vilket vi inte heller tycker går helt enligt XP. Vi vill med vår grupp uppnå en väldigt självgående sammansättning där vi coacher inte ska ha någon avgörande roll, och vi vill även att gruppen ska fungera vi eventuella bortfall. Vi har därför valt det upplägget som vi har och delar istället ut en task på att något annat utvecklingspar granskar en story innan den anses helt färdig.

3.4 UNDERSÖKNINGAR

För att mer konkret kunna analysera vår utvekling och ge oss feedback om hur bra matrialet och vår undervisning är så har vi lämnat ut en enkät [Appendix B]. I enkäten försöker vi adressera de olika faserna och se hur framgångsrika de har varit. Vi har ställt enkla frågor för att se hur lätt det var att ta till sig teorin i fas ett. Att komma igång med ett naturligt arbetsflöde i fas två. Vi undersöker huruvida de känner sig bekväma med Git och får djupare försåelse för det genom terminalen i steg tre, samt hurvida de skulle kunna tänka sig att arbeta med det i framtiden för en eventuellt fas fyra. Vi ger ut enkäten två gånger under kursens gång för att kunna analysera några förändringar. Den första gången är efter cirka halva tiden då de flesta i gruppen är i mitten eller slutet av fas två och den andra gången vi undersöker är på sista mötet då allt arbete med Git ska vara avklarat.

4 RESULTAT

Då vi endast är halvvägs genom projektet har vi inte hunnit samla in den data som behövs för att göra en resultatdel. Här kommer vi presentera resultat och reflektioner. Detta inkluderar resultaten från enkätundersökningar som görs vid 2 tillfälle under projektets gång.

4.1 GIT i AGILA UTVEKLINGSPROJEKT

4.2 UTVECKLING

4.3 ENKÄTUNDERSÖKNING

4.3.1 Snabbguiden gjorde det enkelt att förstå grunderna i Git

4.3.2 Det är lätt att få igång ett bra workflow (liknande update, update, update, commit för SVN)

4.3.3 Att använda Terminalen gör att man förstår bättre hur Git fungerar

4.3.4 Git hanterar mergkonflikter på ett bra sätt

4.3.5 Git fungerar bra tillsammans med Eclipse

4.3.6 När vi stöter på problem med Git kan ni som coacher ofta lösa det

4.3.7 Mitt helhetsintryck av Git är att det är ett bra versionshanteringssystem

4.3.8 Jag kommer fortsätta använda Git som versionshanteringssystem

5 SLUTSATS

Samma som ovan. Kommer innehålla allt vi kommit fram till under projektet.

5.1 GIT SOM STANDARD I XP PROJEKT

Vi har under denna kurs och studie kommit fram till att Git fungerar utmärkt som versionshantering system i agila utveklingsmiljöer. Vi har också fastslagit att det är möjligt att introducera Git, om även på ett mycket grundläggande plan, till en grupp studenter tillika utvecklare som aldrig arbetat med det innan. Verktyget har överlag fått god till mycket god kritik av gruppen. Vi visar härmed att Git skulle vara ett bättre alternativ än nuvarande SVN för kursen med bättre framtidsutsikter vilket är viktigt för blivande Civilingenjörer.

6 TERMINOLOGI

Uttryck	*Betydelse*
Versionshanteringssystem (VCS)	System som hanterar historik för källkod och dokument.
Repositorie (Repo)	Server där projektets dokument, filer och historik lagras.
Merge	När man slår ihop två versioner av samma dokument eller fil.
Branch	En kopia av projeket som utvecklas för sig själv, parallellt med projektet.
Centraliserat VCS (CVCS)	VCS där alla utvecklare kopplar upp sig mot ett gemensamt repositorie.
Decentralisert VCS (DVCS)	VCS där alla utvecklare kan välja att koppla upp sig mot ett gemensamt repositorie eller att dela projektfiler mellan sig.
Concurrent Versions System (CVS)	Är ett centraliserat VCS som idag inte utvecklas längre.
Subversion (SVN)	System som bygger på CVS men som fortfarande utvecklas aktivt.
Git	Är ett decentraliserat VCS.
Java	Är ett programmeringsspråk som används i studien.
Commit	Termen i CVS/SVN för när en utvecklare har ändrat i en fil och lägger upp ändringen i VCS.
Push	Term för när en utvecklare som använder Git trycker ut sitt repositorie till andra utvecklare.
Update	Används i CVS/SVN när en utevecklare hämtar hem de senaste versionerna av projektet.
Pull	Samma som Update för CVS/SVN men för Git.
eXtreme Programming (XP)	Utvecklingsmetod framtagen av Kent Beck.
Story	När en funktion ska utvecklas i XP skrivs denna som en story som beskriver vad som ska göras.
Task	När en story delas upp i mindre problem blir dessa tasks.
Spike	Egentid för att experimentera fram lösningar eller ideer.
Refaktorisering	När kod förenklas och förbättras genom att minska komplexitet och göra saker mer tydliga.
Gate Keeper	Vid användande av Git kan alla nya ändringar skickas till en person som granskar och därefter godkänner eller avslår ändringarna.
JavaDoc	Dokumentation som beskriver en Java-klass och alla dess publika metoder.
JUnit	Program som kan användas för att skriva enhetstester för Java-program.

APPENDIX

REFERENCES

[1] Browser Statistics. W3Schools.com. http://www.w3schools.com/browsers/browsers_stats.asp.(February 2012)

[2] Open Source Developer Preferences Survey. Microsoft Developer Network (MSDN). http://blogs.msdn.com/b/codeplex/archive/2011/07/11/survey-results-open-source-developer-preferences-june-2011.aspx (June 2011)

[3] Beck, K., *Test Driven Development – by Example*. Boston: Addison Wesley, 2003.

[4] Beck, K., *Extreme Programming Explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.

[5] McMillan, R. *After controversy, Torvalds begins work on "git"*. http://www.infoworld.com/t/platforms/after-controversy-torvalds-begins-work-git-721. (April 2005)

[6] Bryan O'Sullivan. *Making sense of revision-control systems*. Commun. ACM 52, 9. (September 2009)

[7] Yip, Chen & Morris. *Pastwatch: a Distributed Version Control System*. MIT Computer Science and AI Laboratory. (May 2006)

[8] A. Mockus.*Amassing and indexing a large sample of version control systems: Towards the census of public source code history*. Avaya Labs. Res. (May 2009)

[9] Hedin, Bendix & Magnusson. *Teaching extreme programming to large groups of students*. Elsevier. (2005)

[10] Asklund, Bendix & Ekman. *Software Configuration Management Praticies for eXtreme Programming Teams*. Department of Computer Science, Lund Institute of Technology. (August 2004)

[11] Brian Berliner and Nayan B. Ruparelia. 2010. Early days of CVS. SIGSOFT Softw. Eng. Notes 35, 5 (October 2010), 5-6. DOI=10.1145/1838687.1838689 http://doi.acm.org/10.1145/1838687.1838689