HAAGA-HELIA
ammattikorkeakoulu

Jatkuva integraatio – CASE: Sovellusalustan julkaisuprosessin kehittäminen Jenkinsillä

Ville Stranius

Opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

2014

HAAGA-HELIA ammattikorkeakoulu

Tiivistelmä

17.11.2014

Tekijä
Ville Stranius

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko	**Sivu- ja liitesivumäärä**
Jatkuva integraatio – CASE: Sovellusalustan julkaisuprosessin kehittäminen Jenkinsillä	20 / 3

Opinnäytetyön otsikko englanniksi
Continuous integration – CASE: Improvement of an applications platform release process using Jenkins

Capgemini Finland Oy toimittaa ulkoiselle asiakkaalle sovellusalustan ylläpito- ja jatkokehityspalvelun. Opinnäytetyön tavoitteena oli kehittää sovellusalustan julkaisuprosessia. Manuaalisia työvaiheita haluttiin karsia ja korvata nämä sovellusautomatiikkaa hyödyntämällä. Tällä tähdättiin riskien vähentämiseen virhealttiutta pienentämällä, sekä julkaisuprosessin keskimääräisen suoritusajan nopeuttamiseen. Sovellusalustan julkaisuprosessin kehittäminen toteutettiin osana palvelunlaajuista jatkuvan integraation kehityshanketta.

Teoriaosuudessa tutkitaan jatkuvaa integraatiota käsitteenä ja läpikäydään sen periaatteita, kuten esimerkiksi kaikkien soveltuvien työvaiheiden sovellusavusteinen automatisointi. Versionhallintaa käsitellään siinä laajuudessa, mikä on tarpeen, jotta voidaan ymmärtää jatkuvaa integraatiota noudattava sovelluskehitysprosessi. Teoriaosuuden lopuksi esitellään toteutusosiossa hyödynnetty vapaan lähdekoodin jatkuvan integraation työvälineratkaisu Jenkins. Lisäksi analysoidaan Jenkins-työkalun soveltuvuutta eri käyttötarkoituksiin, sekä tarkastellaan aiempia hyödyntämiskohteita.

Jenkinsin automatisointi ominaisuuksia hyödyntäen saatiin tavoitteen mukainen lopputulos. Irrallisia julkaisutoimenpiteitä saatiin nivottua Jenkins-ajoihin sisällytettävän logiikan ja komentojen avulla yhteen. Abstraktina tuotoksena sovellusalustalle syntyi uusi tehostettu julkaisuprosessi. Julkaisun virhealttiuden pienentäminen parantaa sovellusalustapalvelun toimitusvarmuutta. Julkaisuprosessin nopeutuminen on puolestaan tekijä palvelun laadun pitämiseen korkeana, sillä ajoittain tarve alustan julkaisemisesta muodostuu lyhyellä varoitusajalla. Tällöin julkaisuversio yleensä myös tarvitaan käytettäväksi mahdollisimman nopeasti.

Jenkinsin käyttöönoton yhteydessä havaittiin olevan hyödyllistä miettiä käyttötarkoitus mahdollisimman pitkälle, sillä oletusasennus ei välttämättä tarjoa kovin erikoistuneita toimintoja, mutta Jenkinsin yhteisötuki on vahva ja erilaisia liitännäisiä on saatavilla runsaasti.

Asiasanat
Jatkuva integraatio, Jenkins, versionhallinta, ohjelmistokehitys

HAAGA-HELIA
University of Applied Sciences

Abstract

17.11.2014

Author
Ville Stranius

Degree programme
Bachelor’s Degree Programme in Information and Communication Technology

Thesis title	**Number of pages and appendix pages**
Continuous integration – CASE: Improvement of an applications platform release process using Jenkins	20 / 3

Capgemini Finland Oy delivers an applications platform management and development service to an external customer. The goal of this study was to improve the release process of the above applications platform. Manual work was expected minimized and the replacing activities were wanted to be carried out via software automation. The aim was to mitigate risks and reduce the average time it takes to perform the platform release. Improvement of the applications platform release process was done as part of a service wide continuous integration development undertaking.

The theory section focuses on the concept and principles of continuous integration, such as the recommendation to automate all applicable work tasks using software automation features. Version control is explored to the extent that is necessary to let the reader understand a software development process which follows continuous integration principles. In this section, the open source continuous integration tool Jenkins is also introduced, along with analysis of its feasibility in different use cases and an overview of previous real-life applications of Jenkins.

Utilizing the software automation features of Jenkins ensured the accomplishment of the study objectives. With the combination of Jenkins Jobs’ logic and commands the decoupled release actions were bound together resulting in the improved release process for the applications platform. Reduced risk in platform release ensures, in its own part, the service delivery. Hastened release process is a factor in delivering high quality service, as time to time the platform release is required on short notice, and as fast as possible.

It was noticed useful to approximate the possible use cases as accurately as possible when taking Jenkins into use. The default installation may not contain very specialized features, but the community support for Jenkins is strong, and a multitude of plug-ins is available.

Keywords
Continuous integration, Jenkins, version control, software development

Sisällys

1 Johdanto 1
2 Jatkuva integraatio 3
  2.1 Lähdekoodi ja versionhallinta 3
  2.2 Käsite ja ideologia 6
  2.3 Käytännön esimerkki 8
3 Jenkins 8
  3.1 Yleisesittely ja perustoiminnallisuudet 9
  3.2 Edut ja haitat 9
  3.3 Hyödyntäminen 10
4 Sovellusalustan julkaisuprosessin kehittäminen 12
  4.1 Työn toimintaympäristö ja tavoite 12
  4.2 Tekninen ympäristö ja työvälineet 13
  4.3 Toteutus 15
  4.4 Tuotos 17
Pohdinta 18
Lähteet 21
Liitteet 23
  Liite 1. Keskitetyn versionhallinnan 18 perustoiminnallisuutta 23
  Liite 2. Alkuperäinen julkaisuprosessi 24
  Liite 3. Kehitetty julkaisuprosessi 25

1 Johdanto

Työn toimeksiantaja on Capgemini Finland Oy. Capgemini on alun perin ranskalainen, nykyään globaali IT-konsulttiyritys, joka toimii 40 maassa ja jossa työskentelee yli 140 000 työntekijää. Capgeminin palvelutarjontaan kuuluu laaja kirjo konsultaatio- ja IT-palveluita, kuten sovellusylläpito, räätälöity sovelluskehitys, liiketoiminnan- ja -johdon konsulttipalvelut, järjestelmäintegraatioiden toteutus, IT-infrastruktuuriylläpito, IT-ulkoistuspalvelut jne. Suomessa toimivalla Capgemini Finlandilla on noin 600 työntekijää.

Opinnäytetyöaiheen taustalla on Capgeminin toimittama räätälöity sovellusylläpitopalvelu, jonka suoritteen osana ylläpidetään ja jatkokehitetään ulkoisen asiakkaan sovellusalustaa. Asiakkaalla on valmiina integraatioympäristö, mutta ympäristöön kuuluvien työvälinesovellusten automatisointitoiminnallisuuksia ei hyödynnetä käytettyjen työskentelytapojen kannalta parhaalla mahdollisella tavalla. Sovellusalustan julkaisuprosessin parantaminen toteutetaan osana palvelussa käynnistettyä jatkuvan integraation kehittämisprojektia.

Sovellusalustan julkaisuprosessi on tällä hetkellä merkittävältä osin manuaalista työtä. Sovellusalusta koostuu noin 20 alimoduulista, joiden jokaisen kohdalla täytyy toistaa tietyt samantyyppiset toimenpiteet. Opinnäytetyön tavoitteena on julkaisuprosessin kehittäminen, jolla tähdätään käsin tehtävän mekaanisen suorittamisen poistamiseen tai merkittävään vähentämiseen. Tähän pyritään hyödyntämällä integraatiotyövälineen (Jenkins) automatisointitoimintoja. Manuaalisen työn vähentämisen nähdään nopeuttavan sovellusalustan julkaisuprosessia, parantavan alustakehittäjien työmotivaatiota, sekä vähentävän prosessin virhealttiutta. Virheriskin pienentäminen on tärkeää esimerkiksi siksi, että julkaisupaketoinnin yhteydessä tapahtuvat virheet pakottavat pahimmassa tapauksessa aloittamaan paketointivaiheen uudestaan. Lisäksi virheen tarkan syyn selvittäminen voi olla haastavaa, jolloin kehitystiimi kuormittuu entisestään keskellä jo valmiiksi työlästä ja lisäksi bisneskriittistä julkaisuprosessia. Paketointivaiheen virheriskin pienentämisen ja manuaalisten työvaiheiden karsimisen oletetaan myös nopeuttavan julkaisuprosessin keskimääräistä suoritusaikaa.

Työ rajataan käsittelemään tarkemmin jatkuvaa integraatiota, jolloin lukijan oletetaan ymmärtävän yleiset ohjelmistointegraation periaatteet, sekä tuntevan ohjelmistotestauksen perusperiaatteet ja termistö osana ohjelmistokehityksen laadunvarmistusprosessia.

Versionhallintaa käsiteltäessä puhutaan pääosin yleisistä versionhallinnan toiminnallisuuksista ja käsitteistä. Versionhallinta voidaan kuitenkin toteuttaa keskitettynä- tai hajautettuna versionhallintana. Versionhallintatuotteen valinta määrää pitkälti käytetäänkö hajautettua vai keskitettyä versionhallintaa. Hajautettua versionhallintaa toteuttavaa versionhallintatyökalua käyttävät kehitystiimit kuitenkin käyttävät usein keskusarkistoa (Central Repository), mutta tietoisen valinnan, ei versionhallintaratkaisun saneleman pakon kautta (Sink 2011, 49).

Versionhallinnan havainnollistaminen ja konkreettiset esimerkit on laadittu keskitetyn versionhallinnan pohjalta, koska sen periaatteet on helpompi ymmärtää (Sink 2011, 63). Tämän opinnäytetyön kannalta keskeistä on jatkuvan integraation käsittely, minkä vuoksi hajautetun versionhallintamallin laajempi käsittely on rajattu työn ulkopuolelle. Luvun 2.1 lopussa kuitenkin sivutaan hajautettua versionhallintaa.

Tässä opinnäytetyössä esitellään aluksi jatkuvan integraation konsepti ja toimintaperiaatteet. Seuraavaksi tutustutaan toteutusosiossa käytettyyn avoimen lähdekoodin jatkuvan integraation työvälineeseen, Jenkinsiin. Tämän jälkeen kuvataan käytännön toimet sovellusalustan julkaisuprosessin parantamiseksi ja läpikäydään lopputulos. Lopuksi Pohdinta-luvussa tarkastellaan työnkulkua, kerrotaan toteutuksen yhteydessä heränneistä havainnoista ja esitetään johtopäätöksiä käsiteltyihin asioihin liittyen.

2 Jatkuva integraatio

Jotta voidaan ymmärtää, mistä jatkuvassa integraatiossa on kyse, tulee ensin ymmärtää lähdekoodin ja versionhallinnan käsitteet. Nämä esitellään tämän luvun ensimmäisessä alaluvussa, jonka lopussa mainitaan yleisempiä versionhallintatuotteita. Tämän jälkeen kerrotaan itse jatkuvan integraation (Continuous Integration) toimintaperiaate ja miten sitä hyödynnetään ketterässä (Agile) ohjelmistokehityksessä. Lopuksi kuvataan käytännön esimerkin kautta tyypillinen käyttötapaus, jossa on mukana jatkuvan integraation elementit.

2.1 Lähdekoodi ja versionhallinta

Lähdekoodi on ohjelmoijan tekstinkäsittelyohjelmalla tai ohjelmointityövälineellä kirjoittamia lausekkeita, jotka noudattavat käytetyn ohjelmointikielen mukaista syntaksia, eli muotosääntöä. Koodi tallennetaan tiedostoon ja tiedostoihin viitatessa puhutaan lähdekooditiedostoista tai yksinkertaisesti lähdekoodista. Joissakin ohjelmointikielissä lähdekoodi käännetään (compile) vielä sellaiseen muotoon, jota tietokoneen prosessori ymmärtää. Esimerkkeinä kääntämistä vaativista kielistä ovat Java, C++ ja C#. Kieli voidaan tulkita myös suoraan samasta muodosta, mihin ohjelmoija sen kirjoittaa. Näin toimii esimerkiksi selaimessa suoritettava JavaScript. (Rouse.)

Lähdekoodin perusteella voidaan luoda tietokoneen muistiin erittäin mutkikkaita oliokokonaisuuksia, joissa paljon liikkuvia osia ja keskinäisiä vaikuttimia. Lähdekoodi on kuitenkin ohjelmiston pysyvin ja muuttumattomin kuvaus. Tämä tila halutaan pitää tallennettuna lähdekoodimuodossa, sillä vähävirheisetkin sovellukset usein tarvitsevat usein parantelua ja toiminnallisuuslisäyksiä. Ohjelmiston muokkauksia on helpointa tehdä juuri lähdekoodiin. (Rouse.)

Versionhallintajärjestelmä on yleensä erillinen sovellus, joka auttaa ohjelmistokehittäjiä työskentelemään tiimissä. Versionhallinnalla on kolme perustehtävää. Ensimmäinen on kehitystiimin samanaikaisen kehittämisen mahdollistaminen, poistaen tarpeen odottaa työvuoroaan, mikäli tiimi työskentelee paljon samojen dokumenttien parissa. Rinnakkain tehtävästä työskentelystä saadut hyödyt halutaan maksimoida ja välttää tilanteita joissa yhden tehtävän suorittaminen vaatii ensin toisen tehtävän valmistumisen. (Sink 2011, 1.)

Toiseksi halutaan välttyä samanaikaisten muutosten tekemisestä aiheutuvista ristiriitatilanteista. Versionhallintajärjestelmät tarjoavat joukon menetelmiä näiden ei-toivottujen tilanteiden välttämiseksi, esimerkkinä lukitusmekanismit ja erilaiset

huomautukset käyttäjille. Kolmanneksi halutaan arkistoida täydellinen työhistoria versionhallintaan tallennettavien dokumenttien osalta. Jokainen koskaan olemassa ollut sekä uusi versio, esimerkiksi lähdekooditiedostoista tai muista dokumenteista, halutaan kirjattavan talteen. Halutaan myös olla selvillä siitä, kuka version teki, milloin se tehtiin ja mitä muutoksia versio sisälsi, eli miksi se tehtiin. (Sink 2011, 1.)

Versiot voidaan siis mieltää tiedoston tai muun versioitavan kohteen, kuten kansion tai kansiorakenteen, eri tiloiksi. Olennaisimpien tilanmuutokseen liittyvien tietojen, kuten lisäysten tai poistojen, lisäksi versiotiedot sisältävät tarpeellisiksi katsotun määrän lisätietoja. Versiotietoihin sisältyy tyypillisesti esimerkiksi version tekijä, version luonnin ajankohta ja version kuvaus. (Sink 2011, 1.)

Keskitetyllä versionhallinnalla on lukuisia toiminnallisuuksia. Sink (2011, 5-15) määrittelee 18 perustoiminnallisuutta, jotka on listattu kokonaisuudessaan liitteessä 1. Näistä Sinkin 18 perustoimminallisuudesta jatkuvan integraation kannalta olennaisin on sisäänkirjaus-toiminto (Commit). Sisäänkirjauksessa kehittäjä pyytää versionhallintajärjestelmää luomaan uuden version versiohallittavasta kohteesta, esimerkiksi lähdekooditiedostosta. Uusi versio sisältää kehittäjän paikallisessa työskentelyversiossa tekemät muutokset. Sisäänkirjaustoiminto voidaan moderneissa versionhallintatuotteissa tehdä usealle kohteelle samaan aikaan. (Sink 2011, 7.)

Sisäänkirjauksen vastapari on uloskirjaus (Check-out). Uloskirjauksessa kerrotaan versionhallintajärjestelmälle, että kehittäjä aikoo tehdä versionhallintakohteelle muutoksia paikallisessa työskentelyversiossaan. Tietyn tulkinnan mukaan uloskirjaus itsessään vasta synnyttää työskentelyversion. Tässä opinnäytetyössä työskentelyversiolla kuitenkin tarkoitetaan kehittäjän paikallisen ympäristön levylle tallennettuja tiedostoja, jotka ovat versionhallinnan piirissä. Uloskirjauksen yhteydessä voidaan useimmissa versionhallintatuotteissa määrittää lukitus, jolloin muut kuin lukon haltija eivät voi tehdä sisäänkirjausta kyseisestä versionhallintakohteesta ennenkuin lukitus on vapautettu. (Sink 2011, 6.)

Päivitystoiminto (Update) tekee kyselyn versionhallintajärjestelmälle uusimmista muutoksista, toisin sanoen versioista. Mikäli uusia versioita on saatavilla, nämä muutokset siirretään kehittäjän paikalliseen työskentelyversioon. Päivitystoimintoa tulee käyttää usein ja varsinkin ennen sisäänkirjausta, konfliktien ja niiden manuaalisen selvittelyn minimoimiseksi. (Sink 2011, 8.)

Versionhallintaa havainnollistetaan usein puurakenteen avulla. Päähaarassa eli puun rungossa (Trunk, Main Branch) on sellainen lähdekoodi, joka on testattua ja sovelluksen toiminnallisten vaatimusten täyttymiselle olennaista. Uusia suuria ominaisuuksia tai erinäisiä kokeiluita yleensä ohjelmoidaan erillisissä kehityshaaroissa (Branch). Vaikka kehityshaaran koodi olisi täysin toimivaa ja testattua, ei välttämättä ole varmuutta siitä, halutaanko kehityshaarassa rakennettuja ominaisuuksia osaksi virallista ja vahvistettua ohjelmakoodia, joka sijaitsee päähaarassa. Kun myönteinen päätös ominaisuuksien siirtämisestä päähaaraan ilmoitetaan, suoritetaan yhteenliittämistoiminto (Merge), jossa kehityshaaran versiot liitetään päähaaraan. Mikäli päätös on kielteinen, työ kehityshaarassa lopetetaan, eikä se vaikuta päähaaraan mitenkään. Tällöin kehityshaara voidaan haluttaessa jopa poistaa. (Sink 2011, 13-14, 176-180.)

Eräs versionhallinnan hyödyistä on se, että aiempia dokumenttiversioita ja niissä tapahtuneita muutoksia verrattuna muihin versioihin voidaan tarkastella milloin hyvänsä. Versiohistoriasta voidaan myös palauttaa mikä tahansa aiempi versio, esimerkiksi lähdekoodin tila, ja uloskirjata se työskentelyversioksi. Näiden ominaisuuksien vuoksi versionhallintaa voidaan pitää eräänlaisena aikakoneena dokumenttien, kansioiden tai lähdekoodin tallentamisessa ja hallinnassa. (Collins-Sussman & Fitzpatrick & Pilato 2011, 1, 14.)

CollabNetin (CollabNet Blog 2011) artikkelin mukaan suosituin versionhallintaohjelmisto on SVN (Subversion). Artikkeli mainitsee, että ”vanhaa dinosaurusta”, CVS:ää (Concurrent Versions System), käytetään yhä jonkin verran myös. Näin siitä huolimatta, että SVN luotiin varta vasten paikkaamaan edeltäjänsä, CVS:n, puutteellisuuksia (Collins-Sussman & Fitzpatrick & Pilato 2011, xiv). Molemmat SVN ja CVS ovat vapaan lähdekoodin ohjelmistoja. Joillakin kaupallisilla toimijoilla on myös omat, maksulliset, versionhallintatuotteensa, esimerkiksi IBM:llä IBM Rational ClearCase ja Microsoftilla VSS (Microsoft Visual Source Safe) sekä pilvipohjainen TFVC (Team Foundation Version Control).

Viime vuosina rajusti suosiotaan on kasvattanut myös niin ikään avoimeen lähdekoodiin perustuva versionhallintatuote Git. Git perustuu hajautettuun versionhallintamalliin, mikä tarkoittaa mm. sitä, että ei ole olemassa virallista versiota päähaarasta tietyllä keskusarkistolla (central repository) eli versionhallintapalvelimella, vaan jokaisella on vain keskitetyn versionhallintamallin työskentelyversiota vastaava oma paikallinen arkisto (local repository) ja muutoksia jaetaan suoraan näiden paikallisesti ylläpidettyjen arkistojen välillä (Git). Toinen laajassa käytössä oleva hajautettua hallintamallia käyttävä vapaan

lähdekoodin versionhallintaohjelmisto on Mercurial, vaikkei sen suosio ylläkään Gitin tasolle. (Zeroturnaround 2013.)

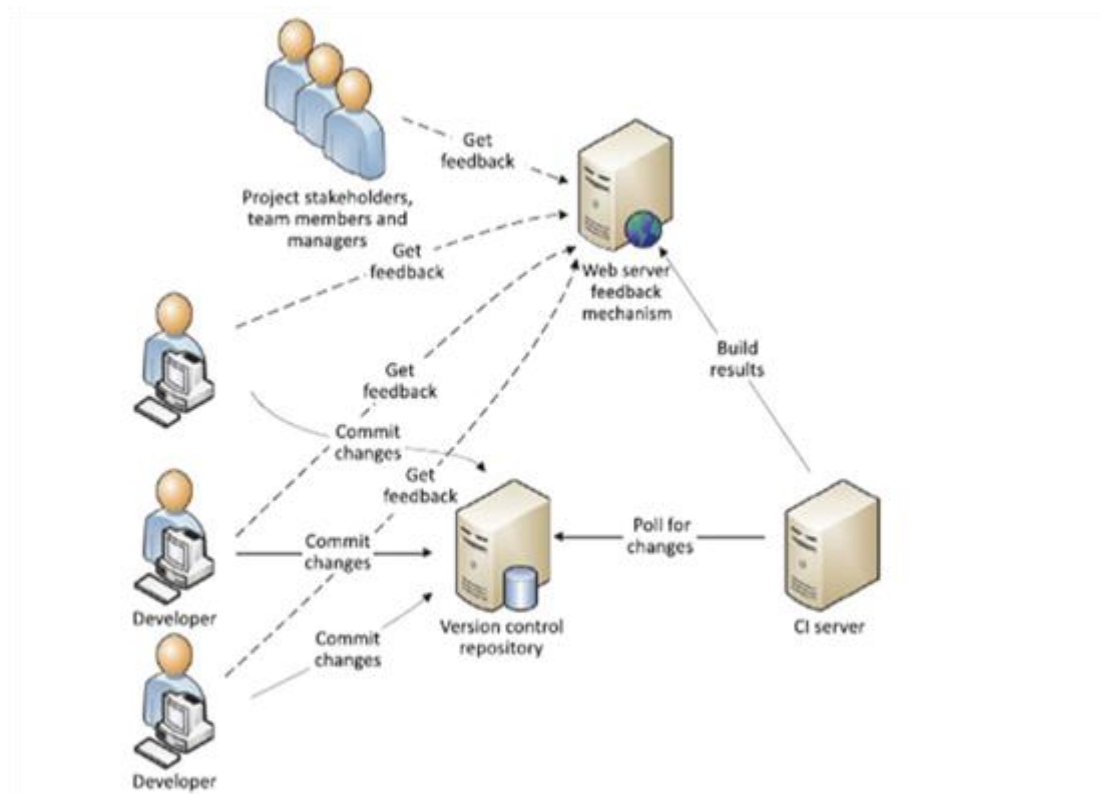
2.2 Käsite ja ideologia

Integraatio, eli tiivistettynä eri sovelluskehittäjien tuottamien ohjelma-osien yhteen liittäminen, tehtiin ohjelmistokehittämisen alkutaipaleella ohjelmistoprojektin loppuvaiheessa. Osien yhteen liittäminen oli teknisesti erittäin haastava projektin osa, joka vieläpä tapahtui kriittisessä vaiheessa, kun lähestyvä sovelluksen luovutustakaraja yleisesti aiheutti paineita kehitystiimille. Teknisen haastavuus aiheutui erillään kehitettyjen ohjelmapalasten käyttäytymisen- ja mahdollisten keskinäisten ristiriitojen vaikeasta ennustettavuudesta palasia integroitaessa yhdeksi kokonaisuudeksi. Integraatiovaiheen kestoa oli teknisen haastavuuden vuoksi vaikeaa tai mahdotonta arvioida, mikä aiheutti lisää riskejä jo ennestäänkin aikataulu- ja budjetissapysymishaasteissa kamppaileville IT-projekteille. (Fowler 2006.)

Jatkuva integraatio on ohjelmistokehityksen menetelmä, jonka perusperiaate on että sovellusintegraatiota tehdään usein, vähintään päivittäin, mielellään useammin. Tarkoituksena on mahdollistaa tämä pilkkomalla työt riittävän pieniin, muutamassa tunnissa toteutettaviin, osatehtäviin, jotka voidaan kuitenkin kokonaisuutena integroida versionhallintaan. Pienempien ohjelmaosien kanssa on helpompi työskennellä, koska integraation jälkeisten ohjelmavirheiden selvittäminen onnistuu helpommin, mitä vähemmän uutta, mahdollisia konflikteja aiheuttavaa, koodia on syntynyt. Jatkuvasta integraatiosta syntyy myös psykologisia hyötyjä. Kun vianselvitys ei ole ohjelmistokehittäjille enää entisen kaltainen suuri urakka ja pieniä osia ohjelmasta tulee jatkuvasti valmiiksi, tulee kehitystiimille tunne aikaansaamisesta ja ilmapiiri pysyy energisempänä. (Fowler 2006.)

Ohjelmistokehitysprojektien päämäärä on tuottaa mahdollisimman laadukas ohjelmisto mahdollisimman lyhyessä ajassa. Ohjelmistojen monimutkaistuessa ja laajentuessa toimintojen puolesta koko ajan suuremmiksi, on laatutavoitteisiin entistä haastavampaa päästä, huolimatta työkalujen kehittymisestä. Jatkuvan integraation periaatteita noudattamalla voidaan toteuttaa ohjelmistokokonaisuus hallitummalla tavalla. Hallintaa ja toimintojen tehostamista jatkuvassa integraatiossa saadaan aikaan sovellusautomatiikan hyödyntämisellä, joka on yksi jatkuvan integraation luonteenomaisimmista ilmentymismuodoista ja avaintekijöistä. (Kawalerowicz & Berntson 2011, 3-5.)

Palaute on keskeisessä roolissa jatkuvassa integraatiossa. Jatkuvan integraation yhteydessä puhutaan palautesilmukasta, jossa on kyse saada testien tulokset, toisin sanoen palaute, mahdollisimman nopeasti saataville (Betteley, 2011). Palautesilmukka ja sen nivoutuminen yhteen koko jatkuvan integraation ohjelmistokehitysprosessin kanssa on havainnollistettu kuviossa 1.



Kuvio 1. Jatkuvan integraation prosessi (MSDN 2013)

Suurimpana haasteena jatkuvalle integraatiolle on vaadittu tapamuutos, kun yhteen liittämistä tehdäänkin perinteisistä ohjelmistoprojekteista poiketen jatkuvasti. Kärsivällisyydellä ja ohjeistuksella on kuitenkin päästy pitkälle ja lähes jokainen jatkuvan integraation osaksi omaa työskentelytapaansa ottanut tiimi on kokenut sen hyödylliseksi. Lisäksi oppikirjan mukaan toteutettu jatkuva integraatio tarvitsee integraatiokoneen tai -palvelimen sekä mahdollisimman paljontuotantoympäristöä vastaavan testiympäristön. Näiden perustaminen ja toiminnallisiksi saaminen vaatii sekä laitteistoa että henkilötyötunteja. Lisäksi versionhallintajärjestelmän käyttäminen on vaadittu, jotta tehokas ja automatisoitu jatkuvan integraation ohjelmistokehitysprosessi pääsee oikeuksiinsa. (Fowler 2006.) Versionhallinnan käyttö on kuitenkin jo valtavirtaa ja lähes kaikki käyttävät jotain versionhallintatyökalua tai –järjestelmää hallitakseen samanaikaisen kehityksen tuottamien eri versioiden tallentamista (Purencool).

2.3 Käytännön esimerkki

Liitettäessä uutta lähdekoodia ohjelmistokokonaisuuteen, tyypillinen jatkuvaa integraatiota hyödyntävän ohjelmistokehitystiimin prosessikulku voisi olla seuraavanlainen. Ohjelmistokehittäjä on saanut valmiiksi ohjelman osan ja todennut lähdekoodinsa kääntyvän, eikä sisältävän suurempia virheitä, koska kääntämisessä käytetty koontiautomaatio-ohjelma (Build Automation Tool), esimerkiksi Apache Maven, suorittaa kääntämisen ja paketoinnin yhteydessä yksikkötestit. Mikäli ohjelmoija on tuottanut täysin uutta koodia, hän on ketterän ohjelmistokehityksen mukaisesti myös luonut uusia yksikkötestejä. Testit voidaan tehdä jo ennen koodin kirjoittamista, jolloin puhutaan testauslähtöisestä ohjelmistokehittämisestä (TDD, Test Driven Development).

Ohjelmoija sisäänkirjaa versionhallintaan uuden version lähdekoodeista tästä yksikkötestatusta tilasta, joka sisältää uudet koodilisäykset. Tämä käynnistää Fowlerin (2006.) kuvaaman, integraatiopalvelimella tai -koneella tapahtuvan ensimmäisen koontiajon (Commit Build), jossa kaikki sen hetkinen päähaaran lähdekoodi käännetään jälleen Mavenilla ja yksikkötestataan. Käännös ja yksikkötestaus tehdään integraatioympäristössä uudestaan siksi, että toiset kehittäjät ovat voineet lisätä päähaaraan omaa uutta koodiaan, mikäli nyt integroivalla kehittäjällä ei ole aivan tuoreinta päähaaraa paikallisessa ympäristössään. Mikäli päähaara kääntyy ja läpäisee yksikkötestit, paketoidaan se uudeksi koontiversioksi.

Tätä versiota testataan edelleen niin laajasti kuin on tarpeellista, tai niin kattavasti kuin integraatioympäristön resurssit riittävät. Mikäli kattavammat testit kestävät esimerkiksi useita tunteja ja kehittäjiä on useita, niin laitteiston ja ohjelmistojen suorituskykyrajat saattavat tulla vastaan. Vasta kun päähaaran uusi versio on läpäissyt tarvittavat testit, voidaan integraation todeta onnistuneen ja päähaaran olevan edelleen terve. Vasta tähän vaiheeseen päästäessä, voidaan Fowlerin (2006) mukaan katsoa ohjelmistokehittäjän työn uuden ominaisuuden kehittämisessä olevan valmiin.

3 Jenkins

Monipuolista Jenkinsiä hyödynnettiin opinnäytetyössä esimerkiksi koonnin ajastustyökaluna (Build Scheduler). Sovellusalustan julkaisuprosessi koostui monesta työvaiheesta usealla eri työväline ohjelmistolla. Jenkins toimi yhdistävänä tekijänä eri osien välillä. Tässä luvussa esitellään Jenkinsiä yleisesti, analysoidaan hyödyllisyyttä eri tarpeissa, sekä tarkastellaan aiempia sovelluskohteita.

3.1 Yleisesittely ja perustoiminnallisuudet

Sun Microsystems kehitti Hudson nimistä jatkuvan integraation palvelinratkaisua. Oracle osti Sunin 2009. Ilmeisesti Oraclen perusteellisempi päätöksentekokulttuuri sai hitaudensa vuoksi joitain Hudsonin avainkehittäjiä haarauttamaan Hudsonin toiseen avoimen lähdekoodin projektiin. Uusi projekti sai nimekseen Jenkins. (Proffitt 2011.)

Riekin (2013, 15) mukaan jatkuvan integraation prosessi saadaan aikaan yhdistelemällä Jenkinsin työyksiköitä, Jenkins Jobeja. Tässä opinnäytetyössä Jenkinsin työyksiköistä puhutaan jatkossa termillä Jenkins-ajo. Riekki kuvaa kookkaidenkin ohjelmistokokonaisuuksien jatkuvan integroinnin olevan mahdollista, sillä Jenkins-ajojen määrää ei ole mitenkään rajoitettu. Rajattomalla määrällä ajoja on siis mahdollista luoda hyvinkin monimutkaisia suoritusketjuja.

Jenkins on web-pohjainen jatkuvassa integraatiossa käytettävä sovellus, jolla voidaan tarkkailla toistuvasti suoritettavia tehtäviä. Esimerkiksi versionhallintaa voidaan tarkkailla ja muutoksien ilmaantuessa aloittaa automaattisesti näiden koonti. Jenkins on ilmainen ja lisensoitu MIT-lisenssillä (Laitinen 2011, 18). Laitinen kertoo konfiguroinnin ja ajojen luonnin hoituvan graafisen käyttöliittymän avulla. Jenkinsissä voi luoda ja nimetä erilaisia näkymiä. Näkymät ovat tarkoitettu Jenkins-ajojen kategorisointiin ja hallintaan. Jenkins-ajoja voi vapaasti siirrellä näkymältä toisille ja tietty ajo voi näkyä usealla eri näkymällä (Laitinen 2011, 34).

Jenkinsin ydintoiminnallisuus muodostuu Jenkins-ajoihin määritellyistä ominaisuuksista. Tällaisia voivat olla esimerkiksi kutsut muihin sovelluksiin, yhteistoiminta versionhallintajärjestelmän kanssa, ajojen laukaiseminen tai ajastaminen tietyin ehdoin ja parametrit, joita voidaan esimerkiksi välittää eteenpäin, kun yksi ajo käynnistää toisen. Jenkins-ajoihin voidaan myös liittää ehdollista logiikkaa, jolla voidaan tarpeen mukaan säädellä ja monipuolistaa ajojen ominaisuuksien suorittamisyhdistelmiä. (Jenkins 2014).

Mustaniemi (2013, 22) esittelee tyypillisen Jenkinsin käyttötavan, nightly buildin. Nightly build termiä käytetään kun Jenkins-ajo ajastetaan käynnistymään yöllä hakemaan uusimmat lähdekoodit versionhallinnasta ja kokoamalla ne paketiksi, jota vasten suoritetaan erilaisia testejä.

3.2 Edut ja haitat

Mustaniemi (2013, 14) listaa Jenkinsin hyödyiksi mm. helpon graafisen käyttöliittymän Jenkins-ajojen konfiguroimiseen, sekä Jenkinsin itsensä asentamisen verrattaisen

helppouden, asennusmedian ollessa vain yksi .war-tiedosto. Hän pitää myös lähdeoodin avoimuudesta, tästä johtuvasta ohjelmiston vapaasta muokattavuudesta sekä lisenssimaksuttomuudesta. Mustaniemi toteaa Jenkinsistä olevan hyötyä tuotantopalvelimien toiminnan testaamisessa. Sillä voidaan toteuttaa ns. pingaaminen, joka kertoo onko tietty palvelu toiminnassa vai ei (Mustaniemi 2013, 18).

Korhonen (2013, 22) mainitsee Jenkinsin eduksi lisäosien paljouden, mutta kääntöpuolena lisäosat ovat Korhosen mukaan myös Jenkinsin heikoin kohta. Esimerkiksi eri liitännäisen dokumentaatioissa ja yhteensopivuudessa havaittiin epätasalaatuisuutta, minkä arveltiin johtuvan siitä, että liitännäisillä on usein eri tekijä.

Laitinen (2011) oli todennut Jenkinsin sopivan parhaiten omaan projektiinsa, web-sovelluksen kehitysprosessin muuttamiseen noudattamaan jatkuvaa integraatiota. Laitinen mainitsi Jenkinsin eduiksi ilmaisuuden, tiheän kehitystahdin ja konfiguraation helppouden sekä yhteensopivuuden Maven koontityökalun kanssa. Toinen varteenotettava vaihtoehto Jenkinsille oli TeamCity-ohjelmisto mutta ilmaisen version rajoitukset koettiin pitkällä tähtäimellä ongelmallisiksi, eikä liitännäisten valikoima pärjännyt Jenkinsille. (Laitinen 2011, 18-19).

Farcic (2014) löytää Jenkinsistä hyvää ja huonoa. Korhosen tavoin liitännäisten paljous ja sitä kautta syntyvä laajennuspotentiaali on katsottu Jenkinsin hyödyksi, mutta samalla myös akilleen kantapääksi. Farcic toteaa laajennettavuutta tukevan arkkitehtuurin tulevan vakauden kustannuksella. Päivitykset aiheuttavat Farcicin mukaan toisinaan Jenkins-ajojen hajoamista tai muita ei-toivottuja vaikutuksia. Hän mainitsee Hudson-työvälineen olevan keskittyneempi yrityskäytössä arvostettuun vakauteen ja suorituskykyyn. Farcic kuitenkin toteaa Jenkinsin olevan eläväisempi ja dynaamisempi vahvemman käyttäjäyhteistötukensa ansiosta. Hän myös arvelee Jenkinsin olevan suosituin jatkuvan integraation työväline. Tämän vuoksi ongelmatilanteissa apu löytyy Farcicin mukaan helposti. Farcic päättää artikkelinsa yhteenvetoon, jossa hän kuvailee Jenkinsiä ohjelmiston koon, konfiguroinnin yms. puolesta raskaaksi pedoksi, mutta vastapainona löytyy Jenkinsin lähes rajattomat hyödyntämismahdollisuudet eri käyttötarkoituksissa. Kevyempiin tarpeisiin Travis-niminen jatkuvan integraation työväline on Farcicin oma suosikki.

3.3 Hyödyntäminen

Lähdeaineistoa Jenkinsin hyödyntämisestä löytyi kiitettävästi. Kontio (2014) esimerkiksi integroi Jenkinsin Robot Framework testiautomaatiokehykseen ja Jenkinsin rooli oli toimia

testiajojen ajastimena ja testidatan organisointivälineenä. Jenkis toimi web-käyttöliittymällisenä arkistona testitapauksille ja niiden tuloksille.

Riekki (2013) puolestaan toteutti Jenkins avusteisesti jatkuvan integraation kehitysprosessin tukiasemaohjelmiston systeemikomponentille. Lähes kaikki testaaminen saatiin hänen mukaan automatisoitua. Riekki myös kertoo Jenkinsin soveltuvan eri käyttöjärjestelmillä toteutettuun koontiin, koontiversioiden testaukseen ja lopulta testituloksien yhdistämiseen. Jenkins voidaan Riekin mukaan valjastaa myös piirtämään grafiikka, jolloin saadaan visualisoitua erilaisia trendejä Jenkins-ajojen suorituksessa.

Mustaniemi (2013) hyödynsi Jenkinsiä JavaEE ohjelmistoprojektien testausjärjestelmän luomisessa tarkoituksenaan tutkia ketterien menetelmien ja jatkuvan integraation yhteistoimintaa ohjelmistokehityksessä. Mustaniemi päätyi Jenkinsiin mm. siksi, että sillä voidaan koota .NET-sovelluksia. Hän tutki myös Mavenin yhteiskäyttöä Jenkinsin kanssa ja kertoi tällä yhdistelmällä suoritettavan projektinsa daily build, jonka päätteeksi sovelluspaketti asennetaan automaattisesti Weblogic palvelimelle Jenkinsin Weblogic deployer lisäosan avulla (Mustaniemi 2013, 18).

4 Sovellusalustan julkaisuprosessin kehittäminen

Tästä alkaa opinnäytetyön empiirinen osa. Aluksi kuvataan sovellusalustaan liittyvä toimintaympäristö sekä esitellään työn tavoite. Seuraavassa aliluvussa kuvataan sovellusalustaan liittyvä tekninen ympäristö. Teknisestä ympäristöstä löytyviä työvälineohjelmistoja ja niiden käyttötarkoituksia käsitellään sovellusalustan julkaisemiseen tarvittavilta osin. Kolmannessa aliluvussa käydään läpi julkaisuprosessin kehittämiseksi tehdyt käytännön toimenpiteet. Viimeisessä aliluvussa tarkastellaan tämän produktityyppisen opinnäytetyön tuotoksia.

4.1 Työn toimintaympäristö ja tavoite

Capgemini Finland Oy toimittaa ulkoiselle asiakkaalle sovelluskehitys ja –ylläpitopalvelua. Sovellusalustan ylläpito ja jatkokehitys on osa palvelun suoritetta. Sovellusalustan voi mieltää sovelluksena, joka ei itsessään tarjoa loppukäyttäjille käyttöliittymiä tai toiminnallisuuksia, mutta tarjoaa teknisen perustan ja toiminnallisuudet, jonka päälle loppukäyttäjäsovelluksia voidaan rakentaa. Tällaisella arkkitehtuurilla ei tarvitse esimerkiksi rakentaa jokaiseen loppukäyttäjäsovelluksen omaa tietokantayhteyskerrosta. Sovellusylläpitopalvelussa käynnistettiin jatkuvan integraation kehittämishanke, jonka osana oli tarkoitus saada parannusta sovellusalustan julkaisuprosessiin.

Sovellusalustan julkaisu tarkoittaa pääpiirteissään tapahtumaketjua, jossa aluksi haetaan uusimmat kehittäjien kirjoittamat lähdekoodit versionhallinan kehityshaarasta, kootaan alustan sovellusosat, eli moduulit, ja paketoidaan ne Java Archive(.jar)-tiedostoiksi. Seuraavaksi alusta testataan sen päälle rakennetulla käyttöliittymätestisovelluksella. Mikäli kaikki testit läpäistään, voidaan versionhallinnan kehityshaaran tiedostoversiot toimittaa integraatiohaaraan. Tämän jälkeen käynnistetään moduulikohtaiset Jenkins julkaisuajot yksitellen tietyssä järjestyksessä. Julkaisuajot tekevät JAR-paketit integraatiohaaran tiedostoversioiden mukaan. Mikäli julkaisupaketin luominen onnistuu Jenkinsissä määritellyllä tavalla, julkaistaan se, eli sijoitetaan Nexus artefaktivarastoon (Artifact Repository). Kun kaikki moduulipaketit on julkaistu Nexukseen, katsotaan sovellusalusta julkaistuksi. Lopuksi ilmoitetaan asianosaisille tahoille uudesta alustaversion julkaisun olevan valmis ja käytettävissä Nexuksessa.

Sovellusalustajulkaisu tehdään noin kuukauden välein, aika ajoin erityistarpeesta useamminkin. Uuden julkaisuversion tekeminen alkuperäisellä julkaisuprosessilla oli merkittävältä osin manuaalista työtä. Osia julkaisuprosessista oli automatisoitu, mutta erityisesti työvälineestä toiseen siirryttäessä oli varaa parantamiseen. Jatkuvan

integraation periaatteiden mukaan kaikki työvaiheet, joissa ihmisen toimenpidettä tai valintaa ei tarvita, tulee toteuttaa sovellusautomaation avulla. Manuaaliset työvaiheet nostavat riskejä esimerkiksi näppäilyvirheen myötä. Lisäksi ”käyttöliittymäkitkan” vuoksi kone suoriutuu lähes poikkeuksetta mekaanisista työsuoritteista ihmistä nopeammin. Nämä tunnustaa myös kehittäjä, jonka työmotivaatioon vaikuttaa negatiivisesti tarpeettomat manuaaliset työvaiheet.

Manuaalisen työn ongelmat kertautuivat sovellusalustan julkaisuprosessissa erityisesti siksi, että sovellusalustakokonaisuus muodostui ylätason hallintamoduulien lisäksi noin 20 alimoduulista, jotka täytyi alustan arkkitehtuurin vuoksi koota, paketoida, testata ja julkaista erillisinä kokonaisuuksina. Jokaisen moduulin kohdalla jouduttiin vähintään uloskirjaamaan tämän Apache Maven pom.xml –ohjaustiedosto, päivittämään moduulin uusi versionumero tiedostoon käsin ja lopuksi sisäänkirjaamaan POM-tiedosto versionhallintaan. Alkuperäisellä prosessin mukaan tehty sovellusalustajulkaisu kesti parhaimmillaankin noin kolme tuntia. Mikäli kohdattiin prosessin normaalikulun estäviä poikkeuksia, kuten alustan lähdekoodin tai työvälineohjelmistojen virheitä, saattoi sovellusalustan julkaiseminen vaatia kokonaisen työpäivän. Alkuperäinen julkaisuprosessi on mallinnettu BPM-prosessikaaviona liitteessä 2.

Opinnäytetön tavoitteena oli siis vähentää sovellusalustan julkaisuprosessin virhealttiutta julkaisijan tekemän manuaalisen työn vähentämisen kautta ja nopeuttaa julkaisuprosessin keskimääräistä suoritusaikaa hyödyntämällä työvälineohjelmistojen, erityisesti Jenkinsin, automatisointitoiminnallisuuksia. Näin voitaisiin toimittaa kustannustehokkaampaa palvelua ja parantaa alustatiimin työmotivaatiota.

4.2 Tekninen ympäristö ja työvälineet

Sovellusalustan kehitystyö ja ja julkaiseminen tapahtuu ulkoisen asiakkaan omassa teknisessä ympäristössä, joka ei ole avoin yleiseen internetiin. Ympäristössä työskennellään virtuaalikoneilla. Omalle pilvityöasemalleen kirjaudutaan kaksivaiheisella tunnistautumisella. Normaalin webbiautentikoinnin lisäksi käytetään henkilökohtaista matkapuhelinta.

Versionhallintaratkaisuna ympäristössä käytetään IBM ClearCase työvälinettä. ClearCase käyttää runko ja haara terminologian sijasta projekteja (Projects), joissa on kiinni integraatiovuo (Integration Stream) ja yksi tai useampia kehittäjävoita (Development Stream). Kun vuosta luodaan omalle työasemalle paikallinen työskentelyversio, puhutaan

näkymästä (View). Vuot ovat kuitenkin erittäin lähellä haara-termiä käsittellisesti, joten ymmärrettävyyden vuoksi voista puhutaan jatkossa haaroina.

Koontiautomaatiotyökaluna lähdekoodin kääntämiseen ja paketointiin käytetään Apache Mavenia. Maven liitetään yhteen Java-pohjaisen JUnit sovelluskehykseen, joka on erikoistunut sovellustestaamiseen. Näin koonnin yhteydessä voidaan esimerkiksi ajaa automaattisesti kaikki lähdekoodia vasten kirjoitetut yksikkötestit. Komentorivi on tyypillinen tapa käyttää Mavenia. Mavenissa keskeisessä roolissa ovat sen käyttämät XML-muotoiset Project Object Model(.pom)-ohjaustiedostot, joihin on kuvattu osat mistä sovelluskokonaisuus koostuu ja miten se rakennetaan ts. kootaan. POM-tiedostoista selviää esimerkiksi moduulien sisäiset- ja toisiin moduuleihin kytköksissä olevat hierarkiarakenteet sekä mahdolliset riippuvuudet kolmannen osapuolen sovelluskirjastoihin.

Jenkins on ympäristössä käytetty jatkuvan integraation työväline. Yksi Jenkinsin rooleista on toimia koonnin ajastajana (Build Scheduler) esimerkiksi yöllisten testausajojen käynnistäjänä. Laajemmin Jenkinsin tarkoituksena voidaan mieltää oleminen toisten sovellusten, kuten koontityökalun ja versionhallintaohjelmiston yhteen liittävänä tekijänä.

Selenium on sovelluskehys webbikäyttöliittymien automaattiseen testaamiseen internet selaimilla. Seleniumilla voidaan esimerkiksi nauhoittaa tietty toimintosarja ja toistaa se. Tätä hyödynnetään esimerkiksi regressiotestauksessa, kun halutaan varmistua etteivät uudet lähdekoodilisäykset riko vanhoja, aiemmin toimiviksi testattuja, toiminnallisuuksia. Sovellusalustan julkaisun yhteydessä ajetaan käyttöliittymien regressiotestejä Seleniumilla JUnit testiluokiksi valmiiksi kirjotettujen testien perusteella.

Windowsin .bat-päätteellisiin Batch tiedostoihin voidaan tallentaa komentokehote-komentoja, jotka suoritetaan kun tiedostoa kutsutaan. Julkaisuprosessin yhteydessä näitä komentojonotiedostoja hyödynnetään esimerkiksi siten, että Jenkins kutsuu tiedostoa ja tähän tallennetut Maven komennot tai Perl-skriptit suoritetaan Windowsin komentokehotteessa.

Kaikki ympäristön työvälineet olivat asiakkaan ympäristössä valmiiksi asennettuna. Vaihtoehtoisia työvälineitä ei ollut tarpeellista lähteä vertailemaan tai selvittämään, etenkin kun jokaisen uuden ohjelmiston saamiseksi asiakkaan ympäristöön on läpikäytävä lisenssiselvitysprosessi. Prosessi kestää pahimmillaan viikkoja ja vaatii sekä Capgeminin että asiakkaan korkean tason edustajien allekirjoituksia.

4.3 Toteutus

Toteutusvaihe alkoi sovellusalustan julkaisuprosessin perinpohjaisella tutkimisella. Ympäristön työvälineiden ja niiden julkaisuprosessin yhteydessä käytettyjen toimintojen analysoinnissa jouduttiin mennä tarkalle, niin sanotulle imperatiiviselle tasolle. Työvälineitä käytetään usein deklaratiivisella tavalla, jolloin tiiviillä ja abstraktilla komennolla saattaa tapahtua suuri määrä toimintoja, eikä komennon antaja aina tiedä täysin, mitä eri toimintoja sovellus suorittaa päästäkseen kehittäjän komennon mukaiseen lopputulokseen. Komennon suorittamisen tapahtumaketju ja käsitteet sen taustalla voivat olla hyvin monimutkaisia. Työnvaiheen valmistumisen tai työn sujuvuuden kannalta tapahtumaketjun yksityiskohtainen ymmärtäminen ei ole välttämättä tarpeellista. Jotta julkaisuprosessia voitiin parantaa oli kuitenkin ymmärrettävä kokonaisuutta ja sen osia syvällisemmin, että mahdollisesti tehtävät muutokset olisivat harkittuja ja niiden vaikutukset selvillä.

Erityisiksi kipukohdiksi alkuperäisessä julkaisuprosessissa tunnistettiin jokaisen noin 20 alimoduulin kohdalla tekemään jouduttu manuaalinen operaatio. Julkaisijan täytyi uloskirjata versionhallinnasta moduulin POM-tiedosto, päivittää siitä löytyviä versionumeroita haluttuun julkaisuversioon ja sisäänkirjata integraatiohaaraan tämä päivitetty POM-tiedosto. Tämän jälkeen käynnistettiin käsin vielä käsin Jenkinsistä moduulikohtainen julkaisuajo. Tavoitteeksi asetettiin koko tämän vaiheen automatisoiminen siten, että tarvittavat työvaiheet suoritettaisiin ohjelmallisesti Jenkins-ajon sisällä.

Istuttiin suunnittelupöytään miettimään ratkaisuja haluttuun lopputulokseen pääsemikseksi. Koska alustan moduulit piti koota sisäisten riippuvuuksien vuoksi tietyssä järjestyksessä, tarvittiin jokin tapa hallita ajojärjestystä. Jenkins-ajojen laukaisin –ominaisuuden pääteltiin soveltuvan käyttötarpeeseen. Syntyi idea ylätason Jenkins-ajosta, joka suorittaisi järjestyksessä alimoduuliajot. Tätä Jenkins-ajoa alettin kutsua julkaisupääajoksi (Release Master Job). Julkaisupääajo konfiguroitiin ajamaan alimoduuliajot järjestyksessä, odottaen aina käynnistetyn moduuliajon valmistumista ennen seuraavan aloittamista. Mikäli jokin moduulikohtainen ajo epäonnistuisi, keskeytettäisiin julkaisupääajo myös.

Alimoduuliajoihin oli kovakoodattu versionhallinnan integraatiohaaran nimi. Tiedettiin, että tulevaisuudessa uusien suurien versiojulkaisujen (Major release) yhteydessä integraatiohaara ei pysyisi samana, vaan näitä varten luotaisiin oma aina omat versionhallintaprojektit. Ilman muutoksia alimoduuliajoihin, olisi myös jouduttu luomaan

jokaista uutta versionhallintaprojektia kohden kovakoodatut alimoduuliajot. Suuria versiojulkaisuja tulee noin kaksi vuodessa ja yli 20 uuden Jenkins ajon luomiseen menee monta tuntia, vaikka vanhoista ajoista saakin pohjan. Tutkittiin siis vaihtoehtoisia ratkaisuja. Oivallettiin, että Jenkinsin parametrienvälitys ominaisuutta voitiin hyödyntää siten, että lähdehaaran nimi voitaisiin muodostaa dynaamisesti. Luotiin sovellusalustan moduuleille uudet Jenkins julkaisuajot, joissa lähdehaara annettiin parametrina. Myös julkaisupääajoon lisättiin lähdehaaraparametri ja asetettiin tämä välittymään alimoduuliajoille, julkaisupääajon näitä käynnistäessä.

Julkaisuajojen hallinta oli saatu siis osapuilleen tavoitellun mukaiseksi. Hieman hankalammaksi osoittaitui moduulikohtaisten POM-tiedostojen käsittely automatisoidusti. Tiedettiin Jenkinsistä löytyvän ominaisuus ajaa mm. komentorivikomentoja. Varsinkin käyttöjärjestelmäriippumattomissa yritykäytössä olevissa työvälineissä on hyvin usein mahdollisuus tehdä komentorivillä samat toiminnot kuin graafisessa käyttöliittymässä. Tyypillisesti komentorivin ohjausmahdollisuudet ovat jopa käyttöliittymää laajemmat. POM-tiedoston ulos- ja sisäänkirjaus IBM Clear Caseen ei ollut edellä mainitun ja hyvän dokumentaation ansiosta yleettömän haasteellista.

POM-tiedostojen sisällön muokkaaminen ohjelmallisesti sen sijaan aiheutti jonkin verran päänvaivaa. Regular Experssion viritelmään ei haluttu sortua, eikä sopivaa XML-muokkausvälinettä löytynyt ympäristöstä. Lopulta ratkaisuksi muovautui yksinkertainen Perl-merkkijonovaihdosskripti, jossa kaikki merkkijonot X korvattiin merkkijonolla Y. Tätä varten julkaisupääajoon ja parametrisoituihin moduuliajoihin luotiin uusi parametri nimeltä PREVIOUS_VERSION kuvaamaan edellistä julkaisuversiota X. Uusi julkaisuversio oli jo ennestään julkaisuajoissa RELEASE_VERSION parametrina ja tätä käytettiinkin arvona Y korvaamaan vanhat versionumerot. Hyväksi onneksi sovellusalustan käyttämä julkaisuversio oli sen verran pitkä ja omalaatuinen, ettei ollut vaaraa esimerkiksi kolmannen osapuolten kirjastojen riippuvuusversioiden ylikirjoittumisesta, mikäli ne sattuisivat vastaamaan arvoa X.

Ympäristön Jenkins ohjelmistossa oli käytettävissä yksi noodi, eli vapaasti kuvailtuna yhdyspiste jonka kautta Jenkins-ajojen toimintoja, kuten käynnistämistä tai laukaisemista, pystyi suorittamaan. Tälle yhdelle noodille oli konfiguroitu yksi suorittajaprosessi (Executor). Tämä tarkoitti, että vain yhtä Jenkins-ajoa pystyttiin suorittamaan samanaikaisesti. Julkaisupääajo on itsessään Jenkins-ajo joten se vaatii yhden yhden suoritusprosessin omaan käyttöönsä. Pääajon käynnistelemät alimoduuliajot vaativat kukin vuorollaan yhden suorittajaprosessin. Pääajo on kokoajan taustalla päällä, jotta se voi pitää kirjaa alimoduulien suoritusjärjestyksestä ja tilasta. Jotta Jenkins-ajot saataisiin

toimimaan suunnitellulla tavalla, tarvittiin siis vähintään kaksi suorittajaprosessia. Jälleen oli onni matkassa ja Jenkinsin ylläpitotiimi lisäsi ilmeisesti muutamalla hiiren klikkauksella toisen suorittajaprosessin käytetylle Jenkins noodille. Samanaikainen Jenkins-ajojen suorittaminen kuitenkin luonnollisesti kuormittaa integraatiokoneen resursseja enemmän, joten suorituskykyä piti jonkin verran tarkkailla, ettei uuden suorittajaprosessin lisääminen aiheuttanut järjestelmään epävakautta.

4.4 Tuotos

Opinnäytetyön tuotoksena syntyi kaksi uutta Jenkins julkaisupääajoa. Kaksi siksi, että ensimmäisessä julkaistaan kaikki alustan moduulit ja toisessa hypätään muutamia moduuleita yli, joita ei tarvita jokaisessa alustajulkaisussa. Julkaisupääjoilla pystytään ajamaan automatisoidusti sovellusalustan moduulikohtaiset Jenkins julkaisuajot. Moduulikohtaisista ajoista luotiin myös uudet lähdehaara riippumattomat versiot, joissa versionhallintahaara annetaan parametrina, uuden ja vanhan julkaisuversion lisäksi. Käytettäessä julkaisupääajoa ilman että ajonaikaisia koontivirheitä ilmenee, pääajo välittää sille käynnistyksen yhteydessä syötetyt parametrit alimoduulien julkaisuajoille, joita se laukaisee yksi kerrallaan tietyssä järjestyksessä.

Abstraktina tuotoksena opinnäytetyöstä syntyi sovellusalustan julkaisuprosessin uusi tapahtumaketju. Liitteessä 3 on mallinnettu tämä kehitetty julkaisuprosessi, jossa jatkuvan integraation periaatteiden mukaisesti hyödynnetään sovellusautomaatiota tehokkaasti. Uusi prosessi on jo otettu käyttöön ja sen on koettu olevan parannus verrattuna alkuperäiseen julkaisuprosessiin.

Pohdinta

Tunnin parin saaminen pois julkaisuprosessin kokonaiskestosta on taloudellisesta näkökulmasta jokseenkin positiivista. Todellinen hyöty oli kuitenkin virhealttiuden pienentymisen kautta syntynyt riskin vähentyminen sovellusalustaa julkaistaessa ja julkaisuprosessin suoritusajan nopeutuminen. Näillä on merkitystä palvelun laadun ylläpitämisen ja toimitusvarmuuden turvaamisen kannalta. Tämä erityisesti siksi, että sovellusalustan varassa toimivat käyttöliittymäsovellukset usein tarvitsevat uuden alustaversion käyttöönsä mahdollisimman nopeasti. Käyttöliittymäsovelluksille tehtävät muutostyöt vaativat toisinaan ensiksi muutoksen sovellusalustassa. Alustamuutokset eivät puolestaan ole käytettävissä käyttöliittymäsovelluksille, ennen kuin alustasta on tehty uusi julkaisuversio.

Julkaisupääajolle ei pystytty toteuttamaan kaikkia suunniteltuja ja haluttuja ominaisuuksia. Tällaisia oli esimerkiksi pääajon keskeytyessä jonkin virheen vuoksi, jatkaminen tietystä kohdasta alimoduulien suoritusketjua. Sen sijaan virheen sattuessa käytännössä joutuu suorittamaan julkaisun loppuun alkuperäisen julkaisuprosessin mukaisesti käsin POM-tiedostoja muokaten ja moduulikohtaiset julkaisuajot käsin käynnistäen. Liite 3 vihjaa julkaisuajon uudelleenaloittamisen olevan mahdollista, ja onkin, mutta vanhan version X korvaaminen julkaisuversiolla Y ei toimi, ellei kaikkien moduulien POM-ohjaustiedostoissa ole yhteneväiset versiot. Onnistuneesti suoriutuneet moduulikohtaiset ajot ovat jo muokanneet omia POM-tiedostojaan virheeseen törmätessä. Mikäli pääajo keskeytyy alkuvaiheessa suoritusketjua voi olla nopeampaa muokata muutaman moduulin POM:it alkutilaan ja aloittaa koko julkaisupääajo uudestaan. Mikäli esimerkiksi yli puolet ajoista on onnistunut, on pienempi vaiva tehdä julkaisu loppuun vanhan prosessin mukaisesti.

Vaikka periaatteessa lisenssiasioiden merkittävyys esimerkiksi lukuisen uutisnäkyvyyttä saaneiden oikeusjuttujen kautta oli tiedossa, oli silti yllätys miten rajoittava tekijä ohjelmistolisenssi voi olla. Vaikka löytyisi kirjasto tai liitännäinen, jolla suunniteltu toiminnallisuus saataisiin toteutettua, niin lisenssinhankkimisprosessi voi olla niin raskas että päädytään viritelmiin tai yritetään kaikkea mahdollista muuta ennen kuin päädytään uuden lisenssin hankintaan. Mainittu julkaisupääajon suorittamisen jatkaminen tietystä kohtaa olisi esimerkiksi todennäköisesti onnistunut erikoistuneella liitännäisellä, mutta tätä vaihtoehtoa ei alettu edellä mainituista syistä edes syvällisemmin selvittää. Lisenssiongelmilta ei välttämättä vältytä vaikka käytettäisiin avointa lähdekoodia. Avoimuudesta huolimatta sovelluksen käytölle on usein tiettyjä edellytyksiä ja yrityksen lakiosasto voi haluta tutustua lisenssiehtoihin, ennen kuin uusi sovellus tai kirjasto voidaan

hyväksyä käyttöön. Täysin vapaasti yrityskäyttöön otettavat ohjelmistot ovat hyvin harvassa.

Toteutusvaiheessa tehty työvälineiden ominaisuuksien tutkiminen tapahtui pääosin dokumentaatioon tutustumalla tai puhtaalla kokeilulla. Työpaikkaverkoston luominen osoittautui kuitenkin olevan päänvaivaa aiheuttavissa tilanteissa kullan arvoista. Seniorimmalta kehittäjältä saattaa löytyä vastaus kuin apteekin hyllyltä ongelmaan, johon omalla selvittelyllä menisi puoli päivää. Kantapään kautta oppiminen ei välttämättä IT-alalla ole se paras tapa, ainakaan konsulttiyrityksen taloudellisen tehokkuuden näkökulmasta.

Kuten toteutusvaiheen kerronnasta saattaa huomata, toteutuksessa kohdattiin muutama onnenkantamoinen. Muutamassa kohdassa oltaisiin voitu törmätä vakaviin toteutusesteisiin, ellei ratkaisu, joka sattumalta tässä työssä toimikin, olisikaan käynyt tiettyjen olosuhteiden ollessa toiset. Mahdollisimman pitkälle suunnitteleminen ja parhaiden käytäntöjen tutkimien etukäteen on yleensä järkevin vaihtoehto. Osaamisen puutteen tai aikarajojen tiukkuuden vuoksi ei kuitenkaan aina ole mahdollista suunnitella riittävässä laajuudessa. Riittävä laajuus myös ymmärretään usein vasta retrospektiivissä.

Nykyaikaisessa ohjelmistokehityksessä, ehkäpä aivan pienimpiä projekteja lukuun ottamatta, ei ole järkevää tehdä manuaalisesti laadunvarmistukseen liittyvää laajaa ohjelmistotestaamista, vaan on järkevää pystyttää automaattiset koonti- ja testausjärjestelmät käyttäen hyväksi jotakin jatkuvan integraation työvälinettä, kuten Jenkinsiä. Isommissa projekteissa laadunvarmistamiseen vaaditut lisäresurssit ilman sovellusautomatiikan hyödyntämistä nostavat kustannukset nopeasti suuremmiksi, kuin mitä jatkuvan integraation prosessin mukaisen ympäristön pystytys työvälineineen vaatii. Pienemmissä projekteissa saattaa riittää pelkkä koonnin ajastaminen, mikäli monimutkaisille suorituskokonaisuuksille ja usean eri työkalusovelluksen hallitulle yhteenliittämiselle ei ole tarvetta. Koonnin voi ajastaa myös melko yksinkertaisilla menetelmillä, Unix-järjestelmissä esimerkiksi cron skripteillä.

Jenkins on hyvin modulaarinen. Vapaan lähdekoodin yhteisön tuki on vahva ja erilaisia liitännäisiä on saatavilla runsaasti. Käyttötarve ja sen mukaiset liitännäiset kannattaa kuitenkin suunnitella mahdollisimman pitkälle. Esimerkiksi monet tässä opinnäytetyössä kuvatut Jenkins-toiminnallisuudet eivät välttämättä ole valmiina Jenkinsin kotisivuilta ladattavassa oletusasennusversiossa. Lisäliitännäisiä tarvitaan mitä erikoistuneisimpiin, esimerkiksi ohjelmistotuotekohtaisiin toimintoihin mennään.

Opinnäytetyön tekeminen oli vaativaa, varsinkin kun olin kokoaikaisessa työssä sitä tehdessäni. Osallistuin kuitenkin intensiivitoteutukselle, jossa yhden viikon ajan työstettiin opinnäytetyötä täysipäiväisesti. Tämä auttoi pääsemään hyvään vauhtiin kirjoittamisessa. Lähdeaineistoon tutustuminen oli jopa varsin mielekästä koska aihe kiinnosti. Mikäli aikaa olisi riittänyt, joitain lähdeteoksia olisi voinut hyvinkin lukea kannesta kanteen. Ajoittain myös tuttuna pitämiini käsitteisiin liittyviä seikkoja tai nyansseja piti tarkistaa, ennen kuin niitä saattoi purkaa tekstiksi. Tämä vahvisti aihealueiden tuntemusta sekä tuotti välillä uusia näkökulmia niihin.

Jatkotutkimusehdotuksena heräsi kysymys: Aiheuttaako hajautetun versionhallinnan käyttö erityisvaatimuksia jatkuvan integraation periaatteiden noudattamiselle?

Lähteet

Betteley, J. 2011. 8 Principles of Continuous Delivery. Luettavissa: http://devopsnet.com/2011/08/04/continuous-delivery/. Luettu: 17.11.2014

CollabNet Blog. 2011. Has Git Killed Subversion & CVS?. Luettavissa: http://blogs.collab.net/subversion/a-look-at-git-cvs-svn. Luettu: 23.10.2014.

Collins-Sussman, B. & Fitzpatrick, B. & Pilato, C. 2011. Version Control with Subversion: For Subversion 1.7. Luettavissa: http://svnbook.red-bean.com/en/1.7/svn-book.pdf. Luettu: 22.10.2014.

Duval, P. & Matyas, S. & Glover, A. 2007. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley. New Jersey.

Farcic, V. 2014. Continuous Delivery: Introduction to concepts and tools. Luettavissa: http://technologyconversations.com/2014/04/29/continuous-delivery-introduction-to-concepts-and-tools/. Luettu: 17.11.2014

Fowler, M. 2006. Continuous Integration. Luettavissa: http://www.martinfowler.com/articles/continuousIntegration.html. Luettu: 20.10.2014.

Git. About Git. Luettavissa: http://www.git-scm.com/about. Luettu: 23.10.2014.

Jenkins. 2014. Building a software project. Luettavissa: https://wiki.jenkins-ci.org/display/JENKINS/Building+a+software+project. Luettu: 17.11.2014

Kawalerowicz, M. & Berntson, C. 2011. Continuous Integration in .NET. Manning Publications. New York.

Kontio, N. 2014. Radioverkko-ohjaimen vertailutestauksen testiautomaatiomittauksien toteutus. Satakunnan ammattikorkeakoulu. Satakunta.

Korhonen, M. 2013. Jatkuvan integraation käyttöönotto ohjelmistokehityksessä. Oulun seudun ammattikorkeakoulu. Oulu.

Laitinen, M. 2011. Jatkuvan integration käyttöönotto. Metropolia Ammattikorkeakoulu.

MSDN. 2013. Continuous integration using TFS on the Cloud. Luettavissa: http://blogs.msdn.com/b/africaapps/archive/2013/03/05/continuous-integration-using-tfs-on-the-cloud.aspx. Luettu: 24.10.2014.

Mustaniemi, J. 2013. Jatkuva integrointi Java Enterprise -kehityksen apuna. Tampereen ammattikorkeakoulu. Tampere.

Proffitt, B. 2011. Oracle responds to Hudson/Jenkins split. Luettavissa: http://www.itworld.com/article/2747577/open-source-tools/oracle-responds-to-hudson-jenkins-split.html. Luettu: 24.10.2014.

Purencool. What is a Version Control System?. Luettavissa: http://purencool.com/what-is-a-version-control-system. Luettu: 22.10.2014.

Riekki, J. 2013. Systeemikomponentin jatkuva integrointi. Oulun seudun ammattikorkeakoulu. Oulu.

Rouse, M. Source code. Luettavissa: http://searchsoa.techtarget.com/definition/source-code. Luettu: 23.10.2014

Sink, E. 2011. Version Control by Example. Luettavissa: http://ericsink.com/vcbe/vcbe_usletter_lo.pdf. Luettu: 23.10.2014.

Smart, J. 2011. Jenkins: The Definitive Guide. O’Reilly Media. California.

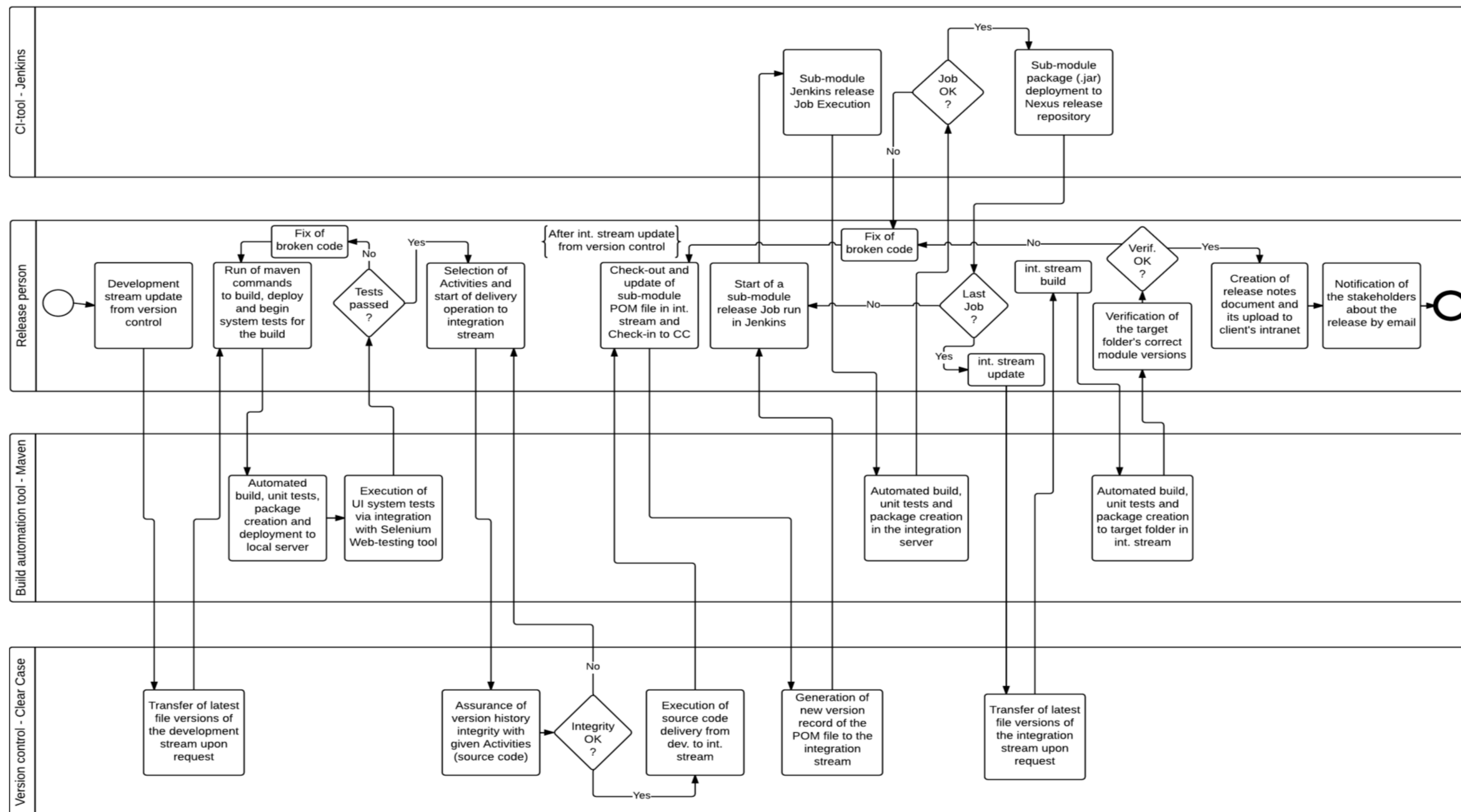
Zeroturnaround, 2013. DevProd Report Revisited: Version Control Systems in 2013. Luettavissa: http://zeroturnaround.com/rebellabs/devprod-report-revisited-version-control-systems-in-2013/. Luettu: 23.10.2014.

Liitteet

Liite 1. Keskitetyn versionhallinnan 18 perustoiminnallisuutta

Create – Uuden keskusarkiston luominen
Checkout – Tiedoston ulkoskirjaaminen
Commit – Luo keskusarkistoon uuden version nykyisestä työskentelyversiosta
Update – Työskentelyversion päivittäminen vastaamaan päähaaran uusinta versiota
Add – Uuden tiedoston lisääminen versionhallintaan
Edit – Tiedoston muokkaaminen (tapahtuu useimmiten automaattisesti)
Delete – Tiedoston poistaminen versionhallinnasta
Rename – Tiedoston uudelleennimeäminen
Move – Tiedoston siirtäminen eri sijaintiin samassa arkistossa
Status – Listaa tiedostot, joihin työskentelyversiossa on tällä hetkellä tehty muutoksia
Diff – Antaa tarkat muutostiedot muuttuneista tiedostoista
Revert – Poistaa työskentelyversion muutokset ja palauttaa niitä edeltäneen tilan
Log – Pyytää keskusarkistolta historiatietoja
Tag – Liittää tiettyyn versioon käyttäjän valitseman lisätunnistetiedon
Branch – Luo uuden haarauman päähaaraan
Merge – Integroi haarojen tai tai työskentelyversioiden tietoja, saattaa aiheuttaa konfliktin
Resolve – Mergen yhteydessä käytetty toiminto, jolla ilmoitetaan konfliktin olevan ratkaistu
Lock – Lukitsee tietyn tiedoston, jolloin siitä ei voi luoda uusia versioita poistamatta lukkoa
(Sink 2011, 5-15)

Liite 2. Alkuperäinen julkaisuprosessi



Liite 3. Kehitetty julkaisuprosessi